Всестороннее изучение структур данных с использованием C# 2.0

Часть II: Очередь, стек и хэш-таблица

Эта статья, вторая в серии из шести статей о структурах данных в .NET Framework, рассматривает три из наиболее часто исследуемых структур данных: очередь (Queue), стек (Stack), и хэш-таблицу (Hashtable). Как мы увидим, очередь и стек являются специализированными Списками (Lists), обеспечивающими хранение любого количества объектов, но ограничивающими порядок в котором можно доступаться к этим объектам. Хэш-таблица (Hashtable) обеспечивает абстракцию подобную массиву с большей гибкостью в индексировании. В то время как массив требует, чтобы его элементы были индексированы порядковыми значениями, хэш-таблица (Hashtables) позволяет индексировать элементы любым типом индексов, таким как, например, строки символов.

1. Введение

В первой части «Всестороннего изучения структур данных» мы рассмотрели что такое структуры данных, как оценить их производительность и как эта производительность влияет на выбор используемой структуры данных в определенном алгоритме. В дополнение к обзору основ структур данных и их анализу мы также рассмотрели наиболее часто используемую структуру данных – массив.

Массив хранит набор однородных элементов индексированных порядковыми значениями. Действительное содержимое массива хранится в виде непрерывного блока в памяти, обеспечивая тем самым быстрый доступ для чтения или записи элемента. В дополнение к стандартному массиву .NET Framework Base Class Library предлагает класс Список (**List**). Также как и массив, Список представляет собой набор однородных данных. При использовании Списка (**List**) вам нет необходимости беспокоится об изменении размеров или ограничениях емкости; Список также предлагает множество методов для поиска, сортировки и изменения данных в Списке. Как это уже обсуждалось в предыдущей статье класс Список (**List**) использует параметризованные типы (Generics) для предоставления типобезопасной, повторно используемой структуры набора данных.

Для начала мы рассмотрим Очередь (Queue) и Стек (Stack), затем мы постараемся максимально досконально изучить такую структуру данных как Хеш таблица (Hashtable). Хэш-таблица, иногда называемая ассоциативным массивом, хранит набор элементов, но индексирует эти элементы произвольным объектом (таким, как, например, строка), в противоположность целочисленному порядковому индексу.

2. Обеспечение обработки задач с организацией «Первым пришел – Первым обслужен»

Если вы создаете любой вид компьютерного сервиса – то есть компьютерную программу, которая может получать множество запросов из нескольких источников, для выполнения

какой-либо задачи – то одна из частей проблемы, при построении сервиса, состоит в принятии решения – в каком порядке будут обрабатываться поступающие запросы. Используются два наиболее общих подхода:

- Первым пришел, первым обслужен
- Приоритетное обслуживание

Первым пришел, первым обслужен – это задача обслуживания запросов, которую вы найдете в ближайшем продуктовом магазине, банке, и в ГАИ при выдаче водительских удостоверений. Те, кто ждут обслуживания, выстраиваются в очередь. Люди, стоящие впереди вас, будут обслужены до вас, а люди стоящие за вами будут обслужены после вас. Обработка, основанная на приоритетах, обслуживает тех, кто имеет приоритет выше перед теми, кто имеет приоритет ниже. Например, отделение скорой помощи в больнице использует эту стратегию, чтобы выбрать кого-нибудь для помощи с потенциально более фатальным повреждением перед тем, как помочь кому-то с менее угрожающим повреждением, вне зависимости от того, кто прибыл первым.

Представьте, что вам нужно построить компьютерный сервис и вы хотите обрабатывать поступающие запросы в порядке их поступления. Так как большое количество новых запросов может появиться быстрее, чем вы успеете их обработать, вам понадобиться разместить запросы в некоторого вида буфер так, чтобы сохранить порядок в котором они поступали.

Один из вариантов состоит в использовании Списка (List) и целочисленной переменной с именем `nextJobPos` для указания позиции следующего задания подлежащего выполнению. Когда приходит новый запрос, то просто используйте метод `Add()` Списка для того, чтобы добавить его в конец. Как только вы будете готовы обработать следующее задание из буфера, извлеките задание в позиции `nextJobPos` Списка и нарастите значение переменной `nextJobPos`. Следующая простая программа иллюстрирует этот алгоритм:

```
public class JobProcessing
{
   private static List<string> jobs = new List<string>(16);
   private static int nextJobPos = 0;

   public static void AddJob(string jobName)
   {
      jobs.Add(jobName);
   }

   public static string GetNextJob()
   {
      if (nextJobPos > jobs.Count - 1)
         return "NO JOBS IN BUFFER";
      else
      {
         string jobName = jobs[nextJobPos];
         nextJobPos++;
         return jobName;
      }
   }

   public static void Main()
   {
      AddJob("1");
      AddJob("2");
      Console.WriteLine(GetNextJob());
```

```
      AddJob("3");
Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      AddJob("4");
      AddJob("5");
      Console.WriteLine(GetNextJob());
   }
}
```

Программа породит следующий вывод:

```
1
2
3
NO JOBS IN BUFFER
NO JOBS IN BUFFER
4
```

В то время как этот подход достаточно прост и прямолинеен, он ужасно неэффективен. Для начинающих: Список будет расти не прекращая при каждом добавлении задания в буфер, даже если задание будет обслужено непосредственно сразу после его добавления в буфер. Рассмотрим случай когда каждую секунду в буфер добавляется новое задание и задание извлекается из буфера. Это означает, что каждую секунду вызывается метод `AddJob()`, который вызывает метод Списка `Add()`. Так как метод `Add()` непрерывно вызывается, то размер внутреннего массива Списка будет постоянно удваиваться в размере при каждой необходимости. После пяти минут (300 секунд) внутренний массив Списка будет иметь размер достаточный для хранения 512 элементов, даже если в нем никогда не было более одного задания в буфере в единицу времени. Эта тенденция, конечно, будет продолжаться так долго, как долго будет выполняться программа и будут поступать запросы на обслуживание.

Причина, по которой Список растет таким нелепым образом, состоит в том, что позиции в буфере, использованные под старые запросы не возвращаются обратно для повторного использования. То есть, когда первый запрос добавлен в буфер, а затем обработан, то очевидно, что первая позиция в Списке доступна для повторного использования. Рассмотрим планирование заданий представленных в предыдущем примере. После первых двух строк — `AddJob("1")` и `AddJob("2")` — Список будет выглядеть так, как это показано на Рисунке 1.

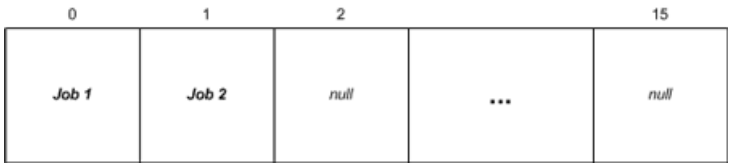

Рисунок 1. ArrayList после выполнения первых двух строк кода

Отметьте, что в данный момент в Списке присутствует 16 элементов, так как Список был инициализирован с емкостью в 16 элементов в выше приведенном коде. Далее вызывается метод `GetNextJob()`, который удаляет первое задание, приводя к результату, изображенному на Рисунке 2.

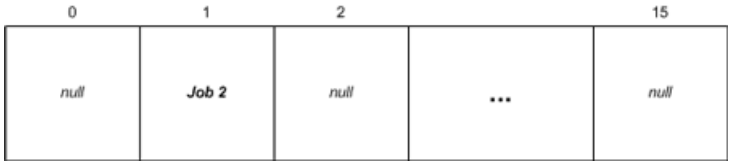

Рисунок 2. Программа после вызова метода GetNextJob()

Когда выполняется метод `AddJob("3")`, нам требуется добавить еще одно задание в буфер. Очевидно, что первый элемент Списка (индекс 0) свободен для повторного использования. Изначально могло бы иметь смысл поместить третье задание в позицию с индексом 0. Однако, такой подход необходимо исключить, если рассмотреть что бы произошло если после `AddJob("3")` мы выполнили бы `AddJob("4")`, с последующими двумя вызовами `GetNextJob()`. Если бы мы поместили третье задание в позицию с индексом 0 и затем четвертое задание в позицию с индексом 2, то мы бы получили проблему изображенную на Рисунке 3.

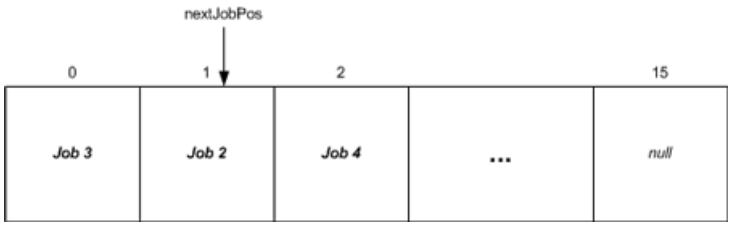

Рисунок 3. Ситуация, созданная в результате размещения задания в позицию с индексом 0

Теперь, когда будет вызван метод `GetNextJob()`, то второе задание будет удалено из буфера и `nextJobPos` будет увеличен на единицу, чтобы указывать на позицию с индексом 2. Поэтому, когда метод `GetNextJob()` будет вызван снова, то *четвертое* задание будет удалено и обслужено перед третьим заданием, нарушая таким образом порядок «первым пришел – первым обслужен», который нам требуется поддерживать.

Причина, по которой возникает эта проблема, состоит в том, что Список представляет список заданий в виде линейно упорядоченной последовательности. То есть, мы должны продолжать добавлять задания справа от старых заданий, чтобы гарантировать правильный порядок обработки. Как только мы достигаем конца Списка, он увеличивается в размере в два раза, даже если существуют неиспользуемые элементы Списка в силу вызовов метода `GetNextJob()`.

Чтобы разрешить эту проблему нам необходимо сделать список кольцевым (*circular*) или замкнутым. Кольцевой массив это такой массив, который не имеет определенного начала или конца. Более того, мы должны использовать переменные для того, чтобы запомнить начальную и конечную позиции массива. Графическое представление кольцевого массива показано на Рисунке 4.

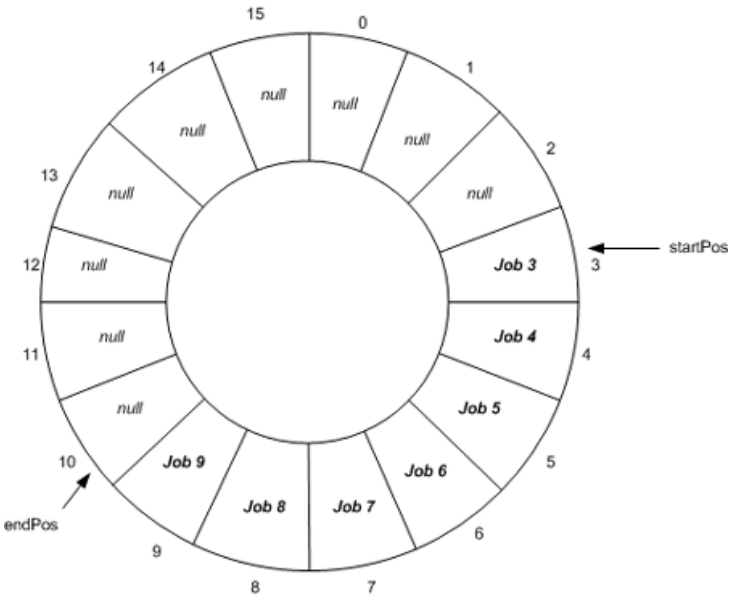

Рисунок 4. Пример кольцевого (замкнутого) массива

В кольцевом массиве метод `AddJob()` добавляет новое задание в позицию `endPos` и затем «наращивает» `endPos`. Метод `GetNextJob()` извлекает задание по индексу `startPos`, устанавливает задание по индексу `startPos` в `null`, и затем «наращивает» `startPos`. Слово «наращивает» помещено в кавычки потому что здесь инкрементирование является несколько сложнее, чем простое добавление единицы к значению переменной. Чтобы увидеть почему мы не можем просто добавить 1, рассмотрим случай когда `endPos` равно 15. Если мы нарастим `endPos` добавляя 1, то `endPos` будет равно 16. При следующем вызове `AddJob()`, будет произведена попытка доступа по индексу 16, которая приведет к исключению `IndexOutOfRangeException`.

Кроме того, когда `endPos` равно 15, мы хотим «нарастить» `endPos` сбрасывая ее в 0. Это можно сделать либо создав функцию `increment(`*`variable`*`)` которая проверяет не перешагнула ли переменная за границу массива, и если так, то сбрасывает ее в 0. В качестве альтернативы переменную можно увеличить на 1 и затем выполнить операцию взятия остатка от деления на размер массива. В таком случае код для `increment()` мог бы иметь следующий вид:

Comment [w1]: Операция деления по модулю %, используемая в виде `x % y`, вычисляет остаток от деления `x` на `y`. Остаток всегда будет величиной в диапазоне от 0 до `y - 1`.

```
int increment(int variable)
{
  return (variable + 1) % theArray.Length;
}
```

Этот подход хорошо работает, если наш буфер никогда не будет содержать более 16 элементов, но что произойдет, если мы захотим добавить новое задание в буфер, когда он уже содержит 16 заданий? Точно так же как и с методом `Add()` Списка, мы должны изменить размер циклического массива соответствующим образом, скажем, удваивая его размер.

2.1 Класс System.Collections.Generic.Queue

Функциональность, которую мы сейчас описали — добавление и удаление элементов в буфер в порядке первым пришел, первым обслужен с максимизацией использования памяти — обеспечивается стандартной структурой данных Очередь (Queue). .NET Framework Base Class Library предоставляет класс System.Collections.Generic.Queue, который использует параметризованные типы (Generics) для обеспечения типобезопасной реализации Очереди (Queue). Ввиду того, что наш предыдущий код предоставлял методы `AddJob()` и `GetNextJob()`, класс Очередь (**Queue**) предоставляет идентичную функциональность с помощью методов `Enqueue(item)` и `Dequeue()`, соответственно. За кулисами класс Очередь (**Queue**) поддерживает внутренний циклический массив и две переменных, которые служат маркерами для начала и конца циклического массива: `head` и `tail`.

Метод `Enqueue()` начинается с определения того достаточно ли места для добавления нового элемента в очередь. Если да, то он просто добавляет элемент в циклический массив по индексу `tail`, и затем «наращивает» значение `tail` используя оператор сложения по модулю, чтобы быть уверенным, что `tail` не превышает размерность массива. Если же, однако, места не достаточно, то массив увеличивается в соответствии с коэффициентом роста. Этот коэффициент роста по умолчанию имеет значение 2.0, удваивая таким образом размер внутреннего массива, но вы можете опционально задать этот коэффициент в конструкторе класса Очередь (Queue).

Метод `Dequeue()` возвращает текущий элемент по индексу head. Он также устанавливает элемент по индексу `head` в null и «наращивает» `head`. Для тех случаев, когда вы хотите просто взглянуть на элемент в начале списка, но не извлекать его, класс Очередь (**Queue**) также предоставляет метод `Peek()`.

Что очень важно понять, так это то, что Очередь (Queue), в отличие от Списка (List), не позволяет доступаться к элементам произвольно. То есть вы не можете посмотреть на третий элемент очереди без извлечения (dequeing) первых двух элементов. Однако класс Очередь (**Queue**) содержит метод `Contains()`, так что вы можете определить содержится ли данный элемент в Очереди. Есть также метод `ToArray()`, который возвращает массив, содержащий все элементы Очереди. Если вы знаете, что вам понадобится произвольный доступ к элементам, то Очередь не является подходящей структурой данных – для этого подходит Список (List). Очередь, однако, идеально подходит для тех случаев, когда вам необходим точный порядок обработки элементов в котором они поступают.

Comment [w2]: Вы могли слышать, что Очереди часто называют структурами данных с организацией FIFO. FIFO происходит от «First In, First Out» и является синонимом порядка обслуживания «первым пришедший первым будет обслужен».

3. Взгляд на структуру данных Стек: «Первым пришел – последним обслужен»

Структура данных Очередь (Queue) обеспечивает обслуживание по принципу «первым пришел – первым обслужен» путем доступа к внутреннему циклическому массиву элементов типа `object`. Очередь (Queue) предоставляет такой доступ с помощью методов `Enqueue()` и `Dequque()`. Обслуживание по принципу «первым пришел – первым обслужен» имеет множество применений в реальном мире, особенно в обслуживающих программах, таких как Веб серверы, очереди печати и другие программы, которые обслуживают большое количество входящих запросов.

Другой распространенной схемой обслуживания в компьютерных программах является дисциплина «первым пришел – *последним* обслужен». Структура данных, которая обеспечивает такую форму доступа, известна как Стек (Stack). The .NET Framework Base Class Library содержит класс Стек (**Stack**) в пространстве имен

`Sytem.Collections.Generic`. Так же как и Очередь (**Queue**), класс Стек (**Stack**) поддерживает свой внутренний массив. Класс Стек (**Stack**) предоставляет свои данные посредством двух методов: `Push(`*`item`*`)`, который добавляет переданный элемент в стек, и `Pop()`, который удаляет и возвращает элемент на вершине стека.

Стек можно графически изобразить как вертикальный набор элементов. Когда элемент погружается (pushed) в стек, то он помещается наверх всех других элементов. Выталкивание (Popping) элемента удаляет элемент с вершины стека. Следующие два рисунка (см. Рисунок 5 и Рисунок 6) графически иллюстрируют стек сперва после того как элементы 1, 2, и 3 были погружены (pushed) в стек в указанном порядке, а затем после извлечения их из стека.

Comment [w3]: Стеки очень часто называю структурами данных LIFO, от «Last In, First Out» («Последним пришел – первым вышел»).

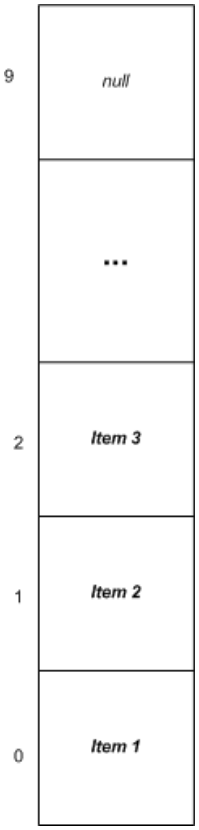

Рисунок 5. Графическое представление стека с тремя элементами

Рисунок 6. Графическое представление стека после извлечения трех элементов

Так же как и в случае Списка (List), когда внутренний массив Стека (Stack) требует изменения размера он автоматически увеличивается в размере в два раза от своего первоначального размера. (Вспомните, что этот коэффициент роста может быть опционально задан в конструкторе.)

3.1 Стеки: общепринятая метафора в компьютерной науке

Разговаривая об очередях можно легко вызвать в памяти параллели из реального мира, такие как очередь в булочной, обработка заданий на печать принтером, и так далее. Однако примеры из реального мира действующих вариантов стека гораздо сложнее вообразить. Несмотря на это стеки являются известной структурой данных в многообразии компьютерных приложений.

Например, рассмотрим любой компьютерный язык программирования, такой как C#. Когда программа C# выполняется, то среда поддержки CLR поддерживает стек вызовов (*call stack*), который, среди прочего, отслеживает вызовы функций. Каждый раз когда производится вызов функции информация о вызове добавляется в стек вызовов. По завершении выполнения функции связанная с ней информация извлекается из стека. Информация на вершине стека представляет текущую выполняемую функцию. (Для наглядной демонстрации стека вызовов функций создайте проект в Visual Studio .NET, задайте точку прерывания и перейдите к началу отладки (Debug/Start). Когда будет достигнута точка прерывания, то откройте окно Стек Вызовов (Call Stack) из меню Debug/Windows/Call Stack.)

Стеки также широко используются при синтаксическом анализе грамматик (от простых алгебраических выражений до компьютерных языков программирования), в качестве средства для реализации рекурсии и даже в качестве модели выполнения кода.

4. Ограничения последовательного индексирования

Вспомните из первой части нашей серии статей, что основным признаком массива является то, что он представляет собой набор однородных данных *индексируемых порядковым значением*. То есть, i-й элемент массива может быть получен за константное время. (Постоянное время обозначается $O(1)$.)

Однако мы редко знаем порядковый номер интересуемых нас данных. Например, рассмотрим базу данных сотрудников. Сотрудники могут быть уникально идентифицированы по их номеру социального обеспечения, который имеет форму DDD-DD-DDDD, где D это цифра (0-9). Если мы имеем всех сотрудников, которые были упорядочены случайным образом (т.е., неупорядочены), то поиск сотрудника 111-22-3333 потребовал бы, потенциально, просмотра *всех* элементов массива сотрудников, или операцию порядка $O(n)$. Несколько лучший подход был бы если бы мы отсортировали массив сотрудников по их номеру социального обеспечения, который снизил бы асимптотическое время поиска до $O(\log n)$.

В идеале мы хотели бы иметь доступ к записи сотрудника за время $O(1)$. Один из способов достичь этого состоит в том, чтобы построить огромный массив, который будет хранить каждый возможный номер социального обеспечения. То есть, наш массив будет начинаться с элемента 000-00-0000 и заканчиваться элементом 999-99-9999, как это показано на Рисунке 7.

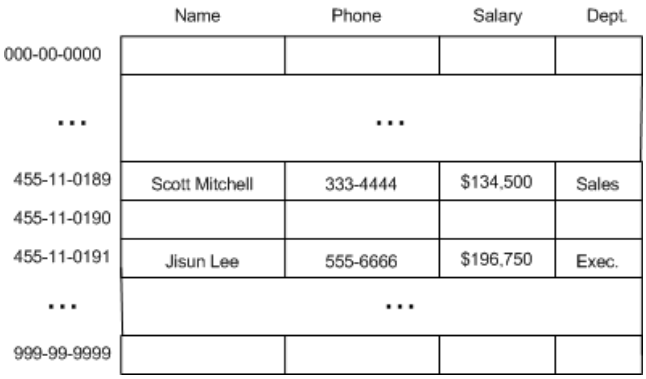

Рисунок 7. Массив, показывающий все возможные элементы для числа из 9-ти цифр

Как показано на этом рисунке каждая запись о сотруднике содержит такую информацию как имя, телефон, зарплату и так далее, и индексирован он номером социального обеспечения сотрудника. По такой схеме информация о сотруднике может быть доступна за константное время. Недостаток этого подхода состоит в крайне больших потерях памяти: существует 10^9 — то есть один *миллиард* (1,000,000,000) — различных номеров социального обеспечения. Для компании с 1,000 сотрудников только 0.0001% данного массива будет использоваться. (В перспективе ваша компания должна будет нанять около одной шестой населения земного шара, чтобы почти полностью использовать возможности такого массива.)

4.1 Сжатие порядкового индексирования с помощью хэш функции

Создание массива состоящего из одного миллиарда элементов для хранения информации об 1,000 сотрудников очевидно неприемлемо с точки зрения использования памяти. Однако, скорость доступа к информации о сотруднике за константное время очень желательна. Одна из возможностей состоит в уменьшении диапазона номеров социального обеспечения путем использования последних четырех цифр номера социального обеспечения сотрудника. То есть вместо того, чтобы массив охватывал номера с 000-00-0000 до 999-99-9999, он будет охватывать диапазон с 0000 до 9999. Рисунок 8, представленный ниже, дает графическое представление такого урезанного массива.

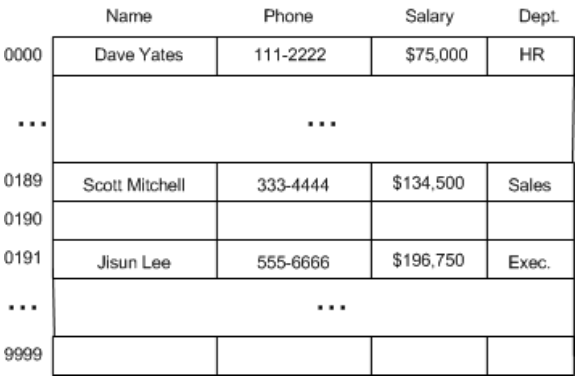

Рисунок 8. Урезанный массив

Этот подход предоставляет как константное время поиска так и более оптимальное использование оперативной памяти. Решение использовать четыре последние цифры номера социального обеспечения было произвольным. Мы могли бы использовать четыре средние цифры, или первую, третью, восьмую и девятую.

Математическое преобразование девятизначного номера социального обеспечения в четырехзначное число называется *хэшированием*. Массив, который использует хэширование для сжатия пространства индексов, называется *хэш-таблицей*.

Хэш-функция это функция, которая выполняет хэширование. Для примера с номером социального обеспечения, наша хэш-функция, *H*, может быть описана следующим образом:

```
H(x) = last four digits of x
```

Входным значением для *H* может быть любой девятизначный номер социального обеспечения, в то время как результатом *H* является четырехзначное число, которое просто представляет собой четыре последних цифры девятизначного номера социального обеспечения. В математических терминах, *H* отображает элементы из множества девятизначных номеров социального обеспечения во множество четырехзначных номеров социального обеспечения, так как это показано на Рисунке 9.

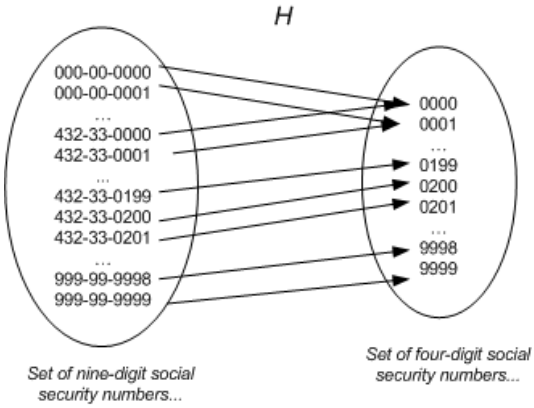

Рисунок 9. Графическое представление хеш-функции

Рисунок, приведенный выше, иллюстрирует поведение функции хеширования, называемое *коллизией*. В общем случае, с помощью функции хеширования вы сможете найти два элемента в большом множестве, которые отображаются в один и тот же элемент в малом множестве. В случае функции хеширования номеров социального обеспечения все номера заканчивающиеся на 0000 будут отображены в 0000. То есть, хеш значения для 000-00-0000, 113-14-0000, 933-66-0000 и многих других все будут равны 0000. (В действительности будет точно 10^5, или 100,000, номеров социального обеспечения, которые заканчиваются на 0000.)

Comment [w4]: Функция хеширования может быть описана в более точных математических обозначениях как функция $f : A \rightarrow B$. Так как $|A| > |B|$, то должны быть случаи, что f не является отображение один к одному; поэтому здесь будут коллизии.

Чтобы вернуться в контекст нашего предыдущего примера рассмотрим что произойдет, если будет добавлен новый сотрудник с номером социального обеспечения 123-00-0191. Попытка добавить этого сотрудника в массив приведет к проблеме, потому что в массиве уже существует сотрудник в позиции 0191 (Jisun Lee).

Ясно, что появление коллизий может привести к проблемам. В следующем разделе мы рассмотрим соотношение между хеш функцией и появлением коллизий, и вкратце исследуем некоторые стратегии обработки коллизий. В следующем разделе мы обратим ваше внимание на класс `System.Collections.Hashtable`, который обеспечивает реализацию хеш-таблицы. Мы рассмотрим хеш функцию класса Хеш-таблицы (Hashtable), стратегию разрешения коллизий, и несколько примеров использования класса Хеш-таблицы (**Hashtable**) на практике. Рассматривая класс **Hashtable**, мы изучим класс Словарь (**Dictionary**), который был добавлен в .NET Framework 2.0 Base Class Library. Класс **Dictionary** идентичен классу Hashtable, за исключением двух различий:

- Он использует параметризованные типы (Generics) и поэтому является строго типизированным.
- Он реализует альтернативную стратегию разрешения коллизий.

4.2 Как избегать и разрешать коллизии

При добавлении элемента к хеш таблице, именно коллизия создает основную проблему всей операции. Без коллизии мы можем добавить вставляемый элемент на место вычисленного хеш значения; однако при наличии коллизии мы должны предпринять некоторое корректирующее действие. По причине увеличения стоимости операции связанной с коллизией, нашей целью будет уменьшение количества коллизий до возможного минимума.

Частота возникновения коллизий напрямую связана с используемой хеш функцией и распределением данных передаваемых в эту хеш функцию. В случае с нашим примером номера социального обеспечения использование четырех последних цифр номера социального обеспечения сотрудника это идеальная хеш функция в предположении, что номера социального обеспечения назначаются случайным образом. Однако, если номер социального обеспечения назначается таким образом, что те, кто рожден в определенный год или в определенной местности имеют те же самые четыре последние цифры, то использование четырех последних цифр в качестве хеш значения может привести к большому количеству коллизий, если даты рождений сотрудников и места рождений сотрудников не распределены равномерно.

Comment [w5]: Основательный анализ значений хеш функций требует некоторого опыта в статистике, что не является предметом рассмотрения в нашей статье. По существу, мы хотим удостовериться, что для хеш таблицы с *k* ячейками, вероятность того, что случайное значение из области определений хеш функции будет отображено (mapped) в определенный элемент, равна 1/*k*.

Выбор соответствующей функции хеширования называется *избеганием коллизий*. Было проведено много исследований в этой области так как использование хеш функций может существенно повлиять на общую производительность хеш таблицы. В следующем разделе мы рассмотрим хеш функцию, используемую в классах **Hashtable** и **Dictionary** из .NET Framework.

В случае коллизии можно реализовать несколько стратегий. Задача которую нужно решить, разрешение коллизии (*collision resolution*), состоит в нахождении некоторого другого места, чтобы разместить вставляемый в хеш таблицу объект, так как нужная ячейка была уже занята. Один из наиболее простых подходов называется линейным просмотром (*linear probing*) и работает следующим образом:

1. Когда производится добавление нового элемента в хеш таблицу, то используется хеш функция для определения места в таблице которому этот элемент принадлежит.
2. Проверяется найденное место в таблице не находится ли там уже элемент данных. Если это место пусто, то в нем размещается новый элемент и процедура завершается, иначен переходим к шагу 3.

3. Если позиция определенная с помощью хеш функции была позицией *i*, просто проверьте позицию *i* + 1 чтобы определить является ли она свободной. Если она также занята, то проверьте *i* + 2, и так далее, пока не будет найдена свободная.

Рассмотрим случай когда в хеш таблицу добавляются следующие пять сотрудников: Алиса (333-33-1234), Боб (444-44-1234), Кэл (555-55-1237), Дэнни (000-00-1235), и Эдвард (111-00-1235). После этих добавлений хеш таблица будет выглядеть следующим образом:

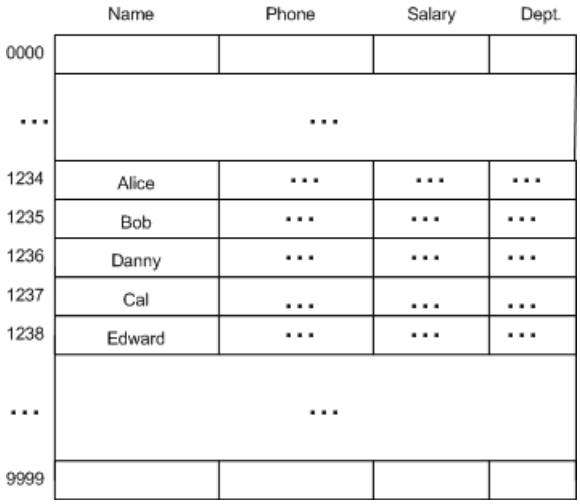

Рисунок 10. Хеш таблица для четырех сотрудников с одинаковыми номерами

Номер социального обеспечения Алисы имеет хеш значение 1234, и она будет вставлена в позицию 1234. Далее, номер социального обеспечения Боба имеет хеш значение 1234, но в этой позиции уже находится Алиса, поэтому Боб занимает следующую пустую позицию, которая имеет номер 1235. После Боба в таблицу вставляется Кэл, его хеш значение равно 1237. так как никто пока не занимает позицию 1237, то Кэл будет добавлен именно в эту позицию. Дэнни идет следующим, и его номер социального обеспечения имеет хеш значение 1235. Позиция 1235 занята, проверяется позиция 1236, и так как позиция 1236 свободна, то информация о Дэнни помещается в нее. И наконец, вставляется запись об Эдварде (Edward), его номер социального обеспечения также имеет хеш значение равное 1235. Позиция 1235 занята, проверяется позиция 1236. Она также занята, поэтому проверяется позиция 1237. Она занята информацией о Кэл, поэтому проверяется позиция 1238, которая является свободной, и поэтому информация об Эдварде помещается в нее.

В дополнение к запутыванию процесса вставки, коллизии также представляют проблему при поиске в хеш таблице. Например, имеем таблицу изображенную на Рисунке 10, и представим, что мы хотим найти информацию об Эдварде (Edward). Поэтому мы берем номер социального обеспечения Эдварда, 111-00-1235, вычисляем его хеш значение 1235, и начинаем наш поиск. Однако в ячейке 1235 мы находим Боба (Bob), а не Эдварда. Поэтому нам надо проверить ячейку 1236, но там находится информация о Дэнни (Danny). Наш линейный поиск будет продолжаться пока мы не найдем Эдварда или пустую ячейку. Нсли мы найдем пустую ячейку, то мы знаем, что информации об Эдварде нет в хеш таблице.

Последовательная проверка, хотя и является простой, не является хорошей стратегией разрешения коллизий потому, что она ведет к кластеризации (*clustering*). То есть представьте, что первые 10 сотрудников, которых мы добавляем в хеш таблицу, все имеют одно и то же хеш значение номера социального обеспечения, скажем 3344. Тогда будут заняты 10 последовательных ячеек с 3344 по 3353. Этот кластер требует последовательной проверки каждый раз, когда производится доступ к одному из этих 10 сотрудников. Более того, любой сотрудник с хеш значением от 3345 до 3353 будет добавлен в этот кластер и увеличит его размер. Для более быстрого поиска нам было бы необходимо иметь более равномерное распределение данных в хеш таблице, а не кластеризованное вокруг каких-нибудь определенных точек.

Более содержательной техникой проверки является квадратичная проверка (*quadratic probing*), которая начинает проверку ячеек на квадратичном расстоянии. То есть, если выбирается ячейка *s*, то вместо проверки ячеек $s + 1$, $s + 2$, и так далее как в линейной проверке, квадратичная проверка исследует сперва ячейку $s + 1^2$, затем $s - 1^2$, затем $s + 2^2$, затем $s - 2^2$, затем $s + 3^2$, и так далее. Однако, даже квадратичное хеширование может привести к кластеризации.

В следующем разделе мы рассмотрим третью технику разрешения коллизий, называемую *повторным хешированием* (*rehasing*), которая является техникой используемой классом **Hashtable** из .NET Framework. В заключительном разделе мА рассмотрим класс **Dictionary**, который использует технику разрешения коллизий, называемую методом построения цепочек (*chaining*).

5. Класс System.Collections.Hashtable

.NET Framework Base Class Library содержит реализацию хеш таблицы в виде класса Хеш-таблица (**Hashtable**). При добавлении элемента в Хеш-таблицу (Hashtable), вы должны задать не только само значение, но и уникальный ключ по которому можно доступиться к данному элементу. И ключ и значение могут быть любого типа. В нашем примере сотрудников ключом мог бы быть номер социального обеспечения сотрудника. Элементы в Хеш-таблицу добавляются с помощью метода `Add()`.

Чтобы извлечь элемент из Хеш-таблицы вы можете использовать в качестве индекса ключ, так же как и в случае индексирования массива порядковым числом. Следующая короткая программа на C# демонстрирует эту концепцию. Она добавляет несколько элементов в Хеш-таблицу, ассоциируя строковый ключ с каждым элементом. Затем определенный элемент можно извлечь, используя этот строковый ключ.

```
using System;
using System.Collections;

public class HashtableDemo
{
    private static Hashtable employees = new Hashtable();

    public static void Main()
    {
        // Add some values to the Hashtable, indexed by a string key
        employees.Add("111-22-3333", "Scott");
        employees.Add("222-33-4444", "Sam");
        employees.Add("333-44-55555", "Jisun");

        // Access a particular key
        if (employees.ContainsKey("111-22-3333"))
```

```
        {
            string empName = (string) employees["111-22-3333"];
            Console.WriteLine("Employee 111-22-3333's name is: " + empName);
        }
        else
            Console.WriteLine("Employee 111-22-3333 is not in the hash
table...");
    }
}
```

Этот код также демонстрирует использование метода `ContainsKey()`, который возвращает логическое значение, указывающее был ли найден данный ключ в хэш-таблице (Hashtable). Класс Хеш-таблицы содержит свойство `Keys`, которое возвращает набор ключей, используемых в хэш-таблице. Это свойство можно использовать для перечисления элементов хэш-таблицы, например так:

```
// Step through all items in the Hashtable
foreach(string key in employees.Keys)
    Console.WriteLine("Value at employees[\"" + key + "\"] = " +
employees[key].ToString());
```

Помните, что порядок, в котором добавляются элементы, и порядок, ключей в коллекции `Keys`, не обязательно тот же самый. Ячейка, в которой хранится элемент данных, зависит от хеш значения ключа и стратегии разрешения коллизий. Если вы выполните код, приведенный выше, то вы увидите, что порядок в котором перечислены элементы не обязательно совпадает с порядком в котором элементы были добавлены в хеш-таблицу. Вот результат выполнения приведенного выше кода:

```
Value at employees["333-44-5555"] = Jisun
Value at employees["111-22-3333"] = Scott
Value at employees["222-33-4444"] = Sam
```

Даже если данные были добавлены в хеш-таблицу в порядке "Scott," "Sam," "Jisun."

5.1 Функция хеширования класса Хеш-таблицы (Hashtable)

Функция хеширования класса Хеш-таблиц (Hashtable) несколько сложнее функции хеширования номера социального обеспечения рассмотренной ранее. Прежде всего помните, что функция хеширования должна возвращать порядковое число. Это было легко сделать в примере с номером социального обеспечения, так как такой номер уже сам по себе число. Чтобы получить подходящее хеш значение мы просто отбрасывали все цифры числа, кроме четырех последних. Однако вы должны понимать, что класс Хеш-таблиц (Hashtable) может использовать ключ *любого* типа. Как мы видели в предыдущем примере, ключами могут быть строки, такие как «Scott» или «Sam». В таком случае естественным желанием будет знать как хеш функция превращает строку в число.

Это магическое преобразование может произойти благодаря методу `GetHashCode()`, который определен в классе `System.Object`. Реализация метода в классе `Object` возвращает уникальное целое число, которое гарантировано не изменится за время жизни объекта. Так как каждый метод является наследником, прямым или непрямым, класса `Object`, то все объекты имеют доступ к этому методу. Поэтому строка или любой другой тип может быть представлен уникальным целым числом. Конечно же, этот метод может быть переопределен для того, чтобы предоставить более подходящую функцию хеширования для данного класса. (Например, класс `Point` в пространстве имен

`System.Drawing`, переопределяет метод `GetHashCode()`, возвращая результат операции XOR над ее членами класса `x` и `y`.)

Функция хеширования класса Хеш-таблиц (Hashtable) определена следующим образом:

```
H(key) = [GetHash(key) + 1 + (((GetHash(key) >> 5) + 1) % (hashsize – 1))] %
hashsize
```

Здесь, GetHash(*key*) это, по умолчанию, результат, возвращаемый вызовом `GetHashCode()` (хотя при использовании Хеш-таблицы вы можете определить свою собственную функцию хеширования `GetHash()`). GetHash(*key*) >> 5 вычисляет хеш для ключа (*key*) и затем сдвигает результат на 5 разрядов вправо – это то же самое, что и выполнение операции деления на 32. Как это уже обсуждалось ранее в этой статье, оператор `%` выполняет модульную арифметику. `hashsize` это общее число ячеек в хеш таблице. (Вспомните также, что *x* `%` *y* возвращает остаток от деления *x* / *y*, и что этот остаток всегда лежит в пределах от 0 до *y* – 1.) Благодаря этой операции взятия остатка конечный результат состоит в том, что *H*(*key*) будет всегда числом в диапазоне от 0 до *hashsize* – 1. Так как *hashsize* это общее число ячеек в хеш таблице, то результирующее хеш значение будет всегда указывать на приемлемый диапазон ячеек.

5.2 Разрешение коллизий в классе Хеш-таблиц (Hashtable)

Вспомните, что при вставке элемента в хеш таблицу или при извлечении элемента из хеш таблицы может произойти коллизия. При добавлении нового элемента необходимо найти пустую ячейку. При извлечении элемента необходимо найти требуемый элемент, если он не находится в ожидаемом месте. Ранее мы вкратце рассмотрели две стратегии разрешения коллизий:

- Линейный просмотр
- Квадратичный просмотр

Класс **Hashtable** использует другую технику, называемую повторным хешированием (*rehasing*). (Некоторые источники называют ее *двойным хешированием* (*double hashing*.))

Повторное хеширование работает следующим образом: пусть имеется набор различных хеш функций H_1 ... H_n, и при вставке или извлечении элемента из хеш таблицы первоначально используется функция H_1. Если это приводит к коллизии, то производится попытка использования хеш функции H_2 и так далее вплоть до H_n при необходимости. В предыдущем разделе было показано использование только одной хеш функции, которая и являлась первоначальной функцией (H_1). Другие хеш функции очень похожи на эту функцию отличаясь только в коэффициенте умножения. В общем случае хеш функция H_k определяется следующим образом:

```
Hk(key) = [GetHash(key) + k * (1 + (((GetHash(key) >> 5) + 1) % (hashsize –
1)))] % hashsize
```

Повторное хеширование обеспечивает лучшую технику избежания коллизий, чем линейный или квадратичный просмотр.

Comment [w6]: При повторном хешировании очень важно, чтобы каждая ячейка в хеш таблице посещалась бы в точности только один раз, когда делается `hashsize` просмотров. То есть, для данного ключа вы не хотите, чтобы H_i и H_j хешировались в одну и ту же ячейку хеш таблицы. С помощью формулы повторного хеширования используемой в классе **Hashtable**, это свойство поддерживается если результат вычисления (1 + (((GetHash(*key*) >> 5) + 1) % (hashsize – 1)) и `hashsize` являются взаимно простыми числами. (Два числа являются взаимно простыми, если они не имеют общих множителей.) Эти два числа будут гарантировано взаимно простыми, если `hashsize` будет простым числом.

5.3 Коэффициент загрузки и расширение Хеш-таблицы

Класс Хеш-таблицы (**Hashtable**) содержит приватный член класса называемый `loadFactor` который задает максимальное отношение количества элементов в хеш таблице к общему числу ячеек. `loadFactor` равный, скажем, 0.5, означает, что хеш таблица может содержать по крайней мере половину из своих ячеек заполненными элементами данных, в то время как вторая половина ячеек будет оставаться пустой.

В переопределенной версии конструктора Хеш-таблицы (Hashtable), вы можете задать значение `loadFactor` числом между 0.1 и 1.0. Помните, однако, что какое бы вы значение ни дали оно будет масштабировано на 72%, так что если вы передадите в конструтор 1.0, то действительное значение `loadFactor` будет равно 0.72. Величина 0.72 была определена в Microsoft как оптимальный коэффициент загрузки, поэтому рассмотрите возможность использования по умолчанию величины 1.0 в качестве коэффициента загрузки (который будет автоматически масштабировано до 0.72).

Comment [w7]: Я провел несколько дней спрашивая в различных форумах и у различных приятелей из Microsoft *почему* применяется это автоматическое масштабирование. Я хотел также знать также почему они не использовали явный диапазон между 0.072 и 0.72? Я закончил обсуждение с командой из Microsoft, которая работала над классом Хеш-таблицы, и они поделились причиной своего решения. В частности, команда определила в результате эмпирического тестирования, что значения больше 0.72 очень серьезно снижали производительность. Они приняли решение, что разработчик, использующий Хеш-таблицу будет ошибаться меньше, если он запомнит величину 1.0 вместо кажущейся произвольной величины 0.72.

Всякий раз когда добавляется новый элемент в класс Хеш-таблицы (Hashtable), производится проверка, чтобы удостовериться, что добавление нового элемента не подтолкнет отношение числа элементов к числу ячеек за определенное максимальное отношение. Если же это приведет к такому увеличению, то Хеш-таблица будет расширена (*expanded*). Расширение таблицы осуществляется в виде двух шагов:

1. Число ячеек в Хеш-таблице (Hashtable) приблизительно удваивается. Более точно число ячеек увеличивается от текущего значения в виде простого числа до следующего простого числа. Вспомните, что для того, чтобы повторное хеширование выполнилось успешно, необходимо чтобы число ячеек в хеш таблице было простым числом.
2. Так как хеш значение каждого элемента в хеш таблице зависит от общего числа ячеек в хеш таблице, то все значения необходимо повторно вычислить (так как число ячеек было увеличено на шаге 1).

К счастью класс Хэш-таблицы (**Hashtable**) прячет всю эту сложность в методе `Add()`, поэтому вам нет необходимости беспокоиться о деталях.

Коэффициент загрузки оказывает влияние на общий размер хеш таблицы и ожидаемое количество просмотров, необходимое при возникновении коллизии. Более высокий коэффициент загрузки, который позволяет иметь относительно плотную хеш таблицу, требует меньше памяти, но большего числа просмотров при возникновении коллизии, чем разреженная хеш таблица. Без вникания в сложность анализа, ожидаемое число необходимых просмотров при возникновении коллизии, равно по крайней мере $1 / (1 - lf)$, где lf это коэффициент загрузки.

Как упоминалось ранее, в Microsoft произвели настройку Хеш-таблицы для использования по умолчанию коэффициента загрузки равного 0.72. Поэтому вы можете ожидать в среднем 3.5 просмотра на коллизию. Так как эта оценка не изменяется на основе числа элементов в Хеш-таблице, то асимптотическое время доступа к элементу для Хеш-таблицы равно $O(1)$, которое превосходит время поиска $O(n)$ для массива.

И наконец, отдавайте себе отчет в том, что расширение Хеш-таблицы не является недорогой операцией. Поэтому, если вы имеете оценку количества элементов, которое будет хранить Хеш-таблица, то вы сможете задать начальную емкость Хеш-таблицы в конструкторе, чтобы избежать нежелательных увеличений размера.

6. Класс System.Collections.Generic.Dictionary

Хэш-таблица (Hastable) это слабо-типизированная структура данных, так как разработчик может добавлять в Хэш-таблицу ключи и значения любого типа. Как мы видели на примере класса Список (**List**), а также классов Очередь (**Queue**) и (**Stack**), с появлением параметризованных типов (Generics) в .NET Framework 2.0, многие из встроенных структур данных были обновлены, чтобы обеспечить поддержку типобезопасных версий с использованием параметризованных типов (Generics). Класс Словарь (**Dictionary**) это типобезопасная реализация Хэш-таблицы (Hastable) и имеет сильную типизацию как для ключей, так и для значений. При создании экземпляра Словаря (Dictionary), вы должны указать типы данных и для ключа и для значения, используя следующий синтаксис:

```
Dictionary<keyType, valueType> variableName = new Dictionary<keyType,
valueType>();
```

Возвращаясь к нашему предыдущему примеру сортировки информации о сотрудниках используя четыре последних цифры номера социального обеспечения в качестве хэш значения мы можем создать Словарь (Dictionary) ключ которого имеет целочисленный тип (девять цифр номера социального обеспечения сотрудника), а значение имеет тип `Employee` (предполагая, что существует некоторый класс `Employee`):

```
Dictionary<int, Employee> employeeData = new Dictionary<int, Employee>();
```

Как только вы создали экземпляр объекта Словарь, вы можете добавлять и удалять элементы точно так же как вы делали с классом ... **Hashtable**.

```
// Добавить несколько сотрудников
employeeData.Add(455110189) = new Employee("Scott Mitchell");
employeeData.Add(455110191) = new Employee("Jisun Lee");
...
// Проверить, числится ли сотрудник с SSN 123-45-6789
if (employeeData.ContainsKey(123456789))
    ...
```

6.1 Разрешение коллизий в классе Dictionary

Класс Словарь (Dictionary) отличается от класса Хеш-таблицы (Hashtable) более чем в одном аспекте. В дополнение к тому, что он является сильно-типизированным, класс Словарь (**Dictionary**) также реализует отличную стратегию разрешения коллизий от класса Hashtable, технику называемую методом построения цепочек (*chaining*). Вспомните, что при просмотре в случае коллизии производится попытка использования другой ячейки. (При использовании повторного хеширования производится повторное вычисление хеша и снова производится попытка использования этой ячейки.) Однако при использовании метода цепочек для разрешения коллизий используется другая структура данных. В частности, каждая ячейка в Словаре (**Dictionary**) содержит массив элементов. В случае коллизии, элемент приводящий к коллизии добавляется в конец списка содержащегося в этом узле (или ячейке).

Чтобы лучше понять как работает технология построения цепочек будет полезным визуализировать Словарь (**Dictionary**) в виде хеш таблицы чьи узлы списков содержат связанные списки элементов, которые имеют один и тот же хеш. Рисунок 11 иллюстрирует то, как набор элементов, которые отображаются (hash) в один и тот же узел списков формируют цепочку в данном узле.

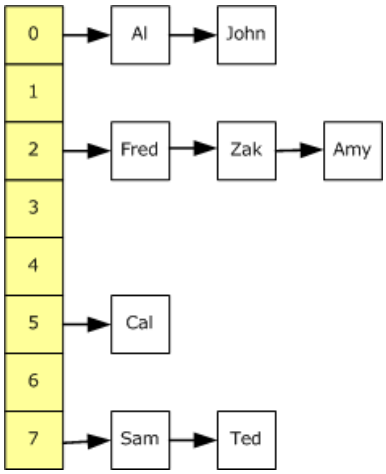

Рисунок 11. Цепочки, созданные последовательностью элементов

Словарь на Рисунке 11 имеет восемь узлов списков, показанные желтым цветом и идущие снизу вверх. Несколько объектов `Employee` были добавлены в Словарь. Когда объект `Employee` добавляется в Словарь, то он добавляется к узлу хеш которого соответствует ключу и если в этом узле уже находится объект класса `Employee`, то он добавляется в начало списка объектов `Employee`. То есть, сотрудники Эл и Джон отображаются в один и тот же узел, так же как и Фред, Зак и Ами, а также Сэм и Тед. Вместо того, чтобы выполнять повторный просмотр в случае коллизии, как это сделано в реализации класса Хеш-таблицы (Hashtable), Словарь (Dictionary) просто строит цепочку элементов приводящих к коллизии в данном узле.

Comment [w8]: В нашем обсуждении класса **Hashtable**, было отмечено, что среднее асимптотическое время выполнения для добавления, удаления и поиска в хеш таблице с использованием просмотра (probing) является константным временем, *O*(1). Добавление элемента в хеш таблицу, которая использует цепочки, также требует константного времени так как это требует только вычисления хеш значения элемента и добавления его в соответствующий узел списка. Поиск и удаление элементов из хеш таблицы с цепочками, однако, имеет в среднем время пропорциональное общему количеству элементов содержащихся в хеш-таблице и количеству узлов. В частности, время выполнения равно *O*(*n*/*m*), где *n* это общее число элементов в хеш-таблице и *m* это число узлов. Класс Словарь (Dictionary) реализован таким образом, что *n* = *m* всегда. То есть сумма всех элементов в цепочках никогда не может превысить числа узлов. Так как *n* никогда не превышает *m*, поиск и удаление производится также за константное время.

7. Заключение

В этой статье мы изучили четыре структуры данных, для которых существуют соответствующие классы в .NET Framework Base Class Library:

- Очередь (Queue)
- Стек (Stack)
- Хэш таблицу (Hashtable)
- Словарь (Dictionary)

Очередь (Queue) и Стек (Stack) предоставляют возможности подобные Списку (List) в том смысле, что они могут хранить произвольное количество элементов. Очередь и Стек отличаются от Списка в том, что Список позволяет прямой и произвольный доступ к его элементам, а Очередь и Стек ограничивают правила доступа к элементам.

Очередь использует стратегию FIFO, или «первым пришел – первым обслужен». То есть, порядок в котором элементы удаляются из Очереди в точности тот же в котором они были добавлены в Очередь. Чтобы обеспечить эту семантику Очередь предлагает два метода: `Enqueue()` и `Dequeue()`. Очереди являются полезными структурами данных для обслуживания заданий или других задач, где порядок в котором обрабатываются элементы основан на порядке в котором они получены.

Стек, с другой стороны, предлагает доступ LIFO, который означает «последним пришел – первым обслужен». Стеки обеспечивают такую схему доступа с помощью методов `Push()` и `Pop()`. Стеки используются в большом количестве областей компьютерных технологий от исполнения кода до синтаксического анализа.

Две заключительные рассмотренные структуры данных были Хеш-таблица (Hashtable) и Словарь (Dictionary). Хеш-таблица расширяет массив (ArrayList) позволяя индексирование элементов с помощью произвольного ключа, в отличие от индексирования порядковым значением. Если вы планируете поиск в массиве по определенному уникальному ключу, то будет более эффективным использовать Хеш-таблицу, так как поиск по значению ключа будет производиться за константное время в отличие от линейного времени для массива. Словарь (Dictionary) предоставляет типобезопасную Хеш-таблицу с альтернативной стратегией разрешения коллизий.

На этом завершается вторая статья из данной серии статей. В третьей статье мы рассмотрим бинарные деревья поиска – структуру данных, которая обеспечивает время поиска $O(\log n)$. Так же как и Хеш-таблицы бинарные деревья поиска служат идеальным выбором перед массивами если вы знаете, что поиск данных будеи производиться достаточно часто.

До следующей встречи, счастливого программирования!

Игорь Изварин