Глава 4
Стеки и очереди

В этой главе рассматриваются три структуры данных: стек, очередь и приоритетная очередь. Сначала будут описаны основные отличия этих структур от массивов, а затем мы рассмотрим каждую структуру по отдельности. Последний раздел посвящен разбору арифметических выражений — области, в которых стек играет особенно важную роль.

Другие структуры

Между структурами данных и алгоритмами, представленными в предыдущих главах, и теми, которые мы будем рассматривать сейчас, существуют значительные различия. Рассмотрим некоторые из них, прежде чем переходить к подробному анализу новых структур.

Инструменты программиста

Массивы, которые рассматривались ранее, а также многие другие структуры, которые встретятся нам позднее (связанные списки, деревья и т. д.), хорошо подходят для хранения информации, типичной для приложений баз данных. Они часто используются для хранения картотек персонала, данных складского учета, финансовых данных и т. д. — данных, представляющих объекты или операции реального мира. Эти структуры обеспечивают удобный доступ к данным: они упрощают вставку, удаление и поиск конкретных элементов.

С другой стороны, структуры и алгоритмы, описанные в этой главе, чаще используются в инструментарии программиста. Они предназначены скорее для упрощения программирования на концептуальном уровне, а не для полноценного хранения данных. Их жизненный цикл обычно короче, чем у структур баз данных. Они создаются и используются для выполнения конкретных задач во время работы программы; когда задача выполнена, эти структуры уничтожаются.

Ограничение доступа

В массиве возможен произвольный доступ к любому элементу — либо напрямую (если известен его индекс), либо поиском по последовательности ячеек, пока нужный элемент не будет найден. Напротив, структуры данных этой главы ограничи-

вают доступ к элементам: в любой момент времени можно прочитать или удалить только один элемент (если действовать по правилам, конечно).

Интерфейс этих структур проектируется с расчетом на поддержку ограничений доступа. Доступ к другим элементам (по крайней мере теоретически) запрещен.

Абстракция

Стеки, очереди и приоритетные очереди являются более абстрактными сущностями, чем массивы и многие другие структуры данных. Они определяются, прежде всего, своим интерфейсом: набором разрешенных операций, которые могут выполняться с ними. Базовый механизм, используемый для их реализации, обычно остается невидимым для пользователя.

Например, как будет показано в этой главе, базовым механизмом для стека может быть массив или связанный список, а базовым механизмом приоритетной очереди — массив или особая разновидность дерева, называемая *кучей*. Мы вернемся к теме реализации структур данных на базе других структур при обсуждении абстрактных типов данных (ADT) в главе 5, «Связанные списки».

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 $3 \times (4 + 5)$.

Стек также удобен в алгоритмах, применяемых при работе с некоторыми сложными структурами данных. В главе 8, «Двоичные деревья», приведен пример его применения при переборе узлов дерева, а в главе 13, «Графы», стек используется для поиска вершин графа (алгоритм, с помощью которого можно найти выход из лабиринта).

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 калькуляторах. Вместо того чтобы вводить арифметическое выражение с круглыми скобками, пользователь сохранял промежуточные результаты в стеке. Тема будет более подробно 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.

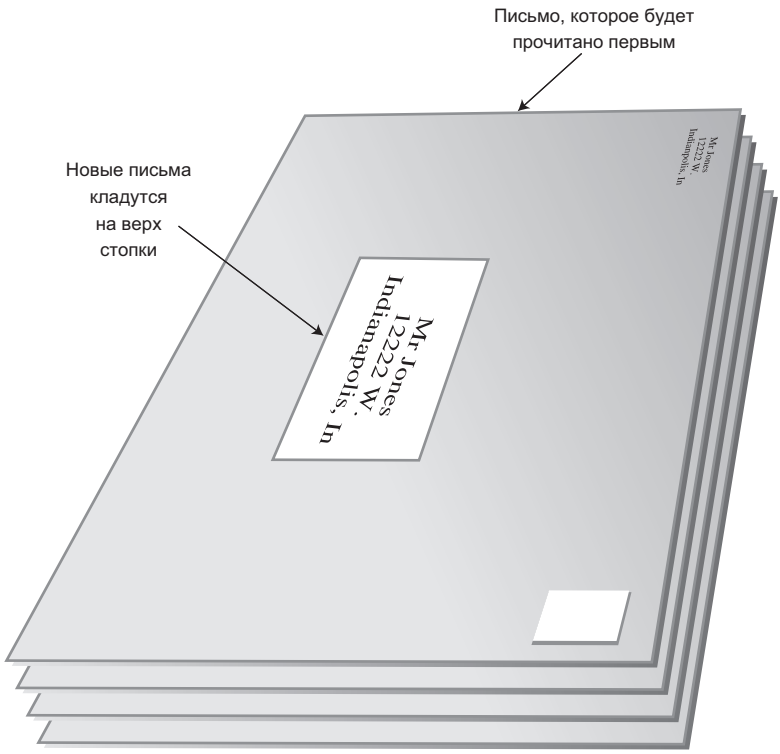


Рис. 4.1. Стопка писем

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 месяцами, 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 перекладывают 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 различных 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 командировочным расходам (C). Во время обсуждения вам срочно звонят из отдела 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 *извлечение* из стека — выполняются только на вершине стека, то есть с его верхним элементом. Говорят, что стек работает по принципу LIFO (Last-In-First-Out), потому что последний занесенный в стек элемент будет первым извлечен из него.

Приложение Stack Workshop

Приложение Stack Workshop поможет вам лучше понять, как работает стек. В окне приложения находятся четыре кнопки: `New`, `Push`, `Pop` и `Peek` (рис. 4.2).

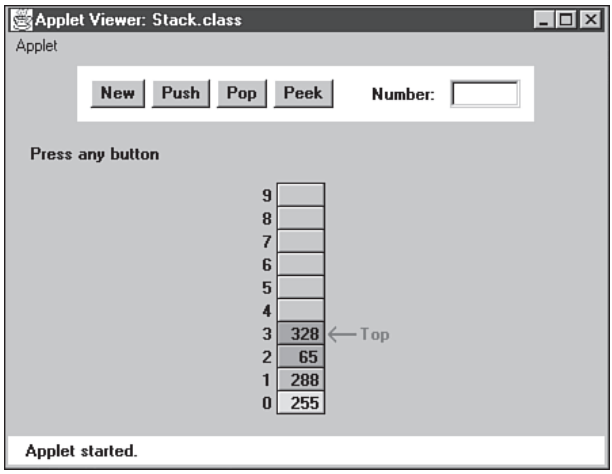


Рис. 4.2. Приложение Stack Workshop

В основу реализации Stack Workshop заложен массив, поэтому в окне выводится ряд ячеек. Однако доступ к стеку ограничивается его вершиной, поэтому вы не сможете обращаться к элементам по индексу. Концепция стека и базовая структура данных, используемая для ее реализации, — совершенно разные понятия. Как упоминалось ранее, стеки также могут реализовываться на базе других структур (например, связанных списков).

Кнопка New

Стек в приложении Stack Workshop изначально содержит четыре элемента. Если вы хотите, чтобы стек создавался пустым, кнопка New создаст стек без элементов. Следующие три кнопки выполняют основные операции со стеком.

Кнопка Push

Кнопка Push вставляет в стек новый элемент данных. После первого нажатия этой кнопки вам будет предложено ввести значение ключа нового элемента. Еще пара щелчков, и вставленный элемент оказывается на вершине стека.

Красная стрелка всегда указывает на вершину стека, то есть на последний вставленный элемент. Обратите внимание на то, как в процессе вставки на одном шаге (нажатии кнопки) смещается вверх указатель Top, а на втором элемент данных собственно вставляется в ячейку. Если бы эти действия выполнялись в обратном порядке, новое значение заменило бы существующий элемент по ссылке Top. При программировании реализации стека важно соблюдать порядок выполнения этих двух операций.

Если стек будет заполнен, то при попытке вставки очередного элемента будет выведено сообщение о ее невозможности. (Теоретически стек на базе ADT переполниться не может, но в реализации на базе массива переполнение не исключено.)

Кнопка Pop

Чтобы извлечь элемент данных с вершины стека, щелкните на кнопке Pop. Извлеченное значение выводится в текстовом поле Number; эта операция соответствует вызову метода pop().

И снова обратите внимание на последовательность действий: сначала элемент извлекается из ячейки, на которую указывает ссылка Top, а затем Top уменьшается и переводится к верхней занятой ячейке. Порядок выполнения этих действий противоположен порядку, используемому при операции Push.

В приложении при нажатии кнопки Pop элемент удаляется из массива, а ячейка окрашивается в серый цвет (признак отсутствия данных). Это не совсем точно — на самом деле удаленные элементы остаются в массиве до тех пор, пока не будут перезаписаны новыми данными. Однако после того, как маркер Top опустится ниже их позиции, эти элементы становятся недоступными, так что на концептуальном уровне они не существуют.

После извлечения из стека последнего элемента маркер Top указывает в позицию –1 под нижней ячейкой. Эта позиция является признаком того, что стек пуст. При попытке извлечь элемент из пустого стека выводится сообщение «Can't pop: stack is empty».

Кнопка Peek

Вставка и извлечение элементов — две основные операции со стеком. Тем не менее в некоторых ситуациях бывает полезно прочитать значение с вершины стека без его

удаления. Нажав кнопку Peek несколько раз, вы увидите, как значение элемента Top копируется в текстовое поле Number, но сам элемент остается в стеке.

Обратите внимание: «подсмотреть» можно только значение верхнего элемента. Архитектура стека такова, что все остальные элементы остаются невидимыми для пользователя.

Размер стека

Как правило, стек представляет собой небольшую, временную структуру данных; по этой причине мы и показываем стек, состоящий всего из 10 ячеек. Конечно, в реальных программах стек может содержать больше элементов, но на самом деле необходимый размер стека оказывается на удивление незначительным. Например, для разбора очень длинного арифметического выражения обычно хватает стека с 10–12 ячейками.

Реализация стека на языке Java

Программа stack.java реализует стек в виде класса с именем StackX. Листинг 4.1 содержит этот класс и короткий метод main() для его тестирования.

Листинг 4.1. Программа stack.java

```
// stack.java
// Работа со стеком
// Запуск программы: C>java StackApp
////////////////////////////////////////////////////////////////
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
```

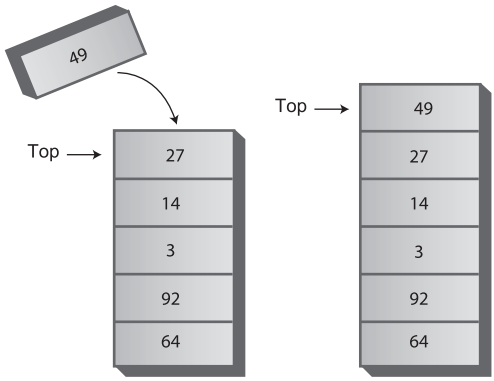
Обратите внимание на обратный порядок данных. Так как последний занесенный элемент стал первым извлеченным элементом, число 80 стоит на первом месте в выходных данных.

Эта версия класса `StackX` хранит элементы данных типа `long`. Как упоминалось в главе 3, «Простая сортировка», их можно заменить любым другим типом, в том числе и объектами.

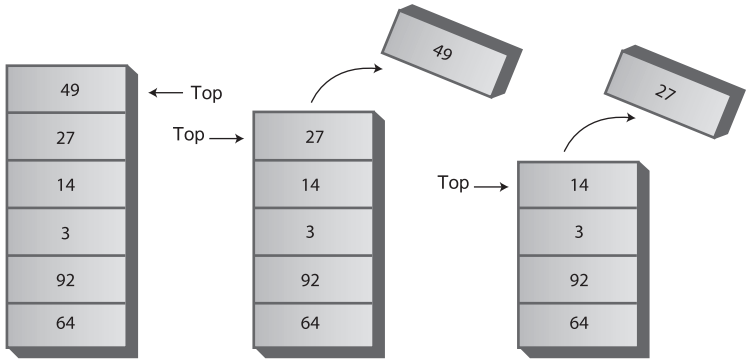
Методы класса StackX

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. (Обратите 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, находящуюся непосредственно над текущей ячейкой, и сохраняет в ней элемент данных. Еще раз обратите внимание: `top` увеличивается *до* вставки элемента.



Новый элемент заносится в стек



Два элемента извлекаются из стека

Рис. 4.3. Действие методов класса StackX

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.

Методы `isEmpty()` и `isFull()` возвращают `true`, если стек пуст или полон соответственно. 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 извлечения элемента из пустого стека?

Ответственность за обработку таких ошибок возлагается на пользователя класса. 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 простой программе мы знаем, что стек не заполнен, потому что только что его сами инициализировали). Проверка пустого стека выполняется в методе `main()` при вызове `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 извлекаются в порядке, обратном порядку их занесения. Код программы `reverse.java` приведен в листинге 4.2.

Листинг 4.2. Программа reverse.java

```
// reverse.java
// Использование стека для перестановки символов строки
// Запуск программы: C>java ReverseApp
import java.io.*;                  // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
```

продолжение ⇩

Листинг 4.2 (*продолжение*)

```
   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
         }
```