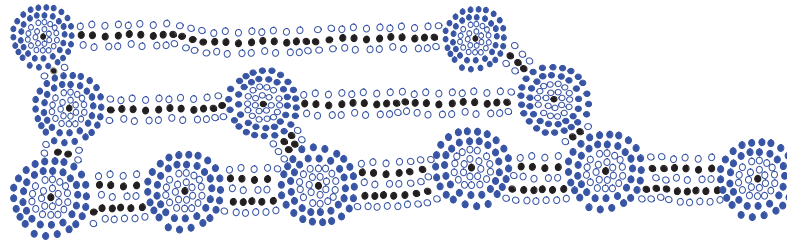
Chapter 9 Hash Tables, Maps, and Skip Lists



Contents

9.1 Maps

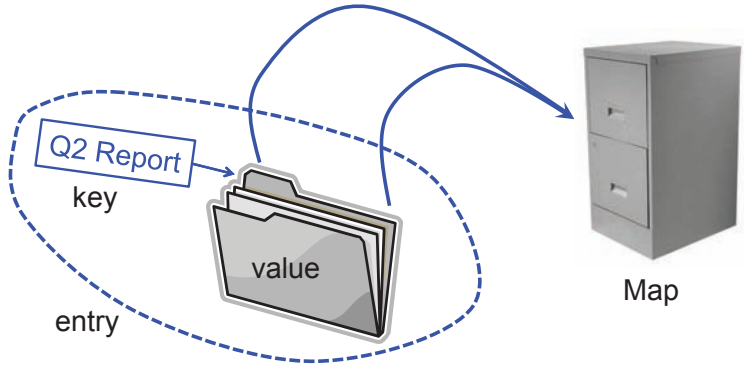


Figure 9.1: A conceptual illustration of the map ADT. Keys (labels) are assigned to values (folders) by a user. The resulting entries (labeled folders) are inserted into the map (file cabinet). The keys can be used later to retrieve or remove values.

A ***map*** allows us to store elements so they can be located quickly using keys. The motivation for such searches is that each element typically stores additional useful information besides its search key, but the only way to get at that information is to use the search key. Specifically, a map stores key-value pairs (k, v), which we call ***entries***, where k is the key and v is its corresponding value. In addition, the map ADT requires that each key be unique, so the association of keys to values defines a mapping. In order to achieve the highest level of generality, we allow both the keys and the values stored in a map to be of any object type. (See Figure 9.1.) In a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number. In some applications, the key and the value may be the same. For example, if we had a map storing prime numbers, we could use each number itself as both a key and its value.

practical use ??? ←

In either case, we use a ***key*** as a unique identifier that is assigned by an application or user to an associated value object. Thus, a map is most appropriate in situations where each key is to be viewed as a kind of unique ***index*** address for its value, that is, an object that serves as a kind of location for that value. For example, if we wish to store student records, we would probably want to use student ID objects as keys (and disallow two students having the same student ID). In other words, the key associated with an object can be viewed as an "address" for that object. Indeed, maps are sometimes referred to as ***associative stores*** or ***associative containers***, because the key associated with an object determines its "location" in the data structure.

Entries and the Composition Pattern

As mentioned above, a map stores key-value pairs, called entries. An entry is actually an example of a more general object-oriented design pattern, the ***composition pattern***, which defines a single object that is composed of other objects. A pair is the simplest composition, because it combines two objects into a single pair object.

To implement this concept, we define a class that stores two objects in its first and second member variables, respectively, and provides functions to access and update these variables. In Code Fragment 9.1, we present such an implementation storing a single key-value pair. We define a class Entry, which is templated based on the key and value types. In addition to a constructor, it provides member functions that return references to the key and value. It also provides functions that allow us to set the key and value members.

```
template <typename K, typename V>
class Entry {                                   // a (key, value) pair
public:                                         // public functions
  Entry(const K& k = K(), const V& v = V())     // constructor
    : _key(k), _value(v) { }
  const K& key() const { return _key; }         // get key
  const V& value() const { return _value; }     // get value
  void setKey(const K& k) { _key = k; }         // set key
  void setValue(const V& v) { _value = v; }     // set value
private:                                        // private data
  K _key;                                       // key
  V _value;                                     // value
};
```

Code Fragment 9.1: A C++ class for an entry storing a key-value pair.

9.1.1 The Map ADT

In this section, we describe a map ADT. Recall that a map is a collection of key-value entries, with each value associated with a distinct key. We assume that a map provides a special pointer object, which permits us to reference entries of the map. Such an object would normally be called a ***position***. As we did in Chapter 6, in order to be more consistent with the C++ Standard Template Library, we define a somewhat more general object called an ***iterator***, which can both reference entries and navigate around the map. Given a map iterator p, the associated entry may be accessed by dereferencing the iterator, namely as $*p$. The individual key and value can be accessed using p->key() and p->value(), respectively.

In order to advance an iterator from its current position to the next, we overload the increment operator. Thus, $++p$ advances the iterator p to the next entry of the

map. We can enumerate all the entries of a map M by initializing p to M.begin() and then repeatedly incrementing p as long as it is not equal to M.end().

In order to indicate that an object is not present in the map, we assume that there exists a special sentinel iterator called end. By convention, this sentinel refers to an imaginary element that lies just beyond the last element of the map.

The map ADT consists of the following:

size(): Return the number of entries in M.

empty(): Return true if M is empty and false otherwise.

find(k): If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.

put(k, v): If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.

erase(k): Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.

erase(p): Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

begin(): Return an iterator to the first entry of M.

end(): Return an iterator to a position just beyond the end of M.

We have provided two means of removing entries, one given a key and the other given an iterator. The key-based operation should be used only when it is known that the key is present in the map. Otherwise, it is necessary to first check that the key exists using the operation "$p = M$.find(k)," and if so, then apply the operation M.erase(p). The iterator-based removal operation has the advantage that it does not need to repeat the search for the key, and hence is more efficient.

The operation put, may either insert an entry or modify an existing entry. It is designed explicitly in this way, since we require that the keys be unique. Later, in Section 9.5, we consider a different data structure, which allows multiple instances to have the same keys. Note that an iterator remains associated with an entry, even if its value is changed.

Example 9.1: *In the following, we show the effect of a series of operations on an initially empty map storing entries with integer keys and single-character values. In the column "Output," we use the notation $p_i : [(k, v)]$ to mean that the operation returns an iterator denoted by p_i that refers to the entry (k, v). The entries of the map are not listed in any particular order.*

Operation	Output	Map
empty()	**true**	$\emptyset$
put$(5,A)$	$p_1 : [(5,A)]$	$\{(5,A)\}$
put$(7,B)$	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
put$(2,C)$	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
put$(2,E)$	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,E)\}$
find(4)	end	$\{(5,A),(7,B),(2,E)\}$
find(2)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
size()	3	$\{(5,A),(7,B),(2,E)\}$
erase(5)	–	$\{(7,B),(2,E)\}$
erase(p_3)	–	$\{(7,B)\}$
find(2)	end	$\{(7,B)\}$

9.1.2 A C++ Map Interface

Before discussing specific implementations of the map ADT, we first define a C++ interface for a map in Code Fragment 9.2. It is not a complete C++ class, just a declaration of the public functions. The interface is templated by two types, the key type K, and the value type V.

```
template <typename K, typename V>
class Map {                                  // map interface
public:
  class Entry;                               // a (key,value) pair
  class Iterator;                            // an iterator (and position)

  int size() const;                          // number of entries in the map
  bool empty() const;                        // is the map empty?
  Iterator find(const K& k) const;           // find entry with key k
  Iterator put(const K& k, const V& v);      // insert/replace pair (k,v)
  void erase(const K& k)                     // remove entry with key k
    throw(NonexistentElement);
  void erase(const Iterator& p);             // erase entry at p
  Iterator begin();                          // iterator to first entry
  Iterator end();                            // iterator to end entry
};
```

Code Fragment 9.2: An informal C++ Map interface (not a complete class).

In addition to its member functions, the interface defines two types, Entry and Iterator. These two classes provide the types for the entry and iterator objects, respectively. Outside the class, these would be accessed with Map<K,V>::Entry and Map<K,V>::Iterator, respectively.

We have not presented an interface for the iterator object, but its definition is similar to the STL iterator. It supports the operator "*", which returns a reference to the associated entry. The unary increment and decrement operators "++" and "−−" move an iterator forward and backwards, respectively. Also, two iterators can be compared for equality using "==".

A more sophisticated implementation would have also provided for a third type, namely a "const" iterator. Such an iterator provides a function for reading entries without modifying them. (Recall Section 6.1.4.) We omit this type in order to keep our interface relatively simple.

The remainder of the interface follows from our earlier descriptions of the map operations. An error condition occurs if the function erase(k) is called with a key k that is not in the map. This is signaled by throwing an exception of type NonexistentElement. Its definition is similar to other exceptions that we have seen. (See Code Fragment 5.2.)

9.1.3 The STL map Class

The C++ Standard Template Library (STL) provides an implementation of a map simply called map. As with many of the other STL classes we have seen, the STL map is an example of a container, and hence supports access by iterators.

In order to declare an object of type map, it is necessary to first include the definition file called "map." The map is part of the std namespace, and hence it is necessary either to use "std::map" or to provide an appropriate "**using**" statement.

The STL map is templated with two arguments, the key type and the value type. The declaration "map<K,V>" defines a map whose keys are of type K and whose values are of type V. As with the other STL containers, an iterator type is provided both for referencing individual entries and enumerating multiple entries. The map iterator type is "map<K,E>::iterator." The (k, v) entries are stored in a composite object called pair. Given an iterator p, its associated key and value members can be referenced using p->*first* and p->*second*, respectively. (These are equivalent to p->key() and p->value() in our map ADT, but note that there are no parentheses following *first* and *second*.)

As with other iterators we have seen, each map object M defines two special iterators through the functions begin and end, where M.begin() yields an iterator to the first element of the map, and M.end() yields an iterator to an imaginary element just beyond the end of the map. A map iterator p is bidirectional, meaning that we can move forwards and backwards through the map using the increment and decrement operators, $++p$ and $--p$, respectively.

The principal member functions of the STL map are given below. Let M be declared to be an STL map, let k be a key object, and let v be a value object for the class M. Let p be an iterator for M.

size(): Return the number of elements in the map.
empty(): Return true if the map is empty and false otherwise.
find(k): Find the entry with key k and return an iterator to it; if no such key exists return end.
operator[k]: Produce a reference to the value of key k; if no such key exists, create a new entry for key k.
insert(pair(k,v)): Insert pair (k,v), returning an iterator to its position.
erase(k): Remove the element with key k.
erase(p): Remove the element referenced by iterator p.
begin(): Return an iterator to the beginning of the map.
end(): Return an iterator just past the end of the map.

Our map ADT is quite similar to the above functions. The insert function is a bit different. In our ADT, it is given two arguments. In the STL map, the argument is a composite object of type pair, whose first and second elements are the key and value, respectively.

The STL map provides a very convenient way to search, insert, and modify entries by overloading the subscript operator ("[ ]"). Given a map M, the assignment "$M[k] = v$" inserts the pair (k,v) if k is not already present, or modifies the value if it is. Thus, the subscript assignment behaves essentially the same as our ADT function put(k,v). Reading the value of $M[k]$ is equivalent to performing find(k) and accessing the value part of the resulting iterator. An example of the use of the STL map is shown in Code Fragment 9.3.

```
map<string, int> myMap;                          // a (string,int) map
map<string, int>::iterator p;                    // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28));      // insert ("Rob",28)
myMap["Joe"] = 38;                               // insert("Joe",38)
myMap["Joe"] = 50;                               // change to ("Joe",50)
myMap["Sue"] = 75;                               // insert("Sue",75)
p = myMap.find("Joe");                           // *p = ("Joe",50)
myMap.erase(p);                                  // remove ("Joe",50)
myMap.erase("Sue");                              // remove ("Sue",75)
p = myMap.find("Joe");
if (p == myMap.end()) cout << "nonexistent\n"; // outputs: "nonexistent"
for (p = myMap.begin(); p != myMap.end(); ++p) { // print all entries
  cout << "(" << p->first << "," << p->second << ")\n";
}
```

Code Fragment 9.3: Example of the usage of STL map.

As with the other STL containers we have seen, the STL does not check for errors. It is up to the programmer to be sure that no illegal operations are performed.

9.1.4 A Simple List-Based Map Implementation

A simple way of implementing a map is to store its n entries in a list L, implemented as a doubly linked list. Performing the fundamental functions, find(k), put(k,v), and erase(k), involves simple scans down L looking for an entry with key k. Pseudo-code is presented in Code Fragments 9.4. We use the notation $[L.\text{begin}(), L.\text{end}())$ to denote all the positions of list L, from $L.\text{begin}()$ and up to, but not including, $L.\text{end}()$.

Algorithm find(k):
Input: A key k
Output: The position of the matching entry of L, or end if there is no key k in L
for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
  if $p.\text{key}() = k$ **then**
    return p
return end {there is no entry with key equal to k}

Algorithm put(k,v):
Input: A key-value pair (k,v)
Output: The position of the inserted/modified entry
for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
  if $p.\text{key}() = k$ **then**
    $*p \leftarrow (k,v)$
    return p {return the position of the modified entry}
$p \leftarrow L.\text{insertBack}((k,v))$
$n \leftarrow n+1$ {increment variable storing number of entries}
return p {return the position of the inserted entry}

Algorithm erase(k):
Input: A key k
Output: None
for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
  if $p.\text{key}() = k$ **then**
    $L.\text{erase}(p)$
    $n \leftarrow n-1$ {decrement variable storing number of entries}

Code Fragment 9.4: Algorithms for find, put, and erase for a map stored in a list L.

This list-based map implementation is simple, but it is only efficient for very small maps. Every one of the fundamental functions takes $O(n)$ time on a map with n entries, because each function involves searching through the entire list in the worst case. Thus, we would like something much faster.

9.2 Hash Tables

The keys associated with values in a map are typically thought of as "addresses" for those values. Examples of such applications include a compiler's symbol table and a registry of environment variables. Both of these structures consist of a collection of symbolic names where each name serves as the "address" for properties about a variable's type and value. One of the most efficient ways to implement a map in such circumstances is to use a ***hash table***. Although, as we will see, the worst-case running time of map operations in an n-entry hash table is $O(n)$. A hash table can usually perform these operations in $O(1)$ expected time. In general, a hash table consists of two major components, a ***bucket array*** and a ***hash function***.

9.2.1 Bucket Arrays

A ***bucket array*** for a hash table is an array A of size N, where each cell of A is thought of as a "bucket" (that is, a collection of key-value pairs) and the integer N defines the ***capacity*** of the array. If the keys are integers well distributed in the range $[0, N-1]$, this bucket array is all that is needed. An entry e with key k is simply inserted into the bucket $A[k]$. (See Figure 9.2.)

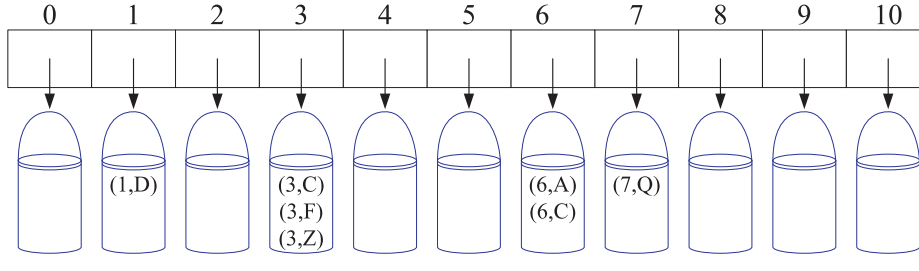


Figure 9.2: A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

If our keys are unique integers in the range $[0, N-1]$, then each bucket holds at most one entry. Thus, searches, insertions, and removals in the bucket array take $O(1)$ time. This sounds like a great achievement, but it has two drawbacks. First, the space used is proportional to N. Thus, if N is much larger than the number of entries n actually present in the map, we have a waste of space. The second drawback is that keys are required to be integers in the range $[0, N-1]$, which is often not the case. Because of these two drawbacks, we use the bucket array in conjunction with a "good" mapping from the keys to the integers in the range $[0, N-1]$.

negative, but we want the set of hash codes assigned to our keys to avoid collisions as much as possible. If the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In addition, to be consistent with all of our keys, the hash code we use for a key k should be the same as the hash code for any key that is equal to k.

Hash Codes in C++

The hash codes described below are based on the assumption that the number of bits of each type is known. This information is provided in the standard include file <limits>. This include file defines a templated class numeric_limits. Given a base type T (such as **char**, **int**, or **float**), the number of bits in a variable of type T is given by "numeric_limits<T>.digits." Let us consider several common data types and some example functions for assigning hash codes to objects of these types.

Converting to an Integer

To begin, we note that, for any data type X that is represented using at most as many bits as our integer hash codes, we can simply take an integer interpretation of its bits as a hash code for X. Thus, for the C++ fundamental types **char**, **short**, and **int**, we can achieve a good hash code simply by casting this type to **int**.

On many machines, the type **long** has a bit representation that is twice as long as type **int**. One possible hash code for a **long** object is to simply cast it down to an integer and then apply the integer hash code. The problem is that such a hash code ignores half of the information present in the original value. If many of the keys in our map only differ in these bits, they will collide using this simple hash code. A better hash code, which takes all the original bits into consideration, sums an integer representation of the high-order bits with an integer representation of the low-order bits.

Indeed, the approach of summing components can be extended to any object x whose binary representation can be viewed as a k-tuple $(x_0, x_1, \ldots, x_{k-1})$ of integers, because we can then form a hash code for x as $\sum_{i=0}^{k-1} x_i$. For example, given any floating-point number, we can sum its mantissa and exponent as long integers, and then apply a hash code for long integers to the result.

Polynomial Hash Codes

The summation hash code, described above, is not a good choice for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$, where the order of the x_i's is significant. For example, consider a hash code for a character string s that sums the ASCII values of the characters

in s. Unfortunately, this hash code produces lots of unwanted collisions for common groups of strings. In particular, `"temp01"` and `"temp10"` collide using this function, as do `"stop"`, `"tops"`, `"pots"`, and `"spot"`. A better hash code takes into consideration the positions of the x_i's. An alternative hash code, which does exactly this, chooses a nonzero constant, $a \neq 1$, and uses

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

as a hash code value. Mathematically speaking, this is simply a polynomial in a that takes the components $(x_0, x_1, \ldots, x_{k-1})$ of an object x as its coefficients. This hash code is therefore called a ***polynomial hash code***. By Horner's rule (see Exercise C-4.16), this polynomial can be rewritten as

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_2 + a(x_1 + a x_0)) \cdots)).$$

Intuitively, a polynomial hash code uses multiplication by the constant a as a way of "making room" for each component in a tuple of values, while also preserving a characterization of the previous components. Of course, on a typical computer, evaluating a polynomial is done using the finite bit representation for a hash code; hence, the value periodically overflows the bits used for an integer. Since we are more interested in a good spread of the object x with respect to other keys, we simply ignore such overflows. Still, we should be mindful that such overflows are occurring and choose the constant a so that it has some nonzero, low-order bits, which serve to preserve some of the information content even if we are in an overflow situation.

We have done some experimental studies that suggest that 33, 37, 39, and 41 are good choices for a when working with character strings that are English words. In fact, in a list of over 50,000 English words formed as the union of the word lists provided in two variants of Unix, we found that taking a to be 33, 37, 39, or 41 produced less than seven collisions in each case! Many implementations of string hashing choose a polynomial hash function, using one of these constants for a, as a default hash code. For the sake of speed, however, some implementations only apply the polynomial hash function to a fraction of the characters in long strings.

Cyclic Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits. Such a function, applied to character strings in C++ could, for example, look like the following. We assume a 32-bit integer word length, and we assume access to a function hashCode(x) for integers. To achieve a 5-bit cyclic shift we form the "bitwise or" (see Section 1.2) of a 5-bit left shift and a 27-bit right shift. As before, we use an unsigned integer so that right shifts fill with zeros.

```
int hashCode(const char* p, int len) {          // hash a character array
  unsigned int h = 0;
  for (int i = 0; i < len; i++) {
    h = (h << 5) | (h >> 27);                   // 5-bit cyclic shift
    h += (unsigned int) p[i];                   // add in next character
  }
  return hashCode(int(h));
}
```

As with the traditional polynomial hash code, using the cyclic-shift hash code requires some fine-tuning. In this case, we must wisely choose the amount to shift by for each new character.

Experimental Results

In Table 9.1, we show the results of some experiments run on a list of just over 25,000 English words, which compare the number of collisions for various shift amounts.

	Collisions			*Collisions*	
Shift	***Total***	***Max***	***Shift***	***Total***	***Max***
0	23739	86	9	18	2
1	10517	21	10	277	3
2	2254	6	11	453	4
3	448	3	12	43	2
4	89	2	13	13	2
5	4	2	14	135	3
6	6	2	15	1082	6
7	14	2	16	8760	9
8	105	2			

Table 9.1: Comparison of collision behavior for the cyclic shift variant of the polynomial hash code as applied to a list of just over 25,000 English words. The "Total" column records the total number of collisions and the "Max" column records the maximum number of collisions for any one hash code. Note that, with a cyclic shift of 0, this hash code reverts to the one that simply sums all the characters.

These and our previous experiments show that if we choose our constant a or our shift value wisely, then either the polynomial hash code or its cyclic-shift variant are suitable for any object that can be written as a tuple $(x_0, x_1, \ldots, x_{k-1})$,

where the order in tuples matters. In particular, note that using a shift of 5 or 6 is particularly good for English words. Also, note how poorly a simple addition of the values would be with no shifting (that is, for a shift of 0).

Hashing Floating-Point Quantities

On most machines, types **int** and **float** are both 32-bit quantities. Nonetheless, the approach of casting a **float** variable to type **int** would not produce a good hash function, since this would truncate the fractional part of the floating-point value. For the purposes of hashing, we do not really care about the number's value. It is sufficient to treat the number as a sequence of bits. Assuming that a **char** is stored as an 8-bit byte, we could interpret a 32-bit **float** as a four-element character array, and a 64-bit **double** as an eight-element character array. C++ provides an operation called a ***reinterpret cast***, to cast between such unrelated types. This cast treats quantities as a sequence of bits and makes no attempt to intelligently convert the meaning of one quantity to another.

For example, we could design a hash function for a **float** by first reinterpreting it as an array of characters and then applying the character-array hashCode function defined above. We use the operator **sizeof**, which returns the number of bytes in a type.

```
int hashCode(const float& x) {                    // hash a float
  int len = sizeof(x);
  const char* p = reinterpret_cast<const char*>(&x);
  return hashCode(p, len);
}
```

Reinterpret casts are generally not portable operations, since the result depends on the particular machine's encoding of types as a pattern of bits. In our case, portability is not an issue since we are interested only in interpreting the floating point value as a sequence of bits. The only property that we require is that float variables with equal values must have the same bit sequence.

9.2.4 Compression Functions

We can built it within the hash code function

The hash code for a key k is typically not suitable for immediate use with a bucket array, because the range of possible hash codes for our keys typically exceeds the range of legal indices of our bucket array A. That is, incorrectly using a hash code as an index into our bucket array may result in an error condition, either because the index is negative or it exceeds the capacity of A. Thus, once we have determined an integer hash code for a key object k, there is still the issue of mapping that integer

into the range $[0, N-1]$. This compression step is the second action that a hash function performs.

The Division Method

One simple ***compression function*** to use is

$$h(k) = |k| \bmod N,$$

which is called the ***division method***. Additionally, if we take N to be a prime number, then this hash function helps "spread out" the distribution of hashed values. Indeed, if N is not prime, there is a higher likelihood that patterns in the distribution of keys will be repeated in the distribution of hash codes, thereby causing collisions. For example, if we hash the keys $\{200, 205, 210, 215, 220, \ldots, 600\}$ to a bucket array of size 100 using the division method, then each hash code collides with three others. But if this same set of keys is similarly hashed to a bucket array of size 101, then there are no collisions. If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$. Choosing N to be a prime number is not always enough, however, because if there is a repeated pattern of key values of the form $iN + j$ for several different i's, then there are still collisions.

The MAD Method

A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys is the ***multiply add and divide*** (or "MAD") method. In using this method, we define the compression function as

$$h(k) = |ak + b| \bmod N,$$

where N is a prime number, and a and b are nonnegative integers randomly chosen at the time the compression function is determined, so that $a \bmod N \neq 0$. This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and to get us closer to having a "good" hash function, that is, one having the probability that any two different keys collide is $1/N$. This good behavior would be the same as if these keys were "thrown" into A uniformly at random.

With a compression function such as this, that spreads n integers fairly evenly in the range $[0, N-1]$, and a mapping of the keys in our map to integers, we have an effective hash function. Together, such a hash function and a bucket array define the key ingredients of the hash table implementation of the map ADT.

But before we can give the details of how to perform such operations as find, insert, and erase, we must first resolve the issue of how we to handle collisions.

9.2.5 Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each entry (k,v) in the "bucket" $A[h(k)]$. This simple idea is challenged, however, when we have two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$. The existence of such ***collisions*** prevents us from simply inserting a new entry (k,v) directly in the bucket $A[h(k)]$. Collisions also complicate our procedure for performing the find(k), put(k,v), and erase(k) operations.

Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket $A[i]$ store a small map, M_i, implemented using a list, as described in Section 9.1.4, holding entries (k,v) such that $h(k) = i$. That is, each separate M_i chains together the entries that hash to index i in a linked list. This ***collision-resolution*** rule is known as ***separate chaining***. Assuming that we initialize each bucket $A[i]$ to be an empty list-based map, we can easily use the separate-chaining rule to perform the fundamental map operations as shown in Code Fragment 9.5.

Algorithm find(k):
  Output: The position of the matching entry of the map, or end if there is no key k in the map
  return $A[h(k)]$.find(k) {delegate the find(k) to the list-based map at $A[h(k)]$}

Algorithm put(k,v):
  $p \leftarrow A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
  $n \leftarrow n+1$
  return p

Algorithm erase(k):
  Output: None
  $A[h(k)]$.erase(k) {delegate the erase to the list-based map at $A[h(k)]$}
  $n \leftarrow n-1$

Code Fragment 9.5: The fundamental functions of the map ADT, implemented with a hash table that uses separate chaining to resolve collisions among its n entries.

For each fundamental map operation involving a key k, the separate-chaining approach delegates the handling of this operation to the miniature list-based map stored at $A[h(k)]$. So, put(k,v) scans this list looking for an entry with key equal to k; if it finds one, it replaces its value with v, otherwise, it puts (k,v) at the end of this list. Likewise, find(k) searches through this list until it reaches the end or

finds an entry with key equal to k. And erase(k) performs a similar search but additionally removes an entry after it is found. We can "get away" with this simple list-based approach because the spreading properties of the hash function help keep each bucket's list small. Indeed, a good hash function tries to minimize collisions as much as possible, which implies that most of our buckets are either empty or store just a single entry. In Figure 9.4, we give an illustration of a hash table with separate chaining.

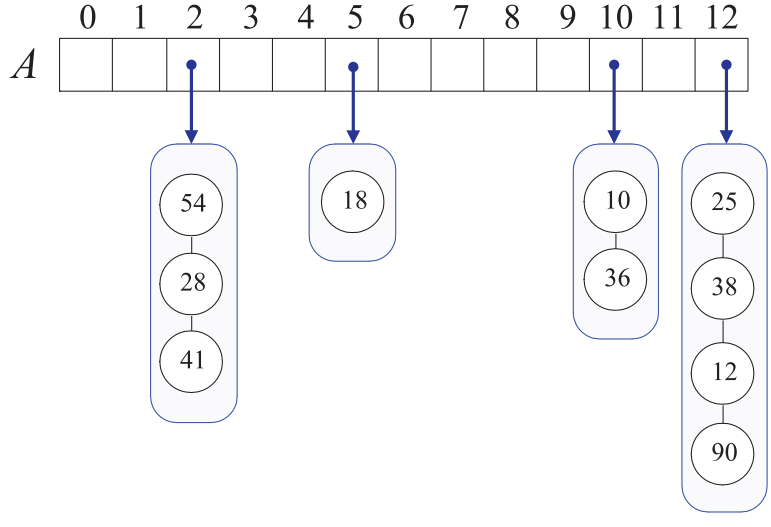


Figure 9.4: A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, we expect each bucket to be of size n/N. This value, called the ***load factor*** of the hash table (and denoted with λ), should be bounded by a small constant, preferably below 1. Given a good hash function, the expected running time of operations find, put, and erase in a map implemented with a hash table that uses this function is $O(\lceil n/N \rceil)$. Thus, we can implement these operations to run in $O(1)$ expected time provided n is $O(N)$.

Open Addressing

The separate-chaining rule has many nice properties, such as allowing for simple implementations of map operations, but it nevertheless has one slight disadvantage. It requires the use of an auxiliary data structure—a list—to hold entries with colliding keys. We can handle collisions in other ways besides using the separate-

chaining rule, however. In particular, if space is at a premium (for example, if we are writing a program for a small handheld device), then we can use the alternative approach of always storing each entry directly in a bucket, at most one entry per bucket. This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions. There are several variants of this approach, collectively referred to as ***open-addressing*** schemes, which we discuss next. Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.

Linear Probing and its Variants

A simple open-addressing method for collision handling is ***linear probing***. In this method, if we try to insert an entry (k,v) into a bucket $A[i]$ that is already occupied (where $i = h(k)$), then we try next at $A[(i+1) \bmod N]$. If $A[(i+1) \bmod N]$ is also occupied, then we try $A[(i+2) \bmod N]$, and so on, until we find an empty bucket that can accept the new entry. Once this bucket is located, we simply insert the entry there. Of course, this collision-resolution strategy requires that we change the implementation of the $\mathsf{get}(k,v)$ operation. In particular, to perform such a search, followed by either a replacement or insertion, we must examine consecutive buckets, starting from $A[h(k)]$, until we either find an entry with key equal to k or we find an empty bucket. (See Figure 9.5.) The name "linear probing" comes from the fact that accessing a cell of the bucket array can be viewed as a "probe."

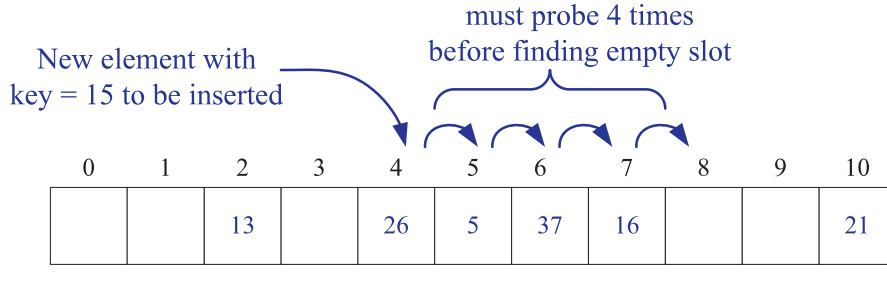


Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

To implement $\mathsf{erase}(k)$, we might, at first, think we need to do a considerable amount of shifting of entries to make it look as though the entry with key k was never inserted, which would be very complicated. A typical way to get around this difficulty is to replace a deleted entry with a special "available" marker object. With this special marker possibly occupying buckets in our hash table, we modify our search algorithm for $\mathsf{erase}(k)$ or $\mathsf{find}(k)$ so that the search for a key k skips over cells

containing the available marker and continue probing until reaching the desired entry or an empty bucket (or returning back to where we started). Additionally, our algorithm for put(k,v) should remember an available cell encountered during the search for k, since this is a valid place to put a new entry (k,v). Thus, linear probing saves space, but it complicates removals.

Even with the use of the available marker object, linear probing suffers from an additional disadvantage. It tends to cluster the entries of the map into contiguous runs, which may even overlap (particularly if more than half of the cells in the hash table are occupied). Such contiguous runs of occupied hash cells causes searches to slow down considerably.

Quadratic Probing

Another open-addressing strategy, known as ***quadratic probing***, involves iteratively trying the buckets $A[(i+f(j)) \bmod N]$, for $j = 0,1,2,\ldots$, where $f(j) = j^2$, until finding an empty bucket. As with linear probing, the quadratic-probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing. Nevertheless, it creates its own kind of clustering, called ***secondary clustering***, where the set of filled array cells "bounces" around the array in a fixed pattern. If N is not chosen as a prime, then the quadratic-probing strategy may not find an empty bucket in A even if one exists. In fact, even if N is prime, this strategy may not find an empty slot if the bucket array is at least half full. We explore the cause of this type of clustering in an exercise (Exercise C-9.9).

Double Hashing

Another open-addressing strategy that does not cause clustering of the kind produced by linear probing or by quadratic probing is the ***double-hashing*** strategy. In this approach, we choose a secondary hash function, h', and if h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the buckets $A[(i+f(j)) \bmod N]$ next, for $j = 1,2,3,\ldots$, where $f(j) = j \cdot h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, N should be a prime. Moreover, we should choose a secondary hash function that attempts to minimize clustering as much as possible.

These ***open-addressing*** schemes save some space over the separate-chaining method, but they are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, depending on the load factor of the bucket array. So, if memory space is not a major issue, the collision-handling method of choice seems to be separate chaining.

9.2.6 Load Factors and Rehashing

In all of the hash-table schemes described above, the load factor, $\lambda = n/N$, should be kept below 1. Experiments and average-case analyses suggest that we should maintain $\lambda < 0.5$ for the open-addressing schemes and we should maintain $\lambda < 0.9$ for separate chaining.

As we explore in Exercise C-9.9, some open-addressing schemes can start to fail when $\lambda \geq 0.5$. Although the details of the average-case analysis of hashing are beyond the scope of this book, its probabilistic basis is quite intuitive. If our hash function is good, then we expect the hash function values to be uniformly distributed in the range $[0, N-1]$. Thus, to store n items in our map, the expected number of keys in a bucket would be $\lceil n/N \rceil$ at most, which is $O(1)$ if n is $O(N)$.

With open addressing, as the load factor λ grows beyond 0.5 and starts approaching 1, clusters of items in the bucket array start to grow as well. These clusters cause the probing strategies to "bounce around" the bucket array for a considerable amount of time before they can finish. At the limit, when λ is close to 1, all map operations have linear expected running times, since, in this case, we expect to encounter a linear number of occupied buckets before finding one of the few remaining empty cells.

Rehashing into a New Table

Keeping the load factor below a certain threshold is vital for open-addressing schemes and is also of concern to the separate-chaining method. If the load factor of a hash table goes significantly above a specified threshold, then it is common to require that the table be resized (to regain the specified load factor) and all the objects inserted into this new resized table. Indeed, if we let our hash table become full, some implementations may crash. When rehashing to a new table, a good requirement is having the new array's size be at least double the previous size. Once we have allocated this new bucket array, we must define a new hash function to go with it (possibly computing new parameters, as in the MAD method). Given this new hash function, we then reinsert every item from the old array into the new array using this new hash function. This process is known as ***rehashing***.

Even with periodic rehashing, a hash table is an efficient means of implementing an unordered map. Indeed, if we always double the size of the table with each rehashing operation, then we can amortize the cost of rehashing all the elements in the table against the time used to insert them in the first place. The analysis of this rehashing process is similar to that used to analyze vector growth. (See Section 6.1.3.) Each rehashing generally scatters the elements throughout the new bucket array. Thus, a hash table is a practical and effective implementation for an unordered map.

9.2.7 A C++ Hash Table Implementation

In Code Fragments 9.6 through 9.13, we present a C++ implementation of the map ADT, called HashMap, which is based on hashing with separate chaining. The class is templated with the key type K, the value type V, and the hash comparator type H. The hash comparator defines a function, $hash(k)$, which maps a key into an integer index. As with less-than comparators (see Section 8.1.2), a hash comparator class does this by overriding the "()" operator.

We present the general class structure in Code Fragment 9.6. The definition begins with the public types required by the map interface, the entry type Entry, and the iterator type Iterator. This is followed by the declarations of the public member functions. We then give the private member data, which consists of the number of entries n, the hash comparator function *hash*, and the bucket array B. We have omitted two sections, which are filled in later. The first is a declaration of some utility types and functions and the second is the declaration of the map's iterator class.

```
template <typename K, typename V, typename H>
class HashMap {
public:                                          // public types
  typedef Entry<const K,V> Entry;                // a (key,value) pair
  class Iterator;                                // a iterator/position
public:                                          // public functions
  HashMap(int capacity = 100);                   // constructor
  int size() const;                              // number of entries
  bool empty() const;                            // is the map empty?
  Iterator find(const K& k);                     // find entry with key k
  Iterator put(const K& k, const V& v);          // insert/replace (k,v)
  void erase(const K& k);                        // remove entry with key k
  void erase(const Iterator& p);                 // erase entry at p
  Iterator begin();                              // iterator to first entry
  Iterator end();                                // iterator to end entry
protected:                                       // protected types
  typedef std::list<Entry> Bucket;               // a bucket of entries
  typedef std::vector<Bucket> BktArray;          // a bucket array
  // ...insert HashMap utilities here
private:
  int n;                                         // number of entries
  H hash;                                        // the hash comparator
  BktArray B;                                    // bucket array
public:                                          // public types
  // ...insert Iterator class declaration here
};
```

Code Fragment 9.6: The class HashMap, which implements the map ADT.

We have defined the key part of Entry to be "const K," rather than "K." This prevents a user from inadvertently modifying a key. The class makes use of two major data types. The first is an STL list of entries, called a Bucket, each storing a single bucket. The other is an STL vector of buckets, called BktArray.

Before describing the main elements of the class, we introduce a few local (protected) utilities in Code Fragment 9.7. We declare three helper functions, finder, inserter, and eraser, which, respectively, handle the low-level details of finding, inserting, and removing entries. For convenience, we define two iterator types, one called BItor for iterating over the buckets of the bucket array, and one called EItor, for iterating over the entries of a bucket. We also give two utility functions, nextBkt and endOfBkt, which are used to iterate through the entries of a single bucket.

```
Iterator finder(const K& k);                                  // find utility
Iterator inserter(const Iterator& p, const Entry& e);         // insert utility
void eraser(const Iterator& p);                               // remove utility
typedef typename BktArray::iterator BItor;                    // bucket iterator
typedef typename Bucket::iterator EItor;                      // entry iterator
static void nextEntry(Iterator& p)                            // bucket's next entry
  { ++p.ent; }
static bool endOfBkt(const Iterator& p)                       // end of bucket?
  { return p.ent == p.bkt->end(); }
```

Code Fragment 9.7: Declarations of utilities to be inserted into HashMap.

We present the class Iterator in Code Fragment 9.8. An iterator needs to store enough information about the position of an entry to allow it to navigate. The members *ent*, *bkt*, and *ba* store, respectively, an iterator to the current entry, the bucket containing this entry, and the bucket array containing the bucket. The first two are of types EItor and BItor, respectively, and the third is a pointer. Our implementation is minimal. In addition to a constructor, we provide operators for dereferencing ("*"), testing equality ("=="), and advancing through the map ("++").

```
class Iterator {                                              // an iterator (& position)
private:
  EItor ent;                                                  // which entry
  BItor bkt;                                                  // which bucket
  const BktArray* ba;                                         // which bucket array
public:
  Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())
    : ent(q), bkt(b), ba(&a) { }
  Entry& operator*() const;                                   // get entry
  bool operator==(const Iterator& p) const;                   // are iterators equal?
  Iterator& operator++();                                     // advance to next entry
  friend class HashMap;                                       // give HashMap access
};
```

Code Fragment 9.8: Declaration of the Iterator class for HashMap.

Iterator Dereferencing and Condensed Function Definitions

Let us now present the definitions of the class member functions for our map's Iterator class. In Code Fragment 9.9, we present an implementation of the dereferencing operator. The function body itself is very simple and involves returning a reference to the corresponding entry. However, the rules of C++ syntax demand an extraordinary number of template qualifiers. First, we need to qualify the function itself as being a member of HashMap's iterator class, which we do with the qualifier HashMap<K,V,H>::Iterator. Second, we need to qualify the function's return type as being HashMap's entry class, which we do with the qualifier HashMap<K,V,H>::Entry. On top of this, we must recall from Section 8.2.1 that, since we are using a template parameter to define a type, we need to include the keyword **typename**.

```
template <typename K, typename V, typename H> // get entry
typename HashMap<K,V,H>::Entry&
HashMap<K,V,H>::Iterator::operator*() const
  { return *ent; }
```

Code Fragment 9.9: The Iterator dereferencing operator (complete form).

In order to make our function definitions more readable, we adopt a notational convention in some of our future code fragments of specifying the scoping qualifier for the code fragment in italic blue font. We omit this qualifier from the code fragment, and we also omit the template statement and the **typename** specifications. Adding these back is a simple mechanical exercise. Although this is not valid C++ syntax, it conveys the important content in a much more succinct manner. An example of the same dereferencing operator is shown in Code Fragment 9.10.

Caution

```
/* HashMap⟨K,V,H⟩ :: */                          // get entry
  Entry& Iterator::operator*() const
    { return *ent; }
```

Code Fragment 9.10: The same dereferencing operator of Code Fragment 9.9 in condensed form.

Definitions of the Other Iterator Member Functions

Let us next consider the Iterator operator "**operator** $==(p)$," which tests whether this iterator is equal to iterator p. We first check that they belong to the same bucket array and the same bucket within this array. If not, the iterators certainly differ. Otherwise, we check whether they both refer to the end of the bucket array. (Since we have established that the buckets are equal, it suffices to test just one of them.) If so, they are both equal to HashMap::end(). If not, we check whether they both

refer to the same entry of the bucket. This is implemented in Code Fragment 9.11.

```
/* HashMap⟨K,V,H⟩ :: */                                  // are iterators equal?
  bool Iterator::operator==(const Iterator& p) const {
    if (ba != p.ba || bkt != p.bkt) return false;       // ba or bkt differ?
    else if (bkt == ba->end()) return true;             // both at the end?
    else return (ent == p.ent);                         // else use entry to decide
  }
```

Code Fragment 9.11: The Iterator operators for equality testing and increment.

Next, let us consider the Iterator increment operator, shown in Code Fragment 9.12. The objective is to advance the iterator to the next valid entry. Typically, this involves advancing to the next entry within the current bucket. But, if we fall off the end of this bucket, we must advance to the first element of the next nonempty bucket. To do this, we first advance to the next bucket entry by applying the STL increment operator on the entry iterator *ent*. We then use the utility function endOfBkt to determine whether we have arrived at the end of this bucket. If so, we search for the next nonempty bucket. To do this, we repeatedly increment *bkt* and check whether we have fallen off the end of the bucket array. If so, this is the end of the map and we are done. Otherwise, we check whether the bucket is empty. When we first find a nonempty bucket, we move *ent* to the first entry of this bucket.

```
/* HashMap⟨K,V,H⟩ :: */                                  // advance to next entry
  Iterator& Iterator::operator++() {
    ++ent;                                              // next entry in bucket
    if (endOfBkt(*this)) {                              // at end of bucket?
      ++bkt;                                            // go to next bucket
      while (bkt != ba->end() && bkt->empty())          // find nonempty bucket
        ++bkt;
      if (bkt == ba->end()) return *this;               // end of bucket array?
      ent = bkt->begin();                               // first nonempty entry
    }
    return *this;                                       // return self
  }
```

Code Fragment 9.12: The Iterator operators for equality testing and increment.

Definitions of the HashMap Member Functions

Before discussing the main functions of class HashMap, let us present the functions begin and end. These are given in Code Fragment 9.13. The function end is the simpler of the two. It involves generating an iterator whose bucket component is the end of the bucket array. We do not bother to specify a value for the entry part of the

iterator. The reason is that our iterator equality test (shown in Code Fragment 9.11) does not bother to compare the entry iterator values if the bucket iterators are at the end of the bucket array.

```
/* HashMap⟨K,V,H⟩ :: */                          // iterator to end
  Iterator end()
    { return Iterator(B, B.end()); }

/* HashMap⟨K,V,H⟩ :: */                          // iterator to front
  Iterator begin() {
    if (empty()) return end();                   // emtpty - return end
    BItor bkt = B.begin();                       // else search for an entry
    while (bkt->empty()) ++bkt;                  // find nonempty bucket
    return Iterator(B, bkt, bkt->begin());       // return first of bucket
  }
```

Code Fragment 9.13: The functions of HashMap returning iterators to the beginning and end of the map.

The function begin, shown in the bottom part of Code Fragment 9.13, is more complex, since we need to search for a nonempty bucket. We first check whether the map is empty. If so, we simply return the map's end. Otherwise, starting at the beginning of the bucket array, we search for a nonempty bucket. (We know we will succeed in finding one.) Once we find it, we return an iterator that points to the first entry of this bucket.

Now that we have presented the iterator-related functions, we are ready to present the functions for class HashMap. We begin with the constructor and simple container functions. The constructor is given the bucket array's capacity and creates a vector of this size. The member n tracks the number of entries. These are given in Code Fragment 9.14.

```
/* HashMap⟨K,V,H⟩ :: */                          // constructor
  HashMap(int capacity) : n(0), B(capacity) { }

/* HashMap⟨K,V,H⟩ :: */                          // number of entries
  int size() const { return n; }

/* HashMap⟨K,V,H⟩ :: */                          // is the map empty?
  bool empty() const { return size() == 0; }
```

Code Fragment 9.14: The constructor and standard functions for HashMap.

Next, we present the functions related to finding keys in the top part of Code Fragment 9.15. Most of the work is done by the utility function finder. It first applies the hash function associated with the given hash comparator to the key k. It converts this to an index into the bucket array by taking the hash value modulo

the array size. To obtain an iterator to the desired bucket, we add this index to the beginning iterator of the bucket array. (We are using the fact mentioned in Section 6.1.4 that STL vectors provide a random access iterator, so addition is allowed.) Let *bkt* be an iterator to this bucket. We create an iterator *p*, which is initialized to the beginning of this bucket. We then perform a search for an entry whose key matches *k* or until we fall off the end of the list. In either case, we return the final value of the iterator as the search result.

```
/* HashMap⟨K,V,H⟩ :: */                          // find utility
  Iterator finder(const K& k) {
    int i = hash(k) % B.size();                  // get hash index i
    BItor bkt = B.begin() + i;                   // the ith bucket
    Iterator p(B, bkt, bkt->begin());            // start of ith bucket
    while (!endOfBkt(p) && (*p).key() != k)      // search for k
      nextEntry(p);
    return p;                                    // return final position
  }

/* HashMap⟨K,V,H⟩ :: */                          // find key
  Iterator find(const K& k) {
    Iterator p = finder(k);                      // look for k
    if (endOfBkt(p))                             // didn't find it?
      return end();                              // return end iterator
    else
      return p;                                  // return its position
  }
```

Code Fragment 9.15: The functions of HashMap related to finding keys.

The public member function find is shown in the bottom part of Code Fragment 9.15. It invokes the finder utility. If the entry component is at the end of the bucket, we know that the key was not found, so we return the special iterator end() to the end of the map. (In this way, all unsuccessful searches produce the same result.) This is shown in Code Fragment 9.15.

The insertion utility, inserter, is shown in the top part of Code Fragment 9.16. This utility is given the desired position at which to insert the new entry. It invokes the STL list insert function to perform the insertion. It also increments the count of the number of entries in the map and returns an iterator to the inserted position.

The public insert function, put, first applies finder to determine whether any entry with this key exists in the map. We first determine whether it was not found by testing whether the iterator as fallen off the end of the bucket. If so, we insert it at the end of this bucket. Otherwise, we modify the existing value of this entry. Later, in Section 9.5.2, we present an alternative approach, which inserts a new entry, even when a duplicate key is discovered.

```
/* HashMap⟨K,V,H⟩ :: */                                  // insert utility
  Iterator inserter(const Iterator& p, const Entry& e) {
    EItor ins = p.bkt->insert(p.ent, e);                 // insert before p
    n++;                                                 // one more entry
    return Iterator(B, p.bkt, ins);                      // return this position
  }

/* HashMap⟨K,V,H⟩ :: */                                  // insert/replace (v,k)
  Iterator put(const K& k, const V& v) {
    Iterator p = finder(k);                              // search for k
    if (endOfBkt(p)) {                                   // k not found?
      return inserter(p, Entry(k, v));                   // insert at end of bucket
    }
    else {                                               // found it?
      p.ent->setValue(v);                                // replace value with v
      return p;                                          // return this position
    }
  }
```

Code Fragment 9.16: The functions of HashMap for inserting and replacing entries.

The removal functions are also quite straightforward and are given in Code Fragment 9.17. The main utility is the function eraser, which removes an entry at a given position by invoking the STL list erase function. It also decrements the number of entries. The iterator-based removal function simply invokes eraser. The key-based removal function first applies the finder utility to look up the key. If it is not found, that is, if the returned position is the end of the bucket, an exception is thrown. Otherwise, the eraser utility is invoked to remove the entry.

```
/* HashMap⟨K,V,H⟩ :: */                                  // remove utility
  void eraser(const Iterator& p) {
    p.bkt->erase(p.ent);                                 // remove entry from bucket
    n--;                                                 // one fewer entry
  }

/* HashMap⟨K,V,H⟩ :: */                                  // remove entry at p
  void erase(const Iterator& p)
    { eraser(p); }

/* HashMap⟨K,V,H⟩ :: */                                  // remove entry with key k
  void erase(const K& k) {
    Iterator p = finder(k);                              // find k
    if (endOfBkt(p))                                     // not found?
      throw NonexistentElement("Erase of nonexistent"); // ...error
    eraser(p);                                           // remove it
  }
```

Code Fragment 9.17: The functions of HashMap involved with removing entries.

9.3 Ordered Maps

In some applications, simply looking up values based on associated keys is not enough. We often also want to keep the entries in a map sorted according to some total order and be able to look up keys and values based on this ordering. That is, in an ***ordered map***, we want to perform the usual map operations, but also maintain an order relation for the keys in our map and use this order in some of the map functions. We can use a comparator to provide the order relation among keys, allowing us to define an ordered map relative to this comparator, which can be provided to the ordered map as an argument to its constructor.

When the entries of a map are stored in order, we can provide efficient implementations for additional functions in the map ADT. As with the standard map ADT, in order to indicate that an object is not present, the class provides a special sentinel iterator called end. The ordered map includes all the functions of the standard map ADT plus the following:

firstEntry(k): Return an iterator to the entry with smallest key value; if the map is empty, it returns end.

lastEntry(k): Return an iterator to the entry with largest key value; if the map is empty, it returns end.

ceilingEntry(k): Return an iterator to the entry with the least key value greater than or equal to k; if there is no such entry, it returns end.

floorEntry(k): Return an iterator to the entry with the greatest key value less than or equal to k; if there is no such entry, it returns end.

lowerEntry(k): Return an iterator to the entry with the greatest key value less than k; if there is no such entry, it returns end.

higherEntry(k): Return an iterator to the entry with the least key value greater than k; if there is no such entry, it returns end.

Implementing an Ordered Map

The ordered nature of the operations given above for the ordered map ADT makes the use of an unordered list or a hash table inappropriate, because neither of these data structures maintains any ordering information for the keys in the map. Indeed, hash tables achieve their best search speeds when their keys are distributed almost at random. Thus, we should consider an alternative implementation when dealing with ordered maps. We discuss one such implementation next, and we discuss other implementations in Section 9.4 and Chapter 10.

9.3.1 Ordered Search Tables and Binary Search

If the keys in a map come from a total order, we can store the map's entries in a vector L in increasing order of the keys. (See Figure 9.6.) We specify that L is a vector, rather than a node list, because the ordering of the keys in the vector L allows for faster searching than would be possible had L been, say, implemented with a linked list. Admittedly, a hash table has good expected running time for searching. But its worst-case time for searching is no better than a linked list, and in some applications, such as in real-time processing, we need to guarantee a worst-case searching bound. The fast algorithm for searching in an ordered vector, which we discuss in this subsection, has a good worst-case guarantee on its running time. So it might be preferred over a hash table in certain applications. We refer to this ordered vector implementation of a map as an ***ordered search table***.

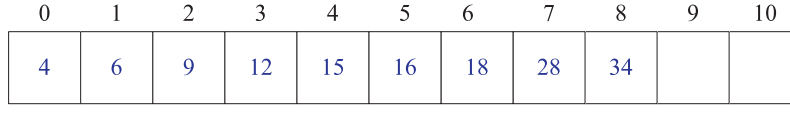


Figure 9.6: Realization of a map by means of an ordered search table. We show only the keys for this map in order to highlight their ordering.

The space requirement of an ordered search table is $O(n)$, which is similar to the list-based map implementation (Section 9.1.4), assuming we grow and shrink the array supporting the vector L to keep the size of this array proportional to the number of entries in L. Unlike an unordered list, however, performing updates in a search table takes a considerable amount of time. In particular, performing the $\mathsf{insert}(k,v)$ operation in a search table requires $O(n)$ time in the worst case, since we need to shift up all the entries in the vector with key greater than k to make room for the new entry (k,v). A similar observation applies to the operation $\mathsf{erase}(k)$, since it takes $O(n)$ time in the worst case to shift all the entries in the vector with key greater than k to close the "hole" left by the removed entry (or entries). The search table implementation is therefore inferior to the linked list implementation in terms of the worst-case running times of the map update operations. Nevertheless, we can perform the find function much faster in a search table.

Binary Search

A significant advantage of using an ordered vector L to implement a map with n entries is that accessing an element of L by its ***index*** takes $O(1)$ time. We recall, from Section 6.1, that the index of an element in a vector is the number of elements preceding it. Thus, the first element in L has index 0, and the last element has index $n-1$. In this subsection, we give a classic algorithm, ***binary search***, to locate an entry in an ordered search table. We show how this method can be used

to quickly perform the find function of the map ADT, but a similar method can be used for each of the ordered-map functions, ceilingEntry, floorEntry, lowerEntry, and higherEntry.

The elements stored in L are the entries of a map, and since L is ordered, the entry at index i has a key no smaller than the keys of the entries at indices $0,\ldots,i-1$, and no larger than the keys of the entries at indices $i+1,\ldots,n-1$. This observation allows us to quickly "home in" on a search key k using a variant of the children's game "high-low." We call an entry of our map a ***candidate*** if, at the current stage of the search, we cannot rule out that this entry has key equal to k. The algorithm maintains two parameters, low and $high$, such that all the candidate entries have index at least low and at most $high$ in L. Initially, $low = 0$ and $high = n-1$. We then compare k to the key of the median candidate e, that is, the entry e with index

$$mid = \lfloor (low + high)/2 \rfloor.$$

We consider three cases:

- If $k = e.\mathsf{key}()$, then we have found the entry we were looking for, and the search terminates successfully returning e
- If $k < e.\mathsf{key}()$, then we recur on the first half of the vector, that is, on the range of indices from low to $mid-1$
- If $k > e.\mathsf{key}()$, we recur on the range of indices from $mid+1$ to $high$

This search method is called ***binary search***, and is given in pseudo-code in Code Fragment 9.18. Operation $\mathsf{find}(k)$ on an n-entry map implemented with an ordered vector L consists of calling $\mathsf{BinarySearch}(L,k,0,n-1)$.

Algorithm $\mathsf{BinarySearch}(L,k,low,high)$:
Input: An ordered vector L storing n entries and integers low and $high$
Output: An entry of L with key equal to k and index between low and $high$, if such an entry exists, and otherwise the special sentinel end

```
if low > high then
    return end
else
    mid ← ⌊(low + high)/2⌋
    e ← L.at(mid)
    if k = e.key() then
        return e
    else if k < e.key() then
        return BinarySearch(L, k, low, mid − 1)
    else
        return BinarySearch(L, k, mid + 1, high)
```

Code Fragment 9.18: Binary search in an ordered vector.

We illustrate the binary search algorithm in Figure 9.7.

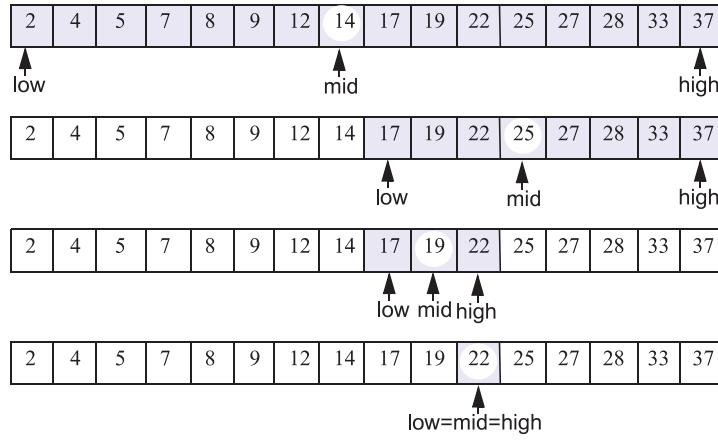


Figure 9.7: Example of a binary search to perform operation find(22), in a map with integer keys, implemented with an ordered vector. For simplicity, we show the keys, not the whole entries.

Considering the running time of binary search, we observe that a constant number of primitive operations are executed at each recursive call of function BinarySearch. Hence, the running time is proportional to the number of recursive calls performed. A crucial fact is that with each recursive call the number of candidate entries still to be searched in the vector L is given by the value

$$high - low + 1.$$

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of *mid*, the number of remaining candidates is either

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor - low \leq \frac{high - low + 1}{2}$$

or

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}.$$

Initially, the number of candidate entries is n; after the first call to BinarySearch, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the ith call to BinarySearch, the number of candidate entries remaining is at most $n/2^i$. In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer m, such that

$$n/2^m < 1.$$

In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have

$$m = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time.

Thus, we can use an ordered search table to perform fast searches in an ordered map, but using such a table for lots of map updates would take a considerable amount of time. For this reason, the primary applications for search tables are in situations where we expect few updates but many searches. Such a situation could arise, for example, in an ordered list of English words we use to order entries in an encyclopedia or help file.

Comparing Map Implementations

Note that we can use an ordered search table to implement the map ADT even if we don't want to use the additional functions of the ordered map ADT. Table 9.2 compares the running times of the functions of a (standard) map realized by either an unordered list, a hash table, or an ordered search table. Note that an unordered list allows for fast insertions but slow searches and removals, whereas a search table allows for fast searches but slow insertions and removals. Incidentally, although we don't explicitly discuss it, we note that a sorted list implemented with a doubly linked list would be slow in performing almost all the map operations. (See Exercise R-9.5.) Nevertheless, the list-like data structure we discuss in the next section can perform the functions of the ordered map ADT quite efficiently.

Method	*List*	*Hash Table*	*Search Table*
size, empty	$O(1)$	$O(1)$	$O(1)$
find	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
insert	$O(1)$	$O(1)$	$O(n)$
erase	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

Table 9.2: Comparison of the running times of the functions of a map realized by means of an unordered list, a hash table, or an ordered search table. We let n denote the number of entries in the map and we let N denote the capacity of the bucket array in the hash table implementation. The space requirement of all the implementations is $O(n)$, assuming that the arrays supporting the hash-table and search-table implementations are maintained such that their capacity is proportional to the number of entries in the map.

9.3.2 Two Applications of Ordered Maps

As we have mentioned in the preceding sections, unordered and ordered maps have many applications. In this section, we explore some specific applications of ordered maps.

Flight Databases

There are several web sites on the Internet that allow users to perform queries on flight databases to find flights between various cities, typically with the intent of buying a ticket. To make a query, a user specifies origin and destination cities, a departure date, and a departure time. To support such queries, we can model the flight database as a map, where keys are Flight objects that contain fields corresponding to these four parameters. That is, a key is a ***tuple***

$$k = (origin, destination, date, time).$$

Additional information about a flight, such as the flight number, the number of seats still available in first (F) and coach (Y) class, the flight duration, and the fare, can be stored in the value object.

Finding a requested flight is not simply a matter of finding a key in the map matching the requested query, however. The main difficulty is that, although a user typically wants to exactly match the origin and destination cities, as well as the departure date, he or she will probably be content with any departure time that is close to his or her requested departure time. We can handle such a query, of course, by ordering our keys lexicographically. Thus, given a user query key k, we could, for instance, call ceilingEntry(k) to return the flight between the desired cities on the desired date, with departure time at the desired time or after. A similar use of floorEntry(k) would give us the flight with departure time at the desired time or before. Given these entries, we could then use the higherEntry or lowerEntry functions to find flights with the next close-by departure times that are respectively higher or lower than the desired time, k. Therefore, an efficient implementation for an ordered map would be a good way to satisfy such queries. For example, calling ceilingEntry(k) on a query key $k = (\text{ORD, PVD, 05May, 09:30})$, followed by the respective calls to higherEntry, might result in the following sequence of entries:

((ORD, PVD, 05May, 09:53), (AA 1840, F5, Y15, 02:05, $251))
((ORD, PVD, 05May, 13:29), (AA 600, F2, Y0, 02:16, $713))
((ORD, PVD, 05May, 17:39), (AA 416, F3, Y9, 02:09, $365))
((ORD, PVD, 05May, 19:50), (AA 1828, F9, Y25, 02:13, $186))

Maxima Sets

Life is full of trade-offs. We often have to trade off a desired performance measure against a corresponding cost. Suppose, for the sake of an example, we are interested in maintaining a database rating automobiles by their maximum speeds and their cost. We would like to allow someone with a certain amount to spend to query our database to find the fastest car they can possibly afford.

We can model such a trade-off problem as this by using a key-value pair to model the two parameters that we are trading off, which in this case would be the pair $(cost, speed)$ for each car. Notice that some cars are strictly better than other cars using this measure. For example, a car with cost-speed pair $(20,000, 100)$ is strictly better than a car with cost-speed pair $(30,000, 90)$. At the same time, there are some cars that are not strictly dominated by another car. For example, a car with cost-speed pair $(20,000, 100)$ may be better or worse than a car with cost-speed pair $(30,000, 120)$, depending on how much money we have to spend. (See Figure 9.8.)

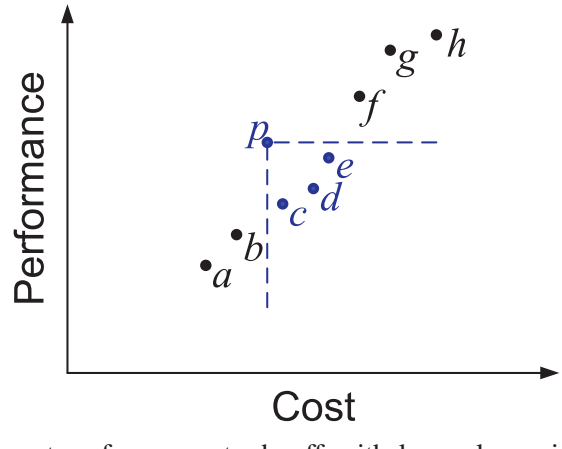


Figure 9.8: The cost-performance trade-off with key-value pairs represented by points in the plane. Notice that point p is strictly better than points c, d, and e, but may be better or worse than points a, b, f, g, and h, depending on the price we are willing to pay. Thus, if we were to add p to our set, we could remove the points c, d, and e, but not the others.

Formally, we say a price-performance pair (a,b) ***dominates*** a pair (c,d) if $a < c$ and $b > d$. A pair (a,b) is called a ***maximum*** pair if it is not dominated by any other pairs. We are interested in maintaining the set of maxima of a collection C of price-performance pairs. That is, we would like to add new pairs to this collection (for example, when a new car is introduced), and we would like to query this collection for a given dollar amount, d, to find the fastest car that costs no more than d dollars.

Maintaining a Maxima Set with an Ordered Map

We can store the set of maxima pairs in an ordered map, M, ordered by cost, so that the cost is the key field and performance (speed) is the value field. We can then implement operations $\mathsf{add}(c,p)$, which adds a new cost-performance pair (c,p), and $\mathsf{best}(c)$, which returns the best pair with cost at most c, as shown in Code Fragments 9.19 and 9.20.

Algorithm $\mathsf{best}(c)$:
Input: A cost c
Output: The cost-performance pair in M with largest cost less than or equal to c, or the special sentinel end, if there is no such pair
return $M.\mathsf{floorEntry}(c)$

Code Fragment 9.19: The best() function, used in a class maintaining a set of maxima implemented with an ordered map M.

Algorithm $\mathsf{add}(c,p)$:
Input: A cost-performance pair (c,p)
Output: None (but M will have (c,p) added to the set of cost-performance pairs)
$e \leftarrow M.\mathsf{floorEntry}(c)$ {the greatest pair with cost at most c}
if $e \neq$ end **then**
  if $e.\mathsf{value}() > p$ **then**
    return {(c,p) is dominated, so don't insert it in M}
$e \leftarrow M.\mathsf{ceilingEntry}(c)$ {next pair with cost at least c}
{Remove all the pairs that are dominated by (c,p)}
while $e \neq$ end **and** $e.\mathsf{value}() < p$ **do**
  $M.\mathsf{erase}(e.\mathsf{key}())$ {this pair is dominated by (c,p)}
  $e \leftarrow M.\mathsf{higherEntry}(e.\mathsf{key}())$ {the next pair after e}
$M.\mathsf{put}(c,p)$ {Add the pair (c,p), which is not dominated}

Code Fragment 9.20: The $\mathsf{add}(c,p)$ function used in a class for maintaining a set of maxima implemented with an ordered map M.

Unfortunately, if we implement M using any of the data structures described above, it results in a poor running time for the above algorithm. If, on the other hand, we implement M using a skip list, which we describe next, then we can perform $\mathsf{best}(c)$ queries in $O(\log n)$ expected time and $\mathsf{add}(c,p)$ updates in $O((1+r)\log n)$ expected time, where r is the number of points removed.

9.4 Skip Lists

An interesting data structure for efficiently realizing the ordered map ADT is the ***skip list***. This data structure makes random choices in arranging the entries in such a way that search and update times are $O(\log n)$ ***on average***, where n is the number of entries in the dictionary. Interestingly, the notion of average time complexity used here does not depend on the probability distribution of the keys in the input. Instead, it depends on the use of a random-number generator in the implementation of the insertions to help decide where to place the new entry. The running time is averaged over all possible outcomes of the random numbers used when inserting entries.

Because they are used extensively in computer games, cryptography, and computer simulations, functions that generate numbers that can be viewed as random numbers are built into most modern computers. Some functions, called ***pseudo-random number generators***, generate random-like numbers, starting with an initial ***seed***. Other functions use hardware devices to extract "true" random numbers from nature. In any case, we assume that our computer has access to numbers that are sufficiently random for our analysis.

The main advantage of using ***randomization*** in data structure and algorithm design is that the structures and functions that result are usually simple and efficient. We can devise a simple randomized data structure, called the skip list, which has the same logarithmic time bounds for searching as is achieved by the binary searching algorithm. Nevertheless, the bounds are ***expected*** for the skip list, while they are ***worst-case*** bounds for binary searching in a lookup table. On the other hand, skip lists are much faster than lookup tables for map updates.

A ***skip list*** S for a map M consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. Each list S_i stores a subset of the entries of M sorted by increasing keys plus entries with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in M and $+\infty$ is larger than every possible key that can be inserted in M. In addition, the lists in S satisfy the following:

- List S_0 contains every entry of the map M (plus the special entries with keys $-\infty$ and $+\infty$)
- For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the entries in list S_{i-1}
- List S_h contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 9.9. It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to h as the ***height*** of skip list S.

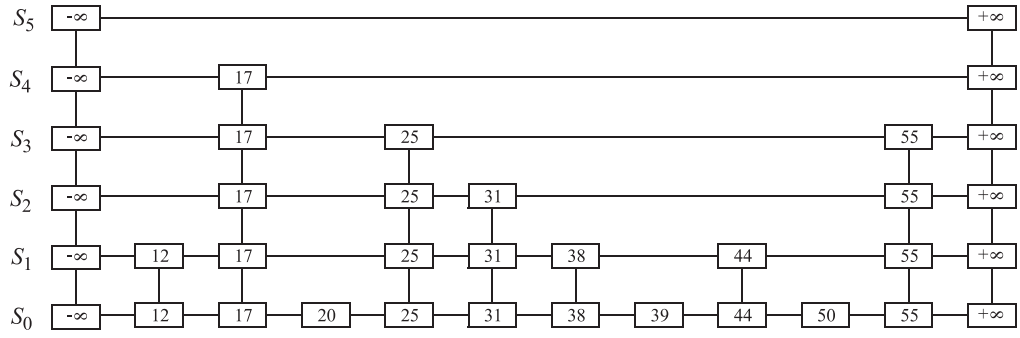


Figure 9.9: Example of a skip list storing 10 entries. For simplicity, we show only the keys of the entries.

Intuitively, the lists are set up so that S_{i+1} contains more or less every other entry in S_i. As can be seen in the details of the insertion method, the entries in S_{i+1} are chosen at random from the entries in S_i by picking each entry from S_i to also be in S_{i+1} with probability $1/2$. That is, in essence, we "flip a coin" for each entry in S_i and place that entry in S_{i+1} if the coin comes up "heads." Thus, we expect S_1 to have about $n/2$ entries, S_2 to have about $n/4$ entries, and, in general, S_i to have about $n/2^i$ entries. In other words, we expect the height h of S to be about $\log n$. The halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists, however. Instead, randomization is used.

Using the position abstraction used for lists and trees, we view a skip list as a two-dimensional collection of positions arranged horizontally into ***levels*** and vertically into ***towers***. Each level is a list S_i and each tower contains positions storing the same entry across consecutive lists. The positions in a skip list can be traversed using the following operations:

Thus ∞,-∞ acts as sentels

after(p): Return the position following p on the same level.

before(p): Return the position preceding p on the same level.

below(p): Return the position below p in the same tower.

above(p): Return the position above p in the same tower.

We conventionally assume that the above operations return a null position if the position requested does not exist. Without going into the details, we note that we can easily implement a skip list by means of a linked structure such that the above traversal functions each take $O(1)$ time, given a skip-list position p. Such a linked structure is essentially a collection of h doubly linked lists aligned at towers, which are also doubly linked lists.

9.4.1 Search and Update Operations in a Skip List

The skip list structure allows for simple map search and update algorithms. In fact, all of the skip list search and update algorithms are based on an elegant SkipSearch function that takes a key k and finds the position p of the entry e in list S_0 such that e has the largest key (which is possibly $-\infty$) less than or equal to k.

Searching in a Skip List

Suppose we are given a search key k. We begin the SkipSearch function by setting a position variable p to the top-most, left position in the skip list S, called the ***start position*** of S. That is, the start position is the position of S_h storing the special entry with key $-\infty$. We then perform the following steps (see Figure 9.10), where key(p) denotes the key of the entry at position p:

1. If S.below(p) is *null*, then the search terminates—we are ***at the bottom*** and have located the largest entry in S with key less than or equal to the search key k. Otherwise, we ***drop down*** to the next lower level in the present tower by setting $p \leftarrow S$.below(p).
2. Starting at position p, we move p forward until it is at the right-most position on the present level such that key$(p) \leq k$. We call this the ***scan forward*** step. Note that such a position always exists, since each level contains the keys $+\infty$ and $-\infty$. In fact, after we perform the scan forward for this level, p may remain where it started. In any case, we then repeat the previous step.

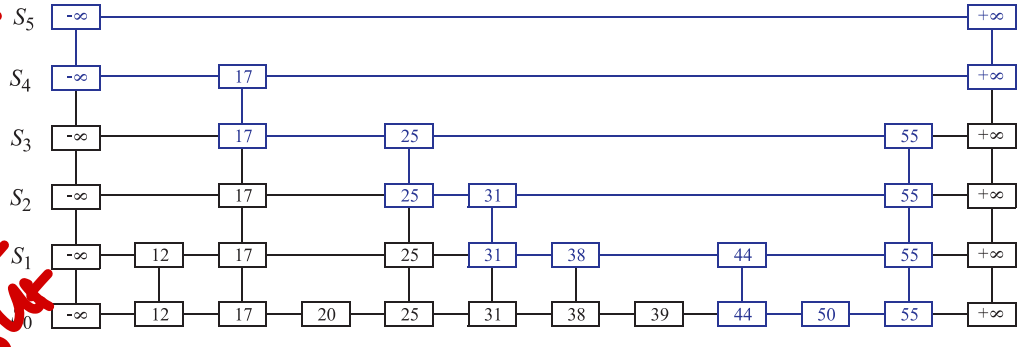


Figure 9.10: Example of a search in a skip list. The positions visited when searching for key 50 are highlighted in blue.

We give a pseudo-code description of the skip-list search algorithm, SkipSearch, in Code Fragment 9.21. Given this function, it is now easy to implement the operation find(k)—we simply perform $p \leftarrow$ SkipSearch(k) and test whether or not key$(p) = k$. If these two keys are equal, we return p; otherwise, we return *null*.

Algorithm SkipSearch(k):
Input: A search key k
Output: Position p in the bottom list S_0 such that the entry at p has the largest key less than or equal to k

```
p ← s
while below(p) ≠ null do
    p ← below(p)          {drop down}
    while k ≥ key(after(p)) do
        p ← after(p)      {scan forward}
return p.
```

Code Fragment 9.21: Search in a skip list S. Variable s holds the start position of S.

As it turns out, the expected running time of algorithm SkipSearch on a skip list with n entries is $O(\log n)$. We postpone the justification of this fact, however, until after we discuss the implementation of the update functions for skip lists.

Insertion in a Skip List

The insertion algorithm for skip lists uses randomization to decide the height of the tower for the new entry. We begin the insertion of a new entry (k,v) by performing a SkipSearch(k) operation. This gives us the position p of the bottom-level entry with the largest key less than or equal to k (note that p may hold the special entry with key $-\infty$). We then insert (k,v) immediately after position p. After inserting the new entry at the bottom level, we "flip" a coin. If the flip comes up tails, then we stop here. Else (the flip comes up heads), we backtrack to the previous (next higher) level and insert (k,v) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new entry (k,v) in lists until we finally get a flip that comes up tails. We link together all the references to the new entry (k,v) created in this process to create the tower for the new entry. A coin flip can be simulated with C++'s built-in, pseudo-random number generator by testing whether a random integer is even or odd.

We give the insertion algorithm for a skip list S in Code Fragment 9.22 and we illustrate it in Figure 9.11. The algorithm uses function insertAfterAbove($p,q,(k,v)$) that inserts a position storing the entry (k,v) after position p (on the same level as p) and above position q, returning the position r of the new entry (and setting internal references so that after, before, above, and below functions work correctly for p, q, and r). The expected running time of the insertion algorithm on a skip list with n entries is $O(\log n)$, which we show in Section 9.4.2.

Algorithm SkipInsert(k,v):
Input: Key k and value v
Output: Topmost position of the entry inserted in the skip list

```
p ← SkipSearch(k)
q ← null
e ← (k,v)
i ← −1
repeat
  i ← i + 1
  if i ≥ h then
    h ← h + 1                {add a new level to the skip list}
    t ← after(s)
    s ← insertAfterAbove(null, s, (−∞, null))
    insertAfterAbove(s, t, (+∞, null))
  while above(p) = null do
    p ← before(p)            {scan backward}
  p ← above(p)               {jump up to higher level}
  q ← insertAfterAbove(p, q, e) {add a position to the tower of the new entry}
until coinFlip() = tails
n ← n + 1
return q
```

Code Fragment 9.22: Insertion in a skip list. Method coinFlip() returns "heads" or "tails," each with probability $1/2$. Variables n, h, and s hold the number of entries, the height, and the start node of the skip list.

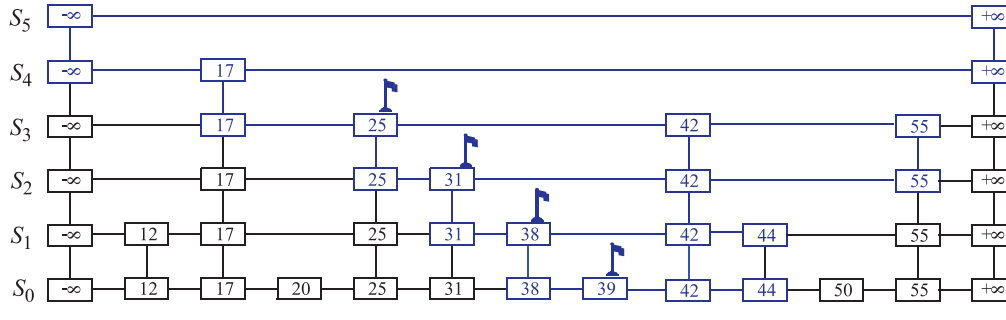


Figure 9.11: Insertion of an entry with key 42 into the skip list of Figure 9.9. We assume that the random "coin flips" for the new entry came up heads three times in a row, followed by tails. The positions visited are highlighted in blue. The positions inserted to hold the new entry are drawn with thick lines, and the positions preceding them are flagged.

Removal in a Skip List

Like the search and insertion algorithms, the removal algorithm for a skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to perform an erase(k) operation, we begin by executing function SkipSearch(k). If the position p stores an entry with key different from k, we return *null*. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in S starting at position p. The removal algorithm is illustrated in Figure 9.12 and a detailed description of it is left as an exercise (Exercise R-9.17). As we show in the next subsection, operation erase in a skip list with n entries has $O(\log n)$ expected running time.

Before we give this analysis, however, there are some minor improvements to the skip list data structure we would like to discuss. First, we don't actually need to store references to entries at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. Second, we don't actually need the above function. In fact, we don't need the before function either. We can perform entry insertion and removal in strictly a top-down, scan-forward fashion, thus saving space for "up" and "prev" references. We explore the details of this optimization in Exercise C-9.10. Neither of these optimizations improve the asymptotic performance of skip lists by more than a constant factor, but these improvements can, nevertheless, be meaningful in practice. In fact, experimental evidence suggests that optimized skip lists are faster in practice than AVL trees and other balanced search trees, which are discussed in Chapter 10.

The expected running time of the removal algorithm is $O(\log n)$, which we show in Section 9.4.2.

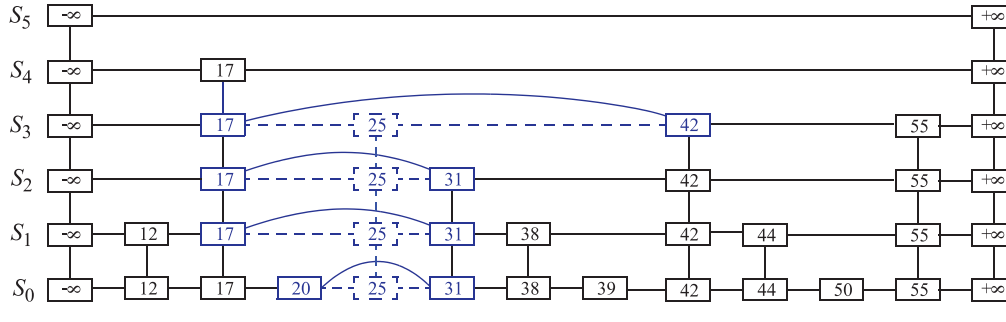


Figure 9.12: Removal of the entry with key 25 from the skip list of Figure 9.11. The positions visited after the search for the position of S_0 holding the entry are highlighted in blue. The positions removed are drawn with dashed lines.

Maintaining the Top-most Level

A skip list S must maintain a reference to the start position (the top-most, left position in S) as a member variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible courses of action we can take, both of which have their merits.

One possibility is to restrict the top level, h, to be kept at some fixed value that is a function of n, the number of entries currently in the map (from the analysis we see that $h = \max\{10, 2\lceil \log n \rceil\}$ is a reasonable choice, and picking $h = 3\lceil \log n \rceil$ is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new position once we reach the top-most level (unless $\lceil \log n \rceil < \lceil \log(n+1) \rceil$, in which case we can now go at least one more level, since the bound on the height is increasing).

The other possibility is to let an insertion continue inserting a new position as long as heads keeps getting returned from the random number generator. This is the approach taken in Algorithm SkipInsert of Code Fragment 9.22. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than $O(\log n)$ is very low, so this design choice should also work.

Either choice still results in the expected $O(\log n)$ time to perform search, insertion, and removal, however, which we show in the next section.

9.4.2 A Probabilistic Analysis of Skip Lists ⋆

As we have shown above, skip lists provide a simple implementation of an ordered map. In terms of worst-case performance, however, skip lists are not a superior data structure. In fact, if we don't officially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an infinite loop (it is not actually an infinite loop, however, since the probability of having a fair coin repeatedly come up heads forever is 0). Moreover, we cannot infinitely add positions to a list without eventually running out of memory. In any case, if we terminate position insertion at the highest level h, then the ***worst-case*** running time for performing the find, insert, and erase operations in a skip list S with n entries and height h is $O(n+h)$. This worst-case performance occurs when the tower of every entry reaches level $h-1$, where h is the height of S. However, this event has very low probability. Judging from this worst case, we might conclude that the skip list structure is strictly inferior to the other map implementations discussed earlier in this chapter. But this would not be a fair analysis because this worst-case behavior is a gross overestimate.

Bounding the Height of a Skip List

Because the insertion step involves randomization, a more accurate analysis of skip lists involves a bit of probability. At first, this might seem like a major undertaking, since a complete and thorough probabilistic analysis could require deep mathematics (and, indeed, there are several such deep analyses that have appeared in the research literature related to data structures). Fortunately, such an analysis is not necessary to understand the expected asymptotic behavior of skip lists. The informal and intuitive probabilistic analysis we give below uses only basic concepts of probability theory.

Let us begin by determining the expected value of the height h of a skip list S with n entries (assuming that we do not terminate insertions early). The probability that a given entry has a tower of height $i \geq 1$ is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is $1/2^i$. Hence, the probability P_i that level i has at least one position is at most

$$P_i \leq \frac{n}{2^i},$$

because the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one position, that is, it is no more than P_i. This means that h is larger than, say, $3\log n$ with probability at most

$$\begin{aligned} P_{3\log n} &\leq \frac{n}{2^{3\log n}} \\ &= \frac{n}{n^3} = \frac{1}{n^2}. \end{aligned}$$

For example, if $n = 1000$, this probability is a one-in-a-million long shot. More generally, given a constant $c > 1$, h is larger than $c\log n$ with probability at most $1/n^{c-1}$. That is, the probability that h is smaller than $c\log n$ is at least $1 - 1/n^{c-1}$. Thus, with high probability, the height h of S is $O(\log n)$.

Analyzing Search Time in a Skip List

Next, consider the running time of a search in skip list S, and recall that such a search involves two nested **while** loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is $O(\log n)$ with high probability, the number of drop-down steps is $O(\log n)$ with high probability.

So we have yet to bound the number of scan-forward steps we make. Let n_i be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level $i+1$. If any of these keys were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in n_i is $1/2$. Therefore, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. This expected value is 2.

Hence, the expected amount of time spent scanning forward at any level i is $O(1)$. Since S has $O(\log n)$ levels with high probability, a search in S takes expected time $O(\log n)$. By a similar analysis, we can show that the expected running time of an insertion or a removal is $O(\log n)$.

Space Usage in a Skip List

Finally, let us turn to the space requirement of a skip list S with n entries. As we observed above, the expected number of positions at level i is $n/2^i$, which means that the expected total number of positions in S is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i}.$$

Using Proposition 4.5 on geometric summations, we have

$$\sum_{i=0}^{h} \frac{1}{2^i} = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 \cdot \left(1 - \frac{1}{2^{h+1}}\right) < 2 \quad \text{for all } h \geq 0.$$

Hence, the expected space requirement of S is $O(n)$.

Table 9.3 summarizes the performance of an ordered map realized by a skip list.

Operation	*Time*
size, empty	$O(1)$
firstEntry, lastEntry	$O(1)$
find, insert, erase	$O(\log n)$ (expected)
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$ (expected)

Table 9.3: Performance of an ordered map implemented with a skip list. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is $O(n)$.

9.5 Dictionaries

Like a map, a dictionary stores key-value pairs (k, v), which we call ***entries***, where k is the key and v is the value. Similarly, a dictionary allows for keys and values to be of any object type. But, whereas a map insists that entries have unique keys, a dictionary allows for multiple entries to have the same key, much like an English dictionary, which allows for multiple definitions for the same word.

The ability to store multiple entries with the same key has several applications. For example, we might want to store records for computer science authors indexed by their first and last names. Since there are a few cases of different authors with the same first and last name, there will naturally be some instances where we have to deal with different entries having equal keys. Likewise, a multi-user computer game involving players visiting various rooms in a large castle might need a mapping from rooms to players. It is natural in this application to allow users to be in the same room simultaneously, however, to engage in battles. Thus, this game would naturally be another application where it would be useful to allow for multiple entries with equal keys.

9.5.1 The Dictionary ADT

The ***dictionary*** ADT is quite similar to the map ADT, which was presented in Section 9.1. The principal differences involve the issue of multiple values sharing a common key. As with the map ADT, we assume that there is an object, called Iterator, that provides a way to reference entries of the dictionary. There is a special sentinel value, end, which is used to indicate a nonexistent entry. The iterator may be incremented from entry to entry, making it possible to enumerate entries from the collection.

As an ADT, a (unordered) dictionary D supports the following functions:

size(): Return the number of entries in D.

empty(): Return true if D is empty and false otherwise.

find(k): If D contains an entry with key equal to k, then return an iterator p referring any such entry, else return the special iterator end.

findAll(k): Return a pair of iterators (b, e), such that all the entries with key value k lie in the range from b up to, but not including, e.

insert(k, v): Insert an entry with key k and value v into D, returning an iterator referring to the newly created entry.

erase(k): Remove from D an arbitrary entry with key equal to k; an error condition occurs if D has no such entry.

erase(p): Remove from D the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

begin(): Return an iterator to the first entry of D.

end(): Return an iterator to a position just beyond the end of D.

Note that operation find(k) returns an ***arbitrary*** entry, whose key is equal to k, and erase(k) removes an arbitrary entry with key value k. In order to remove a specific entry among those having the same key, it would be necessary to remember the iterator value p returned by insert(k,v), and then use the operation erase(p).

Example 9.2: *In the following, we show a series of operations on an initially empty dictionary storing entries with integer keys and character values. In the column "Output," we use the notation $p_i : [(k,v)]$ to mean that the operation returns an iterator denoted by p_i that refers to the entry (k,v).*

Although the entries are not necessarily stored in any particular order, in order to implement the operation findAll, *we assume that items with the same keys are stored contiguously. (Alternatively, the operation* findAll *would need to return a smarter form of iterator that returns keys of equal value.)*

Operation	*Output*	*Dictionary*
empty()	**true**	$\emptyset$
insert(5,A)	$p_1 : [(5,A)]$	$\{(5,A)\}$
insert(7,B)	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
insert(2,C)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
insert(8,D)	$p_4 : [(8,D)]$	$\{(5,A),(7,B),(2,C),(8,D)\}$
insert(2,E)	$p_5 : [(2,E)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(4)	end	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(2)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
findAll(2)	(p_3,p_4)	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
size()	5	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
erase(5)	–	$\{(7,B),(2,C),(2,E),(8,D)\}$
erase(p_3)	–	$\{(7,B),(2,E),(8,D)\}$
find(2)	$p_5 : [(2,E)]$	$\{(7,B),(2,E),(8,D)\}$

The operation findAll(2) *returns the iterator pair (p_3,p_4), referring to the entries $(2,C)$ and $(8,D)$. Assuming that the entries are stored in the order listed above, iterating from p_3 up to, but not including, p_4, would enumerate the entries $\{(2,C),(2,E)\}$.*

9.5.2 A C++ Dictionary Implementation

In this Section, we describe a C++ implementation of the dictionary ADT. Our implementation, called HashDict, is a subclass of the HashMap class, from Section 9.2.7. The map ADT already includes most of the functions of the dictionary ADT. Our HashDict class implements the new function insert, which inserts a key-value pair, and the function findAll, which generates an iterator range for all the values equal to a given key. All the other functions are inherited from HashMap.

In order to support the return type of findAll, we define a nested class called Range. It is presented in Code Fragment 9.23. This simple class stores a pair of objects of type Iterator, a constructor, and two member functions for accessing each of them. This definition will be nested inside the public portion of the HashMap class definition.

```
class Range {                                   // an iterator range
private:
  Iterator _begin;                              // front of range
  Iterator _end;                                // end of range
public:
  Range(const Iterator& b, const Iterator& e)   // constructor
    : _begin(b), _end(e) { }
  Iterator& begin() { return _begin; }          // get beginning
  Iterator& end() { return _end; }              // get end
};
```

Code Fragment 9.23: Definition of the Range class to be added to HashMap.

The HashDict class definition is presented in Code Fragment 9.24. As indicated in the first line of the declaration, this is a subclass of HashMap. The class begins with type definitions for the Iterator and Entry types from the base class. This is followed by the code for class Range from Code Fragment 9.23, and the public function declarations.

```
template <typename K, typename V, typename H>
class HashDict : public HashMap<K,V,H> {
public:                                         // public functions
  typedef typename HashMap<K,V,H>::Iterator Iterator;
  typedef typename HashMap<K,V,H>::Entry Entry;
  // ...insert Range class declaration here
public:                                         // public functions
  HashDict(int capacity = 100);                 // constructor
  Range findAll(const K& k);                    // find all entries with k
  Iterator insert(const K& k, const V& v);      // insert pair (k,v)
};
```

Code Fragment 9.24: The class HashDict, which implements the dictionary ADT.

Observe that, when referring to the parent class, HashMap, we need to specify its template parameters. To avoid the need for continually repeating these parameters, we have provided type definitions for the iterator and entry classes. Because most of the dictionary ADT functions are already provided by HashMap, we need only provide a constructor and the missing dictionary ADT functions.

The constructor definition is presented in Code Fragment 9.25. It simply invokes the constructor for the base class. Note that we employ the condensed function notation that we introduced in Section 9.2.7.

```
/* HashDict⟨K,V,H⟩ :: */                                    // constructor
  HashDict(int capacity) : HashMap<K,V,H>(capacity) { }
```

Code Fragment 9.25: The class HashDict constructor.

In Code Fragment 9.26, we present an implementation of the function insert. It first locates the key by invoking the finder utility (see Code Fragment 9.15). Recall that this utility returns an iterator to an entry containing this key, if found, and otherwise it returns an iterator to the end of the bucket. In either case, we insert the new entry immediately prior to this location by invoking the inserter utility. (See Code Fragment 9.16.) An iterator referencing the resulting location is returned.

```
/* HashDict⟨K,V,H⟩ :: */                                    // insert pair (k,v)
  Iterator insert(const K& k, const V& v) {
    Iterator p = finder(k);                                 // find key
    Iterator q = inserter(p, Entry(k, v));                  // insert it here
    return q;                                               // return its position
  }
```

Code Fragment 9.26: An implementation of the dictionary function insert.

We exploit a property of how insert works. Whenever a new entry (k,v) is inserted, if the structure already contains another entry (k,v') with the same key, the finder utility function returns an iterator to the first such occurrence. The inserter utility then inserts the new entry just prior to this. It follows that all the entries having the same key are stored in a sequence of ***contiguous*** positions, all within the same bucket. (In fact, they appear in the reverse of their insertion order.) This means that, in order to produce an iterator range (b,e) for the call findAll(k), it suffices to set b to the first entry of this sequence and set e to the entry immediately following the last one.

Our implementation of findAll is given in Code Fragment 9.27. We first invoke the finder function to locate the key. If the finder returns a position at the end of some bucket, we know that the key is not present, and we return the empty iterator $(\mathsf{end},\mathsf{end})$. Otherwise, recall from Code Fragment 9.15 that finder returns the first entry with the given key value. We store this in the entry iterator b. We then traverse

the bucket until either coming to the bucket's end or encountering an entry with a key of different value. Let p be this iterator value. We return the iterator range (b, p).

```
/* HashDict⟨K,V,H⟩ :: */                                    // find all entries with k
  Range findAll(const K& k) {
    Iterator b = finder(k);                                  // look up k
    Iterator p = b;
    while (!endOfBkt(p) && (*p).key() == (*b).key()) { // find next unequal key
      ++p;
    }
    return Range(b, p);                                      // return range of positions
  }
```

Code Fragment 9.27: An implementation of the dictionary function findAll.

9.5.3 Implementations with Location-Aware Entries

As with the map ADT, there are several possible ways we can implement the dictionary ADT, including an unordered list, a hash table, an ordered search table, or a skip list. As we did for adaptable priority queues (Section 8.4.2), we can also use location-aware entries to speed up the running time for some operations in a dictionary. In removing a location-aware entry e, for instance, we could simply go directly to the place in our data structure where we are storing e and remove it. We could implement a location-aware entry, for example, by augmenting our entry class with a private *location* variable and protected functions location() and setLocation(p), which return and set this variable respectively. We would then require that the *location* variable for an entry e would always refer to e's position or index in the data structure. We would, of course, have to update this variable any time we moved an entry, as follows.

- ***Unordered list***: In an unordered list, L, implementing a dictionary, we can maintain the *location* variable of each entry e to point to e's position in the underlying linked list for L. This choice allows us to perform erase(e) as L.erase(e.location()), which would run in $O(1)$ time.
- ***Hash table with separate chaining***: Consider a hash table, with bucket array A and hash function h, that uses separate chaining for handling collisions. We use the *location* variable of each entry e to point to e's position in the list L implementing the list $A[h(k)]$. This choice allows us to perform an erase(e) as L.erase(e.location()), which would run in constant expected time.
- ***Ordered search table***: In an ordered table, T, implementing a dictionary, we should maintain the *location* variable of each entry e to be e's index in T. This choice would allow us to perform erase(e) as T.erase(e.location()).

(Recall that location() now returns an integer.) This approach would run fast if entry e was stored near the end of T.

- ***Skip list***: In a skip list, S, implementing a dictionary, we should maintain the *location* variable of each entry e to point to e's position in the bottom level of S. This choice would allow us to skip the search step in our algorithm for performing erase(e) in a skip list.

We summarize the performance of entry removal in a dictionary with location-aware entries in Table 9.4.

List	*Hash Table*	*Search Table*	*Skip List*
$O(1)$	$O(1)$ (expected)	$O(n)$	$O(\log n)$ (expected)

Table 9.4: Performance of the erase function in dictionaries implemented with location-aware entries. We use n to denote the number of entries in the dictionary.