```
template<>
class hash<string>
{
   public:
      size_t operator()(const string theKey) const
      {// Convert theKey to a nonnegative integer.
         unsigned long hashValue = 0;
         int length = (int) theKey.length();
         for (int i = 0; i < length; i++)
            hashValue = 5 * hashValue + theKey.at(i);

         return size_t(hashValue);
      }
};
```

Program 10.15 The specialization `hash<string>`

10.5.3 Linear Probing

The Method

The easiest way to find a place to put 58 into the table of Figure 10.2(a) is to search the table for the next available bucket and then put 58 into it. This method of handling overflows is called **linear probing** (also referred to as linear open addressing).

The 58 gets inserted into position 4. Suppose that the next key to be inserted is 24. 24 % 11 is 2. This bucket is empty, and so the 24 is placed there. Our hash table now has the form shown in Figure 10.2(b). Let us attempt to insert the key 35 into this table. Its home bucket (2) is full. Using linear probing, this key is placed in the next available bucket, and the table of Figure 10.2(c) results. As a final example, consider inserting 98 into the table. Its home bucket (10) is full. The next available bucket is 0, and the insertion is made into this bucket. So the search for the next available bucket is made by regarding the table as circular!

Having seen how insertions are made when linear probing is used, we can devise a method to search such a table. The search begins at the home bucket $f(k)$ of the key k we are searching for and continues by examining successive buckets in the table (regarding the table as circular) until one of the following happens: (1) a bucket containing a pair with key k is reached, in which case we have found the pair we were searching for; (2) an empty bucket is reached; and (3) we return to the home bucket. In the latter two cases, the table contains no pair with key k.

The deletion of a pair must leave behind a table on which the search method

table of Figure 10.2(c), we cannot simply make position 4 of the table `NULL`. Doing so will result in the search method failing to find the pair with key 35. A deletion may require us to move several pairs. The search for pairs to move begins just after the bucket vacated by the deleted pair and proceeds to successive buckets until we either reach an empty bucket or we return to the bucket from which the deletion took place. When pairs are moved up the table following a deletion, we must take care not to move a pair to a position before its home bucket because making such a pair move would cause the search for this pair to fail.

An alternative to this rather cumbersome deletion strategy is to introduce the field `neverUsed` in each bucket. When the table is initialized, this field is set to `true` for all buckets. When a pair is placed into a bucket, its `neverUsed` field is set to `false`. Now condition (2) for search termination is replaced by: a bucket with its `neverUsed` field equal to `true` is reached. We accomplish a removal by setting the table position occupied by the removed pair to `NULL`. A new pair may be inserted into the first empty bucket encountered during a search that begins at the pair's home bucket. Notice that in this alternative scheme, `neverUsed` is never reset to `true`. After a while all (or almost all) buckets have this field equal to `false`, and unsuccessful searches examine all buckets. To improve performance, we must reorganize the table when many empty buckets have their `neverUsed` field equal to `false`. This reorganization could, for example, involve reinserting all remaining pairs into an empty hash table.

C++ Implementation of Linear Probing

Program 10.16 gives the data members and the constructor for our hash table class `hashTable` that uses linear probing. Notice that the hash table is defined as a one-dimensional array `table[]` of type `pair<const K, E>*`.

Program 10.17 gives the method `search` of `hashTable`. This method returns a bucket `b` in the table that satisfies exactly one of the following: (1) `table[b]` points to a pair whose key is `theKey`; (2) no pair in the table has the key `theKey`, `table[b]` is `NULL`, and the pair with key `theKey` may be inserted into bucket `b` if desired; and (3) no pair in the table has the key `theKey`, `table[b]` has a key other than `theKey`, and the table is full.

Program 10.18 implements the method `hashTable<K,E>::find`.

Program 10.19 gives the implementation of the method `insert`. This code begins by invoking the method `search`. From the specification of `search`, if the returned bucket `b` is empty, then there is no pair in the table with key `thePair.first` and the pair `thePair` may be inserted into this bucket. If the returned bucket is not empty, then it either contains a pair with key `thePair.first` or the table is full. In the former case we change the second component of the pair stored in the bucket to `thePair.second`; in the latter, we throw an exception (increasing the table size is an alternative to throwing an exception; this alternative is considered in Exercise 25).

```
// data members of hashTable
pair<const K, E>** table;  // hash table
hash<K> hash;              // maps type K to nonnegative integer
int dSize;                 // number of pairs in dictionary
int divisor;               // hash function divisor

// constructor
template<class K, class E>
hashTable<K,E>::hashTable(int theDivisor)
{
   divisor = theDivisor;
   dSize = 0;

   // allocate and initialize hash table array
   table = new pair<const K, E>* [divisor];
   for (int i = 0; i < divisor; i++)
      table[i] = NULL;
}
```

Program 10.16 Data members and constructor for hashTable

```
template<class K, class E>
int hashTable<K,E>::search(const K& theKey) const
{// Search an open addressed hash table for a pair with key theKey.
 // Return location of matching pair if found, otherwise return
 // location where a pair with key theKey may be inserted
 // provided the hash table is not full.

   int i = (int) hash(theKey) % divisor;  // home bucket
   int j = i;    // start at home bucket
   do
   {
      if (table[j] == NULL || table[j]->first == theKey)
         return j;
      j = (j + 1) % divisor;  // next bucket
   } while (j != i);          // returned to home bucket?

   return j;  // table full
}
```

Program 10.17 The method hashTable<K,E>::search

```
template<class K, class E>
pair<const K,E>* hashTable<K,E>::find(const K& theKey) const
{// Return pointer to matching pair.
 // Return NULL if no matching pair.
   // search the table
   int b = search(theKey);

   // see if a match was found at table[b]
   if (table[b] == NULL || table[b]->first != theKey)
      return NULL;            // no match

   return table[b];  // matching pair
}
```

Program 10.18 The method `hashTable<K,E>::find`

Performance Analysis

We will analyze the time complexity only. Let b be the number of buckets in the hash table. When division with divisor D is used as the hash function, $b = D$. The time needed to initialize the table is $O(b)$. The worst-case insert and find time is $\Theta(n)$ when n pairs are present in the table. The worst case happens, for instance, when all n key values have the same home bucket. Comparing the worst-case complexity of hashing to that of the linear list method to maintain a dictionary, we see that both have the same worst-case complexity.

For average performance, however, hashing is considerably superior. Let U_n and S_n, respectively, denote the average number of buckets examined during an unsuccessful and a successful search when n is large. This average is defined over all possible sequences of n key values being inserted into the table. For linear probing, it can be shown that

$$U_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \tag{10.3}$$

$$S_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \tag{10.4}$$

where $\alpha = n/b$ is the **loading factor**. Although Equation 10.3 is rather difficult to derive, Equation 10.4 can be derived from Equation 10.3 with modest effort (Exercise 21a).

From Equations 10.3 and 10.4, it follows that when $\alpha = 0.5$, an unsuccessful search will examine 2.5 buckets on the average and an average successful search

```
template<class K, class E>
void hashTable<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the dictionary. Overwrite existing
 // pair, if any, with same key.
 // Throw hashTableFull exception in case table is full.
   // search the table for a matching element
   int b = search(thePair.first);

   // check if matching element found
   if (table[b] == NULL)
   {
      // no matching element and table not full
      table[b] = new pair<const K,E> (thePair);
      dSize++;
   }
   else
   {// check if duplicate or table full
      if (table[b]->first == thePair.first)
      {// duplicate, change table[b]->second
         table[b]->second = thePair.second;
      }
      else // table is full
         throw hashTableFull();
   }
}
```

Program 10.19 The method `hashTable<K,E>::insert`

will examine 1.5 buckets. When $\alpha = 0.9$, these figures are 50.5 and 5.5. These figures, of course, assume that n is much larger than 51. When it is possible to work with small loading factors, the average performance of hashing with linear probing is significantly superior to that of the linear list method. Generally, when linear probing is used, we try to keep $\alpha \leq 0.75$.

Analysis of Random Probing

To give you a taste of what is involved in determining U_n and S_n, we derive U_n and S_n formulas for the random probing method to handle overflows. In random probing, when an overflow occurs, the search for a free bucket in which the new key is inserted is done in a random manner (in practice, a pseudorandom number generator is used so we can reproduce the bucket search sequence and use this

Our derivation of the formula for U_n makes use of the following result from probability theory.

Theorem 10.1 *Let p be the probability that a certain event occurs. The expected number of independent trials needed for that event to occur is $1/\alpha$.*

To get a feel for the validity of Theorem 10.1, suppose that you flip a coin. The probability that the coin lands heads up is $p = 1/2$. The number of times you expect to flip the coin before it lands heads up is $1/p = 2$. A die has six sides labeled 1 through 6. When you throw a die, the probability of drawing an odd number is $p = 1/2$. You expect to throw the die $1/p = 2$ times before drawing an odd number. The probability that a die throw draws a 6 is $p = 1/6$, so you expect to throw the die $1/p = 6$ times before drawing a 6.

The formula for U_n is derived as follows. When the loading density is α, the probability that any bucket is occupied is also α. Therefore, the probability that a bucket is empty is $p = 1 - \alpha$. In random probing an unsuccessful search looks for an empty bucket, using a sequence of independent trials. Therefore, the expected number of buckets examined is

$$U_n \approx \frac{1}{p} = \frac{1}{1-\alpha} \tag{10.5}$$

The equation for S_n may be derived from that for U_n. Number the n pairs in the table 1, 2, $\cdots$, n in the order they were inserted. When the ith pair is inserted, an unsuccessful search is done and the pair is inserted into the empty bucket where the unsuccessful search terminates. At the time the ith pair is inserted, the loading factor is $(i-1)/b$ where b is the number of buckets. From Equation 10.5 it follows that the expected number of buckets that are to be examined when searching for the ith pair is

$$\frac{1}{1-\frac{i-1}{b}}$$

Assuming that each pair in the table is searched for with equal probability, we get

$$\begin{aligned} S_n &\approx \frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i-1}{b}} \\ &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{b}} \\ &\approx \frac{1}{n}\int_{0}^{n-1}\frac{1}{1-\frac{i}{b}}\,di \end{aligned}$$

$$\begin{aligned} &\approx \frac{1}{n}\int_{i=0}^{n}\frac{1}{1-\frac{i}{b}}di \\ &= -\frac{b}{n}\log_e(1-i/b)\Big]_0^n \\ &= -\frac{1}{\alpha}\log_e(1-\alpha) \end{aligned} \tag{10.6}$$

Linear probing incurs a performance penalty relative to random probing as far as the number of examined buckets is concerned. For example, when $\alpha = 0.9$, an unsuccessful search using linear probing is expected to examine 50.5 buckets; when random probing is used, this expected number drops to 10. So why do we not use random probing? Here are two reasons:

- Our real interest is run time, not number of buckets examined. It takes more time to compute the next random number than it does to examine several buckets.
- Since random probing searches the table in a random fashion, it pays a run-time penalty because of the cache effect (Section 4.5). Therefore, even though random probing examines a smaller number of buckets than does linear probing, examining this smaller number of buckets actually takes more time except when the loading factor is close to 1.

Choosing a Divisor D

To determine D, we first determine what constitutes acceptable performance for unsuccessful and successful searches. Using the formulas for U_n and S_n, we can determine the largest α that can be used. From the value of n (or an estimate) and the computed value of α, we obtain the smallest permissible value for b. Next we find the smallest integer that is at least as large as this value of b and that either is a prime or has no factors smaller than 20. This integer is the value of D and b to use.

Example 10.12 We are to design a hash table for up to 1000 pairs. Successful searches should require no more than four bucket examinations on average, and unsuccessful searches should examine no more than 50.5 buckets on average. From the formula for U_n, we obtain $\alpha \leq 0.9$, and from that for S_n, we obtain $4 \geq 0.5 + 1/(2(1\text{-}\alpha))$ or $\alpha \leq 6/7$. Therefore, we require $\alpha \leq \min\{0.9, 6/7\} = 6/7$. Hence b should be at least $\lceil 7n/6 \rceil = 1167$. $b = D = 1171$ is a suitable choice. ■

Another way to compute D is to begin with a knowledge of the largest possible value for b as determined by the maximum amount of space available for the hash table. Now we find the largest D no larger than this largest value that is either a prime or has no factors smaller than 20. For instance, if we can allot at most 530