**Структурой данных** называется способ организации данных в памяти компьютера (или на диске). Примерами **структур данных** являются массивы, связанные списки, стеки, двоичные деревья, хеш-таблицы и т. д.

**Алгоритмы** обеспечивают выполнение различных операций с этими структурами — например, поиск определенного элемента данных или сортировку данных.

Какие **задачи** помогает решить хорошее знание структур данных и алгоритмов? Ситуации, в которых они могут пригодиться, можно приблизительно разделить на три категории:

1. Хранение реальных данных.
2. Инструментарий программиста.
3. Моделирование.

**Таблица 1.1. Характеристики структур данных**

|  |  |  |
| --- | --- | --- |
| **Структура данных** | **Достоинства** | **Недостатки** |
| Массив | Быстрая вставка, очень быстрый доступ (если известен индекс) | Медленный поиск, медленное удаление, фиксированный размер |
| Упорядоченный массив | Поиск выполняется быстрее, чем в обычном массиве | Медленная вставка и удаление, фиксированный размер |
| Стек | Доступ в порядке LIFO («последним пришел, первым вышел») | Медленный доступ к другим элементам |
| Очередь | Доступ в порядке FIFO («первым пришел, первым вышел»») | Медленный доступ к другим элементам |
| Связанный список | Быстрая вставка, быстрое удаление | Медленный поиск |
| Двоичное дерево | Быстрый поиск, вставка, удаление (если дерево остается сбалансированным) | Сложный алгоритм удаления |
| Красно-черное дерево | Быстрый поиск, вставка, удаление. Дерево всегда сбалансировано | Сложность |
| Дерево 2-3-4 | Быстрый поиск, вставка, удаление. Дерево всегда сбалансировано. Аналогичные деревья хорошо подходят для хранения данных на диске | Сложность |
| Хеш-таблица | Очень быстрый доступ (если известен ключ) | Медленное удаление, медленный доступ (если ключ неизвестен), неэффективное использование памяти |
| Куча | Быстрая вставка, удаление, доступ к наибольшему элементу | Медленный доступ к другим элементам |
| Граф | Моделирование реальных ситуаций | Некоторые алгоритмы медленны и сложны |

Все структуры данных из табл. 1.1, за исключением массивов, могут рассматриваться как абстрактные типы данных (ADT).

Многие алгоритмы, описанные в книге, предназначены для работы с конкретными структурами данных.

Для большинства структур данных необходимо знать, как выполняются операции:

* Вставки нового элемента данных.
* Поиска заданного элемента.
* Удаления заданного элемента

**Двоичный поиск с методом find()**

Метод find() ищет заданный элемент многократным половинным делением диапазона элементов массива. Метод выглядит так:

**public int find(long searchKey)**

**{**

**int lowerBound = 0;**

**int upperBound = nElems-1;**

**int curIn;**

**while(true)**

**{**

**curIn = (lowerBound + upperBound ) / 2;**

**if(a[curIn]==searchKey)**

**return curIn; // Элемент найден**

**else if(lowerBound > upperBound)**

**return nElems; // Элемент не найден**

**else // Деление диапазона**

**{**

**if(a[curIn] < searchKey)**

**lowerBound = curIn + 1; // В верхней половине**

**else**

**upperBound = curIn - 1; // В нижней половине**

**}**

**}**

**}**

В самом начале метода переменным lowerBound и upperBound приисваиваются ин-

дексы первой и последней занятой ячейки массива. Эти переменные определяют

диапазон, в котором может находиться элемент с искомым ключом searchKey. Затем

в цикле while текущий индекс curIn устанавливается в середину этого диапазона.

Если повезет, curIn сразу укажет на искомый элемент, поэтому мы сначала

проверяем эту возможность. Если элемент найден, мы возвращаем его индекс curIn.

При каждой итерации цикла диапазон уменьшается вдвое. В конечном итоге он со-

кратится настолько, что дальнейшее деление станет невозможным. Эта ситуация

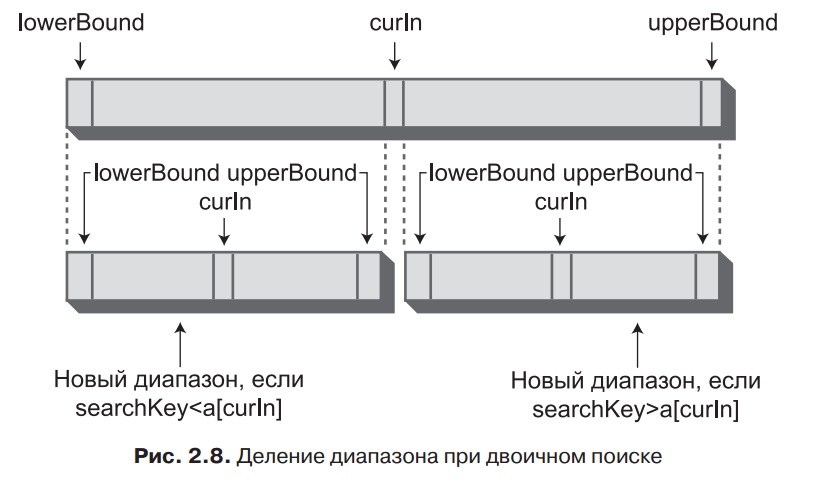
проверяется в следующей команде: если lowerBound больше upperBound, значит, поиск завершен. (Если индексы равны, то диапазон состоит из одного элемента, и понадобится еще одна итерация цикла.) Поиск не может продолжаться без диапазона; если заданный элемент не найден, метод возвращает nElems, общее количество элементов. Это значение не является действительным индексом, потому что последняя заполненная ячейка массива имеет индекс nElems-1. Пользователь класса интерпретирует это значение как признак того, что элемент не был найден.

Если curIn не указывает на искомый элемент, а диапазон все еще остается до-

статочно большим, то его следует разделить пополам. Мы сравниваем значение

по текущему индексу a[curIn], находящееся в середине диапазона, с искомым

значением searchKey.



Если значение searchKey больше, значит, поиск должен осуществляться в верх-

ней половине диапазона. Соответственно lowerBound нужно присвоить значение

curIn, а точнее индекс следующей ячейки, потому что ячейка curIn уже была про-

верена в начале цикла.

Если значение searchKey меньше a[curIn], то поиск следует ограничить нижней

половиной диапазона. Соответственно upperBound смещается до ячейки, пред-

шествующей curIn. На рис. 2.8 показано, как изменяется диапазон в этих двух

ситуациях.

**O-синтаксис**

Критерий сравнения (сложность ал­горитма) должен связывать скорость алгоритма с количеством элементов.

**Таблица 2.5. Время выполнения операций в O-синтаксисе**

|  |  |
| --- | --- |
| **Алгоритм** | **Время выполнения в O-синтаксисе** |
| Линейный поиск | O(N) |
| Двоичный поиск | O(log N) |
| Вставка в неупорядоченном массиве | O(1) |
| Вставка в упорядоченном массиве | O(N) |
| Удаление в неупорядоченном массиве | O(N) |
| Удаление в упорядоченном массиве | O(N) |

На рис. 2.9 показаны некоторые зависимости между временем выполнения

операции и количеством элементов. На основании этого графика можно (весьма

условно) оценить разные сложности:

O(1) — отлично,

O(log N) — хорошо,

O(N) —неплохо,

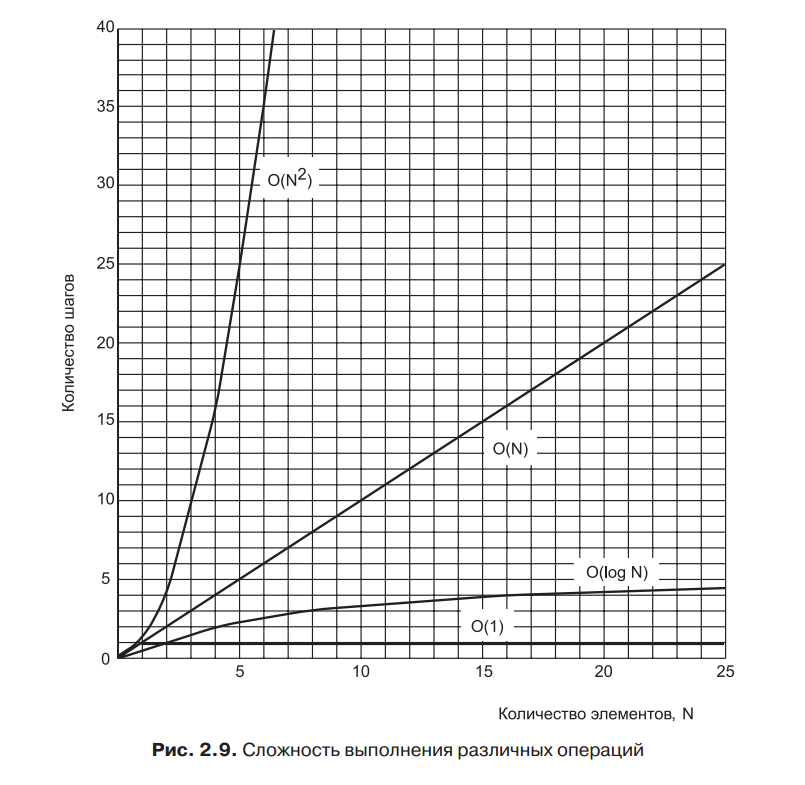
O(N2) — плохо

**O-синтаксис не дает конкретных цифр, а передает общий характер зависимо-**

**сти времени выполнения от количества элементов.** Это самый информативный

механизм сравнения алгоритмов, не считая разве что замера времени выполнения

в реальных условиях.



O-синтаксис предоставляет удобный способ сравнения скорости алгоритмов.

O-синтаксис описывает зависимость сложности алгоритма от количества элементов.

**Пузырьковая сортировка**

Пузырьковая сортировка известна своей низкой скоростью, однако на концептуальном уровне — это простейший алгоритм сортировки, и поэтому мы начнем изучение алгоритмов сортировки именно с него.

Пример пузырьковой сортировки

Представьте, что вы близоруки (как компьютерная программа) и видите не более

двух игроков одновременно и только если они стоят вплотную друг к другу. Как бы

вы отсортировали игроков при таких ограничениях? Допустим, команда состоит

из N игроков, а их позиции пронумерованы от 0 до N–1.

Алгоритм пузырьковой сортировки работает так: вы подходите к левому краю

шеренги (позиция 0) и сравниваете двух игроков в позициях 0 и 1. Если левый

игрок (позиция 0) выше, вы меняете их местами. Если выше правый игрок, они

остаются на своих местах. Затем вы переходите на одну позицию вправо и сравниваете игроков в позициях 1 и 2. И снова, если левый игрок выше, вы меняете их

местами. Схема сортировки показана на рис. 3.3.

Сортировка выполняется по следующим правилам:

1. Сравнить двух игроков.

2. Если левый игрок выше, поменять их местами.

3. Перейти на одну позицию вправо.

Перестановки продолжаются до тех пор, пока не будет достигнут правый край

шеренги. Сортировка еще не завершена, но по крайней мере самый высокий игрок

стоит в крайней правой позиции. Это происходит всегда: как только вы встретите

самого высокого игрока, вы будете переставлять его при каждом сравнении, пока

он (в конечном итоге) не перейдет в крайнюю правую позицию. Именно поэтому

алгоритм называется «пузырьковой сортировкой»: в ходе его работы самый боль-

шой элемент, словно пузырек в жидкости, всплывает до конца массива. На рис. 3.4

показано, как выглядит строй после завершения первого прохода.

При первом проходе по всем данным выполняется N–1 сравнений и от 0 до

N–1 перестановок (в зависимости от начального расположения игроков). Элемент

в конце массива находится на своем месте и снова перемещаться уже не будет.

Теперь алгоритм возвращается к началу шеренги и начинает следующий про-

ход. Он снова последовательно перемещается слева направо, сравнивая элементы

и переставляя их в случае необходимости. На этот раз перебор останавливается за

один элемент до конца, в позиции N–2, потому что последняя позиция (N–1) уже

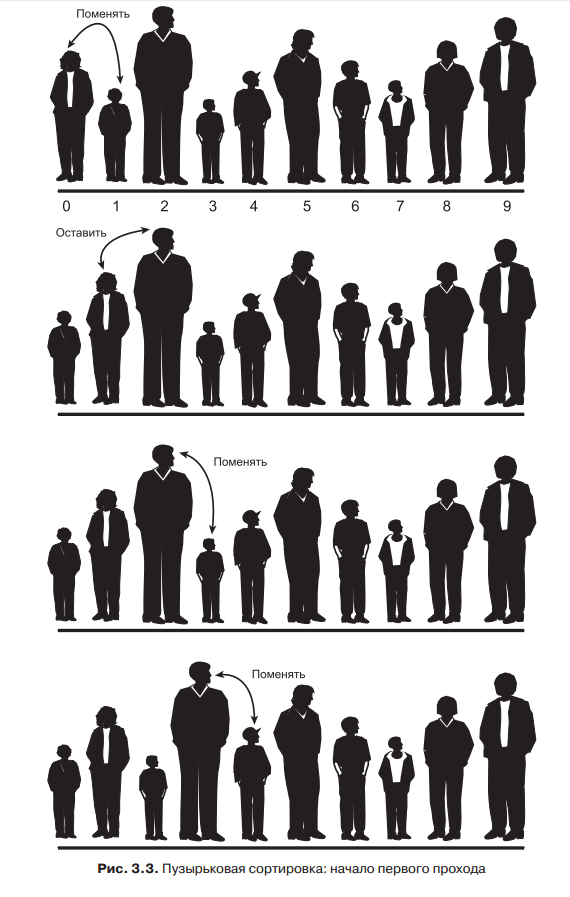
гарантированно содержит наибольший элемент.

Это правило можно сформулировать следующим образом:

4. После того как первый отсортированный элемент окажется на своем месте,

вернуться к началу массива.

Процесс продолжается до тех пор, пока все элементы не будут упорядочены.





**Реализация пузырьковой сортировки**

**на языке Java**

**private void swap(int one, int two)**

**{**

**long temp = a[one];**

**a[one] = a[two];**

**a[two] = temp;**

**}**

**public void bubbleSort()**

**{**

**int out, in;**

**for(out=nElems-1; out>1; out--) // Внешний цикл (обратный)**

**for(in=0; in<out; in++) // Внутренний цикл (прямой)**

**if( a[in] > a[in+1] ) // Порядок нарушен?**

**swap(in, in+1); // Поменять местами**

**}**

В результате сортировки наименьший элемент должен находиться в начале мас-

сива (индекс 0), а наибольший — в конце (индекс nElems-1). Счетчик цикла out внеш-

него цикла for начинает перебор с конца массива (в позиции nElems-1) и последо-

вательно уменьшается в процессе перебора. Элементы с индексами, большими out,

находятся на своих окончательных местах. Переменная out смещается влево после

каждого прохода, чтобы алгоритм не затрагивал уже отсортированные элементы.

Счетчик in начинается с начала массива и последовательно увеличивается при

каждой итерации внутреннего цикла вплоть до out. Внутренний цикл сравнивает

две ячейки, на которые указывают in и in+1, и если элемент в ячейке in больше

элемента в ячейке in+1 — меняет их местами.

**Сложность пузырьковой сортировки**

Количество как перестановок, так и сравнений пропорционально N2. Так как константы в O-синтаксисе не учитываются, можно сказать, что пузырьковая сортировка выполняется за время **O(N2).** Это достаточно медленно.

**Инварианты**

Во многих алгоритмах задействованы условия, которые остаются неизменными в ходе выполнения алгоритма. Такие условия называются *инвариантами.* Умение распознавать инварианты поможет понять суть алгоритма.

В программе **bubbleSort.java** инвариант определяет, что элементы данных справа от **out** отсортированы. Это условие остается истинным.

**-------------------------------------------------------------------------**

**Сортировка методом выбора**

Алгоритм сортировки методом выбора превосходит пузырьковую сортировку по

характеристикам — количество необходимых перестановок сокращается с **O(N2)** до

**O(N).** К сожалению, количество сравнений остается равным **O(N2).** Все же сортировка методом выбора обладает значительными преимуществами для больших записей, которые необходимо физически перемещать в памяти, в результате чего время перестановки оказывается намного более важным, чем время сравнения. (Это не относится к языку Java, в котором перемещаются ссылки, а не целые объекты.)

**Пример сортировки методом выбора**

Вернемся к примеру с бейсбольной командой. При сортировке методом выбора сравнение уже не ограничивается игроками, стоящими рядом друг с другом.

Следовательно, вам понадобится как-то сохранить рост игрока; например, можно

записать его в блокноте.

**Краткое описание**

В сортировке методом выбора вы последовательно перебираете всех игроков и выбираете (отсюда и название) самого низкорослого из них. Этот игрок меняется

местами с тем, который стоит в крайней левой позиции (0). Левый игрок отсортирован, и в дальнейшем уже перемещаться не будет. Обратите внимание: в этом алгоритме отсортированные игроки собираются слева (нижние индексы), тогда как при пузырьковой сортировке они собираются справа. Следующий проход начинается с позиции 1, а обнаруженный минимальный элемент меняется местами с элементом в позиции 1. Процесс продолжается до тех пор, пока не будут отсортированы все игроки.

**Более подробное описание**

Перебор начинается от левого края шеренги игроков. Запишите рост крайнего

левого игрока в блокнот и положите на землю полотенце перед этим игроком.

Затем сравните рост следующего игрока с ростом, записанным в блокноте. Если

новый игрок меньше, вычеркните рост первого игрока и впишите вместо него рост

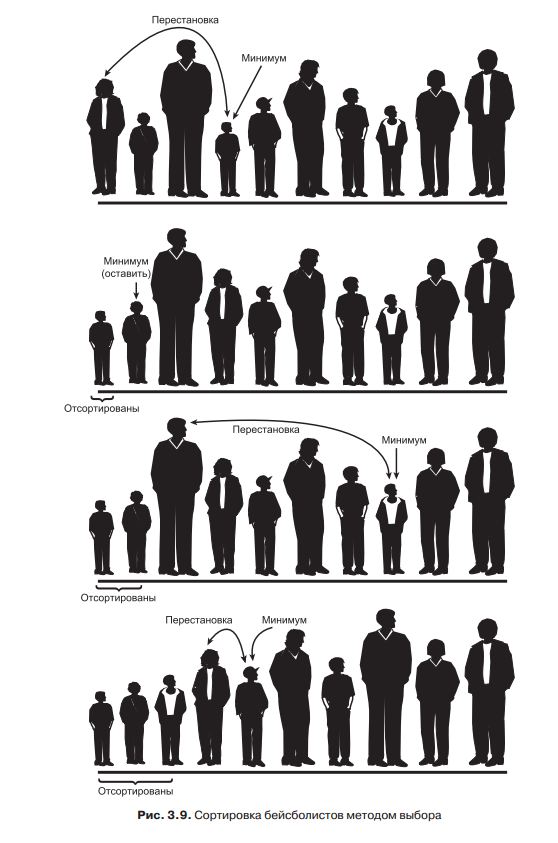
второго игрока — и переложите полотенце перед новым «самым низкорослым» (на

данный момент) игроком. Продолжайте двигаться вдоль шеренги, сравнивая рост

игроков с записанным минимумом. Обнаружив более низкого игрока, изменяйте

записанное в блокноте значение и перекладывайте полотенце. Когда проход будет

закончен, полотенце будет лежать перед самым низкорослым игроком.



Переставьте этого игрока с игроком, находящимся с левого края шеренги. Один

игрок отсортирован. Вы провели N–1 сравнений, но только одну перестановку.

С каждым последующим проходом еще один игрок сортируется и переходит на

левый край, и поиск следующего минимума производится среди меньшего коли-

чества игроков. На рис. 3.9 показано, как выглядит результат сортировки после

первых трех проходов.

**Реализация сортировки методом выбора на языке Java**

Листинг программы selectSort.java похож на листинг bubbleSort.java, разве что

контейнерный класс называется ArraySel вместо ArrayBub, а метод bubbleSort() за-

менен методом selectSort(). Вот как он выглядит:

**public void selectionSort()**

**{**

**int out, in, min;**

**for(out=0; out<nElems-1; out++) // Внешний цикл**

**{**

**min = out; // Минимум**

**for(in=out+1; in<nElems; in++) // Внутренний цикл**

**if(a[in] < a[min] ) // Если значение min больше,**

**min = in; // значит, найден новый минимум**

**swap(out, min); // Поменять их местами**

**}**

**}**

Внешний цикл (переменная out) начинает перебор с начала массива (индекс 0)

и продвигается к большим значениям индексов. Внутренний цикл (переменная in)

начинается с out и также двигается вправо.

В каждой новой позиции in сравниваются элементы a[in] и a[min]. Если a[in]

меньше, то min присваивается значение in. В конце внутреннего цикла min указывает

на минимальное значение текущего прохода, и алгоритм переставляет элементы

массива, на которые ссылаются out и min.

**Сложность сортировки методом выбора**

Сортировка методом выбора выполняет такое же количество сравнений, что

и пузырьковая сортировка: **N × (N–1)/2.** Для каждых 10 элементов данных

выполняются 45 сравнений, однако количество перестановок уменьшается. Для

больших значений N время сравнения является определяющим фактором, поэтому мы можем сказать, что сортировка методом выбора выполняется за время

**O(N2),** как и пузырьковая сортировка. Тем не менее совершенно очевидно, что сортировка методом выбора выполняется быстрее из-за меньшего количества перестановок. Для меньших значений N сортировка методом выбора может работать заметно быстрее, особенно если перестановка выполняется намного медленнее сравнения.

**Инвариант**

В программе selectSort.java элементы данных с индексами, меньшими либо равными out, всегда отсортированы.

**Сортировка методом вставки**

В большинстве случаев сортировка методом вставки является лучшим из элементарных алгоритмов сортировки, описанных в этой главе. Она также выполняется за время **O(N2),** но работает примерно вдвое быстрее пузырьковой сортировки, а в обычных ситуациях немного быстрее сортировки методом выбора. Кроме того, сортировка методом вставки не слишком сложна, хотя ее алгоритм немного сложнее двух других. Она часто используется на завершающей стадии более сложных алгоритмов, например быстрой сортировки.

**Пример сортировки методом вставки**

В исходном состоянии игроки бейсбольной команды выстроены в случайном порядке. Сортировку методом вставки легче описывать с середины процесса, когда команда уже наполовину отсортирована.

**Частичная сортировка**

На этой стадии где-то в середине шеренги устанавливается воображаемый маркер. (Допустим, вы положили красную футболку на землю перед игроком.) Игроки слева от маркера частично отсортированы; это означает, что каждый из них выше своего соседа слева. Однако это вовсе не означает, что игроки занимают свои итоговые позиции, потому что они, возможно, будут перемещены в результате вставки других, еще не отсортированных игроков. Обратите внимание: ситуация частичной сортировки не возникает в ходе пузырьковой сортировки и сортировки методом выбора. В этих алгоритмах подгруппа элементов данных полностью упорядочена в любой конкретный момент времени; в сортировке методом вставки подгруппа элементов упорядочена только частично.

**Помеченный игрок**

Игрок, рядом с которым лежит маркер (будем называть его «помеченным» игро-

ком), и все игроки справа от него еще не отсортированы (рис. 3.11, а).

Здесь мы собираемся вставить помеченного игрока в подходящее место (частич-

но) отсортированной подгруппы. Однако для этого необходимо сдвинуть некоторых

игроков подгруппы вправо, чтобы освободить место для вставки. Для этого по-

меченный игрок выводится из шеренги (в программе соответствующий элемент

данных сохраняется во временной переменной) — этот шаг показан на рис. 3.11, б.

Теперь отсортированные игроки сдвигаются, чтобы освободить место в частично

отсортированной подгруппе. Самый высокий игрок переходит на место помечен-

ного игрока, следующий по росту — на место самого высокого и т. д.

Когда следует остановить перемещение? Представьте, что вы и помеченный

игрок двигаетесь влево по шеренге. В каждой позиции игрок перемещается вправо,

но вы также сравниваете помеченного игрока с текущим. Процесс перемещения

останавливается тогда, когда будет перемещен последний игрок, рост которого превышает рост помеченного игрока. Последнее перемещение открывает свободное

место для помеченного игрока, который после вставки окажется в правильном по-

рядке частичной сортировки. Этот шаг показан на рис. 3.11, в.

Размер частично отсортированной подгруппы увеличивается на одного игрока,

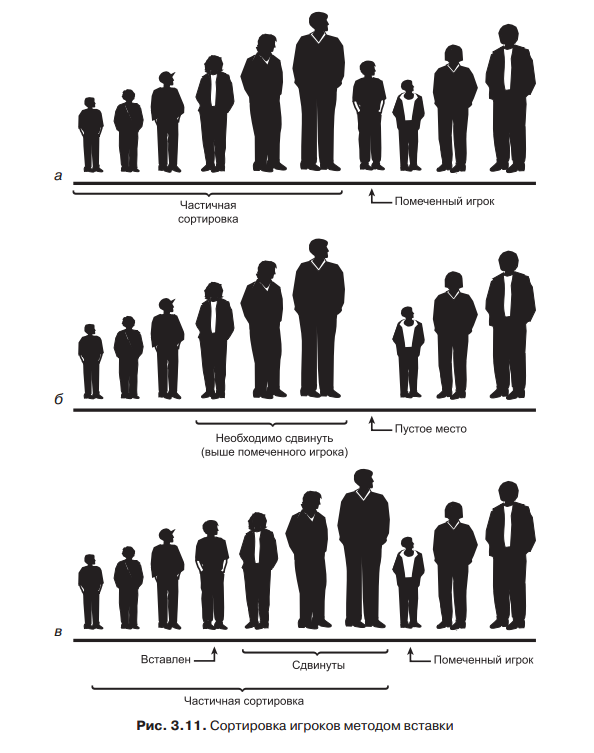
а неупорядоченная подгруппа становится на одного игрока меньше. Маркер (фут-

болка) перемещается на одну позицию вправо, и снова оказывается перед крайним

левым игроком в неупорядоченной группе. Процесс повторяется до тех пор, пока

все неупорядоченные игроки не будут вставлены (отсюда и название алгоритма)

в соответствующие позиции частично отсортированной подгруппы.



**Реализация сортировки методом вставки на языке Java**

Следующий метод из программы insertSort.java выполняет сортировку методом вставки:

**public void insertionSort()**

**{**

**int in, out;**

**for(out=1; out<nElems; out++) // out - разделительный маркер**

**{**

**long temp = a[out]; // Скопировать помеченный элемент**

**in = out; // Начать перемещения с out**

**while(in>0 && a[in-1] >= temp) // Пока не найден меньший элемент**

**{**

**a[in] = a[in-1]; // Сдвинуть элемент вправо**

**--in; // Перейти на одну позицию влево**

**}**

**a[in] = temp; // Вставить помеченный элемент**

**}**

**}**

Во внешнем цикле for счетчик начинает с позиции 1 и двигается вправо. Он

отмечает крайний левый неотсортированный элемент. Во внутреннем цикле while

счетчик in начинает с позиции out и двигается влево — либо пока temp не станет

меньше элемента массива, либо когда дальнейшее смещение станет невозможным.

При каждом проходе по циклу while следующий отсортированный элемент сдви-

гается на одну позицию вправо.

**Сложность сортировки методом вставки**

Сколько операций сравнения и копирования требует этот алгоритм? При первом

проходе сравнивается не более одного элемента, на втором — не более двух и так

далее, вплоть до N–1 сравнений на последнем проходе. В сумме получается:

1 + 2 + 3 + … + N–1 = N × (N–1)/2.

Тем не менее, поскольку при каждом проходе в среднем фактически сравнива-

ется только половина от максимального количества элементов (прежде чем будет

найдена точка вставки), сумму можно уменьшить вдвое; получается:

N × N–1)/4.

Количество операций копирования приблизительно совпадает с количеством

сравнений. Однако копирование занимает меньше времени, чем перестановка, так

что для случайных данных этот алгоритм работает вдвое быстрее пузырьковой

сортировки и быстрее сортировки методом выбора. Как бы то ни было, как и во

всех остальных методах сортировки этой главы, для случайных данных сортировка

методом вставки выполняется за время порядка O(N2

).

Для данных, уже прошедших предварительную сортировку, этот алгоритм

работает гораздо эффективнее. Если данные упорядочены, то условие цикла while

никогда не бывает истинным; следовательно, оно вырождается в простую команду

во внешним цикле, выполняемую N–1 раз. В этом случае алгоритм выполняется

за время O(N). Если данные почти отсортированы, то сортировка методом вставки

выполняется почти за время O(N), а следовательно, является простым и эффек-

тивным способом упорядочения файлов данных с небольшими отклонениями

в порядке сортировки.

Но если данные изначально отсортированы в обратном порядке, алгоритму

придется выполнить все возможные сравнения и перемещения, и тогда вставка

методом сортировки выполняется не быстрее пузырьковой сортировки.

**Сравнение простых алгоритмов сортировки**

Вероятно, пузырьковую сортировку лучше не использовать, разве что если под

рукой у вас не оказалось описаний алгоритмов. Пузырьковая сортировка настолько

проста, что ее вполне можно написать «по памяти». Но даже в этом случае она при-

менима только для относительно небольших объемов данных. (О том, что можно

считать «небольшим» объемом данных, рассказано в главе 15, «Рекомендации по

использованию».)

Сортировка методом выбора сводит к минимуму количество перестановок, но

количество сравнений все равно остается высоким. Такая сортировка может при-

годиться, если объем данных относительно невелик, а перестановка выполняется

намного медленнее сравнения.

Сортировка методом вставки — самый универсальный алгоритм среди тех опи-

санных. Он лучше всего подойдет в большинстве ситуаций, когда объем данных

невелик или данные почти отсортированы. Для больших объемов данных самым

эффективным обычно считается алгоритм быстрой сортировки; он будет рассмо-

трен в главе 7.

Мы сравнивали алгоритмы сортировки по скорости обработки. Другим важным

фактором для любого алгоритма являются затраты памяти. Все три алгоритма этой

главы выполняют сортировку «на месте», иначе говоря, они требуют минимальных

дополнительных затрат памяти за пределами исходного массива. Во всех видах

сортировки используется дополнительная переменная для временного хранения

элемента в ходе перестановки.

**Стеки**

В стеке доступен только один элемент данных: тот, который был в него вставлен

последним. Удалив этот элемент, пользователь получает доступ к предпоследнему

элементу и т. д. Такой механизм доступа удобен во многих ситуациях, связанных

с программированием. В этой главе будет показано, как использовать стек для

проверки сбалансированности круглых, фигурных и угловых скобок в исходном

файле компьютерной программы. А в последнем разделе этой главы стек сыграет

важнейшую роль в разборе (анализе) арифметических выражений вида 3 × (4 + 5).

Стек также удобен в алгоритмах, применяемых при работе с некоторыми слож-

ными структурами данных.

Многие микропроцессоры имеют стековую архитектуру. При вызове метода

адрес возврата и аргументы заносятся в стек, а при выходе они извлекаются из

стека. Операции со стеком встроены в микропроцессор.

Стековая архитектура также использовалась в некоторых старых калькулято-

рах. Вместо того чтобы вводить арифметическое выражение с круглыми скобками,

пользователь сохранял промежуточные результаты в стеке. Тема будет более под-

робно описана при обсуждении разбора арифметических выражений в последнем

разделе этой главы.

**Почтовая аналогия**

Для объяснения идеи стека лучше всего воспользоваться аналогией. Многие люди

складывают приходящие письма стопкой на журнальном столике. Когда появится

свободная минута, они обрабатывают накопившуюся почту сверху вниз. Сначала

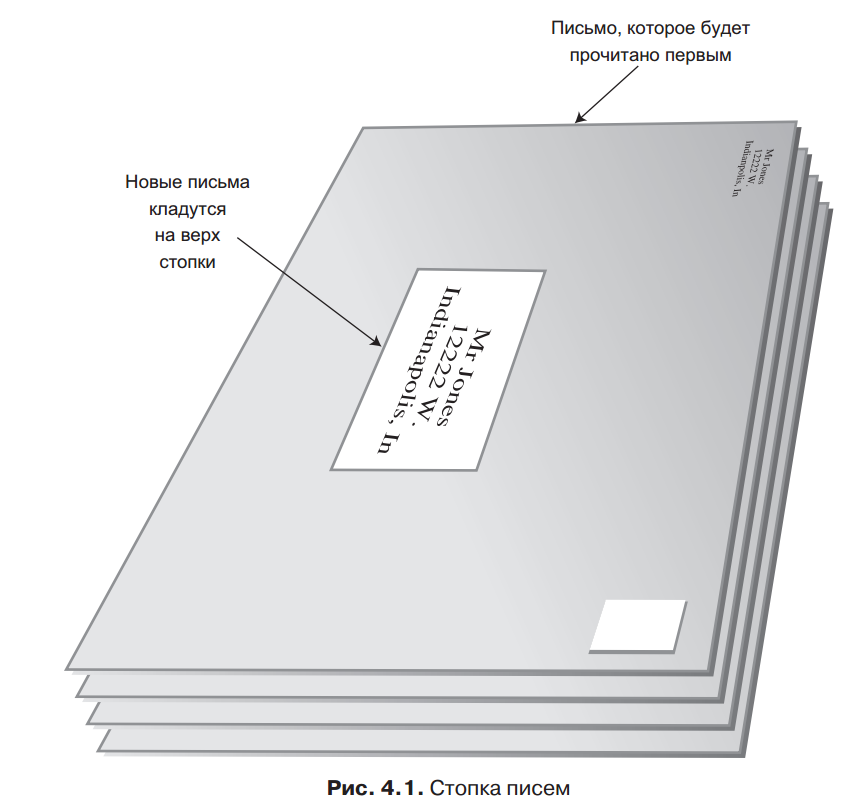
они открывают письмо, находящееся на вершине стопки, и выполняют необходимое

действие — оплачивают счет, выбрасывают письмо и т. д. Разобравшись с первым

письмом, они переходят к следующему конверту, который теперь оказывается на

верху стопки, и разбираются с ним. В конечном итоге они добираются до нижнего

письма (которое теперь оказывается верхним). Стопка писем изображена на рис. 4.1.



Принцип «начать с верхнего письма» отлично работает, при условии, что вся

почта может быть обработана за разумное время. В противном случае возникает

опасность того, что письма в нижней части стопки не будут просматриваться ме-

сяцами, а содержащиеся в них счета будут просрочены.

Конечно, не все люди разбирают почту по принципу «сверху вниз». Одни

предпочитают брать письма снизу стопки, чтобы старые письма обрабатывались

в первую очередь. Другие сортируют почту перед началом обработки и перекла-

дывают важную корреспонденцию наверх. В таких случаях стопка писем уже не

является аналогом стека из информатики. Если письма берутся снизу стопки, это

очередь, а если почта сортируется — приоритетная очередь. Обе возможности будут

описаны позднее.

Стековую архитектуру также можно сравнить с процессом выполнения раз-

личных дел во время рабочего дня. Вы трудитесь над долгосрочным проектом (A),

но ваш коллега просит временно прерваться и помочь ему с другим проектом (B).

В ходе работы над B к вам заходит бухгалтер, чтобы обсудить ваш отчет по ко-

мандировочным расходам (C). Во время обсуждения вам срочно звонят из от-

дела продаж, и вы тратите несколько минут на диагностику своего продукта (D).

Разобравшись со звонком D, вы продолжаете обсуждение C; после завершения C

возобновляется проект B, а после завершения B вы (наконец-то!) возвращаетесь

к проекту A. Проекты с более низкими приоритетами «складываются в стопку»,

ожидая, пока вы к ним вернетесь.

Основные операции со стеком — вставка (занесение) элемента в стек и извле­

чение из стека — выполняются только на вершине стека, то есть с его верхним эле-

ментом. Говорят, что стек работает по принципу LIFO (Last-In-First-Out), потому

что последний занесенный в стек элемент будет первым извлечен из него.

**Реализация стека на языке Java**

**class StackX**

**{**

**private int maxSize; // Размер массива**

**private long[] stackArray;**

**private int top; // Вершина стека**

**//--------------------------------------------------------------**

**public StackX(int s) // Конструктор**

**{**

**maxSize = s; // Определение размера стека**

**stackArray = new long[maxSize]; // Создание массива**

**top = -1; // Пока нет ни одного элемента**

**}**

**//--------------------------------------------------------------**

**public void push(long j) // Размещение элемента на вершине стека**

**{**

**stackArray[++top] = j; // Увеличение top, вставка элемента**

**}**

**//--------------------------------------------------------------**

**public long pop() // Извлечение элемента с вершины стека**

**{**

**return stackArray[top--]; // Извлечение элемента, уменьшение top**

**}**

**//--------------------------------------------------------------**

**public long peek() // Чтение элемента с вершины стека**

**{**

**return stackArray[top];**

**}**

**//--------------------------------------------------------------**

**public boolean isEmpty() // True, если стек пуст**

**{**

**return (top == -1);**

**}**

**//--------------------------------------------------------------**

**public boolean isFull() // True, если стек полон**

**{**

**return (top == maxSize-1);**

**}**

**//--------------------------------------------------------------**

**} // Конец класса StackX**

**////////////////////////////////////////////////////////////////**

**class StackApp**

**{**

**public static void main(String[] args)**

**{**

**StackX theStack = new StackX(10); // Создание нового стека**

**theStack.push(20); // Занесение элементов в стек**

**theStack.push(40);**

**theStack.push(60);**

**theStack.push(80);**

**while( !theStack.isEmpty() ) // Пока стек не станет пустым**

**{ // Удалить элемент из стека**

**long value = theStack.pop();**

**System.out.print(value); // Вывод содержимого**

**System.out.print(" ");**

**}**

**System.out.println("");**

**}**

**} // Конец класса StackApp**

**////////////////////////////////////////////////////////////////**

Метод main() класса StackApp создает стек для хранения 10 элементов, заносит

в него 4 элемента, а затем выводит все элементы, извлекая их из стека, пока он не

опустеет. Результат выполнения программы:

80 60 40 20

Обратите внимание на обратный порядок данных. Так как последний занесен-

ный элемент стал первым извлеченным элементом, число 80 стоит на первом месте

в выходных данных.

**Методы класса StackX**

Конструктор создает новый стек, размер которого передается в аргументе. В полях

класса хранится максимальный размер (размер массива), сам массив и переменная

top, в которой хранится индекс элемента, находящегося на вершине стека. (Об-

ратите внимание: необходимость передачи размера стека объясняется тем, что

стек реализован на базе массива. Если бы стек был реализован, допустим, на базе

связанного списка, то передавать размер было бы не обязательно.)

Метод push() увеличивает top, чтобы переменная указывала на ячейку, находя-

щуюся непосредственно над текущей ячейкой, и сохраняет в ней элемент данных.

Еще раз обратите внимание: top увеличивается до вставки элемента

Метод pop() возвращает значение, находящееся на вершине стека, после чего

уменьшает top. В результате элемент, находящийся на вершине стека, фактически

удаляется; он становится недоступным, хотя само значение остается в массиве (до

тех пор, пока в ячейку не будет занесен другой элемент).

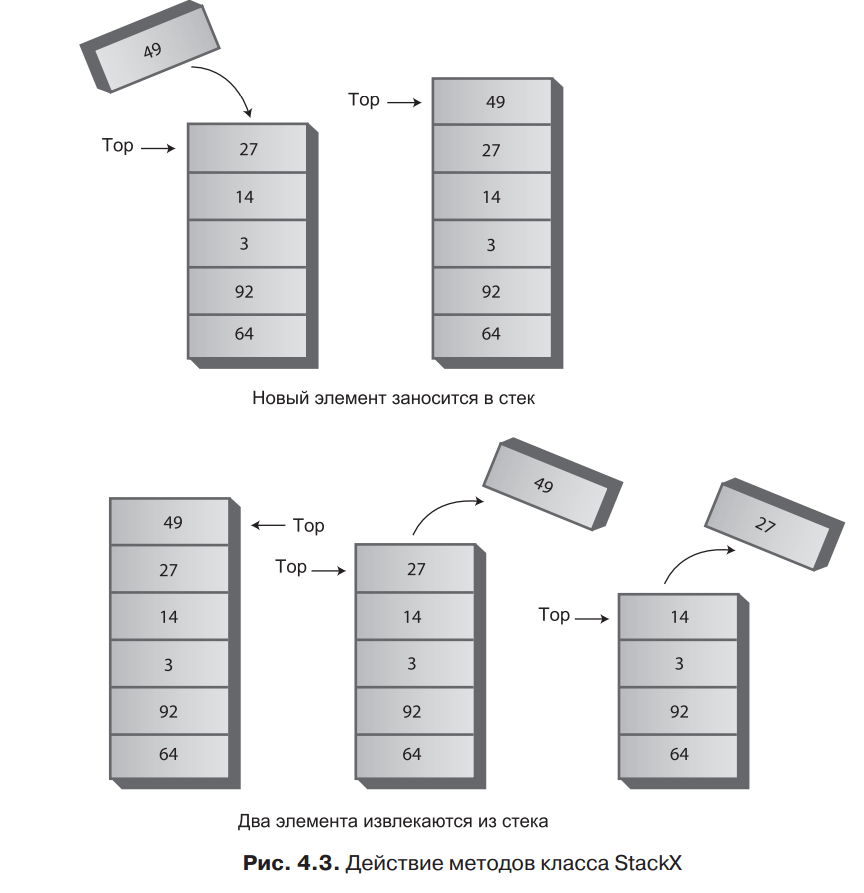
Метод peek() просто возвращает верхнее значение, не изменяя состояние стека.

Стеки 129

Методы isEmpty() и isFull() возвращают true, если стек пуст или полон соответ-

ственно. Для пустого стека переменная top содержит –1, а для полного — maxSize-1.

Рисунок 4.3 показывает, как работают методы класса стека.



**Обработка ошибок**

Единого подхода к обработке ошибок стека не существует. Например, что должно

происходить при занесении элемента в заполненный стек или при попытке извле-

чения элемента из пустого стека?

Ответственность за обработку таких ошибок возлагается на пользователя клас-

са. Прежде чем вставлять элемент, пользователь должен проверить, остались ли

в стеке свободные ячейки:

if( !theStack.isFull() )

insert(item);

else

System.out.print("Can't insert, stack is full");

Ради простоты кода мы исключили проверку из main() (к тому же в этой про-

стой программе мы знаем, что стек не заполнен, потому что только что его сами

инициализировали). Проверка пустого стека выполняется в методе main() при вы-

зове pop(). Многие классы стеков выполняют внутреннюю проверку таких ошибок

в методах push() и pop(). Такое решение считается предпочтительным. В Java класс

стека, обнаруживший ошибку, обычно инициирует исключение, которое может

быть перехвачено и обработано пользователем класса.

**Пример использования стека № 1.**

**Перестановка букв в слове**

Наш первый пример решает очень простую задачу: перестановку букв в слове.

Запустите программу, введите слово и нажмите Enter. Программа выводит слово,

в котором буквы переставлены в обратном порядке.

Для перестановки букв используется стек. Сначала символы последовательно

извлекаются из входной строки и заносятся в стек, а затем извлекаются из стека

и выводятся на экран. Так как стек работает по принципу LIFO, символы извле-

каются в порядке, обратном порядку их занесения. Код программы reverse.java

приведен в **листинге 4.2.**

class StackX

{

private int maxSize;

private char[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int max) // Конструктор

{

maxSize = max;

stackArray = new char[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(char j) // Размещение элемента на вершине стека

{

stackArray[++top] = j;

}

//--------------------------------------------------------------

public char pop() // Извлечение элемента с вершины стека

{

return stackArray[top--];

}

//--------------------------------------------------------------

public char peek() // Чтение элемента с вершины стека

{

return stackArray[top];

}

//--------------------------------------------------------------

public boolean isEmpty() // True, если стек пуст

{

return (top == -1);

}

//--------------------------------------------------------------

} // Конец класса StackX

////////////////////////////////////////////////////////////////

class Reverser

{

private String input; // Входная строка

private String output; // Выходная строка

//--------------------------------------------------------------

public Reverser(String in) // Конструктор

{ input = in; }

//--------------------------------------------------------------

public String doRev() // Перестановка символов

{

int stackSize = input.length(); // Определение размера стека

StackX theStack = new StackX(stackSize); // Создание стека

for(int j=0; j<input.length(); j++)

{

char ch = input.charAt(j); // Чтение символа из входного потока

theStack.push(ch); // Занесение в стек

}

output = "";

while( !theStack.isEmpty() )

{

char ch = theStack.pop(); // Извлечение символа из стека

output = output + ch; // Присоединение к выходной строке

}

return output;

}

//--------------------------------------------------------------

} // Конец класса Reverser

////////////////////////////////////////////////////////////////

class ReverseApp

{

public static void main(String[] args) throws IOException

{

String input, output;

while(true)

{

System.out.print("Enter a string: ");

System.out.flush();

input = getString(); // Чтение строки с клавиатуры

if( input.equals("") ) // Завершение, если [Enter]

break;

// Создание объекта Reverser

Reverser theReverser = new Reverser(input);

output = theReverser.doRev(); // Использование

System.out.println("Reversed: " + output);

}

}

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // Конец класса ReverseApp

////////////////////////////////////////////////////////////////

Класс Reverser переставляет символы своей входной строки. Его важнейший

компонент — метод doRev() — выбирает размер стека в соответствии с длиной

входной строки.

В методе main() мы получаем строку от пользователя, создаем объект Reverser

с передачей этой строки в аргументе конструктора, вызываем метод doRev() объекта

и выводим возвращаемое значение, в котором и содержится строка с переставлен-

ными символами. Пример взаимодействия с программой:

Enter a string: part

Reversed: trap

**Пример № 2. Поиск парных скобок**

Стеки также часто используются при разборе некоторых видов текстовых строк.

Как правило, строки содержат программный код, написанный на компьютерном

языке, и обрабатываются программой-компилятором.

Чтобы дать представление о процессе разбора, мы рассмотрим программу для

проверки парных скобок в строке, введенной пользователем. Строка не обяза-

на содержать настоящий Java-код (хотя это вполне возможно), но скобки в ней

должны использоваться по правилам Java. Нас интересуют фигурные скобки { },

квадратные [ ] и круглые скобки ( ). Каждая открывающая (левая) скобка должна

иметь парную закрывающую (правую) скобку, то есть для каждой скобки { долж-

на существовать парная скобка } и т. д. Кроме того, открывающие скобки ближе

к концу строки закрываются раньше тех, которые расположены ближе к началу.

Несколько примеров:

c[d] // Правильно

a{b[c]d}e // Правильно

a{b(c]d}e // Неправильно; ] не соответствует (

a[b{c}d]e} // Неправильно; у завершающей скобки } нет пары

a{b(c) // Неправильно; у открывающей скобки { нет пары

**Открывающие скобки в стеке**

Программа поиска парных скобок последовательно читает символы строки и за-

носит обнаруженные открывающие скобки в стек. Обнаружив во входных данных

закрывающую скобку, она извлекает верхний элемент из стека и проверяет его на

соответствие закрывающей скобке. Если они относятся к разным типам (скажем,

открывающая фигурная скобка с закрывающей круглой скобкой), происходит

ошибка. Кроме того, если в стеке нет открывающей скобки, парной по отношению

к закрывающей, или если для какой-то скобки в конечном итоге не нашлось пары,

происходит ошибка. Найти непарный ограничитель несложно — он остается в стеке

после того, как будут прочитаны все символы в строке.

Давайте посмотрим, как будет заполняться стек для типичной синтаксически

правильной строки:

a{b(c[d]e)f}

В таблице 4.1 показано, как выглядит стек при чтении очередного символа из

строки. Во втором столбце приводится содержимое стека (вершина справа).

В процессе чтения строки каждая открывающая скобка помещается в стек.

Каждая закрывающая скобка, прочитанная из входных данных, сопоставляет-

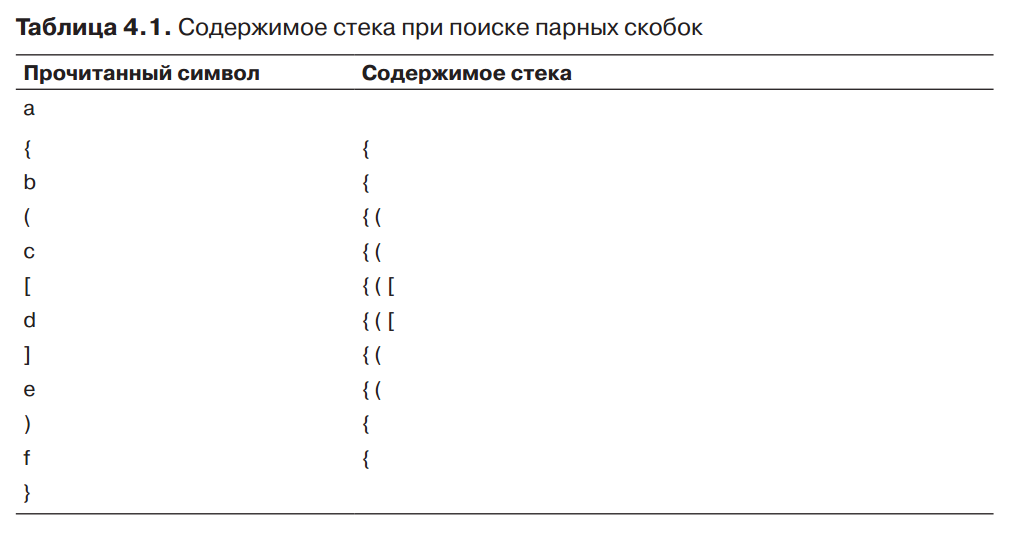
ся с открывающей скобкой, извлеченной с вершины стека. Если они образуют

пару — все хорошо. В стек заносятся только скобки; все остальные символы просто

игнорируются.

Такое решение работает, так как пара скобок, открытая последней, будет закрыта

первой — стоит напомнить, что стек работает по принципу LIFO.



**Реализация brackets.java на языке Java**

Программа разбора скобок brackets.java приведена в листинге 4.3. Поиск парных

скобок выполняется методом check() класса BracketChecker.

Листинг 4.3. Программа brackets.java

// brackets.java

// Использование стека для поиска парных скобок

// Запуск программы: C>java BracketsApp

import java.io.\*; // Для ввода/вывода

////////////////////////////////////////////////////////////////

class StackX

{

private int maxSize;

private char[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int s) // Конструктор

{

maxSize = s;

stackArray = new char[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(char j) // Размещение элемента на вершине стека

{

stackArray[++top] = j;

}

//--------------------------------------------------------------

public char pop() // Извлечение элемента с вершины стека

{

return stackArray[top--];

}

//--------------------------------------------------------------

public char peek() // Чтение элемента с вершины стека

{

return stackArray[top];

}

//--------------------------------------------------------------

public boolean isEmpty() // True, если стек пуст

{

return (top == -1);

}

//--------------------------------------------------------------

} // Конец класса StackX

////////////////////////////////////////////////////////////////

class BracketChecker

{

private String input; // Входная строка

//--------------------------------------------------------------

public BracketChecker(String in) // Конструктор

{ input = in; }

//--------------------------------------------------------------

public void check()

{

int stackSize = input.length(); // Определение размера стека

StackX theStack = new StackX(stackSize); // Создание стека

for(int j=0; j<input.length(); j++) // Последовательное чтение

{

char ch = input.charAt(j); // Чтение символа

switch(ch)

{

case ‘{‘: // Открывающие скобки

case '[':

case '(':

theStack.push(ch); // Занести в стек

break;

case ‘}’: // Закрывающие скобки

case ']':

case ')':

if( !theStack.isEmpty() ) // Если стек не пуст,

{

char chx = theStack.pop(); // Извлечь и проверить

if( (ch=='}’ && chx!=’{‘) ||

(ch==']' && chx!='[') ||

(ch==')' && chx!='(') )

System.out.println("Error: "+ch+" at "+j);

}

else // Преждевременная нехватка элементов

System.out.println("Error: "+ch+" at "+j);

break;

default: // Для других символов действия не выполняются

break;

}

}

// В этой точке обработаны все символы

if( !theStack.isEmpty() )

System.out.println("Error: missing right delimiter");

}//--------------------------------------------------------------

} // Конец класса BracketChecker

////////////////////////////////////////////////////////////////

class BracketsApp

{

public static void main(String[] args) throws IOException

{

String input;

while(true)

{

System.out.print(

"Enter string containing delimiters: ");

System.out.flush();

input = getString(); // Чтение строки с клавиатуры

if( input.equals("") ) // Завершение, если [Enter]

break;

// Создание объекта BracketChecker

BracketChecker theChecker = new BracketChecker(input);

theChecker.check(); // Проверка парных скобок

}

}

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // Конец класса BracketsApp

////////////////////////////////////////////////////////////////

Метод check() использует класс StackX из программы reverse.java (см. ли-

стинг 4.2). Обратите внимание, насколько легко повторно использовать этот класс:

весь необходимый код находится в одном месте. Это одно из преимуществ объек-

тно-ориентированного программирования.

Метод main() класса BracketsApp в цикле запрашивает у пользователя строку

текста, создает объект BracketChecker, передавая полученную строку в аргументе,

после чего вызывает метод check() объекта BracketChecker. Если в ходе разбора бу-

дут обнаружены ошибки, метод check() выводит информацию о них; в противном

случае синтаксис парных скобок правилен.

Метод check() по возможности сообщает номер символа, в котором была обна-

ружена ошибка (начиная с 0 для крайней левой позиции) и обнаруженный непра-

вильный символ. Например, для входной строки

a{b(c]d}e

метод check() выведет следующую информацию:

Error: ] at 5

**Стек как инструмент программиста**

Посмотрите, как удобно было использовать стек в программе brackets.java. Можно

было создать массив, который делал бы то же самое, но тогда пришлось бы отслеживать индекс последнего добавленного символа, а также хранить другую служебную информацию. Стек в нашей задаче более удобен на концептуальном уровне. Благодаря ограничению доступа к элементам и использованию методов push()/pop() стек делает программу более понятной и снижает риск ошибок. (Как вам скажет любой плотник, самый подходящий инструмент наиболее безопасен.)

**Эффективность стеков**

Занесение и извлечение элементов из стека, реализованного в классе StackX, вы-

полняется за время O(1). Иначе говоря, время выполнения операции не зависит от

количества элементов в стеке; следовательно, операция выполняется очень быстро,

не требуя ни сравнений, ни перемещений.

**----------------------------------------------------------------------------**

**Очереди**

Структура данных, называемая в информатике очередью, напоминает стек, но

в очереди первым извлекается элемент, вставленный первым (FIFO, First-In-FirstOut),

тогда как в стеке, как мы видели, первым извлекается элемент, вставленный

последним (LIFO). Она работает по тому же принципу, что и очередь в кино: чело-

век, первым вставшим в очередь, первым доберется до кассы и купит билет. Тот, кто

встанет в очередь последним, последним купит билет (или не купит, если билеты

будут распроданы). На рис. 4.4 показано, как работает очередь.

Очередь — такой же вспомогательный инструмент программиста, как и стек.

Кроме того, очереди используются для моделирования реальных ситуаций ожи-

дания: клиентов в банке, вылета самолетов или передачи данных по Интернету



В операционной системе вашего компьютера (и в сети) трудятся различные

очереди, незаметно выполняющие свои обязанности. В очереди печати задания

ждут освобождения принтера. Данные, вводимые с клавиатуры, тоже сохраняют-

ся в очереди. Если вы работаете в текстовом редакторе, а компьютер на короткое

время отвлекся на выполнение другой операции, нажатия клавиш не будут по-

теряны; они ожидают в очереди, пока у редактора не появится свободное время

для их получения. Очередь обеспечивает хранение нажатий клавиш в исходной

последовательности до момента обработки.

**Реализация очереди на языке Java**

Программа queue.java реализует очередь в виде класса Queue. Класс содержит ти-

пичные методы insert(), remove(), peek(), isFull(), isEmpty() и size().

Программа main() создает очередь из пяти ячеек, вставляет в нее четыре эле-

мента, затем удаляет три элемента и вставляет еще четыре. При шестой вставке

происходит циклический перенос. Затем программа извлекает и выводит значения

всех элементов. Результат выглядит так:

40 50 60 70 80

Код программы queue.java приведен в листинге 4.4.

**Листинг 4.4.** Программа queue.java

class Queue

{

private int maxSize;

private long[] queArray;

private int front;

private int rear;

private int nItems;

//--------------------------------------------------------------

public Queue(int s) // Конструктор

{

maxSize = s;

queArray = new long[maxSize];

front = 0;

rear = -1;

nItems = 0;

}

//--------------------------------------------------------------

public void insert(long j) // Вставка элемента в конец очереди

{

if(rear == maxSize-1) // Циклический перенос

rear = -1;

queArray[++rear] = j; // Увеличение rear и вставка

nItems++; // Увеличение количества элементов

}

//--------------------------------------------------------------

public long remove() // Извлечение элемента в начале очереди

{

long temp = queArray[front++]; // Выборка и увеличение front

if(front == maxSize) // Циклический перенос

front = 0;

nItems--; // Уменьшение количества элементов

return temp;

}

//--------------------------------------------------------------

public long peekFront() // Чтение элемента в начале очереди

{

return queArray[front];

}

//--------------------------------------------------------------

public boolean isEmpty() // true, если очередь пуста

{

return (nItems==0);

}

//--------------------------------------------------------------

public boolean isFull() // true, если очередь заполнена

return (nItems==maxSize);

}

//--------------------------------------------------------------

public int size() // Количество элементов в очереди

{

return nItems;

}

//--------------------------------------------------------------

} // Конец класса Queue

////////////////////////////////////////////////////////////////

class QueueApp

{

public static void main(String[] args)

{

Queue theQueue = new Queue(5); // Очередь из 5 ячеек

theQueue.insert(10); // Вставка 4 элементов

theQueue.insert(20);

theQueue.insert(30);

theQueue.insert(40);

theQueue.remove(); // Извлечение 3 элементов

theQueue.remove(); // (10, 20, 30)

theQueue.remove();

theQueue.insert(50); // Вставка еще 4 элементов

theQueue.insert(60); // (с циклическим переносом)

theQueue.insert(70);

theQueue.insert(80);

while( !theQueue.isEmpty() ) // Извлечение и вывод

{ // всех элементов

long n = theQueue.remove(); // (40, 50, 60, 70, 80)

System.out.print(n);

System.out.print(" ");

}

System.out.println("");

}

} // Конец класса QueueApp

В нашей реализации в полях класса Queue хранятся не только индексы начала

и конца очереди, но и текущее количество элементов nItems. Некоторые реализа-

ции очереди не используют это поле; альтернативное решение будет представлено

позднее.

**Метод insert()**

Метод insert() предполагает, что очередь еще не заполнена. В методе main() со-

ответствующей проверки нет, но обычно insert() вызывается только после пред-

варительного вызова isFull() и проверки возвращаемого значения. (Хотя еще

правильнее включить проверку в insert() и инициировать исключение при попытке

вставки в заполненную очередь.)

В общем случае при вставке сначала увеличивается индекс rear, а затем новый

элемент вставляется в ячейку, на которую ссылается его новое значение. Но если

значение rear уже достигло вершины массива (maxSize-1), то перед выполнением

вставки оно должно вернуться к нижней границе. Для этого полю rear присваи-

вается значение –1, чтобы при увеличении оно стало равным 0 (нижняя граница

массива). Код метода завершается увеличением количества элементов nItems.

**Метод remove()**

Метод remove() предполагает, что очередь не пуста. Вызовите isEmpty() перед remove(),

чтобы убедиться в истинности этого условия, или встройте проверку ошибок

в remove().

Извлечение всегда начинается с получения значения в ячейке front и уве-

личения front. Если значение front при этом выходит за границу массива, оно

возвращается к 0. На время проверки этой возможности возвращаемое значение

сохраняется во временной переменной. Код метода завершается уменьшением

количества элементов nItems.

**Метод peek()**

Метод peek() очень прост: он возвращает значение из ячейки front. Некоторые ре-

ализации также позволяют читать значение из нижней границы массива; в таких

случаях методам обычно присваиваются имена peekFront() и peekRear() (или просто

front() и rear()).

**Методы isEmpty(), isFull() и size()**

Возвращаемое значение методов isEmpty(), isFull() и size() зависит от поля nItems.

Первые два метода сравнивают его с 0 и maxSize, а третий возвращает текущее

значение.

**Реализация без счетчика элементов**

Включение поля nItems в класс Queue слегка снижает эффективность методов

insert() и remove(), которым приходится соответственно увеличивать и уменьшать

эту переменную. На первый взгляд потери невелики, но при очень большом коли-

честве операций вставки и удаления они могут повлиять на производительность.

По этой причине некоторые реализации очередей обходятся без счетчика эле-

ментов, а для проверки заполненности/отсутствия элементов и количества элемен-

тов в очереди используются значения полей front и rear. Однако методы isEmpty(),

isFull() и size() становятся неожиданно сложными, потому что последователь-

ность элементов, как мы уже видели, может быть как непрерывной, так и несвязной.

Также может возникнуть одна странная проблема: при заполненной очереди

указатели front и rear могут занимать точно такие же позиции, как и при пустой

очереди. Очередь выглядит заполненной и пустой одновременно. Для решения этой

проблемы создается массив с количеством ячеек, на единицу большим максималь-

ного количества элементов, которые в нем будут размещаться. Пример реализации

класса очереди без счетчика элементов представлен в листинге 4.5.

**Листинг 4.5.** Реализация очереди без счетчика элементов

class Queue

{

private int maxSize;

private long[] queArray;

private int front;

private int rear;

//--------------------------------------------------------------

public Queue(int s) // Конструктор

{

maxSize = s+1; // Массив на одну ячейку больше

queArray = new long[maxSize]; // требуемого размера

front = 0;

rear = -1;

}

//--------------------------------------------------------------

public void insert(long j) // Вставка элемента в конец очереди

{

if(rear == maxSize-1)

rear = -1;

queArray[++rear] = j;

}

//--------------------------------------------------------------

public long remove() // Извлечение элемента в начале очереди

{

long temp = queArray[front++];

if(front == maxSize)

front = 0;

return temp;

}

//--------------------------------------------------------------

public long peek() // Чтение элемента в начале очереди

{

return queArray[front];

}

//--------------------------------------------------------------

public boolean isEmpty() // true, если очередь пуста

{

return ( rear+1==front || (front+maxSize-1==rear) );

}

//--------------------------------------------------------------

public boolean isFull() // true, если очередь заполнена

return ( rear+2==front || (front+maxSize-2==rear) );

}

//--------------------------------------------------------------

public int size() // (assumes queue not empty)

{

if(rear >= front) // Непрерывная последовательность

return rear-front+1;

else // Несвязная последовательность

return (maxSize-front) + (rear+1);

}

//--------------------------------------------------------------

} // Конец класса Queue

Обратите внимание на усложнение методов isFull(), isEmpty() и size(). На прак-

тике реальная необходимость в таком решении встречается очень редко, поэтому

мы не будем подробно рассматривать его.

**Эффективность очередей**

Вставка и извлечение элементов очереди, как и элементов стека, выполняются за

время O(1).

**Дек**

Дек (deque) представляет собой двустороннюю очередь. И вставка, и удаление

элементов могут производиться с обоих концов. Соответствующие методы могут

называться insertLeft()/insertRight() и removeLeft()/removeRight().

Если ограничиться только методами insertLeft() и removeLeft() (или их эквива-

лентами для правого конца), дек работает как стек. Если же ограничиться методами

insertLeft() и removeRight() (или противоположной парой), он работает как очередь.

По своей гибкости деки превосходят и стеки, и очереди; иногда они использу-

ются в библиотеках классов-контейнеров для реализации обеих разновидностей.

Тем не менее используются они реже стеков или очередей, поэтому подробно рас-

сматривать мы их тоже не будем.

**Приоритетные очереди**

Приоритетная очередь является более специализированной структурой данных,

чем стек или очередь, однако и он неожиданно часто оказывается полезным.

У приоритетной очереди, как и у обычной, имеется начало и конец, а элементы из-

влекаются от начала. Но у приоритетной очереди элементы упорядочиваются по

ключу, так что элемент с наименьшим (в некоторых реализациях — наибольшим)

значением ключа всегда находится в начале. Новые элементы вставляются в по-

зициях, сохраняющих порядок сортировки.

Давайте применим нашу аналогию с сортировкой почты к приоритетным очере-

дям. Каждый раз, когда почтальон отдает вам письмо, вы вкладываете его в стопку

непрочитанных писем в соответствии с приоритетом. Если письмо требует не-

медленного ответа (телефонная компания собирается отключить ваш модем), оно

кладется наверх, а письма, на которые можно ответить в свободное время (письмо

от любимой тетушки), подкладываются под низ стопки. Письма со средним при-

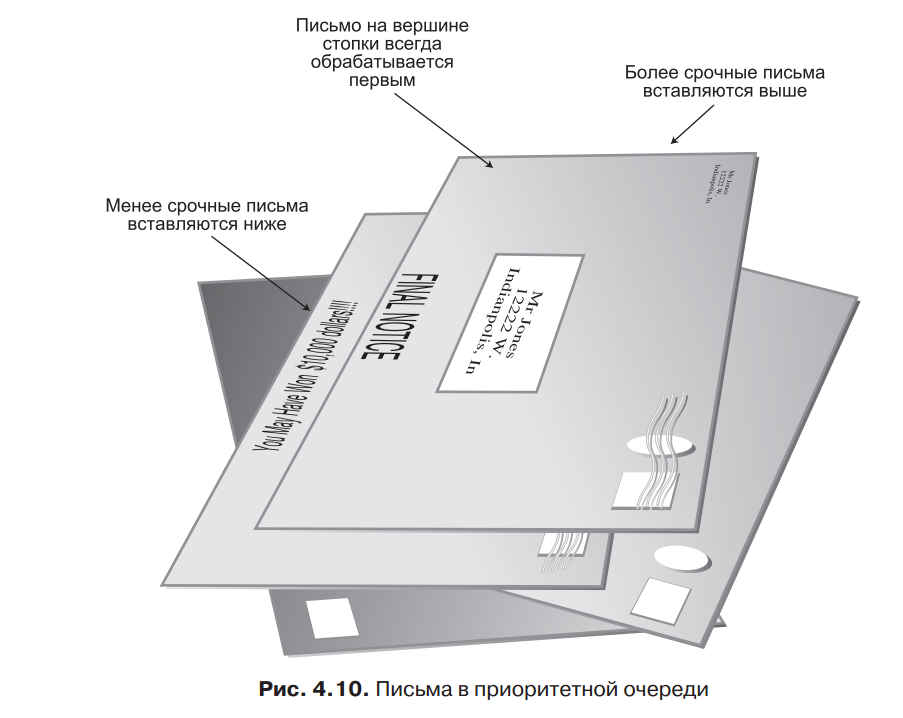
оритетом размещаются где-то в середине; чем выше приоритет, тем выше оказы-

вается письмо в стопке. Верх стопки соответствует началу приоритетной очереди.

Когда у вас появляется время на чтение, вы берете письмо с верха стопки (из

начала очереди); таким образом, самые важные письма будут обработаны первыми.

Ситуация изображена на рис. 4.10.



Приоритетные очереди, как и стеки с очередями, часто используются програм-

мистами как вспомогательные инструменты. Пример такого рода встретится нам

при построении связующего дерева графа в главе 14, «Взвешенные графы».

Кроме того, приоритетные очереди (как и обычные) часто используются в ком-

пьютерных системах. Скажем, в операционной системе с вытесняющей многоза-

дачностью программы могут размещаться в приоритетной очереди, чтобы высоко-

приоритетная программа первой получила процессорное время для ее выполнения.

При работе с приоритетными очередями довольно часто требуется получить

доступ к элементу с наименьшим значением ключа (которое может представлять

самый экономичный или быстрый способ выполнения какой-либо операции) — то

есть самым приоритетным является элемент с наименьшим ключом. Наше обсуж-

дение будет основано именно на таком предположении, хотя во многих ситуациях

самым приоритетным является элемент с наибольшим ключом.

Кроме ускоренного доступа к элементу с наименьшим ключом, приоритетная

очередь также должна обеспечивать относительно быструю вставку. По этой при-

чине приоритетные очереди, как упоминалось ранее, часто реализуются на основе

структуры данных, называемой кучей (heap), — эта структура рассматривается

в главе 12. А в этой главе будет представлена реализация приоритетной очереди

на базе простого массива. Ее недостатком является медленная вставка; с другой

стороны, она проще, а ее применение более уместно, если количество элементов

невелико или скорость вставки не критична.

**Реализация приоритетной очереди на языке Java**

Реализация приоритетной очереди на базе простого массива на языке Java пред-

ставлена в листинге 4.6.

Листинг 4.6. Программа priorityQ.java

class PriorityQ

{

// Элементы массива сортируются по значению ключа,

// от максимумa (0) до минимума (maxSize-1)

private int maxSize;

private long[] queArray;

private int nItems;

//-------------------------------------------------------------

public PriorityQ(int s) // Конструктор

{

maxSize = s;

queArray = new long[maxSize];

nItems = 0;

}

//-------------------------------------------------------------

public void insert(long item) // Вставка элемента

{

int j;

if(nItems==0) // Если очередь пуста,

queArray[nItems++] = item; // вставляем в ячейку 0

else // Если очередь содержит элементы

{

for(j=nItems-1; j>=0; j--) // Перебор в обратном направлении

{

if( item > queArray[j] ) // Если новый элемент больше,

queArray[j+1] = queArray[j]; // сдвинуть вверх

else // Если меньше,

break; // сдвиг прекращается

}

queArray[j+1] = item; // Вставка элемента

nItems++;

}

} //

//-------------------------------------------------------------

public long remove() // Извлечение минимального элемента

{ return queArray[--nItems]; }

//-------------------------------------------------------------

public long peekMin() // Чтение минимального элемента

{ return queArray[nItems-1]; }

//-------------------------------------------------------------

public boolean isEmpty() // true, если очередь пуста

{ return (nItems==0); }

//-------------------------------------------------------------

public boolean isFull() // true, если очередь заполнена

{ return (nItems == maxSize); }

//-------------------------------------------------------------

} // Конец класса PriorityQ

////////////////////////////////////////////////////////////////

class PriorityQApp

{

public static void main(String[] args) throws IOException

{

PriorityQ thePQ = new PriorityQ(5);

thePQ.insert(30);

thePQ.insert(50);

thePQ.insert(10);

thePQ.insert(40);

thePQ.insert(20);

while( !thePQ.isEmpty() )

{

long item = thePQ.remove();

System.out.print(item + " "); // 10, 20, 30, 40, 50

}

System.out.println("");

}

//-------------------------------------------------------------

} // Конец класса PriorityQApp

В методе main() мы сначала вставляем в очередь пять элементов в произвольном

порядке, а затем извлекаем и выводим их. Первым всегда извлекается метод с наи-

меньшим ключом, поэтому результат выглядит так:

10, 20, 30, 40, 50

Метод insert() сначала проверяет, содержит ли очередь хотя бы один элемент;

если элементы отсутствуют, элемент вставляется в ячейку с индексом 0. В про-

тивном случае он начинает с вершины массива сдвигать существующие элементы

вверх, пока не найдет место, подходящее для вставки нового элемента. Затем эле-

мент вставляется, а значение nItems увеличивается. Если существует вероятность

того, что приоритетная очередь заполнена, следует проверить эту возможность

методом isFull() перед вызовом insert().

Поля front и rear в этой реализации не нужны (в отличие от обычной очереди),

потому что, как уже говорилось ранее, значение front всегда равно nItems-1, а зна-

чение rear всегда равно 0.

Метод remove() предельно прост: он уменьшает nItems и возвращает элемент,

находящийся на вершине массива. Метод peekMin() устроен аналогично, если не

считать того, что он не уменьшает nItems. Методы isEmpty() и isFull() проверяют,

содержит ли поле nItems значение 0 или maxSize соответственно.

**Эффективность приоритетных очередей**

В рассмотренной реализации приоритетной очереди вставка выполняется за время

O(N), а извлечение — за время O(1). В главе 12 будет показано, как улучшить время

вставки за счет реализации на базе кучи.

**Разбор арифметических выражений**

Итак, в этой главе были описаны три разные структуры данных. Давайте немного

сменим тему и рассмотрим важное практическое применение одной из этих структур.

Приложение разбирает (то есть анализирует) арифметические выражения вида

2 + 3, или 2 × (3 + 4), или ((2 + 4) × 7) + 3 × (9 – 5). Для хранения компонентов

выражения используется стек. В программе brackets.java (см. листинг 4.3) был

представлен пример использования стека для проверки парности скобок. Анало-

гичным способом (хотя и более сложным) стеки могут применяться и для разбора

арифметических выражений.

Материал данного раздела в каком-то смысле можно рассматривать как «до-

полнительное чтение». Он не обязателен для понимания книги, а в вашей повсе-

дневной работе вам вряд ли придется часто писать код разбора арифметических

выражений (если, конечно, вы не занимаетесь разработкой компиляторов). Кроме

того, реализация программы намного сложнее всего, что рассматривалось нами

ранее. И все же это важное практическое применение стеков чрезвычайно поучи-

тельно, а многие его аспекты представляют интерес сами по себе.

Как оказалось, вычислить результат арифметического выражения довольно

трудно (во всяком случае для компьютерного алгоритма). Проще всего разделить

этот процесс на два шага:

1. Преобразование арифметического выражения в другой формат, называемый

постфиксной записью.

2. Вычисление результата по постфиксной записи.

Шаг 1 относительно сложен, зато шаг 2 выполняется просто. В любом случае

двухшаговая схема обеспечивает более простой алгоритм, чем попытка прямого

разбора арифметических выражений. Конечно, человеку проще работать с обычны-

ми арифметическими выражениями. Вскоре мы вернемся к различиям в подходах

человека и компьютера к решению этой задачи.

Прежде чем переходить к рассмотрению всех подробностей шагов 1 и 2, необ-

ходимо познакомиться с постфиксной записью.

**Постфиксная запись**

В записи традиционных арифметических выражений оператор (+, –, \* или /) раз-

мещается между двумя операндами (числа или обозначающие их символические

имена). Запись, в которой оператор записывается между операндами, называется

инфиксной. Мы пишем 2 + 2, или 4 × 7, или с обозначением чисел буквами — A + B

и A/B.

В постфиксной записи, также называемой обратной польской записью, или RPN

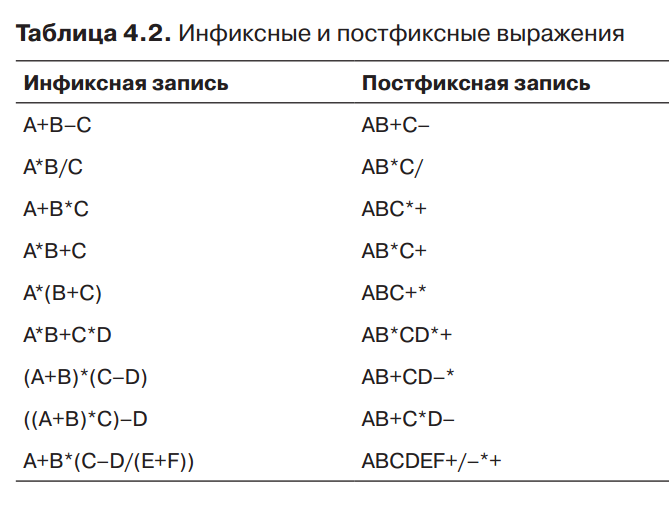
(Reverse Polish Notation), потому что она была изобретена польским математиком,

оператор записывается после двух операндов. Таким образом, A + B превращается

в AB+, а A/B превращается в AB/. Более сложные инфиксные выражения тоже

преобразуются в постфиксную запись; примеры приведены в табл. 4.2. Вскоре вы

узнаете, как строятся постфиксные выражения.



В некоторых языках программирования также существует оператор возведения

в степень (чаще всего символ ^), но в нашем обсуждении эта возможность не рас-

сматривается.

Кроме инфиксной и постфиксной, также существует префиксная запись, при

которой оператор записывается перед операндами: +AB вместо AB+. Эта запись

функционально эквивалентна постфиксной, но на практике она применяется редко.

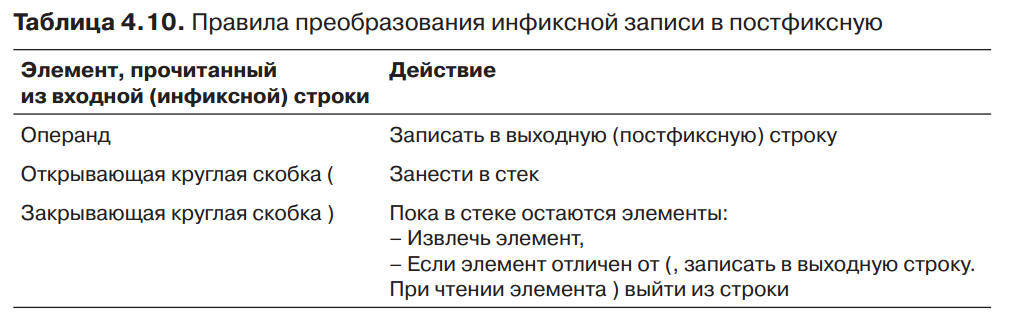
**Правила преобразования**

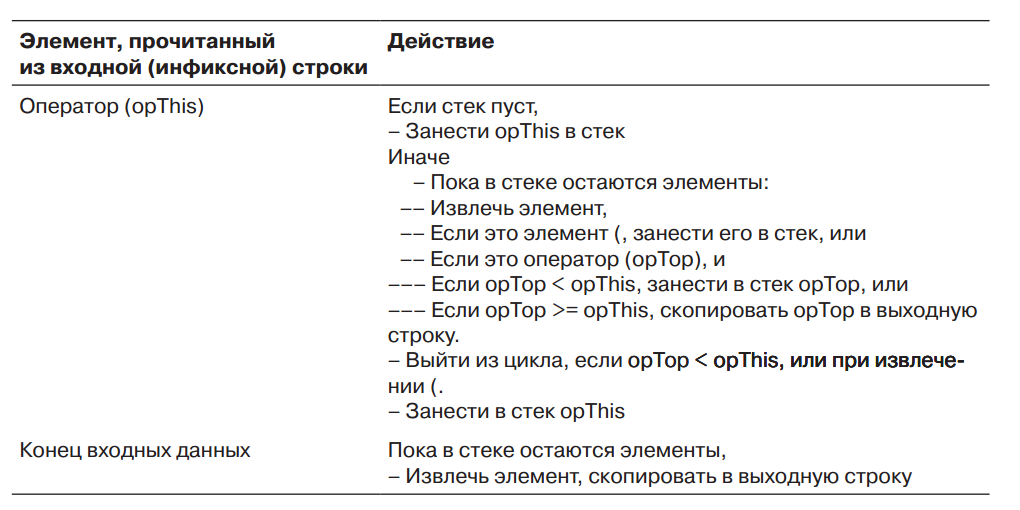
Давайте сформулируем более конкретные правила преобразования инфиксной

записи в постфиксную. С элементами, последовательно читаемыми из инфикс-

ной входной строки, выполняются действия из табл. 4.10. Действия описаны на

псевдокоде.





В этой таблице обозначения < и >= относятся к приоритетам операторов, а не

к числовым значениям. Оператор opThis только что был прочитан из инфиксной

входной строки, а оператор opTop только что был извлечен из стека.

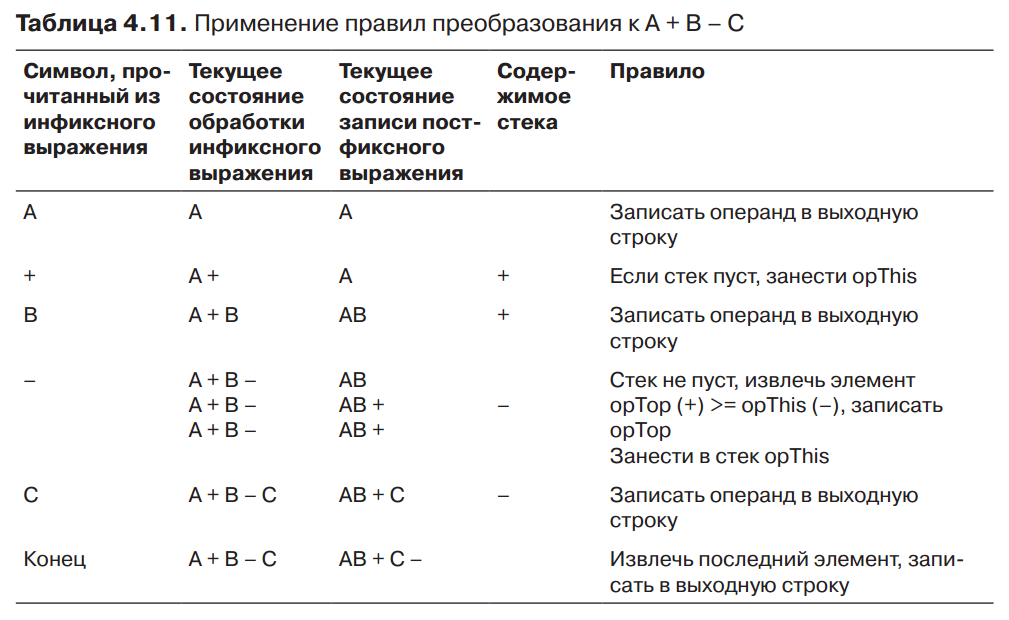
Убедитесь в том, что этот набор правил действительно работает. В табл. 4.11–

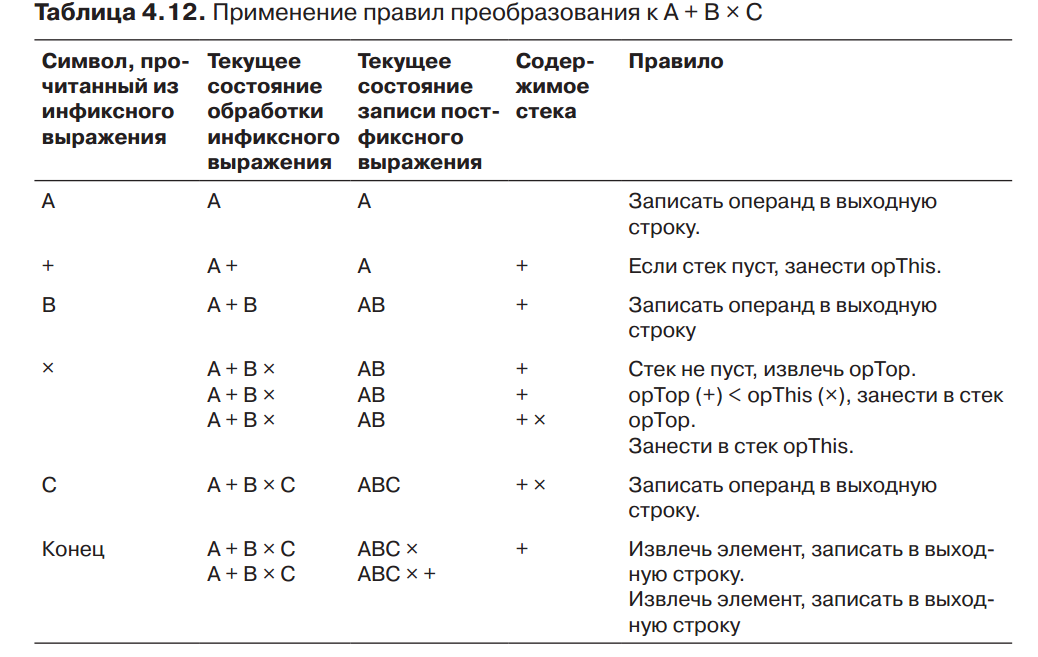
4.13 показано, как эти правила применяются к трем инфиксным выражениям. Эти

таблицы аналогичны табл. 4.6–4.8, не считая того, что для каждого шага добавлены

соответствующие правила. Попробуйте построить такие же таблицы для других

инфиксных выражений.







**Преобразование инфиксной записи**

**в постфиксную на языке Java**

В листинге 4.7 приведена программа infix.java, использующая правила из табл. 4.10

для преобразования инфиксного выражения в постфиксное.

**Листинг 4.7.** Программа infix.java

// infix.java

// Преобразование инфиксных арифметических выражений в постфиксные

// Запуск программы: C>java InfixApp

import java.io.\*; // Для ввода/вывода

////////////////////////////////////////////////////////////////

class StackX

{

private int maxSize;

private char[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int s) // Конструктор

{

maxSize = s;

stackArray = new char[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(char j) // Размещение элемента на вершине стека

{ stackArray[++top] = j; }

//--------------------------------------------------------------

public char pop() // Извлечение элемента с вершины стека

{ return stackArray[top--]; }

//--------------------------------------------------------------

public char peek() // Чтение элемента с вершины стека

{ return stackArray[top]; }

//--------------------------------------------------------------

public boolean isEmpty() // true, если стек пуст

{ return (top == -1); }

//-------------------------------------------------------------

public int size() // Текущий размер стека

{ return top+1; }

//--------------------------------------------------------------

public char peekN(int n) // Чтение элемента с индексом n

{ return stackArray[n]; }

//--------------------------------------------------------------

public void displayStack(String s)

{

System.out.print(s);

System.out.print("Stack (bottom-->top): ");

for(int j=0; j<size(); j++)

{

System.out.print( peekN(j) );

System.out.print(' ');

}

System.out.println("");

}

//--------------------------------------------------------------

} // Конец класса StackX

////////////////////////////////////////////////////////////////

class InToPost // Преобразование инфиксной записи в постфиксную

{

private StackX theStack;

private String input;

private String output = "";

//--------------------------------------------------------------

public InToPost(String in) // Конструктор

{

input = in;

int stackSize = input.length();

theStack = new StackX(stackSize);

}

//--------------------------------------------------------------

public String doTrans() // Преобразование в постфиксную форму

{

for(int j=0; j<input.length(); j++)

{

char ch = input.charAt(j);

theStack.displayStack("For "+ch+" "); // \*диагностика\*

switch(ch)

{

case '+': // + или -

case ‘-’:

gotOper(ch, 1); // Извлечение операторов

break; // (приоритет 1)

case ‘\*’: // \* или /

case ‘/’:

gotOper(ch, 2); // Извлечение операторов

break; // (приоритет 2)

case ‘(‘: // Открывающая круглая скобка

theStack.push(ch); // Занести в стек

break;

case ‘)’: // Закрывающая круглая скобка

gotParen(ch); // Извлечение операторов

break;

default: // Остается операнд

output = output + ch; // Записать в выходную строку

break;

}

}

while( !theStack.isEmpty() ) // Извлечение оставшихся операторов

{

theStack.displayStack("While "); // \*диагностика\*

output = output + theStack.pop(); // write to output

}

theStack.displayStack("End "); // \*диагностика\*

return output; // Возвращение постфиксного выражения

}

//--------------------------------------------------------------

public void gotOper(char opThis, int prec1)

{ // Чтение оператора из входной строки

while( !theStack.isEmpty() )

{

char opTop = theStack.pop();

if( opTop == '(' ) // Если это '('

{

theStack.push(opTop); // Вернуть '('

break;

}

else // Оператор

{

int prec2; // Приоритет нового оператора

if(opTop==’+’ || opTop==’-’) // Определение приоритета

prec2 = 1;

else

prec2 = 2;

if(prec2 < prec1) // Если приоритет нового оператора

{ // меньше приоритета старого

theStack.push(opTop); // Сохранить новый оператор

break;

}

else // Приоритет нового оператора

output = output + opTop; // не меньше приоритета старого

}

}

theStack.push(opThis); // Занесение в стек нового оператора

}

//--------------------------------------------------------------

public void gotParen(char ch)

{ // Прочитана закрывающая скобка

while( !theStack.isEmpty() )

{

char chx = theStack.pop();

if( chx == ‘(‘ ) // Если извлечен элемент ‘(‘

break; // Прервать выполнение

else // Если извлечен оператор

output = output + chx; // Вывести в постфиксную строку

}

}

//--------------------------------------------------------------

} // Конец класса InToPost

////////////////////////////////////////////////////////////////

class InfixApp

{

public static void main(String[] args) throws IOException

{

String input, output;

while(true)

{

System.out.print("Enter infix: ");

System.out.flush();

input = getString(); // Чтение строки с клавиатуры

if( input.equals("") ) // Выход, если нажата клавиша [Enter]

break;

// Создание объекта-преобразователя

InToPost theTrans = new InToPost(input);

output = theTrans.doTrans(); // Преобразование

System.out.println("Postfix is " + output + '\n');

}

}

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // Конец класса InfixApp

////////////////////////////////////////////////////////////////

Метод main() класса InfixApp запрашивает у пользователя инфиксное выраже-

ние. Для ввода данных используется вспомогательный метод readString(). Про-

грамма создает объект InToPost, инициализируемый входной строкой, после чего

вызывает метод doTrans() для выполнения преобразования. Метод возвращает

постфиксную выходную строку, которая выводится на экран.

Правила преобразования из табл. 4.10 реализуются конструкцией switch метода

doTrans(). При чтении оператора вызывается метод gotOper(), а при чтении за-

крывающей круглой скобки ) — метод gotParen(). Эти методы реализуют третье и

четвертое правило из таблицы, более сложные по сравнению с другими правилами.

В класс StackX включен метод displayStack() для вывода всего содержимого

стека. Теоретически это неправильно; предполагается, что при работе со стеком

доступен только элемент, находящийся на вершине. Однако этот метод является

удобным диагностическим средством для просмотра содержимого стека на каждой

стадии преобразования. Пример выполнения программы infix.java:

Enter infix: A\*(B+C)-D/(E+F)

For A Stack (bottom-->top):

For \* Stack (bottom-->top):

For ( Stack (bottom-->top): \*

For B Stack (bottom-->top): \* (

For + Stack (bottom-->top): \* (

For C Stack (bottom-->top): \* ( +

For ) Stack (bottom-->top): \* ( +

For - Stack (bottom-->top): \*

For D Stack (bottom-->top): -

For / Stack (bottom-->top): -

For ( Stack (bottom-->top): - /

For E Stack (bottom-->top): - / (

For + Stack (bottom-->top): - / (

For F Stack (bottom-->top): - / ( +

For ) Stack (bottom-->top): - / ( +

While Stack (bottom-->top): - /

While Stack (bottom-->top): -

End Stack (bottom-->top):

Postfix is ABC+\*DEF+/-

Выходные данные показывают, где вызывается метод displayStack() (из цикла

for, из цикла while или в конце программы), а для цикла for — какой символ был

только что прочитан из входной строки.

Вместо символических имен типа A или B можно использовать цифры (напри-

мер, 3 или 7) — программа тоже интерпретирует их как символы. Пример:

Enter infix: 2+3\*4

For 2 Stack (bottom-->top):

For + Stack (bottom-->top):

For 3 Stack (bottom-->top): +

For \* Stack (bottom-->top): +

For 4 Stack (bottom-->top): + \*

While Stack (bottom-->top): + \*

While Stack (bottom-->top): +

End Stack (bottom-->top):

Postfix is 234\*+

Разумеется, в постфиксной записи 234 означает три разных операнда 2, 3 и 4.

Программа infix.java не проверяет входные данные на наличие ошибок. Если

ввести некорректное инфиксное выражение, программа выдаст некорректный

результат, или попросту «упадет». Поэкспериментируйте с программой. Начните

с простых инфиксных выражений, попробуйте предсказать вид постфиксного вы-

ражения, затем запустите программу и проверьте ответ. Довольно скоро вы станете

экспертом в области постфиксных выражений.

**Вычисление результата постфиксного выражения**

Как видите, преобразование инфиксного выражения в постфиксное — не самая

тривиальная задача. Так ли уж необходимы все эти хлопоты? Да, преимущества

новой записи проявляются при вычислении результата постфиксных выражений.

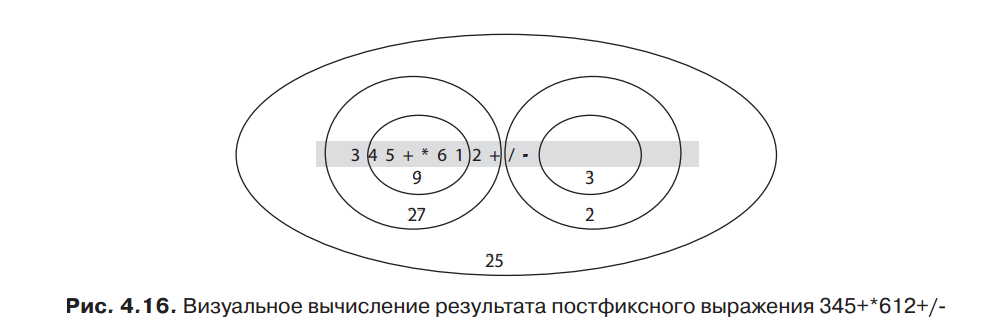
Но прежде чем показывать, насколько упрощается алгоритм, давайте посмотрим,

как подобные вычисления выполняются человеком.

**Как мы вычисляем результат постфиксного выражения**

На рис. 4.16 показано, как человек вычисляет результат постфиксного выражения,

пользуясь карандашом и собственными глазами.



Начните с первого оператора от левого края выражения. Обведите кружком

его и два операнда слева от него. Примените оператор к этим двум операндам, вы-

полните соответствующие вычисления и запишите результат в кружке. На рисунке

вычисление 4 + 5 дает 9.

Теперь перейдите к следующему оператору справа и обведите кружком его, уже

нарисованный кружок и операнд слева от него. Примените оператор к предыдуще-

му результату и новому операнду, запишите результат в новом кружке (3 × 9 = 27).

Продолжайте, пока не будут применены все операторы: 1 + 2 = 3, 6/3 = 2. Ответом

является результат в самом большом кружке: 27 – 2 = 25.

**Правила вычисления результатов**

**постфиксных выражений**

Как написать программу, реализующую этот процесс вычисления? Как видно из

предшествующего описания, оператор каждый раз применяется к двум послед-

ним операндам, которые вы встречаете при обработке выражения. Это наводит

на мысль, что операнды будет уместно хранить в стеке (в отличие от алгоритма

преобразования инфиксной записи в постфиксную, в котором в стеке сохранялись

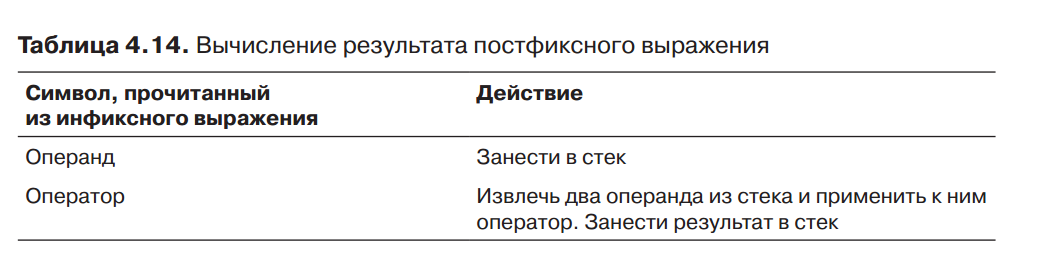
операторы). Правила вычисления результата постфиксных выражений перечис-

лены в табл. 4.14.

Когда обработка будет завершена, остается только извлечь окончательный

результат из стека. Вот и все! Этот процесс является компьютерным аналогом про-

цесса с рисованием кружков на рис. 4.16.



**Вычисление результата постфиксных выражений**

**на языке Java**

В преобразовании инфиксных выражений в постфиксные числа обозначались

символами (A, B и т. д.). Такое решение работало, потому что мы не выполняли с

операндами арифметические операции, а только записывали их в другом формате.

Теперь мы собираемся вычислить результат постфиксного выражения, а это

потребует выполнения арифметических операций и получения ответа. Следова-

тельно, операнды должны быть настоящими числами. Для упрощения программи-

рования будем считать, что операнды состоят только из одной цифры.

Наша программа вычисляет и выводит результат постфиксного выражения.

Не забывайте, что операнды содержат не более одной цифры. Пример простого

взаимодействия с программой:

Enter postfix: 57+

5 Stack (bottom-->top):

7 Stack (bottom-->top): 5

+ Stack (bottom-->top): 5 7

Evaluates to 12

Входная строка содержит только цифры и операторы без пробелов. Программа

находит числовой эквивалент выражения. Хотя операнды могут состоять только

из одной цифры, на результаты (в том числе и промежуточные) это ограничение

не распространяется. Как и в программе infix.java, содержимое стека на каждом

шаге выводится методом displayStack(). Код программы postfix.java приведен

в листинге 4.8.

**Листинг 4.8.** Программа postfix.java

// postfix.java

// Разбор постфиксных арифметических выражений

// Запуск программы: C>java PostfixApp

import java.io.\*; // Для ввода/вывода

////////////////////////////////////////////////////////////////

class StackX

{

private int maxSize;

private int[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int size) // Конструктор

{

maxSize = size;

stackArray = new int[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(int j) // Размещение элемента на вершине стека

{ stackArray[++top] = j; }

//--------------------------------------------------------------

public int pop() // Извлечение элемента с вершины стека

{ return stackArray[top--]; }

//--------------------------------------------------------------

public int peek() // Чтение элемента на вершине стека

{ return stackArray[top]; }

//--------------------------------------------------------------

public boolean isEmpty() // true, если стек пуст

{ return (top == -1); }

//--------------------------------------------------------------

public boolean isFull() // true, если стек заполнен

{ return (top == maxSize-1); }

//--------------------------------------------------------------

public int size() // Текущий размер стека

{ return top+1; }

//--------------------------------------------------------------

public int peekN(int n) // Чтение элемента с индексом n

{ return stackArray[n]; }

//--------------------------------------------------------------

public void displayStack(String s)

{

System.out.print(s);

System.out.print("Stack (bottom-->top): ");

for(int j=0; j<size(); j++)

{

System.out.print( peekN(j) );

System.out.print(' ');

}

System.out.println("");

}

//--------------------------------------------------------------

} // Конец класса StackX

////////////////////////////////////////////////////////////////

class ParsePost

{

private StackX theStack;

private String input;

//--------------------------------------------------------------

public ParsePost(String s)

{ input = s; }

//--------------------------------------------------------------

public int doParse()

{

theStack = new StackX(20); // Создание объекта стека

char ch;

int j;

int num1, num2, interAns;

for(j=0; j<input.length(); j++) // Для каждого символа

{

ch = input.charAt(j); // Чтение символа

theStack.displayStack(""+ch+" "); // \*диагностика\*

if(ch >= ‘0’ && ch <= ‘9’) // Если это цифра

theStack.push( (int)(ch-’0’) ); // Занести в стек

else // Если это оператор

{

num2 = theStack.pop(); // Извлечение операндов

num1 = theStack.pop();

switch(ch) // Выполнение арифметической

{ // операции

case ‘+’:

interAns = num1 + num2;

break;

case '-':

interAns = num1 - num2;

break;

case '\*':

interAns = num1 \* num2;

break;

case '/':

interAns = num1 / num2;

break;

default:

interAns = 0;

}

theStack.push(interAns); // Занесение промежуточного

} // результата в стек

}

interAns = theStack.pop(); // Получение результата

return interAns;

}

} // Конец класса ParsePost

////////////////////////////////////////////////////////////////

class PostfixApp

{

public static void main(String[] args) throws IOException

{

String input;

int output;

while(true)

{

System.out.print("Enter postfix: ");

System.out.flush();

input = getString(); // Ввод строки с клавиатуры

if( input.equals("") ) // Завершение, если нажата клавиша

break; // [Enter]

// Создание объекта для разбора выражения

ParsePost aParser = new ParsePost(input);

output = aParser.doParse(); // Обработка выражения

System.out.println("Evaluates to " + output);

}

}

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // Конец класса PostfixApp

////////////////////////////////////////////////////////////////

Метод main() класса PostfixApp получает постфиксную строку от пользователя

и создает объект ParsePost, инициализированный этой строкой. Затем он вызывает

метод doParse() класса ParsePost для выполнения непосредственной обработки.

Метод doParse() последовательно читает символы входной строки. Если символ

является цифрой, он заносится в стек; если он представляет оператор, то немед-

ленно применяется к двум операторам на вершине стека. (Операторы заведомо

находятся в стеке, потому что входная строка записана в постфиксном формате.)

Результат арифметической операции заносится в стек. После чтения и приме-

нения последнего символа (которым должен быть оператор) стек содержит только

один элемент — результат вычисления всего выражения.

Пример выполнения программы для более сложной входной строки: постфикс-

ного выражения 345 + × 612+/–, обработка которого была продемонстрирована на

рис. 4.16. Это выражение соответствует инфиксной записи 3 × (4 + 5) – 6/(1 + 2).

(Эквивалентное преобразование с символьными именами вместо цифр приво-

дилось в предыдущем разделе: выражение A × (B + C) – D/(E + F) в инфиксной

записи соответствует ABC + × �F +/– в постфиксной записи.) А вот как проис-

ходит вычисление в программе postfix.java:

Enter postfix: 345+\*612+/-

3 Stack (bottom-->top):

4 Stack (bottom-->top): 3

5 Stack (bottom-->top): 3 4

+ Stack (bottom-->top): 3 4 5

\* Stack (bottom-->top): 3 9

6 Stack (bottom-->top): 27

1 Stack (bottom-->top): 27 6

2 Stack (bottom-->top): 27 6 1

+ Stack (bottom-->top): 27 6 1 2

/ Stack (bottom-->top): 27 6 3

- Stack (bottom-->top): 27 2

Evaluates to 25

Программа postfix.java, как и infix.java (см. листинг 4.7), не проверяет входные

данные. Если ввести некорректное постфиксное выражение, то результат выпол-

нения будет непредсказуемым.

Поэкспериментируйте с программой. Попробуйте вводить разные постфикс-

ные выражения и понаблюдайте за их обработкой — так вы поймете суть процесса

быстрее, чем если будете читать о нем.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Связанные списки**

В главе 2, «Массивы», было показано, что хранение данных в массивах имеет ряд

недостатков. В неупорядоченном массиве поиск выполняется относительно мед-

ленно, тогда как в упорядоченном массиве медленно выполняется вставка. Удале-

ние выполняется медленно в обеих разновидностях массивов. Кроме того, размер

массива невозможно изменить после его создания.

В этой главе рассматриваются связанные списки — структура данных, которая

решает некоторые из этих проблем. Вероятно, связанный список является второй

по популярности структурой данных после массива.

Гибкость связанных списков хорошо подходит для многих общих задач хра-

нения данных. Кроме того, связанный список может заменить массив в качестве

базы для других структур хранения данных (таких, как стеки и очереди). Более

того, связанные списки часто могут использоваться вместо массивов (исключение

составляют ситуации с частым произвольным доступом к отдельным элементам

по индексу).

Связанные списки не решают всех проблем хранения данных, но они на ред-

кость универсальны, а на концептуальном уровне более просты, чем другие по-

пулярные структуры данных (например, деревья). Их достоинства и недостатки

будут проанализированы по мере изложения материала.

В этой главе будут рассмотрены простые связанные списки, двусторонние

списки, сортированные списки, двусвязные списки, а также списки с итераторами

(решение проблемы произвольного доступа к элементам списка). Также мы по-

знакомимся с концепцией абстрактных типов данных (ADT), увидим, как стеки

и очереди интерпретируются как типы ADT и как они реализуются на базе связан-

ных списков вместо массивов.