Хэширование для ускорения поиска в коллекциях

Время выполнения алгоритмов простого и бинарного поиска разное, но зависит от размера массива. Идея хеширования заключается в том, чтобы эту зависимость убрать. Это достигается установлением функциональной зависимости между значением элемента массива и его индексом. В этом состоит суть так называемой ***ассоциативной адресации*** (прямой адресации, Н-кодирования, хеширования – от англ. hash – мешанина, крошево). Массив, сформированный по принципу ассоциативной адресации, называется хеш-таблицей. Функция, устанавливающая связь между значением элемента и индексом элемента, называется хеш-функцией. Значение, используемое в качестве аргумента хеш-функции, будем далее называть ключом.

Таким образом, каждому ключу в хеш-таблице определено «свое» место. Поэтому при поиске какого-либо значения сразу же смотрим, есть ли оно на отведенном ему месте.

Итак, в идеальном случае предполагается, что хеш-таблица представлена с помощью массива N элементов, которые более удобно нумеровать от 0 до N–1. Кроме того, предполагается, что существует хеш-функция h(Key), ставящая в соответствие каждому ключу Key некоторое целое число i, различное для разных значений ключа и такое, что 0<=i<=N–1. Тогда достаточно разместить этот ключ Key в элементе хеш-таблицы с индексом i=h(Key), и время поиска становится постоянным. Говорят, что алгоритм поиска в хеш-таблице имеет скорость роста O(1).

К сожалению, эта идеальная схема практически не работает, и вот почему. Посмотрим, каким же критериям должна удовлетворять хеш-функция.

- Во-первых, она должна возвращать индекс элемента в диапазоне индексов хеш-таблицы. Это легко достигается, если последним действием хешфункции сделать вычисление остатка от деления на N.
- Во-вторых, желательно иметь для каждого ключа уникальный индекс. Но этого ни одна хеш-функция гарантировать не может. Причина этого очевидна: множество возможных значений ключа либо бесконечно, либо намного превосходит размер хеш-таблицы. Поэтому в любом случае одному и тому же индексу будет соответствовать некоторое подмножество значений ключа. Например, если в качестве ключа используются строковые значения, а индекс элемента вычисляется как остаток от деления суммы числовых кодов символов строки на N, то два ключа abc и cab будут иметь одинаковые значения хеш-функции, то есть

одинаковые индексы. Говорят, что два ключа Key1 и Key2, таких, что Key1 !=Key2 и h(Key1)=h(Key2), вызывают коллизию.

Существует два вида коллизий: непосредственные коллизии, соответствующие случаю h(Key1)=h(Key2), и коллизии по модулю, когда h(Key1)!=h(Key2), но h(Key1)=h(Key2)%N. Долю коллизий по модулю можно уменьшить, если в качестве размера хеш-таблицы выбирать число, не имеющее делителей меньше 20. Например, это может быть простое число. Ясно, что если таблица имеет много пустых ячеек, то коллизия разрешится быстро. Поэтому размер таблицы выбирается несколько большим, чем предполагаемое число элементов в ней. Для оценки эффективности работы с хеш-таблицей в качестве параметра используют коэффициент заполнения α=size/sizeTable, где size – число элементов в таблице. Оценки [32] показывают, что среднее число сравнений для достаточно больших size примерно равняется 1/(1-α) для неуспешного поиска или включения и (1/α)log2(1/(1-α)) – для успешного поиска. Конкретные результаты приведены в таблице 1.

Таблица 1 Среднее число обращений в хеш-таблицах

Коэффициент заполнения %	Среднее число обращений при включении или неуспешном поиске	Среднее число обращений при успешном поиске
25	1,33	1,15
50	2	1,39
75	4	1,85
90	10	2,56
95	20	3,15

Видно, что число обращений к таблице, то есть временная сложность алгоритма, зависит только от степени заполнения таблицы, а не от ее размера: в таблице, содержащей тысячи элементов, можно найти любой из них в среднем за два с половиной обращения, если только таблица заполнена не более чем на 90%.

Следует также отметить, что на эффективность поиска оказывает влияние и выбор хеш-функции. Однако в любом случае алгоритм поиска в хеш-таблице

имеет скорость роста O(1), т. е. не обнаруживает какой-либо зависимости от размера таблицы.

При возникновении коллизии выполняются действия, называемые обработкой или разрешением коллизии. Существуют различные методы разрешения коллизий. Рассмотрим некоторые из них.

Методы разрешения коллизий

1. МЕТОД ЦЕПОЧЕК

Этот метод заключается в том, что с каждым элементом таблицы связывается набор значений с одинаковым значением индекса. Для хранения такого набора удобно использовать экземпляр класса ArrayList. В этом случае операции добавления, удаления и поиска будут разбиваться на два этапа:

- − определение индекса в хеш-таблице;
- − выполнение соответствующей операции в цепочке с использованием методов класса ArrayList.

Добавить новый элемент в конец цепочки можно с помощью метода Add. Поиск в цепочке элемента с заданным значением ключа выполняется методом IndexOf. Для удаления элемента можно воспользоваться методом Remove.

2. ОТКРЫТАЯ АДРЕСАЦИЯ

Суть метода заключается в том, что при возникновении коллизии значение полученного индекса подвергается коррекции. В простейшем случае выполняется линейный поиск свободного элемента таблицы. При этом значение индекса меняется по линейному закону ind=(ind+step)% sizeTable, где ind – текущее значение индекса, step – шаг поиска, sizeTable – размер хеш-таблицы. Величина шага поиска не должна быть делителем размера таблицы во избежание зацикливания. Недостатком данного метода является эффект группировки элементов данных в смежных элементах, что делает поиск свободного элемента более длительным и менее эффективным. Проблема группировки устраняется при использовании квадратичного поиска. В этом случае шаг поиска является переменным и определяется по формуле k2, где k – номер шага. Отметим, что квадратичный поиск гарантированно завершается успешно, если хеш-таблица заполнена меньше, чем наполовину.

3. ДВОЙНОЕ ХЕШИРОВАНИЕ

При двойном хешировании используются две независимые хешфункции h1 и h2. Для поиска места в хеш-таблице по ключу Key сначала вычисляют индекс ind0 = h1(Key). Если при этом возникает коллизия, то производится повторное вычисление индекса по формуле (ind0+h2(Key))% sizeTable, затем (ind0+2*h2(Key))% sizeTable и так далее. В общем случае идет проверка последовательности ячеек (ind0+i*h2(Key))%sizeTable, где i=0,1,. . .,sizeTable.

В среднем, при грамотном выборе хеш-функций двойное хеширование обеспечивает меньшее число попыток по сравнению с линейным поиском за счет того, что вероятность совпадения значений сразу двух независимых хеш-функций ниже, чем одной

Класс Hashtable в .NET

Для работы с хеш-таблицами платформа .Net предоставляет готовое решение – класс Hashtable, определенный в пространстве имен System.Collections. Этот класс относится к категории коллекцийсловарей, т.е. коллекций, элементами которых являются пары ключ-значение. Как ключ, так и значение могут быть данными любого типа – целыми числами, строками, объектами и т. д. Самыми важными качествами хорошего словаря являются простота добавления новых записей и быстрота получения значений. Некоторые словари организованы так, что записи добавляются в них очень быстро; другие настроены на быстрое извлечение информации. Одним из примеров словаря является хеш-таблица.

Класс Hashtable имеет встроенную хеш-функцию, обращение к которой происходит через вызов метода GetHashCode(Key). Для разрешения коллизий используется метод цепочек. Для хеш-таблицы класса Hashtable введено понятие коэффициента загрузки (load factor), который определяет максимальное значение отношения количества хранящихся в ней элементов данных к длине таблицы. При этом следует иметь в виду, что с каждым занятым элементом таблицы связана цепочка, содержащая в общем случае несколько элементов данных. Чем меньше коэффициент загрузки, тем выше производительность операций в хеш-таблице, но тем больше расходуется памяти. По умолчанию коэффициент загрузки равен 1.0, что, по утверждению фирмы Microsoft, обеспечивает оптимальный баланс между скоростью и размером. Однако программист, создающий экземпляр класса Hashtable, может задать любой коэффициент загрузки.

По мере добавления новых записей в объект Hashtable фактическая загрузка увеличивается, пока она не сравняется с коэффициентом, указанным во время создания объекта. Достигнув заданного коэффициента, объект Hashtable автоматически увеличивает длину таблицы до наименьшего простого числа, большего, чем удвоенная текущая длина.

Конструктор класса имеет несколько перегрузок, позволяющих, в частности, создавать хеш-таблицы с размером по умолчанию, с указанным размером, с указанными размером и коэффициентом загрузки и т. д. Класс Hashtable имеет свойство-индексатор, что позволяет использовать для его

экземпляров нотацию массивов с ключами в качестве индексов. В таблице 2.2 представлены основные свойства и методы класса Hashtable.

Таблица 2.2 Основные свойства и методы класса Hashtable

Метод или свойство	**Описание**
Count	Открытое свойство, позволяющее узнать текущее число элементов
Keys	Открытое свойство, возвращающее коллекцию ключей
Values	Открытое свойство, возвращающее коллекцию значений
Add()	Добавляет запись с указанной парой ключзначение
Clear()	Удаляет все элементы из объекта Hashtable
ContainsKey()	Выясняет, содержит ли объект Hashtable указанный ключ
ContainsValue()	Выясняет, имеется ли в объекте Hashtable указанное значение
GetHashCode()	Возвращает индекс в хеш-таблице по заданному ключу
Remove()	Удаляет запись с указанным ключом

В листинге ниже приведен пример использования объекта класса Hashtable. Отметим, что в данном случае использование метода ToString() при выводе результата является избыточным и сохранено для универсальности кода.

```
using System.Collections;

Hashtable hash = new Hashtable(29);
hash.Add("Имя, фамилия", "Иван Иванов");
hash.Add("Возраст", 18);
hash.Add("Группа", "2пк1");
hash.Add("Успеваемость", 4.5);

hash["Пол"] = "мужской";
foreach (Object key in hash.Keys)
    Console.WriteLine(key.ToString() + ":\t" + hash[key].ToString());
```