CHAPTER 5

Hashing

In Chapter 4 we discussed the search tree ADT, which allowed various operations on a set of elements. In this chapter, we discuss the *hash table* ADT, which supports only a subset of the operations allowed by binary search trees.

The implementation of hash tables is frequently called **hashing**. Hashing is a technique used for performing insertions, deletions, and finds in constant average time. Tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations such as `findMin`, `findMax`, and the printing of the entire table in sorted order in linear time are not supported.

The central data structure in this chapter is the **hash table**. We will . . .

- See several methods of implementing the hash table.
- Compare these methods analytically.
- Show numerous applications of hashing.
- Compare hash tables with binary search trees.

5.1 General Idea

The ideal hash table data structure is merely an array of some fixed size containing the items. As discussed in Chapter 4, generally a search is performed on some part (that is, data member) of the item. This is called the **key.** For instance, an item could consist of a string (that serves as the key) and additional data members (for instance, a name that is part of a large employee structure). We will refer to the table size as *TableSize,* with the understanding that this is part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to *TableSize* − 1; we will see why shortly.

Each key is mapped into some number in the range 0 to *TableSize* − 1 and placed in the appropriate cell. The mapping is called a **hash function,** which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a hash function that distributes the keys evenly among the cells. Figure 5.1 is typical of a perfect situation. In this example, *john* hashes to 3, *phil* hashes to 4, *dave* hashes to 6, and *mary* hashes to 7.

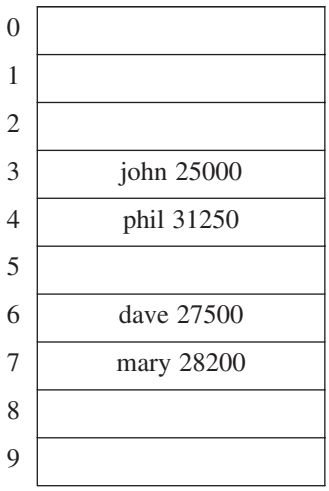


Figure 5.1 An ideal hash table

This is the basic idea of hashing. The only remaining problems deal with choosing a function, deciding what to do when two keys hash to the same value (this is known as a **collision**), and deciding on the table size.

5.2 Hash Function

If the input keys are integers, then simply returning *Key* mod *TableSize* is generally a reasonable strategy, unless *Key* happens to have some undesirable properties. In this case, the choice of hash function needs to be carefully considered. For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is a bad choice. For reasons we shall see later, and to avoid situations like the one above, it is often a good idea to ensure that the table size is prime. When the input keys are random integers, then this function is not only very simple to compute but also distributes the keys evenly.

Usually, the keys are strings; in this case, the hash function needs to be chosen carefully.

One option is to add up the ASCII values of the characters in the string. The routine in Figure 5.2 implements this strategy.

The hash function depicted in Figure 5.2 is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute the keys well. For instance, suppose that *TableSize* = 10,007 (10,007 is a prime number). Suppose all the keys are eight or fewer characters long. Since an ASCII character has an integer value that is always at most 127, the hash function typically can only assume values between 0 and 1,016, which is $127 * 8$. This is clearly not an equitable distribution!

Another hash function is shown in Figure 5.3. This hash function assumes that *Key* has at least three characters. The value 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2. This function examines only the first three characters, but if these are random and the table size is 10,007, as before, then we would expect a

```
1  int hash( const string & key, int tableSize )
2  {
3      int hashVal = 0;
4
5      for( char ch : key )
6          hashVal += ch;
7
8      return hashVal % tableSize;
9  }
```

Figure 5.2 A simple hash function

```
1  int hash( const string & key, int tableSize )
2  {
3      return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
4  }
```

Figure 5.3 Another possible hash function—not too good

reasonably equitable distribution. Unfortunately, English is not random. Although there are $26^3 = 17{,}576$ possible combinations of three characters (ignoring blanks), a check of a reasonably large online dictionary reveals that the number of different combinations is actually only 2,851. Even if none of *these* combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.

Figure 5.4 shows a third attempt at a hash function. This hash function involves all characters in the key and can generally be expected to distribute well (it computes $\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$ and brings the result into proper range). The code computes a polynomial function (of 37) by use of Horner's rule. For instance, another way of computing $h_k = k_0 + 37k_1 + 37^2k_2$ is by the formula $h_k = ((k_2) * 37 + k_1) * 37 + k_0$. Horner's rule extends this to an nth degree polynomial.

```
1  /**
2   * A hash routine for string objects.
3   */
4  unsigned int hash( const string & key, int tableSize )
5  {
6      unsigned int hashVal = 0;
7
8      for( char ch : key )
9          hashVal = 37 * hashVal + ch;
10
11     return hashVal % tableSize;
12 }
```

Figure 5.4 A good hash function

The hash function takes advantage of the fact that overflow is allowed and uses `unsigned int` to avoid introducing a negative number.

The hash function described in Figure 5.4 is not necessarily the best with respect to table distribution, but it does have the merit of extreme simplicity and is reasonably fast. If the keys are very long, the hash function will take too long to compute. A common practice in this case is not to use all the characters. The length and properties of the keys would then influence the choice. For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code. Some programmers implement their hash function by using only the characters in the odd spaces, with the idea that the time saved computing the hash function will make up for a slightly less evenly distributed function.

The main programming detail left is collision resolution. If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it. There are several methods for dealing with this. We will discuss two of the simplest: separate chaining and open addressing; then we will look at some more recently discovered alternatives.

5.3 Separate Chaining

The first strategy, commonly known as **separate chaining,** is to keep a list of all elements that hash to the same value. We can use the Standard Library list implementation. If space is tight, it might be preferable to avoid their use (since these lists are doubly linked and waste space). We assume for this section that the keys are the first 10 perfect squares and that the hashing function is simply $hash(x) = x \bmod 10$. (The table size is not prime but is used here for simplicity.) Figure 5.5 shows the resulting separate chaining hash table.

To perform a `search`, we use the hash function to determine which list to traverse. We then search the appropriate list. To perform an `insert`, we check the appropriate list to see whether the element is already in place (if duplicates are expected, an extra data member is

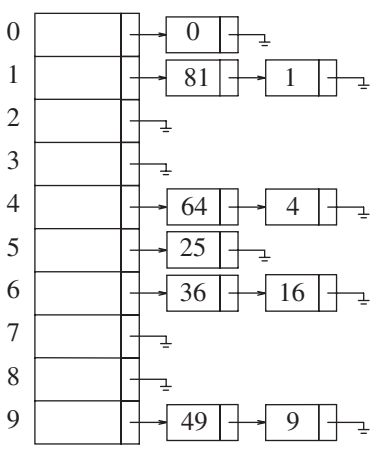


Figure 5.5 A separate chaining hash table

```
1   template <typename HashedObj>
2   class HashTable
3   {
4     public:
5       explicit HashTable( int size = 101 );
6
7       bool contains( const HashedObj & x ) const;
8
9       void makeEmpty( );
10      bool insert( const HashedObj & x );
11      bool insert( HashedObj && x );
12      bool remove( const HashedObj & x );
13
14    private:
15      vector<list<HashedObj>> theLists;     // The array of Lists
16      int currentSize;
17
18      void rehash( );
19      size_t myhash( const HashedObj & x ) const;
20  };
```

Figure 5.6 Type declaration for separate chaining hash table

usually kept, and this data member would be incremented in the event of a match). If the element turns out to be new, it can be inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

The class interface for a separate chaining implementation is shown in Figure 5.6. The hash table stores an array of linked lists, which are allocated in the constructor.

The class interface illustrates a syntax point: Prior to C++11, in the declaration of `theLists`, a space was required between the two `>`s; since `>>` is a C++ token, and because it is longer than `>`, `>>` would be recognized as the token. In C++11, this is no longer the case.

Just as the binary search tree works only for objects that are `Comparable`, the hash tables in this chapter work only for objects that provide a hash function and equality operators (`operator==` or `operator!=`, or possibly both).

Instead of requiring hash functions that take both the object and the table size as parameters, we have our hash functions take only the object as the parameter and return an appropriate integral type. The standard mechanism for doing this uses function objects, and the protocol for hash tables was introduced in C++11. Specifically, in C++11, hash functions can be expressed by the function object template:

```
template <typename Key>
class hash
{
  public:
    size_t operator() ( const Key & k ) const;
};
```

Default implementations of this template are provided for standard types such as `int` and `string`; thus, the hash function described in Figure 5.4 could be implemented as

```
template <>
class hash<string>
{
  public:
    size_t operator()( const string & key )
    {
        size_t hashVal = 0;

        for( char ch : key )
            hashVal = 37 * hashVal + ch;

        return hashVal;
    }
};
```

The type `size_t` is an unsigned integral type that represents the size of an object; therefore, it is guaranteed to be able to store an array index. A class that implements a hash table algorithm can then use calls to the generic hash function object to generate an integral type `size_t` and then scale the result into a suitable array index. In our hash tables, this is manifested in private member function `myhash`, shown in Figure 5.7.

Figure 5.8 illustrates an `Employee` class that can be stored in the generic hash table, using the `name` member as the key. The `Employee` class implements the `HashedObj` requirements by providing equality operators and a hash function object.

The code to implement `makeEmpty`, `contains`, and `remove` is shown in Figure 5.9.

Next comes the insertion routine. If the item to be inserted is already present, then we do nothing; otherwise, we place it in the list (see Fig. 5.10). The element can be placed anywhere in the list; using `push_back` is most convenient in our case. `whichList` is a reference variable; see Section 1.5.2 for a discussion of this use of reference variables.

Any scheme could be used besides linked lists to resolve the collisions; a binary search tree or even another hash table would work, but we expect that if the table is large and the hash function is good, all the lists should be short, so basic separate chaining makes no attempt to try anything complicated.

We define the load factor, λ, of a hash table to be the ratio of the number of elements in the hash table to the table size. In the example above, $\lambda = 1.0$. The average length of a list is λ. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list. In an unsuccessful search, the number

```
1       size_t myhash( const HashedObj & x ) const
2       {
3           static hash<HashedObj> hf;
4           return hf( x ) % theLists.size( );
5       }
```

Figure 5.7 `myHash` member function for hash tables

```
1  // Example of an Employee class
2  class Employee
3  {
4    public:
5      const string & getName( ) const
6        { return name; }
7
8      bool operator==( const Employee & rhs ) const
9        { return getName( ) == rhs.getName( ); }
10     bool operator!=( const Employee & rhs ) const
11       { return !( *this == rhs; }
12
13       // Additional public members not shown
14
15   private:
16     string name;
17     double salary;
18     int    seniority;
19
20       // Additional private members not shown
21 };
22
23 template<>
24 class hash<Employee>
25 {
26   public:
27     size_t operator()( const Employee & item )
28     {
29         static hash<string> hf;
30         return hf( item.getName( ) );
31     }
32 };
```

Figure 5.8 Example of a class that can be used as a `HashedObj`

of nodes to examine is λ on average. A successful search requires that about $1 + (\lambda/2)$ links be traversed. To see this, notice that the list that is being searched contains the one node that stores the match plus zero or more other nodes. The expected number of "other nodes" in a table of N elements and M lists is $(N-1)/M = \lambda - 1/M$, which is essentially λ, since M is presumed large. On average, half the "other nodes" are searched, so combined with the matching node, we obtain an average search cost of $1 + \lambda/2$ nodes. This analysis shows that the table size is not really important but the load factor is. The general rule for separate chaining hashing is to make the table size about as large as the number of elements expected (in other words, let $\lambda \approx 1$). In the code in Figure 5.10, if the load factor exceeds 1, we expand the table size by calling `rehash` at line 10. `rehash` is discussed in Section 5.5. It is also a good idea, as mentioned before, to keep the table size prime to ensure a good distribution.

```
1       void makeEmpty( )
2       {
3           for( auto & thisList : theLists )
4               thisList.clear( );
5       }
6
7       bool contains( const HashedObj & x ) const
8       {
9           auto & whichList = theLists[ myhash( x ) ];
10          return find( begin( whichList ), end( whichList ), x ) != end( whichList );
11      }
12
13      bool remove( const HashedObj & x )
14      {
15          auto & whichList = theLists[ myhash( x ) ];
16          auto itr = find( begin( whichList ), end( whichList ), x );
17
18          if( itr == end( whichList ) )
19              return false;
20
21          whichList.erase( itr );
22          --currentSize;
23          return true;
24      }
```

Figure 5.9 `makeEmpty`, `contains`, and `remove` routines for separate chaining hash table

```
1       bool insert( const HashedObj & x )
2       {
3           auto & whichList = theLists[ myhash( x ) ];
4           if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
5               return false;
6           whichList.push_back( x );
7
8               // Rehash; see Section 5.5
9           if( ++currentSize > theLists.size( ) )
10              rehash( );
11
12          return true;
13      }
```

Figure 5.10 `insert` routine for separate chaining hash table

5.4 Hash Tables without Linked Lists

Separate chaining hashing has the disadvantage of using linked lists. This could slow the algorithm down a bit because of the time required to allocate new cells (especially in other languages) and essentially requires the implementation of a second data structure. An alternative to resolving collisions with linked lists is to try alternative cells until an empty cell is found. More formally, cells $h_0(x), h_1(x), h_2(x), \ldots$ are tried in succession, where $h_i(x) = (hash(x) + f(i)) \mod TableSize$, with $f(0) = 0$. The function, f, is the collision resolution strategy. Because all the data go inside the table, a bigger table is needed in such a scheme than for separate chaining hashing. Generally, the load factor should be below $\lambda = 0.5$ for a hash table that doesn't use separate chaining. We call such tables **probing hash tables**. We now look at three common collision resolution strategies.

5.4.1 Linear Probing

In linear probing, f is a linear function of i, typically $f(i) = i$. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Figure 5.11 shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, $f(i) = i$.

The first collision occurs when 49 is inserted; it is put in the next available spot, namely, spot 0, which is open. The key 58 collides with 18, 89, and then 49 before an empty cell is found three away. The collision for 69 is handled in a similar manner. As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect, known as **primary clustering,** means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Although we will not perform the calculations here, it can be shown that the expected number of probes using linear probing is roughly $\frac{1}{2}(1 + 1/(1 - \lambda)^2)$ for insertions and

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

unsuccessful searches, and $\frac{1}{2}(1 + 1/(1 - \lambda))$ for successful searches. The calculations are somewhat involved. It is easy to see from the code that insertions and unsuccessful searches require the same number of probes. A moment's thought suggests that, on average, successful searches should take less time than unsuccessful searches.

The corresponding formulas, if clustering is not a problem, are fairly easy to derive. We will assume a very large table and that each probe is independent of the previous probes. These assumptions are satisfied by a *random* collision resolution strategy and are reasonable unless λ is very close to 1. First, we derive the expected number of probes in an unsuccessful search. This is just the expected number of probes until we find an empty cell. Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $1/(1 - \lambda)$. The number of probes for a successful search is equal to the number of probes required when the particular element was inserted. When an element is inserted, it is done as a result of an unsuccessful search. Thus, we can use the cost of an unsuccessful search to compute the average cost of a successful search.

The caveat is that λ changes from 0 to its current value, so that earlier insertions are cheaper and should bring the average down. For instance, in the table in Figure 5.11, $\lambda = 0.5$, but the cost of accessing 18 is determined when 18 is inserted. At that point, $\lambda = 0.2$. Since 18 was inserted into a relatively empty table, accessing it should be easier than accessing a recently inserted element, such as 69. We can estimate the average by using an integral to calculate the mean value of the insertion time, obtaining

$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}dx = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

These formulas are clearly better than the corresponding formulas for linear probing. Clustering is not only a theoretical problem but actually occurs in real implementations.

Figure 5.12 compares the performance of linear probing (dashed curves) with what would be expected from more random collision resolution. Successful searches are indicated by an *S,* and unsuccessful searches and insertions are marked with *U* and *I,* respectively.

If $\lambda = 0.75$, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing. If $\lambda = 0.9$, then 50 probes are expected, which is unreasonable. This compares with 4 and 10 probes for the respective load factors if clustering were not a problem. We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full. If $\lambda = 0.5$, however, only 2.5 probes are required on average for insertion, and only 1.5 probes are required, on average, for a successful search.

5.4.2 Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect—the collision function is quadratic. The popular choice is $f(i) = i^2$. Figure 5.13 shows the resulting hash table with this collision function on the same input used in the linear probing example.

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next, 58 collides at position 8. Then the cell one away is

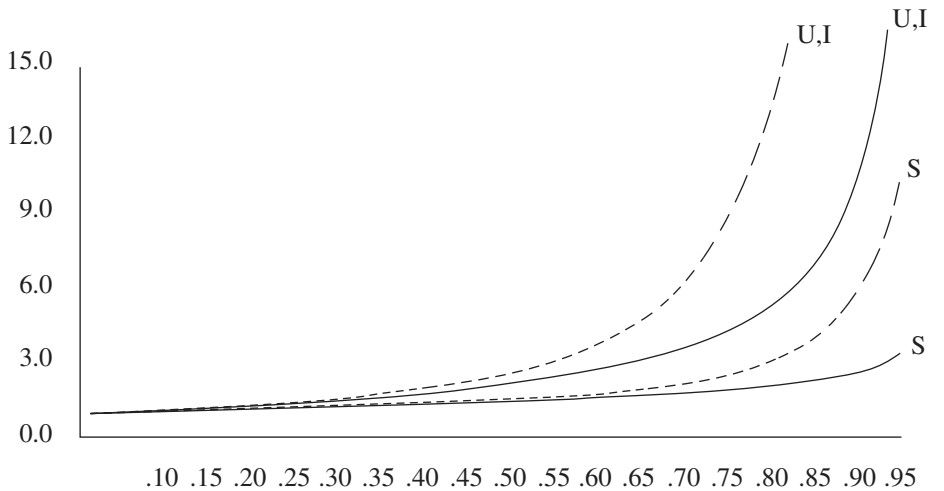


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Hash table with quadratic probing, after each insertion

tried, but another collision occurs. A vacant cell is found at the next cell tried, which is $2^2 = 4$ away. 58 is thus placed in cell 2. The same thing happens for 69.

For linear probing, it is a bad idea to let the hash table get nearly full, because performance degrades. For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

Theorem 5.1

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Proof

Let the table size, *TableSize,* be an (odd) prime greater than 3. We show that the first $\lceil TableSize/2 \rceil$ alternative locations (including the initial location $h_0(x)$) are all distinct. Two of these locations are $h(x) + i^2 \pmod{TableSize}$ and $h(x) + j^2 \pmod{TableSize}$, where $0 \leq i,j \leq \lfloor TableSize/2 \rfloor$. Suppose, for the sake of contradiction, that these locations are the same, but $i \neq j$. Then

$$\begin{aligned} h(x) + i^2 &= h(x) + j^2 \pmod{TableSize} \\ i^2 &= j^2 \pmod{TableSize} \\ i^2 - j^2 &= 0 \pmod{TableSize} \\ (i-j)(i+j) &= 0 \pmod{TableSize} \end{aligned}$$

Since *TableSize* is prime, it follows that either $(i - j)$ or $(i + j)$ is equal to 0 (mod *TableSize*). Since i and j are distinct, the first option is not possible. Since $0 \leq i,j \leq \lfloor TableSize/2 \rfloor$, the second option is also impossible. Thus, the first $\lceil TableSize/2 \rceil$ alternative locations are distinct. If at most $\lfloor TableSize/2 \rfloor$ positions are taken, then an empty spot can always be found.

If the table is even one more than half full, the insertion could fail (although this is extremely unlikely). Therefore, it is important to keep this in mind. It is also crucial that the table size be prime.[1] If the table size is not prime, the number of alternative locations can be severely reduced. As an example, if the table size were 16, then the only alternative locations would be at distances 1, 4, or 9 away.

Standard deletion cannot be performed in a probing hash table, because the cell might have caused a collision to go past it. For instance, if we remove 89, then virtually all the remaining `find` operations will fail. Thus, probing hash tables require lazy deletion, although in this case there really is no laziness implied.

The class interface required to implement probing hash tables is shown in Figure 5.14. Instead of an array of lists, we have an array of hash table entry cells. The nested class `HashEntry` stores the state of an entry in the `info` member; this state is either `ACTIVE`, `EMPTY`, or `DELETED`.

We use a standard enumerated type.

```
enum EntryType { ACTIVE, EMPTY, DELETED };
```

Constructing the table (Fig. 5.15) consists of setting the `info` member to `EMPTY` for each cell. `contains(x)`, shown in Figure 5.16, invokes private member functions `isActive` and `findPos`. The `private` member function `findPos` performs the collision resolution. We ensure in the `insert` routine that the hash table is at least twice as large as the number of elements in the table, so quadratic resolution will always work. In the implementation

[1] If the table size is a prime of the form $4k + 3$, and the quadratic collision resolution strategy $f(i) = \pm i^2$ is used, then the entire table can be probed. The cost is a slightly more complicated routine.

```
1  template <typename HashedObj>
2  class HashTable
3  {
4    public:
5      explicit HashTable( int size = 101 );
6
7      bool contains( const HashedObj & x ) const;
8
9      void makeEmpty( );
10     bool insert( const HashedObj & x );
11     bool insert( HashedObj && x );
12     bool remove( const HashedObj & x );
13
14     enum EntryType { ACTIVE, EMPTY, DELETED };
15
16   private:
17     struct HashEntry
18     {
19         HashedObj element;
20         EntryType info;
21
22         HashEntry( const HashedObj & e = HashedObj{ }, EntryType i = EMPTY )
23           : element{ e }, info{ i } { }
24         HashEntry( HashedObj && e, EntryType i = EMPTY )
25           : element{ std::move( e ) }, info{ i } { }
26     };
27
28     vector<HashEntry> array;
29     int currentSize;
30
31     bool isActive( int currentPos ) const;
32     int findPos( const HashedObj & x ) const;
33     void rehash( );
34     size_t myhash( const HashedObj & x ) const;
35 };
```

Figure 5.14 Class interface for hash tables using probing strategies, including the nested `HashEntry` class

in Figure 5.16, elements that are marked as deleted count as being in the table. This can cause problems, because the table can get too full prematurely. We shall discuss this item presently.

Lines 12 to 15 represent the fast way of doing quadratic resolution. From the definition of the quadratic resolution function, $f(i) = f(i-1) + 2i - 1$, so the next cell to try is a distance from the previous cell tried and this distance increases by 2 on successive probes.

```
1       explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2         { makeEmpty( ); }
3
4       void makeEmpty( )
5       {
6           currentSize = 0;
7           for( auto & entry : array )
8               entry.info = EMPTY;
9       }
```

Figure 5.15 Routines to initialize quadratic probing hash table

```
1       bool contains( const HashedObj & x ) const
2         { return isActive( findPos( x ) ); }
3
4       int findPos( const HashedObj & x ) const
5       {
6           int offset = 1;
7           int currentPos = myhash( x );
8
9           while( array[ currentPos ].info != EMPTY &&
10                 array[ currentPos ].element != x )
11          {
12              currentPos += offset;  // Compute ith probe
13              offset += 2;
14              if( currentPos >= array.size( ) )
15                  currentPos -= array.size( );
16          }
17
18          return currentPos;
19      }
20
21      bool isActive( int currentPos ) const
22        { return array[ currentPos ].info == ACTIVE; }
```

Figure 5.16 `contains` routine (and `private` helpers) for hashing with quadratic probing

If the new location is past the array, it can be put back in range by subtracting *TableSize*. This is faster than the obvious method, because it avoids the multiplication and division that seem to be required. An important warning: The order of testing at lines 9 and 10 is important. Don't switch it!

The final routine is insertion. As with separate chaining hashing, we do nothing if `x` is already present. It is a simple modification to do something else. Otherwise, we place it at the spot suggested by the `findPos` routine. The code is shown in Figure 5.17. If the load

```
1       bool insert( const HashedObj & x )
2       {
3               // Insert x as active
4           int currentPos = findPos( x );
5           if( isActive( currentPos ) )
6               return false;
7
8           array[ currentPos ].element = x;
9           array[ currentPos ].info = ACTIVE;
10
11              // Rehash; see Section 5.5
12          if( ++currentSize > array.size( ) / 2 )
13              rehash( );
14
15          return true;
16      }
17
18      bool remove( const HashedObj & x )
19      {
20          int currentPos = findPos( x );
21          if( !isActive( currentPos ) )
22              return false;
23
24          array[ currentPos ].info = DELETED;
25          return true;
26      }
```

Figure 5.17 Some `insert` and `remove` routines for hash tables with quadratic probing

factor exceeds 0.5, the table is full and we enlarge the hash table. This is called rehashing and is discussed in Section 5.5. Figure 5.17 also shows `remove`.

Although quadratic probing eliminates primary clustering, elements that hash to the same position will probe the same alternative cells. This is known as **secondary clustering**. Secondary clustering is a slight theoretical blemish. Simulation results suggest that it generally causes less than an extra half probe per search. The following technique eliminates this, but does so at the cost of computing an extra hash function.

5.4.3 Double Hashing

The last collision resolution method we will examine is **double hashing**. For double hashing, one popular choice is $f(i) = i \cdot hash_2(x)$. This formula says that we apply a second hash function to x and probe at a distance $hash_2(x), 2hash_2(x), \ldots$, and so on. A poor choice of $hash_2(x)$ would be disastrous. For instance, the obvious choice $hash_2(x) = x \bmod 9$ would not help if 99 were inserted into the input in the previous examples. Thus, the function must never evaluate to zero. It is also important to make sure all cells can be probed (this is not possible in the example below, because the table size is not prime). A function such

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.18 Hash table with double hashing, after each insertion

as $hash_2(x) = R - (x \bmod R)$, with R a prime smaller than *TableSize,* will work well. If we choose $R = 7$, then Figure 5.18 shows the results of inserting the same keys as before.

The first collision occurs when 49 is inserted. $hash_2(49) = 7 - 0 = 7$, so 49 is inserted in position 6. $hash_2(58) = 7 - 2 = 5$, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance $hash_2(69) = 7 - 6 = 1$ away. If we tried to insert 60 in position 0, we would have a collision. Since $hash_2(60) = 7 - 4 = 3$, we would then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here.

As we have said before, the size of our sample hash table is not prime. We have done this for convenience in computing the hash function, but it is worth seeing why it is important to make sure the table size is prime when double hashing is used. If we attempt to insert 23 into the table, it would collide with 58. Since $hash_2(23) = 7 - 2 = 5$, and the table size is 10, we essentially have only one alternative location, and it is already taken. Thus, if the table size is not prime, it is possible to run out of alternative locations prematurely. However, if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. This makes double hashing theoretically interesting. Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice, especially for keys like strings whose hash functions are expensive to compute.

5.5 Rehashing

If the table gets too full, the running time for the operations will start taking too long, and insertions might fail for open addressing hashing with quadratic resolution. This can happen if there are too many removals intermixed with insertions. A solution, then, is to build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (nondeleted) element and inserting it in the new table.

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.19 Hash table with linear probing with input 13, 15, 6, 24

As an example, suppose the elements 13, 15, 24, and 6 are inserted into a linear probing hash table of size 7. The hash function is $h(x) = x \mod 7$. The resulting hash table appears in Figure 5.19.

If 23 is inserted into the table, the resulting table in Figure 5.20 will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime that is twice as large as the old table size. The new hash function is then $h(x) = x \mod 17$. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure 5.21.

This entire operation is called **rehashing.** This is obviously a very expensive operation; the running time is $O(N)$, since there are N elements to rehash and the table size is roughly $2N$, but it is actually not all that bad, because it happens very infrequently. In particular, there must have been $N/2$ insertions prior to the last rehash, so it essentially adds a constant cost to each insertion. This is why the new table is made twice as large as the old table. If this data structure is part of the program, the effect is not noticeable. On the other hand, if the hashing is performed as part of an interactive system, then the unfortunate user whose insertion caused a rehash could see a slowdown.

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 5.20 Hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 5.21 Hash table after rehashing

Rehashing can be implemented in several ways with quadratic probing. One alternative is to rehash as soon as the table is half full. The other extreme is to rehash only when an insertion fails. A third, middle-of-the-road strategy is to rehash when the table reaches a certain load factor. Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

Rehashing for separate chaining hash tables is similar. Figure 5.22 shows that rehashing is simple to implement and provides an implementation for separate chaining rehashing.

5.6 Hash Tables in the Standard Library

In C++11, the Standard Library includes hash table implementations of sets and maps—namely, `unordered_set` and `unordered_map`, which parallel `set` and `map`. The items in the `ordered_set` (or the keys in the `unordered_map`) must provide an overloaded `operator==` and a `hash` function, as described earlier, in Section 5.3. Just as the `set` and `map` templates can

```
1       /**
2        * Rehashing for quadratic probing hash table.
3        */
4       void rehash( )
5       {
6           vector<HashEntry> oldArray = array;
7
8               // Create new double-sized, empty table
9           array.resize( nextPrime( 2 * oldArray.size( ) ) );
10          for( auto & entry : array )
11              entry.info = EMPTY;
12
13              // Copy table over
14          currentSize = 0;
15          for( auto & entry : oldArray )
16              if( entry.info == ACTIVE )
17                  insert( std::move( entry.element ) );
18      }
19
20      /**
21       * Rehashing for separate chaining hash table.
22       */
23      void rehash( )
24      {
25          vector<list<HashedObj>> oldLists = theLists;
26
27              // Create new double-sized, empty table
28          theLists.resize( nextPrime( 2 * theLists.size( ) ) );
29          for( auto & thisList : theLists )
30              thisList.clear( );
31
32              // Copy table over
33          currentSize = 0;
34          for( auto & thisList : oldLists )
35              for( auto & x : thisList )
36                  insert( std::move( x ) );
37      }
```

Figure 5.22 Rehashing for both separate chaining hash tables and probing hash tables

also be instantiated with a function object that provides (or overrides a default) comparison function, `unordered_set` and `unordered_map` can be instantiated with function objects that provide a hash function and equality operator. Thus, for example, Figure 5.23 illustrates how an unordered set of *case-insensitive* strings can be maintained, assuming that some string operations are implemented elsewhere.

```
1  class CaseInsensitiveStringHash
2  {
3    public:
4      size_t operator( ) ( const string & s ) const
5      {
6          static hash<string> hf;
7          return hf( toLower( s ) );            // toLower implemented elsewhere
8      }
9
10     bool operator( ) ( const string & lhs, const string & rhs ) const
11     {
12         return equalsIgnoreCase( lhs, rhs );  // equalsIgnoreCase is elsewhere
13     }
14 };
15
16 unordered_set<string,CaseInsensitiveStringHash,CaseInsensitiveStringHash> s;
```

Figure 5.23 Creating a case-insensitive `unordered_set`

These unordered classes can be used if it is not important for the entries to be viewable in sorted order. For instance, in the word-changing example in Section 4.8, there were three maps:

1. A map in which the key is a *word length*, and the value is a collection of all words of that word length.
2. A map in which the key is a *representative*, and the value is a collection of all words with that representative.
3. A map in which the key is a *word*, and the value is a collection of all words that differ in only one character from that word.

Because the order in which word lengths are processed does not matter, the first map can be an `unordered_map`. Because the representatives are not even needed after the second map is built, the second map can be an `unordered_map`. The third map can also be an `unordered_map`, unless we want `printHighChangeables` to alphabetically list the subset of words that can be changed into a large number of other words.

The performance of an `unordered_map` can often be superior to a `map`, but it is hard to know for sure without writing the code both ways.

5.7 Hash Tables with Worst-Case *O*(1) Access

The hash tables that we have examined so far all have the property that with reasonable load factors, and appropriate hash functions, we can expect $O(1)$ cost on average for insertions, removes, and searching. But what is the expected worst case for a search assuming a reasonably well-behaved hash function?

For separate chaining, assuming a load factor of 1, this is one version of the classic **balls and bins problem**: Given N balls placed randomly (uniformly) in N bins, what is the expected number of balls in the most occupied bin? The answer is well known to be $\Theta(\log N / \log \log N)$, meaning that on average, we expect some queries to take nearly logarithmic time. Similar types of bounds are observed (or provable) for the length of the longest expected probe sequence in a probing hash table.

We would like to obtain $O(1)$ worst-case cost. In some applications, such as hardware implementations of lookup tables for routers and memory caches, it is especially important that the search have a definite (i.e., constant) amount of completion time. Let us assume that N is known in advance, so no rehashing is needed. If we are allowed to rearrange items as they are inserted, then $O(1)$ worst-case cost is achievable for searches.

In the remainder of this section we describe the earliest solution to this problem, namely perfect hashing, and then two more recent approaches that appear to offer promising alternatives to the classic hashing schemes that have been prevalent for many years.

5.7.1 Perfect Hashing

Suppose, for purposes of simplification, that all N items are known in advance. If a separate chaining implementation could guarantee that each list had at most a constant number of items, we would be done. We know that as we make more lists, the lists will on average be shorter, so theoretically if we have enough lists, then with a reasonably high probability we might expect to have no collisions at all!

But there are two fundamental problems with this approach: First, the number of lists might be unreasonably large; second, even with lots of lists, we might still get unlucky.

The second problem is relatively easy to address in principle. Suppose we choose the number of lists to be M (i.e., *TableSize* is M), which is sufficiently large to guarantee that with probability at least $\frac{1}{2}$, there will be no collisions. Then if a collision is detected, we simply clear out the table and try again using a different hash function that is independent of the first. If we still get a collision, we try a third hash function, and so on. The expected number of trials will be at most 2 (since the success probability is $\frac{1}{2}$), and this is all folded into the insertion cost. Section 5.8 discusses the crucial issue of how to produce additional hash functions.

So we are left with determining how large M, the number of lists, needs to be. Unfortunately, M needs to be quite large; specifically $M = \Omega(N^2)$. However, if $M = N^2$, we can show that the table is collision free with probability at least $\frac{1}{2}$, and this result can be used to make a workable modification to our basic approach.

Theorem 5.2

If N balls are placed into $M = N^2$ bins, the probability that no bin has more than one ball is less than $\frac{1}{2}$.

Proof

If a pair (i, j) of balls are placed in the same bin, we call that a collision. Let $C_{i,j}$ be the expected number of collisions produced by any two balls (i, j). Clearly the probability that any two specified balls collide is $1/M$, and thus $C_{i,j}$ is $1/M$, since the number of collisions that involve the pair (i, j) is either 0 or 1. Thus the expected number of

collisions in the entire table is $\sum_{(i,j),\ i<j} C_{i,j}$. Since there are $N(N-1)/2$ pairs, this sum is $N(N-1)/(2M) = N(N-1)/(2N^2) < \frac{1}{2}$. Since the expected number of collisions is below $\frac{1}{2}$, the probability that there is even one collision must also be below $\frac{1}{2}$.

Of course, using N^2 lists is impractical. However, the preceding analysis suggests the following alternative: Use only N bins, but resolve the collisions in each bin by using hash tables instead of linked lists. The idea is that because the bins are expected to have only a few items each, the hash table that is used for each bin can be quadratic in the bin size. Figure 5.24 shows the basic structure. Here, the primary hash table has ten bins. Bins 1, 3, 5, and 7 are all empty. Bins 0, 4, and 8 have one item, so they are resolved by a secondary hash table with one position. Bins 2 and 6 have two items, so they will be resolved into a secondary hash table with four (2^2) positions. And bin 9 has three items, so it is resolved into a secondary hash table with nine (3^2) positions.

As with the original idea, each secondary hash table will be constructed using a different hash function until it is collision free. The primary hash table can also be constructed several times if the number of collisions that are produced is higher than required. This scheme is known as **perfect hashing**. All that remains to be shown is that the total size of the secondary hash tables is indeed expected to be linear.

Theorem 5.3

If N items are placed into a primary hash table containing N bins, then the total size of the secondary hash tables has expected value at most $2N$.

Proof

Using the same logic as in the proof of Theorem 5.2, the expected number of pairwise collisions is at most $N(N-1)/2N$, or $(N-1)/2$. Let b_i be the number of items that hash to position i in the primary hash table; observe that b_i^2 space is used for this cell

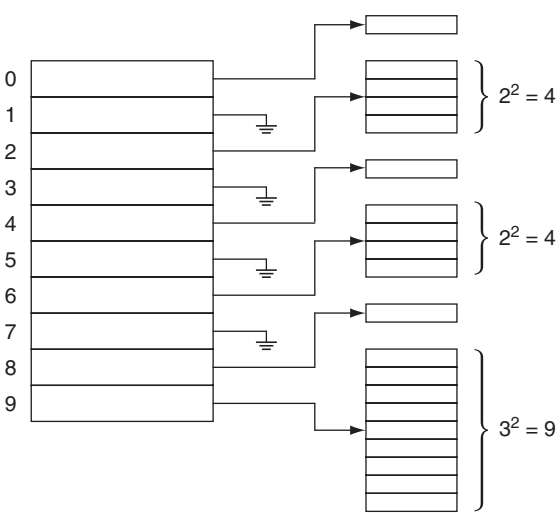


Figure 5.24 Perfect hashing table using secondary hash tables

in the secondary hash table, and that this accounts for b_i $(b_i - 1)/2$ pairwise collisions, which we will call c_i. Thus the amount of space, b_i^2, used for the ith secondary hash table is $2c_i + b_i$. The total space is then $2\sum c_i + \sum b_i$. The total number of collisions is $(N - 1)/2$ (from the first sentence of this proof); the total number of items is of course N, so we obtain a total secondary space requirement of $2(N - 1)/2 + N < 2N$.

Thus, the probability that the total secondary space requirement is more than $4N$ is at most $\frac{1}{2}$ (since, otherwise, the expected value would be higher than $2N$), so we can keep choosing hash functions for the primary table until we generate the appropriate secondary space requirement. Once that is done, each secondary hash table will itself require only an average of two trials to be collision free. After the tables are built, any lookup can be done in two probes.

Perfect hashing works if the items are all known in advance. There are dynamic schemes that allow insertions and deletions (**dynamic perfect hashing**), but instead we will investigate two newer alternatives that appear to be competitive in practice with the classic hashing algorithms.

5.7.2 Cuckoo Hashing

From our previous discussion, we know that in the balls and bins problem, if N items are randomly tossed into N bins, the size of the largest bin is expected to be $\Theta(\log N / \log \log N)$. Since this bound has been known for a long time, and the problem has been well studied by mathematicians, it was surprising when, in the mid 1990s, it was shown that if, at each toss, two bins were randomly chosen and the item was tossed into the more empty bin (at the time), then the size of the largest bin would only be $\Theta(\log \log N)$, a significantly lower number. Quickly, a host of potential algorithms and data structures arose out of this new concept of the "power of two choices."

One of the ideas is **cuckoo hashing**. In cuckoo hashing, suppose we have N items. We maintain two tables, each more than half empty, and we have two independent hash functions that can assign each item to a position in each table. Cuckoo hashing maintains the invariant that an item is always stored in one of these two locations.

As an example, Figure 5.25 shows a potential cuckoo hash table for six items, with two tables of size 5 (these tables are too small, but serve well as an example). Based on the

Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Figure 5.25 Potential cuckoo hash table. Hash functions are shown on the right. For these six items, there are only three valid positions in Table 1 and three valid positions in Table 2, so it is not clear that this arrangement can easily be found.

randomly chosen hash functions, item *A* can be at either position 0 in Table 1 or position 2 in Table 2. Item *F* can be at either position 3 in Table 1 or position 4 in Table 2, and so on. Immediately, this implies that a search in a cuckoo hash table requires at most two table accesses, and a remove is trivial, once the item is located (lazy deletion is not needed now!).

But there is an important detail: How is the table built? For instance, in Figure 5.25, there are only three available locations in the first table for the six items, and there are only three available locations in the second table for the six items. So there are only six available locations for these six items, and thus we must find an ideal matching of slots for our six items. Clearly, if there were a seventh item, *G*, with locations 1 for Table 1 and 2 for Table 2, it could not be inserted into the table by any algorithm (the seven items would be competing for six table locations). One could argue that this means that the table would simply be too loaded (*G* would yield a 0.70 load factor), but at the same time, if the table had thousands of items, and were lightly loaded, but we had *A*, *B*, *C*, *D*, *E*, *F*, *G* with these hash positions, it would still be impossible to insert all seven of those items. So it is not at all obvious that this scheme can be made to work. The answer in this situation would be to pick another hash function, and this can be fine as long as it is unlikely that this situation occurs.

The cuckoo hashing algorithm itself is simple: To insert a new item, *x*, first make sure it is not already there. We can then use the first hash function, and if the (first) table location is empty, the item can be placed. So Figure 5.26 shows the result of inserting *A* into an empty hash table.

Suppose now we want to insert *B*, which has hash locations 0 in Table 1 and 0 in Table 2. For the remainder of the algorithm, we will use (h_1, h_2) to specify the two locations, so *B*'s locations are given by (0, 0). Table 1 is already occupied in position 0. At this point there are two options: One is to look in Table 2. The problem is that position 0 in Table 2 could also be occupied. It happens that in this case it is not, but the algorithm that the standard cuckoo hash table uses does not bother to look. Instead, it preemptively places the new item *B* in Table 1. In order to do so, it must displace *A*, so *A* moves to Table 2, using its Table 2 hash location, which is position 2. The result is shown in Figure 5.27. It is easy to insert *C*, and this is shown in Figure 5.28.

Next we want to insert *D*, with hash locations (1, 0). But the Table 1 location (position 1) is already taken. Note also that the Table 2 location is not already taken, but we don't look there. Instead, we have *D* replace *C*, and then *C* goes into Table 2 at position 4, as suggested by its second hash function. The resulting tables are shown in Figure 5.29.

Table 1	
0	A
1	
2	
3	
4	

Table 2	
0	
1	
2	
3	
4	

A: 0, 2

Figure 5.26 Cuckoo hash table after insertion of *A*

Table 1	
0	B
1	
2	
3	
4	

Table 2	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0

Figure 5.27 Cuckoo hash table after insertion of *B*

Table 1	
0	B
1	C
2	
3	
4	

Table 2	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0
C: 1, 4

Figure 5.28 Cuckoo hash table after insertion of *C*

After this is done, *E* can be easily inserted. So far, so good, but can we now insert *F*? Figures 5.30 to 5.33 show that this algorithm successfully inserts *F*, by displacing *E*, then *A*, and then *B*.

Clearly, as we mentioned before, we cannot successfully insert *G* with hash locations (1, 2). If we were to try, we would displace *D*, then *B*, then *A*, *E*, *F*, and *C*, and then *C*

Table 1	
0	B
1	D
2	
3	E
4	

Table 2	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2

Figure 5.29 Cuckoo hash table after insertion of *D*

Table 1	
0	B
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Figure 5.30 Cuckoo hash table starting the insertion of *F* into the table in Figure 5.29. First, *F* displaces *E*.

Table 1	
0	B
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Figure 5.31 Continuing the insertion of *F* into the table in Figure 5.29. Next, *E* displaces *A*.

Table 1	
0	**A**
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Figure 5.32 Continuing the insertion of *F* into the table in Figure 5.29. Next, *A* displaces *B*.

Table 1	
0	**A**
1	D
2	
3	**F**
4	

Table 2	
0	**B**
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Figure 5.33 Completing the insertion of *F* into the table in Figure 5.29. Miraculously (?), *B* finds an empty position in Table 2.

would try to go back into Table 1, position 1, displacing *G* which was placed there at the start. This would get us to Figure 5.34. So now *G* would try its alternate in Table 2 (location 2) and then displace *A*, which would displace *B*, which would displace *D*, which would displace *C*, which would displace *F*, which would displace *E*, which would now displace *G* from position 2. At this point, *G* would be in a cycle.

The central issue then concerns questions such as what is the probability of there being a cycle that prevents the insertion from completing, and what is the expected number of displacements required for a successful insertion? Fortunately, if the table's load factor is below 0.5, an analysis shows that the probability of a cycle is very low, that the expected number of displacements is a small constant, and that it is extremely unlikely that a successful insertion would require more than $O(\log N)$ displacements. As such, we can simply rebuild the tables with new hash functions after a certain number of displacements are

Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4
G: 1, 2

Figure 5.34 Inserting *G* into the table in Figure 5.33. *G* displaces *D*, which displaces *B*, which displaces *A*, which displaces *E*, which displaces *F*, which displaces *C*, which displaces *G*. It is not yet hopeless since when *G* is displaced, we would now try the other hash table, at position 2. However, while that could be successful in general, in this case there is a cycle and the insertion will not terminate.

detected. More precisely, the probability that a single insertion would require a new set of hash functions can be made to be $O(1/N^2)$; the new hash functions themselves generate N more insertions to rebuild the table, but even so, this means the rebuilding cost is minimal. However, if the table's load factor is at 0.5 or higher, then the probability of a cycle becomes drastically higher, and this scheme is unlikely to work well at all.

After the publication of cuckoo hashing, numerous extensions were proposed. For instance, instead of two tables, we can use a higher number of tables, such as 3 or 4. While this increases the cost of a lookup, it also drastically increases the theoretical space utilization. In some applications the lookups through separate hash functions can be done in parallel and thus cost little to no additional time. Another extension is to allow each table to store multiple keys; again, this can increase space utilization and make it easier to do insertions and can be more cache-friendly. Various combinations are possible, as shown in Figure 5.35. And finally, often cuckoo hash tables are implemented as one giant table with two (or more) hash functions that probe the entire table, and some variations attempt to place an item in the second hash table immediately if there is an available spot, rather than starting a sequence of displacements.

Cuckoo Hash Table Implementation

Implementing cuckoo hashing requires a collection of hash functions; simply using `hashCode` to generate the collection of hash functions makes no sense, since any `hashCode` collisions will result in collisions in all the hash functions. Figure 5.36 shows a simple interface that can be used to send families of hash functions to the cuckoo hash table.

	1 item per cell	2 items per cell	4 items per cell
2 hash functions	0.49	0.86	0.93
3 hash functions	0.91	0.97	0.98
4 hash functions	0.97	0.99	0.999

Figure 5.35 Maximum load factors for cuckoo hashing variations

```
1    template <typename AnyType>
2    class CuckooHashFamily
3    {
4      public:
5        size_t hash( const AnyType & x, int which ) const;
6        int getNumberOfFunctions( );
7        void generateNewFunctions( );
8    };
```

Figure 5.36 Generic `HashFamily` interface for cuckoo hashing

Figure 5.37 provides the class interface for cuckoo hashing. We will code a variant that will allow an arbitrary number of hash functions (specified by the `HashFamily` template parameter type) which uses a single array that is addressed by all the hash functions. Thus our implementation differs from the classic notion of two separately addressable hash tables. We can implement the classic version by making relatively minor changes to the code; however, the version provided in this section seems to perform better in tests using simple hash functions.

In Figure 5.37, we specify that the maximum load for the table is 0.4; if the load factor of the table is about to exceed this limit, an automatic table expansion is performed. We also define `ALLOWED_REHASHES`, which specifies how many rehashes we will perform if evictions take too long. In theory, `ALLOWED_REHASHES` can be infinite, since we expect only a small constant number of rehashes are needed; in practice, depending on several factors such as the number of hash functions, the quality of the hash functions, and the load factor, the rehashes could significantly slow things down, and it might be worthwhile to expand the table, even though this will cost space. The data representation for the cuckoo hash table is straightforward: We store a simple array, the current size, and the collections of hash functions, represented in a `HashFamily` instance. We also maintain the number of hash functions, even though that is always obtainable from the `HashFamily` instance.

Figure 5.38 shows the constructor and `makeEmpty` methods, and these are straightforward. Figure 5.39 shows a pair of private methods. The first, `myHash`, is used to select the appropriate hash function and then scale it into a valid array index. The second, `findPos`, consults all the hash functions to return the index containing item `x`, or -1 if `x` is not found. `findPos` is then used by `contains` and `remove` in Figures 5.40 and 5.41, respectively, and we can see that those methods are easy to implement.

The difficult routine is insertion. In Figure 5.42, we can see that the basic plan is to check to see if the item is already present, returning if so. Otherwise, we check to see if the table is fully loaded, and if so, we expand it. Finally we call a helper routine to do all the dirty work.

The helper routine for insertion is shown in Figure 5.43. We declare a variable `rehashes` to keep track of how many attempts have been made to rehash in this insertion. Our insertion routine is mutually recursive: If needed, `insert` eventually calls `rehash`, which eventually calls back into `insert`. Thus `rehashes` is declared in an outer scope for code simplicity.

```
template <typename AnyType, typename HashFamily>
class CuckooHashTable
{
  public:
    explicit CuckooHashTable( int size = 101 );

    void makeEmpty( );
    bool contains( const AnyType & x ) const;

    bool remove( const AnyType & x );
    bool insert( const AnyType & x );
    bool insert( AnyType && x );

  private:
    struct HashEntry
    {
        AnyType element;
        bool isActive;

        HashEntry( const AnyType & e = AnyType( ), bool a = false )
          : element{ e }, isActive{ a } { }
        HashEntry( AnyType && e, bool a = false )
          : element{ std::move( e ) }, isActive{ a } { }
    };

    bool insertHelper1( const AnyType & xx );
    bool insertHelper1( AnyType && xx );
    bool isActive( int currentPos ) const;

    size_t myhash( const AnyType & x, int which ) const;
    int findPos( const AnyType & x ) const;
    void expand( );
    void rehash( );
    void rehash( int newSize );

    static const double MAX_LOAD = 0.40;
    static const int ALLOWED_REHASHES = 5;

    vector<HashEntry> array;
    int currentSize;
    int numHashFunctions;
    int rehashes;
    UniformRandom r;
    HashFamily hashFunctions;
};
```

Figure 5.37 Class interface for cuckoo hashing

```
1       explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2       {
3           numHashFunctions = hashFunctions.getNumberOfFunctions( );
4           rehashes = 0;
5           makeEmpty( );
6       }
7
8       void makeEmpty( )
9       {
10          currentSize = 0;
11          for( auto & entry : array )
12              entry.isActive = false;
13      }
```

Figure 5.38 Routines to initialize and empty the cuckoo hash table

```
1       /**
2        * Compute the hash code for x using specified function.
3        */
4       int myhash( const AnyType & x, int which ) const
5       {
6           return hashFunctions.hash( x, which ) % array.size( );
7       }
8
9       /**
10       * Search all hash function places. Return the position
11       * where the search terminates or -1 if not found.
12       */
13      int findPos( const AnyType & x ) const
14      {
15          for( int i = 0; i < numHashFunctions; ++i )
16          {
17              int pos = myhash( x, i );
18
19              if( isActive( pos ) && array[ pos ].element == x )
20                  return pos;
21          }
22
23          return -1;
24      }
```

Figure 5.39 Routines to find the location of an item in the cuckoo hash table and to compute the hash code for a given table

```
1   /**
2    * Return true if x is found.
3    */
4   bool contains( const AnyType & x ) const
5   {
6       return findPos( x ) != -1;
7   }
```

Figure 5.40 Routine to search a cuckoo hash table

```
1   /**
2    * Remove x from the hash table.
3    * Return true if item was found and removed.
4    */
5   bool remove( const AnyType & x )
6   {
7      int currentPos = findPos( x );
8      if( !isActive( currentPos ) )
9          return false;
10
11     array[ currentPos ].isActive = false;
12     --currentSize;
13     return true;
14  }
```

Figure 5.41 Routine to remove from a cuckoo hash table

```
1       bool insert( const AnyType & x )
2       {
3           if( contains( x ) )
4               return false;
5
6           if( currentSize >= array.size( ) * MAX_LOAD )
7               expand( );
8
9           return insertHelper1( x );
10      }
```

Figure 5.42 Public insert routine for cuckoo hashing

Our basic logic is different from the classic scheme. We have already tested that the item to insert is not already present. At lines 15 to 25, we check to see if any of the valid positions are empty; if so, we place our item in the first available position and we are done. Otherwise, we evict one of the existing items. However, there are some tricky issues:

```
1       static const int ALLOWED_REHASHES = 5;
2
3       bool insertHelper1( const AnyType & xx )
4       {
5           const int COUNT_LIMIT = 100;
6           AnyType x = xx;
7
8           while( true )
9           {
10              int lastPos = -1;
11              int pos;
12
13              for( int count = 0; count < COUNT_LIMIT; ++count )
14              {
15                  for( int i = 0; i < numHashFunctions; ++i )
16                  {
17                      pos = myhash( x, i );
18
19                      if( !isActive( pos ) )
20                      {
21                          array[ pos ] = std::move( HashEntry{ std::move( x ), true } );
22                          ++currentSize;
23                          return true;
24                      }
25                  }
26
27                  // None of the spots are available. Evict a random one
28                  int i = 0;
29                  do
30                  {
31                      pos = myhash( x, r.nextInt( numHashFunctions ) );
32                  } while( pos == lastPos && i++ < 5 );
33
34                  lastPos = pos;
35                  std::swap( x, array[ pos ].element );
36              }
37
38              if( ++rehashes > ALLOWED_REHASHES )
39              {
40                  expand( );     // Make the table bigger
41                  rehashes = 0;  // Reset the # of rehashes
42              }
43              else
44                  rehash( );     // Same table size, new hash functions
45          }
46      }
```

Figure 5.43 Insertion routine for cuckoo hashing uses a different algorithm that chooses the item to evict randomly, attempting not to re-evict the last item. The table will attempt to select new hash functions (rehash) if there are too many evictions and will expand if there are too many rehashes.

- Evicting the first item did not perform well in experiments.
- Evicting the last item did not perform well in experiments.
- Evicting the items in sequence (i.e., the first eviction uses hash function 0, the next uses hash function 1, etc.) did not perform well in experiments.
- Evicting the item purely randomly did not perform well in experiments: In particular, with only two hash functions, it tended to create cycles.

To alleviate the last problem, we maintain the last position that was evicted, and if our random item was the last evicted item, we select a new random item. This will loop forever if used with two hash functions, and both hash functions happen to probe to the same location, and that location was a prior eviction, so we limit the loop to five iterations (deliberately using an odd number).

The code for `expand` and `rehash` is shown in Figure 5.44. `expand` creates a larger array but keeps the same hash functions. The zero-parameter `rehash` leaves the array size unchanged but creates a new array that is populated with newly chosen hash functions.

```
1       void expand( )
2       {
3           rehash( static_cast<int>( array.size( ) / MAX_LOAD ) );
4       }
5
6       void rehash( )
7       {
8           hashFunctions.generateNewFunctions( );
9           rehash( array.size( ) );
10      }
11
12      void rehash( int newSize )
13      {
14          vector<HashEntry> oldArray = array;
15
16              // Create new double-sized, empty table
17          array.resize( nextPrime( newSize ) );
18          for( auto & entry : array )
19              entry.isActive = false;
20
21              // Copy table over
22          currentSize = 0;
23          for( auto & entry : oldArray )
24              if( entry.isActive )
25                  insert( std::move( entry.element ) );
26      }
```

Figure 5.44 Rehashing and expanding code for cuckoo hash tables

```
template <int count>
class StringHashFamily
{
  public:
    StringHashFamily( ) : MULTIPLIERS( count )
    {
        generateNewFunctions( );
    }

    int getNumberOfFunctions( ) const
    {
        return count;
    }

    void generateNewFunctions( )
    {
        for( auto & mult : MULTIPLIERS )
            mult = r.nextInt( );
    }

    size_t hash( const string & x, int which ) const
    {
        const int multiplier = MULTIPLIERS[ which ];
        size_t hashVal = 0;

        for( auto ch : x )
            hashVal = multiplier * hashVal + ch;

        return hashVal;
    }

  private:
    vector<int> MULTIPLIERS;
    UniformRandom r;
};
```

Figure 5.45 Casual string hashing for cuckoo hashing; these hash functions do not provably satisfy the requirements needed for cuckoo hashing but offer decent performance if the table is not highly loaded and the alternate insertion routine in Figure 5.43 is used.

Finally, Figure 5.45 shows the `StringHashFamily` class that provides a set of simple hash functions for strings. These hash functions replace the constant 37 in Figure 5.4 with randomly chosen numbers (not necessarily prime).

The benefits of cuckoo hashing include the worst-case constant lookup and deletion times, the avoidance of lazy deletion and extra data, and the potential for parallelism.

However, cuckoo hashing is extremely sensitive to the choice of hash functions; the inventors of the cuckoo hash table reported that many of the standard hash functions that they attempted performed poorly in tests. Furthermore, although the insertion time is expected to be constant time as long as the load factor is below $\frac{1}{2}$, the bound that has been shown for the expected insertion cost for classic cuckoo hashing with two separate tables (both with load factor λ) is roughly $1/(1 - (4\lambda^2)^{1/3})$, which deteriorates rapidly as the load factor gets close to $\frac{1}{2}$ (the formula itself makes no sense when λ equals or exceeds $\frac{1}{2}$). Using lower load factors or more than two hash functions seems like a reasonable alternative.

5.7.3 Hopscotch Hashing

Hopscotch hashing is a new algorithm that tries to improve on the classic linear probing algorithm. Recall that in linear probing, cells are tried in sequential order, starting from the hash location. Because of primary and secondary clustering, this sequence can be long on average as the table gets loaded, and thus many improvements such as quadratic probing, double hashing, and so forth, have been proposed to reduce the number of collisions. However, on some modern architectures, the locality produced by probing adjacent cells is a more significant factor than the extra probes, and linear probing can still be practical or even a best choice.

The idea of hopscotch hashing is to bound the maximal length of the probe sequence by a predetermined constant that is optimized to the underlying computer's architecture. Doing so would give constant-time lookups in the worst case, and like cuckoo hashing, the lookup could be parallelized to simultaneously check the bounded set of possible locations.

If an insertion would place a new item too far from its hash location, then we efficiently go backward toward the hash location, evicting potential items. If we are careful, the evictions can be done quickly and guarantee that those evicted are not placed too far from their hash locations. The algorithm is deterministic in that given a hash function, either the items can be evicted or they can't. The latter case implies that the table is likely too crowded, and a rehash is in order; but this would happen only at extremely high load factors, exceeding 0.9. For a table with a load factor of $\frac{1}{2}$, the failure probability is almost zero (Exercise 5.23).

Let *MAX_DIST* be the chosen bound on the maximum probe sequence. This means that item x must be found somewhere in the *MAX_DIST* positions listed in $hash(x)$, $hash(x) + 1, \ldots,$ $hash(x) + (MAX\_DIST - 1)$. In order to efficiently process evictions, we maintain information that tells for each position x, whether the item in the alternate position is occupied by an element that hashes to position x.

As an example, Figure 5.46 shows a fairly crowded hopscotch hash table, using $MAX\_DIST = 4$. The bit array for position 6 shows that only position 6 has an item (*C*) with hash value 6: Only the first bit of Hop[6] is set. Hop[7] has the first two bits set, indicating that positions 7 and 8 (*A* and *D*) are occupied with items whose hash value is 7. And Hop[8] has only the third bit set, indicating that the item in position 10 (*E*) has hash value 8. If *MAX_DIST* is no more than 32, the Hop array is essentially an array of 32-bit integers, so the additional space requirement is not substantial. If Hop[*pos*] contains all 1s for some *pos*, then an attempt to insert an item whose hash value is *pos* will clearly

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11

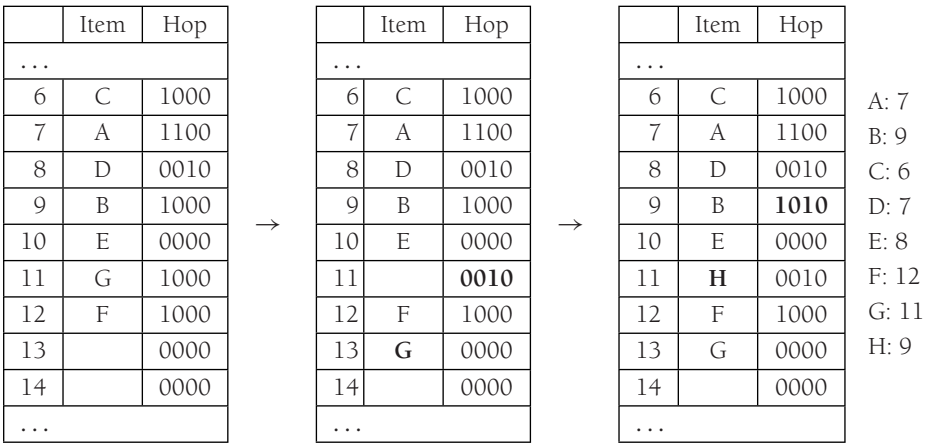
Figure 5.46 Hopscotch hashing table. The hops tell which of the positions in the block are occupied with cells containing this hash value. Thus Hop[8] = 0010 indicates that only position 10 currently contains items whose hash value is 8, while positions 8, 9, and 11 do not.

fail, since there would now be *MAX_DIST* + 1 items trying to reside within *MAX_DIST* positions of *pos*—an impossibility.

Continuing the example, suppose we now insert item *H* with hash value 9. Our normal linear probing would try to place it in position 13, but that is too far from the hash value of 9. So instead, we look to evict an item and relocate it to position 13. The only candidates to go into position 13 would be items with hash value of 10, 11, 12, or 13. If we examine Hop[10], we see that there are no candidates with hash value 10. But Hop[11] produces a candidate, *G*, with value 11 that can be placed into position 13. Since position 11 is now close enough to the hash value of *H*, we can now insert *H*. These steps, along with the changes to the Hop information, are shown in Figure 5.47.

Finally, we will attempt to insert *I* whose hash value is 6. Linear probing suggests position 14, but of course that is too far away. Thus we look in Hop[11], and it tells us that *G* can move down, freeing up position 13. Now that 13 is vacant, we can look in Hop[10] to find another element to evict. But Hop[10] has all zeros in the first three positions, so there are no items with hash value 10 that can be moved. So we examine Hop[11]. There we find all zeros in the first two positions.

So we try Hop[12], where we need the first position to be 1, which it is. Thus *F* can move down. These two steps are shown in Figure 5.48. Notice that if this were not the case—for instance if hash(*F*) were 9 instead of 12—we would be stuck and have to rehash. However, that is not a problem with our algorithm; instead, there would simply be no way to place all of *C*, *I*, *A*, *D*, *E*, *B*, *H*, and *F* (if *F*'s hash value were 9); these items would all have hash values between 6 and 9, and would thus need to be placed in the seven spots between 6 and 12. But that would be eight items in seven spots—an impossibility. However, since this is not the case for our example, and we have evicted an item from position 12, we can now continue. Figure 5.49 shows the remaining eviction from position 9 and subsequent placement of *I*.



	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

→

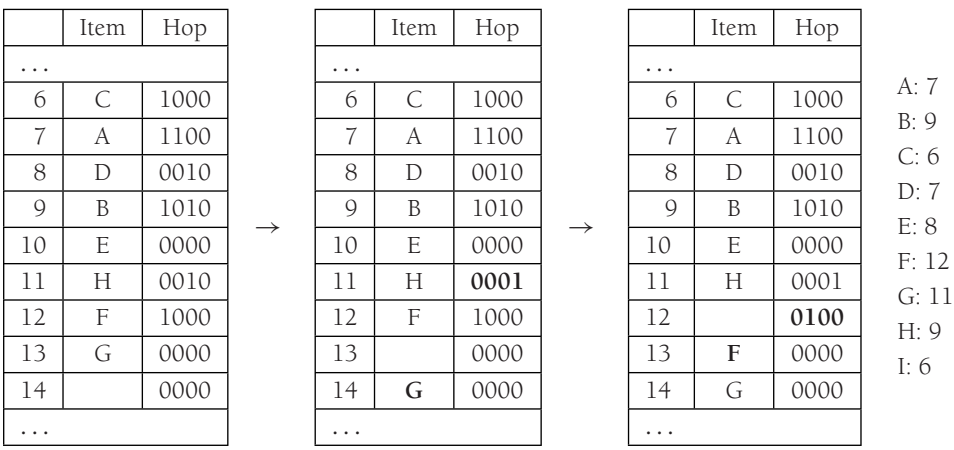
	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11		**0010**
12	F	1000
13	**G**	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	**1010**
10	E	0000
11	**H**	0010
12	F	1000
13	G	0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9

Figure 5.47 Hopscotch hashing table. Attempting to insert *H*. Linear probing suggests location 13, but that is too far, so we evict *G* from position 11 to find a closer position.



	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	**0001**
12	F	1000
13		0000
14	**G**	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		**0100**
13	**F**	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

Figure 5.48 Hopscotch hashing table. Attempting to insert *I*. Linear probing suggests location 14, but that is too far; consulting Hop[11], we see that *G* can move down, leaving position 13 open. Consulting Hop[10] gives no suggestions. Hop[11] does not help either (why?), so Hop[12] suggests moving *F*.

Hopscotch hashing is a relatively new algorithm, but the initial experimental results are very promising, especially for applications that make use of multiple processors and require significant parallelism and concurrency. It remains to be seen if either cuckoo hashing or hopscotch hashing emerge as a practical alternative to the classic separate chaining and linear/quadratic probing schemes.

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9		**0011**
10	E	0000
11	H	0001
12	**B**	0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	**1001**
7	A	1100
8	D	0010
9	**I**	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

Figure 5.49 Hopscotch hashing table. Insertion of *I* continues: Next, *B* is evicted, and finally, we have a spot that is close enough to the hash value and can insert *I*.

5.8 Universal Hashing

Although hash tables are very efficient and have constant average cost per operation, assuming appropriate load factors, their analysis and performance depend on the hash function having two fundamental properties:

1. The hash function must be computable in constant time (i.e., independent of the number of items in the hash table).
2. The hash function must distribute its items uniformly among the array slots.

In particular, if the hash function is poor, then all bets are off, and the cost per operation can be linear. In this section, we discuss **universal hash functions**, which allow us to choose the hash function randomly in such a way that condition 2 above is satisfied. As in Section 5.7, we use M to represent *TableSize*. Although a strong motivation for the use of universal hash functions is to provide theoretical justification for the assumptions used in the classic hash table analyses, these functions can also be used in applications that require a high level of robustness, in which worst-case (or even substantially degraded) performance, perhaps based on inputs generated by a saboteur or hacker, simply cannot be tolerated.

As in Section 5.7, we use M to represent *TableSize*.

Definition 5.1

A family H of hash functions is *universal*, if for any $x \neq y$, the number of hash functions h in H for which $h(x) = h(y)$ is at most $|H|/M$.

Notice that this definition holds for each pair of items, rather than being averaged over all pairs of items. The definition above means that if we choose a hash function randomly from a universal family H, then the probability of a collision between any two distinct items

is at most $1/M$, and when adding into a table with N items, the probability of a collision at the initial point is at most N/M, or the load factor.

The use of a universal hash function for separate chaining or hopscotch hashing would be sufficient to meet the assumptions used in the analysis of those data structures. However, it is not sufficient for cuckoo hashing, which requires a stronger notion of independence. In cuckoo hashing, we first see if there is a vacant location; if there is not, and we do an eviction, a different item is now involved in looking for a vacant location. This repeats until we find the vacant location, or decide to rehash [generally within $O(\log N)$ steps]. In order for the analysis to work, each step must have a collision probability of N/M independently, with a different item x being subject to the hash function. We can formalize this independence requirement in the following definition.

Definition 5.2

A family H of hash functions is *k-universal*, if for any $x_1 \neq y_1$, $x_2 \neq y_2, \ldots, x_k \neq y_k$, the number of hash functions h in H for which $h(x_1) = h(y_1)$, $h(x_2) = h(y_2), \ldots$, and $h(x_k) = h(y_k)$ is at most $|H|/M^k$.

With this definition, we see that the analysis of cuckoo hashing requires an $O(\log N)$-universal hash function (after that many evictions, we give up and rehash). In this section we look only at universal hash functions.

To design a simple universal hash function, we will assume first that we are mapping very large integers into smaller integers ranging from 0 to $M - 1$. Let p be a prime larger than the largest input key.

Our universal family H will consist of the following set of functions, where a and b are chosen randomly:

$$H = \{H_{a,b}(x) = ((ax + b) \mod p) \mod M, \text{ where } 1 \leq a \leq p - 1, 0 \leq b \leq p - 1\}$$

For example, in this family, three of the possible random choices of (a, b) yield three different hash functions:

$$H_{3,7}(x) = ((3x + 7) \mod p) \mod M$$
$$H_{4,1}(x) = ((4x + 1) \mod p) \mod M$$
$$H_{8,0}(x) = ((8x) \mod p) \mod M$$

Observe that there are $p(p - 1)$ possible hash functions that can be chosen.

Theorem 5.4

The hash family $H = \{H_{a,b}(x) = ((ax + b) \mod p) \mod M$, where $1 \leq a \leq p - 1$, $0 \leq b \leq p - 1\}$ is universal.

Proof

Let x and y be distinct values, with $x > y$, such that $H_{a,b}(x) = H_{a,b}(y)$.

Clearly if $(ax + b) \mod p$ is equal to $(ay + b) \mod p$, then we will have a collision. However, this cannot happen: Subtracting equations yields $a(x - y) \equiv 0 \pmod{p}$, which would mean that p divides a or p divides $x - y$, since p is prime. But neither can happen, since both a and $x - y$ are between 1 and $p - 1$.

So let $r = (ax + b) \mod p$ and let $s = (ay + b) \mod p$, and by the above argument, $r \neq s$. Thus there are p possible values for r, and for each r, there are $p - 1$ possible

values for s, for a total of $p(p-1)$ possible (r,s) pairs. Notice that the number of (a,b) pairs and the number of (r,s) pairs is identical; thus each (r,s) pair will correspond to exactly one (a,b) pair if we can solve for (a,b) in terms of r and s. But that is easy: As before, subtracting equations yields $a(x-y) \equiv (r-s) \pmod{p}$, which means that by multiplying both sides by the unique multiplicative inverse of $(x-y)$ (which must exist, since $x-y$ is not zero and p is prime), we obtain a, in terms of r and s. Then b follows.

Finally, this means that the probability that x and y collide is equal to the probability that $r \equiv s \pmod{M}$, and the above analysis allows us to assume that r and s are chosen randomly, rather than a and b. Immediate intuition would place this probability at $1/M$, but that would only be true if p were an exact multiple of M, and all possible (r,s) pairs were equally likely. Since p is prime, and $r \neq s$, that is not exactly true, so a more careful analysis is needed.

For a given r, the number of values of s that can collide mod M is at most $\lceil p/M \rceil - 1$ (the -1 is because $r \neq s$). It is easy to see that this is at most $(p-1)/M$. Thus the probability that r and s will generate a collision is at most $1/M$ (we divide by $p-1$, because, as mentioned earlier in the proof, there are only $p-1$ choices for s given r). This implies that the hash family is universal.

Implementation of this hash function would seem to require two mod operations: one mod p and the second mod M. Figure 5.50 shows a simple implementation in C++, assuming that M is significantly less than $2^{31}-1$. Because the computations must now be exactly as specified, and thus overflow is no longer acceptable, we promote to `long long` computations, which are at least 64 bits.

However, we are allowed to choose any prime p, as long as it is larger than M. Hence, it makes sense to choose a prime that is most favorable for computations. One such prime is $p = 2^{31}-1$. Prime numbers of this form are known as Mersenne primes; other Mersenne primes include 2^5-1, $2^{61}-1$ and $2^{89}-1$. Just as a multiplication by a Mersenne prime such as 31 can be implemented by a bit shift and a subtract, a mod operation involving a Mersenne prime can also be implemented by a bit shift and an addition:

Suppose $r \equiv y \pmod{p}$. If we divide y by $(p+1)$, then $y = q'(p+1) + r'$, where q' and r' are the quotient and remainder, respectively. Thus, $r \equiv q'(p+1) + r' \pmod{p}$. And since $(p+1) \equiv 1 \pmod{p}$, we obtain $r \equiv q' + r' \pmod{p}$.

Figure 5.51 implements this idea, which is known as the **Carter-Wegman trick**. On line 8, the bit shift computes the quotient and the bitwise-and computes the remainder when dividing by $(p+1)$; these bitwise operations work because $(p+1)$ is an exact power

```
1  int universalHash( int x, int A, int B, int P, int M )
2  {
3      return static_cast<int>( ( ( static_cast<long long>( A ) * x ) + B ) % P ) % M;
4  }
```

Figure 5.50 Simple implementation of universal hashing

```
1  const int DIGS = 31;
2  const int mersennep = (1<<DIGS) - 1;
3
4  int universalHash( int x, int A, int B, int M )
5  {
6      long long hashVal = static_cast<long long>( A ) * x + B;
7
8      hashVal = ( ( hashVal >> DIGS ) + ( hashVal & mersennep ) );
9      if( hashVal >= mersennep )
10         hashVal -= mersennep;
11
12     return static_cast<int>( hashVal ) % M;
13 }
```

Figure 5.51 Simple implementation of universal hashing

of two. Since the remainder could be almost as large as p, the resulting sum might be larger than p, so we scale it back down at lines 9 and 10.

Universal hash functions exist for strings also. First, choose any prime p, larger than M (and larger than the largest character code). Then use our standard string hashing function, choosing the multiplier randomly between 1 and $p - 1$ and returning an intermediate hash value between 0 and $p - 1$, inclusive. Finally, apply a universal hash function to generate the final hash value between 0 and $M - 1$.

5.9 Extendible Hashing

Our last topic in this chapter deals with the case where the amount of data is too large to fit in main memory. As we saw in Chapter 4, the main consideration then is the number of disk accesses required to retrieve data.

As before, we assume that at any point we have N records to store; the value of N changes over time. Furthermore, at most M records fit in one disk block. We will use $M = 4$ in this section.

If either probing hashing or separate chaining hashing is used, the major problem is that collisions could cause several blocks to be examined during a search, even for a well-distributed hash table. Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(N)$ disk accesses.

A clever alternative, known as **extendible hashing**, allows a search to be performed in two disk accesses. Insertions also require few disk accesses.

We recall from Chapter 4 that a B-tree has depth $O(\log_{M/2} N)$. As M increases, the depth of a B-tree decreases. We could in theory choose M to be so large that the depth of the B-tree would be 1. Then any search after the first would take one disk access, since, presumably, the root node could be stored in main memory. The problem with this strategy is that the branching factor is so high that it would take considerable processing to determine which leaf the data was in. If the time to perform this step could be reduced, then we would have a practical scheme. This is exactly the strategy used by extendible hashing.

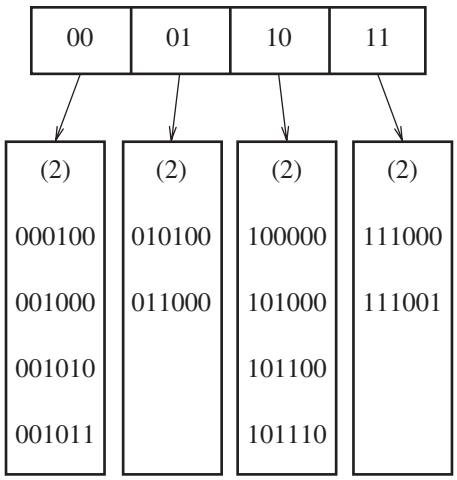


Figure 5.52 Extendible hashing: original data

Let us suppose, for the moment, that our data consists of several 6-bit integers. Figure 5.52 shows an extendible hashing scheme for these data. The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to $M = 4$ elements. It happens that in each leaf the first two bits are identical; this is indicated by the number in parentheses. To be more formal, D will represent the number of bits used by the root, which is sometimes known as the **directory.** The number of entries in the directory is thus 2^D. d_L is the number of leading bits that all the elements of some leaf L have in common. d_L will depend on the particular leaf, and $d_L \leq D$.

Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first *three* bits. This requires increasing the directory size to 3. These changes are reflected in Figure 5.53.

Notice that all the leaves not involved in the split are now pointed to by two adjacent directory entries. Thus, although an entire directory is rewritten, none of the other leaves is actually accessed.

If the key 000000 is now inserted, then the first leaf is split, generating two leaves with $d_L = 3$. Since $D = 3$, the only change required in the directory is the updating of the 000 and 001 pointers. See Figure 5.54.

This very simple strategy provides quick access times for `insert` and search operations on large databases. There are a few important details we have not considered.

First, it is possible that several directory splits will be required if the elements in a leaf agree in more than $D + 1$ leading bits. For instance, starting at the original example, with $D = 2$, if 111010, 111011, and finally 111100 are inserted, the directory size must be increased to 4 to distinguish between the five keys. This is an easy detail to take care of, but must not be forgotten. Second, there is the possibility of duplicate keys; if there are more than M duplicates, then this algorithm does not work at all. In this case, some other arrangements need to be made.

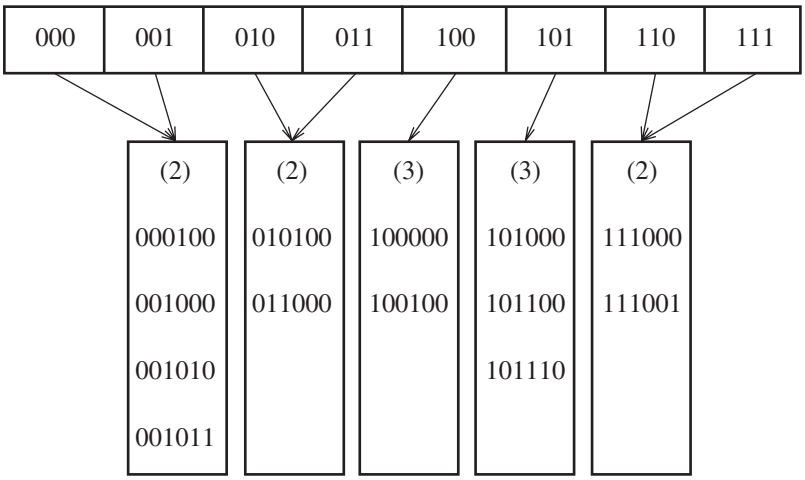


Figure 5.53 Extendible hashing: after insertion of 100100 and directory split

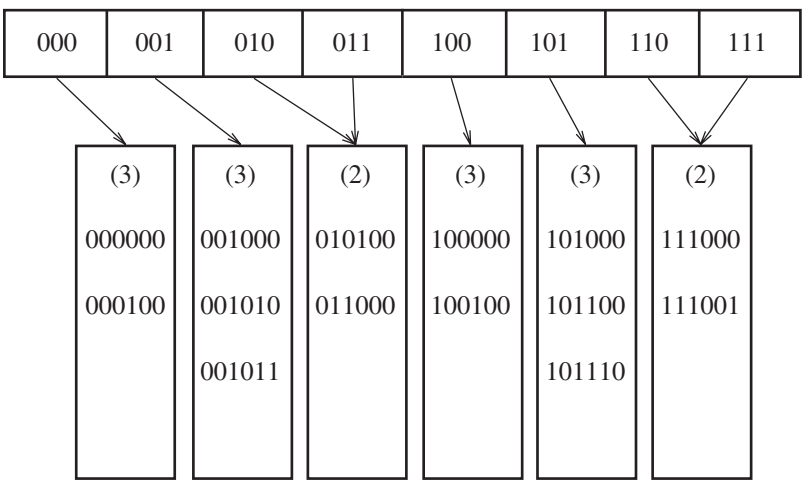


Figure 5.54 Extendible hashing: after insertion of 000000 and leaf split

These possibilities suggest that it is important for the bits to be fairly random. This can be accomplished by hashing the keys into a reasonably long integer—hence the name.

We close by mentioning some of the performance properties of extendible hashing, which are derived after a very difficult analysis. These results are based on the reasonable assumption that the bit patterns are uniformly distributed.

The expected number of leaves is $(N/M) \log_2 e$. Thus the average leaf is $\ln 2 = 0.69$ full. This is the same as for B-trees, which is not entirely surprising, since for both data structures new nodes are created when the $(M + 1)$th entry is added.

The more surprising result is that the expected size of the directory (in other words, 2^D) is $O(N^{1+1/M}/M)$. If M is very small, then the directory can get unduly large. In this case, we can have the leaves contain pointers to the records instead of the actual records, thus increasing the value of M. This adds a second disk access to each search operation in order to maintain a smaller directory. If the directory is too large to fit in main memory, the second disk access would be needed anyway.

Summary

Hash tables can be used to implement the `insert` and `contains` operations in constant average time. It is especially important to pay attention to details such as load factor when using hash tables, since otherwise the time bounds are not valid. It is also important to choose the hash function carefully when the key is not a short string or integer.

For separate chaining hashing, the load factor should be close to 1, although performance does not significantly degrade unless the load factor becomes very large. For probing hashing, the load factor should not exceed 0.5, unless this is completely unavoidable. If linear probing is used, performance degenerates rapidly as the load factor approaches 1. Rehashing can be implemented to allow the table to grow (and shrink), thus maintaining a reasonable load factor. This is important if space is tight and it is not possible just to declare a huge hash table.

Other alternatives such as cuckoo hashing and hopscotch hashing can also yield good results. Because all these algorithms are constant time, it is difficult to make strong statements about which hash table implementation is the "best"; recent simulation results provide conflicting guidance and suggest that the performance can depend strongly on the types of items being manipulated, the underlying computer hardware, and the programming language.

Binary search trees can also be used to implement `insert` and `contains` operations. Although the resulting average time bounds are $O(\log N)$, binary search trees also support routines that require order and are thus more powerful. Using a hash table, it is not possible to find the minimum element. It is not possible to search efficiently for a string unless the exact string is known. A binary search tree could quickly find all items in a certain range; this is not supported by hash tables. Furthermore, the $O(\log N)$ bound is not necessarily that much more than $O(1)$, especially since no multiplications or divisions are required by search trees.

On the other hand, the worst case for hashing generally results from an implementation error, whereas sorted input can make binary trees perform poorly. Balanced search trees are quite expensive to implement, so if no ordering information is required and there is any suspicion that the input might be sorted, then hashing is the data structure of choice.

Hashing applications are abundant. Compilers use hash tables to keep track of declared variables in source code. The data structure is known as a **symbol table.** Hash tables are the ideal application for this problem. Identifiers are typically short, so the hash function can be computed quickly, and alphabetizing the variables is often unnecessary.

A hash table is useful for any graph theory problem where the nodes have real names instead of numbers. Here, as the input is read, vertices are assigned integers from 1 onward by order of appearance. Again, the input is likely to have large groups of alphabetized entries. For example, the vertices could be computers. Then if one particular installation lists its computers as *ibm1, ibm2, ibm3,* . . . , there could be a dramatic effect on efficiency if a search tree is used.

A third common use of hash tables is in programs that play games. As the program searches through different lines of play, it keeps track of positions it has seen by computing a hash function based on the position (and storing its move for that position). If the same position recurs, usually by a simple transposition of moves, the program can avoid expensive recomputation. This general feature of all game-playing programs is known as the **transposition table.**

Yet another use of hashing is in online spelling checkers. If misspelling detection (as opposed to correction) is important, an entire dictionary can be prehashed and words can be checked in constant time. Hash tables are well suited for this, because it is not important to alphabetize words; printing out misspellings in the order they occurred in the document is certainly acceptable.

Hash tables are often used to implement caches, both in software (for instance, the cache in your Internet browser) and in hardware (for instance, the memory caches in modern computers). They are also used in hardware implementations of routers.

We close this chapter by returning to the word puzzle problem of Chapter 1. If the second algorithm described in Chapter 1 is used, and we assume that the maximum word size is some small constant, then the time to read in the dictionary containing W words and put it in a hash table is $O(W)$. This time is likely to be dominated by the disk I/O and not the hashing routines. The rest of the algorithm would test for the presence of a word for each ordered quadruple (*row, column, orientation, number of characters*). As each lookup would be $O(1)$, and there are only a constant number of orientations (8) and characters per word, the running time of this phase would be $O(R \cdot C)$. The total running time would be $O(R \cdot C + W)$, which is a distinct improvement over the original $O(R \cdot C \cdot W)$. We could make further optimizations, which would decrease the running time in practice; these are described in the exercises.

Exercises

5.1 Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \pmod{()}\ 10$, show the resulting
a. separate chaining hash table
b. hash table using linear probing
c. hash table using quadratic probing
d. hash table with second hash function $h_2(x) = 7 - (x \mod 7)$

5.2 Show the result of rehashing the hash tables in Exercise 5.1.

5.3 Write a program to compute the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing.

5.4 A large number of deletions in a separate chaining hash table can cause the table to be fairly empty, which wastes space. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as many elements as the table size. How empty should the table be before we rehash to a smaller table?

5.5 Reimplement separate chaining hash tables using a `vector` of singly linked lists instead of `vector`s.

5.6 The `isEmpty` routine for quadratic probing has not been written. Can you implement it by returning the expression `currentSize==0`?

5.7 In the quadratic probing hash table, suppose that instead of inserting a new item into the location suggested by `findPos`, we insert it into the first inactive cell on the search path (thus, it is possible to reclaim a cell that is marked deleted, potentially saving space).

a. Rewrite the insertion algorithm to use this observation. Do this by having `findPos` maintain, with an additional variable, the location of the first inactive cell it encounters.

b. Explain the circumstances under which the revised algorithm is faster than the original algorithm. Can it be slower?

5.8 Suppose instead of quadratic probing, we use "cubic probing"; here the ith probe is at $hash(x) + i^3$. Does cubic probing improve on quadratic probing?

5.9 Using a standard dictionary, and a table size that approximates a load factor of 1, compare the number of collisions produced by the hash function in Figure 5.4 and the hash function in Figure 5.55.

5.10 What are the advantages and disadvantages of the various collision resolution strategies?

5.11 Suppose that to mitigate the effects of secondary clustering we use as the collision resolution function $f(i) = i \cdot r(hash(x))$, where $hash(x)$ is the 32-bit hash value (not yet scaled to a suitable array index), and $r(y) = |48271y(\ \mathrm{mod}\ (2^{31} - 1))|$

```
1   /**
2    * FNV-1a hash routine for string objects.
3    */
4   unsigned int hash( const string & key, int tableSize )
5   {
6       unsigned int hashVal = 2166136261;
7
8       for( char ch : key )
9           hashVal = ( hashVal ^ ch )* 16777619;
10
11      return hashVal % tableSize;
12  }
```

Figure 5.55 Alternative hash function for Exercise 5.9.

mod *TableSize*. (Section 10.4.1 describes a method of performing this calculation without overflows, but it is unlikely that overflow matters in this case.) Explain why this strategy tends to avoid secondary clustering, and compare this strategy with both double hashing and quadratic probing.

5.12 Rehashing requires recomputing the hash function for all items in the hash table. Since computing the hash function is expensive, suppose objects provide a hash member function of their own, and each object stores the result in an additional data member the first time the hash function is computed for it. Show how such a scheme would apply for the `Employee` class in Figure 5.8, and explain under what circumstances the remembered hash value remains valid in each `Employee`.

5.13 Write a program to implement the following strategy for multiplying two sparse polynomials P_1, P_2 of size M and N, respectively. Each polynomial is represented as a list of objects consisting of a coefficient and an exponent. We multiply each term in P_1 by a term in P_2 for a total of MN operations. One method is to sort these terms and combine like terms, but this requires sorting MN records, which could be expensive, especially in small-memory environments. Alternatively, we could merge terms as they are computed and then sort the result.

a. Write a program to implement the alternative strategy.
b. If the output polynomial has about $O(M + N)$ terms, what is the running time of both methods?

⋆ **5.14** Describe a procedure that avoids initializing a hash table (at the expense of memory).

5.15 Suppose we want to find the first occurrence of a string $P_1P_2 \cdots P_k$ in a long input string $A_1A_2 \cdots A_N$. We can solve this problem by hashing the pattern string, obtaining a hash value H_P, and comparing this value with the hash value formed from $A_1A_2 \cdots A_k, A_2A_3 \cdots A_{k+1}, A_3A_4 \cdots A_{k+2}$, and so on until $A_{N-k+1}A_{N-k+2} \cdots A_N$. If we have a match of hash values, we compare the strings character by character to verify the match. We return the position (in A) if the strings actually do match, and we continue in the unlikely event that the match is false.

⋆ a. Show that if the hash value of $A_iA_{i+1} \cdots A_{i+k-1}$ is known, then the hash value of $A_{i+1}A_{i+2} \cdots A_{i+k}$ can be computed in constant time.
b. Show that the running time is $O(k + N)$ plus the time spent refuting false matches.
⋆ c. Show that the expected number of false matches is negligible.
d. Write a program to implement this algorithm.
⋆ e. Describe an algorithm that runs in $O(k + N)$ worst-case time.
⋆⋆ f. Describe an algorithm that runs in $O(N/k)$ average time.

5.16 A nonstandard C++ extension adds syntax that allows a switch statement to work with the `string` type (instead of the primitive integer types). Explain how hash tables can be used by the compiler to implement this language addition.

5.17 An (old-style) BASIC program consists of a series of statements numbered in ascending order. Control is passed by use of a *goto* or *gosub* and a statement number. Write a program that reads in a legal BASIC program and renumbers the statements so

that the first starts at number F and each statement has a number D higher than the previous statement. You may assume an upper limit of N statements, but the statement numbers in the input might be as large as a 32-bit integer. Your program must run in linear time.

5.18 a. Implement the word puzzle program using the algorithm described at the end of the chapter.
b. We can get a big speed increase by storing, in addition to each word W, all of W's prefixes. (If one of W's prefixes is another word in the dictionary, it is stored as a real word.) Although this may seem to increase the size of the hash table drastically, it does not, because many words have the same prefixes. When a scan is performed in a particular direction, if the word that is looked up is not even in the hash table as a prefix, then the scan in that direction can be terminated early. Use this idea to write an improved program to solve the word puzzle.
c. If we are willing to sacrifice the sanctity of the hash table ADT, we can speed up the program in part (b) by noting that if, for example, we have just computed the hash function for "excel," we do not need to compute the hash function for "excels" from scratch. Adjust your hash function so that it can take advantage of its previous calculation.
d. In Chapter 2, we suggested using binary search. Incorporate the idea of using prefixes into your binary search algorithm. The modification should be simple. Which algorithm is faster?

5.19 Under certain assumptions, the expected cost of an insertion into a hash table with secondary clustering is given by $1/(1-\lambda)-\lambda-\ln(1-\lambda)$. Unfortunately, this formula is not accurate for quadratic probing. However, assuming that it is, determine the following:
a. the expected cost of an unsuccessful search
b. the expected cost of a successful search

5.20 Implement a generic `Map` that supports the `insert` and `lookup` operations. The implementation will store a hash table of pairs (key, definition). You will `lookup` a definition by providing a key. Figure 5.56 provides the `Map` specification (minus some details).

5.21 Implement a spelling checker by using a hash table. Assume that the dictionary comes from two sources: an existing large dictionary and a second file containing a personal dictionary. Output all misspelled words and the line numbers on which they occur. Also, for each misspelled word, list any words in the dictionary that are obtainable by applying any of the following rules:
a. Add one character.
b. Remove one character.
c. Exchange adjacent characters.

5.22 Prove **Markov's Inequality**: If X is any random variable and $a > 0$, then $\Pr(|X| \geq a) \leq E(|X|)/a$. Show how this inequality can be applied to Theorems 5.2 and 5.3.

5.23 If a hopscotch table with parameter *MAX_DIST* has load factor 0.5, what is the approximate probability that an insertion requires a rehash?

```
1  template <typename HashedObj, typename Object>
2  class Pair
3  {
4      HashedObj key;
5      Object    def;
6      // Appropriate Constructors, etc.
7  };
8
9  template <typename HashedObj, typename Object>
10 class Dictionary
11 {
12   public:
13     Dictionary( );
14
15     void insert( const HashedObj & key, const Object & definition );
16     const Object & lookup( const HashedObj & key ) const;
17     bool isEmpty( ) const;
18     void makeEmpty( );
19
20   private:
21     HashTable<Pair<HashedObj,Object>> items;
22 };
```

Figure 5.56 Dictionary skeleton for Exercise 5.20

5.24 Implement a hopscotch hash table and compare its performance with linear probing, separate chaining, and cuckoo hashing.

5.25 Implement the classic cuckoo hash table in which two separate tables are maintained. The simplest way to do this is to use a single array and modify the hash function to access either the top half or the bottom half.

5.26 Extend the classic cuckoo hash table to use d hash functions.

5.27 Show the result of inserting the keys 10111101, 00000010, 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111, 10011110, 11011011, 00101011, 01100001, 11110000, 01101111 into an initially empty extendible hashing data structure with $M = 4$.

5.28 Write a program to implement extendible hashing. If the table is small enough to fit in main memory, how does its performance compare with separate chaining and open addressing hashing?

References

Despite the apparent simplicity of hashing, much of the analysis is quite difficult, and there are still many unresolved questions. There are also many interesting theoretical issues.

Hashing dates to at least 1953, when H. P. Luhn wrote an internal IBM memorandum that used separate chaining hashing. Early papers on hashing are [11] and [32]. A wealth of information on the subject, including an analysis of hashing with linear probing under the assumption of totally random and independent hashing, can be found in [25]. More recent results have shown that linear probing requires only 5-independent hash functions [31]. An excellent survey on early classic hash tables methods is [28]; [29] contains suggestions, and pitfalls, for choosing hash functions. Precise analytic and simulation results for separate chaining, linear probing, quadratic probing, and double hashing can be found in [19]. However, due to changes (improvements) in computer architecture and compilers, simulation results tend to quickly become dated.

An analysis of double hashing can be found in [20] and [27]. Yet another collision resolution scheme is coalesced hashing, described in [33]. Yao [37] has shown the uniform hashing, in which no clustering exists, is optimal with respect to cost of a successful search, assuming that items cannot move once placed.

Universal hash functions were first described in [5] and [35]; the latter paper introduces the "Carter-Wegman trick" of using Mersenne prime numbers to avoid expensive mod operations. Perfect hashing is described in [16], and a dynamic version of perfect hashing was described in [8]. [12] is a survey of some classic dynamic hashing schemes.

The $\Theta(\log N / \log \log N)$ bound on the length of the longest list in separate chaining was shown (in precise form) in [18]. The "power of two choices," showing that when the shorter of two randomly selected lists is chosen, then the bound on the length of the longest list is lowered to only $\Theta(\log \log N)$, was first described in [2]. An early example of the power of two choices is [4]. The classic work on cuckoo hashing is [30]; since the initial paper, a host of new results have appeared that analyze the amount of independence needed in the hash functions and describe alternative implementations [7], [34], [15], [10], [23], [24], [1], [6], [9] and [17]. Hopscotch hashing appeared in [21].

Extendible hashing appears in [13], with analysis in [14] and [36].

Exercise 5.15 (a–d) is from [22]. Part (e) is from [26], and part (f) is from [3]. The FNV-1a hash function described in Exercise 5.9 is due to Fowler, Noll, and Vo.

1. Y. Arbitman, M. Naor, and G. Segev, "De-Amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results," *Proceedings of the 36th International Colloquium on Automata, Languages and Programming* (2009), 107–118.
2. Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced Allocations," *SIAM Journal of Computing*, 29 (1999), 180–200.
3. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM,* 20 (1977), 762–772.
4. A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proceedings of the Twentieth IEEE INFOCOM* (2001), 1454–1463.
5. J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences,* 18 (1979), 143–154.
6. J. Cohen and D. Kane, "Bounds on the Independence Required for Cuckoo Hashing," preprint.
7. L. Devroye and P. Morin, "Cuckoo Hashing: Further Analysis," *Information Processing Letters*, 86 (2003), 215–219.

8. M. Dietzfelbinger, A. R. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal on Computing,* 23 (1994), 738–761.
9. M. Dietzfelbinger and U. Schellbach, "On Risks of Using Cuckoo Hashing with Simple Universal Hash Classes," *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (2009), 795–804.
10. M. Dietzfelbinger and C. Weidling, "Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins," *Theoretical Computer Science*, 380 (2007), 47–68.
11. I. Dumey, "Indexing for Rapid Random-Access Memory," *Computers and Automation*, 5 (1956), 6–9.
12. R. J. Enbody and H. C. Du, "Dynamic Hashing Schemes," *Computing Surveys,* 20 (1988), 85–113.
13. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems,* 4 (1979), 315–344.
14. P. Flajolet, "On the Performance Evaluation of Extendible Hashing and Trie Searching," *Acta Informatica,* 20 (1983), 345–369.
15. D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, 38 (2005), 229–248.
16. M. L. Fredman, J. Komlos, and E. Szemeredi, "Storing a Sparse Table with *O*(1) Worst Case Access Time," *Journal of the ACM,* 31 (1984), 538–544.
17. A. Frieze, P. Melsted, and M. Mitzenmacher, "An Analysis of Random-Walk Cuckoo Hashing," *Proceedings of the Twelfth International Workshop on Approximation Algorithms in Combinatorial Optimization (APPROX)* (2009), 350–364.
18. G. Gonnet, "Expected Length of the Longest Probe Sequence in Hash Code Searching," *Journal of the Association for Computing Machinery*, 28 (1981), 289–304.
19. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* 2d ed., Addison-Wesley, Reading, Mass., 1991.
20. L. J. Guibas and E. Szemeredi, "The Analysis of Double Hashing," *Journal of Computer and System Sciences,* 16 (1978), 226–274.
21. M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch Hashing," *Proceedings of the Twenty-Second International Symposium on Distributed Computing* (2008), 350–364.
22. R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *Aiken Computer Laboratory Report TR-31-81,* Harvard University, Cambridge, Mass., 1981.
23. A. Kirsch and M. Mitzenmacher, "The Power of One Move: Hashing Schemes for Hardware," *Proceedings of the 27th IEEE International Conference on Computer Communications (INFOCOM)* (2008), 106–110.
24. A. Kirsch, M. Mitzenmacher, and U. Wieder, "More Robust Hashing: Cuckoo Hashing with a Stash," *Proceedings of the Sixteenth Annual European Symposium on Algorithms* (2008), 611–622.
25. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* 2d ed., Addison-Wesley, Reading, Mass., 1998.
26. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing,* 6 (1977), 323–350.
27. G. Lueker and M. Molodowitch, "More Analysis of Double Hashing," *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988), 354–359.
28. W. D. Maurer and T. G. Lewis, "Hash Table Methods," *Computing Surveys,* 7 (1975), 5–20.

29. B. J. McKenzie, R. Harries, and T. Bell, "Selecting a Hashing Algorithm," *Software—Practice and Experience,* 20 (1990), 209–224.
30. R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, 51 (2004), 122–144.
31. M. Pătraşcu and M. Thorup, "On the *k*-Independence Required by Linear Probing and Minwise Independence," *Proceedings of the 37th International Colloquium on Automata, Languages, and Programming* (2010), 715–726.
32. W. W. Peterson, "Addressing for Random Access Storage," *IBM Journal of Research and Development,* 1 (1957), 130–146.
33. J. S. Vitter, "Implementations for Coalesced Hashing," *Communications of the ACM,* 25 (1982), 911–926.
34. B. Vöcking, "How Asymmetry Helps Load Balancing," *Journal of the ACM*, 50 (2003), 568–589.
35. M. N. Wegman and J. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," *Journal of Computer and System Sciences*, 22 (1981), 265–279.
36. A. C. Yao, "A Note on the Analysis of Extendible Hashing," *Information Processing Letters,* 11 (1980), 84–86.
37. A. C. Yao, "Uniform Hashing Is Optimal," *Journal of the ACM,* 32 (1985), 687–693.