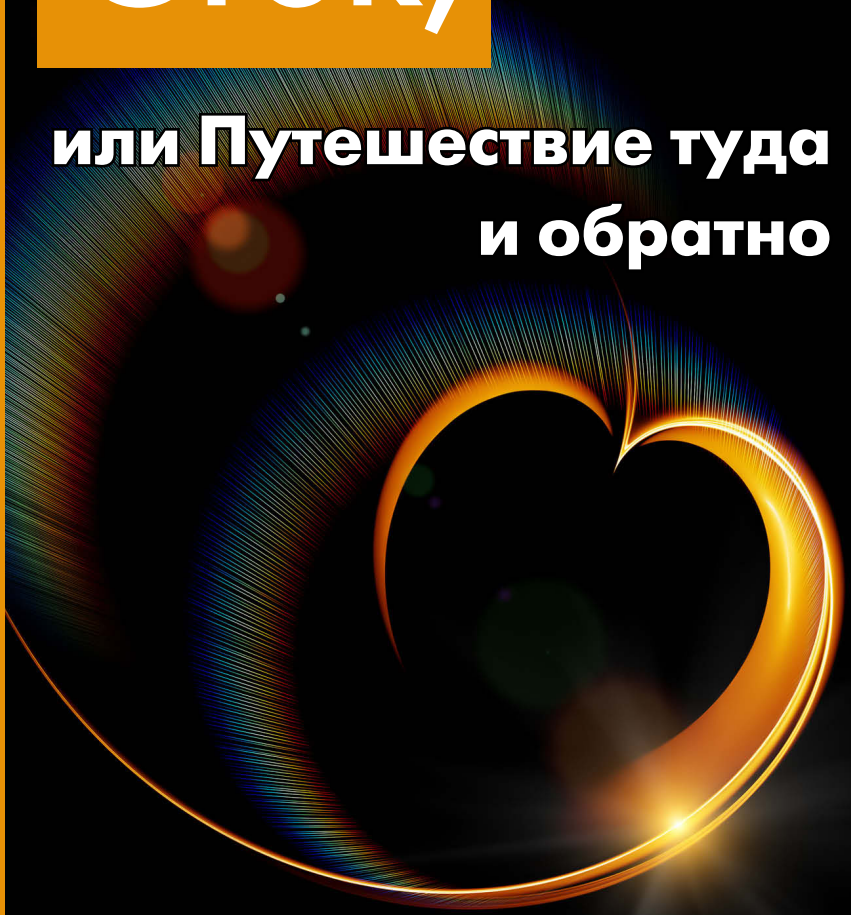
А. Вторников

Стек,

или Путешествие туда и обратно



ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

УДК 004.6:004.42
ББК 32.972
В87

Вторников А. А.
В87 Стек, или Путешествие туда и обратно. – М.: ДМК Пресс, 2017. – 140 с.: ил.

ISBN 978-5-97060-517-2

Книга посвящена простой и удивительно элегантной структуре данных – стеку. Описаны скобочные структуры, подпрограммы (в том числе рекурсивные), передача параметров, разбор и вычисление выражений, распознавание последовательностей символов. Также рассмотрено описание устройства и реализация простой, но достаточно мощной стековой машины; приведены многочисленные примеры программ, а также список задач, в том числе нетривиальных. На сайте издательства dmkpress.com содержатся дополнительные материалы, среди которых исходные коды простого транслятора стековой машины (на языке Java).

Издание предназначено прежде всего пытливым старшеклассникам, студентам вузов, а также тем, для кого программирование – хобби.

УДК 004.6:004.42
ББК 32.972

ISBN 978-5-97060-517-2

Содержание

Часть I

Задачи, приводящие к стеку

Скобочные структуры: элементарный случай

Мы начнем с того, что рассмотрим одну простую задачу, которая послужит отправной точкой всего последующего изложения. Эта задача встречается в разнообразных вариантах (и чуть позже мы расскажем об этом подробнее), но нам пока вполне достаточно такой формулировки: определить – правильно ли расставлены скобки в произвольном алгебраическом выражении. Несколько примеров даст представление о том, что мы имеем в виду и какое решение ищем:

```
a + (b * c /(d – e))
a + (b * c /d – e)
(a + b – c * (d / (e + f)))
```

Если всем входящим в эти выражения переменным (`a`, `b`, ...) придать определенные числовые значения, то эти выражения могут быть вычислены по правилам элементарной арифметики (конечно, исключая случаи, когда знаменатели дробей обратятся в 0). В этих примерах выражений скобки расставлены правильно. Чтобы сделать последнее утверждение более наглядным, давайте оставим только скобки, убрав все «лишнее»; в результате у нас получится следующее:

```
(())
()
(((())))
```

Такого рода выражения будем называть *скобочными структурами*; мы полностью игнорируем все то, что содержится внутри скобок, и сосредоточиваем внимание только на скобках и их взаимном расположении. Все приведенные выше скобочные структуры, очевидно, корректны, т. е. сформированы правильно. Теперь приведем несколько примеров некорректных (неправильно сформированных) скобочных структур:

```
(()
())
)(
```

Давайте немного остановимся на последних примерах и проанализируем, что именно делает эти скобочные структуры некорректными (ошибочными или неправильно сформированными). Для этого удобно называть левые скобки «(» открывающими, а правые «)» – закрывающими.

В первом примере число открывающих скобок больше числа закрывающих. Во втором примере ситуация противоположная – число открывающих скобок меньше числа закрывающих. В таких случаях говорят, что скобки *не сбалансированы* по числу вхождений. Третий пример демонстрирует еще один случай: число открывающих скобок совпадает с числом закрывающих (т. е. скобки сбалансированы по числу вхождений), но скобочная структура тем не менее ошибочна – выражение не может начинаться с закрывающей скобки или заканчиваться открывающей.

Наша ближайшая задача – найти способ (алгоритм) проверки любой заданной скобочной структуры, состоящей только из скобок одного типа (в данном случае скобочной структуры, состоящей только из круглых скобок).

Очевидно, что существует бесконечное количество как правильно, так и неправильно сформированных скобочных структур. Поэтому попытка описать все возможные варианты скобочных структур их простым перечислением не осуществима даже теоретически – рано или поздно встретится такая скобочная структура, которая не была нами предусмотрена. В таком случае программа, осуществляющая проверку скобочной структуры, будет не в состоянии определить ее правильность. Следовательно, нужен иной способ. Такой способ (и очень простой) существует, но, перед тем как описывать его, мы предлагаем читателю немного подумать и попытаться найти его самостоятельно.

Если последовательность скобок начинается с закрывающей или завершается открывающей скобкой (наш третий пример), то, бесспорно, такая скобочная структура сформирована неправильно. Но эти два предельных случая не позволяют решить задачу в общем виде, т. к. первые примеры неправильно сформированных скобочных структур начинаются и завершаются нужными видами скобок – соответственно «(» и «)», но которые тем не менее ошибочны.

Итак, в решении задачи мы пока почти никуда не продвинулись. Но давайте посмотрим на проблему чуть иначе. Отвлечемся на несколько минут от скобок. Представьте лифт. Во избежание аварии необходимо контролировать массу груза (включая людей), перевозимого лифтом. Для этого в лифт встраивается специальный датчик. Каждый внесенный груз или входящий человек увеличивает общую массу груза в кабине и нагрузку на тросы и лебедку лифта. Разумеется, показания датчика увеличиваются. Каждый выходящий из кабины лифта уменьшает массу груза в кабине, и показания датчика уменьшаются.

Наш «груз» – это два вида скобок. Открывающая скобка увеличивает «массу», а закрывающая – уменьшает. Конечно, как и всякая аналогия, наш пример страдает неточностями и упрощениями, но он тем не менее наводит на идею.

Припишем каждой скобке некий «вес»: открывающей скобке – положительное число (скажем, 1), а закрывающей – число, равное по абсолютной величине, но противоположное по знаку (т. е. –1). Датчиком пусть будет целочисленная переменная (назовем ее, например, count) с начальным значением, равным 0 (в примере с лифтом это означает, что изначально кабина пуста). Эта переменная будет играть роль счетчика. Промоделируем поведение нашей «системы» на самой простой скобочной структуре: `()`. Для компактности будем представлять отдельные состояния в виде таблицы, в левой колонке которой указывается уже прочитанная часть скобочной структуры, а в правой (в той же строке) – значение счетчика. Получаем следующий протокол разбора:

```
Прочитанная часть   Счетчик
                       0
(                      1
()                     0
```

В самом начале (когда еще ни одна скобка не прочитана) счетчик равен 0. После обнаружения открывающей скобки соответствующее

ей число (т. е. 1) прибавляется к счетчику. После обнаружения закрывающей скобки соответствующее ей число (т. е. –1) также прибавляется к счетчику. Ясно, что итогом арифметической операции 1 + (–1) является 0 (лифт пуст). Если скобочная структура сформирована правильно, то значение счетчика по окончании разбора будет равно 0. Для закрепления (и дополнительной проверки) разберем другую скобочную структуру (()). Вот протокол разбора:

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
(())                   0
```

Похоже, что для правильно сформированных скобочных структур этот метод работает. А что можно сказать о неправильно сформированных? Рассмотрим это на примере скобочной структуры (():

```
Прочитанная часть    Счетчик
                       0
(                      1
((                     2
(()                    1
```

Вся скобочная структура прочитана, а значение счетчика положительно (в нашем случае равно 1). Если обратиться к аналогии с лифтом, то это означает, что кто-то вошел в лифт и не выходит из него. Из этого немедленно следует, что число открывающих скобок в скобочном выражении больше числа закрывающих и скобочное выражение в целом сформировано неверно. Противоположный случай ()) дает следующий результат:

```
Прочитанная часть    Счетчик
                       0
(                      1
()                     0
())                    -1
```

Вся скобочная структура прочитана, а значение счетчика отрицательно (в нашем случае равно –1). Из этого немедленно следует, что число открывающих скобок в скобочном выражении меньше числа закрывающих и скобочное выражение сформировано неверно (из лифта вышло больше людей, чем в него вошло, что, конечно, невозможно). Наконец, рассмотрим случай)(:

```
Прочитанная часть    Счетчик
                       0
)                      -1
```

Как только значение счетчика станет отрицательным, то оставшуюся часть скобочной структуры можно не проверять (если лифт пуст, то из него некому выходить) – она заведомо ошибочна, и, следовательно, вся эта скобочная структура сформирована неверно. Необходимо сразу прекратить разбор и отвергнуть эту скобочную структуру как недопустимую. Почему нужно прекратить разбор, не проверяя следующих скобок? Во-первых, это бессмысленно, а во-вторых, открывающая скобка установит значение счетчика в 0, и мы можем решить, что такая скобочная структура допустима, хотя, очевидно, это не так. В качестве легкого упражнения проведите разбор следующей скобочной структуры: `(()))`. Первые четыре скобки установят значение счетчика в 0, но последняя (закрывающая скобка) лишняя, и все скобочное выражение должно быть отвергнуто. Кстати, продолжая, что можно сказать о скобочных структурах `(()())`, `((()()` или `(()()`?

Итак, задача решена. Для скобок одного типа (в нашем примере круглых) существует простой и эффективный алгоритм проверки структуры скобок.

Тип скобок может быть и иным, например фигурные `{}`, угловые `<>` или квадратные `[]`. Более того, помимо указанных скобок, в языках программирования используются и другие конструкции, которые могут быть отнесены к скобкам, например begin-end или try-catch (правда, обычно такого рода конструкции называют блоками, но суть дела от этого принципиально не меняется). По-существу, скобками являются и многострочные комментарии вида `/* ... */`. Иными словами, не важно, что мы называем скобками и как они выглядят; скобкой может быть все, что выполняет функции скобки.

Последнее утверждение служит иллюстрацией т. н. «утиного теста»: если нечто похоже на утку, плавает как утка и крякает как утка, то это, скорее всего, утка. Иначе говоря, нет необходимости связывать себя только общепринятыми типами скобок; важна сущность, выполняемые функции.

Простые скобочные структуры, подобные рассмотренным только что, составляют лишь небольшую часть синтаксических структур в языках программирования. Обычно встречаются более сложные скобочные структуры, т. е. скобочные структуры, включающие разные типы скобок. Вот небольшой пример:

```
{
int [] keys;
ArrayList <String> aList;
...
for (int i = 0; i < keys.length(); i++) {
        aList.get (i) = “”;
    }
}
```

Перед нами фрагмент исходного кода (на языке программирования Java, но подобный пример можно встретить во многих других языках), в котором встречаются все четыре «стандартных» типа скобок. Задача все та же: определить, правильно ли сформирована скобочная структура в том случае, когда типов скобок не один, а несколько. Давайте, как и в первом случае, уберем все «лишнее» и оставим только скобки. Вот что у нас получится:

```
{[]<>(()){()}}
```

Как убедиться в правильности скобочной структуры такого вида? Сразу же приходит в голову мысль воспользоваться только что разработанным методом подсчета, приписывая каждой открывающей и каждой закрывающей скобке некий «вес», т. е. определенное число, характеризующее вид скобки – открывающая или закрывающая. Пусть, как и прежде, любой открывающей скобке соответствует 1, а любой закрывающей –1. Для нашего примера после просмотра скобок получится 0 (и при этом счетчик никогда не станет отрицательным), т. е. мы можем сделать вывод, что скобочная структура правильна.

Итак, задача решена? Как бы не так! Посмотрим на такую, например, скобочную структуру: `{(})`. Наш алгоритм даст в результате значение 0, при этом значение счетчика никогда не будет отрицательным. Но эта скобочная структура, совершенно очевидно, сформирована неверно. Вывод: алгоритм со счетчиком не работает. Хуже всего то, что он выдает результат, который выглядит правильным (т. к. счетчик по окончании разбора действительно равен 0), но которому нельзя верить. Надо понять – почему он не работает и можно ли его исправить так, чтобы он был пригоден в новой ситуации с различными типами скобок.

Вообще говоря, причина лежит на поверхности: мы увеличиваем (соответственно, уменьшаем) значение счетчика всякий раз, когда в последовательности встречается открывающая (закрывающая)

скобка. В примере {(}) мы увеличиваем значение счетчика последовательно дважды: сначала, когда встречается скобка {, и потом, когда встречается скобка (. Но ведь это разные типы скобок! Не тут ли корень проблемы? Нельзя ли изменить способ изменения значений счетчика так, чтобы он соответствовал типу скобки? Можно, конечно, и будет правильно, если читатель попробует поискать решение в этом направлении.

Методы анализа скобочных структур стали исследоваться еще в 50-х годах XX века. Если в выражении встречаются скобки только одного вида, то задача, как мы уже видели, решается элементарно. Но по мере усложнения и развития языков программирования скобочные структуры стали включать в себя скобки разных типов, и проблема стала актуальной. Был предложен ряд методов решения (в их числе весьма оригинальные), но все эти методы были сложными в реализации и медленными в работе. Если читатель уже попытался найти такой метод, он, безусловно, должен был убедиться, что проблема нетривиальна. Можно вводить отдельные счетчики для каждого вида скобок, но проблема согласования вложенности одних типов скобок в другие все равно оставалась. Метод со счетчиком, при всей его элегантности, для составных скобочных структур работает неудовлетворительно. Нужно что-то иное, и сейчас мы переходим к обсуждению метода, совершенно отличного от метода подсчета с использованием счетчика.

Вернемся еще раз к нашему примеру {(}). Он, как мы знаем, соответствует недопустимой скобочной структуре. Проанализируем скобки по порядку. Первой встречается открывающая { скобка. Чтобы скобочная структура стала допустимой, где-то в последовательности скобок должна быть закрывающая } скобка. Она, конечно, присутствует (в примере это третья скобка слева). Но, прежде чем мы до нее «доберемся», надо что-то сделать с другой открывающей скобкой, только теперь это (. Для того чтобы (третья по порядку) закрывающая фигурная скобка } соответствовала первой открывающей {, нужно обеспечить для открывающей круглой скобки парную ей закрывающую. Принято говорить, что круглая скобка (вложена в фигурную {, или, чуть более формально, что у открывающей круглой скобки *глубина вложенности* больше, чем у открывающей фигурной. Прежде чем закрывать внешние скобки (в данном случае { и }), нужно, чтобы были закрыты все внутренние. В нашем примере как раз это очевидное правило и не соблюдается: фигурная скобка }, относящаяся к паре с меньшей глубиной вложенности (внешней), появляется

раньше круглой), которая относится к скобкам большей глубины вложенности (внутренней).

Может быть, небольшой пример поможет лучше понять эти довольно абстрактные рассуждения. Представьте себе закрытую шкатулку, внутри которой лежит закрытый конверт. Нам нужно прочесть содержимое конверта, т. е. лежащее в нем письмо, а затем все вернуть в исходное состояние. Шкатулка соответствует скобкам { и }, а конверт – скобкам (и). Прежде чем мы доберемся до конверта, надо открыть шкатулку. Этому соответствует скобка {. Далее мы должны открыть конверт, т. е. скобку (. После этого мы достаем письмо, читаем его, прячем обратно в конверт и закрываем конверт; этому соответствует, разумеется, закрывающая) скобка. Наконец, мы закрываем шкатулку, используя скобку }. Результат, очевидно, такой: {()}, и это правильно сформированная скобочная структура с правильным порядком вложенности. Наш начальный пример {(}) соответствует ситуации, которую можно описать так: открыть шкатулку, открыть конверт, достать письмо, закрыть шкатулку, закрыть конверт. Но как раз последнее действие выполнить и невозможно – шкатулка закрыта, и до конверта, лежащего в ней, добраться уже невозможно!

Работая над этой задачей, мы после некоторого размышления рано или поздно приходим к выводу о том, что решение, основанное на использовании счетчика (или даже нескольких счетчиков), по-видимому, следует отбросить. Счетчик хорошо работал для первого случая, когда у нас были скобки только одного типа. Его было вполне достаточно для хранения всей информации об уже просмотренном участке скобочной структуры; последующие скобки только модифицировали состояние счетчика, но не меняли существа обрабатываемой информации – тип скобки оставался прежним, и мы могли его игнорировать. Как только типов скобок стало больше, метод перестал работать, он подошел к границам применимости. Поступающая информация уже не может быть сохранена только с использованием счетчиков – информации не только стало больше, сама информация стала другой.

Метод с использованием счетчика прост и понятен: нам привычно что-то подсчитывать. Но всему наступает предел, за который надо выходить.

А теперь, после этого несколько затянувшегося описания причин безуспешных попыток анализа скобочной структуры с различными типами скобок, мы переходим к главному действующему лицу – стеку. Но, прежде чем мы продолжим, нам придется ненадолго прерваться.

Следующие несколько страниц будут очень важными, поскольку мы собираемся рассказать, что такое стек, опишем относящуюся к стеку терминологию, введем обозначения, которые позволяют избавиться от многословных описаний, и, наконец, расскажем о некоторых основных операциях со стеком. Возможно, не сразу будет понятно – для чего мы на время отложили нашу задачу, но потерпите. Все это необходимо будет тщательно изучить, так что давайте приступим.

Стек: знакомство

Обратимся к классике и позаимствуем определение стека у Дональда Кнута:

Стек – это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка.

Для знатока структур данных этого, скорее всего, будет достаточно. Но все-таки давайте уделим определению стека некоторое время. Самое важное в этом определении – последние слова, а именно «только на одном из концов». Наверное, проще всего проиллюстрировать, что такое стек, – привести несколько примеров, которые всем, надеемся, будут знакомы и понятны.

Первый пример стека мы обнаруживаем в детской игрушке. Всем, вероятно, знакома детская пирамидка: закрепленный на основании стержень, на который надеты разноцветные диски. Для того чтобы поместить на стержень новый диск, необходимо надеть его на свободный конец стержня. Чтобы убрать со стержня диск, его необходимо снять со свободного конца стержня. Таким образом, и добавление, и удаление дисков осуществляются всегда только с одного конца стержня; ни снимать, ни надевать диски со стороны основания пирамидки нельзя. Далее, невозможно снять диск, над которым есть другие диски, и невозможно добавить новый диск иначе, как поверх других. Диск, добавленный последним, можно будет снять первым; диск, добавленный первым, будет снят в последнюю очередь.

Еще один классический пример стека – стопка подносов в кафе самообслуживания. Мы можем взять только самый верхний поднос из стопки. Когда подносы заканчиваются, сотрудник кафе приносит новые. Поднос на самом верху стопки – это последний добавленный поднос; поднос на дне стопки – самый первый добавленный поднос.

Нет возможности (если, конечно, не стоит задача все поломать и испачкать) взять поднос из середины стопки. Равно как и нет возможности добавить поднос в середину стопки.

Чтобы, наконец, перейти к делу, приведем еще один пример: магазин для хранения патронов в огнестрельном оружии. Очевидно, что первым будет использован тот патрон, который был помещен в магазин последним. Патрон, добавленный в магазин первым, будет использован последним.

Во всех трех примерах мы видели один и тот же тип поведения: то, что было добавлено первым (диск, поднос, патрон), будет использовано последним. То же самое можно сказать и так: то, что было добавлено последним, будет использовано первым. Как раз для этой формы определения существует эквивалентная краткая «*Last In First Out*», или просто – *LIFO*.

Итак, операции вставки и удаления из определения, приведенного в начале раздела, подчиняются дисциплине LIFO. Такая структура данных и есть стек.

Теперь, когда общая идея стека стала (надеемся) понятной, настало время договориться о том, как стек можно изображать графически. Существуют три основных способа изображения стека, и все они широко используются в литературе по программированию и структурам данных. Первые два способа практически идентичны друг другу. Обсудим сначала их.

Поскольку стек, как и всякая другая структура данных (такая как массив, список, дерево и проч.), хранится в памяти, в основе первых двух способов представления стека лежит отображение стека как набора ячеек памяти. Напомним, что память – это последовательность ячеек. Каждая ячейка имеет номер (от 0 до некоторой величины), или *адрес*. Ячейки памяти с меньшими адресами считаются расположенными ближе к началу, а с большими – ближе к концу памяти. Все ячейки памяти совершенно равноправны, и нет никаких оснований считать те или иные из них более или менее важными.

Графически память изображается в виде последовательности смежных ячеек. Стек может расти как в направлении от меньших адресов к большим, так и наоборот – от больших адресов к меньшим. Если условиться, что ячейки с меньшими адресами (т. е. те, что расположены ближе к началу памяти) изображаются ниже ячеек с большими адресами, то первые два способа изображают стек так, как показано на следующем рисунке: