

Оглавление

Введение	18
Второе издание	18
Дополнительные темы	18
Вопросы	19
Упражнения	19
Программные проекты	19
О чем эта книга	19
Чем эта книга отличается от других	20
Доступность	20
Приложения Workshop	21
Примеры кода Java	21
Для кого написана эта книга	22
Что необходимо знать читателю	22
Необходимые программы	22
Как организован материал книги	22
Вперед!	24
Глава 1. Общие сведения	25
Зачем нужны структуры данных и алгоритмы?	25
Хранение реальных данных	26
Инструментарий программиста	26
Моделирование	27
Обзор структур данных	27
Алгоритмы	28
Определения	28
База данных	29
Запись	29
Поле	29
Ключ	30
Объектно-ориентированное программирование	30
Недостатки процедурных языков	30
Объекты в двух словах	31
Создание объектов	32
Вызов методов объекта	33
Пример объектно-ориентированной программы	33
Наследование и полиморфизм	36
Программотехника	37

Java для программистов C++	37
В Java нет указателей	37
Ввод/вывод	41
Вывод	41
Структуры данных библиотеки Java	44
Итоги	44
Вопросы	45

Глава 2. Массивы **46**

Приложение Array Workshop	46
Вставка	48
Поиск	48
Удаление	49
Проблема дубликатов	50
Не слишком быстро	51
Поддержка массивов в Java	52
Создание массива	52
Обращение к элементам массива	52
Инициализация	53
Пример массива	53
Деление программы на классы	56
Классы LowArray и LowArrayApp	58
Интерфейсы классов	58
Не слишком удобно	59
Кто чем занимается?	59
Пример highArray.java	60
Удобство пользователя	63
Абстракция	63
Приложение Ordered Workshop	63
Линейный поиск	64
Двоичный поиск	65
Реализация упорядоченного массива на языке Java	67
Двоичный поиск с методом find()	67
Класс OrdArray	69
Преимущества упорядоченных массивов	72
Логарифмы	72
Формула	73
Операция, обратная возведению в степень	74
Хранение объектов	75
Класс Person	75
Программа classDataArray.java	76
О-синтаксис	79
Вставка в неупорядоченный массив: постоянная сложность	80
Линейный поиск: сложность пропорциональна N	80
Двоичный поиск: сложность пропорциональна $\log(N)$	80
Константа не нужна	81
Почему бы не использовать только массивы?	82
Итоги	83
Вопросы	84

Упражнения	85
Программные проекты	85
Глава 3. Простая сортировка	87
Как это делается?	87
Пузырьковая сортировка	89
Пример пузырьковой сортировки	89
Приложение BubbleSort Workshop	91
Реализация пузырьковой сортировки на языке Java	94
Инварианты	97
Сложность пузырьковой сортировки	97
Сортировка методом выбора	98
Пример сортировки методом выбора	98
Приложение SelectSort Workshop	100
Реализация сортировки методом выбора на языке Java	101
Инвариант	103
Сложность сортировки методом выбора	103
Сортировка методом вставки	104
Пример сортировки методом вставки	104
Приложение InsertSort Workshop	106
Сортировка 10 столбцов	106
Реализация сортировки методом вставки на языке Java	108
Инварианты сортировки методом вставки	111
Сложность сортировки методом вставки	111
Сортировка объектов	112
Реализация сортировки объектов на языке Java	112
Лексикографические сравнения	115
Устойчивость сортировки	115
Сравнение простых алгоритмов сортировки	116
Итоги	116
Вопросы	117
Упражнения	118
Программные проекты	119
Глава 4. Стеки и очереди	121
Другие структуры	121
Инструменты программиста	121
Ограничение доступа	121
Абстракция	122
Стеки	122
Почтовая аналогия	123
Приложение Stack Workshop	124
Реализация стека на языке Java	126
Пример использования стека № 1. Перестановка букв в слове	129
Пример № 2. Поиск парных скобок	132
Эффективность стеков	136
Очереди	136
Приложение Queue Workshop	137
Циклическая очередь	140

Реализация очереди на языке Java	141
Эффективность очередей	146
Дек	146
Приоритетные очереди	146
Приложение The PriorityQ Workshop	148
Реализация приоритетной очереди на языке Java	150
Эффективность приоритетных очередей	152
Разбор арифметических выражений	152
Постфиксная запись	153
Преобразование инфиксной записи в постфиксную	154
Как мы вычисляем результаты инфиксных выражений	154
Вычисление результата постфиксного выражения	170
Итоги	175
Вопросы	176
Упражнения	178
Программные проекты	178
Глава 5. Связанные списки	180
Строение связанного списка	180
Ссылки и базовые типы	181
Отношения вместо конкретных позиций	182
Приложение LinkList Workshop	183
Вставка	183
Поиск	184
Удаление	184
Простой связанный список	185
Класс Link	185
Класс LinkList	186
Программа linkList.java	190
Поиск и удаление заданных элементов	192
Метод find()	195
Метод delete()	195
Другие методы	196
Двусторонние списки	196
Эффективность связанных списков	200
Абстрактные типы данных	200
Реализация стека на базе связанного списка	201
Реализация очереди на базе связанного списка	204
Типы данных и абстракция	207
Списки ADT	208
Абстрактные типы данных как инструмент проектирования	209
Сортированные списки	209
Реализация вставки элемента в сортированный список на языке Java	211
Программа sortedList.java	212
Эффективность сортированных списков	214
Сортировка методом вставки	215
Двусвязные списки	217
Перебор	219
Вставка	219

Удаление	221
Программа doublyLinked.java	222
Двусвязный список как база для построения дека	226
Итераторы	226
Ссылка на элемент списка?	227
Итератор	227
Другие возможности итераторов	228
Методы итераторов	229
Программа interIterator.java	230
На что указывает итератор?	236
Метод atEnd()	236
Итеративные операции	236
Другие методы	238
Итоги	238
Вопросы	239
Упражнения	240
Программные проекты	241
Глава 6. Рекурсия	243
Треугольные числа	243
Вычисление n-го треугольного числа в цикле	244
Вычисление n-го треугольного числа с применением рекурсии	245
Программа triangle.java	247
Что реально происходит?	248
Характеристики рекурсивных методов	249
Насколько эффективна рекурсия?	250
Математическая индукция	250
Факториал	250
Анаграммы	252
Рекурсивный двоичный поиск	257
Замена цикла рекурсией	258
Алгоритмы последовательного разделения	262
Ханойская башня	262
Приложение Towers Workshop	263
Перемещение поддеревьев	264
Рекурсивный алгоритм	264
Программа towers.java	266
Сортировка слиянием	267
Слияние двух отсортированных массивов	268
Сортировка слиянием	270
Приложение MergeSort Workshop	273
Программа mergeSort.java	274
Устранение рекурсии	281
Что дальше?	287
Интересные применения рекурсии	289
Возведение числа в степень	289
Задача о рюкзаке	291
Комбинации и выбор команды	292
Итоги	294

Вопросы	295
Упражнения	297
Программные проекты	297
Глава 7. Нетривиальная сортировка	299
Сортировка Шелла	299
Сортировка методом вставок: слишком много копирования	300
N-сортировка	300
Сокращение интервалов	302
Приложение Shellsort Workshop	303
Реализация сортировки Шелла на языке Java	305
Другие интервальные последовательности	307
Эффективность сортировки Шелла	308
Разбиение	309
Приложение Partition Workshop	309
Остановка и перестановка	313
Быстрая сортировка	316
Алгоритм быстрой сортировки	317
Выбор опорного значения	318
Приложение QuickSort 1 Workshop	323
Вырожденное быстроедействие $O(N^2)$	327
Определение медианы по трем точкам	328
Обработка малых подмассивов	333
Устранение рекурсии	336
Поразрядная сортировка	339
Проектирование программы	340
Эффективность поразрядной сортировки	340
Итоги	341
Вопросы	343
Упражнения	344
Программные проекты	344
Глава 8. Двоичные деревья	346
Для чего нужны двоичные деревья?	346
Медленная вставка в упорядоченном массиве	346
Медленный поиск в связанном списке	347
Деревья приходят на помощь	347
Что называется деревом?	347
Терминология	348
Аналогия	351
Как работают двоичные деревья?	352
Приложение Binary Tree Workshop	352
Представление деревьев в коде Java	354
Поиск узла	356
Поиск узла в приложении Workshop	356
Реализация поиска узла на языке Java	358
Эффективность поиска по дереву	358
Вставка узла	359
Вставка узла в приложении Workshop	359
Реализация вставки на языке Java	359

Обход дерева	361
Симметричный обход	361
Реализация обхода на языке Java	362
Обход дерева из трех узлов	362
Обход дерева в приложении Workshop	363
Симметричный и обратный обход	365
Поиск минимума и максимума	367
Удаление узла	368
Случай 1. Удаляемый узел не имеет потомков	368
Случай 2. Удаляемый узел имеет одного потомка	370
Случай 3. Удаляемый узел имеет двух потомков	372
Эффективность двоичных деревьев	379
Представление дерева в виде массива	381
Дубликаты ключей	382
Полный код программы tree.java	383
Код Хаффмана	391
Коды символов	391
Декодирование по дереву Хаффмана	393
Построение дерева Хаффмана	394
Кодирование сообщения	396
Создание кода Хаффмана	397
Итоги	397
Вопросы	399
Упражнения	400
Программные проекты	401

Глава 9. Красно-черные деревья 403

Наш подход к изложению темы	403
Концептуальное понимание	404
Нисходящая вставка	404
Сбалансированные и несбалансированные деревья	404
Вырождение до $O(N)$	405
Спасительный баланс	406
Характеристики красно-черного дерева	406
Исправление нарушений	408
Работа с приложением RBTree Workshop	408
Щелчок на узле	409
Кнопка Start	409
Кнопка Ins	409
Кнопка Del	409
Кнопка Flip	410
Кнопка RoL	410
Кнопка RoR	410
Кнопка R/B	410
Текстовые сообщения	410
Где кнопка Find?	410
Эксперименты с приложением Workshop	411
Эксперимент 1. Вставка двух красных узлов	411
Эксперимент 2. Повороты	412
Эксперимент 3. Переключение цветов	412

Эксперимент 4. Несбалансированное дерево	413
Эксперименты продолжаются	414
Красно-черные правила и сбалансированные деревья	414
Пустые потомки	415
Повороты	415
Простые повороты	416
Переходящий узел	416
Перемещения поддеревьев	417
Люди и компьютеры	419
Вставка узла	419
Общая схема процесса вставки	419
Переключения цветов при перемещении вниз	420
Повороты после вставки узла	422
Повороты при перемещении вниз	427
Удаление	430
Эффективность красно-черных деревьев	431
Реализация красно-черного дерева	431
Другие сбалансированные деревья	432
Итоги	433
Вопросы	433
Упражнения	435
Глава 10. Деревья 2-3-4	436
Знакомство с деревьями 2-3-4	436
Почему деревья 2-3-4 так называются?	437
Структура дерева 2-3-4	438
Поиск в дереве 2-3-4	439
Вставка	439
Разбиение узлов	440
Разбиение корневого узла	441
Разбиение при перемещении вниз	442
Приложение Tree234 Workshop	442
Кнопка Fill	443
Кнопка Find	443
Кнопка Ins	444
Кнопка Zoom	444
Просмотр разных узлов	445
Эксперименты	446
Реализация дерева 2-3-4 на языке Java	447
Класс Dataltem	447
Класс Node	447
Класс Tree234	448
Класс Tree234App	449
Полный код программы tree234.java	450
Деревья 2-3-4 и красно-черные деревья	457
Преобразование деревьев 2-3-4 в красно-черные деревья	457
Эквивалентность операций	459
Эффективность деревьев 2-3-4	461
Скорость	461
Затраты памяти	462

Деревья 2-3	462
Разбиение узлов	463
Реализация	465
Внешнее хранение	466
Обращение к внешним данным	466
Последовательное хранение	469
В-деревья	471
Индексирование	476
Сложные критерии поиска	478
Сортировка внешних файлов	479
Итоги	482
Вопросы	484
Упражнения	485
Программные проекты	485
Глава 11. Хеш-таблицы	487
Хеширование	487
Табельные номера как ключи	488
Индексы как ключи	488
Словарь	489
Хеширование	492
Коллизии	494
Открытая адресация	495
Линейное пробирование	495
Реализация хеш-таблицы с линейным пробированием на языке Java	500
Квадратичное пробирование	507
Двойное хеширование	509
Метод цепочек	517
Приложение HashChain Workshop	517
Реализация метода цепочек на языке Java	520
Хеш-функции	525
Быстрые вычисления	526
Случайные ключи	526
Неслучайные ключи	526
Хеширование строк	528
Свертка	530
Эффективность хеширования	530
Открытая адресация	530
Метод цепочек	532
Сравнение открытой адресации с методом цепочек	534
Хеширование и внешнее хранение данных	535
Таблица файловых указателей	535
Неполные блоки	535
Полные блоки	536
Итоги	537
Вопросы	538
Упражнения	540
Программные проекты	540

Глава 12. Пирамиды	542
Общие сведения	543
Приоритетные очереди, пирамиды и ADT	544
Слабая упорядоченность	544
Удаление	545
Вставка	547
Условные перестановки	548
Приложение Heap Workshop	549
Заполнение	550
Изменение приоритета	550
Удаление	550
Вставка	550
Реализация пирамиды на языке Java	550
Вставка	551
Удаление	552
Изменение ключа	553
Размер массива	554
Программа heap.java	554
Расширение массива	560
Эффективность операций с пирамидой	560
Пирамидальное дерево	560
Пирамидальная сортировка	562
Ускоренное смещение вниз	562
Сортировка «на месте»	564
Программа heapSort.java	565
Эффективность пирамидальной сортировки	569
Итоги	570
Вопросы	571
Упражнения	572
Программные проекты	572
Глава 13. Графы	574
Знакомство с графами	574
Определения	575
Немного истории	577
Представление графа в программе	578
Добавление вершин и ребер в граф	580
Класс Graph	581
Обход	582
Обход в глубину	583
Обход в ширину	592
Минимальные остовные деревья	599
Приложение GraphN Workshop	600
Реализация построения минимального остовного дерева на языке Java	600
Топологическая сортировка с направленными графами	604
Пример	605
Направленные графы	605
Топологическая сортировка	606

Приложение GraphD Workshop	607
Циклы и деревья	608
Реализация на языке Java	609
Связность в направленных графах	615
Таблица связности	615
Алгоритм Уоршелла	615
Реализация алгоритма Уоршелла	618
Итоги	618
Вопросы	619
Упражнения	620
Программные проекты	620
Глава 14. Взвешенные графы	622
Минимальное остовное дерево во взвешенных графах	622
Пример: кабельное телевидение в джунглях	622
Приложение GraphW Workshop	623
Рассылка инспекторов	624
Создание алгоритма	628
Реализация на языке Java	630
Программа mstw.java	632
Задача выбора кратчайшего пути	637
Железная дорога	638
Направленный взвешенный граф	639
Алгоритм Дейкстры	639
Агенты и поездки	639
Приложение GraphDW Workshop	644
Реализация на языке Java	648
Программа path.java	652
Поиск кратчайших путей между всеми парами вершин	656
Эффективность	658
Неразрешимые задачи	659
Обход доски ходом шахматного коня	660
Задача коммивояжера	660
Гамильтоновы циклы	661
Итоги	661
Вопросы	662
Упражнения	663
Программные проекты	664
Глава 15. Рекомендации по использованию	666
Структуры данных общего назначения	666
Скорость и алгоритмы	667
Библиотеки	668
Массивы	669
Связанные списки	669
Деревья двоичного поиска	670
Сбалансированные деревья	670
Хеш-таблицы	670
Быстродействие структур данных общего назначения	671

Специализированные структуры данных	671
Стек	672
Очередь	672
Приоритетная очередь	673
Сортировка	673
Графы	674
Внешнее хранение данных	675
Последовательное хранение	675
Индексированные файлы	676
B-деревья	676
Хеширование	676
Виртуальная память	676
Итоги	677
Приложение А. Приложения Workshop и примеры программ	678
Приложения Workshop	678
Примеры программ	679
Sun Microsystems SDK	679
Программы командной строки	679
Настройка пути	680
Запуск приложений Workshop	680
Работа с приложениями Workshop	680
Запуск примеров	681
Компилирование примеров	681
Редактирование исходного кода	682
Завершение примеров	682
Одноименные файлы классов	682
Другие системы разработки	682
Приложение Б. Литература	683
Структуры данных и алгоритмы	683
Объектно-ориентированные языки программирования	684
Объектно-ориентированное проектирование и разработка	684
Приложение В. Ответы на вопросы	686
Об авторе	694
Алфавитный указатель	695

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.

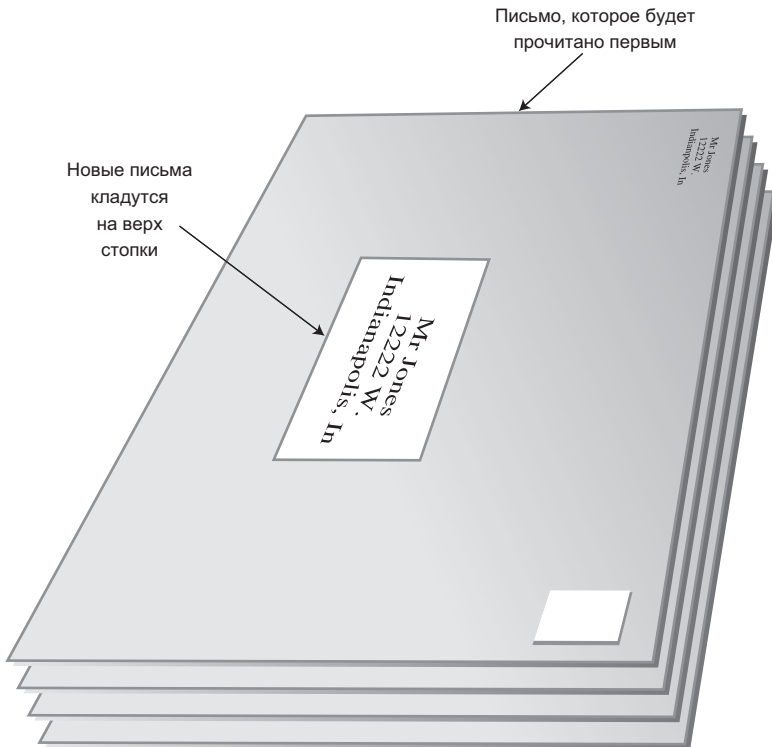


Рис. 4.1. Стопка писем

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 (А), но ваш коллега просит временно прерваться и помочь ему с другим проектом (В). В ходе работы над В к вам заходит бухгалтер, чтобы обсудить ваш отчет по командировочным расходам (С). 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 С; после завершения С возобновляется проект В, а после завершения В вы (наконец-то!) возвращаетесь к проекту А. 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).

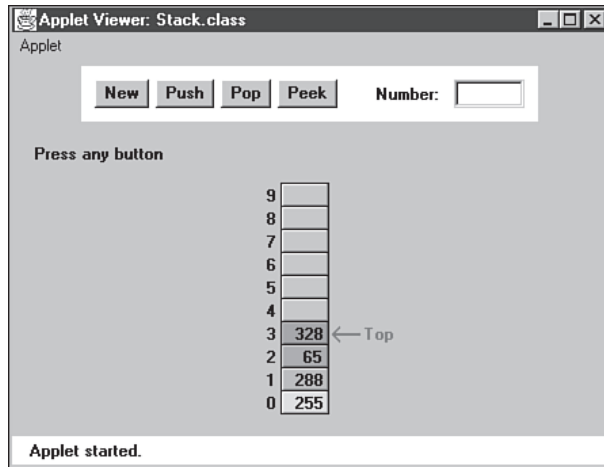


Рис. 4.2. Приложение Stack Workshop

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.

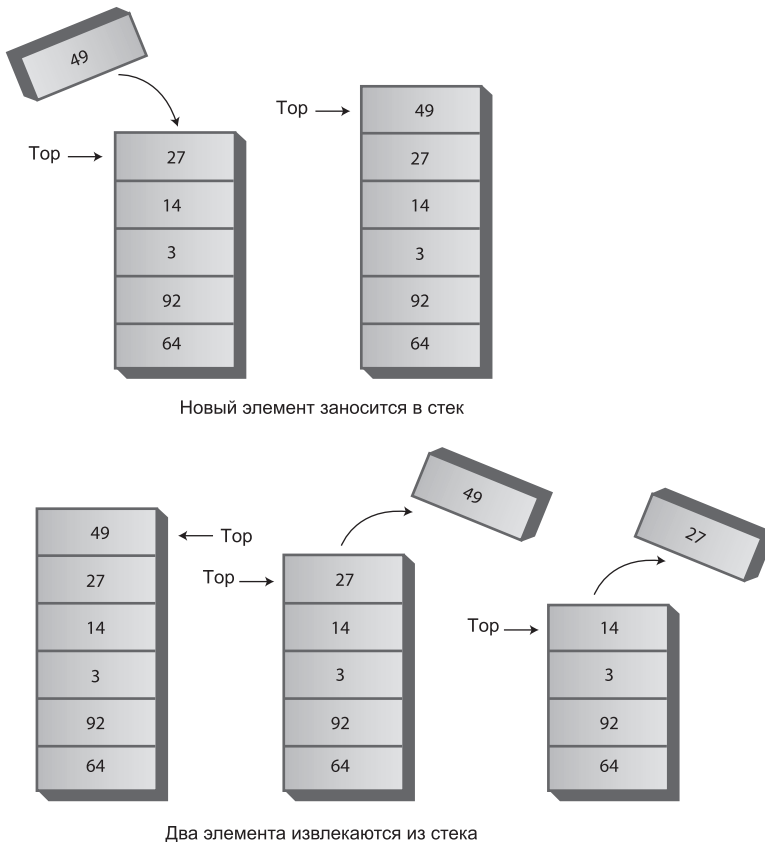


Рис. 4.3. Действие методов класса StackX

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
        }
    }
}

```