14.3. Хеширование

В этой книге мы вывели ряд списковых структур, позволяющих программе-клиенту осуществлять поиск и выборку данных. В каждой такой структуре

метод Find выполняет обход списка и ищет элемент данных, совпадающий с ключом. При этом эффективность поиска зависит от структуры списка. В случае последовательного списка метод Find гарантированно просматривает $O(n)$ элементов, в то время как в случае бинарных поисковых деревьев и при бинарном поиске обеспечивается более высокая эффективность $O(\log_2 n)$. В идеале нам хотелось бы выбирать данные за время $O(1)$. В этом случае число необходимых сравнений не зависит от количества элементов данных. Выборка элемента осуществляется за время $O(1)$ при использовании в качестве индекса в массиве некоторого ключа. Например, блюда из меню в закусочной в целях упрощения бухгалтерского учета обозначаются номерами. Какой-нибудь деликатес типа "бастурма, маринованная в водке" в базе данных обозначается просто #2. Владельцу закусочной остается лишь сопоставить ключ 2 с записью в списке.

Мы знаем и другие примеры. Файл клиентов пункта проката видеокассет содержит семизначные номера телефонов. Номер телефона используется в качестве ключа для доступа к конкретной записи файла клиентов.

Ключи не обязательно должны быть числовыми. Например, формируемая компилятором **таблица символов (symbol table)** содержит все используемые в программе идентификаторы вместе с сопутствующей каждому из них информацией. Ключом для доступа к конкретной записи является сам идентификатор.

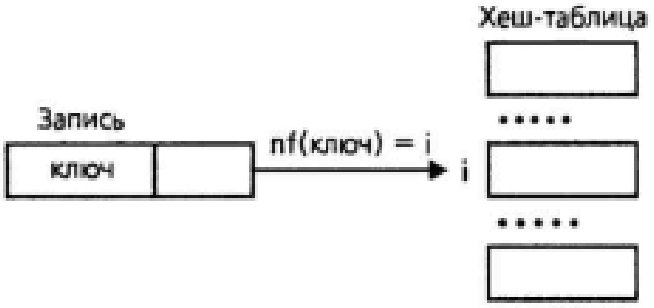
Ключи и хеш-функция

В общем случае ключи не настолько просты, как в примере с закусочной. Несмотря на то, что они обеспечивают доступ к данным, ключи, как правило, не являются непосредственными индексами в массиве записей. Например, телефонный номер может идентифицировать клиента, но вряд ли пункт проката хранит десятимиллионный массив.

В большинстве приложений ключ обеспечивает косвенную ссылку на данные. Ключ отображается во множество целых чисел посредством **хеш-функции (hash function)**. Полученное в результате значение затем используется для доступа к данным. Давайте исследуем эту идею.

Предположим, есть множество записей с целочисленными ключами. Хеш-функция HF отображает ключ в целочисленный индекс из диапазона 0...n-1.

С хеш-функцией связана так называемая **хеш-таблица (hash table)**, ячейки которой пронумерованы от 0 до n-1 и хранят сами данные или ссылки на данные.



Пример 14.1

Предположим, Key — положительное целое, а HF(Key) — значение младшей цифры числа Key. Тогда диапазон индексов равен 0—9. Например, если Key = 49, HF(Key) = HF(49) = 9. Эта хеш-функция в качестве возвращаемого значение использует остаток от деления на 10.

```
// Хеш-функция, возвращающая младшую цифру ключа
int HF(int key)
{
  return key % 10;
}
```

Часто отображение, осуществляемое хеш-функцией, является отображением "многие к одному" и приводит к **коллизиям (collisions)**. В примере 14.1 HF(49) = HF(29) = 9. При возникновении коллизии два или более ключа ассоциируются с одной и той же ячейкой хеш-таблицы. Поскольку два ключа не могут занимать одну и ту же ячейку в таблице, мы должны разработать стратегию разрешения коллизий. Схемы разрешения коллизий обсуждаются после знакомства с некоторыми типами хеш-функций.

Хеш-функции

Хеш-функция должна отображать ключ в целое число из диапазона 0...n-1. При этом количество коллизий должно быть ограниченным, а вычисление самой хеш-функции — очень быстрым. Некоторые методы удовлетворяют этим требованиям.

Наиболее часто используется **метод деления (division method)**, требующий двух шагов. Сперва ключ должен быть преобразован в целое число, а затем полученное значение вписывается в диапазон 0...n-1 с помощью оператора получения остатка. На практике метод деления используется в большинстве приложений с хешированием.

Пример 14.2

1. Ключ — пятизначное число. Хеш-функция извлекает две младшие цифры. Например, если это число равно 56389, то HF(56389) = 89. Две младшие цифры являются остатком от деления на 100.

```
int HF(int key)
{
  return key % 100      // метод деления на 100
}
```

Эффективность хеш-функции зависит от того, обеспечивает ли она равномерное рассеивание ключей в диапазоне 0...n-1. Если две последние цифры соответствуют году рождения, то будет слишком много коллизий при идентификации подростков, играющих в юношеской бейсбольной лиге.

2. Ключ — символьная строка языка C++. Хеш-функция отображает эту строку в целое число посредством суммирования первого и последнего символов и последующего деления на 101 (размер таблицы).

```
// хеш-функция для символьной строки.
// возвращает значение в диапазоне от 0 до 100
int HF(char *key)
{
  int len = strlen(key), hashf = 0;

  // если длина ключа равна 0 или 1, возвратить key[0].
  // иначе сложить первый и последний символ
  if (len <= 1)
    hashf = key[0];
  else
    hashf = key[0] + key[len-1];

  return hashf % 101;
}
```

Эта хеш-функция приводит к коллизии при одинаковых первом и последнем символах строки. Например, строки "start" и "slant" будут отображаться в индекс 29. Так же ведет себя хеш-функция, суммирующая все символы строки.

```
int HF(char *key)
{
  int hashf = 0;

  // просуммировать все символы строки и разделить на 101
  while (*key)
    hashf += *key++;

  return hashf % 101;
}
```

Строки "bad" и "dab" преобразуются в один и тот же индекс. Лучшие результаты дает хеш-функция, производящая перемешивание битов в символах. Пример более удачной хеш-функции для строк представлен вместе с программой 14.2.

В общем случае при больших n индексы имеют больший разброс. Кроме того, математическая теория утверждает, что распределение будет более равномерным, если n — простое число.

Другие методы хеширования

Метод середины квадрата (midsquare technique) предусматривает преобразование ключа в целое число, возведение его в квадрат и возвращение в качестве значения функции последовательности битов, извлеченных из середины этого квадрата. Предположим, что ключ есть целое 32-битовое число. Тогда следующая хеш-функция извлекает средние 10 бит возведенного в квадрат ключа.

```
// возвратить средние 10 бит произведения key*key
int HF(int key);
{
  key *= key;              // возвести ключ в квадрат
  key >>= 11;              // отбросить 11 младших бит
  return key % 1024        // возвратить 10 младших бит
}
```

При мультипликативном методе (multiplicative method) используется случайное действительное число f в диапазоне $0 \le f < 1$. Дробная часть произведения f*key лежит в диапазоне от 0 до 1. Если это произведение умножить на n (размер хеш-таблицы), то целая часть полученного произведения даст значение хеш-функции в диапазоне от 0 до n–1.

```
// хеш-функция, использующая мультипликативный метод;
// возвращает значение в диапазоне 0...700
int HF(int key);
{
  static RandomNumber rnd;
  float f;

  // умножить ключ на случайное число из диапазона 0...1
  f = key * rnd.fRandom();
  // взять дробную часть
  f = f - int(f);
  // возвратить число в диапазоне 0...n-1
  return 701*f;
}
```

Разрешение коллизий

Несмотря на то, что два или более ключей могут хешироваться одинаково, они не могут занимать в хеш-таблице одну и ту же ячейку. Нам остаются два пути: либо найти для нового ключа другую позицию в таблице, либо создать для каждого значения хеш-функции отдельный список, в котором будут все ключи, отображающиеся при хешировании в это значение. Оба варианта представляют собой две классические стратегии разрешения коллизий — **открытую адресацию с линейным опробыванием (linear probe open addressing)** и **метод цепочек (chaining with separate lists)**. Мы проиллюстрируем на примере открытую адресацию, а сосредоточимся главным образом на втором методе, поскольку эта стратегия является доминирующей.

Открытая адресация с линейным опробыванием. Эта методика предполагает, что каждая ячейка таблицы помечена как незанятая. Поэтому при добавлении нового ключа всегда можно определить, занята ли данная ячейка таблицы или нет. Если да, алгоритм осуществляет "опробывание" по кругу, пока не встретится "открытый адрес" (свободное место). Отсюда и название метода. Если размер таблицы велик относительно числа хранимых там клю-

чей, метод работает хорошо, поскольку хеш-функция будет равномерно рассеивать ключи по всему диапазону и число коллизий будет минимальным. По мере того как коэффициент заполнения таблицы приближается к 1, эффективность процесса заметно падает. Проиллюстрируем линейное опробывание на примере семи записей.

Пример 14.3

Предположим, что данные имеют тип DataRecord и хранятся в 11-элементной таблице.

```
struct DataRecord
{
  int key;
  int data;
};
```

В качестве хеш-функции HF используется остаток от деления на 11, принимающий значения в диапазоне 0—10.

```
HF(item) = item.key % 11
```

В таблице хранятся следующие данные. Каждый элемент помечен числом проб, необходимых для его размещения в таблице.

```
Список: (54,1), (77,3), (94,5), (89,7), (14,8), (45,2), (76,9)
```

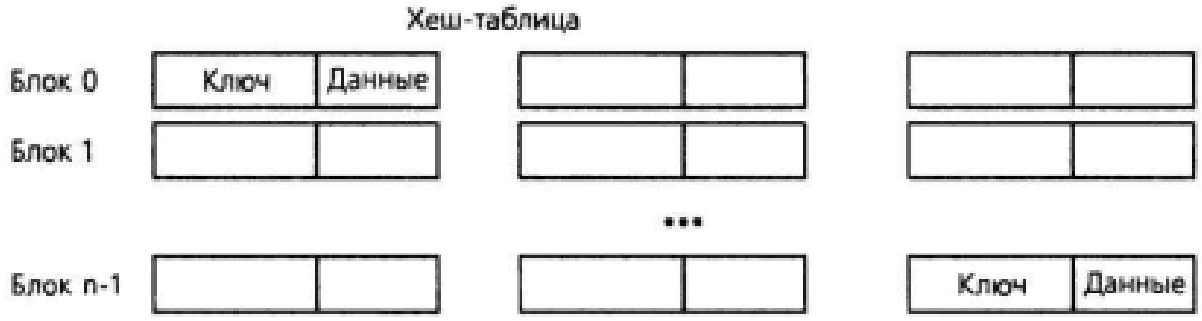
Хеш-таблица

77(1) 3	89(1) 7	45(2) 2	14(1) 8	76(6) 9		94(1) 5				54(1) 1
0	1	2	3	4	5	6	7	8	9	10

Хеширование первых пяти ключей дает пять различных индексов, по которым эти ключи запоминаются в таблице. Например, HF({54,1}) = 10, и этот элемент попадает в Table[10]. Первая коллизия возникает между ключами 89 и 45, так как оба они отображаются в индекс 1. Элемент данных {89,7} идет первым в списке и занимает позицию Table[1]. При попытке записать {45,2} оказывается, что место Table[1] уже занято. Тогда начинается последовательное опробывание ячеек таблицы с целью нахождения свободного места. В данном случае это Table[2]. На ключе 76 эффективность алгоритма сильно падает. Этот ключ хешируется в индекс 10 — место, уже занятое. В процессе опробывания осуществляется просмотр еще пяти ячеек, прежде чем будет найдено свободное место в Table[4]. Общее число проб для размещения в таблице всех элементов списка равно 13, т.е. в среднем 1,9 проб на элемент.

Реализация алгоритма открытой адресации дана в упражнениях.

Метод цепочек. При другом подходе к хешированию таблица рассматривается как массив связанных списков или деревьев. Каждая такая цепочка называется **блоком (bucket)** и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Эта стратегия разрешения коллизий называется **методом цепочек.**



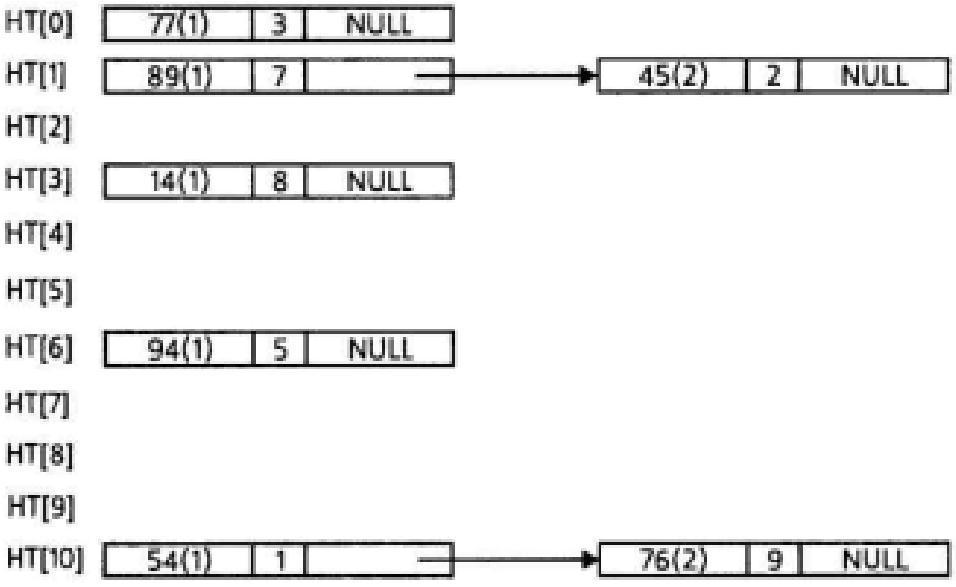
Если таблица является массивом связанных списков, то элемент данных просто вставляется в соответствующий список в качестве нового узла. Чтобы обнаружить элемент данных, нужно применить хеш-функцию для определения нужного связанного списка и выполнить там последовательный поиск.

Пример 14.4

Проиллюстрируем метод цепочек на семи записях типа DataRecord и хеш-функции HF из примера 14.3.

```
Список: {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9}
HF(item) = item.key % 11
```

Каждый новый элемент данных вставляется в хвост соответствующего связанного списка. В следующей таблице каждое значение данных сопровождается числом проб, требуемых для запоминания этого значения в таблице.



Заметьте, что если считать пробой вставку нового узла, то их общее число при включении семи элементов равно 9, т.е. в среднем 1,3 пробы на элемент данных.

В общем случае метод цепочек быстрее открытой адресации, так как просматривает только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, открытая адресация предполагает наличие таблицы фиксированного размера, в то время как в методе цепочек элементы таблицы создаются динамически, а длина списка ограничена лишь количеством памяти. Основным недостатком метода цепочек являются дополнительные затраты памяти на поля указателей. В общем случае динамическая структура метода цепочек более предпочтительна для хеширования.

14.4. Класс хеш-таблиц

В этом разделе определяется общий класс HashTable, осуществляющий хеширование методом цепочек. Этот класс образуется от базового абстрактного класса List и обеспечивает механизм хранения с очень эффективными методами доступа. Допускаются данные любого типа с тем лишь ограничением, что для этого типа данных должен быть определен оператор ==. Чтобы сравнить ключевые поля двух элементов данных, прикладная программа должна перегрузить оператор ==.

Мы также рассмотрим класс HashTableIterator, облегчающий обработку данных в хеш-таблице. Объект типа HashTableIterator находит важное применение при сортировке и печати данных.

Объявления и реализации этих классов находятся в файле hash.h.

Спецификация класса HashTableIterator

ОБЪЯВЛЕНИЕ

```
#include "array.h"
#include "list.h"
#include "link.h"
#include "iterator.h"

template <class T>
class HashTableIterator;

template <class T>
class HashTable: public List<T>
{
  protected:
    // число блоков; представляет размер таблицы
    int numBuckets;

    // хеш-таблица есть массив связанных списков
    Array< LinkedList<T> > buckets;

    // хеш-функция и адрес элемента данных,
    // к которому обращались последний раз
    unsigned long (*hf)(T key);
    T *current;

  public:
    // конструктор с параметрами, включающими
    // размер таблицы и хеш-функцию
    HashTable(int nbuckets, unsigned long hashf(T key));

    // методы обработки списков
    virtual void Insert(const T& key);
    virtual int Find(T& key);
    virtual void Delete(const T& key);
    virtual void ClearList(void);
    void Update(const T& key);

    // дружественный итератор, имеющий доступ к
    // данным-членам
    friend class HashTableIterator<T>
}
```

ОПИСАНИЕ

Объект типа HashTable есть список элементов типа T. В нем реализованы все методы, которые требует абстрактный базовый класс List. Прикладная

программа должна задать размер таблицы и хеш-функцию, преобразующую элемент типа T в длинное целое без знака. Такой тип возвращаемого значения допускает хеш-функции для широкого диапазона данных. Деление на размер таблицы осуществляется внутри.

Методы Insert, Find, Delete и ClearList являются базовыми методами обработки списков. Отдельный метод Update служит для обновления элемента, уже имеющегося в таблице.

Методы ListSize и ListEmpty реализованы в базовом классе. Элемент данных current всегда указывает на последнее доступное значение данных. Он используется методом Update и производными классами, которые должны возвращать ссылки. Пример такого класса дан в разделе 14.7.

ПРИМЕР

Предположим, что NameRecord есть запись, содержащая поле наименования и поле счетчика.

```
struct NameRecord
{
  String name;
  int count;
};

// 101-элементная таблица, содержащая данные типа NameRecord
// и имеющая хеш-функцию hash
HashTable<NameRecord> HF(101,hash);

// вставить запись {"Betsy",1} в таблицу
NameRecord rec;          // переменная типа NameRecord
rec.name = "Betsy";      // присвоение name = "Betsy
rec.count = 1;           // и count = 1
HF.Insert(rec);          // Вставить запись
cout << HF.ListSize();   // распечатать размер таблицы

// выбрать значение данных, соответствующее ключу "Betsy",
// увеличить поле счетчика на 1 и обновить запись
rec.name = "Betsy";
if (HF.Find(rec)         // найти "Betsy"
{
  rec.cout += 1;         // обновить поле данных
  HF.Update(rec);        // обновить запись в таблице
}
else
  cerr << "Ошибка: \"Ключ Betsy должен быть в таблице.\"\n;
```

Класс HashTableIterator образован из абстрактного класса Iterator и содержит методы для просмотра данных в таблице.

Спецификация класса HashTableIterator

ОБЪЯВЛЕНИЕ

```
template <class T>
class HashTableIterator: public Iterator<T>
{
  private:
    // указатель таблицы, подлежащей обходу
    HashTable<T> *HashTable;

    // индекс текущего просматриваемого блока
    // и указатель на связанный список
    int currentBucket;
```

```
    LinkedList<T> *currBucketPtr;

    // утилита для реализации метода Next
    void SearchNextNode(int cb);

  public:
    // конструктор
    HashTableIterator (HashTable<T>& ht);

    // базовые методы итератора
    virtual void Next(void);
    virtual void Reset(void);
    virtual T& Data(void);

    // подготовить итератор для сканирования другой таблицы
    void SetList(HashTable<T>& lst);
};
```

ОПИСАНИЕ

Метод Next выполняет прохождение таблицы список (блок) за списком, проходя узлы каждого списка. Значения данных, выдаваемые итератором, никак не упорядочены. Для обнаружения очередного списка, подлежащего прохождению, метод Next использует функцию SearchNextNode.

ПРИМЕР

```
// объявить итератор для объекта HF типа HashTable
HashTableIterator<NameRecord> hiter(HF);

// сканировать все элементы базы данных
for (hiter.Reset(); !hiter.EndOfList; hiter.Next())
{
  rec = hiter.Data();
  cout << rec.name << ": " << rec.count << endl;
}
```

Приложение: частота символьных строк

Класс HashTable используется для хранения множества символьных строк и определения частоты их появления в файле. Каждая символьная строка хранится в объекте типа NameRecord, содержащем наименование строки и частоту ее повторяемости.

```
struct NameRecord
{
  String name;
  int count;
};
```

Хеш-функция перемешивает биты символов строки путем сдвига текущего значения функции на три бита влево (умножая на 8) перед тем, как прибавить следующий символ. Для n-символьной строки $c_0c_1\ldots c_{n-2}c_{n-1}$

$$\text{hash}(s) = \sum_{i=1}^{n-1} c_i \, 8^{n-i-1}$$

Такое вычисление предотвращает проблемы хеширования символьных строк, рассмотренные в примере 14.2.

```
// функция для использования в классе HashTable
unsigned long hash (NameRecord elem)
```

```
{
  unsigned long hashval = 0;

  // сдвинуть hashval на три бита влево и
  // сложить со следующим символом
  for (int i=0; i<elem.Length(); i++)
    hashval = (hashval << 3) + elem.name[i];
  return hashval;
}
```

Программа 14.2. Вычисление частот символьных строк

Эта программа вводит символьные строки из файла strings.dat и запоминает их в 101-элементной таблице. Каждая символьная строка вводится из файла и, если еще не встречалась ранее, помещается в таблицу. Для дублирующихся строк из хеш-таблицы выбирается соответствующая запись, где и производится увеличение на единицу поля счетчика. В конце программы определяется итератор, который используется для просмотра и печати всей таблицы. Определения NameRecord, хеш-функции и оператора == для данных типа NameRecord находятся в файле strfreq.h.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

#include "hash.h"
#include "strclass.h"
#include "strfreq.h"

void main(void)
{
  // ввести символьные строки из входного потока
  ifstream fin;
  NameRecord rec;
  String token;
  HashTable<NameRecord> HF(101, hash);

  fin.open("strings.dat"), ios::in | ios::nocreate);
  if (!fin)
  {
    cerr << "Невозможно открыть \"strings.dat\"!" << endl;
    exit(1);
  }

  while (fin >> rec.name)
  {
    // искать строку в таблице. если найдена, обновить поле count
    if (HF.Find(rec))
    {
      rec.count += 1;
      HF.Update(rec);
    }
    else
    {
      rec.count = 1;
      HF.Insert(rec);
    }
  }
```

```
    // печатать символьные строки вместе с частотами
    HashTableIterator<NameRecord> hiter(HF);

    for(hiter.Reset(); !hiter.EndOfList(); hiter.Next())
    {
      rec = hiter.Data();
      cout << rec.name << ": " << rec.count << endl;
    }
  }
  /*
  <Файл strings.dat>

  Columbus Washington Napoleon Washington Lee Grant
  Washington Lincoln Grant Columbus Washington

  <Прогон программы 14.2>

  Lee: 1
  Washington: 4
  Lincoln: 1
  Napoleon: 1 Grant: 2
  Columbus: 2
  */
```

Реализация класса HashTable

Данный класс образован от абстрактного класса List, предоставляющего методы ListSize и ListEmpty. Мы обсудим элементы данных класса HashTable и операции, реализующие чистые виртуальные функции Insert, Find, Delete и ClearList.

Ключевым элементом данных класса является объект buckets типа Array, который определяет массив связанных списков, образующих хеш-таблицу. Указатель функции hf определяет хеш-функцию, а numBuckets является размером таблицы. Указатель current идентифицирует последний элемент данных, к которому осуществлялся доступ тем или иным методом класса. Его значение задается методами Find и Insert и используется методом Update для обновления данных в таблице.

Методы обработки списков. Метод Insert вычисляет значение хеш-функции (индекс блока) и ищет объект типа LinkedList, чтобы проверить, есть ли уже такой элемент данных в таблице или нет. Если есть, то Insert обновляет этот элемент данных, устанавливает на него указатель current и возвращает управление. Если такого элемента в таблице нет, Insert добавляет его в хвост списка, устанавливает на него указатель current и увеличивает размер списка.

```
template <class T>
void HashTable<T>::Insert(const T& key)
{
  // hashval - индекс блока (связанного списка)
  int hashval = int(hf(key) % numBuckets);

  // lst - псевдоним для buckets[hashval].
  // помогает обойтись без индексов
  LinkedList<T>& lst = buckets[hashval];

  for (lst.Reset(); !lst.EndOfList(); lst.Next())
    // если ключ совпал, обновить данные и выйти
    if (lst.Data() == key)
```

```
    {
      lst.Data() = key;
      current = &lst.Data();
      return;
    }
  // данные, соответствующие этому ключу, не найдены. вставить элемент в список
  lst.InsertRear(key);
  current = &lst.Data();
  size++;
}
```

Метод Find применяет хеш-функцию и просматривает указанный в результате список на предмет совпадения с входным параметром. Если совпадение обнаружено, метод копирует данные в key, устанавливает указатель current на соответствующий узел и возвращает True. В противном случает метод возвращает False.

```
template <class T>
int HashTable<T>::Find(T& key)
{
  // вычислить значение хеш-функции и установить lst
  // на начало соответствующего связанного списка
  int hashval = int(hf(key) % NumBuckets);
  LinkedList<T>& lst = buckets[hashval];

  // просматривать узлы связанного списка в поисках key
  for (lst.Reset(); !lst.EndOfList(); lst.Next())
    // если ключ совпал, получить данные, установить current и выйти
    if (lst.Data() == key)
    {
      key = lst.Data();
      current = &lst.Data();
      return 1;                  // вернуть True
    }
  return 0;                      // иначе вернуть False
}
```

Метод Delete просматривает указанный список и удаляет узел, если совпадение произошло. Этот метод (вместе с методами ClearList и Update) находится в файле hash.h.

Реализация класса HashTableIterator

Этот класс должен просматривать данные, разбросанные по хеш-таблице. Поэтому он более интересен и более сложен с точки зрения реализации, чем класс HashTable. Обход элементов таблицы начинается с поиска непустого блока в массиве списков. Обнаружив непустой блок, мы просматриваем все узлы этого списка, а затем продолжаем процесс, взяв другой непустой блок. Итератор заканчивает обход, когда просмотрен последний непустой блок.

Итератор должен быть привязан к списку. В данном случае переменной hashTable присваивается адрес таблицы. Поскольку класс HashTableIterator является дружественным по отношению к HashTable, он имеет доступ ко всем закрытым данным-членам последнего, включая массив buckets и его размер numBuckets. Переменная currentBucket является индексом связанного списка, который просматривается в данный момент, а currBucketPtr — указателем этого списка. Прохождение каждого блока осуществляется итератором, встроенным в класс LinkedList. На рис. 14.4 показано, как итератор проходит таблицу с четырьмя элементами.

Метод SearchNextNode вызывается для обнаружения очередного списка, подлежащего прохождению. Просматриваются все блоки, начиная с cb, пока

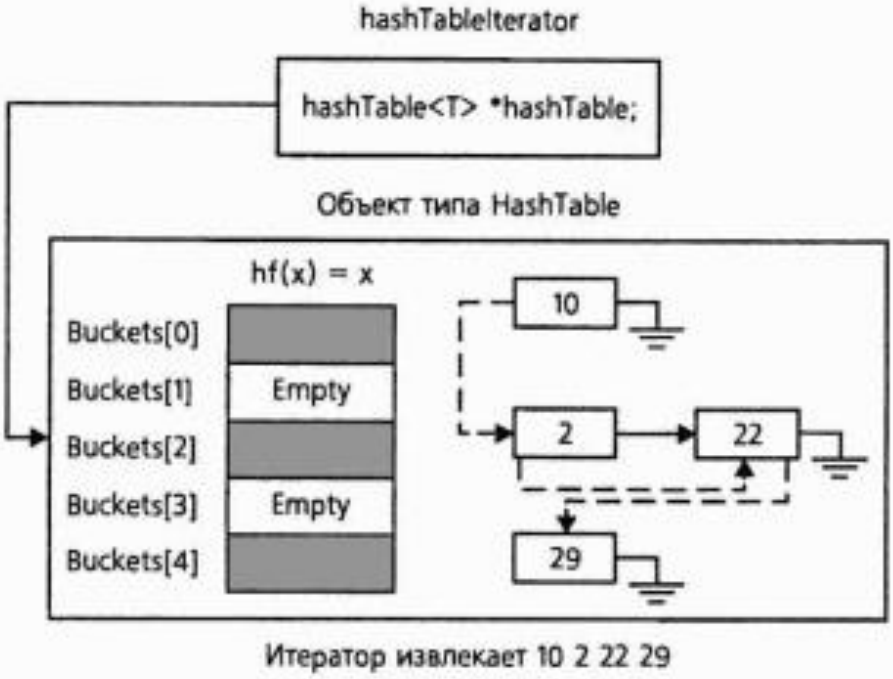


Рис. 14.4. Итератор хэш-таблиц

не встретится непустой список. Переменной currentBucket присваивается индекс этого списка, а переменной currBucketPtr — его адрес. Если непустых списков нет, происходит возврат с currentBucket = −1.

```
// начиная с cb, искать следующий непустой список для просмотра
template <class t>
void HashTableIterator<T>::SearchNextNode(int cb)
{
  currentBucket = -1;

  // если индекс cb больше размера таблицы, прекратить поиск
  if (cb > hashTable->numBuckets)
    return;

  // иначе искать, начиная с текущего списка до конца таблицы,
  // непустой блок и обновить частные элементы данных
  for (int i=cb; i<hashTable->numBuckets; i++)
    if (!hashTable->buckets[i].ListEmpty())
    {
      // перед тем как вернуться, установить currentBucket равным i
      // и в currBucketPtr поместить адрес нового непустого списка
      currBucketPtr = &hashTable->buckets[i];
      currBucketPtr ->Reset();
      currentBucket = i;
      return;
    }
}
```

Конструктор инициализирует базовый класс Iterator и присваивает закрытому указателю hashTable адрес таблицы. Непустой список обнаруживается с помощью вызова SearchNextNode с нулевым параметром.

```
// конструктор. инициализирует базовый класс и класс HashTable
// SearchNextNode идентифицирует первый непустой блок в таблице
template <class T>
HashTableIterator<T>::HashTableIterator(HashTable<T>& hf):
        Iterator<T>(hf), HashTable(&hf)
{
  SearchNextNode(0);
}
```

С помощью метода Next осуществляется продвижение вперед по текущему списку на один элемент. По достижении конца списка функция SearchNextNode настраивает итератор на следующий непустой блок.

```
// перейти к следующему элементу данных в таблице
template <class T>
void HashTableIterator<T>::Next(void)
{
  // продвинуться к следующему узлу текущего списка
  currBucketPtr->Next();

  // по достижении конца списка вызвать SearchNextNode
  // для поиска следующего непустого блока в таблице
  if (currBucketPtr->EndOfList())
    SearchNextNode(++currentBucket);

  // установить флажок iterationComplete, если непустых списков
  // больше нет
  iterationComplete = currentBucket == -1;
}
```

Источники:

1. У. Топп, У. Форд – Структуры данных в C++
2. https://habr.com/ru/post/534596/ - про хэш-функции
3. https://otus.ru/nest/post/1835/ - хэширование простыми словами
4. https://blog.skillfactory.ru/glossary/heshirovanie/ - основы про хэш