Дрождин В.В., Володин В.М. Анализ эффективности и эволюция структур данных. // Проблемы информатики в образовании, управлении, экономике и технике: Сб. статей IX Междунар. научно-техн. конф. – Пенза: ПДЗ, 2009. – С.33-44.

АНАЛИЗ ЭФФЕКТИВНОСТИ И ЭВОЛЮЦИЯ СТРУКТУР ДАННЫХ

В.В. Дрождин, А.М. Володин

Пензенский государственный педагогический университет им. В.Г. Белинского, г. Пенза, Россия

Рассмотрена проблема организации и обработки данных в информационных системах. Проведен анализ эффективности реализаций структур данных в Java Collection Framework. Отмечены недостатки классического подхода к организации данных. Предложены методы эволюции и повышения эффективности структур данных.

Drozhdin V.V., Volodin A.M. The analysis of effectiveness and evolution of data structure.

The problem of data organization and processing in the information system is taken up. The effectiveness of data structure realizations in Java Collection Framework is analyzed. The disadvantages of the classical approach to the data organization are pointed out. The methods of evolution and increase of effectiveness of data structure are suggested.

Эффективность обработки данных является одной из важнейших характеристик любой информационной системы (ИС). Эффективность обработки данных в ИС обеспечивают физические структуры данных (ФСД), определяющие способ размещения данных в памяти и методы доступа к ним [1]. ФСД можно сравнивать между собой по следующим основным критериям:

- логическая организация данных (ЛОД);
- способ физического хранения данных – физическая организация данных (ФОД);
- методы доступа к данным;
- эффективность операций обработки данных.

ЛОД отражает множество элементов данных структуры и отношения между ними. Например, ЛОД может быть представлена в виде упорядоченной или неупорядоченной последовательности элементов (блоков), в виде таблицы, многоуровневого индекса и т.д.

Способ физического хранения данных определяет представление и размещение элементов данных в памяти компьютера. Физическая структура данных может храниться различными способами. При этом ФОД может быть реализована в виде композиции нескольких структур, содержать значения данных сжатыми определенным методом, а также включать дополнительные (управляющие) подструктуры, способствующие повышению эффективности обработки данных. Разнообразие ФОД для одной ЛОД не влияет на логику обработки ФСД, а изменяет только эффективность обработки.

Метод доступа определяет то, каким образом можно получить доступ к одному или нескольким элементам данных. Например, элемент можно получить

по индексу, по ключу или путем выполнения итерации. В связи с этим различают последовательный, прямой, индексный и другие методы доступа [2].

Метод доступа предполагает совокупность средств и алгоритмов доступа к данным. Например, в языках программирования для доступа к элементам коллекций необходимо пользоваться специальными функциями или операциями. Наиболее простой и эффективный метод доступа к элементам предоставляет массив, позволяющий получить доступ к произвольному элементу по его индексу, преобразуемому в адрес физической памяти.

В объектно-ориентированных языках программирования, таких, как Java, для спецификации методов доступа часто используются интерфейсы. Интерфейс представляет собой множество требований, предъявляемых к соответствующему классу [3]. Интерфейс коллекции содержит сигнатуру функций (методов доступа), реализация которых осуществляется в конкретных классах, и тем самым определяет тип коллекции.

Определение эффективности операций обработки данных проводится на основе оценки временной и емкостной сложности соответствующих алгоритмов [4]. Каждая структура данных имеет свои особенности выполнения операций обработки данных и, следовательно, разные оценки сложности. Поэтому для структур существуют условия, при которых их операции будут выполняться наиболее эффективно (определенные размер и количество элементов данных, частота использования и др.). Эффективность структуры данных в целом зависит от соотношения и стационарности частот выполнения различных операций в определенном контексте.

Наиболее важными показателями эффективности структуры данных являются скорость доступа к элементам данных и эффективность использования памяти. Скорость доступа зависит от количества связей, по которым необходимо пройтись для достижения требуемого элемента. Эффективность использования памяти определяется отношением полезного объема памяти, необходимого для хранения элементов данных коллекции, к общему объему выделенной памяти.

Часто скорость доступа и эффективность использования памяти являются взаимно обратными характеристиками. Например, стандартные массивы имеют фиксированный размер и адресацию на основе смещения относительно начального элемента. Поэтому скорость доступа к произвольному элементу массива очень высока. Однако объем памяти, необходимый для хранения многомерного массива, возрастает в геометрической прогрессии при увеличении размерности массива и в арифметической – при увеличении количества элементов в одном измерении.

С другой стороны, время доступа к произвольному элементу в разреженной матрице, реализованной совокупностью связанных списков, является переменной величиной и превышает время доступа к элементу стандартного массива. Однако в разреженной матрице хранятся только элементы с ненулевыми значениями, поэтому объем памяти для ее хранения линейно зависит от количества ненулевых элементов и не зависит от количества измерений и их размерности.

Примеры разработанных структур данных можно найти в стандартных библиотеках компонентов, таких, как Standard Template Library, Java Collection Framework, Apache Common Collections, Google Collections. Рассмотрим реализацию коллекций в Java Collection Framework. Данная библиотека основана на

реализации следующих интерфейсов, определяющих тип коллекции (ЛОД и метод доступа):

- интерфейс Collection содержит основные операции, управляющие содержимым коллекции: добавление, удаление, проверка наличия элемента и др.;
- интерфейс List определяет коллекцию упорядоченных элементов, значения которых могут быть доступны по номеру позиции;
- интерфейс Set – коллекция уникальных по значению элементов;
- интерфейс Map – коллекция, элементы которой хранятся и извлекаются по ключу.

Приведенные выше интерфейсы реализуются с помощью массивов, связных списков или более сложных структур, таких, как деревья поиска, двоичные кучи или хеш-таблицы. Выбор конкретной реализации следует делать преимущественно на основе производительности (скорости доступа), так как эффективность хранения этих структур примерно одинакова, поскольку в них используется динамическое выделение памяти. Размер структур, реализованных на основе массивов, также изменяется динамически, исходя из количества содержащихся в них элементов.

Для исследования производительности различных реализаций будем использовать инфраструктуру, описанную К. Беком [5]. Тестирование будем проводить на строковом наборе данных.

В результате анализа были получены следующие экспериментальные данные. При небольшом количестве элементов (от 1 до 1000) скорость выполнения операций обработки данных примерно равна для всех коллекций. Поэтому при незначительном объеме данных целесообразно использовать наиболее простую из возможных реализаций. На рис. 1 показан график с результатами сравнения производительности различных коллекций. При количестве элементов, равном 1024, максимальное время выполнения одной операции немного превышает 40 мс.

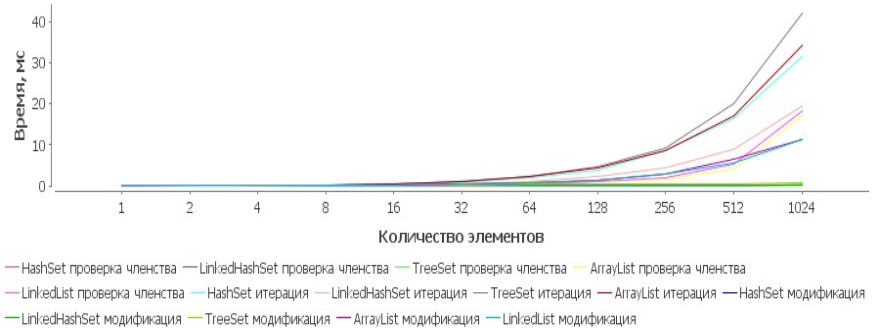
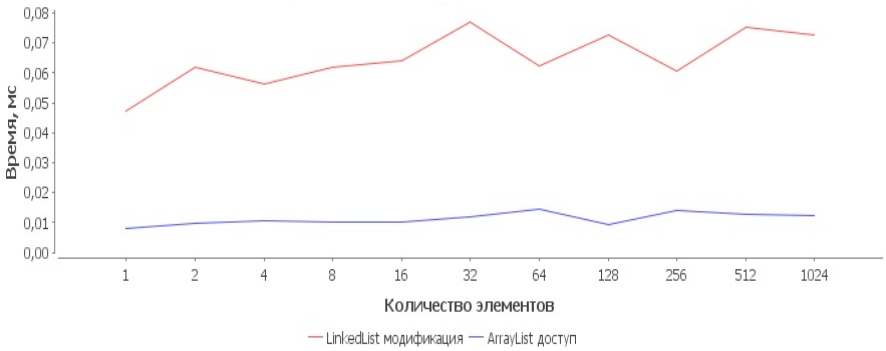
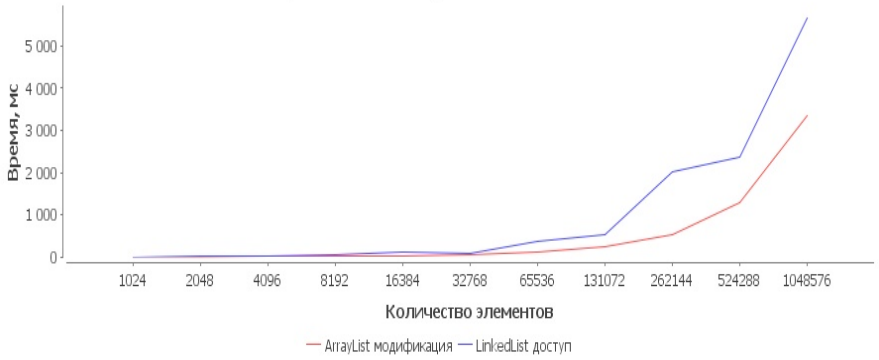


Рис. 1. Сравнение производительности коллекций на небольшом наборе данных

Скорости выполнения различных операций для одной и той же структуры данных могут существенно отличаться. Разница становится более заметной при существенном увеличении количества элементов (более одного миллиона). Так, ArrayList обеспечивает быстрый доступ к элементам и медленно добавляет и удаляет их, а LinkedList медленно предоставляет доступ к элементам, но быстро их добавляет и удаляет. Это связано с тем, что ArrayList реализован на основе массива, а LinkedList – на основе двусвязного списка. Результаты экспериментов приведены на рис. 2.



а)



б)

Рис. 2. Сравнение ArrayList и LinkedList:
а – быстрые операции;
б – медленные операции

Как видно из графиков, профили производительности рассмотренных реализаций списков являются зеркальными отображениями друг друга. Поэтому в приложениях, использующих упорядоченную (по местоположению элементов) коллекцию с преобладанием операций добавления или удаления элементов, целесообразно использовать LinkedList. Если доминируют операции получения доступа к элементам по индексам, то − ArrayList.

Множество Set реализовано в Java Collection Framework как словарь Map, у которого ключ – элемент множества данных, а значение – объект класса Object. В пакете java.util имеются три класса реализации интерфейса Map: HashMap, LinkedHashMap и TreeMap. На рис. 3 приведены оценки производительности этих коллекций.

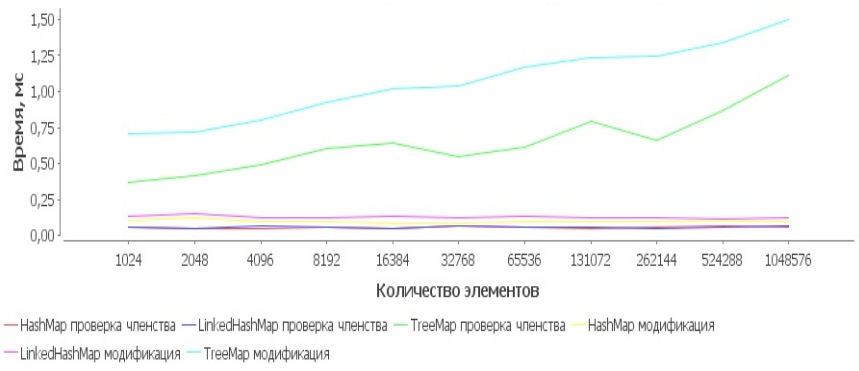


Рис. 3. Сравнение HashMap, LinkedHashMap и TreeMap

HashMap – наиболее быстрый и простой способ реализации, основанный на использовании хеш-таблицы (которая, в свою очередь, реализуется с помощью массива и односвязного списка). Однако его элементы не находятся в каком-либо порядке. LinkedHashMap сохраняет порядок элементов, в котором они были добавлены, но требует дополнительных затрат времени при выполнении операций добавления и удаления элементов. TreeMap реализуется на основе красно-черного дерева. TreeMap хранит свои элементы в отсортированном порядке, но доступ к ним и операции модификации выполняются за время, пропорциональное $\log_2 n$.

Таким образом, наиболее простые реализации коллекций имеют наиболее высокие оценки скорости доступа к элементам. Не случайно в большинстве приложений используются ArrayList и HashMap [5]. Однако данные коллекции могут существенно проигрывать другим реализациям при выполнении операций модификации (например, ArrayList) или могут появиться некоторые специфические требования (например, упорядоченность элементов, отсутствие дубликатов и др.). В этом случае необходимо использовать специализированные классы коллекций.

Однако не всегда удается легко и быстро (не прибегаю к существенному рефакторингу) заменить одну реализацию коллекции на другую. Методы доступа различных структур данных существенно отличаются друг от друга (например, интерфейсы List и Map). Кроме того, в контексте приложения может возникнуть потребность в использовании операций, не предусмотренных существующим интерфейсом. Добавление новой функциональности и изменение реализации структур данных часто является затруднительным, поскольку требует существенной модификации исходного кода и может существенно снижать эффективность обработки операций данных. Более подробно данная проблема исследована в работе [6].

Для обеспечения независимости приложений от особенностей используемых структур данных могут использоваться библиотеки, позволяющие генерировать достаточно эффективный код для реализации специфических коллекций. В качестве примера можно привести DiSTiL [7] и Predator [8]. Для спецификации коллекций в программах используется декларативный предметно-ориентированный язык программирования (domain specific language, DSL), на основе которого генерируется соответствующий код на C++.

Использование подобных библиотек позволяет обеспечить поддержку произвольных, заранее не предусмотренных (ad hock) запросов и специализацию интерфейса коллекции для решения конкретных задач. Однако это не решает всех проблем, связанных с использованием структур данных в приложениях. Во-первых, не существует эффективной реализации для декларативных языков, если они универсальны.

Во-вторых, при генерации программного кода используются различные аспекты реализации классических структур данных. Количество таких аспектов ограничено числом реализаций структур данных, изначально заложенных в библиотеке. Библиотека не может сгенерировать код на основе иных структур, которые при определенных условиях могут быть существенно более эффективны. Например, в DiSTiL и Predator используются только бинарные деревья, не-

смотря на то, что класс древовидных структур существенно шире и включает в себя 2-3 деревья, B-деревья, B^+-деревья, R-деревья и др.

В-третьих, для решения проблем, связанных с производительностью, может потребоваться ручное изменение исходного кода приложения, т.е. программист должен будет использовать некоторые иные конструкции языка DSL. При этом он может заменить спецификацию одной структуры на другую, которая, по его мнению, будет более эффективна. Таким образом, эти библиотеки не позволяют полностью отстранить человека от решения проблем оптимизации в процессе разработки и использования программного обеспечения.

Классические подходы к организации данных строятся на предположении о стационарности поведения внешней среды и равновероятном доступе к любым элементам структуры данных. Все подструктуры структуры данных организованы одинаково. Время обработки каждого элемента при такой организации является постоянной величиной. Поведение структуры с течением времени не изменяется. Не учитывается ее взаимодействие с внешней средой. Вследствие этого при увеличении количества элементов в структуре данных ее характеристики ухудшаются.

Еще большему ухудшению характеристик способствует нестационарность среды, в которой используется структура данных. Доступ к различным элементам коллекции осуществляется с разной интенсивностью. Не все данные являются одинаково востребованными и равнозначными для пользователей. В реальных условиях работы приложения высокая интенсивность одних операций обработки данных сменяется активизацией других операций. Человек не может и не должен постоянно заниматься настройкой и оптимизацией программы, заменяя одну реализацию структуры данных на другую. Ситуация осложняется тем, что результаты, полученные им в результате нагрузочного тестирования, могут существенно отличаться от тех условий, в которых реально будет функционировать программа. Поэтому найденное им решение может оказаться неудовлетворительным на практике.

Перечисленные недостатки свидетельствуют о необходимости применения принципиально иных подходов и методов к организации данных. К сожалению, следует отметить, что многие современные подходы сводятся лишь к совершенствованию приемов программирования. При этом игнорируется опыт создания искусственных систем, накопленный в других областях знания, таких, как кибернетика, синергетика, теория систем.

Сложные ИС целесообразно строить из автономных компонентов организации данных (АКОД) [9], способных самостоятельно функционировать и адаптироваться к изменениям внешней среды и внутренней организации в широких пределах. АКОД формируют и поддерживают структуры данных в рамках эволюционной модели данных [10, 11]. Для адаптации структур могут использоваться:

- параметрическая настройка,
- включение дополнительной информации [12],
- трансформация структуры.

Некоторые классические структуры данных содержат параметры, оказывающие влияние на логику обработки структуры. Параметрами являются до-

пустимый размер одного элемента данных, количество элементов (кардинальность) в подструктуре (например, в узле m-арного дерева), объем физической памяти в области переполнения (для B-деревьев) и т.д.

Для хеш-таблиц, элементы которых хранятся в массиве ячеек, одним из наиболее важных параметров является коэффициент заполнения – отношение числа заполненных ячеек к общему числу ячеек. Если значение коэффициента заполнения становится равным некоторой константе C_{max} ($0,5 < C_{max} \leq 1$), то хеш-таблицу необходимо расширить, выделив новый массив с большим числом ячеек. Если значение коэффициента заполнения достигает нижней оценки C_{min} ($0 < C_{min} < 0,5$), хеш-таблицу необходимо сжать для более эффективного использования памяти. По такому принципу работают HashMap и HashSet. Однако операции расширения и сжатия хеш-таблицы выполняются вместе с обработкой данных, что снижает быстродействие выполнения операций.

Для того чтобы приспособиться к изменениям внешней среды, часто бывает достаточно расширить существующую структуру данных путем сохранения и использования в ней дополнительной информации. Дополнительная информация может быть представлена следующими элементами:

- признаки, представляющие информацию об определенных свойствах (аспектах) всей структуры в целом, ее отдельных подструктур и элементов данных;
- параметры, позволяющие настраивать структуру данных на требуемое использование и обработку.

Признак – это некоторое свойство, которым обладает или не обладает структура данных или некоторая ее часть. Примерами таких свойств являются наличие или отсутствие повторяющихся элементов данных, количество элементов в подструктуре, способ упорядочения, диапазон значений и др.

Признаки позволяют выполнять определенные операции быстрее, а также реализовать дополнительные операции. Например, в [12] описано дерево порядковой статистики, являющееся расширением красно-черного дерева. Помимо обычных полей (элемент данных, цвет, ссылка на левое и правое поддеревья), в каждом узле дерева имеется поле, в котором хранится размер поддерева (левого и правого, включая данный узел). С помощью этого признака становится возможным быстро найти значение с i-м порядковым номером или определить порядковый номер требуемого элемента в упорядоченном множестве. При этом доказано, что поддержание дополнительной информации в актуальном состоянии в каждом узле дерева порядковой статистики не влияет на асимптотические оценки операций вставки и удаления элементов. Однако для других структур может оказаться не всегда возможным эффективно (без ухудшения асимптотических оценок) поддерживать дополнительную информацию в актуальном состоянии. Кроме того, для хранения этой информации требуется дополнительная память.

Параметры – это изменяемые элементы структуры, выполняющие роль настроек, которые после назначения становятся ограничениями. Дополнительно включаемыми параметрами могут быть допустимое (максимальное и минимальное) количество элементов в структуре, объем выделенной физической памяти, логическое ограничение (предикат) на значение элемента, допустимость

дубликатов, любые сведения о неоднородности подструктур и элементов данных. Изменяя значения соответствующих параметров, можно влиять на производительность, компактность и другие характеристики структуры данных.

Элементы, дополнительно включаемые в структуру данных с целью изменения существующих или придания структуре новых свойств, называются модификаторами. В качестве модификаторов могут выступать признаки, параметры, дополнительные элементы данных (ключевые элементы, дубликаты и др.), указатели на некоторые выделенные элементы (например, на наиболее часто используемые или элементы с определенными свойствами), специализированные методы доступа и обработки данных.

Модификаторы позволяют повысить надежность и эффективность обработки структуры данных, не нарушая логики ее организации и функционирования.

Процесс адаптации структуры данных к изменениям внешней среды на основе модификаторов можно разбить на следующие этапы:

1. Определение модификаторов, добавляемых в базовую структуру данных.
2. Оценка асимптотических характеристик обработки структуры данных с использованием модификаторов.
3. Модификация существующих алгоритмов обработки данных для поддержки дополнительной информации в актуальном состоянии.
4. Настройка параметров структуры на оптимальное использование.
5. Формирование новых специализированных методов доступа и обработки данных.

Процесс адаптации структуры данных может повторяться многократно. При этом могут включаться новые модификаторы и удаляться существующие, не отвечающие требованиям среды. Однако с течением времени количество модификаторов увеличивается, структура данных все больше отклоняется от базовой и ее свойства ухудшаются. В этом случае необходимо выполнить трансформацию к структуре данных другого типа. Для этого на основе предыстории и прогноза использования данных необходимо определить структуру данных, максимально полно удовлетворяющую всем требованиям.

Текущая реорганизация структуры данных не должна приводить к останову программы (невыполнению основных функций). Поэтому небольшие преобразования структуры данных могут осуществляться в реальном времени, т.е. в течение относительно небольших перерывов между выполнением операций обработки данных.

Более сложную реорганизацию структур данных целесообразно осуществлять во время «бездействия системы», когда не выполняются запросы на обработку данных либо их поступление маловероятно.

Таким образом, проведенный анализ используемых структур данных и предложенные методы адаптации позволяют разрабатывать и поддерживать структуры данных, обеспечивающие высокую надежность и эффективность обработки данных в течение всего времени существования системы.

Библиографический список

1. Дрождин В.В., Володин А.М. Методы физической организации данных, поддерживаемые существующими системами управления данных // Известия ПГПУ им. В.Г. Белинского. Физико-математические и технические науки. – Пенза : ПГПУ, 2008. – № 8 (12) – С. 103 – 106.

2. Мартин Дж. Организация баз данных в вычислительных системах. – М. : Мир, 1980. – 662 с.

3. Хорстман К., Корнелл Г. Java 2. Библиотека профессионала. Т. 1. Основы. – М.: Вильямс, 2008. – 816 с.

4. Юдин Д.Б., Юдин А.Д. Математики измеряют сложность. – М. : Книжный дом «Либроком», 2009. – 192 с.

5. Бек К. Шаблоны реализации корпоративных приложений. – М. : Вильямс, 2008. – 176 с.

6. Дрождин В.В, Володин А.М. Разнообразие структур данных // Известия ПГПУ имени В.Г. Белинского. Физико-математические и технические науки. – Пенза : ПГПУ, 2009.

7. Yannis Smaragdakis, Don Batory. DiSTiL: a transformation library for data structures. – Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997. Pages: 20.

8. Marty Sirkin, Don Batory, Vivek Singhal. Software Components in a Data Structure Precompiler. – International Conference on Software Engineering. Proceedings of the 15th international conference on Software Engineering table of contents. Baltimore, Maryland, United States, 1993. Pages: 437 – 446.

9. Дрождин В.В, Володин А.М. Автономный компонент организации данных // Проблемы информатики в образовании, управлении, экономике и технике: сб. статей VIII Всерос. науч.-техн. конф. – Пенза : ПГПУ, 2008. – С. 7 – 14.

10. Дрождин В.В. Открытость структур в эволюционной модели данных. // Программные продукты и системы. – № 2. – Тверь : НТП «Фактор», 2009. – С. 135 – 137.

11. Дрождин В.В. Системный подход к построению модели данных эволюционных баз данных // Программные продукты и системы. – № 3. – Тверь : НТП «Фактор», 2007. – С. 52 – 55.

12. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ. – М. : Вильямс, 2007. – 1296 с.