ИЗВЕСТИЯ
ПЕНЗЕНСКОГО ГОСУДАРСТВЕННОГО
ПЕДАГОГИЧЕСКОГО УНИВЕРСИТЕТА
имени В. Г. БЕЛИНСКОГО
ФИЗИКО-МАТЕМАТИЧЕСКИЕ И ТЕХНИЧЕСКИЕ НАУКИ
№ 13 (17) 2009

IZVESTIA
PENZENSKOGO GOSUDARSTVENNOGO
PEDAGOGICHESKOGO UNIVERSITETA
imeni V. G. BELINSKOGO
PHYSICAL, MATHEMATICAL AND TECHNICAL SCIENCES
№ 13 (17) 2009

УДК 004.62

РАЗНООБРАЗИЕ СТРУКТУР ДАННЫХ

© А. М. ВОЛОДИН, В. В. ДРОЖДИН
Пензенский государственный педагогический университет имени В. Г. Белинского,
кафедра прикладной математики и информатики
e-mail: drozhdin@spu-penza.ru

Володин А. М., Дрождин В. В. – Разнообразие структур данных // Известия ПГПУ им. В. Г. Белинского. 2009. № 13 (17). С. 79–83. – *Рассматриваются особенности реализации структур данных в стандартных библиотеках компонентов. Отмечаются достоинства и недостатки существующих реализаций структур данных. Приводятся особенности и разновидности коллекций данных. Делается вывод об экспоненциальном росте числа возможных реализаций структур данных в зависимости от учета новых признаков.*

Ключевые слова: структура данных, коллекция структур данных, библиотека компонентов, эффективность обработки данных

Volodin A. M., Drozhdin V. V. – The variety of data strusture // Izv. Penz. gos. pedagog. univ. im.i V. G. Belinskogo. 2009. № 13 (17). P. 79–83. – *The article deals with the peculiarities of the realization of data structure in standard components' library. The advantages and disadvantages of the existing realization of data structure are considered. The distinctive features and varieties of data collections are listed. The conclusion about exponential growth of possible number of data structure realization depending on taking into consideration new features is made.*

Keywords: data structure, collection of data structure, components' library, the effectiveness of data processing

Одна из центральных проблем в разработке программного обеспечения заключается в создании, обработке и эволюции структур данных. Структуры данных реализуются программистами при создании приложений или используются из стандартных библиотек компонентов (например, Standard Template Library, Booch Components, Java Collection Framework, Apache Common Collections, Google Collections Library и др.). Библиотеки имеют существенные преимущества, избавляя программистов от необходимости реализации большого числа рутинных операций. Они включают множество готовых хорошо спроектированных компонентов, содержащих достаточно эффективные (для различных условий) структуры данных и алгоритмы их обработки. Эти компоненты надежно протестированы (для них написаны модульные тесты) и документированы. К тому же, большинство библиотек являются свободно распространяемыми.

Однако библиотеки накладывают ряд ограничений, которые могут существенно снизить их преимущества. Они содержат множество компонентов, но разнообразие реализаций структур данных и алгоритмов их обработки не так велико. Все компоненты, как правило, имеют единственную реализацию, что не всегда эффективно в требуемых условиях функционирования системы (контексте). Для повышения универсальности использования структур данных разработчики компонентов обычно исходят из предположения равновероятного доступа к любому элементу данных. Но такое предположение часто оказывается неверным. Например, операция поиска данных может применяться с разной интенсивностью к различным элементам некоторого множества данных. В этом случае элементы коллекции целесообразно классифицировать по интенсивности использования и реализовать каждое подмножество собственной физической структурой данных. В связи с этим можно выделить следующие подмножества данных: часто изменяемые данные; условно постоянные и постоянные данные, которые преимущественно участвуют в операциях поиска данных; архивные данные, которые очень редко участвуют в операциях обработки данных и другие [1].

Среди всех операций обработки данных может существовать некоторая операция, которая применяется с высокой интенсивностью к подавляющему большинству элементов структуры. В этом случае физическую структуру данных целесообразно ориентировать на эффективное выполнение этой операции. Например, добавление данных более эффективно в неупорядоченный список, чем в упорядоченный.

С другой стороны, в процессе длительного существования структуры данных интенсивность обработки конкретных элементов данных может существенно изменяться. Поэтому для сохранения высокой

эффективности обработки данных целесообразно, чтобы физическая структура данных также изменялась динамически путем перехода отдельных элементов из одних подструктур в другие или выполнения в определенные моменты времени реструктуризации (перестройки) всей структуры данных.

В стандартных библиотеках для реализации абстрактных структур данных, как правило, используются либо хеш-таблицы, либо иерархические структуры – (сбалансированные) деревья некоторого вида. Использование данных структур определяется тем, что они могут обеспечить в среднем высокую эффективность выполнения большинства операций обработки данных.

На выбор методов физической организации структур данных дополнительное ограничение накладывает принятая в библиотеке иерархия компонентов. Поэтому изменения одних компонентов могут повлечь за собой модификации других. В большинстве существующих библиотек имеются компоненты, предоставляющие последовательный доступ ко всем элементам некоторой структуры данных, не раскрывая при этом ее внутреннего представления. Такие компоненты называются итераторами (Iterator), или курсорами (Cursor). Итераторы зависят от конкретной реализации коллекции, с которой они связаны. Поэтому, например изменение физической реализации абстрактных структур множества (Set) или словаря (Map) с хеш-таблицы на иерархическую структуру (дерево) неизбежно приведет к изменению соответствующих итераторов. Все методы итератора должны быть переопределены в соответствии с реализацией абстрактной структуры данных.

В худшем случае некоторые операции итератора могут иметь линейную временную сложность и требовать при выполнении обращения ко всем элементам структуры. Это будет существенно снижать эффективность исполнения приложений, в которых для просмотра элементов коллекции используются итераторы. Высокая вычислительная сложность [3] операций итератора, учитывающая временную и емкостную сложности, становится особенно неадекватной при формировании композиции структур, реализующих сложную структуру данных. Для быстрого доступа к данным целесообразно использовать прямые ссылки на элементы коллекции, но это приводит к снижению логической независимости данных и увеличению емкостной сложности структуры.

Структурная сложность физической структуры, реализующей абстрактную структуру данных, может быть произвольной, но она должна быть скрыта от программистов. При создании производных логических структур данных должно наследоваться, прежде всего, поведение структур данных, а не их внутренняя организация [2]. В объектно-ориентированном программировании мы вынуждены наследовать физическую реализацию, хотя подклассы могут быть представлены физически более эффективным способом. В большинстве случаев эта проблема решается путем добавления нового уровня иерархии, что неизбежно увеличивает вычислительную сложность системы.

Главное преимущество стандартных библиотек компонентов состоит в возможности многократного использования кода с минимальными доработками. Но при этом можно столкнуться с проблемой вертикальной и горизонтальной иерархии [4]. Для построения сложных компонентов эффективнее использовать композицию более мелких (вертикальная иерархия) объектов, т. к. ими проще оперировать. При этом программистам потребуется писать меньше строчек кода, что приведет к меньшему числу ошибок.

Однако по мере укрупнения компонентов вероятность их повторного использования становится ниже. Сложным компонентам свойственна более сложная и специфическая для определенного контекста логика, а также меньшая производительность. Поэтому число приложений, в которых они могут использоваться, уменьшается. Компоненты могут иметь сложные интерфейсы, плохо совместимые с прикладными программами. Добавление новой функциональности и изменение структур данных становится затруднительным, т. к. требует существенной модификации исходного кода. Чаще всего такая модификация осуществляется по принципу «повторного использования белого ящика» (white box reuse) и требует больших затрат на тестирование и отладку. Эти затраты будут пропорциональны количеству новых компонентов, добавляемых в библиотеку.

Таким образом, вертикальная иерархия приводит к сокращению числа потенциальных приложений, в которых могут успешно применяться компоненты стандартной библиотеки. Поэтому программисты вынуждены разрабатывать собственные модификации компонентов, исходя из логики приложений. Классификация компонентов на основе их особенностей называется горизонтальной иерархией [4]. Горизонтальная иерархия приводит к созданию компонентов, пригодных для более широкого круга приложений.

Рассмотрим иерархию компонентов на примере. В таблице 1 представлены абстрактные структуры данных, содержащиеся в библиотеке компонентов Г. Буча [5].

Приведенные структуры данных имеют свои разновидности и на языке C++ представляются в виде абстрактных классов. Различные варианты реализации этих структур можно рассматривать как пример горизонтальной иерархии компонентов. На практике конкретные варианты абстрактных структур реализуются, как правило, на основе механизма наследования.

Г. Буч выделяет несколько особенностей (характерных черт) коллекций, но основе которых могут быть получены конкретные разновидности. Одним из таких признаков является ограниченность по размеру (объему выделенной памяти) (boundedness). Размерность коллекции может быть статическая (фиксированная) (bounded) или динамическая (переменная) (unbounded, dynamic). В коллекциях с фиксированным размером объем выделенной памяти остается постоянным, а количество элементов в структуре может изменяться в пределах этого объема. Размер структуры определяется в момент ее создания (инициализации

Таблица 1

Абстрактные структуры данных Booch Component Library

№ п/п	Название структуры	Описание структуры
1	Коллекция (Collection)	Набор элементов, допускающий индексацию.
2	Множество (Set)	Коллекция, содержащая уникальные элементы
3	Множество с дубликатами (мультимножество, сумка) (Bag)	Коллекция, которая может содержать повторяющиеся элементы (дубликаты).
4	Список (List)	Последовательность, просмотр элементов в которой осуществляется последовательно, начиная с начала последовательности, а добавление и удаление элементов – в любое место в соответствии с заданным отношением порядка.
5	Очередь (Queue)	Последовательность элементов с дисциплиной обслуживания «первым вошел – первым вышел» (FIFO). В очереди добавление нового элемента осуществляется в конец последовательности, а выборка и удаление – с начала последовательности.
6	Стек (Stack)	Последовательность элементов с дисциплиной обслуживания «последним вошел – первым вышел» (LIFO). В стеке добавление, выборка и удаление элементов возможны только с одного конца последовательности – вершины стека.
7	Дек (Deque)	Последовательность, в которой добавление, выборка и удаление элементов возможны с любого конца последовательности
8	Цепочка (String)	Индексируемая последовательность элементов, поведение которой включает манипуляции с фрагментами последовательности (подцепочками).
9	Кольцо (Ring)	Последовательность, в которой добавление, выборка и удаление элементов могут производиться из вершины (заголовка), являющейся входом в кольцевую структуру.
10	Словарь (Map)	Коллекция, которая содержит множество пар ключ/значение.
11	Дерево (Tree)	Коллекция со строгой упорядоченностью элементов (узлов), в которой все узлы, кроме корня (являющегося входом в структуру), имеют строго одного предшественника и все узлы, кроме листьев (конечных узлов), имеют одного или несколько последователей.
12	Граф (Graph)	Коллекция узлов и дуг (вершин и ребер), которая может содержать циклы и перекрестные ссылки. В отличие от дерева, не имеет корня.

объекта). Такие структуры хорошо ориентированы на поиск данных, т.к. элементы могут индексироваться и к ним будет возможен как последовательный, так и произвольный доступ. Время поиска элементов в коллекции будет достаточно небольшим.

Реализация коллекций с динамической размерностью наиболее эффективна для структур, размер которых существенно изменяется в процессе функционирования системы, т.е. когда интенсивно выполняются операции вставки и удаления элементов. При этом память используется эффективно: выделяется и очищается по мере необходимости. Однако поиск или вставка элемента, например, в середину двусвязного списка может потребовать последовательного просмотра данных, начиная с начала последовательности.

Вторым важным признаком классификации коллекций является использование механизмов синхронизации параллельных потоков. Коллекции, которые не поддерживают синхронизацию, могут использоваться только для однопоточных программ, имеющих монопольный доступ к имеющимся ресурсам. Однако при одновременном обращении нескольких потоков к одним и тем же данным может возникнуть ситуация, когда результат программы будет зависеть от случайных факторов, таких как временное чередование исполнения операций несколькими потоками. Поэтому в многопоточных приложениях (особенно в системах реального времени) необходима синхронизация одновременных потоков управления в рамках одной системы. Различают три основных вида синхронизации потоков обработки объектов:

– последовательная (sequential) – вызывающие процессы должны координировать свои действия еще до входа в вызываемый объект, так что в любой момент времени внутри объекта находится ровно один поток управления. При наличии нескольких потоков управления не могут гарантироваться семантика и целостность объекта;

– охраняемая (guarded) – семантика и целостность объекта гарантируются при наличии нескольких потоков управления путем упорядочивания вызовов всех охраняемых операций объекта. По существу, в каждый момент времени может выполняться только одна операция над объектом, что сводит такой подход к последовательному;

– параллельная (concurrent) – семантика и целостность объекта при наличии нескольких потоков управления гарантируются тем, что операция рассматривается как атомарная.

Другими признаками классификации коллекций могут быть итерируемость (иметь или не иметь итератор), наличие сборщика мусора (управляемый

или неуправляемый сборщик мусора), способ доступа к элементам (последовательный, произвольный, индексный и т. д.), способ выделения памяти для новых элементов, работа в транзакционном или нетранзакционном режиме и т. д. Такие коллекции, как дек или очередь, можно также разделить по следующим признакам: могут ли в коллекциях удаляться элементы из позиций, отличных от начала и конца последовательности, могут ли последовательности упорядочиваться по значению некоторого поля (в этом случае очередь называется очередью с приоритетом) и др.

Все перечисленные признаки могут иметь разные сочетания для получения большего разнообразия реализаций абстрактных структур данных. Так, в библиотеке компонентов Г. Буча можно выделить 26 вариантов комбинаций этих признаков для очередей [5]. Учет новых особенностей (например, способы выделения памяти для добавляемых элементов и д.р.) приводит к геометрическому росту числа потенциальных реализаций. В результате произойдет существенный рост количества компонентов в библиотеке, которые будут отвечать потребностям сравнительно небольшого числа приложений.

Хотя библиотеки содержат достаточно небольшое число компонентов общего назначения, однако их явно недостаточно для нужд системных и прикладных программ, особенно реализующих сложную логику. Например, платформа Java 2 содержит компонент Collection Framework, который предоставляет унифицированную архитектуру для организации и манипулирования коллекциями различных типов, не вникая в детали их реализации. Это довольно большая и богатая библиотека. Она основана на конкретных реализациях нескольких интерфейсов, содержащих набор операций для различных типов коллекций. Подробная информация о Java Collections Framework содержится в [8].

Основными интерфейсами являются: коллекция (Collection), множество (Set), список (List), очередь (Queue) и словарь (Map). Интерфейс содержат набор операций обработки структуры соответствующего типа (прежде всего методы выборки, вставки, модификации и удаления элементов данных).

Collection – корневой интерфейс иерархии, позволяющий формировать группу объектов (элементов), представляемых в виде коллекции с определенными свойствами.

Set – коллекция, не содержащая повторяющиеся элементы (дубликаты). Конкретные реализации множеств могут упорядочиваться.

List – упорядоченная коллекция (последовательность), допускающая дубликаты. Предоставляет прямой (позиционный) доступ к элементам по их индексу (позиции).

Queue – коллекция, элементы которой упорядочены по принципу FIFO, LIFO или по приоритету.

Deque – коллекция, поддерживающая вставку и удаление элементов с любого конца.

Map – коллекция, которая содержит множество пар "ключ – значение", не допускающая дубликаты ключей.

Приведенные интерфейсы реализуются с помощью хеш-таблиц (Hash Table), динамически изменяемых массивов (Resizable Array), сбалансированных деревьев (Balanced Tree) (деревья поиска (Search Tree), кучи (Heap) и др.) и связных списков (Linked List). Ниже приведена таблица, на которой перечислены основные классы, реализующие интерфейсы коллекций.

Таблица 2

Реализация интерфейсов коллекций в Java Collection Framework

Интерфейсы (Interfaces)	Реализации (Implementations)				
	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue			PriorityQueue		
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Как видно из таблицы 2, библиотека содержит различные варианты реализации интерфейсов, обладающие разными свойствами и имеющие определенные преимущества и недостатки. Например, в хеш-таблицах доступ к элементам практически не зависит от размера коллекции, но в то же время сами элементы хранятся в произвольном порядке. В деревьях поиска элементы отсортированы по значению и доступ к ним пропорционален логарифму от числа содержащихся элементов.

С другой стороны класс LinkedList реализует сразу два интерфейса – List и Deque, т. е. проявляет свойства обеих коллекций.

Все коллекции реализуются на основе структур, реализованных в языках программирования, – прежде всего массивов и указателей. Классы ArrayList, ArrayDeque, HashSet, HashMap реализованы с помощью массивов, поэтому они способны предоставить произвольный доступ к элементам. LinkedList, PriorityQueue, TreeSet, TreeMap представляют собой динамические структуры данных (связные списки или деревья), формируемые с помощью указателей. Доступ к элементам в этих структурах является последовательным. Два класса LinkedHashSet и LinkedHashMap являются гибридными структурами, созданными с помощью массивов и указателей.

Таблица 2 содержит незаполненные ячейки, что свидетельствует о неполной реализации возможных методов представления и обработки абстрактных структур данных.

Библиотеки Apache Common Collections и Google Collections Library расширяют Java Collections Framework, предоставляя новые интерфейсы и реализации абстрактных структур данных (таблица 3).

Таблица 3

Коллекции Apache Common Collections и Google Collections Library

Описание коллекции	Apache Common Collections		Google Collections Library	
	Интерфейс	Реализации	Интерфейс	Реализации
Мультимножество (может содержать повторяющиеся элементы)	**Bag**	HashBag TreeBag	**MultiSet**	HashMultiSet TreeMultiSet LinkedHashMultiSet
Мультисловарь (может содержать дубликаты ключей, т. е. одному ключу соответствует несколько значений)	**MultiMap**	MiltiHashMap MultiValueMap	**MultiMap**	HashMultiMap ArrayListMultiMap TreeMultiMap LinkedListMultiMap LinkedHasMultiMap
«Инвертированный» словарь (содержит уникальные значения ключей и значений; по ключу можно найти значение и наоборот)	**BidiMap**	DualHashBidiMap DualTreeBidMap TreeBidiMap	**BiMap**	HashBiMap

Описание представленных коллекций можно найти в [6, 7].

На основе анализа таблиц 2 и 3 можно выявить правило построения представленных коллекций и предсказать появление новых. Новые интерфейсы образуются путем сочетания различных свойств и методов обработки данных, а их реализации – путем композиции реализаций более простых структур данных. Например, список (List) представляет собой упорядоченную (в порядке поступления элементов) коллекцию, которая может содержать повторяющиеся значения элементов. Если наложить запрет на дубликаты, то получим новую коллекцию – UniqueList.

Попытка разработки библиотеки компонентов, реализующей широкий круг функциональных возможностей, приводит к комбинаторному взрыву [4]. Размер и сложность такой библиотеки будут очень большими, т.к. появится множество иерархий наследования и классов-оберток (wrappers). В результате пользователи библиотеки получат программное обеспечение с невысоким быстродействием (низкой эффективностью) и низкой вероятностью повторного использования кода, поскольку библиотечные модули будут решать специфические задачи, а сложность добавления новых компонентов или модификация существующих будет очень трудоемкой.

СПИСОК ЛИТЕРАТУРЫ

1. Баканов А.Б., Дрождин В.В, Самуйлов С.В. Структуры и алгоритмы компьютерной обработки данных: Учебное пособие. – Пенза.: Пенз. гос. пед. ун-т им. В.Г. Белинского, 2007. – 80 с.
2. К. Дж. Дейт. Введение в системы баз данных. – М.: Издательский дом «Вильямс», 2008. – 1328 с.
3. Юдин Д.Б., Юдин А.Д. Математики измеряют сложность. – М.: Книжный дом “Либроком”, 2009. – 192 с.
4. Biggerstaff T. A Perspective of Generative Reuse. Microsoft Research. – Annals of Software Engineering, Volume 5, 1998. – Pages: 169 – 226.
5. Marco J., Franch X. Reengineering the Booch Component Library. – Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies. Lecture Notes In Computer Science, Volume 1845, 2000. Pages: 96 – 111.
6. Google Collections Library – http://code.google.com/p/google-collections/
7. Apache Common Collections – http://commons.apache.org/collections/
8. Sun Developer Network (SDN) – http://java.sun.com/