

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

№ 644

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ  
ИНСТИТУТ СТАЛИ и СПЛАВОВ

Технологический университет

МИСиС



Кафедра инженерной кибернетики

Б.В. Черкасский

# Комбинаторные алгоритмы

Курс лекций

Рекомендовано редакционно-издательским  
советом института

Москва Издательство «УЧЕБА» 2006

УДК 519.712

Ч-48

Р е ц е н з е н т

профессор *E.A. Калашников*;

доктор физико-математических наук, профессор кафедры исследования  
операций С.-Петербургского государственного университета *И.В. Романовский*

**Черкасский Б.В.**

Ч-48 Комбинаторные алгоритмы: Курс лекций. – М.: МИСиС,  
2006. – 159 с.

Курс лекций состоит из десяти разделов, охватывающих материал полу-  
годового курса «Комбинаторные алгоритмы».

В разделах приведены основные определения, касающиеся алгоритмов,  
их классификация, описание и способы программной реализации.

Издание снабжено обширным иллюстративным материалом, а также  
программами, поясняющими работу алгоритмов.

Курс лекций предназначен для студентов третьего курса, обучающихся  
по специальности 230401 (0730) «Прикладная математика», а также может  
быть рекомендован всем, кто интересуется данной темой.

## ОГЛАВЛЕНИЕ

От автора .....	4
1. Алгоритмы: классификация, сложность.....	5
2. Представление сетей в компьютере.....	16
3. Алгоритм нахождения компонент связности графа.....	32
4. Задача построения кратчайшего связывающего дерева .....	42
5. Алгоритм построения стабильного бракосочетания .....	54
6. Задача построения кратчайших путей .....	63
7. Задача построения кратчайших путей (алгоритм Беллмана – Форда).....	78
8. Задача построения кратчайших путей (алгоритм Дейкстры) .....	94
9. Кучи .....	109
10. Деревья поиска.....	133
Библиографический список .....	158

## **От автора**

В настоящем пособии приведены лекции полугодового курса «Комбинаторные алгоритмы», который читается студентам третьего курса кафедры «Инженерная кибернетика». Создание пособия оказалось возможным благодаря тому, что студентки О. Щеголева В. Грошева предоставили автору свои конспекты в виде файлов, которые и послужили толчком для написания этого курса лекций.

Письменный текст отличается от устной речи. Поэтому в ходе записи некоторые лекции довольно сильно изменились и не вполне соответствуют тому материалу, который обычно излагается на занятиях. Но принципиальных отличий нет, и этим пособием можно пользоваться при подготовке к экзамену.

Выражаю благодарность О. Щеголовой и В. Грошевой, без которых это издание не вышло бы в свет.

# 1. АЛГОРИТМЫ: КЛАССИФИКАЦИЯ, СЛОЖНОСТЬ

Под *алгоритмом* обычно понимается последовательность действий, которые необходимо выполнить для решения задачи. В рамках настоящего курса нас вполне устроит это интуитивное понимание алгоритма. Точное определение комбинаторных алгоритмов мы тоже давать не будем. Считается, что комбинаторный алгоритм имеет дело с дискретными объектами, такими как строки, слова, матрицы, сети, в отличие от непрерывных объектов типа кривых, поверхностей, функций. Однако во многих случаях различия условны. Так, если сама функция – это непрерывный объект, то ее запись в виде строки, состоящей из конечного числа символов, уже можно рассматривать как объект дискретный. Тем не менее все эти туманности не помешают нам изучать конкретные комбинаторные алгоритмы.

Разработать алгоритм – означает придумать последовательность шагов, которые приведут нас к решению задачи.

## 1.1. Классификация алгоритмов по типу решения

Сначала приведем примерную классификацию алгоритмов с точки зрения характера получаемого решения.

### 1.1.1. Точные алгоритмы

Такие алгоритмы дают точное решение поставленной задачи. Например, задан массив из  $n$  целых чисел. Среди них надо найти минимальное. Точный алгоритм решения этой задачи знает каждый.

Другим примером задачи, точный алгоритм решения которой известен, является *задача о рюкзаке*.

На складе находятся  $n$  типов предметов, причем количество предметов одного типа не ограничено. Для каждого типа предметов  $i$  известен вес  $a_i$  и цена  $c_i$ ,  $i = 1 \dots n$ .

Задана вместимость рюкзака:  $A$  – максимальный вес, который он может выдержать.

Обозначим через  $x_i$  количество предметов  $i$ -го типа, находящихся в рюкзаке (естественно,  $x_i \geq 0$  – целое число). Тогда должно выполняться следующее ограничение:

$$\sum_{i=1}^n x_i a_i \leq A$$

Задача состоит в максимизации суммарной стоимости предметов, помещенных в рюкзак:

$$\sum_{i=1}^n x_i c_i \rightarrow \max$$

### **1.1.2. Приближенные алгоритмы**

Приближенные алгоритмы не дают оптимальное решение задачи, но качество этого решения можно оценить с помощью некоторого критерия.

Например, если в задаче о рюкзаке обозначить через  $C = \sum_{i=1}^n x_i c_i$  функционал некоторого допустимого решения, через  $C_{\max}$  – функционал оптимального решения, а через  $\varepsilon$  – заданную погрешность, то  $\varepsilon$ -приближенным решением является такое, что  $C \geq C_{\max}(1 - \varepsilon)$ . Алгоритм, который для заданного  $\varepsilon$  строит  $\varepsilon$ -приближенное решение, называется *приближенным алгоритмом*.

Бывают алгоритмы, которые могут находить  $\varepsilon$ -приближенное решение только для некоторых фиксированных  $\varepsilon$  или для  $\varepsilon$ , находящихся в заданном интервале. Такие алгоритмы тоже можно называть приближенными.

Иногда встречаются алгоритмы, которые не ищут приближенное решение, а строят точное решение задачи, «близкой к исходной». Так, для задачи о рюкзаке такой алгоритм мог бы построить решение, для которого  $\sum_{i=1}^n x_i a_i \leq A' = A(1 + \varepsilon)$ , при этом среди всех решений с вместимостью рюкзака  $A'$  суммарная ценность взятых предметов максимальна.

### **1.1.3. Эвристические алгоритмы**

Эвристические алгоритмы – алгоритмы, дающие решение, точность которого оценить невозможно. Такие алгоритмы могут оказаться полезными при решении сложных задач, для которых точный или приближенный алгоритм придумать не удается.

Критерием качества эвристического алгоритма является то, что на практике он дает удовлетворительное решение, которое «устраивает заказчика». Другим способом оценки качества эвристического алгоритма может служить его сравнение с другими алгоритмами, предназначенными для решения той же задачи. Наш алгоритм может давать лучшее решение и/или быстрее работать.

Примером эвристического алгоритма является алгоритм «иди в ближайший» в задаче коммивояжера.

Задача коммивояжера формулируется так: заданы  $n$  городов и матрица расстояний между ними. Надо выбрать такую последовательность обезода городов, чтобы побывать в каждом из них ровно один раз, вернуться в исходный пункт и чтобы при этом суммарная длина маршрута была бы минимальной.

В алгоритме «иди в ближайший» в качестве очередного пункта из городов, еще не включенных в маршрут, выбирается город, расположенный ближе всего к тому городу, где мы находимся в настоящий момент.

Часто алгоритм «иди в ближайший» позволяет получить неплохое решение задачи коммивояжера, однако нетрудно придумать пример, когда он не только не дает точное решение, но маршрут, построенный с его помощью, оказывается значительно хуже оптимального.

Вообще говоря, эвристическими алгоритмами (за редким исключением) стоит пользоваться только тогда, когда ни точного, ни приближенного алгоритма мы предложить не можем или по каким-то причинам они нас не устраивают. Но даже в этом случае к эвристическим алгоритмам надо относиться с осторожностью: даже если 1000 задач с его помощью решены успешно, нет никаких гарантий, что 1001-е решение также будет удовлетворительным. С особой осторожностью следует относиться к утверждениям авторов о том, что их эвристические алгоритмы «всегда отлично работают» – такое бывает, но очень редко.

#### **1.1.4. Вероятностные алгоритмы**

*Вероятностными* можно назвать алгоритмы, относительно которых удается доказать, что какое-то свойство получаемого с их помощью решения выполняется с «некоторой вероятностью». Такие алгоритмы почти не встречаются. Для того чтобы говорить о вероятности выполнения чего-то, необходимо на множестве задач вво-

дить меру. Затем надо доказывать, например, что мера подмножества задач, на которых алгоритм дает точное решение, больше 0,99.

Так, для эвристического алгоритма решения задачи коммивояжера «иди в ближайший» было доказано, что при  $n \rightarrow \infty$  вероятность того, что для задачи с  $n$  городами с помощью алгоритма не удается получить точное решение, стремится к нулю. Сам по себе результат весьма любопытен, однако мало что дает для решения практических задач. Ведь из того, что доказано, что мера множества «плохих задач» мала или даже стремится к нулю, вовсе не следует, что взятая нами конкретная задача не будет решена плохо. Возможно, именно поэтому вероятностные оценки качества алгоритмов так и не получили широкого распространения.

## 1.2. Классификация алгоритмов по способу выполнения операций

Классифицировать алгоритмы можно не только по характеру получаемого с их помощью решения, но и по другим параметрам. Так, алгоритмы можно разбить на два больших класса: последовательные и параллельные.

### 1.2.1. Последовательные алгоритмы

Как правило, большая часть изучаемых алгоритмов является последовательными, то есть операции выполняются последовательно, одна за другой. Все алгоритмы, которые мы будем рассматривать в данном курсе лекций, относятся именно к этому классу. Такие алгоритмы хорошо реализуются на однопроцессорных машинах, которые раньше составляли абсолютное большинство. Исключением были многопроцессорные системы, специально предназначавшиеся для решения важных конкретных задач.

### 1.2.2. Параллельные алгоритмы

В последнее время параллельные вычисления и параллельные алгоритмы становятся все более популярными из-за широкого распространения многопроцессорных систем. В параллельных алгоритмах несколько операций выполняются параллельно (одновременно), каждая на своем процессоре.

Так, например, с помощью  $n^2$  процессоров можно быстро перемножить две матрицы размером  $n \times n$  каждая. При этом перемноже-

ние каждой пары «строка – столбец» осуществляется с помощью «своего» процессора. Разные процессоры при работе не будут зависеть друг от друга, так что ни один из них не будет ожидать результатов работы другого.

Вообще, часто говорят, что алгоритм хорошо «распараллеливается», если от увеличения количества процессоров время решения задачи существенно уменьшается. Алгоритм перемножения матриц распараллеливается замечательно. Однако не все последовательные алгоритмы таковы. Часто приходится придумывать алгоритмы, специально предназначенные для работы на многопроцессорных системах. Параллельные вычисления – это особая наука, которая, в частности, включает в себя изучение различных классов параллельных алгоритмов и многое другое, чего мы в этом курсе лекций касаться не будем.

### **1.3. Классификация алгоритмов по детерминированности операций**

#### **1.3.1. Детерминированные алгоритмы**

Еще одним классифицирующим признаком является предопределенность действий алгоритма. Как правило, рассматриваемые алгоритмы являются *детерминированными*, но встречаются и *рандомизированные* алгоритмы.

*Детерминированными* называются алгоритмы, в которых каждый шаг однозначно определяется текущим состоянием данных, то есть исходными данными и последовательностью предыдущих шагов. Абсолютное большинство классических алгоритмов является детерминированными, что вполне соответствует природе стандартных компьютерных вычислений и наших взглядов на жизнь.

#### **1.3.2. Рандомизированные алгоритмы**

*Рандомизированными* называются алгоритмы, в которых очередной шаг выбирается из некоторого набора шагов в соответствии с имеющимся распределением вероятностей, то есть исходные данные и последовательность ранее сделанных шагов не определяют следующий шаг однозначно, а только задают набор возможных действий и вероятностей их реализации.

Аналогом рандомизированного алгоритма является поведение человека, блуждающего по городу, который, подходя к очередному перекрестку, выбирает улицу, по которой идти дальше, подбрасывая монетку.

В ряде случаев применение рандомизированных алгоритмов дает лучшие результаты по сравнению с детерминированными. Но рассмотрение алгоритмов такого типа выходит за рамки настоящего курса лекций.

## 1.4. Сложность алгоритмов

Одними из самых важных показателей качества алгоритма являются время его работы (скорость) и объем используемой памяти. *Сложность* – это теоретическая оценка качества алгоритмов с точки зрения их скорости.

Каждая задача может быть по определенным правилам закодирована с помощью входного слова – последовательности символов некоторого алфавита. Обозначим через  $R$  длину входного слова – количество символов, из которых оно состоит. Число  $R$  будем называть *размером задачи*.

### 1.4.1. Оценка сложности сверху

Обозначим через  $F(R)$  функцию, дающую оценку сверху максимального числа операций, которые нужны алгоритму для решения задачи размером  $R$ ; то есть для решения любой задачи размером  $R$  алгоритм использует не более чем  $F(R)$  операций. Такая *оценка сложности* называется *оценкой сверху*. Как правило, сложность алгоритмов оценивают с точностью до константы. Причина этого состоит в том, что точное количество операций алгоритма зависит от особенностей конкретного компьютера и используемого алгоритмического языка и считать их трудно, да и не имеет особого смысла. К тому же оценка сложности с точностью до константы верна для всех принципиально не отличающихся друг от друга компьютеров и, как правило, дает адекватное представление о качестве алгоритма. Будем полагать, что алгоритм имеет сложность порядка  $O(F(R))$ , если на решение любой задачи размером  $R$  им будет потрачено не более чем  $C_1 F(R) + C_2$  операций, где  $C_1$  и  $C_2$  – константы, не зависящие от размеров задачи.

Иногда вместо термина «сложность» мы будем употреблять как синоним слово «трудоемкость».

Полезно ли умение находить оценку сложности алгоритма и имеют ли оценки сложности сверху какое-то отношение к практике? В основном да, хотя к таким оценкам, как к любой абстракции, надо подходить осторожно. Если оценка сверху хороша, то можно быть

уверенным, что алгоритм будет работать быстро. Плохая оценка сложности наводит на грустные размышления: наверное, алгоритм будет работать плохо, хотя бывают исключения. Например, симплекс-метод решения задачи линейного программирования.

Такое неполное соответствие оценки сверху и реальной сложности задачи легко объясняется. Если из тысячи задач одна решается 100 секунд, а остальные – по одной секунде, то оценка сверху составит 100, при этом, как правило, попадаться будут односекундные задачи. Обычно решаемая задача далека от худшего случая, и, следовательно, время ее решения существенно меньше, чем его оценка. Тем не менее, несмотря на грубость, оценки сверху часто дают нам неплохое представление о качестве алгоритма. Если сравнивать два алгоритма, то, как правило, если оценка сверху одного из них лучше, то и алгоритм лучше. Так что оценками сверху имеет смысл пользоваться. Во всяком случае, существенно более совершенных инструментов анализа качества алгоритмов, к сожалению, пока не придумано.

#### **1.4.2. Простые свойства оценки сложности**

Нам пригодятся несколько простых правил, позволяющих оперировать с оценкой сложности.

1. Если  $k$  – константа, то  $O(kF(R)) = O(F(R))$ , то есть константа не влияет на оценку сложности.
2. Если  $F(R) > H(R)$ , а  $k_1, k_2$  – константы, то  $O(k_1F(R) + k_2H(R)) = O(F(R))$ , то есть меньшая функция поглощается большей<sup>1</sup>.
3. Для любого  $a > 1$   $O(\log_a R) = O(\log_2 R)$ .

Благодаря последнему правилу можно в оценках сложности не указывать основание логарифма – с точностью до константы все логарифмы одинаковы.

#### **1.4.3. Оценка сложности в среднем**

Интуитивно ясно, что если мы говорим, что какая-то оценка сложности является оценкой в среднем, то имеется в виду, что если

---

<sup>1</sup> Похожее правило о поглощении меньшего наказания большим применяется и в Уголовном кодексе РФ.

просуммировать время решения «всех» задач и вычислить среднее значение, то оно будет соответствовать приведенной оценке.

Более точно это понятие можно определить следующим образом. Вначале делаются предположения о распределении вероятностей на множестве решаемых задач. После чего, на основании этих предположений, вычисляется оценка математического ожидания времени решения на соответствующем множестве задач. Такая оценка времени решения и называется *оценкой в среднем*.

Что лучше: оценка сверху или оценка в среднем? Однозначный ответ дать непросто. Оценка сверху дает гарантию, что независимо от задачи время ее решения не превысит указанное. Такое свойство оценки бывает очень важно для алгоритмов, используемых в системах реального времени. Бессмысленно говорить о том, что в среднем наша программа работает быстро, если из-за того, что попалась очень «плохая» задача, которую компьютер не успел решить, крылатая ракета врезалась в холм. В этом случае правильным критерием качества будет оценка сверху. А вот при массовом решении однотипных задач, когда важно суммарное время и не столь важно, если решение какой-либо задачи будет длиться долго, лучше использовать оценку в среднем.

Как правило, при анализе алгоритмов используются оценки сверху. Одна из причин состоит в том, что получать такие оценки гораздо проще, чем оценки в среднем. Мы будем следовать этой традиции.

Примером оценки сверху может служить оценка сложности алгоритма сортировки массива *методом пузырька*:  $O(n^2)$ . Здесь мы найдем полную «гармонию»: и оценка плохая, и алгоритм работает плохо. Удивительно, но алгоритм быстрой сортировки (*Quicksort*), признаваемый на практике самым быстрым алгоритмом сортировки, имеет верхнюю оценку сложности такую же, как и сортировка пузырьком –  $O(n^2)$ , хотя работает намного быстрее. Правда, оценка в среднем алгоритма быстрой сортировки составляет  $O(n \log n)$ . Для алгоритма быстрой сортировки, как видите, оценка сверху далека от реального времени работы. Алгоритм пирамидальной сортировки или, иначе, *сортировки деревом*, который мы будем разбирать далее, обладает оценкой сверху  $O(n \log n)$ , то есть высокая скорость работы алгоритма гарантирована, хотя на практике он работает несколько медленнее хорошо настроенного алгоритма быстрой сортировки.

#### **1.4.4. Оценка сложности снизу**

Применимые на практике *оценки снизу* удается получать очень редко. Оценка снизу для задачи некоторого размера означает, что для любого алгоритма определенного класса найдется задача этого размера, которая не может быть решена за меньшее число операций. Так, например, доказано, что задача сортировки массива из  $n$  элементов в общем случае не может быть решена быстрее, чем за  $O(n \log n)$  операций при довольно естественном предположении, что основной операцией алгоритма является попарное сравнение двух элементов массива.

К сожалению, такие «достижения» встречаются редко. В большинстве случаев оценки снизу удается получать при столь серьезных ограничениях на алгоритмы, что результаты теряют практическое значение.

#### **1.4.5. Полином и экспонента**

С точки зрения оценки сложности различают два класса алгоритмов – полиномиальные и экспоненциальные.

Алгоритм называется *полиномиальным*, если функция  $F(R)$ , дающая оценку сверху, ограничена сверху полиномом от размеров задачи. Примеры полиномиальных оценок:  $O(n)$ ,  $O(n^2 \log n)$ ,  $O\left(\sum_i \log a_i\right)$ .

Если  $F(R)$  с увеличением размеров задачи растет быстрее любого полинома, то такие алгоритмы называются *экспоненциальными*, например:  $O(n!)$ ,  $O(2^{\sqrt{m}})$ .

Полиномиальные алгоритмы принято считать хорошими, а экспоненциальные – плохими. Конечно, все это условно, и вряд ли можно считать хорошим алгоритм со сложностью  $O(n^{1000})$ . Тем не менее экспоненциальные и полиномиальные алгоритмы различаются принципиально, и чтобы в этом убедиться рассмотрим табл. 1.1, в которой по горизонтали располагается размер задачи, по вертикали – сложность алгоритма, а в клетках таблицы – время решения. Если единица времени не указана, то время дано в секундах. Считаем, что наш компьютер выполняет 1 миллион операций в секунду.

Таблица 1.1

**Время работы алгоритмов**

	<b>10</b>	<b>30</b>	<b>50</b>	<b>100</b>	<b>500</b>	<b>1000</b>
$1000n$	0,01	0,03	0,05	0,1	0,5	1
$1000n \log n$	0,03	0,15	0,3	0,7	4,5	10
$100n^2$	0,01	0,09	0,25	1	25	2 мин
$10n^3$	0,01	0,3	12	10	20 мин	3 ч
$0,01 \cdot 2^n$	$10^{-5}$	10	4 мес.	$4 \cdot 10^{14}$ лет	—	—
$0,00001 \cdot 3^n$	$5 \cdot 10^{-6}$	30 мин	$5 \cdot 10^6$ лет	—	—	—

Из таблицы видно, что время работы экспоненциальных алгоритмов растет очень быстро с увеличением размеров задачи. Для алгоритма с оценкой сложности  $O(2^n)$  увеличение мощности машины в 1000 раз позволяет за то же время решить задачу, размер которой вырос всего на 10.

Из этого следует вывод, что время работы зависит не столько от мощности процессора, сколько от эффективности алгоритма. Так, например, задача, имеющая размер 30, будет решена лучшим алгоритмом за 0,03 секунды, а худшим – за 30 минут. Так что весьма вероятно, что хороший алгоритм на «медленной» машине будет работать лучше плохого алгоритма на «быстрой». Целесообразнее хорошо подумать об оптимизации алгоритма, чем, быстро написав плохую программу, потом часами ждать завершения его работы. А лучше всего работает хороший алгоритм на хорошей машине.

## 1.5. Implementation – техника реализации

Работа любого алгоритма зависит от техники его реализации (*implementation*). Что это такое – в точности определить невозможно. В это понятие входят и соответствующие требованиям алгоритма структуры данных, и эффективные алгоритмы обработки информации в этих структурах, и небольшие хитрости и технические приемы, позволяющие избегать усложненных программных конструкций, и пр. На сегодняшний момент в этой области наработано много красивых идей, методов и приемов. Конечно, можно каждый раз при реализации алгоритма придумывать все с начала (если получится), но лучше овладеть этой техникой. И тогда, не тратя время на изобретение велосипеда, вы сумеете быстро собрать необходимую вам эффективную структуру из стандартных блоков.

В рамках этого курса лекций будет уделено много внимания не только теории, но и технике реализации алгоритмов, что является не вполне традиционным для читаемых в России теоретических курсов по компьютерной математике. Тем не менее автор считает, что именно такой подход является наилучшим при изучении комбинаторных алгоритмов.

## **1.6. На каком языке следует писать программы**

В данном пособии приведены программы. Они написаны на языке, похожем на язык **C**, но при этом мы будем опускать «технические детали» типа описания переменных и операторов ввода–вывода.

При написании наших «программ» мы не будем использовать структуры и указатели, а ограничимся старыми добрыми массивами и индексами. Мне кажется, что программы, написанные в примитивном стиле, читаются проще. Разумеется, это не значит, что такими должны быть настоящие программы. И структуры, и указатели делают программу хуже читаемой, но более эффективной.

## 2. ПРЕДСТАВЛЕНИЕ СЕТЕЙ В КОМПЬЮТЕРЕ

### 2.1. Основные понятия

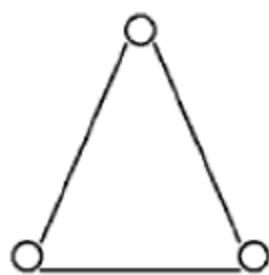
*Графом* называется множество вершин и множество дуг, соединяющих эти вершины.

*Сетью*, как правило, называют граф, на дугах и/или вершинах которого определены какие-либо функции: расстояние, время, стоимость и т.п. Сеть – очень удобный механизм моделирования различных объектов.

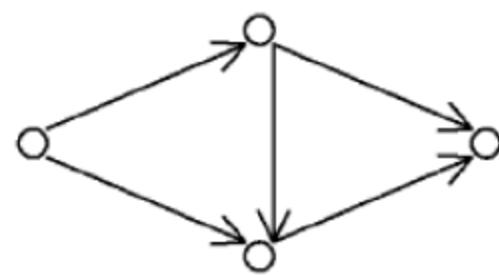
Дуга графа соединяет две его вершины. Граф называется *ориентированным*, если пара вершин, соединяемых дугой, является упорядоченной, то есть дуги графа имеют направление. Если же пара вершин, соединяемых дугой, является неупорядоченной, то безразлично, в каком направлении двигаться по дуге, и такой граф называется *неориентированным*.

Вершины графа иногда именуют *узлами* (*node*, *vertex*). Неориентированные дуги обычно называют *ребрами* и часто обозначают через *e* (*edge* – ребро), а ориентированные – *дугами* и обозначают через *a* (*arc* – дуга).

Обычно вершины графа изображают точками или кружками, а дуги – линиями, их соединяющими. Ориентированные дуги, как правило, изображают стрелками. На рис. 2.1 приведены два графа: один неориентированный, а другой ориентированный.



Неориентированный  
граф



Ориентированный  
граф

Рис. 2.1

Чтобы программировать сетевые алгоритмы, надо уметь хранить сети в компьютере, и от того, насколько представление сети будет удобным, зависит эффективность реализации алгоритма. Начнем с плохих способов представления сети, которым любят учить студентов.

## 2.2. Матрица смежности

Задан ориентированный граф с  $n$  вершинами и  $m$  дугами. Матрица смежности имеет размер  $n \times n$ . На пересечении  $i$ -й строки и  $j$ -го столбца в матрице смежности стоит единица, если в графе есть дуга, идущая из вершины  $i$  в вершину  $j$ . В остальных клетках матрицы записаны нули.

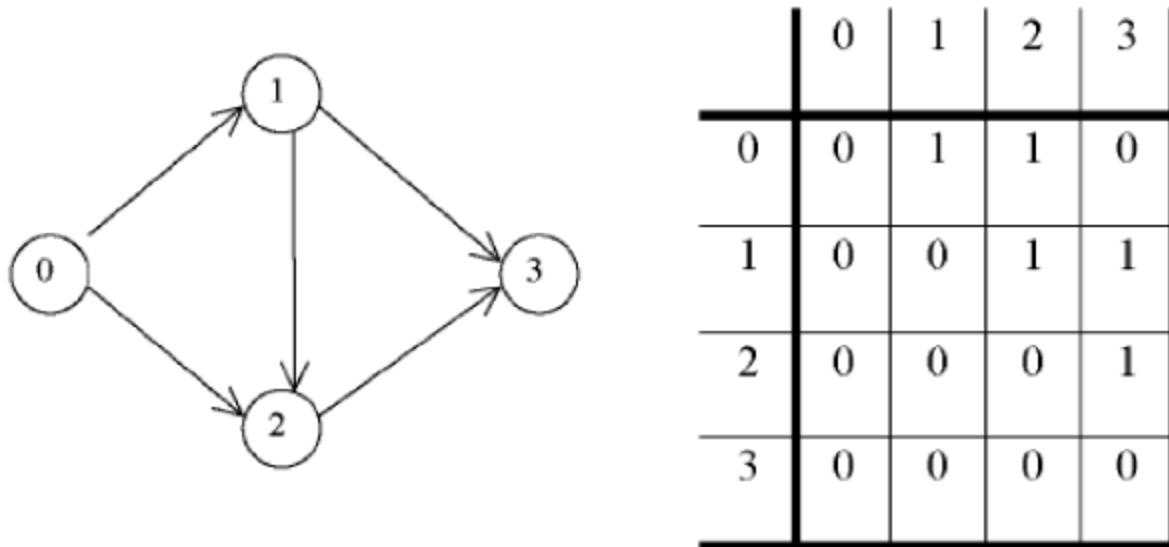


Рис. 2.2

На рис. 2.2 изображен ориентированный граф, а справа – его матрица смежности. И хотя в теоретических курсах любят рассказывать про матрицу смежности – это плохой способ представления сети в компьютере.

Перечислим недостатки матрицы смежности.

1. Матрица смежности занимает очень много места в памяти компьютера –  $n^2$ , причем, как правило, большая часть хранимой информации – нули.

2. Просмотр дуг, выходящих из одной вершины, требует очень много действий:  $O(n)$ .

Например, для графа средних размеров, содержащего 4000 вершин и 16000 дуг, для хранения матрицы смежности потребуется  $4000 \times 4000 = 16\,000\,000$  (шестнадцать миллионов) ячеек памяти, причем для каждой вершины будет выделено место для 4000 дуг, тогда как реальное число дуг, выходящих из данной вершины, в среднем равно четырем. В каждой строке «бесполезные» ячейки матрицы будут занимать в среднем 3996 мест из 4000. Поэтому при просмотре дуг, выходящих из данной вершины, большая часть времени будет потрачена впустую. А поскольку в реальных задачах сети, как правило, достаточно велики, а количество дуг, выходящих из одной вершины, напротив, мало, то получается, что использовать матрицы смежности при решении реальных сетевых задач крайне неразумно.

### 2.3. Матрица инцидентности

Мы убедились, что матрица смежности – плохой способ кодировки сети. Но еще худший способ – матрица инцидентности.

Матрица инцидентности имеет размер  $n \times m$ , где  $n$  – число вершин сети, а  $m$  – число дуг. Строки матрицы инцидентности соответствуют вершинам сети, а столбцы – дугам. В матрице инцидентности ориентированного графа на пересечении  $i$ -й строки и  $k$ -го столбца стоит «+1», если дуга  $k$  выходит из вершины  $i$ , и «-1», если дуга  $k$  входит в эту вершину. На рис. 2.3 изображена матрица инцидентности того же графа, для которого мы выписывали матрицу смежности. Возле дуг находятся их номера.

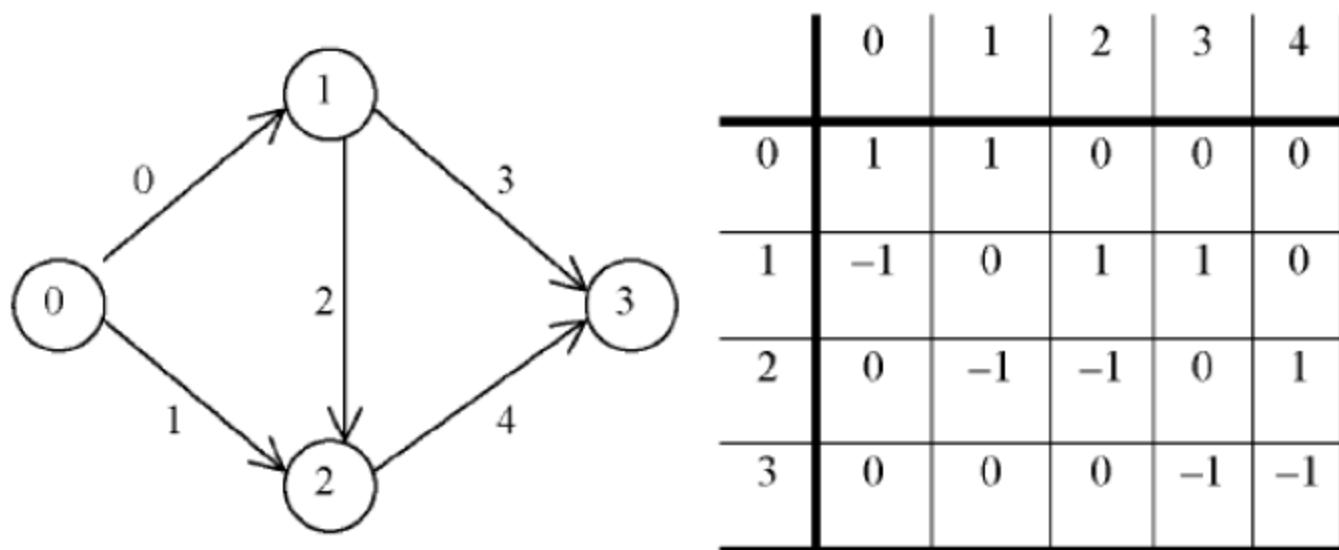


Рис. 2.3

Недостатки матрицы инцидентности такие же, как и у матрицы смежности, но в еще большем масштабе.

1. Матрица инцидентности занимает места больше, чем матрица смежности:  $n \times m$ .

2. На просмотр дуг, выходящих из вершины, тратится еще больше времени:  $O(m)$ .

Например, у графа, содержащего 4000 вершин, 16000 дуг, матрица инцидентности потребует  $4000 \times 16000 = 64\,000\,000$  (шестьдесят четыре миллиона) ячеек памяти, причем для каждой вершины будет выделено место на 16000 дуг, тогда как реальное число дуг, выходящих из нее, в среднем равно четырем.

Разумеется, в серьезных программах, работающих с графиками размером более 50...100, матрицы смежности и инцидентности не используются. Автор надеется, что вы в своих программах тоже больше никогда не будете их применять.

Рассмотрим более рациональные способы кодировки сетей.

## 2.4. Список дуг

Будем считать, что все вершины сети пронумерованы числами от 0 до  $(n-1)$ , а дуги – числами от 0 до  $(m-1)$ . Информация о дугах хранится в двух массивах:  $I[0..m-1]$  и  $J[0..m-1]$ . В массиве  $I$  хранятся начала дуг, а в массиве  $J$  – их концы, то есть для  $k$ -й дуги  $I[k]$  – номер вершины, из которой она выходит (*tail*), а  $J[k]$  – номер вершины, в которую эта дуга входит (*head*). Если у дуги есть еще какой-либо числовые характеристики: длина, пропускная способность, цена проезда, коэффициент усиления и т.п., то для каждого из таких параметров понадобится дополнительный массив. Если нужно хранить длину дуги, то заведем дополнительный массив  $C[0..m-1]$ , где переменная  $C[k]$  будет содержать длину  $k$ -й дуги. Если перевести это на более современный язык, то можно сказать, что одной дуге сети соответствует некоторая структура, а вся сеть – это массив структур. Однако из соображений наглядности мы будем по старинке использовать для представления сети обычные массивы, тем более что на понимании сути дела это никак не сказывается, а каждый, кто захочет, сможет завести в своих программах «правильные» структуры. На рис. 2.4 изображен граф, рассматривавшийся в предыдущих подразделах, и соответствующий ему список дуг.

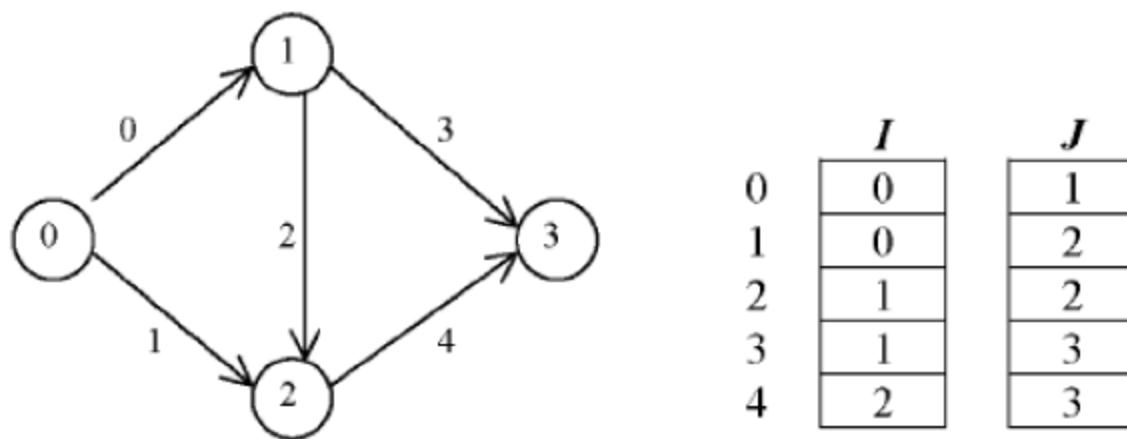


Рис. 2.4

Для хранения списка дуг графа, содержащего  $n$  вершин и  $m$  дуг, нам потребуется  $2m$  переменных, то есть объем памяти линейно зависит от размеров сети. Если же понадобится хранить еще какие-либо характеристики дуг сети (длину, пропускную способность и т.п.), то каждый из таких параметров потребует дополнительно  $m$  переменных. Очевидно, что с позиций рационального использования памяти

список дуг несравненно лучше и матрицы смежности, и матрицы инцидентности.

Так, для графа, содержащего 4000 вершин, 16000 дуг, потребуется  $16000 \times 2 = 32000$  переменных, тогда как для матрицы смежности необходимо 16 000 000, а для матрицы инцидентности – 64 000 000 переменных. Конечно, современные компьютеры по объему оперативной памяти на порядки превосходят своих предшественников, но это не значит, что память можно тратить впустую.

В приведенном примере случайно оказалось, что дуги сети упорядочены по начальной вершине. Конечно, в общем случае это может быть не так, поскольку нумерация дуг и вершин произвольна. Про список дуг часто говорят, что дуги в нем «лежат навалом», то есть не упорядочены ни по началу, ни по концу. Если алгоритм позволяет просматривать дуги, выходящие из разных вершин, вперемежку, то такой способ хранения дуг является вполне удовлетворительным. В нашем курсе лекций примером такого алгоритма будет служить алгоритм построения кратчайшего связывающего дерева. Однако абсолютное большинство сетевых алгоритмов требуют просматривать подряд все дуги, выходящие из заданной вершины. Поскольку заранее для произвольного списка дуг неизвестно, где именно в массивах расположены интересующие нас дуги, то для их просмотра придется пройти целиком массив  $\Gamma$ . Таким образом, для просмотра дуг, выходящих из заданной вершины, нам потребуется порядка  $O(m)$  операций, что ничуть не лучше, чем при использовании матрицы инцидентности. Так что же: получается, что, сэкономив память, мы проиграли по времени? Оказывается, что это не так и, слегка усовершенствовав представление списка дуг, мы сможем сэкономить не только память, но и время.

Существует два часто применяемых способа решения этой задачи: упорядоченный список дуг и списки пучков дуг.

## 2.5. Упорядоченный список дуг

Будем называть *пучком* множество дуг, выходящих из одной вершины. Предположим, что список дуг является упорядоченным, то есть дуги расположены в порядке возрастания номера начальной вершины. Это значит, что дуги, входящие в один пучок, в списке расположены последовательно. Допустим, что у нас дополнительно имеется массив входов  $S[0..n]$ , содержащий информацию о позициях, в которых расположены первые дуги каждого пучка. То есть в  $S[i]$  записан но-

мер первой дуги, выходящей из вершины  $i$ . Поскольку массив  $\Gamma$  упорядочен, то все дуги, выходящие из  $i$ , расположены, начиная с позиции  $S[i]$  до позиции  $S[i+1]-1$ . Если дуг, выходящих из  $i$ , нет, то  $S[i+1] = S[i]$ . Переменная  $S[n]$  всегда равна  $m$ . Она нужна, чтобы пучок дуг, выходящих из вершины  $(n-1)$ , удавалось просматривать по тем же правилам, что и остальные. Ниже приведен массив входов  $S$  для графа, изображенного на предыдущем рис. 2.4:

$S$	0	2	4	5	5
	0	1	2	3	4

Приведем фрагмент программы, позволяющий просматривать пучок дуг, выходящих из заданной вершины  $i$ .

```
// Просмотр пучка дуг, выходящих из вершины i
for ( k=S[i]; k < S[i+1]; k++ )
{ // просматриваем k-ю дугу
    // i - начало k-й дуги
    j = J[k]; // j - конец k-й дуги
    // выполняем необходимые операции с дугой (i,j)
    . . . . .
} // обход пучка закончен
```

При таком алгоритме просмотра количество действий, затраченных на просмотр пучка дуг, пропорционально числу дуг пучка и не зависит от размеров сети. Таким образом, на просмотр одной дуги расходуется  $O(1)$  операций, то есть меньше и быть не может. Осталось только научиться по заданному списку дуг быстро строить упорядоченный список дуг. Оказывается это можно сделать очень эффективно с помощью следующего алгоритма.

## 2.6. Построение упорядоченного списка дуг

Для построения упорядоченного списка дуг можно было бы применить один из стандартных методов сортировки. Сложность лучших алгоритмов сортировки в общем случае равна  $O(m \log m)$ , где  $m$  – количество упорядочиваемых элементов. Мы справимся быстрее. Воспользовавшись спецификой сети, можно предложить более эффективный способ упорядочения дуг и построения массива входов  $S$ , требующий порядка  $O(m)$  операций, где  $m$  – количество дуг сети. Это означает, что сложность предлагаемого алгоритма упорядочения сети по порядку

величины будет минимально возможной, поскольку при упорядочении дуг посмотреть на каждую из них придется обязательно.

## Алгоритм быстрого упорядочения дуг сети

### Этап 1

Подсчитаем количество дуг, выходящих из каждой вершины. Для этого просмотрим массив  $I$  (начальные вершины дуг). Число дуг, выходящих из  $i$ -й вершины, будем запоминать в переменной  $S[i+1]$ . Трудоемкость этого этапа равна  $O(m)$ , так как количество операций пропорционально числу элементов массива  $I$ .

```
// Вычисление размеров пучков
S[0..n] = 0;
for( k = 0; k < m; k++ ) //просмотр дуг сети
{
    i = I[k]; //номер вершины, из которой
               //выходит дуга k
    S[i+1]++; //количество дуг, выходящих из
               //вершины i, увеличиваем на 1.
}
```

После завершения первого этапа  $S[0]=0$ , а переменная  $S[i+1]$  содержит число дуг, выходящих из вершины  $i$ .

### Этап 2

На этом этапе заполним массив входов  $S$ . Напомним, что для массива входов в переменной  $S[i]$  должна содержаться позиция, начиная с которой в упорядоченном списке должны располагаться дуги, выходящие из вершины  $i$ . Если обозначить через  $d_j$  количеству дуг, выходящих из  $j$ -й вершины, то ясно, что  $S[i] = \sum_{j=0}^{i-1} d_j$ . Таким

образом, после первого этапа в массиве  $S$  содержится достаточно информации, чтобы заполнить массив входов. Очевидно, что в массиве входов  $S[0]=0$ , а  $S[i+1] = S[i] + d_i$ . Но поскольку после завершения первого этапа  $S[i+1] = d_i$ , то выполняется следующее рекуррентное соотношение:  $S_{\text{NEW}}[i+1] = S_{\text{NEW}}[i] + S_{\text{OLD}}[i+1]$ .

```
// Заполнение массива входов
for( i = 1; i ≤ n; i++ )
    S[i] += S[i-1];
// Вот и все – дальше объяснять
```

После завершения второго этапа в массиве  $S$  будут записаны входы упорядоченного списка дуг. Легко видеть, что трудоемкость этапа 2 –  $O(n)$ .

### Этап 3

На втором этапе был заполнен массив входов. Иными словами, мы «разметили» массив  $\Gamma$ . Теперь известно, где должны располагаться дуги, выходящие из вершины  $i$ . Они должны находиться в позициях от  $S[i]$  до  $S[i+1] - 1$ . Осталось только расставить дуги по местам, что и будет сделано на третьем этапе алгоритма.

Будем называть *блоком* набор позиций в массивах  $\Gamma$  и  $J$ , в которых должны располагаться дуги одного пучка. Так,  $i$ -й блок должен содержать дуги, выходящие из  $i$ -й вершины и, соответственно, занимать места от  $S[i]$  до  $S[i+1] - 1$ . Выделим массив указателей  $P[0..n-1]$ . Указатель  $P[i]$  показывает на позицию в  $i$ -м блоке. На этом этапе в процессе работы алгоритма в первом сегменте блока от  $S[i]$  до  $P[i] - 1$  порядок уже наведен – там находятся дуги, выходящие из вершины  $i$ . А во втором сегменте блока от  $P[i]$  до  $S[i+1] - 1$  порядок не гарантирован – там могут располагаться произвольные дуги. В начале третьего этапа «хорошие» сегменты блоков пусты, поскольку мы не знаем, как расположены дуги сети. По ходу третьего этапа, расставляя дуги по местам, мы будем расширять «хорошие» сегменты блоков и, соответственно, уменьшать «плохие» до тех пор, пока все блоки не станут «хорошими» полностью.

В ходе третьего этапа мы последовательно рассматриваем все блоки от 0-го до  $(n - 1)$ -го и в каждом наводим порядок. К следующему блоку мы переходим только после того, как в текущем хороший сегмент стал занимать весь блок.

Одна итерация расстановки дуг по местам состоит в том, что в  $i$ -м блоке мы рассматриваем дугу, находящуюся в позиции  $P[i]$ . Если эта дуга выходит из вершины  $i$ , то она находится на своем месте. Расширяем правильный сегмент блока и переходим к рассмотрению следующей дуги.

Если же рассматриваемая дуга выходит из другой вершины, например из  $j$ , то меняем ее местами с первой дугой из плохого сегмента блока  $j$ , то есть с дугой, на которую показывает  $P[j]$ . При этом хороший сегмент блока  $j$  расширяется, потому что перешедшая в этот блок дуга выходит из вершины  $j$ , и, следовательно, попадает на свое место.

После перестановки опять рассматриваем дугу, на которую указывает  $P[i]$ , то есть дугу, только что попавшую на эту позицию из блока  $j$ . На рис. 2.5 проиллюстрирована одна итерация алгоритма, во время которой меняются местами дуги блоков  $i$  и  $j$ .

После того как от плохого сегмента блока  $i$  ничего не останется, переходим к рассмотрению следующего блока и так далее, пока не обработаем их все.

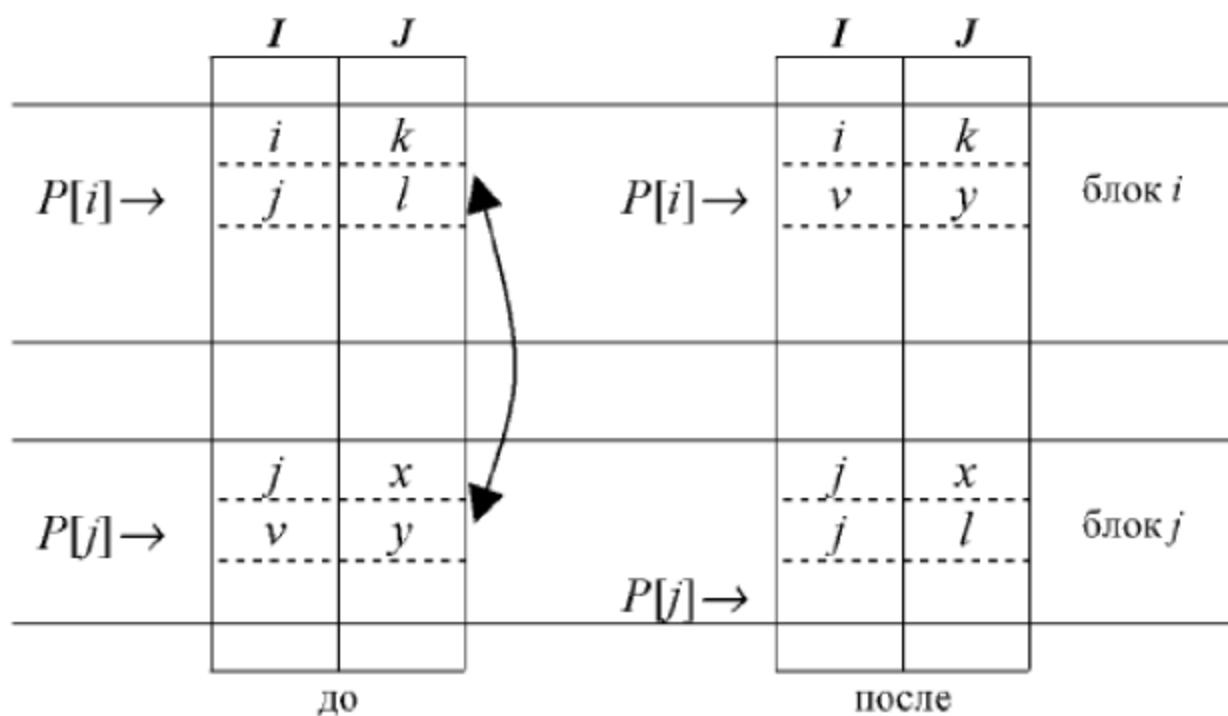


Рис. 2.5

Оценка сложности третьего этапа –  $O(m)$ . Это следует из того, что на приведение всех блоков в порядок нам потребуется не более чем  $m$  перестановок, так как при любой перестановке хотя бы одна дуга попадает на свое место и больше уже никуда перемещаться не будет. Очевидно, что количество действий третьего этапа, не связанных с осуществлением перестановок, не превышает  $O(m)$ , и, следовательно,  $O(m)$  является общей оценкой сложности третьего этапа.

Суммируя оценки сложности всех трех этапов, получаем оценку сложности всего алгоритма:  $O(m) + O(n) + O(m) = O(m)$ .

```

// Расстановка дуг по местам
P[0..n-1] = S[0..n-1]
for( i = 0; i < n-1; i++ ) // Цикл по всем
                           // блокам, кроме
                           // последнего
// Обработка i-го блока
for( k = P[i]; k < S[i+1]; )
{
    if( I[k] == i ) //дуга стоит на своем месте
        k++;           //двигаемся дальше
    else
        {j=I[k];          //дуга не на месте
         l=P[j];          //она выходит из вершины j
         //перемещаем ее в j-й блок
         l=P[j];          //куда ставить дугу k
         //в блоке j
         // меняем местами дугу k и дугу l
         I,J[k] ⇔ I,J[l];
         P[j]++;          //передвигаем указатель в блоке j
         //хорошая часть блока j
         //увеличилась
    }
}
} // конец обработки блока

```

Обратите внимание на одну деталь. Мы не стали обрабатывать последний  $(n-1)$ -й блок, потому что если во всех предыдущих блоках все дуги стоят на месте, то и в нем находятся только дуги, выходящие из  $(n-1)$ -й вершины.

Если для дуги надо хранить еще и дополнительную информацию: длину, пропускную способность и т.п., то алгоритм упорядочения почти не меняется; просто при перестановке дуги надо еще представлять соответствующие элементы дополнительных массивов.

Упорядоченный список дуг является весьма удобным и экономичным способом хранения информации о сети, как с точки зрения используемой памяти, так и с точки зрения времени прохода по заданному пучку дуг. К тому же построение такого списка, как мы с вами убедились, выполняется очень быстро. По всем этим причинам во многих программах, реализующих сетевые алгоритмы, используется именно такой способ представления сети в компьютере.

Однако даже самые хорошие способы не бывают идеальными и их применение в отдельных случаях оказывается не очень удобным. К не-

достаткам упорядоченного списка дуг можно отнести его не слишком большую «гибкость». Представьте себе, что по каким-либо причинам сеть изменилась: какие-то дуги были в нее добавлены, а какие-то удалены. Тогда, чтобы и дальше работать с такой сетью, понадобится заново упорядочивать все дуги. Если изменения сети происходят постоянно, то это может представлять определенные неудобства.

Рассмотрим еще один способ представления сети в компьютере, позволяющий гораздо быстрее реагировать на изменения сети.

## 2.7. Списки пучков дуг

Сцепим дуги каждого пучка при помощи одностороннего списка. Для этого нам дополнительно понадобятся два массива  $H[0..n-1]$  и  $L[0..m-1]$ . В массиве  $H$  мы будем хранить головы списков:  $H[i]$  – номер первой дуги в списке дуг, выходящих из вершины  $i$ . В массиве  $L$  будем запоминать ссылки элементов списков друг на друга:  $L[k]$  – номер следующей в списке дуги, выходящей из той же вершины, что и дуга  $k$ . Последняя дуга списка ссылается на фиктивный номер, в качестве которого мы будем использовать  $-1$ . Таким образом, каждый пучок дуг будет представлен в виде одностороннего списка, начинающегося головой  $H[i]$  и заканчивающийся  $-1$ . Несмотря на то что всего списков  $n$ , удается «упаковать» все ссылки в одном массиве  $L$ , поскольку каждая дуга включена только в один список, и поэтому надо хранить для нее только одну ссылку.

На рис. 2.6 приведены списки пучков дуг для того же графа, в котором дуги перенумерованы так, чтобы в массивах они не располагались последовательно. Порядок появления дуг пучка в списке может быть произвольным, поэтому нет никаких противоречий в том, что в приведенных списках дуги идут «сверху вниз».

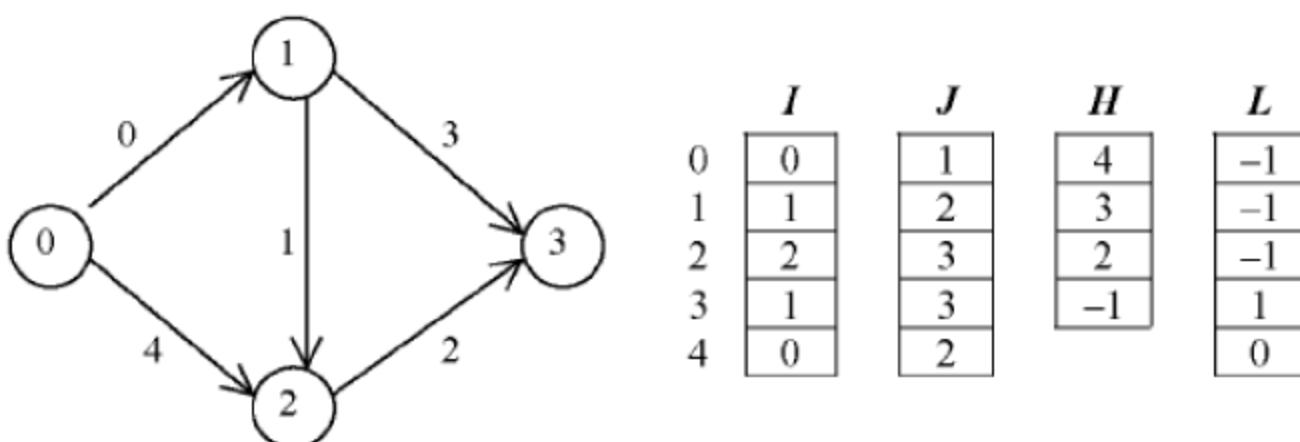


Рис. 2.6

В массивах  $H$  и  $L$  закодированы 4 списка, каждый из которых соответствует пучку дуг. Пучок  $i$  содержит дуги, выходящие из  $i$ -й вершины:

$0 : 4 \rightarrow 0 \rightarrow -1;$	$(H[0] = 4, L[4] = 0, L[0] = -1)$
$1 : 3 \rightarrow 1 \rightarrow -1;$	$(H[1] = 3, L[3] = 1, L[1] = -1)$
$2 : 2 \rightarrow -1;$	$(H[2] = 2, L[2] = -1)$
3: пуст;	$(H[3] = -1)$

Приведем фрагмент программы, позволяющий обойти все дуги, выходящие из вершины  $i$ .

```
// Просмотр пучка дуг, выходящих из вершины i
for ( k = H[i]; k != -1; k = L[k] )
{
    // просматриваем k-ю дугу
        // i - начало k-й дуги
    j = J[k]; // j - конец k-й дуги
    // выполняем необходимые операции с дугой (i, j)
    . . . . .
} // обход пучка закончен
```

Обратите внимание на заголовок цикла. Такой заголовок часто будем использоваться для обхода пучка дуг в различных программах, реализующих сетевые алгоритмы.

Трудоемкость обхода пучка дуг – такая же, как и в случае для упорядоченного списка: пропорциональна количеству дуг пучка. Таким образом, на просмотр одной дуги мы также тратим порядка  $O(1)$  операций, а лучшего и желать невозможно. Правда, по сравнению с упорядоченными списками нам пришлось завести дополнительный массив  $L$  размером  $m$ , но, пожалуй, это не слишком большая плата за то, что при изменениях сети списки пучков дуг гораздо легче перестраивать, чем сортировать дуги заново в упорядоченных списках.

Осталось убедиться, что можно построить списки пучков дуг так же легко и быстро, как упорядочить дуги сети. Оказывается, что сделать это еще проще.

## 2.8. Построение списков пучков дуг

Программа построения списков пучков дуг крайне проста. Она сводится к последовательному просмотру дуг и добавлению очередной дуги в список, соответствующий ее начальной вершине.

```
// Инициализация списков
// Вначале все списки пусты
H[0...n-1] = -1;
// Обход дуг сети
for ( k = 0; k < m; k++ )
{
    i = I[k]; // начальная вершина дуги
    // добавляем дугу k в начало списка i
    L[k] = H[i]; // k ссылается на дугу, которая
                  // раньше была первой в списке i
    H[i] = k; // k становится первой дугой в
              // списке i
} // списки пучков дуг построены
```

Как видите – проще некуда. Цикл содержит всего три оператора. Очевидно, что сложность построения списка пучков дуг составляет  $O(m)$ , равно как и сложность построения упорядоченного списка дуг.

Очевидно, что по основным характеристикам упорядоченный список дуг и списки пучков дуг практически одинаковы. Какой способ хранения сети предпочтеть? Однозначного ответа нет. Трудоемкость просмотра пучка дуг и организации структур в обоих случаях одинакова, но списковые структуры более гибки и легче настраиваются на нестандартные ситуации. Представьте себе, что надо просматривать пучки не только выходящих, но и входящих дуг. Если мы работаем со списками, то все просто: надо выделить еще два массива  $H_j$ ,  $L_j$  и хранить в них списки пучков входящих дуг. Конечно, можно организовать структуру данных, позволяющую эффективно просматривать входящие и выходящие дуги и при упорядоченной сети, но это будет более громоздко.

В обоих случаях для просмотра дуги требуется константа операций. Фактически просмотр дуги сводится к проверке окончания цикла и переходу к следующей дуге. Проверка окончания цикла в обоих случаях «стоит» одинаково, а вот оператор перехода для упо-

рядоченных списков « $k++$ » выполняется быстрее, чем оператор перехода для списков пучков дуг « $k=L[k]$ ». К тому же дуги пучка в упорядоченном списке расположены в памяти рядом, что может ускорить работу программы при активном использовании кэш-памяти по сравнению со списками пучков дуг, где дуги одного пучка, вообще говоря, разбросаны по массиву. Конечно, можно сказать, что все это мелочи, однако иногда именно такие мелочи влияют на скорость работы программы, тем более что циклы просмотра пучков дуг часто бывают «очень внутренними».

Подытоживая все эти рассуждения, можно сказать, что на самом деле оба предложенных способа являются хорошими и в абсолютном большинстве случаев оба будут работать эффективно. Какой из них лучше использовать в программе, зависит от особенностей задачи, деталей и от индивидуальных склонностей. Так, автор больше любит пользоваться списками пучков дуг, но, когда надо, не пренебрегает упорядоченными списками.

## 2.9. Представление неориентированной сети

До сих пор мы рассматривали только ориентированные сети. Трудность при кодировке *неориентированной сети* состоит в том, что мы не знаем, какая вершина является началом, а какая концом, то есть ребро  $[i, j]$  «выходит» одновременно из двух вершин и «входит» в них же. Простейший способ решения этой проблемы состоит в удвоении числа ребер. Вместо одного ребра  $[i, j]$  можно хранить две дуги:  $(i, j)$  и  $(j, i)$ . Иногда при сведении задач на неориентированной сети к задачам на ориентированной сети именно так и поступают, однако не всегда этот способ является оптимальным. Во-первых, потому что удваивается объем памяти, необходимый для хранения сети, а во-вторых, потому что не всегда удобно, что информация об одной дуге рассредоточена по двум позициям.

Поэтому мы рассмотрим еще один способ представления неориентированной сети, который позволяет обойтись без дублирования дуг и удобен не только для неориентированной, но и для ориентированной сети в случае, когда нужно уметь двигаться по дуге в разных направлениях (например, в задаче построения максимального потока).

Будем располагать информацию о ребрах вместо двух массивов  $I$  и  $J$  в одном массиве  $IJ$  длиной  $2m$ , где  $m$  – число ребер сети. Ребро  $[i, j]$  с номером  $k$  записывается в двух позициях массива  $IJ$ :

$IJ[k] = i$ ,  $IJ[2m-k-1] = j$ . Образно говоря, мы берем массив  $J$ , переворачиваем его «вверх ногами» и «приставляем снизу» к массиву  $I$ . На рис. 2.7 проиллюстрирован переход от двух массивов  $I$  и  $J$  к одному  $IJ$ .

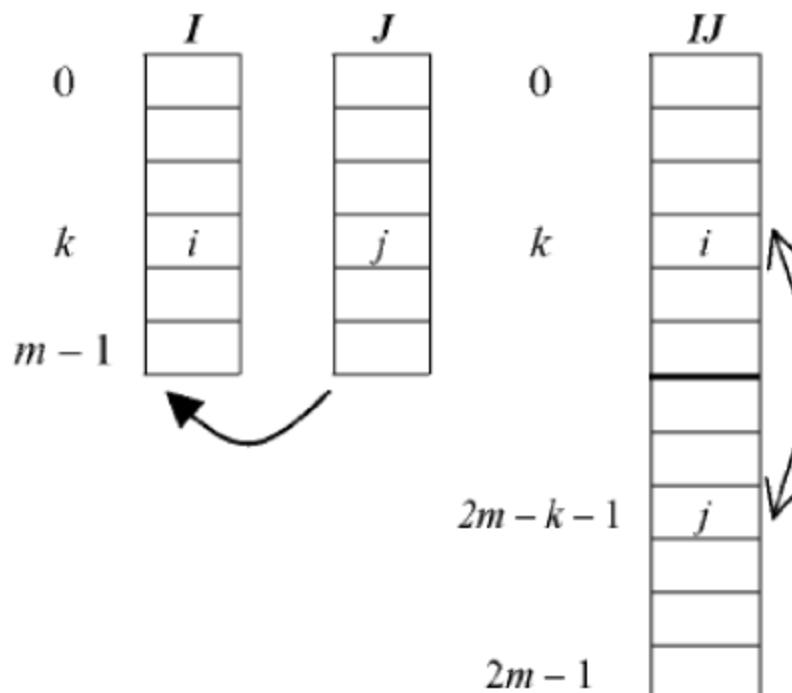


Рис. 2.7

Теперь, если один конец ребра находится в позиции  $k$ , то другой его конец будет находиться в позиции  $2m-1-k$ , независимо от того, меньше  $k$ , чем  $m$ , или больше. Именно то, что при таком представлении очень легко переходить от одного конца ребра к другому, и является причиной его выбора. Мы могли бы приставить массив  $J$  к массиву  $I$  снизу «не переворачивая», то есть хранить ребро в позициях  $k$  и  $m-1+k$ , но тогда при переходе от одного конца к другому пришлось бы разбираться, в какой половине массива мы находимся, и в зависимости от этого либо прибавлять, либо вычитать  $m$ . А так можно обойтись простой формулой, единой для всех случаев.

Если необходимо хранить какие-либо другие сведения о ребрах, то, как правило, их запоминают в дополнительных массивах длиной  $m$ , в которых информация о ребре  $k$  находится в  $k$ -й позиции.

Для просмотра всех ребер, инцидентных данной вершине, надо построить списки. Каждое ребро должно попасть в два списка, соответствующих вершинам, расположенным на концах этого ребра.

Начала списков, как и прежде, будем хранить в массиве  $H[0..n-1]$ , а ссылки в массиве  $L[0..2m-1]$ . Программа, строя-

щая такие списки, практически не отличается от программы построения списков пучков дуг для ориентированной сети:

```
// Инициализация списков
// Вначале все списки пусты
H[0..n-1] = -1;
// Обход ребер сети
for ( k = 0; k < 2*m; k++ )
{
    i = IJ[k]; // один из концов ребра
    // добавляем ребро в начало списка i
    L[k] = H[i];
    H[i] = k;
} // списки пучков ребер построены
// каждое ребро попало в два списка
```

Обход списка ребер, инцидентных вершине  $i$ , также практически не отличается от аналогичной процедуры для ориентированной сети:

```
for( k = H[i]; k != -1; k = L[k] )
{
    // i - один конец ребра
    j = IJ[2*m-1-k]; // другой его конец
    // выполняем все действия с ребром [i,j]
    . . .
} // обход пучка ребер закончен
```

Теперь мы готовы к тому, чтобы писать программы, реализующие сетевые алгоритмы. Рассмотренные в этом разделе способы представления сети в компьютере или их небольшие изменения и дополнения позволяют в абсолютном большинстве случаев хорошо реализовывать сетевые алгоритмы.

### 3. АЛГОРИТМ НАХОЖДЕНИЯ КОМПОНЕНТ СВЯЗНОСТИ ГРАФА

#### 3.1. Задача нахождения компонент связности графа

Дан неориентированный граф  $G = \{N, E\}$ , где  $N$  – множество вершин графа,  $E$  – множество ребер.

*Путем* в графе называется последовательность ребер  $W = [i_0, j_0], [i_1, j_1], \dots, [i_k, j_k], \dots, [i_t, j_t]$ , такая что для любого  $k$   $j_k = i_{k+1}$ .

Вершины  $i_0, i_1, \dots, i_k, \dots, i_t, j_t$  входят в путь  $W$  или, иначе говоря, являются вершинами этого пути. Путь называется *простым*, если все его вершины различны.

*Циклом* называется путь, у которого начальная и конечная вершины совпадают.

Вершины графа  $i$  и  $j$  называются *связанными*, если существует путь, их соединяющий, то есть путь, начинающийся в  $i$  и заканчивающийся в  $j$ . На неориентированном графе отношение связности является отношением эквивалентности.

Напомним, что отношение эквивалентности  $\cong$  должно удовлетворять трем свойствам:

1. *Рефлексивность*:  $\forall i \ i \cong i$ .
2. *Симметричность*:  $\forall i, j \text{, если } i \cong j \text{, то } j \cong i$ .
3. *Транзитивность*:  $\forall i, j, k \text{, если } i \cong j \text{ и } j \cong k \text{, то } i \cong k$ .

Нетрудно убедиться, что в неориентированном графе для отношения связности все три перечисленные свойства выполняются.

Таким образом, отношение связности разбивает вершины графа на *классы эквивалентности*, которые называются *компонентами связности*. Любые две вершины из одной компоненты связности могут быть соединены путем. Если же вершины входят в разные компоненты связности, то в графе не существует пути, их соединяющего.

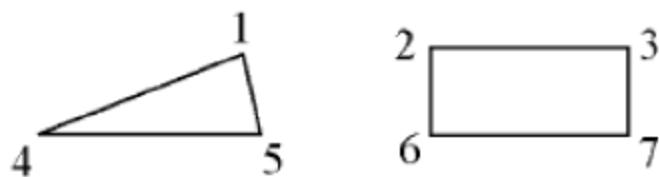


Рис. 3.1

На рис. 3.1 изображен граф из семи вершин. Вершины 1, 4 и 5 входят в одну компоненту связности, а вершины 2, 3, 6 и 7 – в другую.

Граф называется *связным*, если он состоит из одной компоненты связности, то есть любые две его вершины могут быть соединены путем.

Мы будем решать задачу нахождения компонент связности неориентированного графа. Результатом работы алгоритма нахождения компонент связности графа будет число компонент связности графа и указанный для каждой вершины номер компоненты связности, в которую эта вершина вошла.

Алгоритм находит вершину, которая еще не попала ни в одну компоненту связности, и, начиная с нее, обходит все вершины, которые находятся в той же компоненте, что и исходная.

Если после этого осталась хотя бы одна вершина, которая еще не вошла ни в одну компоненту связности, то, начиная с нее, повторяется построение компоненты.

Нахождение всех вершин, входящих в одну компоненту связности, является основной частью алгоритма. С этой целью будем использовать обход графа, называемый *поиском в глубину* (*depth first search*).

### 3.2. Алгоритм обхода одной компоненты связности

Образно говоря, при поиске в глубину мы хотим попасть в те вершины графа, в которых еще не побывали, и продвигаемся вперед, не заботясь о том, что, может быть, какие-то вершины «сбоку» от основного пути остались еще непросмотренными.

Алгоритм обхода графа состоит из трех основных блоков:

1. *Просмотр ребер.*
2. *Шаг вперед.*
3. *Шаг назад.*

Во время работы алгоритма какая-то одна вершина является «текущей» и мы просматриваем выходящие из нее ребра. Кроме того, постоянно хранится стек «активных вершин»  $S = i_0, i_1, \dots, i_k$ .

Обход компоненты начинается с некоторой вершины  $i_0$ . Вершину  $i_0$  считаем помеченной, а все остальные вершины – непомеченными. Все ребра считаем непросмотренными. В начале обхода стек активных вершин пуст, а текущей вершиной является  $i_0$ .

**Просмотр ребер.** Берем первое непросмотренное ребро, выходящее из текущей вершины  $i$ , и помечаем его как просмотренное. Пусть  $j$  – вершина, расположенная на противоположном конце этого ребра. Если  $j$  – непомеченная вершина (мы в ней еще не были), – делаем *шаг вперед*. Если  $j$  – помеченная вершина (мы в ней уже были), – переходим к следующему непросмотренному ребру, выходящему из вершины  $i$ .

Если все ребра, выходящие из  $i$ , просмотрены, – делаем *шаг назад*.

**Шаг вперед.** Помечаем вершину  $j$ . Вершину  $i$  заносим в стек активных вершин. Объявляем вершину  $j$  текущей и переходим к *просмотру ребер*, выходящих из этой вершины.

**Шаг назад.** Берем вершину из стека<sup>1</sup>. Объявляем ее текущей и переходим к *просмотру ребер*, выходящих из этой вершины.

Если стек активных вершин пуст, то обход компоненты связности закончен, помечены все вершины, которые входят в ту же компоненту, что и  $i_0$ .

Отметим одно свойство стека активных вершин. Из любой вершины этого стека  $i_t$  в следующую вершину стека  $i_{t+1}$  ведет ребро, вдоль которого был сделан *шаг вперед*, а из последней вершины стека такое ребро ведет в текущую вершину. Таким образом, из самой первой вершины стека  $i_0$  в текущую вершину ведет путь. Следовательно, любая вершина, которая стала по ходу работы алгоритма текущей, связана с исходной вершиной  $i_0$  и, значит, входит в ту же компоненту связности, что и она, то есть все помеченные вершины входят в одну компоненту связности.

### 3.3. Пример работы алгоритма

Проиллюстрируем работу алгоритма на примере графа, состоящего из двух компонент связности (рис. 3.2).

Ребра возле вершин пронумерованы в том порядке, в котором мы их будем просматривать в ходе работы алгоритма.

Обход графа начнем с вершины 0. Начинаем просмотр ребер с ребра 1. Это ребро ведет в непомеченную вершину 1. Делаем шаг вперед и помечаем эту вершину. Вершина 1 становится текущей, а в стеке появляется вершина 0.

---

<sup>1</sup> Из свойств стека следует, что забираем мы вершину, которая попала в него последней.

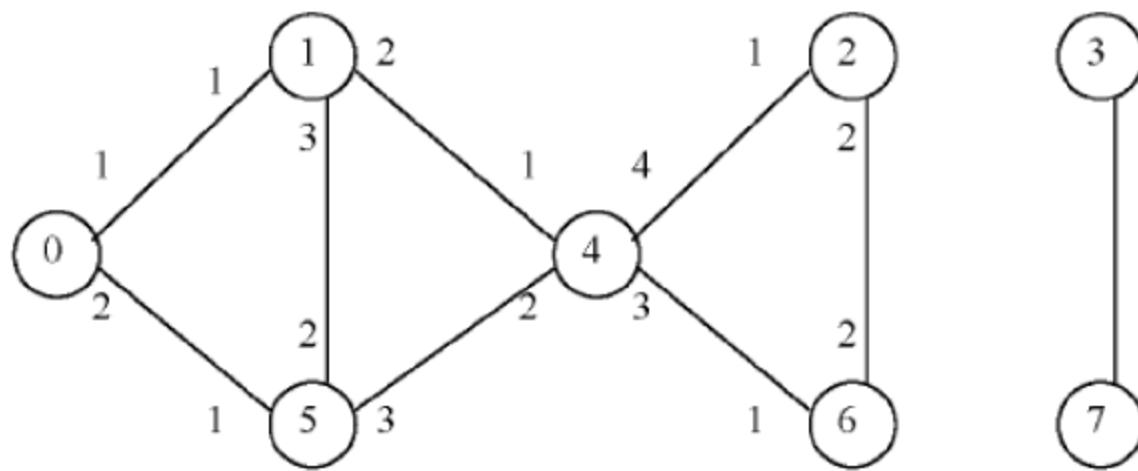


Рис. 3.2

Просматриваем ребра, выходящие из вершины 1. Ребро 1 ведет в помеченную вершину. Помечаем это ребро как просмотренное и переходим к следующему ребру 2. Оно ведет в непомеченную вершину 4, поэтому делаем шаг вперед. В стеке – вершины 0, 1.

Смотрим на ребра, выходящие из вершины 4. Ребро 1 ведет назад, а ребро 2 приводит в непомеченную вершину 5. В стеке – вершины 0, 1, 4.

Просматривая ребра, выходящие из вершины 5, убеждаемся, что все они ведут в помеченные вершины. Просмотр вершины 5 закончен. Делаем шаг назад.

Берем из стека последнюю вершину 4 и начинаем просмотр ребер с первого ребра, оставшегося непросмотренным, – это ребро 3. Вдоль ребра 3 делаем шаг вперед в вершину 6, а из нее – шаг вперед в вершину 2. В стеке – вершины 0, 1, 4, 6.

В каждой из этих вершин нам придется сделать шаг назад, потому что ни одно из ребер, оставшихся непросмотренным, не ведет в непомеченную вершину.

При попытке сделать шаг назад из вершины 0 убеждаемся, что стек пуст. Это значит, что обход компоненты связности закончен. Мы последовательно обошли вершины 0, 1, 4, 5, 6, 2, которые образуют одну из компонент связности графа.

Две вершины – 3 и 7 – остались непомеченными. Следовательно, граф несвязный, то есть состоит более чем из одной компоненты связности. Начиная с вершины 3 обойдем вторую компоненту, после чего непомеченных вершин не останется.

Обход графа, используемый в этом алгоритме, является поиском в глубину «в чистом виде». Собственно говоря, кроме обхода вершин, как такового, мы почти ничего больше в этом алгоритме не

делаем. Поиском в глубину такой обход назван потому, что мы пытаемся «пробраться» как можно дальше в глубь графа. Характерными чертами такого обхода являются:

- наличие стека активных вершин, который представляет собой путь от первой вершины стека до текущей вершины;
- необходимость запоминания для каждой вершины этого пути первой непросмотренной дуги.

Для лучшего усвоения алгоритма поиска в глубину можно позволить себе слегка фривольное сравнение субъекта,двигающегося по вершинам графа, с ветреным молодым человеком. Как только он видит новую девушку, с которой раньше знаком не был, то сразу переходит к ней, бросая свою «текущую» подругу (вершину). Он никогда не идет вновь к девушке, с которой уже был знаком, а возвращается обратно к прежней подруге только после того, как обошел всех, кого смог. В жизни такое поведение нельзя признать образцовым, но в целом ряде алгоритмов на графах, в том числе и в очень сложных, именно такая стратегия приводит к успеху.

### 3.4. Обоснование корректности алгоритма

Докажем корректность алгоритма обхода одной компоненты связности. Из свойства стека активных вершин мы знаем, что все вершины, помеченные при поиске в глубину, начиная с заданной вершины  $i_0$ , находятся в одной компоненте связности. Осталось доказать, что любая вершина, в которую ведет какой-либо путь из  $i_0$ , будет помечена в ходе работы алгоритма.

Пусть  $i_0, i_1, \dots, i_k, \dots, i_t$  – вершины произвольного пути  $W$ , выходящего из вершины  $i_0$ . Докажем по индукции, что все вершины этого пути будут помечены при поиске в глубину. Вершина  $i_0$  помечена с самого начала. Предположим, что вершины  $i_0, i_1, \dots, i_{k-1}$  были помечены в процессе обхода. Докажем, что и вершина  $i_k$  тоже была помечена. Из описания алгоритма следует, что в некоторый момент нам придется сделать шаг назад из вершины  $i_{k-1}$ . Однако прежде чем сделать шаг назад из этой вершины мы обязаны были просмотреть все выходящие из нее ребра. Следовательно, мы должны были просмотреть и на ребро  $(i_{k-1}, i_k)$ . Если в момент просмотра этого ребра вершина  $i_k$  еще оставалась непомеченной, то мы должны были сделать шаг вперед и пометить эту вершину. Таким образом, верши-

на  $i_k$ , а следовательно, и все вершины пути, выходящего из  $i_0$ , будут помечены в ходе поиска в глубину, а это значит, что все вершины, входящие в ту же компоненту связности, что и  $i_0$ , будут помечены.

Корректность алгоритма доказана.

### 3.5. Сложность алгоритма

Сложность алгоритма нахождения компонент связности оценить нетрудно. Количество шагов вперед не превышает  $n$ , поскольку шагнуть вперед мы можем только в непомеченную вершину, после чего помечаем ее и, следовательно, второй раз шаг вперед в нее не сделаем.

Число шагов назад также не превышает  $n$ . Сделать шаг назад можно только из текущей вершины. Вершина становится текущей только если в нее сделали шаг вперед или назад. Снова шаг вперед в такую вершину сделан быть не может, потому что эта вершина уже помечена. Шаг назад может быть сделан только в вершину, находящуюся в стеке. Но вершина, из которой сделан шаг назад, удалена из стека, а попасть в него не может, так как для этого нужно, чтобы в нее сделали шаг вперед.

Поскольку количество действий (операций), необходимых для того, чтобы сделать один шаг вперед или назад, по порядку составляет  $O(1)$ , то всего на исполнение шагов мы потратим не более чем  $O(n)$  действий, где  $n$  – число вершин графа.

Просмотр ребер в вершине каждый раз начинается с непросмотренного ранее ребра. Из этого следует, что каждое ребро, выходящее из данной вершины, мы обрабатываем не более одного раза. Следовательно, общее количество операций, необходимых для просмотра ребер, имеет порядок  $O(m)$ , где  $m$  – количество ребер графа.

Еще порядка  $O(n)$  операций потребуется для просмотра вершин с целью выяснения, попали ли эти вершины в какую-либо компоненту связности из рассмотренных ранее.

Суммируя все эти оценки сложности, получаем, что общая сложность алгоритма равна  $O(n) + O(m) + O(n) = O(m)$ . Таким образом, рассмотренный алгоритм имеет линейную оценку сложности, а меньшей оценки не может быть теоретически.

### 3.6. Реализация алгоритма

Первое, что нужно сделать при реализации любого алгоритма – спроектировать структуру данных. Структура должна соответствовать выполняемым операциям и не портить теоретическую оценку сложности. Проектирование структуры сводится к анализу необходимых для исполнения алгоритма действий и подбору правильной организации данных, которая позволит выполнить эти действия эффективно.

При просмотре ребер нам надо обрабатывать ребра, выходящие из текущей вершины. Поскольку граф неориентированный, то ребро  $[i, j]$  должно быть доступно и из вершины  $i$ , и из вершины  $j$ . В этой ситуации целесообразно воспользоваться структурой данных, предложенной в предыдущем разделе для неориентированного графа:

- массив  $IJ[0..2m-1]$ , в котором хранятся концы ребер;
- списки пучков ребер, расположенные в двух массивах:  
 $H[0..n-1]$  – начала списков;  
 $L[0..2m-1]$  – ссылки.

Вспомним, что просмотр ребер, выходящих из вершины, выполняется неоднократно и каждый раз начинается с первого непросмотренного ребра. Следовательно, для каждой вершины нам потребуется дополнительно запоминать номер первого непросмотренного ребра. Придется выделить для этого дополнительный массив  $Hp[0..n-1]$ , где  $Hp[i]$  – номер первого непросмотренного ребра, выходящего из вершины  $i$ . Вначале  $Hp=H$ . Можно было бы не выделять новый массив  $Hp$ , а по ходу алгоритма «портить» содержимое массива  $H$ , но нехорошо оставлять после себя испорченные списки – они могут еще пригодиться.

При обходе нам понадобится помечать вершины. Кроме того, где-то надо хранить номер компоненты связности, в которую попала данная вершина. Совместим эти функции. В массиве  $K[0..n-1]$  в переменной  $K[i]$  будем хранить  $-1$ , если вершина  $i$  не помечена, а если помечена, то – номер компоненты связности, в которую эта вершина попала. Компоненты связности будем нумеровать, начиная с нуля.

Вспомним еще про стек активных вершин, который используется при шагах вперед и назад. Для его хранения выделим  $S[0..n-1]$ . Переменная  $w$  будет указывать на первую свободную позицию в стеке.

Итак, структура данных готова (рис. 3.3).

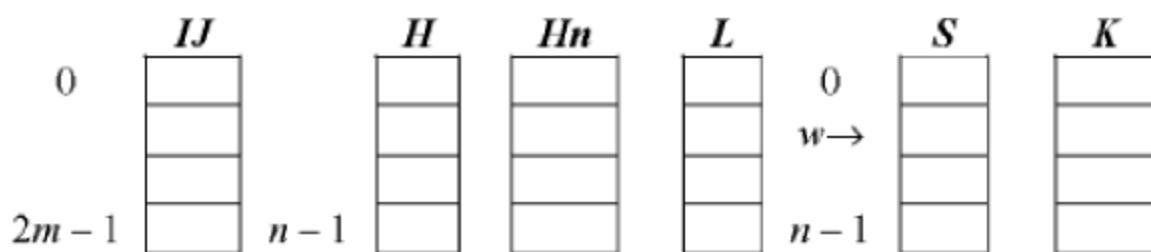


Рис. 3.3

Осталось написать программу, но теперь это сделать несложно.

```
//ПРОГРАММА НАХОЖДЕНИЯ КОМПОНЕНТ СВЯЗНОСТИ  
//НЕОРИЕНТИРОВАННОГО ГРАФА  
  
//Инициализация  
//Построение списков ребер H и L (раздел 2)  
· · · · · · · · · · · · · · · · · ·  
//Списки построены  
K[0..n-1] = -1; //Все вершины не помечены  
  
//Первое непросмотренное ребро - в начале списка  
Hn[0..n-1] = H[0..n-1];  
w = 0; //Стек пуст  
x = -1; //Номер компоненты связности  
  
  
for( i0 = 0; i0 < n; i0++ ) //Цикл по вершинам  
{  
    if( K[i0] != -1) continue;  
    //Если вершина уже принадлежит какой-то  
    //компоненте связности, то ничего делать не  
    //надо. А если не принадлежит (K[i0] == -1), то  
    //начинаем обход компоненты связности с вершины  
    //i0.  
  
    x++; //Номер текущей компоненты связности  
    i = i0; // i - текущая вершина  
  
    While(1) //Пока не попробуем сделать шаг  
        //назад при пустом стеке
```

```

{   K[i] = x; //Помечаем текущую вершину,
    //присваивая ей номер
    // компоненты связности

//ПРОСМОТР РЕБЕР, ТЕКУЩЕЙ ВЕРШИНЫ, НАЧИНАЯ С
//ПЕРВОГО НЕПРОСМОТРЕННОГО
    for( k = Hn[i]; k != -1; k = L[k] )
    {
        j = IJ[2*m-1-k]; //Вершина на другом
                           //конце ребра
        if( K[j] == -1 ) break;
        //Если нашли непомеченную вершину -
        //выходим из цикла
    }
    // Выход из цикла либо оператором "break" -
    // нашли непомеченную вершину и надо делать
    // шаг вперед, либо после исчерпания списка
    // ребер - просмотр ребер закончен и надо
    // делать шаг назад

    if( k != -1 )
    { //ШАГ ВПЕРЕД
        Hn[i] = L[k]; //Запоминаем первую
                       //непросмотренную дугу
                       //в вершине i
        S[w] = i; //Записываем вершину i в стек
        w++;      //Перемещаем указатель стека
        i = j;    //Теперь текущей вершиной
                  //будет j
        //Сделали шаг вперед
    }
    else
    { // ШАГ НАЗАД
        if(w == 0) break;
        //Стек пуст. Обошли всю компоненту.
        //Выходим из цикла while(1)
    }
    else
    { //Стек не пуст
        w--; //Перемещаем указатель стека
        i = S[w]; //Берем из стека
    }
}

```

```
                //последнюю вершину
            }
        //Сделали шаг назад
    }
    //После шага вперед или назад переходим к
    //просмотру дуг из новой текущей вершины i
} //Конец цикла while(1)

} //Конец цикла по вершинам,
//а с ним и всей программы
```

Как видите, программа простая и короткая. Она заняла две страницы лишь потому, что большую ее часть занимают комментарии. Мы не испортили алгоритм этой реализацией, хотя можно было бы в ущерб наглядности написать программу более компактно. Если записывать в стек не вершины, а ребра, то можно обойтись без дополнительного массива Нр. Желающие могут выполнить это упражнение самостоятельно.

## 4. ЗАДАЧА ПОСТРОЕНИЯ КРАТЧАЙШЕГО СВЯЗЫВАЮЩЕГО ДЕРЕВА

### 4.1. Основные понятия

Напомним, что *деревом* называется связный граф без циклов. Дерево, состоящее из  $n$  вершин, содержит  $n - 1$  ребро.

Пусть задан неориентированный связный граф  $G = \{N, E, c\}$ , где  $N$  – множество вершин графа;  $E$  – множество ребер графа;  $c$  – весовая функция, заданная на ребрах:  $c(e)$ ,  $e \in E$  – вес (длина) ребра.

Дерево  $T$  называется *связывающим* или *остовным*, если выполняются следующие условия:

$N(T) = N(G)$  – множества вершин графа и связывающего дерева совпадают;

$E(T) \subset E(G)$  – связывающее дерево содержит только ребра исходного графа.

Двигаясь по ребрам связывающего дерева, из любой вершины графа можно попасть в любую другую.

*Длиной дерева* называется  $C(T) = \sum_{e \in E} c(e)$  – сумма длин входящих в дерево ребер.

*Минимальным связывающим деревом* называется дерево, длина которого минимальна среди всех связывающих.

Нам потребуется одно простое свойство связывающего дерева. Если к множеству ребер связывающего дерева добавить какое-нибудь ребро графа, то это ребро вместе с частью ребер дерева образуют замыкающий цикл. Если удалить любое ребро замыкающего цикла, то вновь образуется связывающее дерево. Справедливость этого утверждения непосредственно следует из приведенных выше определений и проиллюстрирована на рис. 4.1.

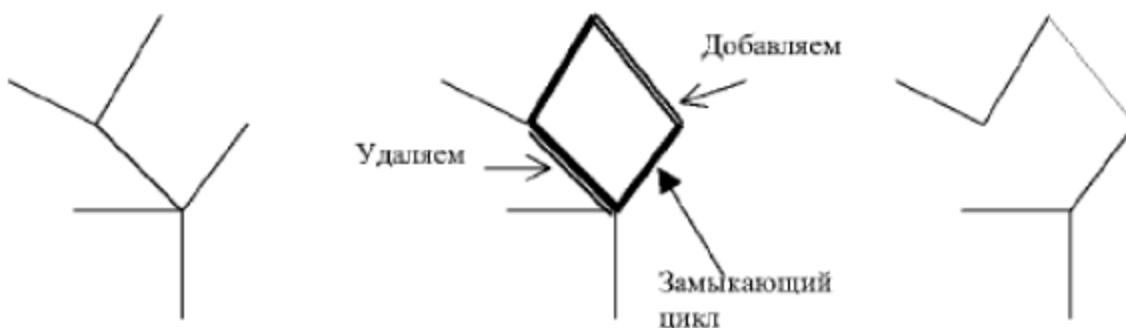


Рис. 4.1

## 4.2. Алгоритм Краскала

В 50-е годы был предложен алгоритм построения кратчайшего связывающего дерева, называемый *алгоритмом Краскала*.

Алгоритм состоит из двух этапов.

### 1. Упорядочение ребер по возрастанию длины.

В результате получаем последовательность ребер  $e_1, e_2, \dots, e_k, \dots, e_m$ , в которой для любого  $k$   $c(e_k) \leq c(e_{k+1})$ .

### 2. Выбор ребер для включения в кратчайшее связывающее дерево.

Начинаем с графа  $K$ , множество вершин которого совпадает с множеством вершин графа  $G$ , а множество ребер пусто. Просматриваем ребра графа  $G$  в порядке возрастания длины. На  $k$ -м шаге рассматриваем ребро  $e_k$ . Если оно не образует цикл с ранее выбранными ребрами, то включаем его в граф  $K$ . Если же такой цикл существует, то ребро  $e_k$  в граф  $K$  не включаем и переходим к просмотру следующего ребра  $e_{k+1}$ .

Работа алгоритма заканчивается после просмотра всех ребер исходного графа  $G$ . Утверждается, что граф  $K$  является кратчайшим связывающим деревом.

## 4.3. Обоснование алгоритма

Докажем сначала, что после завершения работы алгоритма Краскала будет построено связывающее дерево.

Предположим противное: пусть граф  $K$  не связан. Тогда существуют хотя бы две компоненты связности  $N_1$  и  $N_2$ , между которыми нет соединяющего ребра графа  $K$ . Но поскольку исходный граф  $G$  связный, то существует ребро  $e$  графа  $G$ , соединяющее  $N_1$  и  $N_2$ .

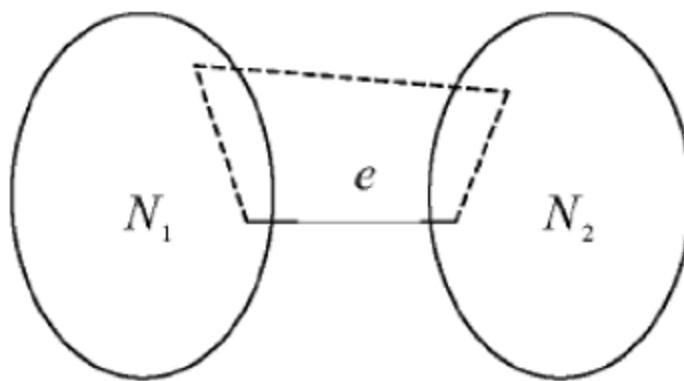


Рис. 4.2

Ребро  $e$  не образует циклов с ребрами графа  $K$ , потому что никакие ребра этого графа не соединяют вершины из множеств  $N_1$  и  $N_2$ , а цикл, содержащий  $e$ , должен был бы перейти из одного множества в другое по крайней мере дважды (рис. 4.2). Но в этом случае, когда при работе алгоритма Краскала мы просматривали ребро  $e$ , то должны были включить его в граф  $K$ , поскольку оно не образует циклов с остальными ребрами этого графа. По предположению, ребро  $e$  не содержится в графе  $K$ , и, следовательно, мы приходим к противоречию.

Итак, ребра, включенные в множество  $K$ , образуют связный граф. Из описания алгоритма следует, что в этом графе нет циклов, и, следовательно, он является связывающим деревом.

Теперь докажем, что построенное связывающее дерево – кратчайшее. Предположим, что настоящее кратчайшее связывающее дерево меньше по весу дерева  $K$ . Выберем среди кратчайших деревьев «максимально похожее» на дерево  $K$ , построенное с помощью алгоритма Краскала.

Обозначим через  $K = (k_1, k_2, \dots, k_l, k_{l+1}, \dots, k_{n-1})$  дерево, построенное с помощью алгоритма Краскала. Ребра дерева  $K$  упорядочены по возрастанию длины.

Пусть  $T = (t_1, t_2, \dots, t_l, t_{l+1}, \dots, t_{n-1})$  – некоторое кратчайшее связывающее дерево, ребра которого также упорядочены по возрастанию длины.

Сходство деревьев  $T$  и  $K$  определяется длиной отрезка совпадающих ребер:  $t_1 = k_1, t_2 = k_2, \dots, t_l = k_l, t_{l+1} \neq k_{l+1}$ . Для самого похожего на  $K$  дерева длина отрезка  $l$ , на котором деревья  $T$  и  $K$  совпадают, максимальна среди всех кратчайших связывающих деревьев. Ясно, что  $l < n - 1$ , поскольку иначе  $K = T$  и  $K$  тоже было бы кратчайшим связывающим деревом.

Докажем, что  $c(t_{l+1}) \geq c(k_{l+1})$ . Если бы первое ребро дерева  $T$ , отличающееся от соответствующего ребра дерева  $K$ , было бы короче него:  $c(t_{l+1}) < c(k_{l+1})$ , то тогда по ходу работы алгоритма Краскала

мы посмотрели бы на ребро  $t_{l+1}$  раньше, чем на ребро  $k_{l+1}$ . Ребро  $t_{l+1}$  не образует цикла с ребрами  $t_1, t_2, \dots, t_l$ , потому что входит вместе с ними в дерево  $T$ . Значит,  $t_{l+1}$  не образует цикл с ребрами  $k_1, k_2, \dots, k_l$ , поскольку  $t_1 = k_1, t_2 = k_2, \dots, t_l = k_l$ . Тогда при просмотре ребра  $t_{l+1}$  мы должны были включить его в дерево  $K$  раньше ребра  $k_{l+1}$ . Но поскольку мы этого не сделали, то предположение, что  $c(t_{l+1}) < c(k_{l+1})$ , неверно, и  $c(t_{l+1}) \geq c(k_{l+1})$ .

Добавим к дереву  $T$  ребро  $k_{l+1}$ . Это ребро вместе с некоторыми ребрами дерева  $T$  образует замыкающий цикл. Этот цикл не может содержать только ребра  $t_1 = k_1, t_2 = k_2, \dots, t_l = k_l$ , потому что ребро  $k_{l+1}$  с ними цикл не образует. Значит, на замыкающем цикле найдется ребро  $t_x$  такое, что  $x \geq l+1$ . Удалим это ребро и получим новое дерево  $\bar{T} = T + k_{l+1} - t_x$ . Но  $x \geq l+1$ , поэтому  $c(t_{l+1}) \leq c(t_x)$ . Ранее мы доказали, что  $c(k_{l+1}) \leq c(t_{l+1})$ , следовательно,  $c(k_{l+1}) \leq c(t_x)$ . Отсюда следует, что  $C(\bar{T}) = C(T) + c(k_{l+1}) - c(t_x) \leq C(T)$ . Но ведь  $T$  было кратчайшим связывающим деревом. Значит,  $C(\bar{T})$  не может быть меньше  $C(T)$ , то есть  $C(\bar{T}) = C(T)$  и  $\bar{T}$  – тоже кратчайшее связывающее дерево. Однако дерево  $\bar{T}$  содержит ребро  $k_{l+1}$  и, следовательно, деревья  $K$  и  $\bar{T}$  совпадают на отрезке длины  $l+1$ , что противоречит предположению о том, что дерево  $T$  было наиболее похожим на  $K$  кратчайшим связывающим деревом.

Следовательно, исходное предположение неверно и дерево  $K$ , построенное с помощью алгоритма Краскала, является кратчайшим связывающим деревом. Корректность алгоритма Краскала доказана.

#### 4.4. Задача Объединить – Найти

Чтобы реализовать алгоритм Краскала, надо научиться проверять, образует ли рассматриваемое ребро цикл с ребрами, выбранными ранее.

В каждый момент работы алгоритма граф  $K$  является лесом – набором поддеревьев. В частности, таким поддеревом может быть изолированная вершина. Работа алгоритма заканчивается, когда все поддеревья объединятся в одно дерево, которое и будет кратчайшим связывающим.

Очевидно, что просматриваемое ребро образует цикл с ребрами, выбранными ранее, если оба конца этого ребра находятся в одном поддереве. Следовательно, для того чтобы проверить, образует ли ребро цикл, достаточно узнать, в одном или в разных поддеревьях находятся концы просматриваемого ребра.

Таким образом, наша задача сводится к классической задаче *Union – Find (Объединить – Найти)*.

Постановка задачи Объединить – Найти такова:

Задано множество  $M$  и разбиение его на непересекающиеся подмножества  $M_1, \dots, M_t : M = \bigcup_{i=1}^t M_i$  и  $\forall k, l \quad M_k \cap M_l = \emptyset$ .

Необходимо уметь выполнять произвольную последовательность операций *Найти (Find)* и *Объединить (Union)*.

**Find (i)** – для заданного элемента  $i$  найти подмножество  $M(i)$ , содержащее этот элемент.

**Union ( $M_k, M_l$ )** – объединить подмножества  $M_k$  и  $M_l$ .

При реализации алгоритма Краскала  $M$  – это множество вершин исходного графа, а подмножества – вершины связных поддеревьев, образованные ребрами, включенными в граф  $K$ .

Для того чтобы выяснить, образует ли рассматриваемое ребро  $(i, j)$  цикл с ранее выбранными ребрами, достаточно найти, в каких подмножествах содержатся концы этого ребра, то есть выполнить две операции *Найти*:  $M_i = \text{Find}(i)$  и  $M_j = \text{Find}(j)$ . Если  $M_i = M_j$ , то это значит, что вершины  $i$  и  $j$  находятся в одном и том же поддереве, ребро  $(i, j)$  образует цикл с ребрами этого поддерева, и, следовательно, его не следует включать в граф  $K$ . Если же  $M_i \neq M_j$ , то вершины  $i$  и  $j$  находятся в разных поддеревьях, ребро  $(i, j)$  не образует цикл с ребрами, ранее включенными в дерево, и, следовательно, должно быть включено в граф  $K$ . После добавления этого ребра поддеревья  $M_i$  и  $M_j$  сливаются в одно связное поддерево, а значит, надо объединить подмножества  $M_i$  и  $M_j$ , для чего необходимо выполнить операцию *Объединить*:  $\text{Union}(M_i, M_j)$ . После этого можно переходить к просмотру следующего ребра.

Таким образом, для реализации второго этапа алгоритма Краскала нам достаточно научиться эффективно решать задачу *Объединить – Найти*.

Сложность первого этапа алгоритма Краскала равна сложности сортировки массива и составляет  $O(m \log m)$ , где  $m$  – число ребер графа. Если обозначить через  $F$  трудоемкость операции *Найти*, а через  $U$  – трудоемкость операции *Объединить*, то сложность второго этапа алгоритма Краскала оценивается так:  $O(mF + nU)$ . Это является следствием того, что нам приходится просматривать не более чем  $m$  ребер, а при просмотре одного ребра выполняется две операции *Найти*. Операций *Объединить* всего будет  $(n-1)$ , так как количество подмножеств сначала было равно  $n$ , а при каждом объединении оно уменьшается на единицу, пока все подмножества не объединятся в одно связное дерево.

Таким образом, сложность второго этапа алгоритма Краскала полностью определяется сложностью реализации алгоритма решения задачи *Объединить – Найти*.

#### **4.5. Наивный способ решения задачи Объединить – Найти**

Рассмотрим сначала *наивный способ* решения задачи *Объединить – Найти*.

В массиве  $M[0..n - 1]$  будем хранить на  $i$ -м месте номер подмножества, в которое входит вершина  $i$ . Используя такие данные, выполнить операцию *Найти* очень легко:

**Find(i) = M[i]**, и трудоемкость этой операции равна  $O(1)$ . В начале работы алгоритма каждая из вершин образует отдельное подмножество, поэтому сначала  $M[i] = i$  для всех вершин графа.

Для того чтобы объединить подмножества с номерами  $l$  и  $k$  (новому подмножеству присваивается номер  $k$ ), достаточно просмотреть массив  $M$  и каждый раз, когда мы видим число  $l$ , заменять его на  $k$ . Трудоемкость операции *Объединить*, таким образом, составляет  $O(n)$ , где  $n$  – число вершин графа. Программы, реализующие наивный способ решения задачи *Объединить – Найти*, так просты, что писать их мы не будем. Сложность второго этапа алгоритма Краскала при таком способе решения задачи *Объединить – Найти* составляет  $O(mF + nU) = O(m + n^2) = O(n^2)$ , где  $n$  – число вершин,  $m$  –

число ребер графа, поскольку  $n^2 \geq m$ . Суммируя эту оценку с оценкой сложности первого этапа  $O(m \log m)$ , получаем общую оценку сложности алгоритма Краскала при наивном способе решения задачи Объединить – Найти:  $O(m \log m + n^2)$ .

#### 4.6. Реализация алгоритма Краскала

Теперь напишем программу, реализующую алгоритм Краскала, предположив, что сортировку массива и операции *Объединить*, *Найти* выполняют внешние функции.

Пусть **Sort(I, J, C)** является функцией, упорядочивающей ребра по возрастанию длины, **Find(i)** – функцией, реализующей операцию *Найти*, а **Union(k, l)** – функцией, реализующей операцию *Объединить*.

В массиве **I** будем хранить начало ребра, в массиве **J** – конец ребра, а в массиве **C** – его длину. Размерность этих массивов:  $[0..m-1]$ .

Номера включенных в кратчайшее связывающее дерево ребер будем запоминать в массиве **K**  $[0..n-2]$ . Переменная **w** указывает на позицию, куда надо заносить номер очередного выбранного ребра (рис. 4.3).

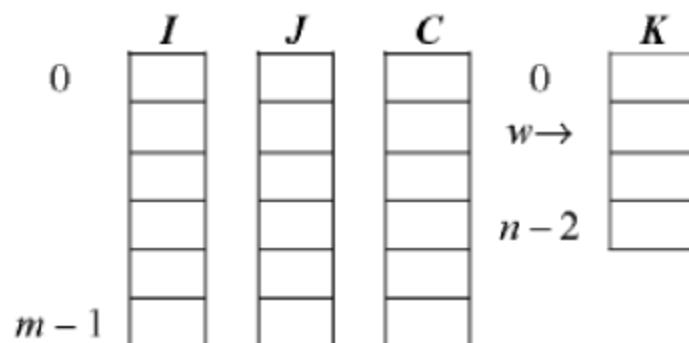


Рис. 4.3

```
//ПОСТРОЕНИЕ КРАТЧАЙШЕГО СВЯЗЫВАЮЩЕГО ДЕРЕВА
//АЛГОРИТМ КРАСКАЛА
//
Sort(I, J, C); // Сортируем дуги по возрастанию
                    // длины
w = 0; //Указатель записи устанавливаем на первую
        //свободную позицию в массиве K
```

```

//Просмотр ребер графа по возрастанию длины
for( k=0; k<m && w<n-1; k++ )
{
    //Заканчиваем просмотр после исчерпания дуг
    //графа или когда полностью сформируется
    //кратчайшее связывающее дерево

    i = I[k]; //Начало дуги k
    j = J[k]; //Конец дуги k
    mi = Find(i); //Номера подмножеств, в которых
    mj = Find(j); //находятся элементы i и j.
    if(mi != mj)// i и j находятся в разных
                  // подмножествах
    {
        K[w] = k; //Добавляем ребро k в граф K
        w++;      //Передвигаем указатель
        Union(mi, mj); //Объединяем множества mi, mj
    }
    //Если концы ребра k находятся в разных
    //подмножествах, то ничего делать не надо

} // Конец основного цикла
// Кратчайшее связывающее дерево построено

```

#### **4.7. Логарифмический алгоритм решения задачи Объединить – Найти**

Наивный способ решения задачи Объединить – Найти производит впечатление неэффективного. Напрашиваются два усовершенствования, которые могли бы улучшить этот алгоритм.

Вспомним, что при объединении множеств  $M_k$  и  $M_l$  в массиве M надо заменить все числа  $l$  на числа  $k$ .

Во-первых, хорошо бы просматривать для этого не весь массив M, а проходить только те позиции, где находятся числа  $l$ .

Во-вторых, если мы научимся проходить только нужные нам позиции в массиве M, то лучше проходить меньшее из подмножеств  $M_k$  и  $M_l$ , поскольку для нас не имеет значения, какой номер получит объединенное подмножество.

Если операция *Объединить* удовлетворяет обоим сформулированным требованиям, а число вспомогательных операций не слиш-

ком велико, то трудоемкость этой операции равна числу элементов в меньшем из подмножеств  $M_k$  и  $M_l$ , или, иначе говоря, числу элементов, которые попадут в новое подмножество. Будем называть такие элементы «перекрашиваемыми» – попав в новое подмножество, они как будто сменили окраску.

Оказывается, в этом случае интуиция нас не подвела и модернизированная операция *Объединить* становится лучше не только с точки зрения здравого смысла, но и по оценке трудоемкости. Трудоемкость одной операции *Объединить* может, как и раньше, быть порядка  $O(n)$  действий, но  $(n-1)$  операция потребует не  $O(n^2)$ , а  $O(n \log n)$  действий. Докажем это утверждение.

При сделанных предположениях суммарная трудоемкость всех операций *Объединить* равна общему числу перекрашиваемых элементов. Докажем, что оно не превышает  $n \log n$ .

Докажем, что один элемент не может быть перекрашен более чем  $\log n$  раз. Обозначим через  $x_k$  размер множества, в котором он находится после  $k$ -й перекраски.

Сначала  $x_0 = 1$  – данный элемент находился в множестве, состоящем из него самого.

Очевидно, что  $x_k \geq 2x_{k-1}$ , потому что при объединении двух множеств мы перекрашиваем меньшее, оно вливается в большее, и суммарный размер объединенного подмножества по крайней мере в два раза превышает размер меньшего из объединяемых множеств. Следовательно,  $x_k \geq 2^k$ . Размер всего множества равен  $n$ . Поскольку размер подмножества не может быть больше размера всего множества, то для любого  $k$   $x_k \leq n$  или, иначе,  $2^k \leq n$ . Отсюда  $k \leq \log n$  и, следовательно, каждый элемент не может быть перекрашен более чем  $\log n$  раз. Значит, всего перекрасок будет не больше чем  $n \log n$ , что и требовалось доказать.

При логарифмическом алгоритме решения задачи *Объединить – Найти сложность второго этапа алгоритма Краскала* будет равна  $O(m + n \log n)$ , что не превышает сложности первого этапа. Таким образом, общая оценка сложности алгоритма Краскала составит  $O(m \log m)$ .

## 4.8. Реализация логарифмического алгоритма

Реализовать логарифмический вариант решения задачи Объединить – Найти нетрудно. Как это часто бывает, достаточно уметь грамотно пользоваться списками. Представим каждое подмножество в виде списка, что даст нам возможность проходить не весь массив  $M$ , а только нужное подмножество. А для того чтобы проходить меньшее из двух подмножеств, для каждого подмножества будем дополнительно хранить его размер.

Нам понадобятся дополнительно три массива:

$H[0..n - 1]$  :  $H[k]$  – номер первой вершины, входящей в подмножество  $k$ ;

$L[0..n - 1]$  :  $L[i]$  – следующая вершина, входящая в то же подмножество, что и вершина  $i$ ;

$X[0..n - 1]$  :  $X[k]$  – число вершин в  $k$ -м подмножестве.

```
//Инициализация данных в задаче Объединить–  
Найти  
for( k = 0, i = 0; k < n; k++, i++ )  
{ // k – индекс подмножества  
  // i – индекс вершины  
  // конечно же, k=i, а разные индексы для вершины  
  // и подмножества мы завели «для порядка»  
  H[k] = i;  
  L[i] = -1;  
  X[k] = 1;  
  M[i] = k;  
}
```

После инициализации данные в массивах соответствуют одноделементным подмножествам.

```
//Функция Найти  
Find ( i )  
{ return M[i]; }  
  
//Функция Объединить  
Union(mi, mj)  
{  
  if (X[mi] < X[mj]) // выбираем меньшее из двух
```

```

        // подмножество: k - номер
        // меньшего, l - номер большего
    {k = mi; l = mj; }
    else
    {k = mj; l = mi; }

    // обход меньшего подмножества
    for( i=H[k]; L[i]!=-1; i=L[i] )
        M[i] = 1; // перекрашиваем вершины

    M[i] = 1; // перекрашиваем последнюю вершину
    L[i] = H[l]; // к списку k прицепляем список l
    H[l] = H[k]; // переносим начало списка
    X[l] += X[k]; // размер нового подмножества
}

```

Если таким способом реализовать алгоритм Краскала и на первом этапе использовать не слишком плохую процедуру сортировки, то эта программа окажется достаточно быстрой и будет хорошо строить кратчайшее связывающее дерево, если только нет необходимости в решении задач «сверхбольших» размеров. Существуют и другие алгоритмы построения кратчайшего связывающего дерева. Одним из таких алгоритмов является хорошо известный алгоритм Прима, который тоже можно заставить работать эффективно, а в некоторых случаях применять его даже удобней, чем алгоритм Краскала. В реализации алгоритма Прима многое поучительного, но, к сожалению, времени на его изучение у нас нет.

Вообще говоря, существуют современные, теоретически гораздо более эффективные алгоритмы построения кратчайшего связывающего дерева, оценка сложности которых очень близка к линейной. Однако целесообразность их применения на практике вызывает сомнения.

Что касается задачи Объединить – Найти, то она используется в качестве вспомогательной в целом ряде алгоритмов. Рассмотренный нами логарифмический способ решения этой задачи – вполне приемлем для практического применения, но существует еще более эффективный и очень красивый алгоритм решения этой задачи, который называется *алгоритмом сжатия путей*.

Как мы с вами убедились, задача построения кратчайшего связывающего дерева решается эффективно. Интересно, что близкая к ней

задача Штейнера оказывается сложнее на много порядков. Задача Штейнера очень похожа на задачу о построении кратчайшего связывающего дерева, но в ней надо соединить кратчайшим деревом не все вершины графа, а некоторое их подмножество. Так вот, задача Штейнера относится к классу так называемых  $NP$ -трудных. К сожалению, теория  $NP$ -сложности не входит в наш курс, но про  $NP$ -трудные задачи можно сказать, что, хотя этот факт до сих пор еще не доказан, крайне маловероятно существование полиномиального алгоритма их решения. Таким образом, с весьма высокой долей вероятности можно утверждать, что решение задачи Штейнера может быть получено только при помощи экспоненциального алгоритма.

## 5. АЛГОРИТМ ПОСТРОЕНИЯ СТАБИЛЬНОГО БРАКОСОЧЕТАНИЯ

### 5.1. Понятие стабильного бракосочетания

Дано  $n$  мужчин:  $m_1, m_2, \dots, m_n$  и  $n$  женщин:  $w_1, w_2, \dots, w_n$ . У каждого мужчины есть список всех женщин, упорядоченных в соответствии с его вкусом. Такой же список есть у каждой женщины.

Если для мужчины  $m$  женщина  $w_i$  предпочтительней женщины  $w_j$ , то мы будем это записывать так:  $w_i > w_j$ . Для какого именно мужчины женщина  $w_i$  лучше  $w_j$ , будет ясно из контекста. Аналогично, если для женщины  $w$  мужчина  $m_k$  лучше мужчины  $m_l$ , то запишем это так:  $m_k > m_l$ . То есть будем использовать знак « $>$ » для обозначения предпочтений.

В списке предпочтений мужчины  $m_i (w_{i1}, w_{i2}, \dots, w_{in})$  женщины упорядочены по убыванию их качества с точки зрения этого мужчины:  $w_{i1} > w_{i2} > \dots > w_{in}$ . Аналогично  $m_{i1} > m_{i2} > \dots > m_{in}$  в списке предпочтений женщины  $w_i$ .

*Бракосочетанием* называется набор из  $n$  пар, каждая из которых состоит из мужчины и женщины, причем каждый входит ровно в одну пару. Мужчину и женщину, входящих в одну пару, будем называть мужем и женой, или супругами.

Обозначим через  $w(m, Q)$  жену мужчины  $m$  и, соответственно, через  $m(w, Q)$  мужа женщины  $w$  в бракосочетании  $Q$ . На диаграммах мы часто будем соединять мужа и жену стрелкой: .

Мужчина и женщина  $(m, w)$ , не являющиеся мужем и женой, называются *парой нестабильности* в бракосочетании  $Q$ , если одновременно выполняются два условия:

для  $w \quad m > m(w, Q)$ , а для  $m \quad w > w(m, Q)$ .

Иными словами, для мужчины женщина  $W$ , образующая с ним пару нестабильности, предпочтительнее чем его жена в бракосочетании  $Q$ , а для женщины мужчина  $m$  лучше ее мужа. Понятно, что в жизни такое может закончиться двумя разводами и новым браком, потому-то  $(m, w)$  и названо парой нестабильности.

Бракосочетание называется *стабильным*, если в нем нет ни одной пары нестабильности.

Рассмотрим примеры нестабильного и стабильного бракосочетаний. Обозначим через  $a, b, c$  – мужчин, а через  $A, B, C$  – женщин.

Списки предпочтения мужчин      Списки предпочтения женщин

$a (A, B, C)$	$A (b, a, c)$
$b (A, B, C)$	$B (a, c, b)$
$c (B, C, A)$	$C (a, b, c)$

Бракосочетание  $Q_1 = (aC, bB, cA)$  не является стабильным. Рассмотрим пару  $bA$ . Для мужчины  $b$  женщина  $A$  лучше его жены  $B$ , а для женщины  $A$  мужчина  $b$  лучше ее мужа  $c$ . Следовательно,  $bA$  – пара нестабильности, и бракосочетание  $Q_1$  не является стабильным.

Докажем, что бракосочетание  $Q_2 = (aB, bA, cC)$  стабильно. Женщины  $A, B$  и мужчина  $b$  не могут входить в пару нестабильности, так как их супругами являются наилучшие с точки зрения их предпочтений партнеры. Значит, потенциально возможной парой нестабильности могла бы быть только пара  $aC$ . Но с точки зрения  $a$  его жена  $B$  лучше  $C$ . То есть эта пара тоже не является парой нестабильности, и, следовательно, бракосочетание  $Q_2$  стабильно.

Приведем алгоритм, который называется *Алгоритмом предложений*, при помощи которого можно построить стабильное бракосочетание для произвольных списков предпочтений.

## 5.2. Алгоритм предложений

### 5.2.1. Общая схема алгоритма

Алгоритм состоит из последовательности предложений, которые делают женщинам неженатые мужчины.

Произвольный неженатый мужчина делает предложение выйти за него замуж некоторой женщине в соответствии с *мужской стратегией*.

Если женщина, следуя *женской стратегии*, соглашается, то они женятся. Если женщина отказывается, то брак не заключается и любой неженатый мужчина делает следующее предложение.

Работа алгоритма заканчивается после того, как не осталось ни одного неженатого мужчины.

Порядок выбора мужчин, делающих предложение, не регламентирован.

Женатые мужчины никому предложений не делают. Разводы происходят только по инициативе женщины.

Как это обычно и бывает в жизни, мужская стратегия очень проста.

### **5.2.2. Мужская стратегия**

Мужчина всегда делает предложение самой лучшей женщине, не вычеркнутой из его списка предпочтений.

В начале работы алгоритма из списка предпочтений мужчины не вычеркнут никто.

Если мужчина делает женщине предложение, а она ему отказывает, то он вычеркивает ее из списка.

Если женщина разводится с мужчиной, то он тоже вычеркивает ее из своего списка.

Женская стратегия, как этого и следовало ожидать, гораздо более причудлива.

### **5.2.3. Женская стратегия**

Незамужняя женщина соглашается на любое предложение.

Замужняя женщина, получив предложение, сравнивает своего мужа с претендентом. Если претендент в соответствии с ее списком предпочтений хуже ее мужа, то она ему отказывает. Если же претендент лучше мужа, то она разводится с мужем и выходит замуж за того, кто сделал ей предложение.

После развода бывший муж получает статус неженатого.

## **5.3. Обоснование алгоритма**

По ходу работы алгоритма в каждый момент времени мы имеем текущее решение: часть мужчин и женщин состоят в браке, а часть не состоят. В процессе работы алгоритма мужчина может сменить нескольких жен, а женщина – нескольких мужей.

Отметим следующие свойства текущего решения.

### *1. Каждая следующая жена хуже предыдущей.*

Это напрямую следует из того, что мужчина делает предложения женщинам в соответствии со своим списком предпочтений, в порядке убывания их качества.

### *2. Один раз выйдя замуж, женщина остается замужней до конца работы алгоритма.*

Очевидное следствие женской стратегии.

### *3. Каждый следующий муж лучше предыдущего.*

В справедливости этого свойства легко убедиться, вспомнив, что женщина меняет мужа, только если ей сделал предложение более качественный претендент.

Докажем сначала, что после завершения работы алгоритма неженатых мужчин и незамужних женщин не останется и сформируется бракосочетание, состоящее из  $n$  пар.

Предположим, что остался неженатый мужчина  $m$ . Но это значит, что есть и незамужняя женщина  $w$ . Поскольку этому мужчине больше некому предлагать замужество, то из его списка все женщины, в том числе и  $w$ , вычеркнуты. Если она вычеркнута, то значит в какой-то момент он предлагал ей выйти за него замуж. Из второго свойства решения следует, что если она не замужем сейчас, то не могла быть замужем и тогда. Но по женской стратегии незамужняя женщина соглашается на любое предложение и, следовательно,  $w$  должна была выйти замуж за  $m$ . Однако, по тому же свойству, женщина, выйдя замуж, остается замужней навсегда, что противоречит предположению о том, что она сейчас не замужем. Следовательно, неженатых мужчин после окончания работы алгоритма оставаться не может, и будет сформировано бракосочетание из  $n$  супружеских пар.

Обозначим бракосочетание, полученное при помощи алгоритма предложений, через  $P$ . Докажем, что это бракосочетание является стабильным.

Возьмем произвольную пару  $(m, w)$ , не являющуюся супругами в бракосочетании  $P$ . Докажем, что эта пара не является парой нестабильности (рис. 5.1).

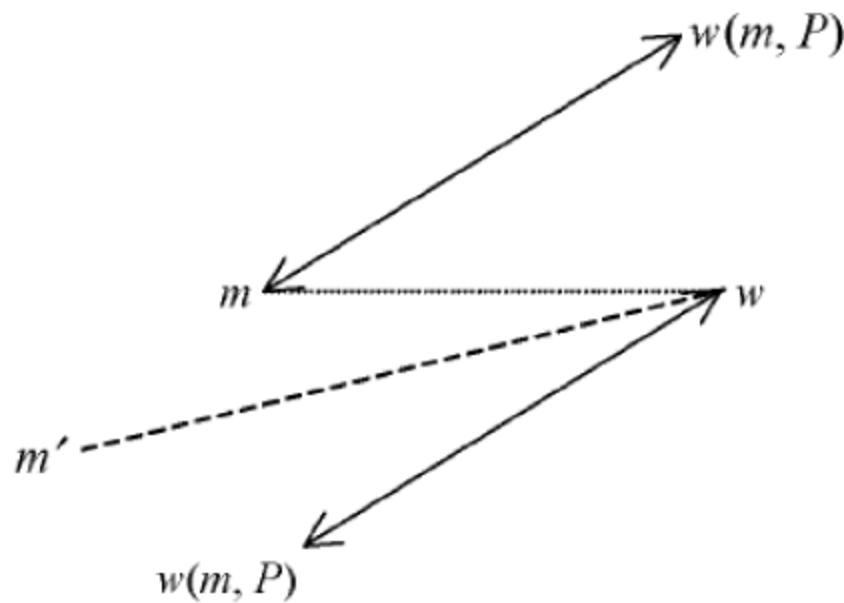


Рис. 5.1

Если для мужа  $m$  его жена  $w(m, P)$  лучше, чем  $w$ , то пара  $(m, w)$  не является парой нестабильности. Предположим, что  $w > w(m, P)$ . Но тогда  $w$  находится в списке предпочтений  $m$  раньше  $w(m, P)$ . Это значит, что до того как жениться на  $w(m, P)$ ,  $m$  делал предложение  $w$ . Поскольку они в бракосочетании  $P$  не женаты, то значит  $w$  либо отказалась ему сразу, потому что ее муж был лучше, либо развелась с ним потом, когда появился более качественный претендент. Обозначим через  $m'$  мужчину, из-за которого не состоялся брак  $m$  с  $w$  (из-за отказа или в результате развода). Ясно, что  $m' > m$  и что  $w$  после того, как отказалась  $m$ , была замужем за  $m'$ . В бракосочетании  $P$   $w$  замужем за  $w(m, P)$ . Это либо  $m'$ , либо кто-то другой, за кого  $w$  вышла замуж после того, как побывала замужем за  $m'$ . Но по третьему свойству текущего решения каждый следующий муж лучше предыдущего, поэтому  $w(m, P) \geq m' > m$ . Следовательно, для  $w$  ее муж лучше  $m$ , и пара  $(m, w)$  не является парой нестабильности. Поскольку пара  $(m, w)$  была выбрана произвольно, то из этого следует, что бракосочетание  $P$  стабильно.

**Свойство стабильного бракосочетания.** Стабильное бракосочетание существует при любых списках предпочтений.

Сложность алгоритма предложений определяется максимальным количеством предложений, которые могут быть сделаны в ходе его работы. Очевидно, что количество вспомогательных операций при подготовке и обработке одного предложения является константой:  $O(1)$ . В соответствии с мужской стратегией мужчина предлагает

женщине выйти за него замуж не более одного раза. Следовательно, общее число предложений не превышает  $n^2$ , а сложность алгоритма составляет  $O(n^2)$ .

#### 5.4. Свойства построенного стабильного бракосочетания

Стабильное бракосочетание  $P$ , построенное при помощи алгоритма предложений, обладает двумя интересными свойствами: оно является *оптимистичным для мужчин и пессимистичным для женщин*.

**Оптимистичность для мужчин.** Для любого мужчины  $m$  его жена в любом другом стабильном бракосочетании  $Q$  не лучше, чем его жена в бракосочетании  $P$ , построенном с помощью алгоритма предложений, то есть  $\forall m, Q \quad w(m, P) \geq w(m, Q)$ .

**Пессимистичность для женщин.** Для любой женщины  $w$  ее муж в бракосочетании  $P$ , построенном с помощью алгоритма предложений, не лучше, чем ее муж в любом другом стабильном бракосочетании  $Q$ , то есть  $\forall w, Q \quad m(w, P) \leq m(w, Q)$ .

Сначала докажем свойство оптимистичности для мужчин.

Предположим, что это свойство не выполняется и найдется такой мужчина  $m_1$  и такое стабильное бракосочетание  $Q$ , что его жена в этом бракосочетании лучше его жены в бракосочетании  $P$ :  $w(m_1, Q) > w(m_1, P)$ . Обозначим  $w(m_1, Q)$  через  $w_1$  (рис. 5.2).

Поскольку  $w_1 > w(m_1, P)$ , то из этого следует, что в процессе работы алгоритма предложений  $m_1$  до того, как жениться на  $w(m_1, P)$ , делал предложение  $w_1$ . В итоге в бракосочетании  $P$  брак  $m_1$  и  $w_1$  не состоялся из-за некоторого мужчины  $m_2$ , который в момент предложения  $m_1$  либо был мужем  $w_1$ , либо потом стал причиной их развода.

Поскольку  $w_1$  отказалась  $m_1$  из-за  $m_2$ , то для нее  $m_2 > m_1$ . Если бы жена  $m_2$  в бракосочетании  $Q$ :  $w_2 = w(m_2, Q)$  была бы с точки зрения  $m_2$  хуже  $w_1$ , то пара  $(m_2, w_1)$  была бы парой нестабильности для бракосочетания  $Q$ . Но бракосочетание  $Q$  стабильно, следовательно, для  $m_2$  его жена в бракосочетании  $Q$   $w_2 = w(m_2, Q)$  лучше  $w_1$ :  $w_2 > w_1$ . Так как  $w_1$  отказалась  $m_1$  из-за  $m_2$ , то в течение какого-то времени работы алгоритма предложений она была женой  $m_2$ . Поскольку каждая следующая жена в процессе работы алгоритма предло-

жений хуже предыдущей, то  $w_1 \geq w(m_2, P)$  и, следовательно,  $w(m_2, Q) = w_2 > w_1 \geq w(m_2, P)$ . То есть нашелся еще один мужчина  $m_2$ , чья жена  $w_2$  в бракосочетании  $Q$  лучше его жены в бракосочетании  $P$ .

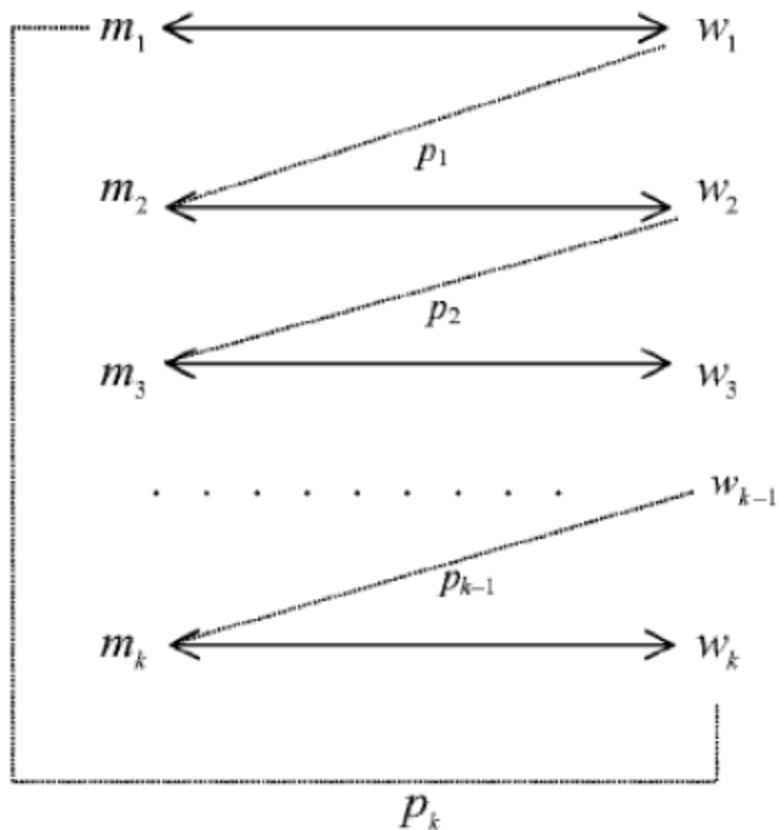


Рис. 5.2

Применяя к  $m_2$  аналогичные рассуждения, получаем, что найдется еще один мужчина  $m_3$ , из-за которого не состоялся брак  $m_2$  и  $w_2$  в бракосочетании  $P$ , чья жена  $w_3 = w(m_3, Q)$  в бракосочетании  $Q$  лучше его жены в бракосочетании  $P$ . И так далее. Однако число мужчин конечно, поэтому рано или поздно очередной мужчина  $m_{k+1}$ , из-за которого не состоялся брак  $m_k$  и  $w_k$  в бракосочетании  $P$ , чья жена  $w_{k+1} = w(m_{k+1}, Q)$  в бракосочетании  $Q$  лучше его жены в бракосочетании  $P$ , будет одним из мужчин, уже встречавшихся в этой последовательности. Без потери общности можно считать, что  $m_{k+1} = m_1$ .

Обозначим через  $p_1$  предложение, которое делал мужчина  $m_2$  женщины  $w_1$  в процессе работы алгоритма предложений, через  $p_2$  предложение  $m_3$   $w_2$ . Соответствующее предложение  $m_{i+1}$   $w_i$  будем

обозначать через  $p_i$ . Предложение  $p_k$  было сделано женщине  $w_k$  мужчиной  $m_{k+1} = m_1$ .

Утверждается, что для любого  $i$  в процессе работы алгоритма предложение  $p_i$  было сделано раньше предложения  $p_{i-1}$ . Действительно, поскольку для  $m_i \ w_i > w_{i-1}$ , то, пока  $w_i$  ему не откажет, он не будет делать предложения  $w_{i-1}$ . Но  $w_i$  отказывает ему из-за  $m_{i+1}$ . Следовательно, сначала  $m_{i+1}$  сделает  $w_i$  предложение  $p_i$ , и только потом  $m_i$ , отвергнутый  $w_i$ , будет делать  $w_{i-1}$  предложение  $p_{i-1}$ . Значит, при работе алгоритма предложений последовательность предложений была упорядочена во времени так:  $p_k \rightarrow p_{k-1} \rightarrow \dots \rightarrow p_2 \rightarrow p_1$ . Но поскольку схема «зацикlena» и  $m_1 = m_{k+1}$  находится с  $w_k$  и  $m_k$  в тех же отношениях, что и  $m_2$  с  $w_1$  и  $m_1$ , то, рассуждая аналогичным образом, можно убедиться, что предложение  $p_1$  было по времени сделано раньше, чем предложение  $p_k$ . То есть последовательность предложений во времени становится «зацикленной»:

$$\boxed{\overrightarrow{p_k \rightarrow p_{k-1} \rightarrow \dots \rightarrow p_2 \rightarrow p_1}}.$$

Поскольку никакое событие не может произойти раньше самого себя, то мы пришли к противоречию, и, следовательно, исходное предположение о том, что у какого-то мужчины в каком-то стабильном бракосочетании жена будет лучше, чем в бракосочетании  $P$ , неверно. Свойство оптимистичности для мужчин доказано.

Свойство пессимистичности для женщин доказывается гораздо проще. Предположим противное, что найдется такая женщина  $W$  и такое стабильное бракосочетание  $Q$ , что ее муж в этом бракосочетании  $m(w, Q)$  хуже ее мужа  $m = m(w, P)$  в бракосочетании  $P$ , построенного при помощи алгоритма предложений (рис. 5.3).

По предположению,  $m > m(w, Q)$  для  $w$ . Но из свойства оптимистичности для мужчин следует, что  $w = w(m, P) > w(m, Q)$  для  $m$ .

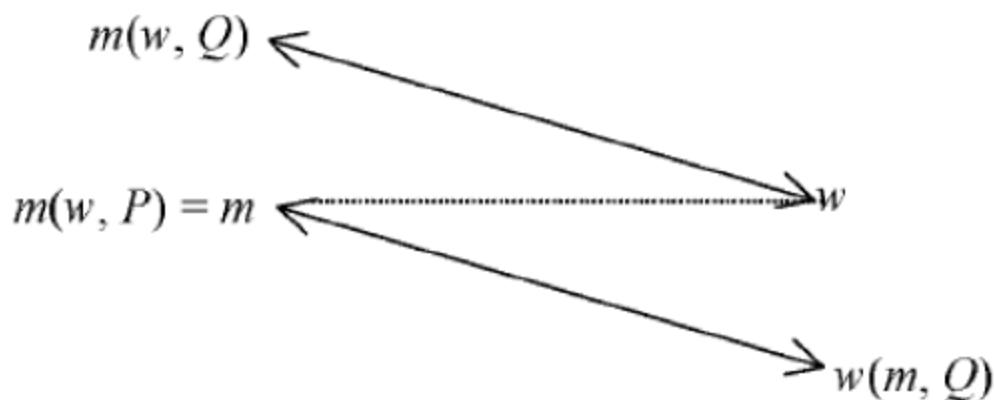


Рис. 5.3

Следовательно, пара  $(m, w)$  является парой нестабильности для бракосочетания  $Q$ , что противоречит предположению о его стабильности. Следовательно, исходное предположение неверно, и муж любой женщины  $w$  в любом стабильном бракосочетании  $Q$  не хуже ее мужа в бракосочетании  $P$ , построенном при помощи алгоритма предложений. Свойство пессимистичности для женщин доказано.

**Следствие из свойств построенного стабильного бракосочетания.** Независимо от порядка, в каком неженатые мужчины делают предложения, алгоритм предложений всегда строит одно и то же стабильное бракосочетание.

## 5.5. Мораль

Из свойств решения, построенного с помощью алгоритма предложений, можно сделать определенные житейские выводы. В ходе работы алгоритма мужчины активны, а женщины пассивно ждут предложений. По ходу алгоритма мужчины больше страдают: им отказывают, с ними разводятся, качество их жен ухудшается. Женщины, наоборот, в процессе работы алгоритма чувствуют себя комфортно: они все время замужем, а качество их мужей постоянно растет. Однако итоги работы алгоритма прямо противоположные: для мужчин оптимистичные, а для женщин пессимистичные. Мужчина получает лучшую жену из потенциально возможных, а женщина – худшего мужа. Мораль проста: большого успеха добивается тот, кто активен, хотя при этом ему приходится испытать множество неприятностей. Ну, а выводы пусть каждый делает для себя сам.

Конечно, все изложенное в этом разделе не более чем абстрактная модель. Реальное поведение мужчин и женщин в жизни сильно отличается от использованной примитивной схемы. Тем не менее выводы из свойств решения, получаемого с помощью алгоритма предложений, выглядят достаточно реалистично.

## 6. ЗАДАЧА ПОСТРОЕНИЯ КРАТЧАЙШИХ ПУТЕЙ

### 6.1. Описание задачи

Задан ориентированный граф  $G = \{N, A, c\}$ , где  $N$  – множество вершин графа,  $A$  – множество дуг,  $c$  – вещественная функция, заданная на дугах: для  $a \in A$   $c(a)$  – длина дуги.

Длиной пути  $W = (a_0, a_1, \dots, a_k, \dots, a_l)$  называется сумма длин входящих в него дуг:  $c(W) = \sum_{k=0}^l c(a_k)$ .

Обозначим через  $s(W)$  начальную вершину, а через  $t(W)$  – конечную вершину пути  $W$ . Если из контекста понятно, про какой путь идет речь, то будем обозначать эти вершины просто  $s$  и  $t$ .

Кратчайшим путем между вершинами  $s$  и  $t$  называется путь минимальной длины среди всех путей, соединяющих эти вершины. Задача построения кратчайших путей на сети является одной из самых известных сетевых оптимизационных задач. Эта задача широко используется на практике и как самостоятельная, и как вспомогательная при решении более сложных сетевых задач. Разработано много алгоритмов, позволяющих эффективно строить кратчайшие пути. Мы разберем некоторые из этих алгоритмов, может быть не самые эффективные теоретически, но вполне пригодные для использования на практике.

### 6.2. Классификация сетей

Сначала классифицируем задачи построения кратчайших путей. В зависимости от того, какой именно тип задачи нужно решать, следует применять тот или иной алгоритм. Первый признак, по которому мы будем классифицировать задачи – тип сети. Можно выделить следующие типы сетей, влияющие на выбор алгоритма решения задачи построения кратчайших путей.

1. Сеть с единичными длинами дуг.
2. Сеть с неотрицательными длинами дуг.
3. Сеть с произвольными длинами дуг. Отсутствие отрицательного цикла гарантировано.
4. Сеть с произвольными длинами дуг. Отсутствие отрицательного цикла не гарантировано.
5. Ациклическая сеть.

Обратите внимание, что для сетей 1–4 каждый следующий тип является обобщением предыдущего, то есть алгоритм, решающий задачу на сети типа 3, сумеет решить эту задачу и на сетях типа 1 и 2. Так, может быть, нам достаточно одного алгоритма для сетей типа 4, а применять мы его будем к сетям всех типов. Так, конечно, можно поступать, однако для более «узких» задач могут быть предложены алгоритмы проще и эффективней.

В стороне от общего ряда находится сеть пятого типа – ациклическая. Сеть называется *ациклической*, если в ней нет ориентированных циклов. Этот признак не имеет отношения к длинам дуг, а определяется топологией сети. Ациклические сети появляются в некоторых специальных задачах, и для них существуют особые, очень эффективные алгоритмы построения кратчайших путей.

Задача построения кратчайших путей на сети имеет смысл только в том случае, если в сети нет отрицательных циклов, то есть циклов, длина которых отрицательна. Если в сети есть такой цикл, то, двигаясь из вершины  $i$  в вершину  $j$ , мы доберемся до него и начнем двигаться вдоль этого цикла, «накручивая петли», отчего длина пути будет только уменьшаться. Само понятие кратчайшего пути в этих условиях становится бессмысленным. Поэтому строить кратчайшие пути можно только на сетях без отрицательных циклов. Отсюда следует, что при решении задачи типа 4, то есть задачи на сети, для которой априори не гарантировано отсутствие отрицательных циклов, можно придерживаться двух разных стратегий.

Можно сначала произвести «препроцессную обработку» сети, то есть запустить специальный алгоритм, который обследует сеть и ответит на вопрос, есть ли в ней отрицательные циклы. При ответе «да» дальнейшая работа прекращается, а при ответе «нет» можно воспользоваться алгоритмами, решающими задачу на сетях с гарантированным отсутствием отрицательных циклов.

Можно поступить по-другому: воспользоваться алгоритмом, который содержит специальные проверки, гарантирующие, что в процессе построения кратчайшего пути мы не натолкнемся на отрицательный цикл. К сожалению, такие проверки усложняют алгоритмы построения кратчайших расстояний и делают их менее эффективными.

Целесообразно, особенно если на сети надо решать не одну, а несколько задач построения кратчайших путей, применить первую стратегию и сначала убедиться, что отрицательных циклов нет.

Алгоритмы поиска отрицательных циклов, хотя и похожи на алгоритмы построения кратчайших путей, но все же обладают опреде-

ленной спецификой и в рамках данного пособия рассматриваться не будут. В следующих двух разделах мы разберем алгоритмы построения кратчайших путей на сетях первых трех типов. В этом разделе будет рассмотрен алгоритм построения кратчайших путей на сети с единичными длинами дуг – *метод поиска в ширину (breadth first search)*, а в двух последующих – алгоритм Беллмана – Форда для сетей с произвольными длинами дуг и алгоритм Дейкстры – для сетей с положительными длинами дуг.

Вообще говоря, про задачу построения кратчайших путей можно рассказать так много интересного, что это заняло бы по меньшей мере пару месяцев. Тем не менее, ограничимся перечисленным «джентльменским набором» алгоритмов.

### 6.3. Классификация задач

Кроме того, что приходится строить кратчайшие пути на сетях различного типа, сами по себе задачи тоже могут несколько различаться. Принято выделять три вида задач.

1. *От одной до одной.*
2. *От одной до всех.*
3. *От всех до всех.*

В первой задаче нужно построить кратчайший путь между заданной парой вершин. Во второй – построить кратчайшие пути от данной вершины до всех остальных вершин сети. А в третьей – построить кратчайшие пути между всеми парами вершин.

Очевидно, что обобщением всех трех задач является задача построения кратчайших путей от любой вершины подмножества  $M$  до любой вершины подмножества  $N$ , но почему-то в такой формулировке задачу построения кратчайших путей рассматривать не принято.

Мы будем разбирать алгоритмы решения второй задачи. Дело в том, что решение задачи 2 не намного сложнее решения задачи 1. Хотя формально задача 2 эквивалентна *n* задачам 1 (*n* – число вершин сети), но на самом деле задачи 1 и 2 решаются за примерно одинаковое время. До последнего времени алгоритмы построения кратчайших путей фактически не могли решить задачу 1, не решив почти полностью задачу 2. И только в последние годы положение начало меняться. Стали появляться алгоритмы, которые решают задачу 1 намного эффективней, чем задачу 2, но, к сожалению, их описание выходит за рамки данного курса лекций.

Задача 3 сводится к  $n$  задачам 2. При этом до сих пор не придумано никаких принципиальных усовершенствований, которые могли бы улучшить оценку сложности или существенно уменьшить время решения задачи 3 по сравнению с  $n$  решениями задачи 2. Существуют специальные алгоритмы, которые решают именно задачу 3 вместо  $n$  решений задачи 2 (алгоритм Флойда, алгоритм Данцига), однако ни по оценке сложности для разреженных сетей, ни по своей фактической эффективности их использование не лучше, чем решение  $n$  раз задачи 2. Так что задачей 3 мы тоже специально заниматься не будем, а порекомендуем каждому, кто захочет получить полную матрицу кратчайших путей,  $n$  раз решить задачу 2.

#### 6.4. Дерево кратчайших путей

Решением задачи 1 является одно число – длина кратчайшего пути и сам путь – последовательность дуг сети.

Решением задачи 3 является матрица кратчайших расстояний и, соответственно, матрица кратчайших путей.

Решение задачи 2 при построении кратчайших путей от заданной вершины  $s$  – это массив кратчайших расстояний  $R[0..n-1]$ , где  $R_i$  – длина кратчайшего пути от вершины  $s$  до вершины  $i$ . Кроме того, надо каким-то образом запомнить кратчайшие пути от  $s$  до всех достижимых из  $s$  вершин сети. Казалось бы, для этого придется хранить целую матрицу, поскольку каждый путь может содержать максимум  $(n-1)$  дугу, а количество путей составляет  $(n-1)$ . Однако скоро мы увидим, что для хранения всех кратчайших путей хватит одного массива. Это связано с возможностью так подобрать кратчайшие пути, выходящие из одной вершины, что они будут образовывать дерево – граф без циклов. А для кодировки дерева хватит и одного массива.

Сначала покажем, что любой фрагмент кратчайшего пути также является кратчайшим путем (рис. 6.1). Пусть  $W_{kl}$  – фрагмент кратчайшего пути  $W_{ij}$ , а  $W'_{kl}$  – какой-то другой путь, соединяющий вершины  $k$  и  $l$ . Рассмотрим путь  $W'_{ij} = W_{ik} + W'_{kl} + W_{lj}$ . Поскольку путь  $W_{ij}$  кратчайший, то  $c(W_{ij}) \leq c(W'_{ij})$ . Значит,  $c(W_{ik}) + c(W_{kl}) + c(W_{lj}) \leq c(W_{ik}) + c(W'_{kl}) + c(W_{lj})$ , а поэтому и  $c(W_{kl}) \leq c(W'_{kl})$ , то есть путь  $W_{kl}$  – кратчайший среди всех путей, соединяющих вершины  $k$  и  $l$ .

Теперь докажем, что существует такой набор кратчайших путей от заданной вершины  $s$  до остальных достижимых из  $s$  вершин сети, объединение которых является деревом.

Напомним, что  $n$  – число вершин сети. Возьмем произвольное множество кратчайших путей от вершины  $s$  до всех достижимых

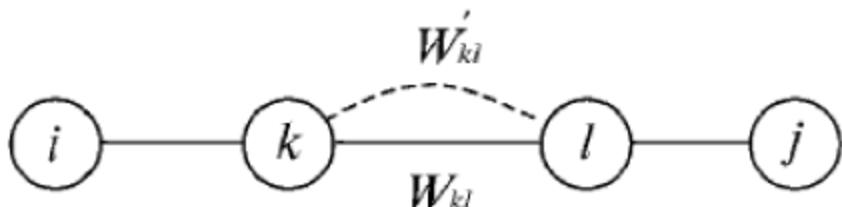


Рис. 6.1

вершин сети. Обозначим через  $H$  граф, образованный всеми кратчайшими путями, выходящими из вершины  $s$ , а через  $m$  – число дуг в этом графе.

Граф  $H$  является связным. Если  $m = n - 1$ , то  $H$  – дерево. Если  $m > n - 1$ , то найдется хотя бы одна вершина, в которую входит больше одной дуги графа  $H$ , потому что если бы в каждую вершину, кроме  $s$ , входила ровно одна дуга (в  $s$  не входит ни одной дуги), то количество дуг было бы в точности равно  $(n - 1)$ . Пусть два кратчайших пути  $W_{si}$  и  $W_{sj}$  входят в вершину  $t$ . Обозначим через  $(k, t)$  дугу пути  $W_{si}$ , входящую в вершину  $t$ , а через  $(l, t)$  – соответствующую дугу пути  $W_{sj}$  (рис. 6.2).

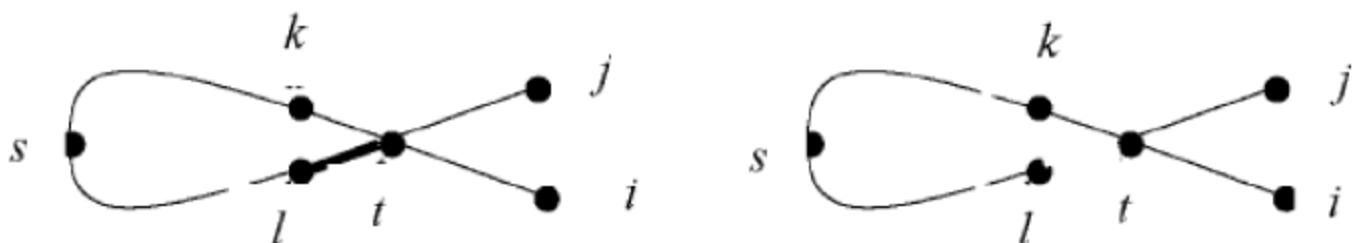


Рис. 6.2

Удалим из графа  $H$  дугу  $(l, t)$ . Докажем, что новый граф  $H'$  тоже будет содержать кратчайшие пути из  $s$  во все достижимые вершины. Если кратчайший путь в графе  $H$  не содержит дугу  $(l, t)$ , то этот путь присутствует и в графе  $H'$ . Если же путь  $W_{sx}$  содержит дугу  $(l, t)$ , то этот путь состоит из двух фрагментов:  $W_{sx} = W_{slt} + W_{tx}$ . Рассмотрим путь  $W'_{sx} = W_{skt} + W_{tx}$ . Поскольку оба пути  $W_{si}$  и  $W_{sj}$  кратчайшие, то их фрагменты  $W_{skt}$  и  $W_{slt}$  – кратчайшие пути, соединяющие  $s$  и  $t$ . Значит,  $c(W_{skt}) = c(W_{slt})$  и, следовательно,

$c(W'_{st}) = c(W_{st})$ . Отсюда следует, что путь  $W'_{sx}$  тоже кратчайший, и граф  $H'$  содержит кратчайший путь до вершины  $x$ .

Мы доказали, что если количество дуг в графе  $H$ , являющемся объединением кратчайших путей от  $s$  до всех достижимых вершин, больше  $(n - 1)$ , то найдется граф  $H'$ , содержащий все кратчайшие пути, но с числом дуг на единицу меньше. Отсюда следует, что минимальное число дуг в графе, являющемся объединением кратчайших путей во все достижимые вершины, равно  $(n - 1)$ , где  $n$  – число достижимых вершин. А значит, этот граф – обозначим его  $T_s$  – является деревом, что непосредственно следует из одного из определений дерева. Будем называть граф  $T_s$  деревом кратчайших путей с корнем в вершине  $s$ . При построении кратчайших путей от одной вершины до всех остальных будет построено именно это дерево.

## 6.5. Представление в компьютере дерева кратчайших путей

Существует много различных способов кодировки деревьев. К какой из них целесообразно использовать, зависит от конкретной ситуации. В нашем случае в дереве  $T$  есть одна выделенная вершина  $s$ , которую мы будем называть корнем. Для каждой вершины  $i$  существует единственный путь  $W_i$ , ведущий в нее из корня. Обозначим через  $p_i$  вершину, которая непосредственно предшествует  $i$  в пути  $W_i$ . Эта вершина называется предком вершины  $i$  в дереве  $T$ . У корня предка нет. Для того чтобы задать дерево, достаточно для каждой вершины указать ее предка, то есть для кодировки дерева нужно хранить один массив  $P[0..n-1]$ , в котором в переменной  $P[i]$  содержится номер предка вершины  $i$ . Такой массив  $P$  обычно называют списком предков. На рис. 6.3 изображено дерево и приведен список его предков.

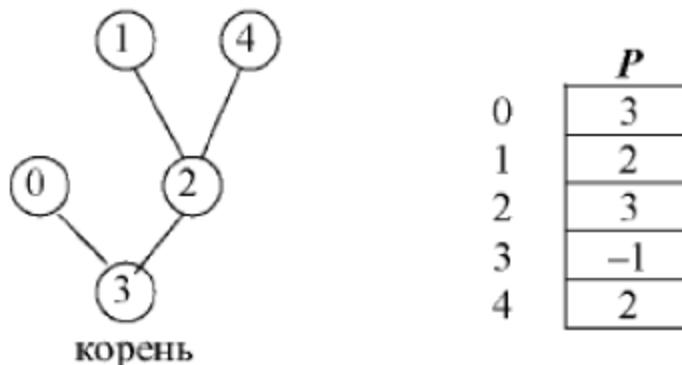


Рис. 6.3

Список предков во многих случаях, в том числе и при построении кратчайших путей, является удобным представлением дерева. Если в результате работы какого-либо алгоритма построения дерева кратчайших путей будет получен список предков, то этого достаточно для того, чтобы восстановить весь путь из вершины  $s$  в любую достижимую вершину. Возьмем какую-то вершину  $i$ . Вершина  $P[i]$  – ее предок – предпоследняя вершина на пути из  $s$  в  $i$ .  $P[P[i]]$  – вершина, находящаяся на этом пути перед  $P[i]$ , и так далее. Один за другим находя предков, мы дойдем до корня, а переписав эту последовательность вершин в обратном порядке, получим кратчайший путь, ведущий из  $s$  в  $i$ .

Таким образом, решение задачи построения дерева кратчайших путей будет представлено в двух массивах – списке предков  $P[0..n-1]$  и в массиве расстояний  $R[0..n-1]$ , в котором в переменной  $R[i]$  хранится величина кратчайшего расстояния от  $s$  до  $i$ , если вершина  $i$  достижима, и неопределенная величина, если она не достижима.

В качестве предка корня  $s$  мы будем указывать какое-либо фиктивное значение например «–1», а в качестве предка недостижимой вершины – «еще более фиктивное» значение, например «–2».

На самом деле в задачах построения кратчайших путей целесообразно в списке предков в переменной  $P[i]$  задавать не номер предка, а номер дуги, которая в дереве кратчайших путей ведет из предка в вершину  $i$ . Это ничуть не усложняет задачу нахождения вершин пути, поскольку, зная номер дуги, легко найти ее начальную и конечную вершины. С другой стороны, если известен номер дуги, а не номер вершины, то будет гораздо легче вычислять разные характеристики пути, поскольку, как правило, они привязаны именно к дугам, а не к вершинам. К тому же, если в сети есть параллельные дуги (несколько дуг, соединяющих одну и ту же пару вершин), то будет гораздо легче определить, какая именно из этих дуг входит в дерево кратчайших расстояний.

Итак, решением задачи построения дерева кратчайших путей будут два массива:  $R$  и  $P$ . В  $R$  хранятся величины кратчайших расстояний, а в  $P$  – модифицированный список предков:  $P[i]$  – номер дуги кратчайшего пути из  $s$ , входящей в вершину  $i$ .

## 6.6. Алгоритм построения кратчайших путей на сети с единичными длинами дуг: поиск в ширину

Если длины всех дуг равны единице, то длина пути равна количеству входящих в него дуг. Для построения дерева кратчайших путей на такой сети можно воспользоваться обходом графа, который называется *поиск в ширину* (*breadth first search*).

В процессе работы алгоритма вершины могут находиться в следующих альтернативных состояниях: помечена—непомечена, просмотрена—непросмотрена. В начале работы алгоритма все вершины являются непросмотренными и, кроме исходной – непомеченными. По ходу работы алгоритма достижимые вершины получают пометки. На очередной итерации выбираем помеченную, но непросмотренную вершину и просматриваем все выходящие из нее дуги. После этого вершина становится просмотренной и остается таковой до конца работы алгоритма.

На рис. 6.4 проиллюстрирована эволюция вершин в ходе работы алгоритма. Через  $S$  обозначено множество просмотренных вершин, через  $T$  – множество непомеченных, а через  $Q$  – множество помеченных, но непросмотренных.

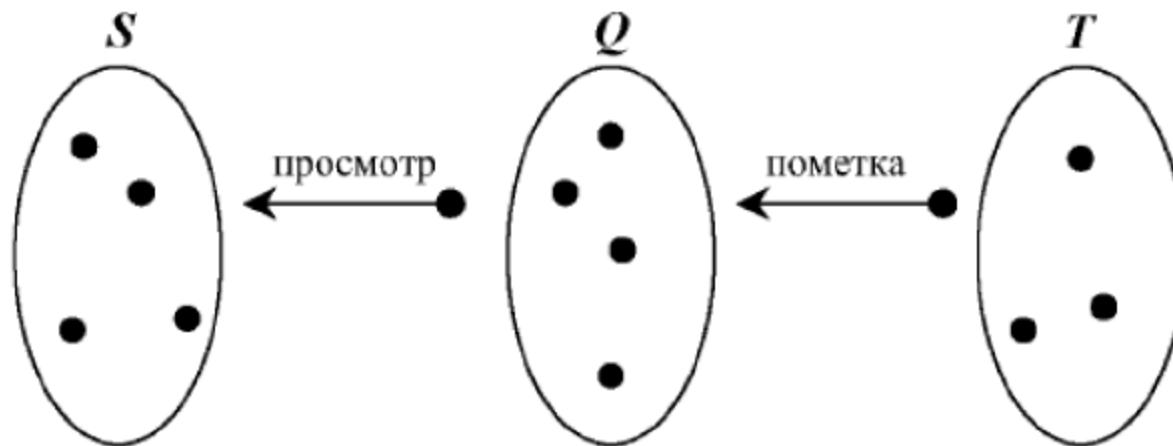


Рис. 6.4

### Описание алгоритма

Строим дерево кратчайших путей от заданной вершины  $S$  до всех достижимых из  $S$  вершин. Алгоритм состоит из последовательности итераций. На каждой итерации сначала выбираем помеченную, но непросмотренную вершину, а затем просматриваем все выходящие из нее дуги.

**Инициализация.** Все вершины объявляем непросмотренными, а все вершины, кроме исходной вершины  $s$ , – непомеченными. Вер-

шину  $s$  помещаем в множество  $Q$  – помеченных, но непросмотренных вершин. Текущее расстояние  $r_i$  для всех  $i \neq s$  делаем равным «бесконечности», а  $r_s$  полагаем равным нулю.

В качестве «бесконечности» можно использовать число, которое заведомо больше самого длинного кратчайшего пути. Поскольку количество дуг в любом пути не превышает  $(n - 1)$  ( $n$  – число вершин сети), то длина пути тоже не может быть больше  $(n - 1)$  и, значит, «бесконечностью» может служить число  $n$ .

**Выбор вершины.** Вершины из множества  $Q$  выбираются для просмотра в порядке очереди, то есть вершина, которая раньше была помечена, и, соответственно, раньше попала в множество  $Q$ , будет раньше выбрана для просмотра. Такой порядок выбора можно выразить следующей формулой: «первым помечен – первым просмотрен» («first labeled – first scanned»): выбираем для просмотра вершину  $i$ , находящуюся в начале очереди. Удаляем  $i$  из множества  $Q$ .

На первой итерации для просмотра выбирается вершина  $s$ , поскольку после инициализации других вершин в очереди нет.

Когда очередь станет пустой, работа алгоритма заканчивается – дерево кратчайших расстояний построено.

**Просмотр ребер.** Берем первую дугу, выходящую из просматриваемой вершины  $i$ . Пусть  $j$  – вершина, находящаяся на другом конце этой дуги.

Если вершина  $j$  помечена, то сразу переходим к просмотру следующей дуги, выходящей из  $i$ .

Если  $j$  не помечена, то помечаем ее и записываем в очередь на просмотр.

Фиксируем, что в дереве кратчайших путей  $i$  является предком  $j$ , а  $r_j$  – расстояние до  $j$  – полагаем равным  $(r_i + 1)$ .

Переходим к просмотру следующей дуги.

После того как все выходящие из  $i$  ребра обработаны, эта вершина объявляется просмотренной.

## 6.7. Обоснование алгоритма и оценка его сложности

Как видите, алгоритм очень прост. Столь же просто и доказательство его корректности. Рангом вершины  $i$  будем называть число дуг в пути, ведущем в дереве кратчайших путей из  $s$  в  $i$ . В сети с еди-

ничными длинами дуг ранг равен кратчайшему расстоянию, поэтому будем обозначать его, как и кратчайшее расстояние, через  $r_i$ .

Индукцией по величине ранга докажем следующие утверждения.

1. Все вершины ранга  $k$  просматриваются последовательно: никакая вершина иного ранга не может быть просмотрена в промежутке между двумя вершинами ранга  $k$ .

2. После завершения просмотра вершин ранга  $k$  в очереди на просмотр будут записаны только вершины ранга  $(k+1)$  и их текущее расстояние будет равно  $(k+1)$ .

Ранжирование вершин сети представлено на рис. 6.5.

В начале работы алгоритма в очереди на просмотр находится единственная вершина  $s$  ранга 0. Поэтому первое утверждение для ранга 0 очевидно.

В каждую вершину ранга 1 ведет дуга из  $s$ . Из алгоритма просмотра следует, что после просмотра дуг, выходящих из  $s$ , все вершины первого ранга будут помечены, записаны в очередь  $Q$ , текущим расстоянием для них будет 1, и никаких других вершин в  $Q$  не будет, то есть второе утверждение для ранга 0 также выполняется.

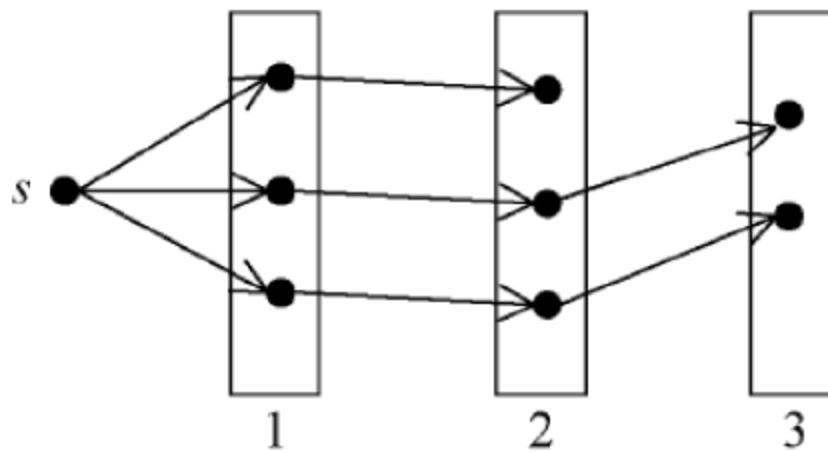


Рис. 6.5

Пусть оба утверждения выполняются для ранга  $k$ . Докажем, что они справедливы и для ранга  $(k+1)$ .

В соответствии с первым утверждением все вершины ранга  $k$  просматриваются последовательно, при этом из второго утверждения следует, что во время просмотра их текущие расстояния будут равны  $k$ . Поскольку ранг равен кратчайшему расстоянию, то из вершины ранга  $k$  дуга может вести только в вершины, ранг которых не превышает  $(k+1)$ . Так как все вершины ранга, меньшего  $(k+1)$ , к

моменту просмотра вершин ранга  $k$  помечены, то при просмотре вершин этого ранга в очередь  $Q$  могут попасть только вершины ранга  $(k+1)$ , и их текущее расстояние при этом будет равно  $(k+1)$ . При этом после завершения просмотра вершин ранга  $k$  в очереди будут все вершины ранга  $(k+1)$ , потому что в каждую вершину этого ранга из вершины ранга  $k$  ведет дуга, принадлежащая дереву кратчайших путей.

Таким образом, после завершения просмотра вершин ранга  $k$  в очереди будут только вершины ранга  $(k+1)$  с корректно вычисленными величинами расстояний от  $s$ , то есть второе утверждение справедливо.

Правильность первого утверждения для ранга  $(k+1)$  следует из того, что поскольку вершины просматриваются в порядке поступления в очередь  $Q$ , а в данный момент никаких других вершин, кроме вершин ранга  $(k+1)$ , в очереди нет, то эти вершины будут просмотрены последовательно, раньше всех остальных вершин, которые появятся в очереди потом. Оба утверждения доказаны.

Корректность алгоритма непосредственно следует из второго утверждения, которое означает, что вершина попадает в очередь на просмотр с правильно вычисленным кратчайшим расстоянием. Из описания алгоритма следует, что это расстояние уже не изменится, значит, и в конце работы алгоритма оно останется правильным.

Корректность алгоритма доказана.

Теперь оценим сложность алгоритма. Каждую вершину мы просматриваем ровно один раз, следовательно, каждую дугу мы просматриваем тоже ровно один раз. Таким образом, на просмотр дуг будет потрачено порядка  $O(m)$  действий, где  $m$  – число дуг сети. На помещение вершины в очередь и выбор ее из очереди мы тратим  $O(1)$  действий. Поскольку вершина может попасть в очередь на просмотр не более одного раза, то суммарное число действий, потраченных на эти операции, не превышает  $O(n)$ . Итак, общая оценка сложности алгоритма составляет  $O(m) = O(m) + O(n)$ .

## 6.8. Общее описание поиска в ширину

В третьем разделе в алгоритме нахождения компонент связности был использован обход графа, называемый поиском в глубину. В алгоритме построения кратчайших путей на сети с единичными длинами мы применили другой способ обхода графа, который называется *поиском в ширину*.

При поиске в ширину должны соблюдаться следующие правила.

1. Вершина может находиться в множестве  $T$  (не помечена), в множестве  $Q$  (помечена, но не просмотрена) или в множестве  $S$  (просмотрена).
2. В начале обхода, как правило, одна вершина (иногда несколько) помечена, но не просмотрена, а остальные – не помечены.
3. Вершины могут перемещаться между подмножествами только в указанном направлении:  $T \rightarrow Q \rightarrow S$ .
4. Очередная вершина для просмотра выбирается из множества  $Q$  в порядке очереди, то есть соблюдается правило: «первым помечен – первым просмотрен».
5. При просмотре дуг, выходящих из вершины, получают пометки те и только те вершины, находящиеся на противоположном конце просматриваемых дуг, которые не были помечены ранее.
6. Во время просмотра вершины мы не можем начать просматривать другую. Просмотр вершины заканчивается и она получает статус просмотренной только после того, как просмотрены все выходящие из нее дуги.

Перечислив эти правила, мы фактически еще раз описали алгоритм построения дерева кратчайших путей на сети с единичными длинами дуг.

Если вспомнить аналогию с поведением ветреного юноши, которую мы использовали при описания поиска в глубину, то поведение при поиске в ширину выглядит гораздо основательней. Мы не переходим к просмотру новой вершины, пока не используем «все возможности» текущей. Поэтому, в отличие от поиска в глубину, при поиске в ширину имеется не стек, а «фронт», состоящий из помеченных, но непросмотренных вершин.

В чистом виде поиск в ширину используется в алгоритмах на се-тях реже поиска в глубину. Однако в некоторых алгоритмах применяются процедуры, похожие на поиск в ширину, когда нарушаются некоторые из приведенных правил. Иногда, например, вершина мо-

жет вернуться из множества  $S$  в множество  $Q$ . А иногда вершины для просмотра выбираются из множества  $Q$  не по очереди, а в соответствии с более сложными правилами. Именно такие алгоритмы построения кратчайших путей мы рассмотрим в двух следующих разделах.

## 6.9. Реализация алгоритма

Как всегда, начнем со структуры данных. Поскольку нужно последовательно просматривать дуги, выходящие из одной вершины, то воспользуемся стандартным представлением сети в виде списков пучков дуг. Сами дуги записаны в массивах  $I[0..m-1]$  – начало дуги и  $J[0..m-1]$  – конец дуги, а списки пучков дуг – в массивах  $H[0..n-1]$  – головы списков и  $L[0..m-1]$  – ссылки.

Необходимо хранить величины расстояний до исходной вершины. Для этого понадобится массив  $R[0..n-1]$ . В переменной  $R[i]$  хранится расстояние от исходной вершины  $s$  до вершины  $i$ . В начале работы все расстояния, кроме  $R[s]$ , равны бесконечности, в качестве которой можно использовать  $n$  – число вершин в сети.

Кроме того, понадобится запоминать дерево кратчайших путей, для чего мы будем использовать модифицированный список предков  $P[0..n-1]$ , где в переменной  $P[i]$  хранится номер дуги, входящей в вершину  $i$  в дереве кратчайших путей. Для корня  $s$  в этом массиве хранится «-1», а для недостижимой вершины «-2».

Еще понадобится очередь  $Q$ , в которой будут находиться вершины для просмотра. Проще всего организовать очередь в виде массива  $Q[0..n-1]$  и двух указателей  $x$  и  $w$ . Переменная  $x$  указывает на голову очереди – позицию, где находится первый в очереди элемент. Переменная  $w$  указывает на первое свободное место в массиве  $Q$ . Поскольку за время работы алгоритма каждая вершина может побывать в очереди не более одного раза, то для хранения информации об очереди нам хватит  $n$  переменных и не придется прибегать к каким-либо ухищрениям.

Помеченные и непомеченные вершины мы будем различать с помощью, например, значения переменной  $R[i]$ . Вершина  $i$  помечена, если  $R[i] < n$ . Специально отмечать просмотренные вершины нет необходимости – они просто удаляются из очереди  $Q$  (рис. 6.6).

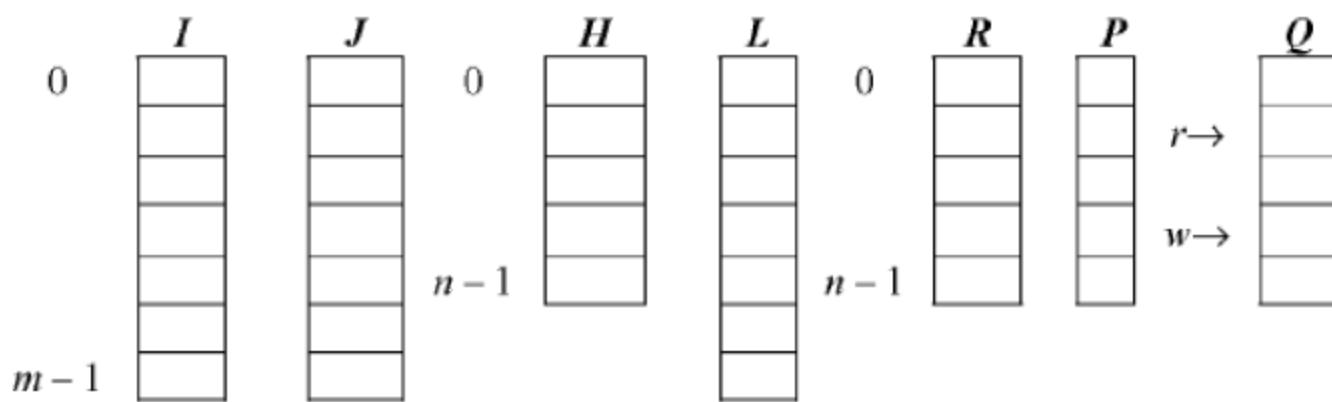


Рис. 6.6

Теперь не составит труда написать программу построения дерева кратчайших путей на сети с единичными длинами дуг.

```
//ПРОГРАММА ПОСТРОЕНИЯ ДЕРЕВА КРАТЧАЙШИХ ПУТЕЙ
//НА СЕТИ С ЕДИНИЧНЫМИ ДЛИНАМИ ДУГ

//Инициализация
//Построение списков ребер Н и Л (раздел 2)
. . . . .
//Списки построены
R[0..n-1]=n // Вершин в сети n, то есть «п=∞»
P[0..n-1]=-2 // Пока все вершины недоступны

R[s]=0; // s - исходная вершина
P[s]=-1;
// Инициализация очереди
Q[0]=s; // Исходную вершину - в очередь
// на просмотр
r=0; // Откуда читать
w=1; // Куда писать
// Инициализация закончена

// ОСНОВНОЙ ЦИКЛ
while ( r < w ) // Пока очередь не пуста
{
    i = Q[r]; // Берем очередную вершину для
    // просмотра
    r++; // Передвигаем указатель чтения

    // Просмотр дуг, выходящих из вершины i
    for ( k = H[i]; k != -1; k = L[k] )
        if ( P[k] < 0 )
            P[k] = R[i] + 1
            Q[w] = k
            w++
}
```

```

{
    j = J[k]; // Противоположная вершина
    if ( R[j] == n )
    { // Вершина j не помечена
        R[j] = R[i]+1; //Расстояние до j на
                         //единицу больше, чем до
                         // i
        P[j] = k;          //Последняя дуга на пути
                           //из s в j - это k
        Q[w]=j;           //Помещаем вершину j в
                           //очередь на просмотр
        w++;              //Передвигаем указатель записи
    }
    // Если j помечена, то ничего делать не
    // надо
} // Конец цикла просмотра дуг
} // Конец основного цикла, а вместе с ним и
// всей программы

```

Как видим, программа очень простая. К сожалению (или к счастью), длины дуг не всегда равны единице. Для сетей с произвольными длинами дуг приходится использовать чуть более сложные алгоритмы, которые мы рассмотрим в двух последующих разделах.

## 7. ЗАДАЧА ПОСТРОЕНИЯ КРАТЧАЙШИХ ПУТЕЙ (АЛГОРИТМ БЕЛЛМАНА – ФОРДА)

В этом разделе мы рассмотрим задачу построения дерева кратчайших путей на сети с дугами произвольной (в том числе отрицательной) длины. Будем предполагать, что отрицательных циклов в сети нет.

Прежде чем описывать алгоритм Беллмана – Форда, приведем сначала «общий» алгоритм построения дерева кратчайших путей. Абсолютное большинство (а, пожалуй, даже все) алгоритмов построения дерева кратчайших путей являются конкретизацией общего алгоритма.

### 7.1. Общий алгоритм

Общий алгоритм за конечное число шагов строит дерево кратчайших путей от заданной вершины  $s$  до остальных вершин сети, достижимых из  $s$ .

В начале работы алгоритма текущее расстояние  $r_i$  для всех  $i \neq s$  делаем равным  $\infty$ , а  $r_s$  полагаем равным нулю. Текущее дерево кратчайших путей вначале состоит из одной вершины  $s$ , а остальные вершины графа к нему не присоединены. Все дуги считаем непросмотренными.

#### *Описание алгоритма*

Алгоритм состоит из последовательности итераций. На каждой из них мы просматриваем одну дугу сети и, возможно, изменяем дерево кратчайших путей.

**Выбор дуги.** Выбираем  $a_{ij}$  – любую непросмотренную дугу сети. Если таких дуг нет – работа алгоритма заканчивается. Дерево кратчайших путей построено.

**Проверка основного соотношения.** Если  $r_i + c(a_{ij}) < r_j$ , то дерево кратчайших путей можно улучшить. В противном случае помечаем дугу  $a_{ij}$  как просмотренную и переходим к выбору новой дуги.

**Улучшение дерева.** Если вершина  $j$  ранее была вне текущего дерева кратчайших путей – включаем ее в дерево.

Если вершина  $j$  ранее была в текущем дереве кратчайших путей, то исключаем из него дугу  $(P_j, j)$ , где  $P_j$  – предок  $j$  в дереве кратчайших путей.

Полагаем  $r_j = r_i + c(a_{ij})$ .

Предком вершины  $j$  в текущем дереве кратчайших путей будет вершина  $i$ :  $P_j = i$ , а дугу  $a_{ij}$  включаем в дерево.

После улучшения дерева объявляем все дуги непросмотренными и переходим к **Выбору дуги**.

## 7.2. Пример одной итерации общего алгоритма

Если вершина  $j$  не входила в текущее дерево кратчайших путей, то в результате просмотра дуги  $a_{ij}$  был впервые найден путь до этой вершины, она стала достижимой и была включена в текущее дерево кратчайших путей (рис. 7.1).

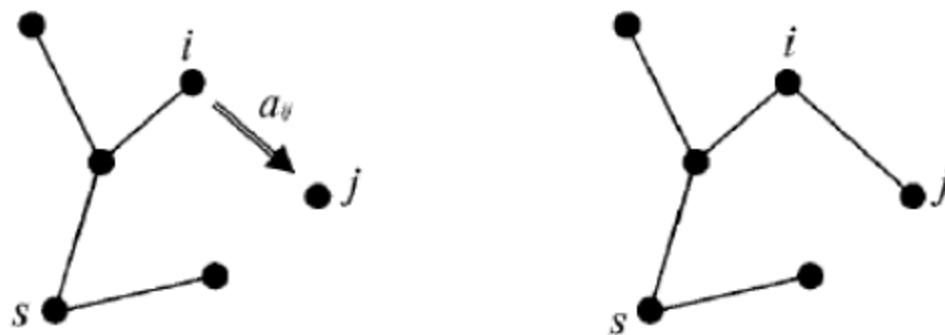


Рис. 7.1

Для случая, когда вершина  $j$  уже находится в дереве кратчайших путей, далее, при обосновании алгоритма, будет доказано, что текущее расстояние  $r_j$  равно длине некоторого пути  $V_j$ , ведущего из  $s$  в  $j$ . Во время улучшения дерева путь  $V_j$  заменяется на новый, более короткий, ведущий в  $j$  через вершину  $i$  (рис. 7.2).

Осталось доказать, что процесс улучшения дерева конечен и в конце работы общего алгоритма будет построено дерево кратчайших путей и массив кратчайших расстояний, соответствующий этому дереву.



Рис. 7.2

### 7.3. Обоснование общего алгоритма

При улучшении дерева кратчайших путей при помощи дуги  $a_{ij}$  текущее расстояние  $r_j$  до вершины  $j$  уменьшается, то есть каждое улучшение дерева влечет за собой уменьшение текущего расстояния до некоторой вершины.

Докажем, что если в сети нет отрицательных циклов, то каждому конечному текущему расстоянию  $r_j$  соответствует какой-то простой путь  $V_i$  длины  $r_j$  от вершины  $s$  до вершины  $j$ .

Будем доказывать это утверждение индукцией по номеру улучшения дерева. До самого первого улучшения утверждение очевидно. Единственное текущее расстояние, отличное от бесконечности, это  $r_s = 0$ , и ему соответствует пустой путь, начинающийся и заканчивающийся в вершине  $s$ .

Предположим, что это утверждение верно для  $(k - 1)$ -го улучшения дерева. Докажем, что оно останется справедливым и после  $k$ -го улучшения.

Предположим, что на  $k$ -й итерации с помощью дуги  $a_{ij}$  было уменьшено текущее расстояние вершины  $j$ , то есть  $r_i + c(a_{ij}) < r_j$ . Тогда после этой итерации  $r_j^{new} = r_i + c(a_{ij})$ , и если вершина  $j$  не находится на пути  $V_i$ , то путь  $V_j = V_i + a_{ij}$  – простой и длина его равна  $c(V_j) = r_j^{new} = r_i + c(a_{ij})$  (рис. 7.3), то есть утверждение индукции верно. Осталось доказать, что вершина  $j$  не может находиться на пути  $V_i$ . Предположим противное (рис. 7.4). Длина пути  $V_i$  равна  $c(V_i) = r_i$ , при этом  $c(V_i) = c(s, j) + c(j, i)$ . Длина пути  $(s, j)$  –  $c(s, j) \geq r_j$ , поскольку с того времени, как текущее расстояние до  $j$  было равно  $c(s, j)$ , в результате работы алгоритма оно могло только уменьшиться.

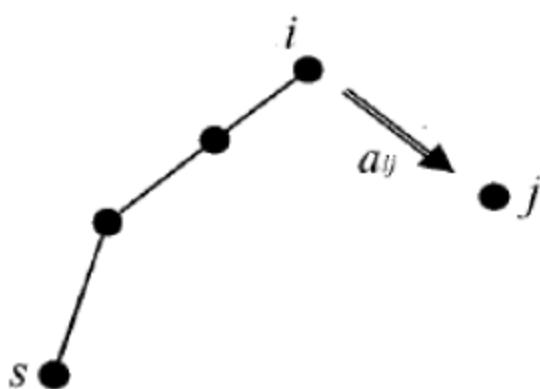


Рис. 7.3

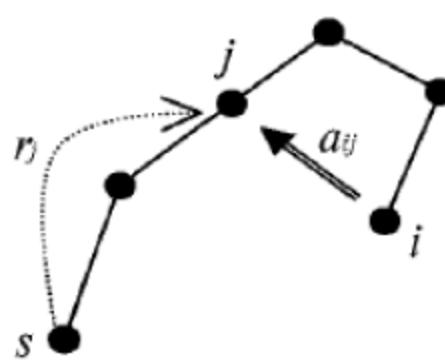


Рис. 7.4

По предположению  $r_i + c(a_{ij}) < r_j$ . Следовательно,  $r_i + c(a_{ij}) = c(s, j) + c(j, i) + c(a_{ij}) < r_j \leq c(s, j)$ . Отсюда следует, что  $c(j, i) + c(a_{ij}) < 0$ , то есть, вопреки предположению, в сети есть отрицательный цикл. Но поскольку отрицательных циклов в сети нет, то это означает, что вершина  $j$  не могла находиться на пути  $V_i$ . Таким образом, мы доказали, что после очередной итерации вершине  $j$ , до которой уменьшилось текущее расстояние, соответствует новый простой путь  $V_i$  длины  $r_j$  от вершины  $s$  до  $j$ .

Отсюда естественным образом следует конечность общего алгоритма. Число простых путей от вершины  $s$  до любой вершины  $i$  конечно. Поскольку при каждом уменьшении текущего расстояния его новой величине соответствует новый простой путь от  $s$  до  $i$ , то, значит, уменьшаться текущее расстояние до любой вершины может конечное число раз. При улучшении дерева уменьшается текущее расстояние до какой-то вершины. Следовательно, общее число улучшений дерева ограничено и после конечного числа итераций работа алгоритма завершится: все дуги сети просмотрены и ни одна из них не позволяет улучшить дерево кратчайших путей.

Осталось доказать, что если ни одна дуга сети не дает возможность улучшить текущие расстояния, то эти расстояния являются кратчайшими, а построенное дерево является деревом кратчайших путей.

Расстояние, найденное с помощью общего алгоритма, не может быть меньше кратчайшего, так как выше было доказано, что ему соответствует длина некоторого простого пути.

Пусть  $T$  – какое-то дерево кратчайших путей. *Рангом вершины* назовем количество дуг в пути, ведущем в дереве кратчайших путей из  $s$  в эту вершину.

Предположим, что найдется вершина, у которой найденное с помощью общего алгоритма расстояние больше кратчайшего. Среди таких вершин возьмем вершину наименьшего ранга. Обозначим ее через  $j$ , найденное расстояние через  $r_j$ , а кратчайшее расстояние через  $r_j^{\min} < r_j$ . Пусть  $i$  – предок вершины  $j$  в дереве кратчайших расстояний  $T$ , а  $a_{ij}$  – дуга дерева, их соединяющая. Поскольку ранг  $i$  меньше ранга  $j$ , то для дуги  $a_{ij}$  построенное расстояние равно минимальному:  $r_i^{\min} = r_i$ . По определению дерева кратчайших путей

$r_j^{\min} = r_i^{\min} + a_{ij}$ . Следовательно,  $r_j > r_i + c(a_{ij})$ , и вопреки тому, что работа алгоритма закончена и дерево улучшить нельзя, нашлась дуга  $a_{ij}$ , улучшающая дерево. Следовательно, исходное предположение неверно и построенные расстояния равны кратчайшим. Отсюда также следует, что построенное дерево является деревом кратчайших путей, поскольку длины путей, состоящих из включенных в него дуг, равны величинам кратчайших расстояний.

Итак, корректность общего алгоритма доказана. В результате его работы на сети без отрицательных циклов будет построено дерево кратчайших путей независимо от того, в каком порядке происходит просмотр дуг.

Оценка сложности общего алгоритма экспоненциальна, и мы не будем ее выписывать. В дальнейшем будут рассмотрены алгоритмы построения дерева кратчайших путей с полиномиальными оценками.

Фактически все алгоритмы построения дерева кратчайших путей являются конкретизацией общего алгоритма и отличаются друг от друга только порядком просмотра дуг. Выбирая разные стратегии просмотра, можно получить разные алгоритмы решения этой задачи.

#### 7.4. Алгоритм Беллмана – Форда

Очевидно, что если напрямую запрограммировать общий алгоритм, то работать он будет очень плохо. Однако можно определять разумный порядок просмотра дуг исходя из соображений, помогающих уменьшить время решения задачи.

Первое напрашивающееся улучшение состоит в том, что если уже мы начали просматривать дуги, выходящие из одной «перспективной» вершины, то имеет смысл просмотреть их все и только потом переходить к другим дугам. Это разумно, поскольку если через вершину проходит один кратчайший путь, то, вероятно, пройдут и другие. При такой стратегии следует говорить не о выборе дуги для просмотра, а о выборе вершины.

Второе улучшение относится к выбору вершины для просмотра. Имеет смысл просматривать только те вершины, до которых после предыдущего просмотра уменьшилось текущее расстояние. Целесообразность такого улучшения очевидна: если текущее расстояние не уменьшилось, то ничего нового мы не получим.

Эти два правила применяются во всех известных алгоритмах построения дерева кратчайших путей. Различаются алгоритмы между собой только порядком выбора «перспективных» вершин для просмотра.

В алгоритме Беллмана – Форда, предложенном в 50-х годах, при выборе просматриваемой вершины используется очередь: как только текущее расстояние до вершины уменьшилось – она попадает в очередь для просмотра. Вершины выбираются для просмотра в порядке их поступления в очередь.

Опишем алгоритм Беллмана – Форда<sup>1</sup>.

### *Описание алгоритма*

Алгоритм строит дерево кратчайших путей от заданной вершины  $s$  до всех достижимых из  $s$  вершин. Алгоритм состоит из последовательности итераций. На каждой итерации сначала выбираем из очереди вершину, а затем просматриваем все выходящие из нее дуги.

**Инициализация.** Вершину  $s$  помещаем в очередь  $Q$ . Текущее расстояние  $r_i$  для всех  $i \neq s$  делаем равным «бесконечности», а  $r_s$  полагаем равным нулю.

Здесь в качестве «бесконечности» нужно использовать число, которое заведомо больше самого длинного кратчайшего пути. Это число можно определить из существа задачи или из ограничений, заданных разрядностью используемых переменных.

**Выбор вершины.** Выбираем для просмотра первую вершину  $i$  из очереди  $Q$ .

Вершина, которая раньше попала в множество  $Q$ , будет раньше выбрана для просмотра.

Если очередь пуста – работа алгоритма заканчивается – дерево кратчайших расстояний построено.

Удаляем  $i$  из очереди  $Q$ .

На первой итерации для просмотра будет выбрана вершина  $s$ , поскольку после инициализации других вершин в очереди нет.

**Просмотр дуг.** Берем очередную дугу  $a_{ij}$ , выходящую из просматриваемой вершины  $i$ . Пусть  $j$  – вершина, находящаяся на другом конце этой дуги.

Проверяем основное соотношение. Если оно нарушено:  $r_i + c(a_{ij}) < r_j$ , то дерево кратчайших путей можно улучшить.

---

<sup>1</sup> В шестидесятые годы в Советском Союзе этот алгоритм иногда называли «алгоритмом метлы».

Если основное соотношение для дуги  $a_{ij}$  выполнено:  $r_i + c(a_{ij}) \geq r_j$ , то сразу переходим к просмотру следующей дуги, выходящей из  $i$ .

**Улучшение дерева.** Фиксируем, что в дереве кратчайших путей  $i$  является предком  $j$ , а  $r_j$  – расстояние до  $j$  – полагаем равным  $r_i + c(a_{ij})$ . Включаем дугу  $a_{ij}$  в дерево кратчайших путей.

Если вершина  $j$  не находится в очереди  $Q$ , то записываем ее в конец этой очереди. Если  $j$  находится в очереди, то ничего не меняем.

Переходим к просмотру следующей дуги, выходящей из  $i$ .

После того, как все выходящие из  $i$  ребра обработаны, эта вершина объявляется просмотренной, и мы переходим к выбору следующей вершины.

Алгоритм Беллмана – Форда является частным случаем общего алгоритма построения дерева кратчайших путей. Действительно, если очередь на просмотр пуста, то, значит, нет ни одной вершины, текущее расстояние до которой уменьшилось бы после последнего просмотра. Но тогда очевидно, что никакая дуга не может улучшить дерево кратчайших расстояний, поскольку для всех выполнено основное соотношение. Таким образом, специально обосновывать корректность алгоритма Беллмана – Форда нет необходимости, она следует из корректности общего алгоритма. А вот оценка сложности алгоритма Беллмана – Форда, благодаря предложенным улучшениям, оказывается полиномиальной, в чем мы скоро убедимся.

Обратите внимание, что способ просмотра вершин в алгоритме Беллмана – Форда очень похож на поиск в ширину. Разница состоит в том, что вершина может попадать в очередь на просмотр многократно.

#### 7.4. Оценка сложности алгоритма Беллмана – Форда

Напомним, что рангом вершины  $i$  называется количество дуг в пути, ведущем в дереве кратчайших путей из  $s$  в  $i$ .

В процессе работы алгоритма Беллмана – Форда происходит просмотр вершин. Выделим фазы просмотра.

Фаза 0 заканчивается после просмотра всех вершин 0-го ранга (то есть вершины  $s$ ).

Фаза 1 начинается после завершения фазы 0 и заканчивается после просмотра всех вершин ранга 1.

Фаза  $k$  начинается после завершения фазы  $(k - 1)$  и заканчивается после просмотра всех вершин ранга  $k$ .

**Утверждение.** После завершения  $k$ -й фазы все вершины  $(k + 1)$ -го ранга будут находиться в очереди на просмотр, а текущие расстояния до всех этих вершин будут равны кратчайшим расстояниям до них от вершины  $s$ .

**Доказательство.** Докажем утверждение индукцией по величине ранга. Поскольку на нулевой фазе будут просмотрены все дуги, выходящие из  $s$ , а в каждую вершину 1-го ранга в дереве кратчайших путей из  $s$  ведет дуга, то после завершения нулевой фазы все вершины 1-го ранга окажутся в очереди на просмотр и текущее расстояние до них будет равно кратчайшему.

Предположим, что утверждение справедливо для ранга  $(k - 1)$ . Это значит, что на  $k$ -й фазе будут просмотрены все вершины ранга  $k$  и при этом у каждой из них в момент просмотра текущее расстояние будет равно кратчайшему.

В каждую вершину  $j$   $(k + 1)$ -го ранга в дереве кратчайших расстояний ведет дуга  $(i, j)$  из какой-то вершины  $i$  ранга  $k$ . Очевидно, что при просмотре этой дуги текущее расстояние до вершины  $j$  станет равно кратчайшему и эта вершина попадет в очередь на просмотр. Поскольку просмотр вершин происходит в порядке очереди, а все вершины ранга  $k$  находились в очереди до начала  $k$ -й фазы, то никакая вершина  $(k + 1)$ -го ранга не будет просмотрена до окончания фазы  $k$  и, следовательно, все они в момент завершения этой фазы будут находиться в очереди на просмотр. Утверждение доказано.

Если текущее расстояние до вершины стало равно кратчайшему, то уменьшиться оно уже не может, и такая вершина больше никогда не попадет в очередь на просмотр.

Поскольку ранг вершины не может быть больше  $(n - 1)$  ( $n$  – число вершин сети), то из доказанного утверждения следует, что после окончания  $(n - 2)$ -й фазы у всех вершин текущее расстояние будет равно кратчайшему, и, следовательно, после завершения  $(n - 1)$ -й фазы очередь на просмотр будет пуста и работа алгоритма завершится.

Из сказанного выше независимо от доказательства корректности общего алгоритма следует конечность и корректность алгоритма Беллмана – Форда, но помимо этого отсюда легко получить еще и полиномиальную оценку сложности.

Пусть  $n$  – число вершин сети, а  $m$  – число дуг. Во время одной фазы просмотра каждая вершина может быть просмотрена не более одного раза. Следовательно, каждая дуга сети в течение фазы тоже будет просмотрена не более одного раза. Поскольку наибольшее количество действий во время одной фазы тратится на просмотр дуг, то это значит, что трудоемкость фазы не превышает  $O(m)$ . Выше было отмечено, что общее число фаз не может быть больше, чем  $(n - 1)$ . Отсюда следует, что оценка сложности алгоритма Беллмана – Форда полиномиальна и имеет порядок  $O(nm)$ .

## 7.5. Реализация алгоритма Беллмана – Форда

Структуру данных, необходимую для эффективной реализации алгоритма Беллмана – Форда, уже можно считать привычной. Поскольку выполняемые в алгоритме Беллмана – Форда действия очень похожи на те, что мы производим при построении дерева кратчайших путей на сети с единичными длинами дуг, то и структура данных будет почти такой же.

Сеть кодируется в трех массивах:  $I[0..m-1]$  – начало дуги,  $J[0..m-1]$  – конец дуги,  $C[0..m-1]$  – длина дуги.

Списки пучков дуг, как и раньше, содержатся в массивах  $H[0..n-1]$  – головы списков и  $L[0..m-1]$  – ссылки.

Дерево кратчайших путей и величины текущих расстояний, как и в алгоритме, рассмотренном в предыдущем разделе, будут записаны в двух массивах по тем же самым правилам, что и раньше:  $P[0..n-1]$  – дуги дерева, а  $R[0..n-1]$  – текущие расстояния.

По-другому придется хранить только очередь. На это есть две причины. Во-первых, за время работы алгоритма одна вершина может попадать в очередь многократно, поэтому общее число вершин, побывавших в очереди, заранее не известно. Во-вторых, когда текущее расстояние до вершины уменьшается, нужно помещать вершину в очередь только в том случае, если она в этот момент в очереди не находилась, то есть надо уметь быстро определять, находится ли данная вершина в очереди. Проще всего решить эти задачи, если представить очередь не в виде массива, а в виде списка, который будет находиться в массиве

$Q[0..n-1]$ . В переменной  $h\_Q$  будем хранить начало очереди: номер первой вершины, а в переменной  $t\_Q$  – конец очереди: номер последней вершины. Если вершина  $i$  находится вне очереди, то в переменной  $Q[i]$  будет записано «-2», а если  $i$  попала в очередь, в  $Q[i]$  будет храниться номер следующей после  $i$  вершины. Если  $i$  – последняя вершина в очереди, то  $Q[i] = -1$ . Таким образом, мы решим проблему многократного попадания вершины в очередь, потому что добавлять и удалять вершину в список мы сможем столько раз, сколько это понадобится, без какой-либо дополнительной памяти и дополнительных ухищрений. Столь же легко теперь будет разобраться, где находится вершина: если  $Q[i] = -2$ , то – вне очереди, а в остальных случаях – в очереди. Пустую очередь от непустой мы отличим по значению переменной  $h\_Q$  – если очередь пуста, то  $h\_Q = -1$ .

Теперь осталось написать программу, которая, как вы вскоре убедитесь, крайне проста.

```

//ПРОГРАММА ПОСТРОЕНИЯ ДЕРЕВА КРАТЧАЙШИХ ПУТЕЙ
//НА СЕТИ С ПРОИЗВОЛЬНЫМИ ДЛИНАМИ ДУГ
//БЕЗ ОТРИЦАТЕЛЬНЫХ ЦИКЛОВ
//АЛГОРИТМ БЕЛЛМАНА – ФОРДА

//Инициализация
//Построение списков ребер Н и L (раздел 2)
. . . . .
//Списки построены
R[0..n-1] = ∞ // Все текущие расстояния
                // равны ∞
P[0..n-1] = -2 // Все вершины недоступны

R[s] = 0;           // s – исходная вершина
P[s] = -1;          // У нее нет предка
// Инициализация очереди
Q[0..n-1] = -2 // Все вершины вне очереди
h_Q = s;           // Исходную вершину – в начало
                  // очереди
t_Q = s;           // Она же в очереди последняя
Q[s] = -1;
// Инициализация закончена

// ОСНОВНОЙ ЦИКЛ

```

```

while ( h_Q != -1 ) // Пока очередь не пуста
{
    i = h_Q;           // Берем очередную вершину
                        // для просмотра
    h_Q = Q[h_Q];    // Передвигаем начало очереди
    Q[i] = -2;        // Теперь вершина вне очереди

    // Просмотр дуг, выходящих из вершины i
    for ( k = H[i]; k != -1; k = L[k] )
    {
        j = J[k]; // Противоположная вершина
        // ПРОВЕРКА ОСНОВНОГО СООТНОШЕНИЯ
        xj = R[j];
        if ( R[i] + C[k] < xj )
        { // Основное соотношение нарушено
            R[j] = R[i] + C[k]; //Новое текущее
                                //расстояние до j
            P[j] = k; //Последняя дуга на пути
                        //из s в j

        *
        *     if ( Q[j] == -2 ) // j вне очереди
        *     { // Добавляем j в конец очереди
        *         if ( h_Q != -1 )// очередь не пуста
        *             Q[t_Q] = j;
        *         else           // очередь пуста
        *             h_Q = j;
        *             // а это надо делать всегда
        *             t_Q = j;
        *             Q[j] = -1;
        *         }
        }

        // Если основное соотношение выполнено,
        // то ничего делать не надо
    } // Конец цикла просмотра дуг
} // Конец основного цикла и всей программы

```

Удивительно, но такая простая программа очень неплохо работает, правда, на не слишком больших сетях. Если число вершин сети

не превышает 1000, то с помощью этой программы можно строить дерево кратчайших путей.

Интересно, что если совсем немного изменить алгоритм, а с ним и программу, то их можно очень сильно «испортить». Если для выбора вершин вместо очереди использовать стек, то на средних и больших сетях модифицированная программа будет работать в сотни и даже в тысячи раз медленней. Каждый сможет убедиться в этом сам, если в приведенной программе заменит несколько операторов, помеченных символом «\*», на следующие:

```
if ( Q[j] == -2 ) // j вне стека
{ // Добавляем j в стек
    Q[j] = h_Q;
    h_Q = j;
}
```

При этом переменная  $t\_Q$  оказывается не нужна, а программа становится короче. Эффект кардинального ухудшения качества от таких небольших изменений производит сильное впечатление. Оказывается, это имеет под собой теоретическую базу. Можно показать, что алгоритм со стеком в худшем случае имеет экспоненциальную оценку сложности в отличие от алгоритма Беллмана – Форда, оценка сложности которого полиномиальна. И в данном случае мы наблюдаем замечательный эффект: оценка алгоритма Беллмана – Форда намного лучше, чем оценка алгоритма со стеком, и работает он также значительно лучше. В этом многое правды, но, увы, не вся. Теоретические оценки часто, но, к сожалению, не всегда дают представление о качестве алгоритма. Пример обратного будет проиллюстрирован в следующем подразделе.

## 7.6. Алгоритм двусторонней очереди

Будучи достаточно эффективным при работе на небольших сетях, при увеличении размеров сетей алгоритм Беллмана – Форда начинает работать медленно. Одной из причин этого является «неправильный» порядок просмотра вершин. Предположим, что некоторая вершина  $i$  была просмотрена и в результате этого просмотра у каких-то вершин уменьшились текущие расстояния. Назовем эти вершины потомками  $i$ . Эти вершины попали в очередь на просмотр. Пока они находились в очереди, уменьшилось текущее расстояние у вершины  $i$ . Она опять попала в очередь, но уже после своих потом-

ков. Сейчас текущие расстояния у ее потомков заведомо неправильные, потому что если бы мы просмотрели дуги, выходящие из  $i$  сразу, то текущие расстояния у потомков уменьшились. Тем не менее, в порядке очереди сначала будут просмотрены потомки, которые породят новые неверные расстояния, и только потом будет просмотрена вершина  $i$ . Можно сказать, что по сети пойдет «волна ложных расстояний». Конечно, потом эта волна будет погашена, но лишнего времени будет потрачено немало.

В конце 60-х годов Левитом и, независимо от него, Пэйпом было предложено эвристическое улучшение алгоритма Беллмана – Форда, которое сглаживало влияние волн ложных расстояний. Этот алгоритм называется *алгоритмом двусторонней очереди*, и он оказывается весьма эффективным при решении реальных задач, зачастую работая намного быстрее алгоритма Беллмана – Форда.

Улучшение состоит в том, что если у вершины, которая уже хотя бы один раз была просмотрена и в данный момент находится вне очереди, уменьшается текущее расстояние, то такую вершину мы ставим не в конец, а в начало очереди. Эта стратегия позволяет быстрее гасить волны ложных расстояний в результате того, что быстрее начинается исправление «совершенных ошибок».

Чтобы получить программу, реализующую алгоритм двусторонней очереди, достаточно в приведенной выше программе заменить фрагмент, помеченный «\*», на следующий:

```
if ( Q[j] == -2 ) // j вне очереди
{ // Добавляем j в двустороннюю очередь
    if ( rj == ∞ )
        { // Добавляем в конец очереди
            if ( h_Q != -1 ) // очередь не пуста
                Q[t_Q] = j;
            else // очередь пуста
                h_Q = j;
            // а это надо делать всегда
            t_Q = j;
            Q[j] = -1;
        }
    else
        { // j уже была в очереди,
            // добавляем в начало
            Q[j] = h_Q;
```

```
    if ( h_Q == -1 ) t_Q = j;
    h_Q = j;
}
}
```

Как видите, программа почти не усложнилась, а при решении практических задач работает значительно быстрее. К сожалению, этот эффект никак не объяснен теоретически. Дело в том, что оценка сложности алгоритма двусторонней очереди в худшем случае экспоненциальна. Тем не менее при решении практических задач этот алгоритм работает очень хорошо. Этот пример, увы, демонстрирует несовершенство теории. Конечно, при решении более сложных и объемных задач алгоритм двусторонней очереди иногда начинает проявлять свою «экспоненциальную сущность», но все же во многих случаях он работает очень хорошо, что никак не согласуется с его теоретической оценкой. Так что последние два примера продемонстрировали нам и пользу теории, и одновременно ее ограниченность. Приходится констатировать, что в настоящее время во многих случаях для комбинаторных алгоритмов «только практика является критерием истины», хотя значение теории ни в коем случае не следует отрицать.

Вообще говоря, использование двусторонней очереди – это попытка угадать (или аппроксимировать) «правильный» порядок просмотра вершин. Ведь если бы мы смогли угадать порядок так, что просматривали бы вершины в порядке возрастания их ранга, то каждую вершину нужно было бы просмотреть ровно один раз. К сожалению, это не более чем мечта типа «знал бы прикуп – жил бы в Сочи», и мы никак не можем предугадать правильный порядок просмотра вершин до того как построим дерево кратчайших путей. Однако попытки попробовать угадывать порядок на основании имеющейся информации иногда приводят к неплохим результатам. Так, очень хорошо работает предложенный относительно недавно алгоритм Гольдберга – Радзика, в котором используется метод «топологической сортировки» вершин на очередном этапе. Мы не будем рассматривать этот алгоритм, поскольку и так уже слишком много времени уделили задаче построения дерева кратчайших путей.

## 7.7. Некоторые обобщения задачи построения дерева кратчайших путей

Если надо строить дерево кратчайших путей на неориентированной сети, то можно поступить двояко. Например, слегка изменить рассмотренные ранее программы, приспособив их к какой-то кодировке неориентированной сети. В частности, к кодировке, приведенной в разделе 2. При этом необходимые изменения программ будут минимальными. Если же не хочется делать даже это, то можно заменить неориентированную сеть на ориентированную. Для этого достаточно каждое неориентированное ребро  $[i, j]$  длиной  $c$  заменить на две ориентированные дуги:  $(i, j)$  и  $(j, i)$  – обе длиной  $c$ . Очевидно, что после такой замены новая задача будет эквивалентна предыдущей. Правда, при этом число дуг удвоится.

Вообще говоря, в жизни кратчайшие пути не всегда самые подходящие, и иногда необходимо строить оптимальные пути исходя из других критериев. Предположим, что на дугах задана не длина, а пропускная способность  $b(a)$ . Тогда пропускной способностью пути  $W = (a_1, \dots, a_k, \dots, a_l)$  является минимум пропускных способностей входящих в него дуг:  $b(W) = \min_{k=1, \dots, l} b(a_k)$ . Оказывается, что «самые широкие» пути можно искать с помощью практически таких же алгоритмов, с помощью которых строятся и кратчайшие пути. Разница состоит только в том, что при сравнении двух путей надо проверять другое соотношение. Если раньше  $r_i$  и  $r_j$  обозначали текущие расстояния до вершин, то теперь они обозначают текущие пропускные способности. В этом случае при просмотре дуги  $a_{ij}$  придется сравнивать  $r_j$  и  $\min(r_i, b(a_{ij}))$ . Если  $\min(r_i, b(a_{ij})) > r_j$ , то новый путь шире,  $r_j$  увеличится, и самый широкий путь в вершину  $j$  будет теперь проходить через  $i$ .

В остальном алгоритмы и программы построения дерева самых широких путей останутся такими же, как и для кратчайших путей, за исключением того что при инициализации в качестве начальной пропускной способности (при неотрицательных пропускных способностях) всем вершинам, кроме  $s$ , надо назначать ноль, а исходной вершине  $s$  – «бесконечность».

Еще одним примером применения подобных алгоритмов является поиск «самого безопасного пути». Предположим, что на каждой дуге  $a$  задано число  $0 \leq p(a) \leq 1$  – вероятность аварии при проезде по

этой дуге. Тогда наиболее безопасные пути можно строить, пользуясь приведенными выше программами и алгоритмами, с учетом того что вероятность неблагополучного исхода при выборе пути

$$W = (a_1, \dots, a_k, \dots, a_l) \text{ равна } p(W) = 1 - \prod_{k=1}^l (1 - p(a_k)), \text{ и на опти-}$$

мальном пути эту вероятность надо минимизировать. При решении этой задачи, проверяя основное соотношение, нужно будет сравнивать  $r_j$  и  $1 - (1 - r_i)(1 - p(a_{ij}))$  и выбирать минимальную из этих величин.

Вообще говоря, существуют и многие другие модификации, обобщения и усложнения задачи построения дерева кратчайших путей, но мы их рассматривать не будем.

## 8. ЗАДАЧА ПОСТРОЕНИЯ КРАТЧАЙШИХ ПУТЕЙ (АЛГОРИТМ ДЕЙКСТРЫ)

### 8.1. Особенности алгоритма Дейкстры

Посвятим еще один раздел задаче нахождения кратчайших путей. В ней мы разберем классический алгоритм Дейкстры<sup>1</sup>, с помощью которого можно строить дерево кратчайших путей на сети с неотрицательными длинами дуг. Оценка сложности алгоритма Дейкстры, даже в наивном варианте, лучше оценки сложности алгоритма Беллмана – Форда, но это не единственная причина, по которой стоит в нем разобраться. В более изощренной реализации алгоритма Дейкстры используется структура данных, называемая *черпаками*, и с этой структурой тоже будет полезно познакомиться.

К тому же оценка сложности алгоритма Беллмана – Форда  $O(nm)$  весьма далека от идеальной  $O(m)$ , которой удалось достичь на сетях с единичными длинами дуг. Поэтому хотелось бы иметь алгоритмы с более низкой оценкой сложности.

Алгоритм Дейкстры является конкретизацией общего алгоритма построения дерева кратчайших путей. Для просмотра мы выбираем некоторую перспективную вершину и обрабатываем все выходящие из нее дуги. В алгоритме Дейкстры, как и в других алгоритмах, близких к методу поиска в ширину, можно выделить три множества:  $S$  – просмотренные вершины,  $T$  – непомеченные и  $Q$  – помеченные, но непросмотренные. Однако правило выбора вершины для просмотра не такое, как в алгоритме Беллмана – Форда. Из множества  $Q$  для просмотра выбирается «самая лучшая вершина», а именно: вершина с наименьшим текущим расстоянием среди всех вершин, находящихся в множестве  $Q$ .

#### *Описание алгоритма*

Алгоритм Дейкстры строит дерево кратчайших путей от заданной вершины  $s$  до всех достижимых из  $s$  вершин на сети с неотрицательными длинами дуг.

**Инициализация.** См. алгоритм Беллмана – Форда.

---

<sup>1</sup> Это тот самый Дейкстра, который в свободное от структурного программирования время иногда придумывал алгоритмы.

**Выбор вершины.** Из множества  $Q$  выбираем для просмотра вершину  $i$  с минимальным текущим расстоянием  $r_i$ .

Если множество  $Q$  пустое – работа алгоритма заканчивается – дерево кратчайших расстояний построено.

Удаляем  $i$  из множества  $Q$ .

**Просмотр дуг.** См. алгоритм Беллмана – Форда.

**Улучшение дерева.** См. алгоритм Беллмана – Форда с поправкой на то, что множество  $Q$  не является очередью.

Как и было обещано, алгоритм Дейкстры отличается от алгоритма Беллмана – Форда только способом выбора вершины для просмотра. Тем не менее свойства этих алгоритмов различны.

## 8.2. Обоснование алгоритма Дейкстры

Докажем, что текущее расстояние  $r_i$  до вершины  $i$ , выбираемой для просмотра, равно кратчайшему расстоянию от  $s$  до этой вершины. Будем доказывать это утверждение индукцией по порядку просмотра вершин.

Для первой просматриваемой вершины  $s$  утверждение очевидно: при неотрицательных длинах дуг расстояние до нее не может быть меньше нуля.

Предположим, что утверждение верно для всех вершин, просмотренных раньше, чем  $i$ . Докажем, что для вершины  $i$  оно также справедливо.

Перед выбором для просмотра вершины  $i$  текущее дерево кратчайших путей состояло из уже просмотренных вершин (множество  $S$ ) и вершин помеченных, но непросмотренных (множество  $Q$ ). По предположению индукции текущие расстояния вершин множества  $S$  равны кратчайшим и, следовательно, часть дерева, состоящая из дуг, входящих в просмотренные вершины, – это фрагмент итогового дерева кратчайших путей, и величина кратчайшего расстояния до любой вершины  $j \in S$  равна длине пути, идущего из  $s$  в  $j$ , в этом фрагменте дерева.

Кроме просмотренных, в текущем дереве кратчайших расстояний находятся помеченные, но непросмотренные вершины, в совокупности образующие множество  $Q$ . В каждую вершину множества  $Q$  ведет дуга из просмотренной вершины, поэтому все пути до таких вершин в дереве текущих путей состоят из начального участка пути, находящегося в  $S$ -фрагменте дерева, и последней дуги, ведущей из

просмотренной в помеченную вершину. При этом текущие расстояния до помеченных, но непросмотренных вершин равны длинам этих путей. В будущем эти расстояния могут уменьшиться, а пути, ведущие в помеченные, но непросмотренные вершины – измениться, то есть дуги, изображенные на рис. 8.1 сплошными линиями, войдут в итоговое дерево кратчайших путей, а дуги, изображенные пунктиром – не обязательно.

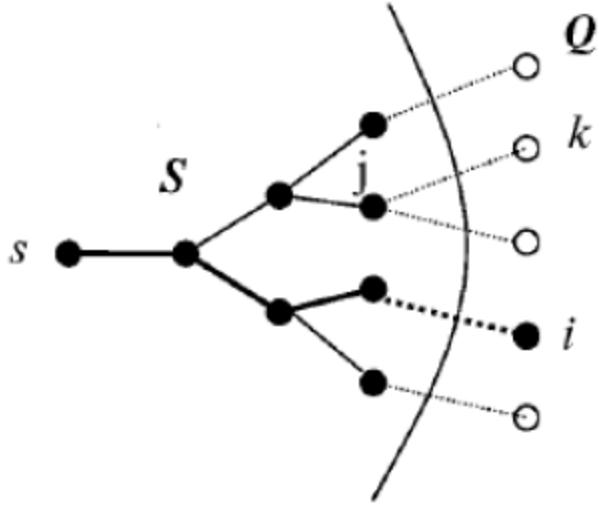


Рис. 8.1

Среди всех путей, ведущих из  $s$  в какую-либо помеченную, но непросмотренную вершину  $k$ , таких, что все вершины этих путей, кроме последней вершины  $k$ , принадлежат множеству  $S$ , самым коротким является путь, состоящий из дуг текущего дерева кратчайших путей. Пусть  $W = (s, \dots, j, k)$  – такой путь. Кратчайший путь из  $s$  в  $j$  по предположению индукции уже

построен и состоит из дуг текущего дерева. Поскольку все вершины множества  $S$  просмотрены, а текущее расстояние до вершины  $k$  получено при просмотре дуг, выходящих из  $j$ , то остальные пути такого типа длиннее. Таким образом, величина  $r_k$  для любой помеченной, но непросмотренной вершины  $k$  не больше длины любого пути из  $s$  в  $k$ , все вершины которого, кроме последней, принадлежат множеству  $S$ .

Среди всех вершин множества  $Q$  у выбираемой для просмотра вершины  $i$  текущее расстояние минимально. Докажем, что оно равно кратчайшему. Возьмем какой-то путь  $W$  из  $s$  в  $i$ . Пусть  $k$  – первая вершина на этом пути, не входящая в множество  $S$ . Разобьем путь  $W$  на два участка:  $W[s..k]$  от  $s$  до  $k$  и  $W[k..i]$  от  $k$  до  $i$  (рис. 8.2). Только что было доказано, что длина первого участка не меньше, чем величина текущего расстояния вершины  $k$ :  $c(W[s..k]) \geq r_k$ . По предположению длины дуг сети неотрицательны, следовательно  $c(W[k..i]) \geq 0$ . Значит,  $c(W) = c(W[s..k]) + c(W[k..i]) \geq r_k$ . Но в соответствии с выбором вершины  $r_k \geq r_i$ . Поэтому длина любого пути из  $s$  в  $i$  не меньше, чем  $r_i$ . Итак, мы доказали, что величина текущего расстояния у выбранной вершины равна величине кратчайшего расстояния.

Непосредственно из последнего утверждения следует корректность алгоритма Дейкстры. Легко видеть, что каждая достижимая вершина в ходе работы алгоритма будет помечена и затем просмотрена. Следовательно, после завершения работы алгоритма у всех достижимых вершин кратчайшие расстояния будут вычислены правильно, а построенное дерево будет деревом кратчайших путей.

Нам понадобится еще одно утверждение, относящееся к просматриваемым вершинам. Обозначим через  $r^k$  текущее расстояние до  $k$ -й по порядку просматриваемой вершины. Докажем, что  $r^0 \leq r^1 \leq \dots \leq r^{k-1} \leq r^k \leq \dots$ , то есть текущие расстояния просматриваемых вершин возрастают по ходу работы алгоритма. Будем доказывать это утверждение индукцией по порядку просматриваемой вершины. Расстояние

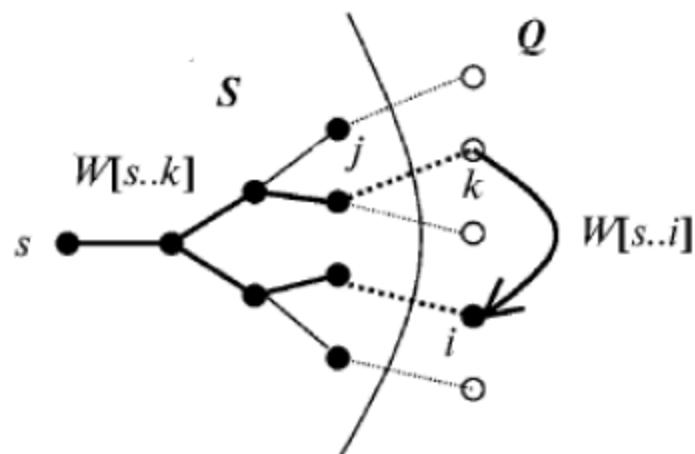


Рис. 8.2

$r^0 \leq r^1 \leq \dots \leq r^{k-1}$ , потому что  $r^0 = 0$ , а длины дуг сети неотрицательны. Предположим, что  $r^{k-1}$  больше или равно текущим расстояниям до всех ранее просмотренных вершин. Докажем, что  $r^{k-1} \leq r^k$ .

Пусть на  $(k - 1)$ -м шаге была просмотрена вершина  $i$ , а на  $k$ -м – вершина  $j$ . Если во время просмотра дуг, выходящих из  $i$ , текущее расстояние до вершины  $j$  не изменилось, то  $r^{k-1} = r_i \leq r_j = r^k$ , так как на  $(k - 1)$ -м шаге была выбрана вершина с минимальным текущим расстоянием, то есть  $r_i$  было не больше, чем  $r_j$ . Если же на  $(k - 1)$ -м шаге текущее расстояние до вершины  $j$  уменьшилось, то это случилось во время просмотра дуги  $(i, j)$  и, стало быть,  $r^k = r_j^{new} = r_i + c(i, j) \geq r_i = r^{k-1}$ , то есть и в этом случае  $r^{k-1} \leq r^k$ . Утверждение доказано.

Отметим еще одно свойство алгоритма Дейкстры, непосредственно следующее из доказанных утверждений. Если нам надо построить не все дерево кратчайших путей, а только путь между двумя заданными вершинами  $i$  и  $j$ , то в отличие от алгоритма Беллмана – Форда можно остановиться, не дожидаясь того времени, когда множество  $Q$  станет пустым, а в тот момент, когда для просмотра будет

выбрана вершина  $j$ , текущее расстояние до которой, как было только что доказано, уже равно минимальному.

Из обоснования алгоритма Дейкстры становится ясно, почему этот алгоритм применим только для сетей с неотрицательными длинами дуг. Основное достоинство этого алгоритма состоит в том, что один раз просмотренную вершину больше просматривать не придется. Это следует из того, что расстояние до любой просматриваемой вершины равно минимальному и больше не изменится. При доказательстве этого факта мы использовали утверждение, что длина пути  $W[k,j]$  неотрицательна, что верно только при неотрицательных длинах дуг. Если же в сети встречаются дуги с отрицательными длинами, то нет никаких гарантий, что одну вершину мы не будем просматривать несколько раз. Нетрудно придумать соответствующие примеры.

### 8.3. Оценка сложности алгоритма Дейкстры

Будем, как и раньше, обозначать число вершин сети через  $n$ , а число дуг сети – через  $m$ . Мы доказали, текущее расстояние до просматриваемой вершины равно минимальному, значит, уменьшиться оно уже не может и, следовательно, в алгоритме Дейкстры каждая вершина просматривается не более одного раза. Отсюда следует, что и любая дуга будет просмотрена не более одного раза. Следовательно, количество операций, которое будет затрачено на просмотр дуг, имеет порядок  $O(m)$ .

Осталось посчитать, какое количество действий потребуется для выполнения операции с вершинами: включение вершины в множество  $Q$  и выбор вершины с минимальным текущим расстоянием. Если не прибегать к использованию более сложных структур данных, а ограничиться представлением множества  $Q$  в виде списка, то включение вершины в множество  $Q$  потребует  $O(1)$  действий, а выбор минимума в худшем случае – порядка  $O(n)$  действий, потому что придется просмотреть все вершины множества  $Q$ . Минимум выбирается  $n$  раз, поэтому количество действий, затраченное на операции с вершинами, имеет порядок  $O(n^2)$ .

Таким образом, общая оценка сложности алгоритма Дейкстры равна  $O(m) + O(n^2) = O(n^2)$ . На плотных сетях, для  $m \approx n^2$ , эта оценка на порядок лучше оценки сложности  $O(nm)$  алгоритма Беллмана – Форда. И это в наивной реализации алгоритма Дейкстры.

ры, без приложения усилий для того, чтобы как-то ускорить процедуру выбора минимума. Вообще говоря, в этом алгоритме мы частично достигли идеала: каждая вершина и дуга просматривается ровно один раз. Но за все приходится платить, и мы расплачиваемся лишней работой для «правильного» выбора вершины.

Прежде чем пытаться улучшать алгоритм Дейкстры, приведем его наивную реализацию.

#### 8.4. Наивная реализация алгоритма Дейкстры

Удивительно, но структура данных, необходимая для наивной реализации алгоритма Дейкстры, почти такая же, как и для алгоритма Беллмана – Форда. Единственное отличие состоит в том, что множество  $Q$  у Дейкстры не является очередью, поэтому нам не понадобится запоминать последнюю его вершину.

```
//ПРОГРАММА ПОСТРОЕНИЯ ДЕРЕВА КРАТЧАЙШИХ ПУТЕЙ
//НА СЕТИ С НЕОТРИЦАТЕЛЬНЫМИ ДЛИНАМИ ДУГ
//АЛГОРИТМ ДЕЙКСТРЫ (НАИВНАЯ РЕАЛИЗАЦИЯ)

//Инициализация
//Построение списков ребер Н и Л (раздел 2)
. . . . .
//Списки построены
R[0..n-1] = ∞ // Все текущие расстояния
                // равны ∞
P[0..n-1] = -2 // Все вершины недоступны

R[s] = 0;          // s - исходная вершина
P[s] = -1;         // У нее нет предка
// Инициализация множества Q
h_Q = s;           // Исходную вершину - в Q
Q[s] = -1;
// Инициализация закончена

// ОСНОВНОЙ ЦИКЛ
while ( h_Q != -1 ) // Пока очередь не пуста
{
//ВЫБОР ВЕРШИНЫ С МИНИМАЛЬНЫМ ТЕКУЩИМ РАССТОЯНИЕМ
min = ∞;          // Текущий минимум
//j - текущая вершина при проходе по списку
```

```

//pj - предыдущая при проходе по списку
for ( pj = -1, j = h_Q;
      j != -1;
      pj = j, j = Q[j] )
if ( R[j] < min )
{ min = R[j]; //новый минимум
  pi = pj; //достигается на вершине,
             //следующей за pi
}

//Вершину Q[pi], на которой достигается
//минимум, выбираем для просмотра и
//удаляем из Q
if ( pi != -1 )
{ // i - не первая в списке
  i = Q[pi];
  Q[pi] = Q[i]; // Удаляем i из списка
}
else
{ // i - первая в списке
  i = h_Q;
  h_Q = Q[i]; //Передвигаем начало списка
}

// Просмотр дуг, выходящих из вершины i
for ( k = H[i]; k != -1; k = L[k] )
{
  j = J[k]; // Противоположная вершина
  // ПРОВЕРКА ОСНОВНОГО СООТНОШЕНИЯ
  rj = R[j];
  if ( R[i] + C[k] < rj )
  { // Основное соотношение нарушено
    R[j] = R[i] + C[k]; //Новое текущее
                          //расстояние до j
    P[j] = k; //Последняя дуга на пути
               //из s в j
    if ( rj == infinity ) // j была не помечена
    { // Добавляем j в Q
      Q[j] = h_Q;
      h_Q = j;
    }
  }
}

```

```

    }
}

// Если основное соотношение выполнено,
// то ничего делать не надо
} // Конец цикла просмотра дуг
} // Конец основного цикла и всей программы

```

Как видите, программа, представляющая собой наивную реализацию алгоритма Дейкстры, очень проста. У нее всего один, но существенный недостаток – она медленно работает. Несмотря на то что оценка сложности алгоритма Дейкстры –  $O(n^2)$  – лучше, чем у алгоритма Беллмана – Форда –  $O(pt)$ , при решении реальных задач картина прямо противоположная. Наивная реализация алгоритма Дейкстры работает значительно медленнее, чем программа, реализующая алгоритм Беллмана – Форда. Причина проста. Мы слишком много времени тратим на выбор вершины. Если избавиться от этого недостатка, то алгоритм Дейкстры начинает работать намного лучше.

Существует много разных способов выбора минимального элемента из подмножества. Один из таких способов мы рассмотрим впоследствии. Это структура данных под названием *куча* (*heap*). Алгоритм Дейкстры, реализованный с помощью кучи, работает неплохо. Но в этом разделе мы рассмотрим другую полезную структуру данных, называемую *черпаками* (*buckets*), которая тоже позволяет значительно ускорить работу алгоритма Дейкстры.

## 8.5. Черпаки

Сначала опишем черпаки как абстрактную структуру данных, с помощью которой можно выполнять определенные операции. Будем для простоты считать, что все длины дуг и, соответственно, все текущие расстояния являются целыми числами. В одном черпаке  $B_r$  будут находиться все помеченные и непросмотренные вершины, текущее расстояние до которых сейчас в данный момент равно  $r$ .

Чтобы реализовать алгоритм Дейкстры с помощью черпаков, надо уметь эффективно выполнять три операции:

**INSERT** (  $i, r$  ) – поместить вершину  $i$  в черпак  $r$ ;  
**REMOVE** (  $i, r$  ) – удалить вершину  $i$  из черпака  $r$ ;

`GET (  $x$  )` – взять любую вершину из черпака  $r$  и удалить ее оттуда.

Через некоторое время мы убедимся, что можно сделать так, чтобы все эти операции выполнялись за время порядка  $O(1)$ . А пока будем это предполагать.

На основе сделанных предположений реализовать алгоритм Дейкстры с помощью черпаков совсем несложно. При выборе для просмотра вершины с минимальным текущим расстоянием пробуем при помощи операции `GET` взять какую-нибудь вершину из того черпака  $B_r$ , из которого была взята вершина в прошлый раз. Если черпак не пуст, то берем из него вершину и начинаем ее просмотр, а если пуст, то переходим к черпаку  $B_{r+1}$ . И так далее, пока не закончатся все черпаки. В самый первый раз берем вершину из черпака  $B_0$ .

То, что таким способом будет правильно выбрана вершина с минимальным текущим расстоянием непосредственно следует из определения черпаков и из доказанного ранее утверждения, что текущие расстояния до просматриваемых вершин не убывают по ходу работы алгоритма. Это означает, что при просмотре данного черпака мы можем быть уверены, что непросмотренные вершины не могут находиться в черпаках с меньшим номером и, следовательно, для просмотра будет выбрана вершина с минимальным текущим расстоянием.

Когда вершина получит пометку, нужно с помощью операции `INSERT` поместить ее в черпак, соответствующий найденному текущему расстоянию. Если текущее расстояние до вершины уменьшилось, то с помощью двух операций – `REMOVE` и `INSERT` – перемещаем ее из черпака, соответствующего прежнему текущему расстоянию, в черпак, соответствующий новому текущему расстоянию.

Таким образом, перечисленных операций с черпаками хватает на то чтобы реализовать алгоритм Дейкстры.

## 8.6. Оценка сложности алгоритма Дейкстры с черпаками

Чтобы найти сложность алгоритма с черпаками, оценим сначала максимально возможное число черпаков, которое придется просмотреть в ходе работы алгоритма. По определению, черпак соответствует величине текущего расстояния. Это значит, что нам не понадобится черпаков больше, чем величина самого длинного пути, ведущего в какую-нибудь вершину. В таком пути не больше чем  $(n - 1)$  дуг, где  $n$  –

число вершин в сети, и, стало быть, его длина заведомо меньше, чем  $nC$ , где  $C$  – максимальная длина дуги. Таким образом, нам не придется просматривать более чем  $nC$  черпаков.

Будем предполагать, что все операции с черпаками требуют порядка  $O(1)$  действий. Тогда просмотр вершин и ребер обойдется нам, как и раньше, в  $O(m)$  действий, поскольку максимум лишних операций, которые мы можем выполнить при просмотре ребра – это операции REMOVE и INSERT, а по предположению их сложность равна  $O(1)$ .

Вместо поиска минимальной вершины мы берем ее из черпака. То есть на выбор минимума вместо  $O(n^2)$  действий в наивном варианте алгоритма Дейкстры будет потрачено столько действий, сколько раз придется выполнить операцию GET. В результате выполнения этой операции  $n$  раз будут выбраны вершины для просмотра, а при выполнении остальных операций мы будем убеждаться, что очередной черпак пуст, и переходить к следующему. Таким образом, всего мы выполним операцию GET не более, чем  $(n + nC)$  раз, так как выше было показано, что максимальное число черпаков не превышает  $nC$ . Отсюда следует, что число действий, нужных для выбора минимальной вершины, будет иметь порядок  $O(nC)$ , а следовательно, сложность всего алгоритма составит  $O(m + nC)$ .

На первый взгляд мы только ухудшили оценку алгоритма. По сравнению с наивным вариантом вместо  $n$  в формуле появилась переменная  $C$ , которая может быть очень велика. Более того, оценка алгоритма перестала быть полиномиальной и стала экспоненциальной. С теоретической точки зрения так оно и есть. Однако на практике часто случается, что  $C$  не слишком велика, и опыт показывает, что при решении реальных задач алгоритм Дейкстры с черпаками в большинстве случаев работает значительно быстрее наивной реализации алгоритма Дейкстры, а во многих случаях гораздо быстрее и алгоритма Беллмана – Форда, и даже алгоритма двусторонней очереди.

С теоретической точки зрения все тоже обстоит не так уж плохо. Дальнейшие усовершенствования структуры черпаков приводят, при сохранении и повышении эффективности работы программы, к полиномиальным оценкам:  $R$ -кучи, многоуровневые черпаки. Но все это выходит за рамки данного курса лекций, а нам остается только констатировать, что, несмотря на теоретические «изъяны», на практике алго-

ритм Дейкстры с черпаками часто работает очень неплохо и в ряде случаев его можно рекомендовать для практического использования.

## 8.7. Реализация алгоритма Дейкстры с черпаками

Для начала приведем процедуру, реализующую алгоритм Дейкстры с черпаками, без расшифровки процедур REMOVE, INSERT и GET. В следующем подразделе мы убедимся, что эти процедуры также крайне просты. Структура данных остается почти такой же, как и для наивной реализации. Лишним оказывается только массив  $Q$  для хранения очереди, а структуру данных, которая понадобится для хранения черпаков, добавим потом.

```
//ПРОГРАММА ПОСТРОЕНИЯ ДЕРЕВА КРАТЧАЙШИХ ПУТЕЙ
//НА СЕТИ С НЕОТРИЦАТЕЛЬНЫМИ ДЛИНАМИ ДУГ
//АЛГОРИТМ ДЕЙКСТРЫ С ЧЕРПАКАМИ

//Инициализация
//Построение списков ребер H и L (раздел 2)
. . . . .
//Списки построены
R[0..n-1] = ∞ // Все текущие расстояния
                // равны ∞
P[0..n-1] = -2 // Все вершины недоступны

R[s] = 0;           // s - исходная вершина
P[s] = -1;          // У нее нет предка
// Инициализация черпаков
M = n * C          // Максимальное число черпаков
Bucket[0..M] = Ø; // Черпаки пусты
INSERT ( s, 0 ); // Исходную вершину - в
                  // нулевой черпак
// Инициализация закончена

// ЦИКЛ ПРОСМОТРА ЧЕРПАКОВ
for ( b = 0; b <= M; b++ )

//ВЫБОР ВЕРШИН С МИНИМАЛЬНЫМ РАССТОЯНИЕМ
//ИЗ ТЕКУЩЕГО ЧЕРПАКА, ПОКА ОН НЕ СТАНЕТ ПУСТЫМ
```

```

while ( ( i = GET ( b ) ) != -1 )

    // Просмотр дуг, выходящих из вершины i
    for ( k = H[i]; k != -1; k = L[k] )
    {
        j = J[k]; // Противоположная вершина
        // ПРОВЕРКА ОСНОВНОГО СООТНОШЕНИЯ
        rj = R[j];
        if ( R[i] + C[k] < rj )
        { // Основное соотношение нарушено
            R[j] = R[i] + C[k]; //Новое текущее
                                //расстояние до j
            P[j] = k; //Последняя дуга на пути
                        //из s в j
            if ( rj != ∞ ) // j помечена и
                            //находится в черпаке rj
                REMOVE ( j, rj ); // удаляем j из
                                    // черпака rj
            INSERT ( j, R[j] ); // помещаем j в
                                // новый черпак
        }
        // Если основное соотношение выполнено,
        // то ничего делать не надо
    } // Конец цикла просмотра дуг
// Конец основного цикла и всей программы

```

Как видите, программа, реализующая алгоритм Дейкстры с черпаками, проще наивной реализации. Осталось только убедиться в том, что программная реализация черпаков столь же проста.

## 8.8. Реализация операций с черпаками

Эффективно реализовать операции с черпаками не просто, а очень просто. Представим черпак в виде двухстороннего списка. Голова  $k$ -го черпака хранится в переменной  $B[k]$ , так что для хранения голов нам понадобится массив  $B[0..M]$ . Ссылки «вперед» и «назад» будем хранить в двух массивах:  $Fw[0..n-1]$  и  $Bk[0..n-1]$ ,  $Fw[i]$  – номер следующей после  $i$  вершины в том же черпаке, а  $Bk[i]$  – номер предыдущей. Поскольку в каждый момент времени вершина может находиться только в одном черпаке, то

этих двух массивов достаточно для того, чтобы хранить списки для всех черпаков.

При инициализации нужно в головы списков записать некоторую переменную, которая свидетельствует о том, что список пуст. Традиционно для этих целей мы используем «-1», то есть при инициализации черпаков надо написать:

```
B[0..M] = -1;
```

Операция GET – обычное удаление первого элемента списка:

```
// Взять вершину из черпака k
GET ( k )
{ i = B[k]; // первая вершина в черпаке или
    // -1, если черпак пуст
    if ( i != -1 ) B[k] = Fw[i];
        // удалили i из черпака
    return ( i );
}
```

Операция INSERT – добавление элемента в список. Как известно, добавлять проще всего в начало:

```
// Добавить вершину i в черпак k
INSERT ( i, k )
{ j = B[k]; // первая вершина в черпаке или
    // -1, если черпак пуст
    // Вставка элемента в начало двухстороннего
    // списка
    Fw[i] = j;
    if ( j != -1 ) Bk[j] = i;
    B[k] = i;
}
```

Операция REMOVE – удаление заданного элемента из двухстороннего списка:

```
// Удалить вершину i из черпака k
REMOVE ( i, k )
{ // Удаление элемента из двухстороннего
    // списка
```

```

fi = Fw[i]; // Следующая в черпаке
bi = Bk[i]; // Предыдущая в черпаке
if ( i == B[k] )
    B[k] = fi; // i была первой в черпаке
else
    { Fw[bi] = fi; // i была не первой

        if ( fi != -1 ) // i не последняя
            Bk[fi] = bi;
    }
}

```

Итак, мы убедились, что все операции с черпаками действительно требуют  $O(1)$  действий и, следовательно, оценка сложности алгоритма Дейкстры с черпаками, приведенная в предыдущем подразделе, соответствует действительности.

## 8.9. Улучшения реализации алгоритма Дейкстры с черпаками

Одной из негативных черт реализации алгоритма Дейкстры с черпаками является то, что требуется большой объем памяти для хранения черпаков: длина массива черпаков  $B$  равна  $nC$ , где  $n$  – число вершин сети, а  $C$  – максимальная длина дуги. Конечно, это много, но, оказывается, можно почти без усилий сократить требуемый объем памяти в  $n$  раз.

Докажем, что в каждый момент времени расстояние между самым близним и самым дальним непустыми черпаками не может быть больше  $C$ . Пусть в настоящий момент просматривается вершина из черпака  $k$ , то есть вершина, расстояние до которой равно  $k$ . Все предшествующие черпаки уже просмотрены, а значит, пусты и интереса не представляют. Выше было доказано, что расстояние до всех вершин, просматриваемых до этого момента, было не больше  $k$ . Все помеченные вершины находятся в черпаках, соответствующих их текущему расстоянию. Текущее расстояние до любой помеченной вершины  $j$  равно расстоянию до некоторой просмотренной вершины  $i$  плюс длина дуги  $(i, j)$ :  $R_j = R_i + c_{ij}$ . Но  $R_i \leq k$ , а  $c_{ij} \leq C$ . Следовательно,  $R_j \leq k + C$ , то есть если черпак не пуст, то он отстоит от просматриваемого черпака не больше чем на величину  $C$ . А это зна-

чит, что фактически в каждый момент времени нам будет достаточно  $C$  черпаков, то есть размер массива  $B$  можно сократить с  $nC$  до  $C$ .

Для того чтобы написать более экономную программу, достаточно знать правила, по которым устанавливается соответствие между номером черпака в массиве черпаков и текущим расстоянием, соответствующим этому черпаку.

Пусть, как и раньше,  $k$  обозначает расстояние до просматриваемой вершины. Через  $k_0$  обозначим величину «базы». Номер черпака  $b$ , в котором расположены просматриваемые вершины, находящиеся на расстоянии  $k$ , вычисляется по формуле  $b = k - k_0$ . Когда номер просматриваемого черпака станет равным  $C$ , увеличиваем базу на  $C$ , и номер просматриваемого черпака оказывается равным нулю.

Величину  $R$  – текущее расстояние до вершины – переводим в номер черпака по следующему алгоритму:

```
b = R - k0;  
if ( b >= C ) b = b - C;
```

При необходимости выполнения обратного преобразования, следует поступать так:

```
if ( b < k ) b = b + C;  
R = b + k0;
```

Наглядной иллюстрацией описанного преобразования могло бы стать длинное полотно массива  $B$ , «наматываемое» на барабан длиной  $C$ . Очевидно, что этот вариант алгоритма Дейкстры тоже не трудно реализовать используя уже знакомую нам технику.

## 9. КУЧИ

### 9.1. Кучи: свойства и основные операции

При написании программ часто необходимо уметь эффективно выбирать минимальный элемент из заданного множества. При этом множество не задано раз и навсегда, а изменяется: в него могут войти новые элементы и могут быть удалены старые, например, минимальный элемент. К тому же каждый элемент, входящий во множество, может уменьшиться или увеличиться.

Такая ситуация была рассмотрена в предыдущем разделе при реализации алгоритма Дейкстры. Из множества помеченных, но не просмотренных вершин нужно было выбирать вершину с минимальным текущим расстоянием и после просмотра делать ее помеченной. Новая вершина попадала в очередь на просмотр, когда она впервые получала пометку (конечное расстояние), а текущее расстояние до вершины по ходу работы алгоритма могло уменьшаться. В предыдущем разделе для решения этой задачи мы воспользовались *черпаками*. Эта структура данных красива и интересна. С ее помощью нам удалось получить эффективную на практике реализацию алгоритма Дейкстры, однако у нее есть недостатки.

Во-первых, предполагается, что все длины дуг – целые числа. Такое предположение, как правило, не очень обременительно, но иногда ограничивает возможности. Во-вторых, оценка сложности алгоритма получается не полиномиальной:  $O(m + nC)$ , где  $m$  – число дуг сети,  $n$  – число вершин, а  $C$  – максимальная длина дуги. Кроме того, при больших  $C$ , как мы видели, требуется значительный дополнительный объем памяти и начинает увеличиваться время работы алгоритма, то есть такая структура данных неплохая, но не универсальная. А хочется иметь не менее эффективный, но более «неприхотливый инструмент», который можно было бы применять не задумываясь.

*Куча (heap)* как раз и является такой структурой данных. С ее помощью можно эффективно выполнять следующие операции:

1. Выбор и удаление минимального элемента.
2. Добавление элемента.
3. Удаление элемента.
4. Увеличение элемента.
5. Уменьшение элемента.
6. Окучивание (*heapify*) – преобразование неупорядоченного массива данных в кучу.

Если куча содержит  $n$  элементов, то для выполнения операции 1–5 потребуется порядка  $O(\log n)$  действий, а для окучивания массива из  $n$  элементов понадобится порядка  $O(n)$  действий. Это означает, что основные операции при работе с кучей требуют логарифмического времени, а предварительная подготовка – линейного. Это совсем не плохо, хотя и хуже, чем при использовании черпаков, где каждая из операций 1–5 «обошлась» бы нам в константу –  $O(1)$  – действий, плюс необходимость просматривать пустые черпаки при выборе минимального элемента. Зато при использовании куч не накладывается никаких ограничений на величины элементов, не требуется дополнительная память, и кучи предельно просты в реализации. Такие свойства кучи делают ее часто используемой структурой данных при написании программ.

Куча представляется в виде *ровного* бинарного дерева, содержащего в каждой вершине некоторое число. Для чисел, находящихся в куче, должно выполняться «соотношение порядка».

Бинарное дерево называется *ровным*, если его уровни заполняются элементами последовательно сверху вниз и слева направо. В ровном бинарном дереве при передвижении по каждому уровню слева направо ни одна позиция не может быть пропущена. На рис. 9.1 изображены два дерева. Левое дерево ровное, а правое нет: в нем на третьем уровне отсутствует вторая слева вершина.

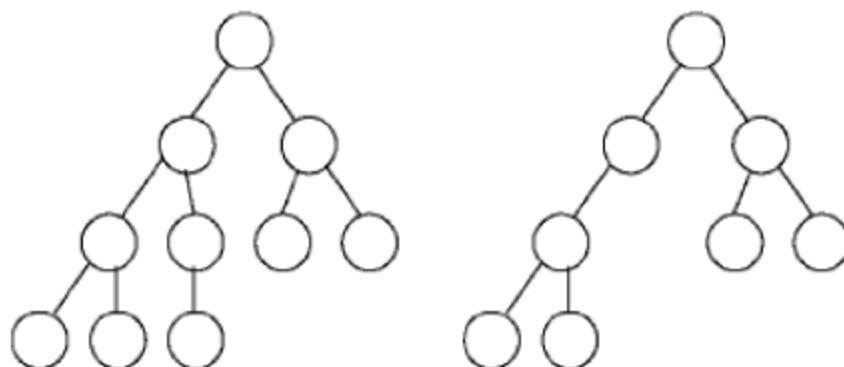


Рис. 9.1

У всех вершин ровного бинарного дерева, кроме вершин, находящихся на предпоследнем и на последнем уровнях, имеется по два потомка<sup>1</sup>. У вершин последнего уровня потомков нет. А у вершин

<sup>1</sup> Напомним, что потомками вершины дерева являются вершины, расположенные непосредственно после нее на пути из корня, а предком – вершина, находящаяся на пути из корня непосредственно перед данной. Вершины без потомков называются листьями. Корень – единственная вершина дерева, у которой нет предка.

предпоследнего уровня, вообще говоря, сначала имеется по два потомка, затем, может быть, найдется вершина с одним потомком, а после нее – вершины без потомков. Все эти варианты представлены на рис. 9.1 в левом дереве. При этом в предпоследнем слое может не быть вершин каких-либо из этих типов. Например, в предпоследнем слое могут быть вершины только с двумя потомками и без потомков или вершины только с двумя потомками и т.п.

*Соотношение порядка* должно соблюдаться в каждой вершине кучи, имеющей хотя бы одного потомка. Пусть у вершины  $k$  имеются потомки  $i$  и  $j$ . Обозначим через  $r_k, r_i, r_j$  числа, хранящиеся в этих вершинах. Чтобы ровное бинарное дерево было кучей, необходимо, чтобы в каждой вершине выполнялись следующие соотношения:  $r_k < r_i$  и  $r_k < r_j$ <sup>1</sup>.

На рис. 9.2 изображены два ровных бинарных дерева. В кружках находятся числа, хранящиеся в соответствующих вершинах. Нетрудно убедиться в том, что левое дерево является кучей: в каждой его вершине, имеющей потомков, выполняется соотношение порядка. Правое же дерево кучей не является. Жирным выделена пара вершин, для которых нарушено соотношение порядка.

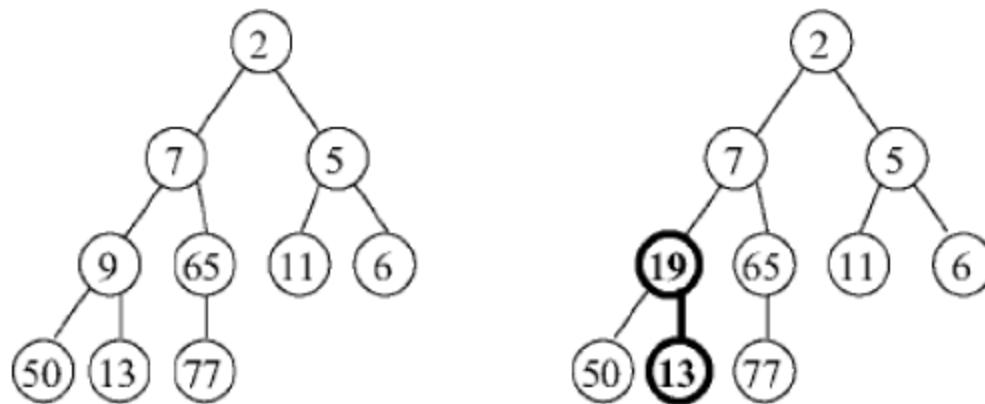


Рис. 9.2

**Утверждение 1.** Минимальный элемент кучи находится в ее корне.

Из соотношения порядка следует, что вдоль пути, ведущего из корня в любую вершину, расположенные в вершинах пути числа могут только увеличиваться. Значит, число, расположенное в корне, меньше любого другого числа кучи.

---

<sup>1</sup> Мы для простоты будем предполагать, что в куче не может быть одинаковых по величине элементов. Так проще, а обобщение труда не составляет.

**Утверждение 2.** Высота дерева<sup>1</sup>, представляющего кучу из  $n$  элементов, по порядку величины составляет  $O(\log n)$ .

В каждом слое ровного бинарного дерева, кроме, может быть, последнего слоя, в два раза больше вершин, чем в предыдущем. Значит, число вершин в  $k$ -м слое равно  $2^k$ , а в последнем слое – от 1 до  $2^h$ , где  $h$  – высота дерева. Дерево высоты  $h$  имеет минимальное количество вершин, если в последнем слое у него находится одна вершина. Посчитаем количество вершин в таком дереве:

$$n_{\min} = \sum_{k=1}^{h-1} 2^k + 1 = 2^h.$$

Таким образом, для любого дерева высотой  $h$  выполняется неравенство  $n \geq n_{\min} = 2^h$ . А следовательно,  $h \leq \log n$ , что и доказывает справедливость утверждения.

Для работы алгоритма может потребоваться извлечение из кучи не минимального, а максимального элемента. В этом случае техника выполнения операций полностью сохраняется, но во всех соотношениях порядка надо знак «меньше» заменить на «больше», то есть предок должен быть больше обоих своих потомков. Всё изложенное в этом разделе распространяется и на такие кучи с точностью до замены минимума на максимум и знака «меньше» на «больше».

## 9.2. Представление кучи в компьютере

Существует множество различных способов кодировки деревьев. Каждый из них применяется в конкретном случае, но все они требуют дополнительной памяти. Все, кроме одного. Для кодировки кучи дополнительной памяти не требуется. Этот факт наряду с простотой и достаточно высокой эффективностью реализации перечисленных выше операций делает кучи очень удобным инструментом для использования в программах.

Чтобы закодировать кучу, запишем в одномерный массив числа, находящиеся в вершинах, по слоям – сверху вниз, а в каждом слое – слева направо. В нулевой позиции массива будет находиться число, стоящее в корне, затем два потомка корня и т.д. Ниже приведен массив A, в котором закодирована куча, представленная на рис. 9.2.

---

<sup>1</sup> Напомним, что высотой дерева называется число ребер в пути от корня до самого дальнего листа.

A	2	7	5	9	65	11	6	50	13	77
	0	1	2	3	4	5	6	7	8	9

Для выполнения операций с кучами нам нужно будет находить потомков и предка заданной вершины. Оказывается, что при такой записи элементов кучи в массив сделать это очень просто. Если элемент кучи находится в ячейке массива с номером  $k$ , то номера ячеек, в которых находятся его потомки, определяются формулами  $2k + 1$  и  $2k + 2$ , а номера ячеек, в которых находится его предок – формулой  $\left[\frac{k-1}{2}\right]$ , где квадратные скобки обозначают целую часть числа. Например, число 9 расположено в ячейке номер 3, следовательно, его потомки находятся в ячейках с номерами 7 и 8, а предок – в ячейке номер 1:

A	2	7	5	9	65	11	6	50	13	77
	0	1	2	3	4	5	6	7	8	9
		$\left[\frac{k-1}{2}\right]$		$k$				$2k + 1$	$2k + 2$	
		предок						потомки		

Взглянув на изображение дерева (см. рис. 9.2), можно убедиться, что в данном случае предок и потомки найдены правильно. Докажем, что формулы, определяющие позиции потомков и предка, верны всегда.

Сначала убедимся в справедливости этих формул для потомков. Будем доказывать это утверждение индукцией по номеру ячейки, в которой расположен элемент кучи.

Для корня, находящегося в 0-й ячейке, утверждение верно: его потомки находятся в ячейках 1 и 2.

Предположим, что формула верна для элемента с номером  $k$ , то есть его потомки расположены в ячейках  $(2k + 1)$  и  $(2k + 2)$ . Элемент с номером  $(k + 1)$  находится в дереве по порядку обхода непосредственно после элемента  $k$ . Потомки же элемента  $(k + 1)$  находятся в дереве по порядку обхода сразу после потомков элемента  $k$ . Это значит, что в массиве они будут записаны сразу после потомков элемента  $k$ , то есть в позициях  $(2k + 3)$  и  $(2k + 4)$ . Но, а  $2k + 3 = 2(k + 1) + 1$ , а  $2k + 4 = 2(k + 1) + 2$ , значит, соответствующие формулы верны и для элемента  $(k + 1)$ .

Справедливость формулы для номера предка следует из того, что если применить ее к  $(2k + 1)$  и  $(2k + 2)$ , то получится элемент, находящийся в ячейке под номером  $k$ , действительно являющийся их предком.

Таким образом, для того чтобы найти в куче потомков данной вершины или ее предка, не нужно хранить дополнительные ссылки. Позиции потомков и предка вычисляются с помощью элементарных арифметических действий.

Пусть  $n$  – число элементов кучи. Если  $2k + 1 \geq n$ , то элемент  $k$  является листом, то есть у него нет потомков. Если же  $2k + 1 = n - 1$ , то у элемента  $k$  только один потомок.

### 9.3. Операции с кучами

Опишем, как выполняются все перечисленные выше операции, и проиллюстрируем алгоритм их работы на примерах.

По ходу выполнения операций куча всегда будет оставаться равным бинарным деревом. Однако соотношение порядка в какой-то одной вершине может оказаться нарушенным. Для восстановления соотношения порядка применяется процедура, которую мы будем называть *ремонт кучи*. Нам понадобится два вида ремонта: ремонт *наружный* и ремонт *внутренний*.

#### 9.3.1. Наружный ремонт (увеличение элемента)

Наружный ремонт применяется в случае, если элемент кучи увеличился и перестало соблюдаться соотношение порядка между ним и его потомками, то есть элемент кучи стал больше одного или обоих своих потомков. На рис. 9.3 число 7 увеличилось в 8 раз и стало равно 56. Теперь оно больше своего потомка 19, соотношение порядка нарушилось, и структура данных перестала быть кучей.

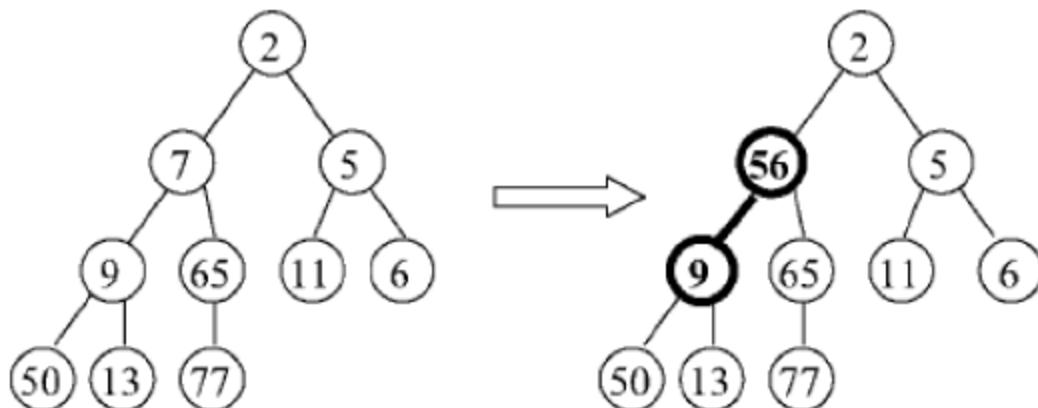


Рис. 9.3

В этом случае необходимо выполнить процедуру наружного ремонта.

Предположим, что соотношение порядка нарушено в вершине  $k$ , вершины  $i$  и  $j$  – ее потомки, а вершина  $p$  – предок. При этом во всех остальных вершинах дерева соотношение порядка сохранилось. Обозначим через  $r_k, r_i, r_j, r_p$  – числа, хранящиеся в соответствующих вершинах, а через  $\bar{r}_k$  – число, ранее находившееся в вершине  $k$ . Из соотношений порядка следует, что  $\bar{r}_k < r_i$ ,  $\bar{r}_k < r_j$  и  $\bar{r}_k > r_p$ . Однако содержимое вершины  $k$  увеличилось:  $r_k > \bar{r}_k$ , и, как следствие этого, оказалось нарушенным соотношение порядка между вершиной  $k$  и ее потомками, то есть  $r_k > \min(r_i, r_j)$ .

Наружный ремонт кучи состоит из последовательности итераций, на каждой из которых восстанавливается соотношение порядка в заданной вершине  $k$ . При этом, возможно, соотношение порядка окажется нарушенным в одном из потомков  $k$  – вершине следующего слоя.

Для восстановления соотношения порядка в вершине  $k$  выберем минимальное значение среди ее потомков. Пусть, например,  $r_i < r_j$ . Меняем местами  $r_k$  с потомком, содержащим минимальное значение, то есть, в данном случае, с  $r_i$  (рис. 9.4).

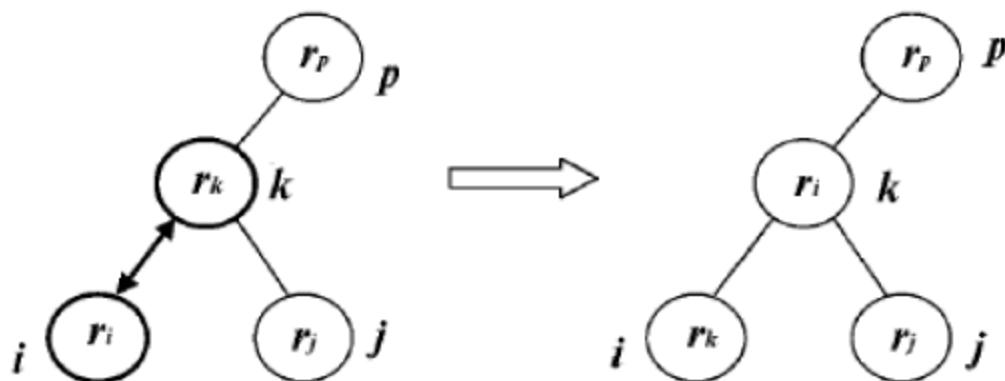


Рис. 9.4

Теперь в вершине  $k$  содержится  $r_i$ , а в вершине  $i$  –  $r_k$ . Это значит, что в вершине  $k$  соотношение порядка восстановлено:  $r_i < r_j$  по предположению, а  $r_i < r_k$ , поскольку соотношение порядка в вершине  $k$  было нарушено.

В вершине  $p$  соотношение порядка сохранилось, так как теперь в ее потомке  $k$  находится число  $r_i$ . Однако в куче до изменения в вершине  $k$  выполнялись все соотношения порядка, то есть  $r_p < \bar{r}_k$  и  $\bar{r}_k < r_i$ . Следовательно,  $r_p < r_i$ .

Во всех остальных вершинах, кроме  $i$ , соотношение порядка измениться не могло. А вот в вершине  $i$  это соотношение могло нарушиться, потому что новое число  $r_k$  больше  $r_i$ , которое раньше в этой вершине содержалось, то есть  $i$  – единственная вершина в куче, в которой после перестановки может оказаться нарушено соотношение порядка.

Возможны три варианта.

1 У вершины  $i$  нет потомков, поэтому соотношение порядка нарушено быть не может.

2 У вершины  $i$  есть потомки, но содержащиеся в них числа больше  $r_k$ .

3 У вершины  $i$  есть потомки, и, по крайней мере, одно из содержащихся в них чисел меньше  $r_k$ .

В первом и во втором случаях в вершине  $i$ , а следовательно, и во всей куче соотношения порядка выполняются. Таким образом, наружный ремонт кучи закончен.

В третьем случае ремонт кучи следует продолжить. При этом следующая итерации повторяет предыдущую, так как ситуация не изменилась: теперь увеличился элемент, находящийся в вершине  $i$ .

Каждая следующая итерация проходит в вершине, находящейся в слое, отстоящем дальше от корня. Следовательно, общее число итераций не может превышать высоту дерева. Таким образом, для наружного ремонта кучи не может потребоваться более чем  $h = O(\log n)$  итераций, где  $h$  – высота дерева, а  $n$  – число элементов кучи. Каждая итерация требует константы действий. Следовательно, общее количество действий, необходимых для проведения наружного ремонта, оценивается величиной порядка  $O(\log n)$ .

Ниже на рис. 9.5 проиллюстрировано проведение наружного ремонта кучи, когда, как на рис. 9.3, вместо числа 7 в вершине оказалось число 56.

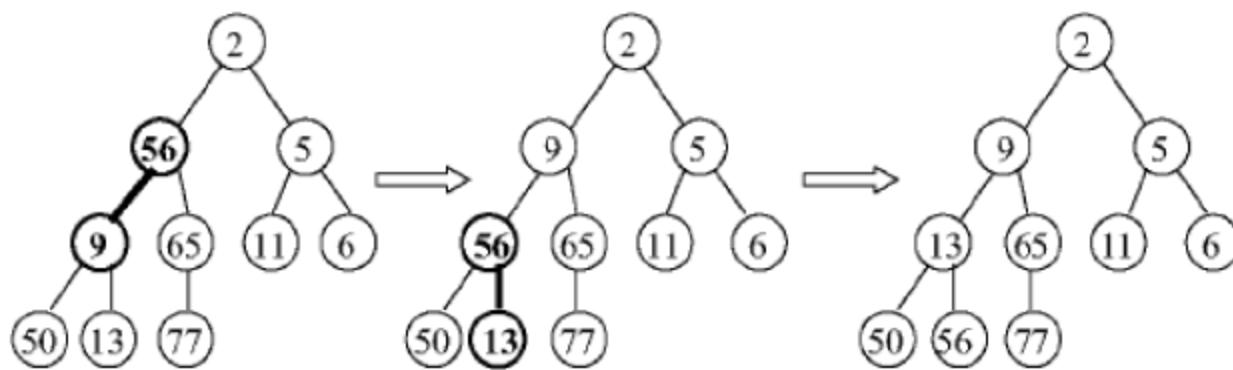


Рис. 9.5

Научившись выполнять наружный ремонт, мы освоили операцию *Увеличение элемента*. Предположим, что какой-то элемент кучи увеличился. Если соотношение порядка сохранилось, то все в порядке. Если же оно оказалось нарушено, то выполняем наружный ремонт. Таким образом, сложность этой операции составляет  $O(\log n)$ .

### 9.3.2. Внутренний ремонт (уменьшение элемента)

Внутренний ремонт применяется, когда элемент кучи уменьшился и перестало соблюдаться соотношение порядка между ним и его предком, то есть элемент кучи стал меньше своего предка. На рис. 9.6 видно, что элемент 9 стал равен 1. Теперь он меньше своего предка 7. Куча нуждается в ремонте.

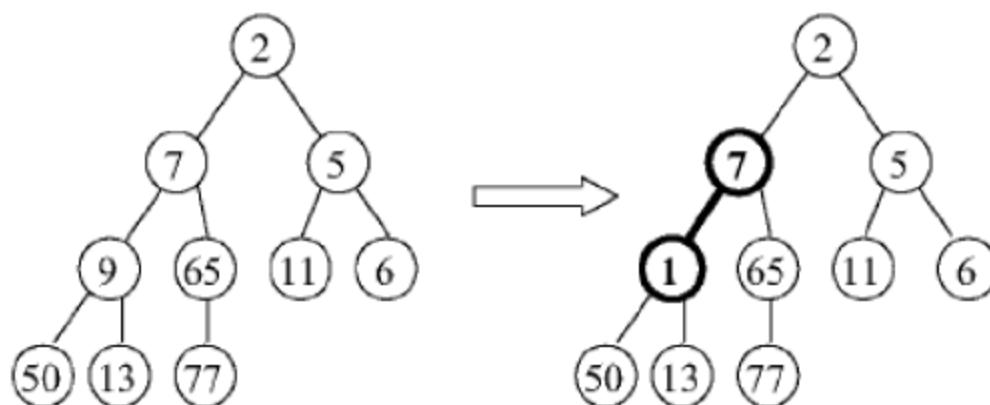


Рис. 9.6

Внутренний ремонт похож на наружный, но его выполнение еще проще. При наружном ремонте «дефектная вершина», в которой было нарушено соотношение порядка, перемещалась в направлении от корня к листьям, а при внутреннем ремонте – дефектная вершина передвигается в сторону корня.

Предположим, уменьшился элемент, находящийся в вершине  $k$ . При этом он стал меньше элемента, содержащегося в вершине  $p$  – предке  $k$ , то есть в вершине  $p$  оказалось нарушено соотношение порядка. Во всех остальных вершинах дерева соотношение порядка сохранилось. Обозначим через  $r_k, r_p$  – числа, хранящиеся в соответствующих вершинах, а через  $\bar{r}_k$  – число, ранее находившееся в вершине  $k$ . По предположению  $r_p > r_k$ , а из соотношения порядка следует, что  $r_p < \bar{r}_k$ . Чтобы восстановить нарушенное соотношение, поменяем местами  $r_p$  и  $r_k$  (рис. 9.7).

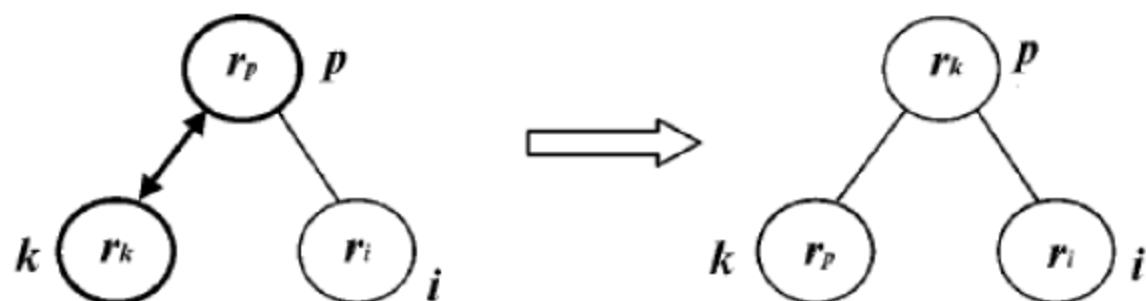


Рис. 9.7

После проведенной перестановки соотношение порядка в вершине  $p$  становится выполненным. Поскольку соотношение порядка было нарушено, то  $r_k < r_p$ . Осталось убедиться в том, что  $r_k < r_i$ . Но это очевидно, так как  $r_k < \bar{r}_k < r_i$ .

В вершине  $k$  теперь находится число  $r_p$ . Поскольку  $r_p < \bar{r}_k$ , а, когда в этой вершине находилось  $\bar{r}_k$ , соотношение порядка было выполнено, то, значит, оно останется выполненным и после перестановки.

Единственная вершина, где после перестановки соотношение порядка может оказаться нарушенным – это предок  $p$ , потому что число  $r_k$ , которое сейчас находится в этой вершине, меньше числа  $r_p$ , находившегося в ней ранее.

Если  $r_k$  меньше числа, находящегося в предке вершины  $p$ , то ситуация повторяется, и нам придется выполнить следующую итерацию внутреннего ремонта: переставить числа, находящиеся в вершине  $p$  и в ее предке. При этом каждый раз вершина, в которой нарушено соотношение порядка, приближается к корню на один слой.

Если же  $r_k$  оказалось больше своего предка, то во всех вершинах кучи соотношение порядка выполнено, и на этом внутренний ремонт заканчивается.

Таким образом, внутренний ремонт кучи состоит из последовательности итераций, на каждой из которых переставляются числа, находящиеся в вершине и ее предке, а вершина с нарушенным соотношением порядка приближается к корню. Внутренний ремонт заканчивается, когда после очередной итерации мы либо дойдем до корня, либо соотношение порядка окажется выполненным для предка переставленной вершины.

Число всех итераций при внутреннем ремонте не может превышать высоту дерева, то есть для внутреннего ремонта кучи не может потребоваться более чем  $h = O(\log n)$  итераций, где  $h$  – высота дерева, а  $n$  – число элементов кучи. Для каждой итерации, как и при наружном ремонте, требуется константа действий. Следовательно, общее количество действий, необходимых для проведения внутреннего ремонта, по порядку величины также составляет  $O(\log n)$ .

На рис. 9.8 проиллюстрирован внутренний ремонт кучи, когда вместо числа 9 в вершине оказалось число 1.

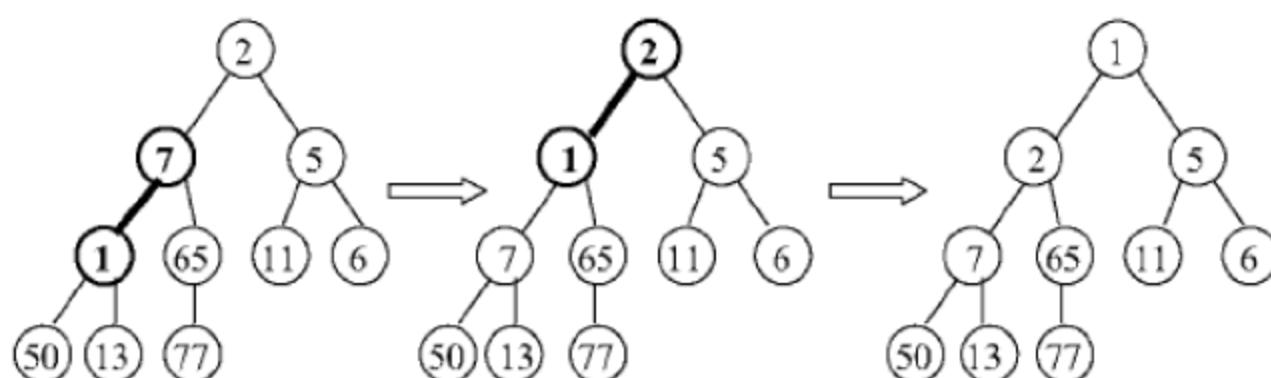


Рис. 9.8

В данном случае внутренний ремонт закончился в корне дерева, но мог прерваться и раньше, если бы после первой итерации в вершине 2 оказалось выполнено соотношение порядка.

Внутренний ремонт – это процедура, которую надо выполнять при операции *Уменьшение элемента*. Следовательно, трудоемкость ремонта кучи при уменьшении одного элемента составляет  $O(\log n)$ .

Все остальные операции с кучами основываются на наружном или внутреннем ремонте.

### 9.3.3. Выбор и удаление минимального элемента

Минимальный элемент находится в корне кучи, поэтому найти его труда не составляет. Если же нужно удалить минимальный элемент из кучи, то на его место в корень следует поместить последний элемент из самого дальнего слоя дерева.

Поскольку любой элемент кучи больше того, который находился в корне, то можно считать, что элемент, находящийся в корне, увеличился. В остальном все в порядке: куча осталась ровным бинарным деревом и больше нигде соотношение порядка не нарушено.

Поскольку элемент, находящийся в корне, увеличился, то для того, чтобы привести кучу в порядок, достаточно провести наружный ремонт, начиная с корня. На рис. 9.9 показано состояние кучи до и после необходимого наружного ремонта.

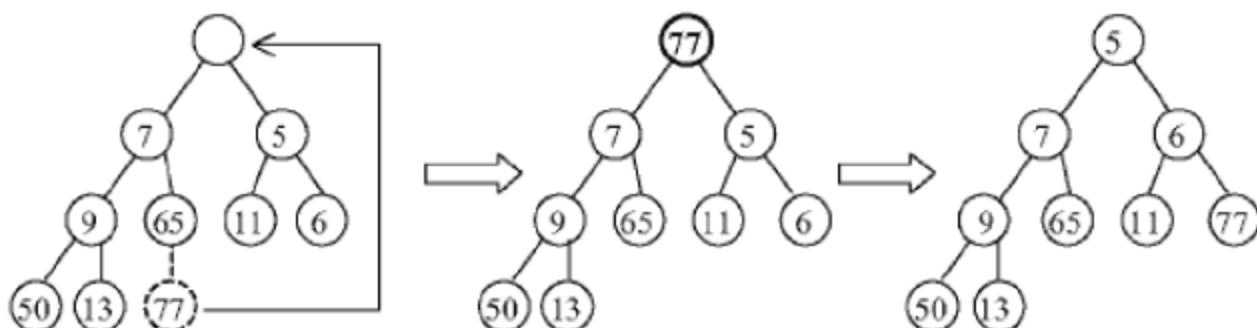


Рис. 9.9

Для перестановки последнего элемента в корень требуется константа действий. Следовательно, трудоемкость выбора и удаления минимального элемента равна по порядку величины трудоемкости наружного ремонта и составляет  $O(\log n)$ .

### 9.3.4. Добавление элемента

Новый элемент следует разместить сразу после последней вершины дерева. Предположим, что в кучу надо поместить число 4. Оно располагается после 77 и становится потомком 65. Куча осталась ровным бинарным деревом. Однако может оказаться нарушенным соотношение порядка в предке новой вершины, из-за того что добавленный элемент меньше своего предка. Таким образом, можно считать, что вновь добавленный элемент «уменьшился». Для того чтобы отремонтировать кучу, нужно выполнить внутренний ремонт, начиная с новой вершины. На рис. 9.10 показана куча до и по окончании ремонта, потребовавшегося после добавления числа 4.

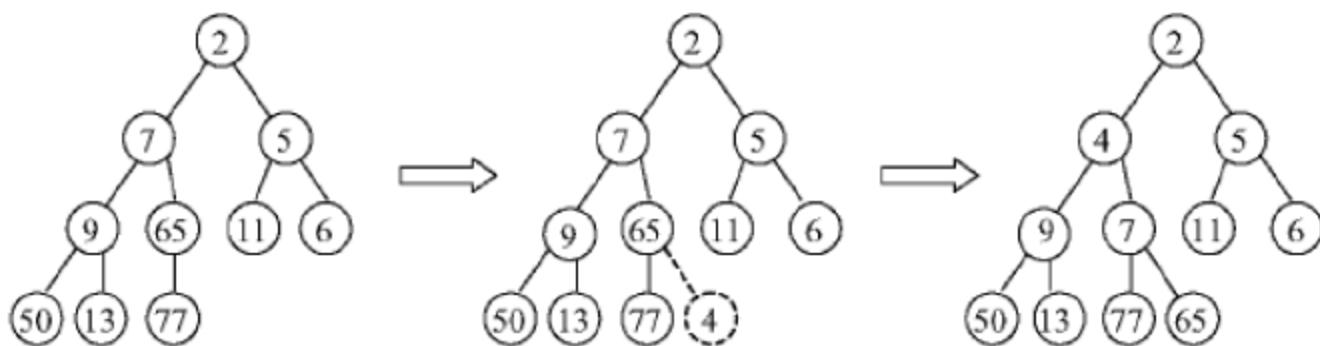


Рис. 9.10

Трудоемкость добавления элемента равна трудоемкости внутреннего ремонта:  $O(\log n)$ .

### 9.3.5. Удаление элемента

Если надо удалить из кучи не корень, а какой-то другой элемент, то, как и при удалении корня, в освободившуюся вершину дерева нужно переместить последний элемент кучи<sup>1</sup>. Последний элемент кучи, оказавшийся на новом месте, может быть либо больше, либо меньше удаленного. Если перемещенный элемент больше удаленного, то надо выполнить наружный ремонт, а если меньше – внутренний. В обоих случаях трудоемкость удаления составит  $O(\log n)$ .

На рис. 9.11 показано удаление элемента 5. На его место попадает 77, то есть надо делать наружный ремонт. Очевидно, что при наружном ремонте поменяются местами числа 77 и 6, после чего соотношение порядка будет восстановлено.

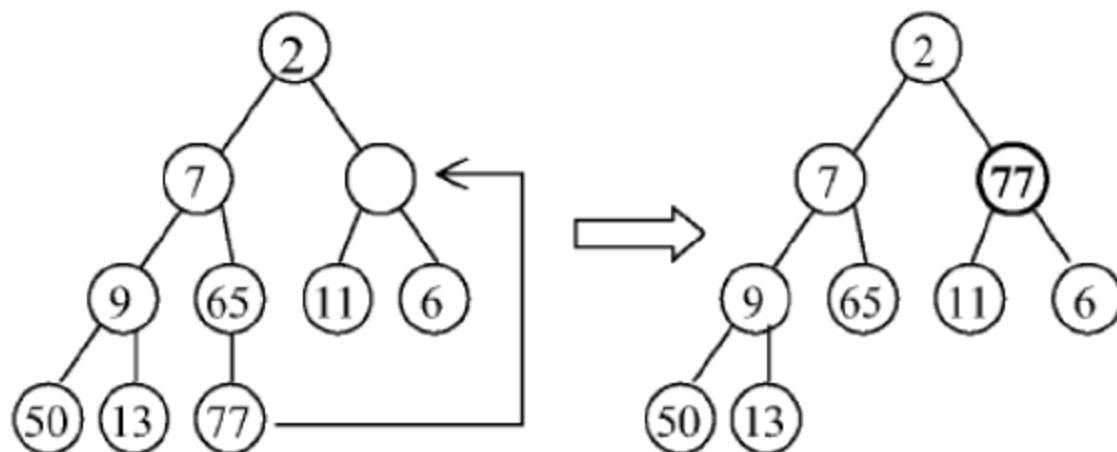


Рис. 9.11

<sup>1</sup> За исключением того случая, когда удаляемый элемент является последним.

### 9.3.6. Окучивание (*heapify*)

Задано ровное бинарное дерево, каждая вершина которого содержит некоторое число. Не будем предполагать, что соотношение порядка выполнено хотя бы в одной вершине этого дерева, то есть элементы по дереву разбросаны произвольно. *Окучиванием* называется преобразование такого дерева в кучу. После него соотношение порядка должно выполняться во всех вершинах.

Мы будем добиваться выполнения соотношения порядка последовательно, двигаясь от последней вершины дерева к корню и восстанавливая соотношение порядка в текущей вершине. На самом деле, не все так плохо. По крайней мере в половине вершин соотношение порядка уже выполнено. Это последние  $n/2$  вершин<sup>1</sup>. У этих вершин нет потомков. А раз нет потомков, то нет и проблем с выполнением соотношения порядка. Следовательно, соотношение порядка может быть нарушено только начиная с «середины дерева» – с тех вершин, у которых есть потомки. На рис. 9.12 жирными кружками выделены вершины, которые придется ремонтировать, а стрелками показан порядок их обхода: справа налево и снизу вверх.

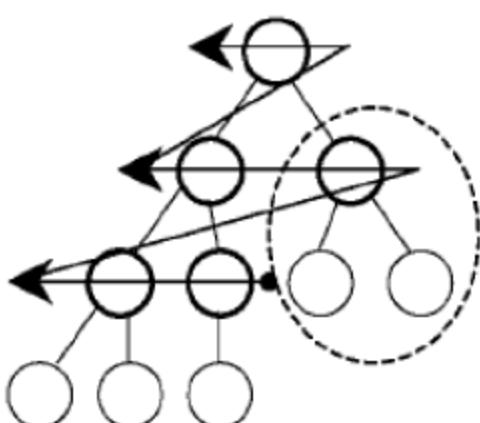


Рис. 9.12

Находясь в очередной вершине, будем приводить в порядок поддерево с корнем в этой вершине. На рис. 9.12 серым цветом выделена некоторая вершина, а контуром обведено поддерево, корнем которого она является. При этом мы будем предполагать, что во всех потомках текущей вершины соотношение порядка уже выполнено. Очевидно, что для первой просматриваемой вершины это справедливо, поскольку ее потомки – листья, а в них соотношение порядка не может быть нарушено.

Рассматриваемое поддерево само по себе является ровным бинарным деревом. По предположению во всех его вершинах, кроме, может быть, корня, соотношение порядка выполняется. Если корень этого поддерева меньше своих потомков, значит, это поддерево – куча. Если же корень больше какого-то из потомков, то в корне надо провести наружный ремонт, после чего поддерево станет кучей.

<sup>1</sup> Напомним, что  $n$  – это число элементов кучи.

Таким образом, мы добиваемся того, чтобы во всех просматриваемых вершинах выполнялось соотношение порядка. При этом в вершинах, просмотренных ранее, соотношение порядка сохраняется. Отсюда следует справедливость сделанного предположения о том, что во всех потомках текущей вершины соотношение порядка выполнено. Действительно, когда мы начнем ремонтировать текущую вершину, то среди ее потомков могут быть только листья и вершины, просмотренные ранее.

Итак, двигаясь в указанном направлении, мы добиваемся того, что во всех пройденных вершинах соотношение порядка выполнено. А значит, когда мы дойдем до корня всего дерева и выполним в нем необходимый ремонт, все дерево станет кучей. На этом окучивание заканчивается.

Оценим теперь сложность окучивания. В первом приближении получается  $O(n \log n)$ . Мы просматриваем  $n/2$  вершин, в каждой из которых, возможно, придется выполнять наружный ремонт. Трудоемкость наружного ремонта в одной вершине составляет  $O(\log n)$ , откуда и получается оценка  $O(n \log n)$ . Но если посмотреть на процесс окучивания более внимательно, то можно доказать, что на самом деле сложность окучивания линейна:  $O(n)$ .

Воспользуемся тем, что для большинства вершин «стоимость» наружного ремонта гораздо меньше, чем  $O(\log n)$ , а вершин, для которых его трудоемкость близка к  $O(\log n)$ , очень мало.

Трудоемкость наружного ремонта поддерева с корнем в данной вершине по порядку величины равна высоте этого поддерева.

У  $n/2$  вершин (листьев) мы не производим ремонт, потому что высота их поддеревьев нулевая. У  $n/4$  вершин высота поддерева равна 1. У  $n/8$  вершин высота поддерева равна 2. У  $n/2^k$  вершин высота поддерева составляет  $(k - 1)$ . Найдем суммарную высоту  $H$  всех поддеревьев, что позволит оценить общую трудоемкость всех наружных ремонтов, а следовательно, и всего процесса окучивания:

$$H = \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \dots + \frac{n}{2^{k+1}} \cdot k + \dots = \frac{n}{2} \left( \frac{1}{2} + \frac{2}{2^2} + \dots + \frac{k}{2^k} + \dots \right) < \frac{n}{2} \sum_{k=1}^{\infty} \frac{k}{2^k}.$$

Однако из основ математического анализа известно, что этот бесконечный ряд сходится:  $\sum_{k=1}^{\infty} \frac{k}{2^k} = C$ . Следовательно,  $H < \frac{Cn}{2} = O(n)$ .

Поэтому сложность окучивания равна  $O(n)$ .

#### 9.4. Программная реализация операций с кучами

Программная реализация всех операций с кучами чрезвычайно проста. Это является одной из причин того, что кучи часто используются на практике. Мы будем предполагать, что кучи закодированы в массиве  $A$  так, как было показано выше, то есть если элемент кучи находится в ячейке массива с номером  $k$ , то его потомки находятся в ячейках  $(2k + 1)$  и  $(2k + 2)$ , а предок – в ячейке  $\left[\frac{k-1}{2}\right]$ . При кодировании процедур будем считать, что массив  $A$  и  $n$  – число элементов кучи – являются глобальными параметрами. С помощью значка  $\Leftrightarrow$  обозначим операцию, меняющую местами два элемента массива. Все комментарии содержатся в теле процедур.

```
//НАРУЖНЫЙ РЕМОНТ
//УВЕЛИЧИЛСЯ ЭЛЕМЕНТ КУЧИ В ПОЗИЦИИ k0
REM_N ( k0 )
{
    //ЦИКЛ ПО СЛОЯМ ДЕРЕВА – ПОКА У ВЕРШИНЫ
    //ЕСТЬ ПОТОМКИ
    for( k = k0; k < (n-1)/2; k = k1 )
    {
        k1 = 2*k + 1;      //первый потомок k
        k2 = k1 + 1;       //второй потомок k
        if ( k2 < n          //Второй потомок есть
            &&                  //и
            A[k2] < A[k1] //он меньше первого
        )
        k1 = k2; //сравниваем со вторым потомком,
                  //а иначе работаем с первым

        if( A[k] < A[k1]) break;
        //Соотношение порядка в вершине k выполнено
```

```

//Ремонт закончен
//Иначе:
//Меняем местами A[k] и его потомка A[k1]
A[k] ↔ A[k1];
//и продолжаем итерации
} //конец цикла
} //конец наружного ремонта

//ВНУТРЕННИЙ РЕМОНТ
//УМЕНЬШИЛСЯ ЭЛЕМЕНТ КУЧИ В ПОЗИЦИИ k0
REM_V ( k0 )
{
    //ЦИКЛ ПО СЛОЯМ ДЕРЕВА ДО КОРНЯ
    for( k = k0; k > 0; k = k1 )
    {
        k1 = (k-1)/2; //предок k
        if ( A[k1] < A[k] ) break;
        //Соотношение порядка в вершине k1
        //выполнено. Ремонт закончен
        //Иначе:
        //Меняем местами A[k] и его предка A[k1]
        A[k] ↔ A[k1];
        //и продолжаем итерации
    } //конец цикла
} //конец внутреннего ремонта

//ВЫБОР И УДАЛЕНИЕ МИНИМАЛЬНОГО ЭЛЕМЕНТА
GET_MIN ()
{
    min = A[0]; //минимум - в корне
    A[0] = A[n-1]; //последний элемент - в корень
    n--;
    //куча уменьшилась
    REM_N ( 0 ); //наружный ремонт в корне
    return( min );
} //минимум найден и удален из кучи

//ДОБАВЛЕНИЕ ЭЛЕМЕНТА a
ADD ( a )
{
    A[n] = a; //новый элемент - в конец кучи
}

```

```

n++;                      //куча увеличилась
REM_V ( n-1 );           //внутренний ремонт в новой
                         //вершине
} //новый элемент a добавлен

//УДАЛЕНИЕ ЭЛЕМЕНТА
//УДАЛЯЕМ ЭЛЕМЕНТ КУЧИ ИЗ ПОЗИЦИИ k0
REMOVE ( k0 )
{
    a = A[k0]; //запоминаем величину удаляемого
    A[k0] = A[n-1]; //последний элемент -
                     //на свободное место
    n--;
    if ( A[k0] > a ) //элемент увеличился
        REM_N ( k0 ); //наружный ремонт
    else               //элемент уменьшился
        REM_V ( k0 ); //внутренний ремонт
} //элемент из позиции k0 удален

//ОКУЧИВАНИЕ
HEAPIFY ()
{
    //ОТ СЕРЕДИНЫ КУЧИ ДО КОРНЯ
    for( k = (n-1)/2; k >= 0; k-- )
        REM_N ( k ); //наружный ремонт поддерева
    } //массив стал кучей
}

```

Итак, все операции реализованы. Удивительно, что такие простые программы действительно хорошо работают на практике. Они настолько компактны, что часто не имеет смысла оформлять их в виде процедур. Необходимые несколько операторов размещаются непосредственно в теле программы.

## 9.5. Сортировка деревом (пирамидальная сортировка)

Алгоритм *сортировки деревом* или, как его иногда называют, алгоритм *пирамидальной сортировки* является одним из лучших алгоритмов упорядочения одномерного массива. Его трудоемкость –  $O(n \log n)$  – минимально возможная. Основу алгоритма составляет техника работы с кучами, он не требует дополнительной памяти и очень прост для реализации. Скоро мы в этом убедимся.

Задан массив чисел  $A[0..n-1]$ . Требуется отсортировать его в порядке убывания.

### 9.5.1. Общее описание алгоритма

Алгоритм состоит из двух этапов.

На *первом этапе* мы преобразуем массив в кучу, то есть применяем к нему уже известную нам процедуру окучивания.

*Второй этап* состоит из последовательности итераций. Перед очередной итерацией массив разделен «забором» на две части. Справа от забора<sup>1</sup> находится отсортированный фрагмент – числа, упорядоченные по убыванию. Слева от забора – куча из оставшихся чисел.

При этом соблюдается *неравенство разделения*: любое число, находящееся в куче слева от забора, больше любого числа в отсортированном фрагменте справа от забора.

Таким образом, справа от забора находятся самые маленькие числа массива, упорядоченные по убыванию. Очевидно, что когда массив будет упорядочен по убыванию полностью, отсортированный фрагмент полностью сохранится. Итак, справа от забора находится готовый кусок упорядоченного массива, который не нуждается в дальнейшей обработке. Ниже приведен массив чисел, в котором ранее была закодирована куча. На рис. 9.13 забор отстоит от конца массива на две позиции.

Перед первой итерацией второго этапа забор расположен в конце массива, то есть отсортированный фрагмент пуст и весь массив является кучей. Из этого следует, что вначале неравенство разделения справедливо. После очередной итерации размер отсортированного фрагмента увеличивается на единицу, а неравенство разделения сохраняется. Таким образом, после  $(n - 1)$  итерации массив  $A$  будет упорядочен по убыванию.

A	6	7	11	9	65	13	77	50	5	2
	0	1	2	3	4	5	6	7	8	9
куча										
забор										
отсортирован- ный фрагмент										

<sup>1</sup> То есть в части массива с большими индексами.

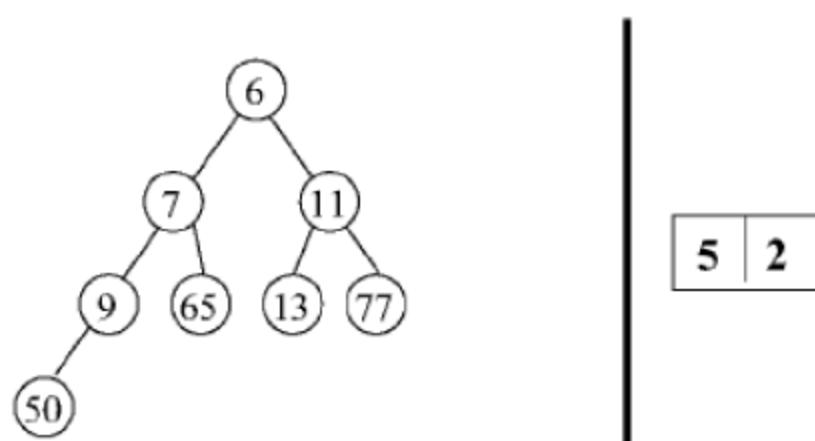


Рис. 9.13

### 9.5.2. Описание одной итерации второго этапа.

1. Меняем местами первый и последний элемент слева от забора (рис. 9.14).
2. Передвигаем забор на одну позицию влево.
3. Осуществляем наружный ремонт в корне уменьшенной кучи.

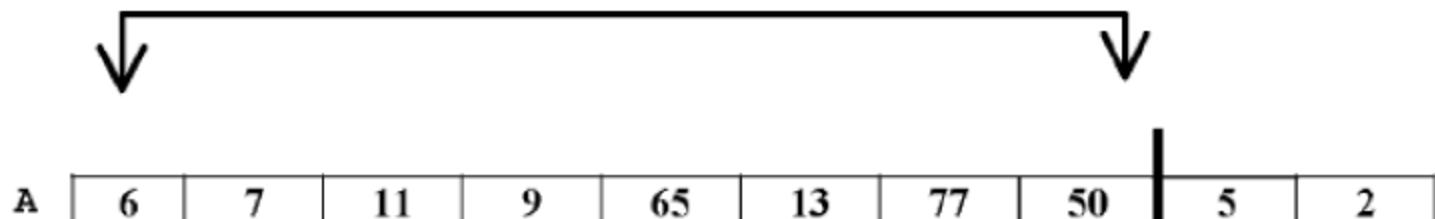


Рис. 9.14

Первым элементом массива является корень кучи. Он содержит минимальный элемент кучи и, следовательно, является минимальным слева от забора. Таким образом, после перестановки этот элемент будет меньше всех оставшихся элементов кучи, но, как следствие неравенства разделения, – больше всех элементов, находящихся в отсортированном фрагменте. Следовательно, если передвинуть забор на одну позицию влево, то справа от забора будут расположены числа, упорядоченные по убыванию, и при этом останется выполненным «неравенство разделения»: любое число слева от забора больше любого числа справа.

Чтобы левая часть массива опять стала кучей, достаточно выполнить в корне наружный ремонт, поскольку во всех остальных вершинах дерева соотношение порядка не могло быть нарушено.

На рис. 9.15 изображено состояние массива  $A$  после очередной итерации.

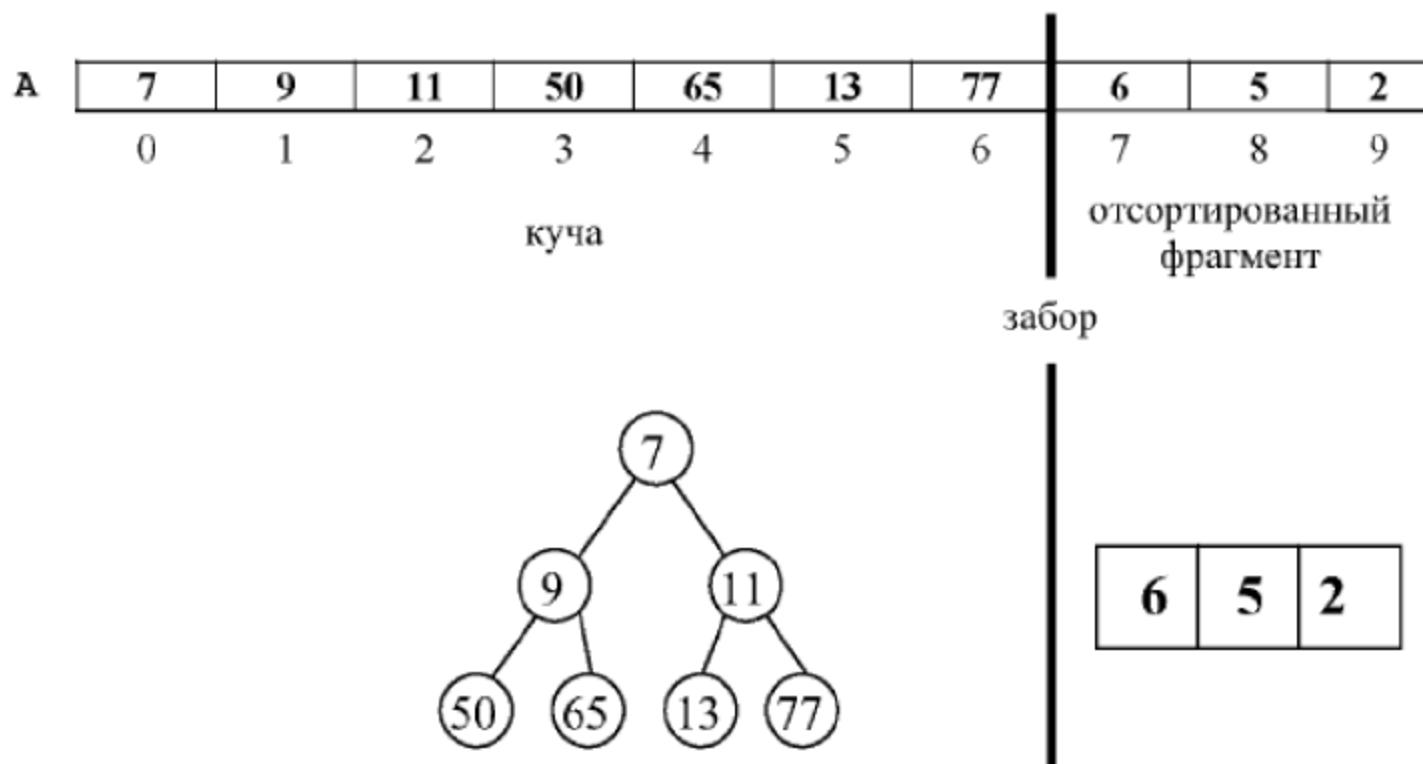


Рис. 9.15

Таким образом, после очередной итерации упорядоченный фрагмент увеличился на единицу, слева осталась куча, и между кучей и упорядоченным фрагментом сохранилось неравенство разделения. Из этого следует корректность алгоритма: после  $(n - 1)$  итерации будет построен упорядоченный по убыванию массив.

Если нужно упорядочить массив не по убыванию, а по возрастанию, то следует использовать кучу с обратным соотношением порядка, у которой в корне находится не минимум, а максимум.

Трудоемкость алгоритма сортировки деревом складывается из трудоемкости двух этапов. Как было показано выше, трудоемкость окучивания составляет  $O(n)$ . Остается оценить трудоемкость второго этапа. Каждая итерация, помимо фиксированных операций, связанных с перестановкой двух элементов массива и передвижением забора, требует одного наружного ремонта. Следовательно, трудоемкость одной итерации –  $O(\log n)$ , а трудоемкость всего второго этапа –  $O(n \log n)$ . Суммируя оценки трудоемкости первого и второго этапов, получаем  $O(n) + O(n \log n) = O(n \log n)$ , то есть сложность сортировки деревом – минимально возможная с точки зрения теории.

Программная реализация алгоритма сортировки деревом очень простая. Будем обозначать, как и раньше, через  $n$  размер кучи, а через глобальную переменную  $n_0$  – число элементов в массиве  $A$ .

```
//СОРТИРОВКА ДЕРЕВОМ
SORT_TREE ()
{
    HEAPIFY (); //1-й этап - окучивание
    //2-й этап
    for( n = n0; n >= 0; )
        {
            //итерация
            A[0] ↔ A[n-1]; //расширяем
                            //отсортированный фрагмент
            n--;           //уменьшаем размер кучи
            REM_N ( 0 ); //наружный ремонт в корне
        } //конец 2-го этапа
    } // сортировка закончена
```

С точки зрения теории алгоритм сортировки деревом идеален: минимально возможная сложность и отсутствие дополнительной памяти. Тем не менее самым быстрым на практике считается не он, а алгоритм *быстрой сортировки* – *quicksort*, хотя его оценка сложности в худшем случае больше и составляет  $O(n^2)$ . Таким образом, в худшем случае алгоритм быстрой сортировки работает гораздо медленней алгоритма сортировки деревом, но зато в среднем при массовом решении задач – быстрее.

Какой из этих алгоритмов следует использовать в своей программе – дело вкуса и ситуации. Алгоритм сортировки деревом проще написать, он в отличие от алгоритма быстрой сортировки не нуждается в настройке и всегда дает гарантированно хороший результат. Последнее обстоятельство может быть чрезвычайно важным в ситуации, когда задачи решаются в режиме реального времени и медленное решение даже одной задачи из тысячи может привести к недопустимым последствиям. А вот при массовом решении задач упорядочения хорошо настроенный и «вылизанный» алгоритм быстрой сортировки может оказаться все же несколько быстрее. Так что решайте сами, хотя, если не надо «бить рекорды», возможно, стоит предпочесть сортировку деревом.

А вот какой алгоритм совершенно точно не надо использовать, так это сортировку пузырьком. На больших задачах «пузырек» в десятки, сотни и тысячи раз медленнее сортировки деревом, при том что программировать алгоритм пузырька ничуть не проще.

Хочется надеяться, что каждый, кто дочитал до этого места, больше никогда в жизни не будет использовать сортировку пузырьком.

## 9.6. Обобщения

В этом разделе мы рассматривали бинарные кучи, то есть кучи, в которых у элементов, не являющихся листьями, имеется по два потомка. Однако аналогично можно пользоваться тернарными кучами – кучами с тремя потомками, а также с четырьмя, пятью и более.

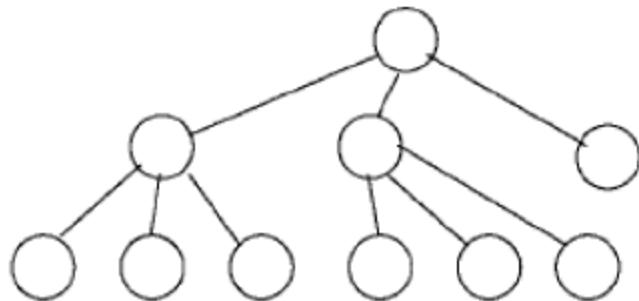


Рис. 9.16

На рис. 9.16 приведен пример дерева, соответствующего тернарной куче. У всех ее вершин либо три, либо ни одного потомка. Только у одной вершины предпоследнего слоя может быть один или два потомка. Практически всё, что было изложено в этом разделе, с небольшими из-

менениями применимо и к таким кучам. Например, в массиве для тернарной кучи потомки элемента с номером  $k$  находятся в ячейках

$(3k + 1)$ ,  $(3k + 2)$  и  $(3k + 3)$ , а предок – в ячейке  $\left[\frac{k - 1}{3}\right]$ . Алгоритмы

выполнения операций тоже почти не меняются, только в ходе наружного ремонта при поиске минимального потомка придется выбирать не из двух, а из трех вершин.

Какие кучи следует применять на практике? У кучи с большим ветвлением меньше высота. Высота  $k$ -арной кучи из  $n$  элементов не превосходит  $\log_k n$ . Таким образом, чем больше  $k$ , тем меньше высота кучи. Число итераций при наружном и внутреннем ремонте не больше высоты кучи. Поэтому для  $k$ -арной кучи трудоемкость внутреннего ремонта составит  $O(\log_k n)$ , а трудоемкость наружного ремонта –  $O(k \log_k n)$ , поскольку на каждой итерации приходится искать минимум среди  $k$  потомков, то есть с ростом  $k$  сложность внутреннего ремонта уменьшается, а наружного – возрастает. Использовать кучи с коэффициентом ветвления  $k > 2$  целесообразно, когда

предполагаемое число внутренних ремонтов значительно превосходит число наружных. Например, такая ситуация может возникнуть при реализации с помощью куч алгоритма Дейкстры для построения дерева кратчайших расстояний. В этом алгоритме наружный ремонт кучи нужен, когда вершину с минимальным текущим расстоянием выбирают для просмотра, а внутренний ремонт – когда при просмотре дуги у помеченной, но еще не просмотренной вершины уменьшается текущее расстояние. Отсюда следует, что наружных ремонтов может быть не больше  $n$ , а внутренних – не больше  $m$ , где  $n$  – число вершин сети, а  $m$  – число дуг. Если количество дуг намного больше количества вершин, то, возможно, кучи с большим коэффициентом ветвления будут работать более эффективно. Результаты вычислительных экспериментов показывают, что при решении задач некоторых классов алгоритм Дейкстры, использующий кучи с  $k = 3$  или  $k = 4$ , работает быстрее алгоритма с бинарными кучами. Однако, видимо, не следует без серьезных причин вместо бинарных куч использовать  $k$ -арные. Выигрыш (если будет) может оказаться невелик, а программы, пусть чуть-чуть, но сложнее. Думать о замене бинарной кучи следует только тогда, когда скорость работы программы становится очень важной.

Бывают и другие типы куч: R-кучи, кучи Фибоначчи. Это интересные и красивые структуры данных, но большого практического смысла они, видимо, не имеют. Поэтому, к сожалению, их изучение останется за пределами нашего курса лекций.

## 10. ДЕРЕВЬЯ ПОИСКА

### 10.1. Задача поиска

Рассмотренная в предыдущем разделе структура данных, называемая кучей, позволяет решать задачу выбора минимума из множества элементов и поддержание изменяющегося множества в таком состоянии, что выбор минимума осуществляется эффективно.

Однако кучи не позволяют решать еще одну очень важную задачу: поиск элемента во множестве. В задаче поиска требуется дать ответ, есть ли в ранее сформированном множестве данный элемент.

В разных вариантах задача поиска является одной из самых распространенных в компьютерной математике и в программировании. Ей посвящена половина третьего тома «Сортировка и поиск» классической монографии Д. Кнута. Работая на компьютере, мы все время сталкиваемся с этой задачей: текстовый редактор проверяет правописание, толковый словарь ищет перевод заданного слова, компьютер отслеживает, был ли ранее объявлен встреченный в тексте программы идентификатор и какие у него атрибуты, база данных делает выборку по заданному ключу. Поскольку все эти действия совершаются постоянно и много раз, нужно, чтобы используемые в них алгоритмы работали быстро и не требовали слишком большого объема памяти.

В этом разделе задача поиска будет рассмотрена в простейшем варианте: множество состоит из элементов, не обладающих внутренней структурой и хранящихся в оперативной памяти. Во множестве задано *отношение порядка*: про любые два элемента известно, какой из них больше, а какой меньше. В качестве простейшей модели такой задачи идеально подходят числа, каждое из которых хранится в отдельной переменной. Слова для такой модели не подходят, потому что их можно начинать сравнивать по буквам, а не целиком, а это дает дополнительные возможности при поиске.

Конечно, наша задача не сводится к одному только поиску. Множество со временем может изменяться: что-то добавится, что-то будет удалено. Поэтому надо будет научиться эффективной реализации операций добавления и удаления элементов.

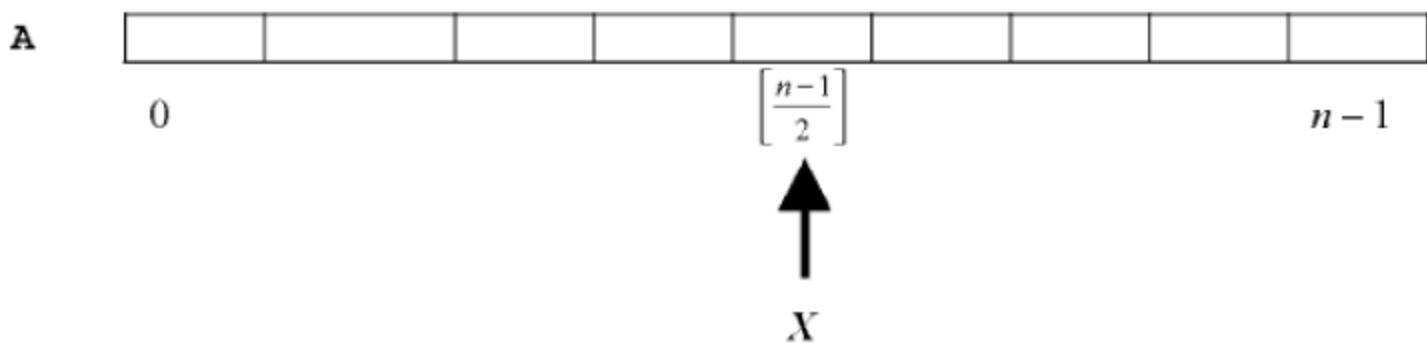
## 10.2. Дихотомический поиск

Начнем с наивного способа решения задачи поиска. Пусть все числа записаны в произвольном порядке в массиве. Тогда, чтобы найти интересующий нас элемент, нужно просмотреть массив от начала до конца, сравнивая величину нового элемента со всеми переменными массива. Это самый простой способ, но, к сожалению, не самый эффективный. Очевидно, что его сложность составляет  $O(n)$ . Хотелось бы большей эффективности. Из теоретических соображений представляется, что результат, к которому надо стремиться, –  $O(\log n)$ .

Получить такую оценку только для поиска несложно. Для этого надо использовать метод дихотомии. Помните задачу: угадать число, меньшее 1000, за 10 вопросов<sup>1</sup>? Первый вопрос: «Больше 500?». Таким образом, отрезок, на котором может находиться число, с каждым вопросом уменьшается в два раза, потому что вопрос мы подбираем так, чтобы разделить отрезок пополам.

Аналогичным образом можно действовать и при поиске элемента в массиве. Предположим, что массив  $A[0..n - 1]$  упорядочен по возрастанию. Надо узнать, встречается ли среди элементов этого массива заданное число  $x$ .

Сначала сравниваем  $x$  с  $A[(n - 1) / 2]$ , то есть с элементом, находящимся посередине массива:



Если  $x = A[(n - 1) / 2]$ , то искомый элемент найден.

Если  $x < A[(n - 1) / 2]$ , то искомый элемент может находиться только в первой половине массива: от 0 до  $(n - 1) / 2$ .

Если  $x > A[(n - 1) / 2]$ , то искомый элемент может находиться только во второй половине массива: от  $(n - 1) / 2$  до  $(n - 1)$ .

Таким образом, после одного сравнения либо искомый элемент будет найден, либо отрезок массива, где он может находиться,

<sup>1</sup> На самом деле, за 10 вопросов можно угадать число до  $1024 = 2^{10}$ .

уменьшится в два раза. Отсюда следует, что после  $\log n$  сравнений либо искомый элемент обнаружится, либо мы убедимся, что такого элемента в массиве нет, то есть трудоёмкость дихотомического поиска составляет  $O(\log n)$ , чего мы и хотели добиться.

Применять дихотомический поиск имеет смысл только тогда, когда массив поиска не изменяется. Если же множество динамично, то желаемой эффективности не достичь. Добавить новый элемент, сохранив упорядоченность, можно только поместив его на строго определенное место. Поэтому массив придется «раздвинуть», освобождая необходимую позицию. После удаления элемента образовавшееся пустое место надо заполнить, поэтому придется «сдвигать» массив. И в том, и в другом случае это потребует порядка  $O(n)$  действий. Поэтому дихотомический поиск решает поставленную задачу не более чем отчасти.

В данном разделе мы разберем структуру данных, называемую *деревьями поиска*. В рассматриваемом варианте эта структура тоже не позволит решить поставленную задачу – выполнять поиск, добавление и удаление элемента за  $O(\log n)$  действий. Однако деревья поиска станут первым шагом к цели. Окончательный результат может быть достигнут при помощи *сбалансированных деревьев поиска*, являющихся усовершенствованными деревьями поиска, рассматриваемыми в этом разделе.

Прежде чем окончательно заняться деревьями поиска, разберем еще один алгоритм поиска в упорядоченном массиве, который можно назвать *взвешенной дихотомией*. Соображения, лежащие в его основе, таковы: не следует делить отрезок ровно пополам. Надо попытаться «угадать», где может находиться элемент, который мы ищем, и постараться попасть в эту точку. Попробуем научиться угадывать.

Если  $x < A[0]$  или  $x > A[n - 1]$ , то такого элемента в массиве нет. Если же  $A[0] < x < A[n - 1]$ , то место, где могут располагаться элементы, близкие к  $x$  по величине, определяется значением  $x$ . Если  $x$  находится недалеко от  $A[0]$ , то искать надо в начале массива, а если он находится около  $A[n - 1]$ , то искать следует ближе к концу массива. Примерное представление, где мог бы располагаться  $x$  в предположении, что величины элементов массива распределены равномерно, можно получить из рис. 10.1. Каждый, кто еще помнит школьную алгебру и геометрию, может легко вывести, что  $k = n \frac{x - A_0}{A_{n-1} - A_0}$ .

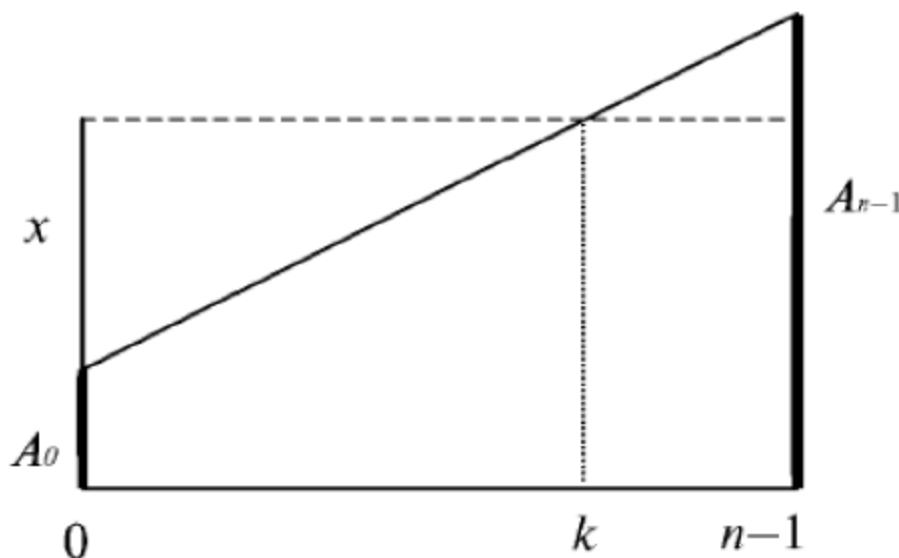


Рис. 10.1

Оказывается, что если по аналогичной формуле определять на каждом шаге номер следующего элемента для сравнения, то в предположении о равномерности распределения можно доказать, что среднее время поиска будет составлять  $O(\log \log n)$ . На практике такой алгоритм, который при поиске пытается «угадать», где может находиться искомое число, действительно работает быстрее обычной дихотомии, но, к сожалению, это никак не помогает при добавлении и удалении элементов.

### 10.3. Деревья поиска

Чтобы найти пути решения поставленной задачи, займемся, наконец, деревьями поиска.

*Деревом поиска* мы будем называть бинарное дерево с корнем. В отличие от кучи деревом поиска может быть любое бинарное дерево, то есть дерево, у которого в каждой вершине количество потомков не превышает двух. Никаких дополнительных требований к структуре этого дерева не предъявляется.

Если вершина не является листом, то у нее может быть один или два потомка. Потомки бывают *левые* и *правые*. Если потомков два, то один из них левый, а другой правый. Если потомок у вершины один, то известно, какой он: левый или правый. При изображении дерева поиска мы так и будем рисовать потомков: левого внизу и слева, а правого — внизу и справа от предка.

В каждой вершине дерева поиска располагается число. Для любой вершины  $i$  должно соблюдаться *соотношение поиска*: элемент  $a_i$ , содержащийся в вершине  $i$ , должен быть больше содержимого

любой вершины левого поддерева и меньше содержимого любой вершины правого поддерева<sup>1</sup>.

Если через  $L_i$  обозначить левое поддерево вершины  $i$ , а через  $R_i$  – правое поддерево, то соотношение поиска в вершине  $i$  можно запи-

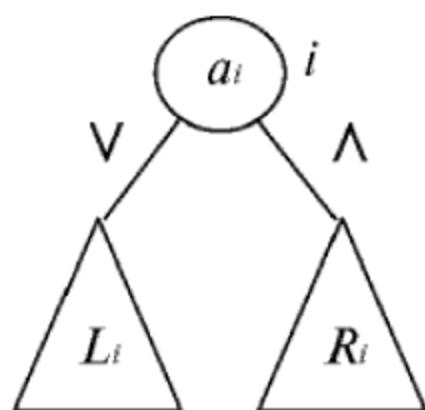


Рис. 10.2

сать следующим образом:  $\forall j \in L_i, a_i > a_j$  и  $\forall k \in R_i, a_i < a_k$ , то есть всё, что находится в левом поддереве, меньше содержимого корня, а всё, что находится в правом поддереве – больше (рис. 10.2). Такое соотношение должно выполняться для всех вершин дерева. Распространенная ошибка состоит в том, что проверяют только соотношение поиска между корнем поддерева и его потомками и если оно выполнено, то делается вывод о

том, что дерево является деревом поиска. Это неправильно. Соответствующие неравенства должны соблюдаться между содержимым корня и всеми остальными вершинами правого и левого поддерева. На рис. 10.3 изображены два бинарных дерева. Левое является деревом поиска, а правое нет. Пара вершин, для которой соотношение поиска нарушено, выделена в правом дереве жирными кружками.

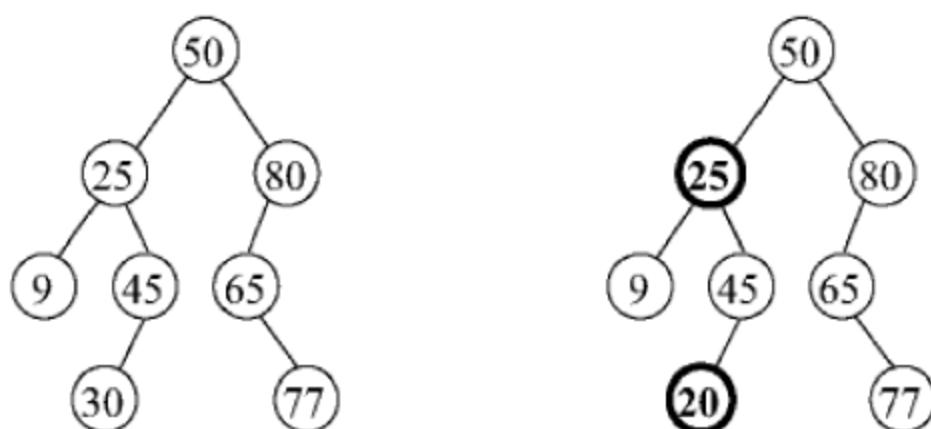


Рис. 10.3

## 10.4. Поиск

Алгоритм поиска по дереву очень прост и напоминает диахотомию. Он состоит из последовательности итераций, на каждой из ко-

<sup>1</sup> Для простоты будем предполагать, что одинаковых элементов в дереве быть не может.

торых новый элемент  $x$  сравнивается со значением, хранящимся в одной из вершин. Сравнения начинаются с корня.

### Итерация поиска в вершине $i$

1. Сравниваем  $x$  с  $a_i$ .
2. Если  $x = a_i$ , то поиск закончен: элемент  $x$  есть в дереве.
3. Если  $x < a_i$ , то поиск продолжается: переходим в вершину  $l_i$  – левый потомок  $i$  – и начинаем следующую итерацию.
4. Если  $x > a_i$ , то поиск продолжается: переходим в вершину  $r_i$  – правый потомок  $i$  – и начинаем следующую итерацию.
5. Если та вершина, в которую надо перейти, отсутствует, то есть у вершины  $i$  нет соответствующего левого или правого потомка, то поиск заканчивается. Элемента  $x$  в дереве нет.

Проиллюстрируем работу алгоритма поиска на примерах, а потом докажем его корректность.

На рис. 10.4 на левом дереве показано, как происходит поиск числа 65. Сравнение начинается в корне, затем мы переходим в вершину 80, где убеждаемся, что искомый элемент во множестве есть. Пройденный при поиске путь изображен пунктирными линиями.

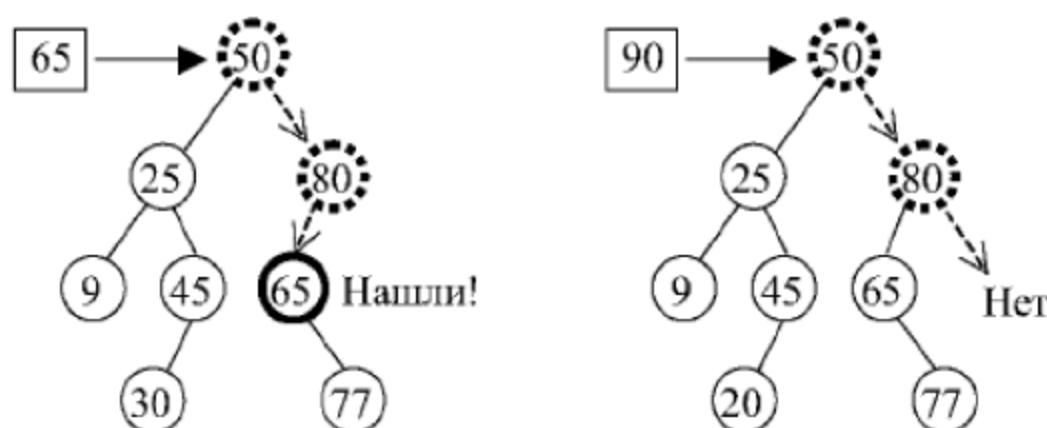


Рис. 10.4

Правое дерево иллюстрирует поиск числа 90. В последней пройденной вершине 80 нужно перейти в вершину, являющуюся ее правым потомком, но такой в дереве нет. Следовательно, число 90 во множестве отсутствует.

Обоснование алгоритма следует из определения дерева поиска. Если элемент найден, то доказывать нечего. Остается доказать, что если алгоритм поиска дал отказ, то искомого элемента действительно нет в дереве.

На первой итерации мы сравниваем новый элемент с корнем. Если он меньше содержимого корня, то переходим в левое поддерево, а если больше, то в правое.

Если мы перешли в левое поддерево, то это значит, что новый элемент меньше корня. Но из соотношения поиска следует, что все элементы правого под дерева больше корня. Следовательно, новый элемент меньше не только корня, но и всех элементов, хранящихся в правом поддереве. Отсюда следует, что если новый элемент и существует во множестве, то он может находиться только в левом поддереве.

Такие же рассуждения показывают, что если мы перешли в правое поддерево, то только в нем может находиться новый элемент, то есть если новый элемент не будет найден в том поддереве, в которое мы перешли, то его нет во всем множестве.

Это рассуждение справедливо и для каждой последующей итерации. Таким образом, когда поддеревом, в котором надо осуществлять поиск, становится пустое дерево, то из этого следует, что искомого элемента в дереве нет: в пустом дереве нет ничего, а из приведенных рассуждений следует, что в оставшейся части дерева искомого элемента также быть не может.

Этим завершается обоснование алгоритма.

## 10.5. Добавление

Прежде чем добавить элемент, сначала надо проверить, есть ли он в дереве поиска, то есть сначала включается процедура поиска, описанная выше. Если элемент найден, то добавлять ничего не надо. Если же искомого элемента в дереве нет, то добавляем его туда, где «прервался поиск». После последнего сравнения нужно было перейти к одному из потомков вершины, с содержимым которой мы сравнивали новый элемент. А этого потомка в дереве не оказалось. Поиск на этом заканчивается. Новый элемент следует расположить на том самом месте, где должен был быть отсутствующий потомок. На рис. 10.5 изображено новое дерево, появившееся после добавления числа 90, поиск которого был показан раньше.

Добавляемая вершина не всегда попадает в самый нижний слой, но всегда становится листом. Легко убедиться, что после добавления нового элемента дерево сохраняет все свойства дерева поиска. Очевидно, что оно остается бинарным. Доказать, что во всех вершинах сохраняется соотношение поиска, тоже легко. Нарушиться оно мог-

ло только в тех вершинах, которые были просмотрены в процессе поиска, потому что у остальных вершин их правые и левые поддеревья не изменились. Возьмем какую-то вершину  $i$ , лежащую на пройденном пути. Если после сравнения мы перешли в левое поддерево,

то, значит, новый элемент  $x$  был меньше содержимого вершины  $i$ . Поскольку в этом случае он будет добавлен в левое поддерево, то соотношение поиска в вершине  $i$  сохраняется. Аналогичные рассуждения применимы, когда в вершине мы переходим в правое поддерево: в этом случае  $x$  больше содержимого вершины  $i$ . То же самое справедливо и для любой другой пройденной вершины. Следовательно, после добавления нового элемента дерево осталось деревом поиска.

Оценим сложность алгоритмов поиска и добавления. Очевидно, что сложность добавления по порядку величины не отличается от сложности поиска. Сложность поиска определяется максимальным количеством сравнений, которое не может быть больше высоты дерева – длины максимального пути от корня до листа, то есть трудоемкость поиска и добавления составляет  $O(h)$ , где  $h$  – высота дерева поиска.

Таким образом, эффективность алгоритмов работы с деревом поиска непосредственно зависит от высоты дерева. В прошлом разделе было показано, что высота ровного бинарного дерева равна  $\log n$ , где  $n$  – количество его вершин. Если бы можно было добиться, чтобы высота дерева поиска тоже стала логарифмической, то поставленная задача была бы решена: сложность поиска и добавления состав  $O(\log n)$ . К сожалению, не все так просто, и сразу получить логарифмическую высоту дерева поиска не удастся. Поскольку мы никак не контролируем «рост» дерева, то оно может удлиняться «в одну сторону».

На рис. 10.6 изображено бинарное дерево поиска, получающееся в результате последовательного включения в него убывающих чисел. Высота такого дерева равна  $n$ , а значит, и трудоемкость поиска в нем имеет порядок  $O(n)$ . Это явно не то, к чему следовало стремиться. Получается, что при неблагоприятном раскладе дерево оказывается хуже диахотомического поиска. При диахотомическом поиске  $O(n)$  требовалось для добавления элемента, а поиск «стоил»  $O(\log n)$ .

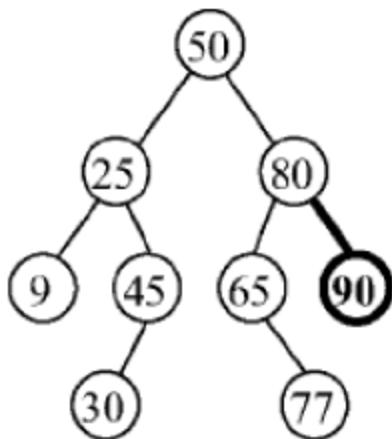


Рис. 10.5

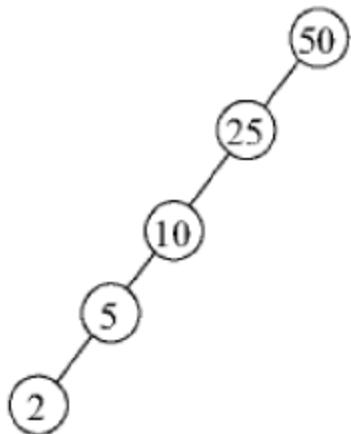


Рис. 10.6

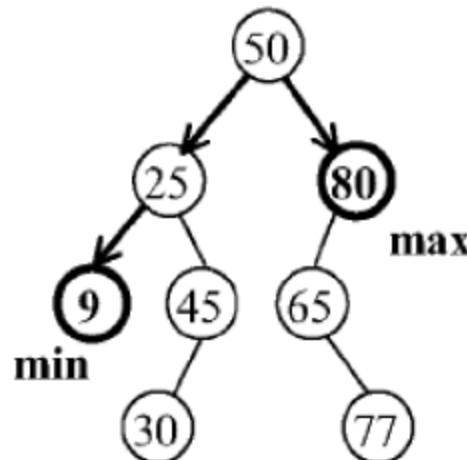


Рис. 10.7

Минимальный элемент кучи находится в ее корне. А где расположена минимальный элемент дерева поиска? Не так близко, но найти его не составит большого труда (рис. 10.7). Надо, начав с корня, все время, пока это возможно, переходить к левому потомку. Если в очередной вершине левая ссылка пустая, то этот элемент является минимальным среди всех, хранящихся в дереве. Доказывать тут нечего. «Самый левый» элемент меньше всех пройденных и заведомо меньше элементов, находящихся в их правых поддеревьях. У найденной вершины может быть только правое поддерево, а слева ничего нет. Отсюда следует, что минимум содержится именно в ней.

Аналогично находится максимальный элемент дерева. Надо идти от корня до конца, все время переходя не к левому, а к правому потомку. На рис. 10.7 вершина, содержащая минимальный элемент, является листом — у нее нет потомков. Так бывает не всегда, в чем легко убедиться, посмотрев на максимальную вершину. У минимальной вершины может быть непустое правое поддерево, а у максимальной — непустое левое.

## 10.6. Представление деревьев поиска в компьютере

В отличие от куч, без дополнительной памяти при кодировке деревьев поиска не обойтись. В каждой вершине дерева понадобится ссылка на левого потомка, ссылка на правого потомка и еще одна переменная для хранения содержимого этой вершины. Обозначим через  $n$  число вершин в дереве поиска и предположим, что все они пронумерованы от 0 до  $n - 1$ . На рис. 10.8 изображено ранее встречавшееся нам дерево с проставленными номерами вершин.

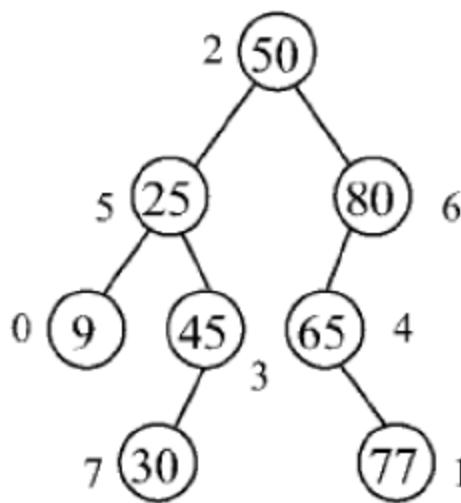


Рис. 10.8

Для кодировки дерева нам понадобится три массива:

$L[0..n-1]$  – массив для хранения ссылок на левых потомков:

$L[i]$  содержит номер вершины – левого потомка вершины  $i$  или  $-1$ , если левого потомка нет;

$R[0..n-1]$  – массив для хранения ссылок на правых потомков:

$R[i]$  содержит номер вершины – правого потомка вершины  $i$  или  $-1$ , если правого потомка нет;

$A[0..n-1]$  – массив для хранения содержимого вершин дерева:

$A[i]$  содержит число, находящееся в вершине  $i$ .

Кроме того, понадобится переменная  $root$ , в которой записан номер корня.

На рис. 10.9 приведено изображенное выше дерево, закодированное в соответствии с этими правилами. Кодировка простая и естественная, но есть у нее и недостатки: в массивах  $L$  и  $R$  приходится хранить много «лишних» ссылок ( $-1$ ) на отсутствующих потомков. Увы, придется смириться с этими «издержками производства».

	<i>L</i>	<i>R</i>	<i>A</i>
0	-1	-1	9
1	-1	-1	77
<i>root</i> → 2	5	6	50
3	7	-1	45
4	-1	1	65
5	0	3	25
6	4	-1	80
<i>n</i> - 1	-1	-1	30

Рис. 10.9

## 10.7. Программная реализация поиска и добавления

Процедуры поиска и добавления элемента очень просты и компактны. Как правило, нет смысла оформлять их в виде отдельных процедур. Соответствующие операторы обычно записываются непосредственно в теле программы. Для наглядности запишем эти операции в виде процедур, из чего совсем не следует, что так надо поступать всегда.

Будем предполагать, что дерево поиска закодировано, как было показано выше: с помощью трех массивов L, R, A и переменных root и n. Все эти переменные будем считать глобальными. Комментарии, как обычно, находятся в теле процедур.

```
//ПОИСК ПО ДЕРЕВУ
//ПРОВЕРЯЕМ, ЕСТЬ ЛИ В ДЕРЕВЕ ПОИСКА ЭЛЕМЕНТ x

Search ( x )
{
    //ЦИКЛ ОТ КОРНЯ ДЕРЕВА
    for( i = root; i != -1; )
    {
        if ( x == A[i] ) //элемент x нашёлся
            return ( 0 );
        if ( x < A[i] )
            i = L[i];      //к левому потомку
        else
            i = R[i];      //к правому потомку
        //переходим к следующей итерации
    } //конец цикла

    //элемента x нет в дереве
    return ( 1 );
} //Конец поиска:
//если поиск успешен, возвращается «0»,
//если нет – «1»
```

Добавление элемента в дерево поиска не намного сложнее. Сначала надо проверить, есть ли искомый элемент в дереве, и если нет, то следует «подвесить» его с нужной стороны под последней пройденной вершиной. Но при этом нужно решить две технические проблемы.

Во-первых, надо определить, в какую позицию в массиве записывать новый элемент. Будем предполагать, что на каждый массив выделено место «с запасом», то есть реальная длина массивов больше текущего размера дерева. Тогда достаточно будет увеличить  $n$  – число вершин в дереве – и записать новую вершину в конец массива. В дальнейшем мы разберем чуть более рациональную стратегию работы с резервной памятью, но пока этого достаточно. Для «универсальности» будем считать, что индекс свободной позиции в массиве определяет процедура `GetRoom()`, которая пока будет совсем простой, а потом мы сделаем ее чуть более сложной, но зато эффективной.

```
//ВЫДЕЛЕНИЕ МЕСТА (временная процедура)
Get_Room( )
{
    n++; //число вершин увеличилось
    return (n-1); //свободное место
}
```

Чтобы избежать технических деталей, которые каждый при необходимости может добавить сам, мы сознательно опускаем проверку выхода индекса за границы массива. Предполагаем, что такого не может быть.

Если механически перенести в процедуру добавления цикл из процедуры поиска, то возникнет одна проблема. Когда переменная  $i$  станет равной  $-1$  и понадобится добавлять новый элемент в дерево, то информация о последней пройденной при поиске вершине (хранившаяся в переменной  $i$ ) уже будет стерта. Эту проблему решить несложно: надо запоминать вершину, предшествующую текущей вершине в ходе поиска. Хорошо бы еще запоминать, куда мы из предыдущей вершины пошли: налево или направо, но это, как мы сейчас убедимся, не обязательно. В этом разделе нам не раз придется «подвешивать» одну вершину под другой, выполняя при этом стандартные действия. Чтобы не записывать много раз одно и то же, оформим это действие в виде стандартной процедуры.

```
//ПОДВЕСИТЬ ВЕРШИНУ
//ВЕРШИНУ i ПОДВЕШИВАЕМ ПОД ВЕРШИНУ p
//С ТОЙ СТОРОНЫ, ГДЕ ДОЛЖНА БЫТЬ
//ВЕРШИНА, СОДЕРЖАЩАЯ x
//ЕСЛИ p=-1, ТО i СТАНОВИТСЯ КОРНЕМ
```

```

Hang( i, x, p )
{ if ( p == -1 ) //дерево пустое
    root = i; //i становится корнем
else
    if ( x < A[p] )
        L[p] = i; //подвесили i слева от p
    else
        R[p] = i; //подвесили i справа от p
}

```

Теперь совсем легко написать процедуру добавления нового элемента в дерево поиска. Через *i* будем обозначать текущую вершину, в которой происходит сравнение, а через *p* – предшествующую ей.

```

//ПОИСК И ДОБАВЛЕНИЕ
//ПРОВЕРЯЕМ, ЕСТЬ ЛИ В ДЕРЕВЕ ПОИСКА ЭЛЕМЕНТ x
//ЕСЛИ НЕТ, ТО ДОБАВЛЯЕМ ЕГО НА НОВОЕ МЕСТО

Add ( x )
{
    //ЦИКЛ ОТ КОРНЯ ДЕРЕВА
    //i - текущая вершина
    //p - предыдущая
    for( i = root, p = -1; i != -1; )
    { if ( x == A[i] ) //элемент x нашёлся
        return ( 0 );
        p = i; //готовимся к переходу
        if ( x < A[i] )
            i = L[i]; //к левому потомку
        else
            i = R[i]; //к правому потомку
        //переходим к следующей итерации
    } //конец цикла

    //элемента x нет в дереве – надо добавить
    i = Get_Room( ); //место для новой вершины
    A[i] = x; //запоминаем значение
    L[i] = R[i] = -1; //новая вершина – лист
    Hang ( i, x, p ); //подвесили i под p
    return ( 1 );
}

```

```

} //Конец поиска и добавления:
//если нашли, возвращаем «0»,
//если элемент добавлен – «1»

```

Итак, мы научились искать элемент и, если нужно, добавлять его в дерево. Осталось освоить удаление вершины. Но прежде чем перейти к этой операции, немного отвлечемся. Давно известно, что «правильно» написанная программа «сумнее своего автора», то есть она верно работает даже в тех случаях, которые автор забыл предусмотреть. Давайте проверим, что произойдет, если дерево пусто, то есть  $n=0$ , а  $root=-1$ . Нетрудно убедиться, что все будет хорошо. Цикл завершится сразу, в качестве позиции для новой вершины будет определен ноль, а при подвешивании эта вершина станет корнем. Мы можем быть довольны. Хотя при кодировании мы и не подумали про пустое дерево, но «правильно» написанная программа не подвела нас.

## 10.8. Удаление

Удалить элемент из дерева сложнее, чем добавить, хотя и ненамного. Возможны три случая: удаляемый элемент находится в вершине с 0, 1 или 2 потомками.

### 10.8.1. Удаление элемента, находящегося в листе – вершине без потомков

Рассмотрим самый простой случай. Для удаления такого элемента  $x$  нужно удалить из дерева поиска вершину, его содержащую. А для этого достаточно у предка «обнулить» ссылку (левую или правую), указывающую на удаляемую вершину. Больше ничего в дереве менять не надо. Очевидно, что получившееся дерево остается деревом поиска. На рис. 10.10 через  $i$  обозначена удаляемая вершина, а через  $p$  – ее предок.

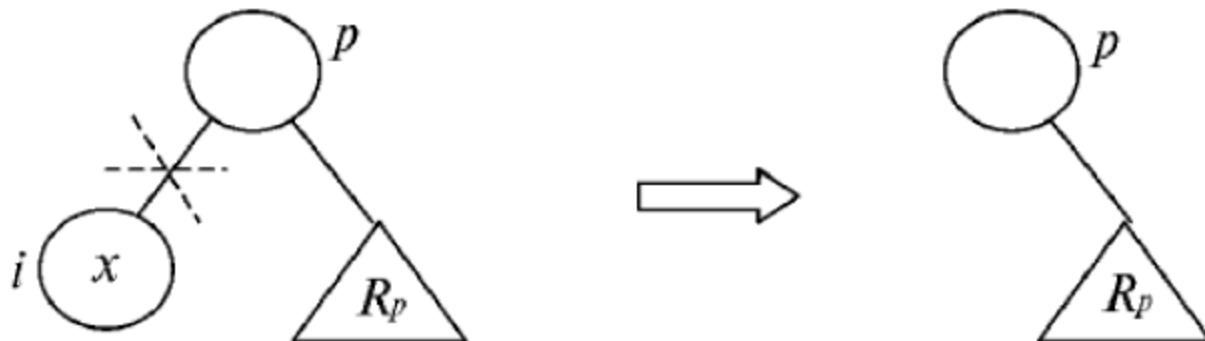


Рис. 10.10

На этом рисунке вершина  $i$  является левым потомком  $p$ , а через  $R_p$  обозначено поддерево с корнем в правом потомке  $p$ . Если бы вершина  $i$  была правым потомком  $p$ , то картинка была бы симметричной. Для всех последующих рисунков без потери общности будем считать удаляемую вершину  $i$  левым потомком своего предка  $p$ , а правое поддерево  $p$ , чтобы не загромождать рисунок, изображать не будем.

### 10.8.2. Удаление элемента, находящегося в вершине с одним потомком

Это чуть сложней, но тоже достаточно просто. В дереве на место вершины, содержащей удаляемый элемент  $x$ , помещаем ее единственного потомка, то есть ссылка из  $p$  на  $i$  не обнуляется, а потомком  $p$  становится единственный потомок  $i$ . На рис. 10.11 правый потомок  $i$  обозначен через  $k$ , а треугольник, содержащий  $k$ , изображает поддерево, корнем которого является вершина  $k$ .

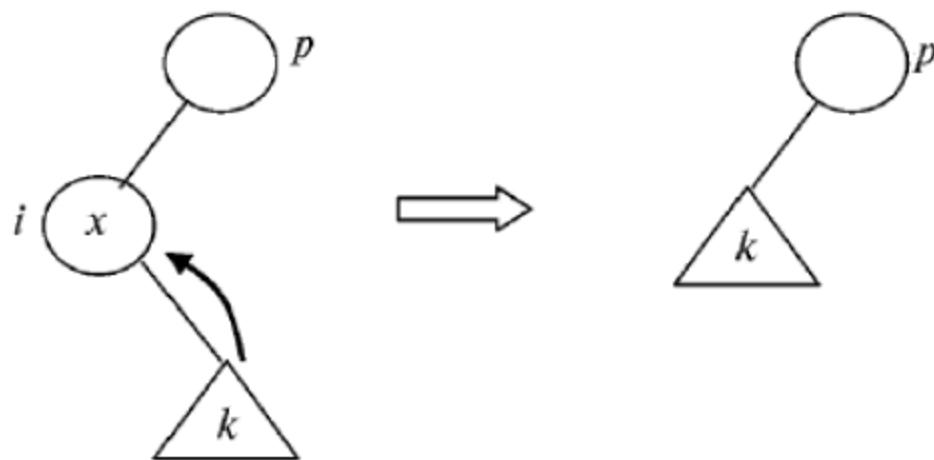


Рис. 10.11

Если бы единственным потомком  $i$  был левый потомок, то точно так же он оказался бы в дереве на месте, ранее занимаемом  $i$ .

В некотором смысле, удаляя вершину без потомков, в первом случае мы делали то же самое, что и сейчас. Ссылка из  $p$  на  $i$  была заменена пустой ссылкой, которую можно трактовать как ссылку на одного из потомков  $i$ , которых все равно у листа нет.

Новое дерево осталось деревом поиска, так как поддерево с корнем в  $k$  как было, так и осталось с той же стороны от  $p$ , с какой находилась вершина  $i$ . Таким образом, соотношение поиска не нарушились.

### 10.8.3. Удаление элемента, находящегося в вершине с двумя потомками

В этом случае придется потрудиться немного больше. Поскольку у вершины два потомка, то просто удалить ее из дерева нельзя: потомков два, и оба они стать потомками  $p$  не могут – есть только одно вакантное место. Приходится оставлять вершину  $i$  на месте, заменив ее содержимое  $x$  на элемент, хранящийся в какой-то другой вершине дерева. Чтобы заменить содержимое вершины  $i$  и при этом сохранить в ней соотношение поиска, надо выбрать такой элемент, который был бы одновременно больше всех элементов  $L_i$  – левого поддерева  $i$  и меньше всех элементов  $R_i$  – правого поддерева  $i$ . Таких элементов два: самый маленький элемент правого поддерева и самый большой левого поддерева. Возьмем для определенности  $y$  – самый маленький элемент правого поддерева. Если переместить  $y$  в вершину  $i$ , а вершину, где находился  $y$ , удалить, то в получившемся дереве соотношения поиска будут соблюдены. Поскольку  $y$  находится в правом поддереве  $i$ , то он больше содержимого корня, а следовательно, больше любого элемента из левого поддерева. Но поскольку  $y$  – минимальный элемент правого поддерева, то он меньше любого элемента, оставшегося в правом поддереве.

Таким образом, чтобы удалить элемент, находящийся в вершине с двумя потомками, нужно переместить в эту вершину минимальный элемент правого поддерева и удалить вершину, ранее содержащую этот элемент. Поскольку у «минимальной вершины» любого поддерева не может быть левого потомка (иначе она не была бы минимальной), то такую вершину мы удалять умеем, так как у нее не более одного потомка.

Из приведенных рассуждений следует, что после описанного преобразования дерево останется деревом поиска. Проиллюстрируем на рис. 10.12 операцию удаления элемента, содержащегося в вершине с двумя потомками.

Мы убедились, что в удалении элемента, находящегося в вершине с двумя потомками, тоже ничего особо сложного нет. В вырожденном варианте, когда у вершины  $k$  нет левого потомка, по существу ничего не меняется. В этом случае минимальный элемент правого поддерева содержится в самой вершине  $k$ , он перемещается в  $i$ , а на место  $k$  попадает, как и в общем случае, вершина  $l$ , являющаяся правым потомком  $k$ .

Очевидно, что сложность процедуры удаления по порядку величины не превышает высоты дерева и составляет  $O(h)$ .

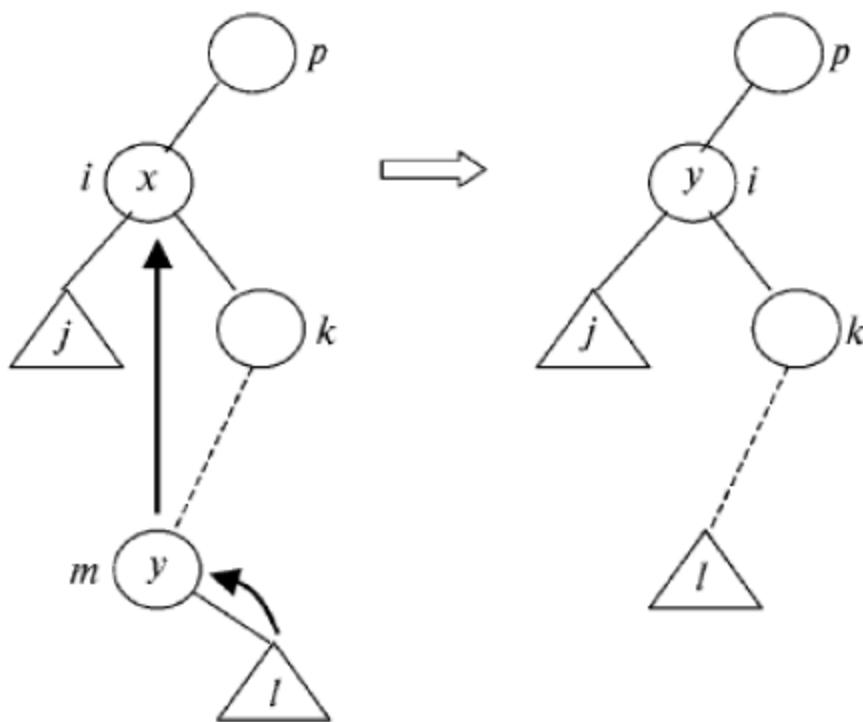


Рис. 10.12

### 10.9. Программная реализация удаления

Процедура удаления вершины не сложная, но более «хлопотливая» по сравнению с поиском и добавлением. Тем не менее приведем ее полностью. Но прежде, чем писать эту процедуру, необходимо решить один технический вопрос. Информация, находящаяся в удаляемой вершине, располагалась в некоторой позиции массивов. Что теперь делать с освободившимся местом? Терять его было бы нерационально, если операции добавления и удаления идут вперемежку. Пустое место понадобится нам, чтобы разместить новые вершины. Выход очень прост. Надо организовать список, состоящий из свободных позиций. Если позиция освобождается – включаем ее в список, а когда требуется место для размещения новой вершины – берем его из списка. Для организации такого списка даже не понадобится дополнительной памяти. Ссылки можно расположить в пустующих ячейках массива  $L$ , которые не используются, потому что эти позиции включены в список свободных мест. Единственное, что понадобится дополнительно – это голова списка:  $head$ . На рис. 10.13 показано дерево из трех вершин, располагающееся в массиве, рассчитанном на шесть позиций. Ссылки списка свободных позиций изображены стрелками.

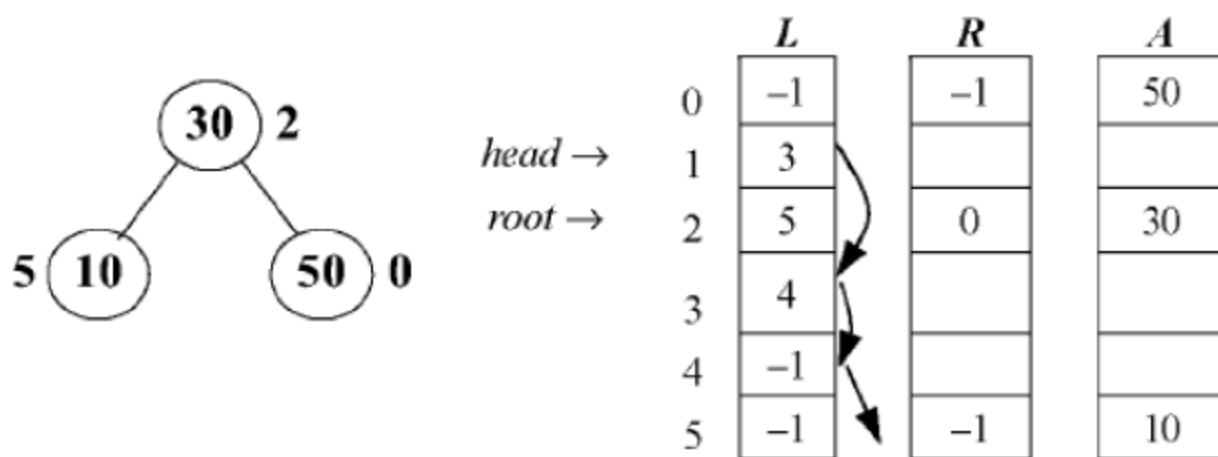


Рис. 10.13

Для работы со списком свободных позиций нам понадобится три небольших процедуры: инициализация списка, сохранение освободившейся позиции (включение элемента в список) и выделение места (исключение элемента из списка). Последняя процедура заменит приведенную ранее временную процедуру выделения места, причем основную процедуру поиска и добавления менять не придется. Все процедуры являются элементарными манипуляциями с обычными списками.

```

//ИНИЦИАЛИЗАЦИЯ (ПУСТОЕ ДЕРЕВО)
Init_List( )
{
    head = 0;           //начало списка
    for ( k = 0; k < n-1; k++ )
        L[k] = k+1;    //ссылка на следующего
    L[n-1] = -1;        //конец списка
}

//СОХРАНЕНИЕ МЕСТА
Take_Room( k )
{
    L[k] = head;      //новую позицию
    head = k;          //в начало списка
}

//ВЫДЕЛЕНИЕ МЕСТА
Get_Room( )
{
    k = head;         //свободное место
                      //или -1, если места нет
}

```

```

if ( head != -1 )
head = L[head]; // к следующему элементу списка
return ( k ); // возвращаем свободное место
}

```

Теперь все готово для того, чтобы написать процедуру удаления элемента из дерева поиска. Начнем ее, как и при добавлении, с поиска соответствующего элемента.

```

//ПОИСК И УДАЛЕНИЕ
//ПРОВЕРЯЕМ, ЕСТЬ ЛИ В ДЕРЕВЕ ПОИСКА ЭЛЕМЕНТ x
//ЕСЛИ ЕСТЬ, ТО УДАЛЯЕМ

Remove ( x )
{
    //ЦИКЛ ОТ КОРНЯ ДЕРЕВА
    //i - текущая вершина
    //p - предыдущая
    for( i = root, p = -1; i != -1; )
    {
        if ( x == A[i] ) //элемент x нашёлся
            break;
        p = i;           //готовимся к переходу
        if ( x < A[i] )
            i = L[i];     //к левому потомку
        else
            i = R[i];     //к правому потомку
        //переходим к следующей итерации
    } //конец цикла: i=-1 - элемента нет

    if ( i != -1 )      //есть что удалять
        {//удаление элемента x из вершины i
        if ( L[i] == -1 ) //нет левого потомка
            //вешаем правого потомка i под p
            //с той же стороны, где было i
            Hang ( R[i], x, p );
            //Это работает верно и когда
            //правого потомка нет
        else             //есть левый потомок
            if ( R[i] == -1 ) //нет правого потомка
                //вешаем левого потомка i под p
        }
    }
}

```

```

        //с той же стороны, где было i
        Hang ( L[i], x, p );
    else           //есть два потомка
    { //ищем минимум в правом поддереве i
        //m - текущая вершина
        //pm - предыдущая
        for ( m = R[i], pm = i;
              L[m] != -1;
              pm = m, m = L[m]
            ); //тело цикла пусто
        //m - вершина, содержащая миним.
        //элемент правого поддерева
        A[i] = A[m]; //миним. элемент
                      //перемещаем в i
        i = m;      //будем удалять m, а не i
        //вешаем правого потомка m под pm
        //с той же стороны, где было m
        Hang ( R[m], A[m], pm );
                      //работает верно и когда потомка
                      //у m нет, и когда pm=i
    } //с двумя потомками тоже справились

    Take_Room ( i ); //сохраняем
                      //освободившееся место
    return ( 0 );   //удаление прошло успешно
}
else           //удалять нечего
    return ( 1 );
} //Конец поиска и удаления:
//если нашли и удалили, возвращаем «0»,
//если не нашли - «1»

```

Программа, как видите, скучноватая, но, в сущности, простая.

## 10.10. Обход дерева поиска

Дерево поиска содержит информацию, которая позволяет с минимальными затратами получить перечень содержащихся в нем элементов, упорядоченных по возрастанию или по убыванию.

Для всего дерева поиска и любого его поддерева выполняется «соотношение поиска»: содержимое корня больше содержимого лю-

боя вершины левого поддерева, но меньше содержимого любой вершины правого поддерева. Следовательно, если нужно обойти вершины дерева поиска в порядке возрастания, то сначала надо обойти вершины левого поддерева, затем – корень, а потом – все вершины правого поддерева, то есть последовательность обхода вершин должна быть такой:  $(L, root, R)$ , как это показано на рис. 10.14. Точно такое же утверждение применимо к любому поддереву дерева поиска: сначала надо обойти левое поддерево его корня, потом корень, а потом – правое поддерево. То, что для обхода любого поддерева можно применять в точности ту же процедуру, что и при обходе всего дерева, немедленно наводит на мысль, что самой простой реализацией алгоритма обхода является рекурсивная процедура. Так оно и есть.

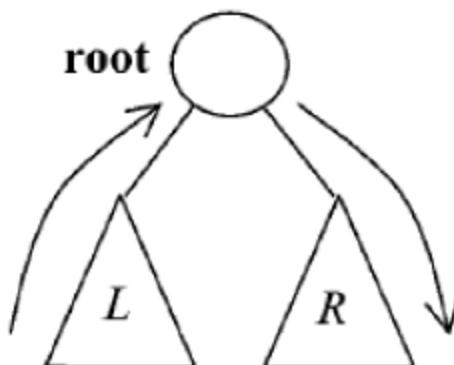


Рис. 10.14

```

//ОБХОД ДЕРЕВА ПОИСКА (рекурсивная процедура)
//СОВЕРШАЕТ ОБХОД ДЕРЕВА ПОИСКА С КОРНЕМ В
//ВЕРШИНЕ i
Tour( i )
{
    //Обход левого поддерева, если оно не пусто
    if ( L[i] != -1 ) Tour ( L[i] );

    Возьми ( i );      //настала очередь корня

    //Обход правого поддерева, если оно не пусто
    if ( R[i] != -1 ) Tour ( R[i] );
} //вот и все!
  
```

Предполагается, что тот, кому понадобился обход дерева, напишет процедуру `Возьми(i)`, обрабатывающую очередную поступившую в ее распоряжение вершину.

Теперь для того чтобы выдать «заказчику» все элементы дерева поиска в порядке возрастания, достаточно вызвать процедуру `Tour`, передав ей корень дерева в качестве параметра: `Tour (root)`.

Реализованный алгоритм обхода обладает линейной сложностью. Это следует, например, из того, что число совершаемых операций при каждом вызове процедуры `Tour` – константа. При этом одна вершина оказывается пройденной. Следовательно, сложность этого алгоритма обхода дерева поиска –  $O(n)$ , где  $n$  – число вершин дерева.

Методы организации работы рекурсивных программ хорошо известны: постоянный код и стек локальных переменных. Загадок здесь нет никаких. Но все же в каждой рекурсивной процедуре есть какая-то тайна. Людям консервативных взглядов, к которым, несомненно, относится автор, иногда трудно на интуитивном уровне понять, «как же это все-таки работает». Конечно, во многих случаях никак иначе писать программу не следует – именно рекурсия дает наиболее рациональный способ кодирования алгоритма. Но все-таки иногда полезно попробовать записать тот же самый алгоритм «честно», без рекурсий. Попробуем это сделать.

Совсем без стека не обойтись. Придется использовать дополнительный массив `S [0 .. h-1]`, где  $h$  – высота дерева. Он будет нужен для запоминания пройденного от корня пути. В него будут записаны те и только те вершины, из которых при обходе дерева был сделан шаг влево, и при этом сама вершина еще не обработана. Переменная `t` служит указателем на первую свободную ячейку стека.

```
//ОБХОД ДЕРЕВА ПОИСКА (итеративная процедура)
//СОВЕРШАЕТ ОВХОД ДЕРЕВА ПОИСКА
```

```
t = 0;                                //стек пуст
i = root;                               //текущая вершина
L1:
//переходим к левому потомку,
//пока это возможно
for (
;
(*)      L[i] != -1;
        i = L[i]
)
{
    S[t] = i; t++; //текущую вершину в стек
```

```

    }

L2:
    Возьми ( i ); //отдаем корень поддерева
    if ( R[i] == -1 ) // можно ли шагнуть вправо
        { //правое дерево пусто
            //делаем шаг вверх
        (#)      if ( t == 0 ) //стек пуст
            exit ( ) //обход закончен
        //стек не пуст
        i = S[t]; t--; //шаг вверх
        goto L2; //ужас!!
    }
else
    { //правое дерево не пусто
        i = R[i]; //шаг вправо
        goto L1; //кошмар!!
    }

```

Программа красивая, но непонятная. При взгляде на нее не становится очевидно, что она выполняет обход дерева поиска. Конечно, каждый может проверить это на примерах. Однако примеры – это еще не окончательный аргумент. Надо доказать, что эта программа работает правильно. Давайте попробуем это сделать.

Будем доказывать это утверждение по индукции. Докажем, что если для дерева поиска в вершине  $i$  выполнена строка программы, помеченная знаком (\*), то после этого программа выполнит обход вершин поддерева с корнем в  $i$  в порядке возрастания содержащихся в этих вершинах элементов. После завершения этого обхода будет выполнена строка программы, помеченная символом (#), в результате чего, если  $i$  была корнем всего дерева, работа программы завершится, а если  $i$  не была корнем, из стека будет извлечена вершина, записанная в него до выполнения строки (\*) с вершиной  $i$ .

Для дерева, состоящего из одной вершины, справедливость утверждения очевидна. Вначале стек пуст. Поскольку  $L[i] = -1$ ,  $i$  не будет занесено в стек, выполнение цикла while сразу закончится, после чего будет выполнен оператор Возьми (i). Затем, поскольку  $R[i] = -1$  и стек пуст, работа программы завершится. Таким образом, утверждение верно при  $n = 1$ .

Предположим, что оно верно для всех деревьев поиска с числом вершин, меньшим  $n$ . Докажем, что для дерева поиска с  $n$  вершинами это утверждение тоже справедливо.

В начале работы программы стек пуст. Исполняем строку (\*) со значением  $i$ , равным  $root$  – корню дерева.

Если у  $i$  есть левый потомок, то заносим  $root$  в стек, переходим в  $L[i]$  и опять выполняем строку (\*) со значением  $i = L[i]$ . Число вершин в поддереве с корнем  $L[i]$  меньше  $n$ , поэтому по предположению индукции программа обойдет все его вершины в порядке возрастания элементов. Затем будет выполнена строка (#), после чего  $i$  станет равной находившейся в стеке переменной  $root$ , стек станет пустым, и мы перейдем на метку  $L2$ .

Если у  $i$  нет левого потомка, то выполнение цикла *while* сразу будет закончено, ни одной вершины не будет выдано,  $i$  сохранит свое значение  $root$ , стек останется пустым и мы приедем к метке  $L2$ .

После этого в обоих случаях будет выполнен оператор *Возьми* ( $root$ ). Если у корня нет правого потомка, то на этом выполнение программы будет завершено. При этом все элементы левого поддерева, а потом корень – то есть все дерево – будут выданы в порядке возрастания.

Если у корня есть правый потомок, то мы перейдем на метку  $L1$ , а значит, на строку (\*) с пустым стеком и со значением  $i = R[i]$ . По предположению индукции мы обойдем правое дерево в порядке возрастания элементов и после этого выйдем на строку (#). Стек пуст, поэтому программа будет завершена. В этом случае мы сначала обошли левое поддерево корня, потом прошли корень, а потом обошли его правое поддерево.

На этом доказательство корректности программы завершается.

Конечно, никакое доказательство не может гарантировать правильность программы. И в доказательствах бывают ошибки, и ни одно доказательство не сможет выявить ошибку, когда, глядя на переменную  $j$ , вы упорно думаете, что это  $i$ . Но, тем не менее, такие доказательства для логически сложных программ могут оказаться полезны для выявления «тонких ошибок». Я не призываю доказывать корректность каждой написанной программы. В большинстве случаев в этом нет никакого смысла. Но владеть таким инструментом необходимо.

И еще один сюжет, связанный с последней программой обхода дерева поиска. В ней есть элементы, которые в «приличном программистском обществе» всячески осуждаются, а именно: метки и операторы *goto*. Позволю себе высказаться по этому поводу, хотя вопрос, пожалуй, уже утратил большую часть своей остроты. Безусловно, операторов *goto* надо, по возможности, избегать. Их неуме-

ренное употребление делают программу трудночитаемой и плохо отлаживаемой. Однако бывают случаи, когда использование меток и операторов перехода, наоборот, упрощают программу, а чтобы их избежать, приходится прибегать к различным ухищрениям.

Автор резюмировал бы так: старайтесь избегать меток и операторов *goto*, но если надо – употребляйте, не стесняясь. Это грех, но небольшой. В качестве аналогии можно привести употребление не-нормативной лексики в разговоре. Если через каждые два слова используются известные всем термины, то трудно признать такую речь привлекательной. Однако ничто не может заменить крепкого слова, иногда сказанного к месту. То же относится и к оператору *goto*.

Впрочем, приведенную выше программу можно написать и без *goto*, и при этом более компактно, хотя и еще менее понятно.

```
//ОБХОД ДЕРЕВА ПОИСКА
// (итеративная процедура без goto)

t = 0;
for ( i = root; ; i = R[i] )
{ for ( ; L[i] != -1; i = L[i] ) S[t++] = i;
  while ( 1 )
  { Возьми ( i );
    if ( R[i] != -1 ) break;
    if ( t == 0 ) exit ();
    i = S[t--];
  }
}
```

При внимательном рассмотрении можно убедиться, что эта программа полностью эквивалентна программе, приведенной ранее. Каждая из них больше нравится, каждый решает для себя сам. Дело вкуса, особенно, если он есть.

## **БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

- Адельсон-Вельский Г.М., Диниц Е.А., Карзанов А.В.* Потоковые алгоритмы. М.: Наука, 1975. 119 с.
- Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.* М.: Вильямс, 2005. 1296 с.
- Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 535 с.
- Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982. 416 с.
- Кристофидес Н.* Теория графов. Алгоритмический подход. М.: Мир, 1978. 432 с.
- Липский В.* Комбинаторика для программистов. М.: Мир, 1988. 211 с.
- Пападимитриу Х., Стайглиц К.* Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1985. 510 с.
- Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980. 473 с.
- Романовский И.В.* Алгоритмы решения экстремальных задач. М.: Наука, 1977. 351 с.
- Романовский И.В.* Дискретный анализ. СПб.: Невский диалект, 2001. 240 с.

*ЧЕРКАССКИЙ Борис Васильевич*

## **КОМБИНАТОРНЫЕ АЛГОРИТМЫ**

### **Курс лекций**

Редактор *T.A. Кравченко*

Компьютерная верстка *M.A. Шамариной*

---

Подписано в печать 21.08.06    Бумага офсетная

Формат 60 × 90  $\frac{1}{16}$                       Печать офсетная    Уч.-изд. л. 9,93

Рег. № 788                              Тираж 200 экз.    Заказ 1133

---

Московский государственный институт стали и сплавов,  
119049, Москва, Ленинский пр-т, 4

Издательство «Учеба» МИСиС,  
117419, Москва, ул. Орджоникидзе, 8/9  
Тел.: 954-73-94, 954-19-22

Отпечатано в типографии издательства «Учеба» МИСиС,  
117419, Москва, ул. Орджоникидзе, 8/9