doi: https://doi.org/10.15407/emodel.41.06.107
УДК 004.422.639

А.Н. Примушко
Национальный технический университет Украины
«Киевский политехнический институт им. Игоря Сикорского»
(Украина, 03056, Киев, пр-т Победы, 37,
e-mail: arsentiy.prymushko@gmail.com)

Исследование характеристик структуры данных Vector в языке программирования Scala

Представлен обзор принципов и подходов, позволяющих ознакомиться со структурой данных Vector, используемой в языке программирования Scala. Рассмотрено префиксное дерево как структура данных, лежащая в основе структуры Vector. Проанализировано понятие эффективно константного времени выполнения операций над структурой Vector. Приведены практические данные об эффективности структуры Vector.

К л ю ч е в ы е с л о в а: вектор, префиксное дерево, сложность алгоритма, асимптотическая сложность, память.

В повседневной жизни человек сталкивается с различными задачами и для их максимально эффективного решения выбирает наиболее качественные подходы. Поэтому у любого рационального существа на нашей планете выбор ограничен определенной совокупностью аргументов относительно какого-либо объекта. Программист не является исключением.

Для того что бы эффективно решить задачу, программисту необходимо понимать ее суть и знать специфику инструмента, который может быть потенциально использован для достижения конечного результата. Будем считать это правильным подходом, но сначала определим, что такое Trie.

Trie — это префиксное дерево [1], т.е. структура данных (рис. 1), позволяющая хранить ассоциативный массив[1]. Ключами префиксного

[1] Абстрактный тип данных, позволяющий хранить пары вида ключ-значение.

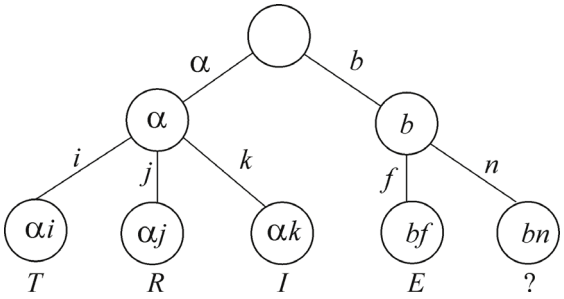

Рис. 1

дерева являются строки. Некоторые узлы могут быть выделены и это свидетельствует о том, что строку-ключ можно прочитать от корня до некоторого узла, единственного для этой строки. Ключ корня дерева — пустая строка.

В отличие от бинарных деревьев поиска [2] ключ, идентифицирующий конкретный узел дерева, не явно хранится в данном узле, а определяется положением данного узла в дереве. Для того чтобы получить ключ, следует пройти по соответствующим ребрам от корня до нужного узла. Одним из важных показателей, характерным для префиксного дерева, является коэффициент ветвления[2].

Существуют три основные операции, проводимые над префиксным деревом: проверка наличия ключа в дереве (find), удаление ключа из дерева (delete), вставка нового ключа в дерево (insert). Каждая операция реализуется с помощью спуска по дереву из корня. Эффективность такого спуска зависит от организации узлов. Существует три подхода организации узлов [1].

Для более эффективного использования памяти вместо обычного префиксного дерева можно использовать сжатое префиксное дерево [3] (базисное дерево), где узел A, — единственный потомок узла B, сливается с узлом B. Временная сложность операций поиска, добавления и удаления элемента из базисного дерева оценивается как $O(k)$, где k — длина обрабатываемого элемента. Время работы не зависит от числа элементов в дереве.

Пример реализации префиксного дерева. Представим, что машина может принимать три вида инструкций: A, B, C. Существуют различные

[2] Количество прямых потомков в каждом узле. Если это значение не одинаково для всех узлов, может быть вычислен средний коэффициент ветвления.

комбинации этих инструкций и их необходимо где-то хранить. Для этого будем использовать префиксное дерево с узлами, которые, по сути, являются массивами из трех элементов, так как есть только три возможных инструкции. Сохраним в этом дереве комбинацию *ACB*. Для хранения первой инструкции создадим новый массив из трех элементов и в нем один из элементов обозначим *A*. Получим результат, представленный на рис. 2.

Далее, необходимо сохранить вторую инструкцию. Для этого будем использовать новый узел, а значит, и новый массив из трех элементов. Связью между первым и вторым массивами будет ребро, которое обозначим *C*. В результате получим структуру, показанную на рис. 3.

В соответствии с предыдущей методикой добавим третий массив и третью инструкцию *B*. Конечный результат представлен на рис. 4.

Как поступить, если возникла необходимость в этом же дереве хранить комбинацию *ACA* ? Ведь *C* уже занято. Тогда используем ту же позицию *C* как ссылку на новую инструкцию *A*. Таким образом, получим дерево, изображенное на рис. 5.

При рассмотрении всех возможных комбинаций инструкций каждый узел будет содержать массив из трех элементов, которые фактически являются ссылками на другие массивы из трех элементов-ссылок.

Scala Vector. Vector [4—6] является одной из наиболее используемых коллекций в языке Scala, предоставляя доступ к любому элементу последовательности за эффективно константное время. В этой структуре операции *eC* занимают эффективно константное время, и это может зависеть от максимальной длины структуры и расределения хеш ключей. Построение и модификация структуры Vector происходит так же, как и любой другой последовательности в Scala.

Пример построения структуры данных Vector:

$$val\ vector: Vector[Int] =$$
$$= scala.collection.imutable.Vector.empty :+ 1 :+ 2.$$

Структура Vector сохраняет хороший баланс между быстрым случайным доступом к элементам и быстрыми случайными функциональными операциями и поэтому является типовой реализацией немутабельной индексированной последовательности.

Структура данных Vector реализуется посредством префиксного дерева с коэффициентом ветвления $n = 32$. Каждый узел в дереве содержит

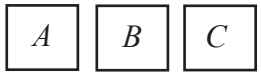

Рис. 2

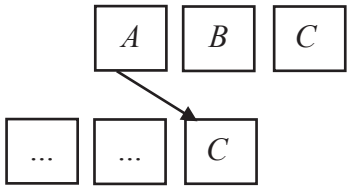

Рис. 3

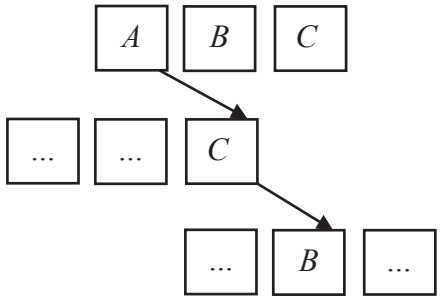

Рис.4

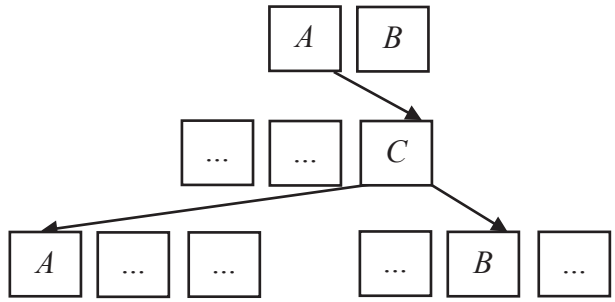

Рис. 5

либо 32 элемента либо 32 ссылки на последующие узлы. Если структура хранит не более 32-х элементов, то она может быть представлена одним узлом или полным ветвлением[3] в случае, если она хранит не более 1024-х элементов.

Структура Vector может хранить не более 2^{31} элементов. Это обусловлено максимальным размером целого числа. Для того что бы найти *i*-й элемент структуры Vector, надо пройти каждый узел в глубину дерева. Для перехода из корня дерева до нужного элемента при общем числе элементов $h < 2^{15}$ необходимо совершить два прыжка, при $h < 2^{20}$ потребуется три прыжка, при $h < 2^{25}$ — четыре прыжка, при $h < 2^{30}$ — пять прыжков, при $h < 2^{31}$ — шесть прыжков.

Эффективно константное время. При изучении коллекций в Scala можно сформировать ошибочное мнение о том, что эффективно константно

[3] Каждому элементу в узле соответствует новый узел с таким же количеством элементов. В узле всего 32 возможных элемента и каждому из них соответствует новый узел, поэтому таких элементов 32×32 = 1024.

n	Память, используемая для построения элементов n, байт		Время посторения структуры данных, мс		Время индексированного поиска, мс		Время выполнения операции конкатенации, мс	
	Vector	Array	Vector	List	Vector	Array	Vector	Array
0	56	16	2	1	0	0	5	80
1	216	40	15	4	1	1	44	82
4	264	96	99	12	4	1	185	84
16	456	336	415	69	15	1	325	89
1 048 576	21 648 072	20 971 533	131 150 000	9 100 000	4 190 000		14 470 000	1 590 000

ное время это то же самое, что и константное время или, что еще хуже, принять справедливым неравенство $eC < C$. Но это будет большим заблуждением по следующей причине.

Все операции в структуре Vector занимают время $O(\log_{32}(n))$. Тем не менее, операции над первым элементом (head) и над остальными (tail) более медленные, чем при использовании коллекции List. При сравнении быстродействия операций считывания элемента структурами Vector и Array установлено преимущество последней коллекции. В общем случае это зависит от максимального размера коллекции Vector и от распределения хеш ключей. Однако при параллельно распределяемых алгоритмах Vector лучше, чем List, если учитывать возможность управлять data locality [4].

Как известно, каждый узел в дереве состоит из 32-размерного массива. Манипуляции такой структурой значительно эффективнее, чем аналогичные операции с бинарными деревьями. Любая операция в структуре Vector не занимает более шести шагов. На основании этого формируется понятие эффективного константного времени. Вычислительная сложность операций [7, 8] не самая эффективная у структуры Vector, но она настолько мала и варьируется в таких малых пределах, что можно считать ее эффективно константной.

Теперь рассмотрим, как в структуре Vector используется память, и сравним некоторые операции из различных коллекций, используя тесты [9].

Использование памяти и быстродействие операций. Рассмотрим, сколько времени и памяти (в байтах) потребуется для Vector и Array при количестве элементов n = 0; 1; 16; 1048576 (см. таблицу). Как видно из

[4] Подразумевается перемещение вычислений вместо перемещений данных для сохранения пропускной способности. Проще переместить 120 Mb условных вычислений куда-либо чем 120 Gb данных. Такой подход минимизирует нагрузку на сеть и повышает производительность системы.

таблицы, Vector при малом количестве элементов уступает в эффективности использования памяти Array. При n = 1 048 576 разница в занимаемой памяти составляет всего лишь 0,676536 Mb. Но для критически нагруженных систем, когда разработчики борются за каждую миллисекунду, эта мизерная цифра может сыграть колоссальную роль. Это позволяет сделать вывод о том, что структуры с малым количеством элементов неэффективно расходуют память.

В таблице приведено время, необходимое для построения структуры данных по одному элементу. Как видим, Vector в 15 раз медленнее, чем List. В таблице также приведены показатели, полученные для операций индексированного поиска структурами Vector и List. Как видим, Vector значительно медленнее Array.

Было проведено сравнение времени, использованного на операцию конкатенации равноразмерных частей векторов. Из таблицы видно, что и в этом случае Vector значительно медленнее Array.

Выводы

В результате исследований установлено, что Vector-коллекция менее эффективна, чем коллекции List и Array [10]. При этом следует учесть, что эффективно константное время с определенными потерями будет большим, чем быстрое линейное время до тех пор, пока размер данных не будет достаточно большим, т.е. чем больше данных, тем эффективнее будет Vector. Использование более эффективных структур данных может сократить использование памяти и уменьшить финансовые издержки.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Wikipedia.* Префиксное дерево. [Электронный ресурс] Режим доступа: https://en.wikipedia.org/wiki/Tri. Дата обращения: 27.10.19.
2. *Wikipedia.* Двоичное дерево поиска. [Электронный ресурс] Режим доступа: https://en.wikipedia.org/wiki/Binary_search_tree. Дата обращения: 27.10.19.
3. *Wikipedia.* Базисное дерево. [Электронный ресурс] Режим доступа: https://en.wikipedia.org/ wiki/Radix_tree. Дата обращения: 27.10.19.
4. *Haoyi's Programming.* How does a Scala Vector work? [Электронный ресурс] Режим доступа: http://www.lihaoyi.com/post/ScalaVectoroperationsarent EffectivelyConstant-time.html#how-does-a-scala-vector-work. Дата обращения: 27.10.19.
5. *Scala-lang.* Concrete immutable collection classes. [Электронный ресурс] Режим доступа: https://docs.scala-lang.org/overviews/collections/concrete-immutable-collection-classes.html#vectors. Дата обращения: 27.10.19.
6. *47deg.* Adventures with Scala Collections. [Электронный ресурс] Режим доступа: https://www.47deg.com/blog/adventures-with-scala-collections/#vector-law-abiding-16. Дата обращения: 27.10.19.
7. *Wikipedia.* Bio O notation. [Электронный ресурс] Режим доступа: https://en.wikipedia.org/ wiki/Big_O_notation#Multiplication_by_a_constant. Дата обращения: 27.10.19.

8. *Haoyi's Programming.* Scala collections benchmarking. [Электронный ресурс] Режим доступа: http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html#vectors-are-ok. Дата обращения: 27.10.19.
9. *Scala-lang.* Performance characteristics. [Электронный ресурс] Режим доступа: https://docs.scala-lang.org/overviews/collections/performance-characteristics.html. Дата обращения: 27.10.19.
10. *Github:* Benchmarks. [Электронный ресурс] Режим доступа: https://japgolly.github.io/scalajs-benchmark/res/scala213-full.html. Дата обращения: 27.10.19.

Получена 28.10.19

REFERENCES

1. “Trie”. Available at: https://en.wikipedia.org/wiki/Tri. Accessed October 27, 2019.
2. Binary search tree. https://en.wikipedia.org/wiki/Binary_search_tree. Accessed October 27, 2019.
3. “Radix tree”, available at: https://en.wikipedia.org/wiki/Radix_tree. Accessed October 27, 2019.
4. “How does a Scala Vector work?”. Available at: http://www.lihaoyi.com/post/ ScalaVectoroperationsarentEffectivelyConstanttime.html#how-does-a-scala-vector-work. Accessed October 27, 2019.
5. “Concrete immutable collection classes”. Available at: https://docs.scala-lang.org/ overviews/ collections/concrete-immutable-collection-classes.html#vectors. Accessed October 27, 2019.
6. “Adventures with Scala Collections” Available at: https://www.47deg.com/blog/ adventures-with-scala-collections/#vector-law-abiding-16. Accessed October 27, 2019.
7. “Bio O notation”. Available at: https://en.wikipedia.org/wiki/Big_O_notation#Multiplication_by_a_constant. Accessed October 27, 2019.
8. “Scala collections benchmarking”http://www.lihaoyi.com/post/BenchmarkingScala Collections.html#vectors-are-ok. Accessed October 27, 2019.
9. “Performance characteristics”. Available at: https://docs.scala-lang.org/overviews/ collections/performance-characteristics.html. Accessed October 27, 2019.
10. “Benchmarks”. Available at: https://japgolly.github.io/scalajs-benchmark/res/scala213- full.html. Accessed October 27, 2019.

Received 28.10.19

А.Н. Примушко

ДОСЛІДЖЕННЯ ХАРАКТЕРИСТИК СТРУКТУРИ ДАНИХ VECTOR У МОВІ ПРОГРАМУВАННЯ SCALA

Подано огляд принципів і підходів, які дозволяють ознайомитися зі структурою даних Vector у мові програмування Scala. Розглянуто префіксне дерево як структура даних, що лежить в основі структури Vector. Проаналізовано поняття ефективно константного часу виконання операцій над структурою Vector. Наведено реальні дані про ефективність структури Vector.

К л ю ч о в і с л о в а: вектор, префіксне дерево, складність алгоритму, асимптотична складність, пам’ять.

A.N. Prymushko

VECTOR DATA STRUCTURE RESEARCH
IN SCALA PROGRAMMING LANGUAGE

This article provides an overview of the principles and approaches that allow you to become familiar with such a data structure in the Scala programming language as Vector. The prefix tree is considered as the data structure underlying Vector. Understands the concept of effectively constant execution time on Vector. Real data on the effectiveness of Vector are presented.

K e y w o r d s: Vector, prefix tree, algorithm complexity, asymptotic complexity, memory.

ПРИМУШКО Арсентий Николаевич, магистр Национального технического университета Украины «Киевский политехнический институт им. Игоря Сикорского», бакалаврат которого окончил в 2019 г. Область научных исследований – искусственный интеллект, машинное обучение, искусственные нейронные сети, программирование, системы навигации и ориентации.