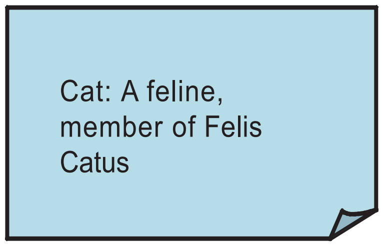
Chapter 12: Dictionary (or Map) ADT and Hash Tables

In the containers we have examined up to now, the emphasis has been on the values themselves. A bag, for example, is used to hold a collection of elements. You can add a new value to a bag, test to see whether or not a value is found in the bag, and remove a value from the bag. In a *dictionary*, on the other hand, we separate the data into two parts. Each item stored in a dictionary is represented by a key/value pair. The key is used to access the item. With the key you can access the value, which typically has more information.



The name itself suggests the metaphor used to understand this abstract data type. Think of a dictionary of the English language. Here a word (the key) is matched with a definition (the value). You use the key to find the definition. But, the connection is one way. You typically cannot search a dictionary using a definition to find the matching word.

A keyed container, such as a dictionary, can be contrasted with an indexed container, such as an array. Index values are a form of key; however, they are restricted to being integers and must be drawn from a limited range: usually zero to one less than the number of elements. Because of this restriction the elements in an array can be stored in a contiguous block; and the index can be used as part of the array offset calculation. In an array, the index need not be stored explicitly in the container. In a dictionary, on the other hand, a key can be any type of object. For this reason, the container must maintain both the key and its associated value.

The Dictionary ADT

The abstract data type that corresponds to the dictionary metaphor is known by several names. Other terms for keyed containers include the names *map*, *table*, *search table*, *associative array*, or *hash*. Whatever it is called, the idea is a data structure optimized for a very specific type of search. Elements are placed into the dictionary in key/value pairs. To do a retrieval, the user supplies a key, and the container returns the associated value. Each key identifies one entry; that is, each key is unique. However, nothing prevents two different keys from referencing the same value. The contains test is in the dictionary replaced by a test to see if a given key is legal. Finally, data is removed from a dictionary by specifying the key for the data value to be deleted.

As an ADT, the dictionary is represented by the following operations:

get(key)	Retrieve the value associated with the given key.
put(key, value)	Place the key and value association into the dictionary
containsKey(key)	Return true if key is found in dictionary

removeKey(key)	Remove key from association
keys()	Return iterator for keys in dictionary
size()	Return number of elements in dictionary

The operation we are calling put is sometimes named set, insertAt, or atPut. The get operation is sometimes termed at. In our containers a get with an invalid key will produce an assertion error. In some variations on the container this operation will raise an exception, or return a special value, such as null. We include an iterator for the key set as part of the specification, but will leave the implementation of this feature as an exercise for the reader.

The following illustrates some of the implementations of the dictionary abstraction found in various programming languages.

Operation	C++ Map<keytype, valuetype>	Java HashMap<keytype, valuetype>	C# hashtable
get	Map[key]	Get(key)	Hash[key]
put	Insert(key, value)	Put(key, value)	Add(key, value)
containsKey	Count(key)	containsKey(key)	
removeKey	Erase(key)	Remove(key)	

Applications of the Dictionary data type

Maps or dictionaries are useful in any type of application where further information is associated with a given key. Obvious examples include dictionaries of word/definition pairs, telephone books of name/number pairs, or calendars of date/event-description pairs. Dictionaries are used frequently in analysis of printed text. For example, a concordance examines each word in a text, and constructs a list indicating on which line each word appears.

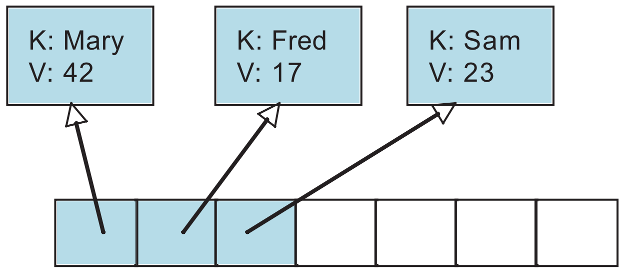
Implementations of the Dictionary ADT

We will describe several different approaches to implementing the dictionary data type. The first has the advantage of being easy to describe, however it is relatively slow. The remaining implementation techniques will introduce a new way of organizing a collection of data values. This technique is termed hashing, and the container built using this technique is called a hash table.

The concept of hashing that we describe here is useful in implementing both Bag and Dictionary type collections. Much of our presentation will use the Bag as an example, as dealing with one value is easier than dealing with two. However, the generalization to a Dictionary rather than a Bag is straightforward, and will be handled in programming projects described at the end of the chapter.

A Dictionary Built on top of a Bag

The basic idea of the first implementation approach is to treat a dictionary as simply a bag of key/value pairs. We will introduce a new name for this pair, calling it an *Association*. An Association is in some ways similar to a link. Instances of the class Association maintain a key and value field.



But we introduce a subtle twist to the definition of the type Association. When one association is compared to another, the result of the comparison is determined solely by the key. If two keys are equal, then the associations are considered equal. If one key is less than another, then the first association is considered to be smaller than the second.

With the assistance of an association, the implementation of the dictionary data type is straightforward. To see if the dictionary contains an element, a new association is created. The underlying bag is then searched. If the search returns true, it means that there is an entry in the bag that matches the key for the collection. Similarly, the remove method first constructs a new association. By invoking the remove method for the underlying bag, any element that matches the new association will be deleted. But we have purposely defined the association so that it tests only the key. Therefore, any element that matches the key will be removed.

This type of Map can be built on top of any of our Bag abstractions, and is explored in worksheet 36.

Hashing Background

We have seen how containers such as the skip list and the AVL tree can reduce the time to perform operations from O(n) to O(log n). But can we do better? Would it be possible to create a container in which the average time to perform an operation was O(1)? The answer is both yes and no.

To illustrate how, consider the following story. Six friends; Alfred, Alessia, Amina, Amy, Andy and Anne, have a club. Amy is in charge of writing a program to do bookkeeping. Dues are paid each time a member attends a meeting, but not all members attend all meetings. To help with the programming Amy uses a six-element array to store the amount each member has paid in dues.

Alfred	\$2.65	F = 5 % 6 = 5
Alessia	\$6.75	E = 4 % 6 = 4
Amina	\$5.50	I = 8 % 6 = 2
Amy	\$10.50	Y = 24 % 6 = 0
Andy	\$2.25	D = 3 % 6 = 3
Anne	\$0.75	N = 13 % 6 = 1

Amy uses an interesting fact. If she selects the third letter of each name, treating the letter as a number from 0 to 25, and then mods (%) the number by 6, each name yields a different number. So in O(1) time Amy can change a name into an integer index value, then use this value to index into a table. This is faster than an ordered data structure, indeed almost as fast as a subscript calculation.

What Amy has discovered is called a *perfect hash function*. A *hash function* is a function that takes as input an element and returns an integer value. Almost always the index used by a hash algorithm is the remainder after dividing this value by the hash table size. So, for example, Amy's hash function returns values from 0 to 25. She mods (%) by the table size (6) in order to get an index.

The idea of *hashing* can be used to create a variety of different data structures. Of course, Amy's system falls apart when the set of names is different. Suppose Alan wishes to join the club. Amy's calculation for Alan will yield 0, the same value as Amy. Two values that have the same hash are said to have *collided*. The way in which collisions are handed is what separates different hash table techniques.

Almost any process that converts a value into an integer can be used as a hash function. Strings can interpret characters as integers (as in Amy's club), doubles can use a portion of their numeric value, structures can use one or more fields. Hash functions are only required to return a value that is integer, not necessarily positive. So it is common to surround the calculation with abs( ) to ensure a positive value.

Open Address Hashing

There are several ways we can use the idea of hashing to help construct a container abstraction. The first technique you will explore is termed *open-address hashing*. (Curiously, also sometimes called *closed hashing*). We explain this technique by first constructing a Bag, and then using the Bag as the source for a Dictionary, as described in the first section. When open-address hashing is used all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different. The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So to find a location to store the value Anne we *probe* for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6.

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred	Anne	Aspen

No suppose Agnes wishes to join the club. Her hash value, 6, is already filled. The probe moves forward to the next position, and when the end of the array is reached it continues with the first element, in a fashion similar to the dynamic array deque you examined in Chapter 7. Eventually the probing halts, finding position 1:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes		Andy	Alessia	Alfred	Anne	Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes	Alan	Andy	Alessia	Alfred	Anne	Aspen

We now have as many elements as can fit into this table. The ratio of the number of elements to the table size is known as the *load factor*, written λ. For open address hashing the load factor is never larger than 1. Just as a dynamic array was doubled in size when necessary, a common solution to a full hash table is to move all values into a new and larger table when the load factor becomes larger than some threshold, such as 0.75. To do so a new table is created, and every entry in the old table is rehashed, this time dividing by the new table size to find the index to place into the new table.

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead of immediately halting, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered.

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value in the table given above, and then once more performed a search for Alan. What would happen?

One solution to this problem is to not allow removals. This is the technique we will use. The second solution is to create a special type of marker termed a *tombstone*. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

λ	(1/(1-λ))
0.25	1.3
0.5	2.0
0.6	2.5
0.75	4.0
0.85	6.6
0.95	19.0

How fast are hash table operations? The analysis depends upon several factors. We assume that the time it takes to compute the hash value itself is constant. But what about distribution of the integers returned by the hash function? It would be perfectly legal for a hash function to always return the value zero – legal, but not very useful.

The best case occurs when the hash function returns values that are uniformly distributed among all possible index values; that is, for any input value each index is equally likely. In this situation one can show that the number of elements that will be examined in performing an addition, removal or test will be roughly $1/(1 - \lambda)$. For a small load factor this is acceptable, but degrades quickly as the load factor increases. This is why hash tables typically increase the size of the table if the load factor becomes too large.

Worksheet 37 explores the implementation of a bag using open hash table techniques.

Caching

Indexing into a hash table is extremely fast, even faster than searching a skip list or an AVL tree. There are many different ways to exploit this speed. A *cache* is a data structure that uses two levels of storage. One level is simply an ordinary collection class, such as a bag dictionary. The second level is a hash table, used for its speed. The cache makes no attempt to handle the problem of collisions within the hash table. When a search request is received, the cache will examine the hash table. If the value is found in the cache, it is simply returned. If it is not found, then the original data structure is examined. If it is found there, the retrieved item *replaces* the value stored in the cache. Because the new item is now in the cache, a subsequent search for the same value will be very fast.

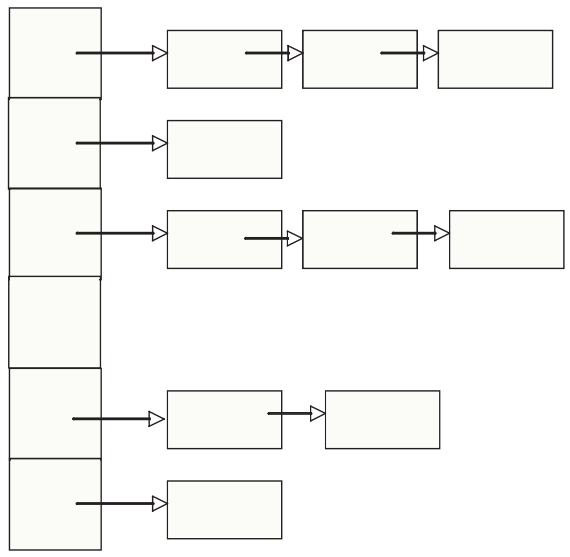
The concept of a cache is generally associated with computer memory, where the underlying container represents paged memory from a relatively slow device (such as a disk), and the cache holds memory pages that have been recently accessed. However the concept of a two-level memory system is applicable in any situation where searching is a more common operation than addition or removal, and where a search for one value means that it is very likely the value will again be requested in the near future. For example, a cache is used in the interpreter for the language Smalltalk to match a message (function applied to an object) with a method (code to be executed). The cache stores the name and class for the most recent message. If a message is sent to the same class of object, the code is found very quickly. If not, then a much slower search of the class hierarchy is performed. However, after this slower search the code is placed into the cache. A subsequent execution of the same message (which, it turns out, is very likely) will then be much faster.

Like the self-organizing linked list (Chapter 8) and the skew heap (worksheet 35), a cache is a self-organizing data structure. That is, the cache tries to improve future performance based on current behavior.

Hash Table with Buckets

In earlier sections you learned about the concept of hashing, and how it was used in an open address hash table. An entirely different approach to dealing with collisions is the idea of hash tables using buckets.

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a header for a linked list. All elements that hash into the same location will be stored in the list. For the Bag type abstraction the link stores only a value and a pointer to the next link. For a dictionary type abstraction, such as we will construct, the link stores the key, the value associated with the key, and the pointer to the next link.



Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken after dividing by the table size. This yields a table index. Next, linked list indicated by the table index is examined. The algorithms for the latter are very similar to those used in the linked list.

As with open address hash tables, the load factor (λ) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the contains test and removal is proportional to the length of the list, they are O(λ). Therefore the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address hash table) if the load factor becomes larger than 10.

Hash tables with buckets are explored in worksheet 39.

Bit Set Spell Checker

In Chapter 8 you learned about the bit set abstraction. Also in that chapter you read how a set can be used as part of a spell checker. A completely different technique for creating a spelling checker is based on using a BitSet in the fashion of a hash table. Begin with a BitSet large enough to hold t elements. We first take the dictionary of correctly spelled words, hash each word to yield a value h, then set the position h % t. After processing all the words we will have a large hash table of zero/one values. Now to test any particular word we perform the same process; hash the word, take the remainder after dividing by the table size, and test the entry. If the value in the bit set is zero, then the word is misspelled. Unfortunately, if the value in the table is 1, it may still be misspelled since two words can easily hash into the same location.

To correct this problem, we can enlarge our bit set, and try using more than one hash function. An easy way to do this is to select a table size t, and select some number of prime numbers, for example five, that are larger than t. Call these p_1, p_2, p_3, p_4 and p_5. Now rather than just using the hash value of the word as our starting point, we first take the remainder of the hash value when divided by each of the prime numbers, yielding five different values. Then we map each of these into the table. Thus each word may set five different bits. This following illustrates this with a table of size 21, using the values 113, 181, 211, 229 and 283 as our prime numbers:

Word	Hashcode	H%113	H%181	H%211	H%229	H%283
This	6022124	15(15)	73(10)	184(16)	111(6)	167(20)
Text	6018573	80(17)	142(16)	9(9)	224(14)	12(12)
Example	1359142016	51(9)	165(18)	75(12)	223(13)	273(0)

1 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 1

The key assumption is that different words may map into the same locations for one or two positions, but not for all five. Thus, we reduce the chances that a misspelled word will score a false positive hit.

Analysis shows that this is a very reliable technique as long as the final bit vector has roughly the same number of zeros as ones. Performance can be improved by using more prime numbers.

Chapter Summary

Key concepts

- Map
- Association
- Hashing
- Hash functions
- collisions
- Hash Tables
- Open address hashing
- Hash tables with Buckets

Whereas bags, stacks, queues and the other data abstractions examined up to this point emphasize the collection of individual values, a dictionary is a data abstraction designed to maintain *pairs* consisting of a key and a value. Entries are placed into the dictionary using both key and value. To recover or delete a value, the user provides only the key.

In this chapter we have explored several implementation techniques that can be used with the dictionary abstraction. Most importantly, we have introduced the idea of a hash table.

To *hash* a value means simply to apply a function that transforms a possibly non-integer key into an integer value. This simple idea is the basis for a very powerful data structuring technique. If it is possible to discover a function that transforms a set of keys via a one-to-one mapping on to a set of integer index values, then it is possible to construct a vector using non-integer keys. More commonly, several key values will map into the same integer index. Two keys that map into the same value are said to have *collided*.

The problem of collisions can be handled in a number of different ways. We have in this chapter explored three possibilities. Using open-address-hashing we probe, or search, for the next free unused location in a table. Using caching, the hash table holds the most recent value requested for the given index. Collisions are stored in a slower dictionary, which is searched only when necessary. Using hash tables with buckets, collisions are stored in a linked list of associations.

The state of a hash table can be described in part by the load factor, which is the number of elements in a table divided by the size of the hash table. For open address hashing, the load factor will always be less than 1, as there can be no more elements than the table size. For hash tables that use buckets, the load factor can be larger than 1, and can be interpreted as the average number of elements in each bucket.

The process of hashing permits access and testing operations that potentially are the fastest of any data structure we have considered. Unfortunately, this potential depends upon the wise choice of a hash function, and luck with the key set. A good hash function must uniformly distribute key values over each of the different buckets. Discovering a good hash function is often the most difficult part of using the hash table technique.

Self-Study Questions

1. In what ways is a dictionary similar to an array? In what ways are they different?

2. What does it mean to hash a value?

3. What is a hash function?

4. What is a perfect hash function?

5. What is a collision of two values?

6. What does it mean to probe for a free location in an open address hash table?

7. What is the load factor for a hash table?

8. Why do you not want the load factor to become too large?

9. In searching for a good hash function over the set of integer elements, one student thought he could use the following:

```
int hash = (int)Math.sin(value);
```

explain why this was a poor choice.

Short Exercises

1. In the dynamic array implementation of the dictionary, the **put** operation removes any prior association with the given key before insertion a new association. An alternative would be to search the list of associations, and if one is found simply replace the value field. If no association is found, then insert a new association. Write the **put** method using this approach. Which is easier to understand? Which is likely to be faster?

2. When Alan wishes to join the circle of six friends, why can't Amy simply increase the size of the vector to seven?

3. Amy's club has grown, and now includes the following members:

Abel	Abigail	Abraham	Ada
Adam	Adrian	Adrienne	Agnes
Albert	Alex	Alfred	Alice
Amanda	Amy	Andrew	Andy
Angela	Anita	Anne	Antonia
Arnold	Arthur	Audrey	

Find what value would be computed by Amy's hash function for each member of the group.

4. Assume we use Amy's hash function and assign each member to a bucket by simply dividing the hash value by the number of buckets. Determine how many elements would be assigned to each bucket for a hash table of size 5. Do the same for a hash table of size 11.

5. In searching for a good hash function over the set of integer values, one student thought he could use the following:

```
int index = (int) Math.sin(value);
```

What was wrong with this choice?

6. Can you come up with a perfect hash function for the names of the week? The names of the months? The names of the planets?

7. Examine a set of twelve or more telephone numbers, for example the numbers belonging to your friends. Suppose we want to hash into seven different buckets. What would be a good hash function for your set of telephone numbers? Will your function continue to work for new telephone numbers?

8. Experimentally test the birthday paradox. Gather a group of 24 or more people, and see if any two have the same birthday.

9. Most people find the birthday paradox surprising because they confuse it with the following: if there are *n* people in a room, what is the probability that somebody else shares *your* birthday. Ignoring leap years, give the formula for this expression, and evaluate this when *n* is 24.

10. The function containsKey can be used to see if a dictionary contains a given key. How could you determine if a dictionary contains a given value? What is the complexity of your procedure?

11.

Analysis Exercises

1.A variation on a map, termed a multi-map, allows multiple entries to be associated with the same key. Explain how the multimap abstraction complicates the operations of search and removal.

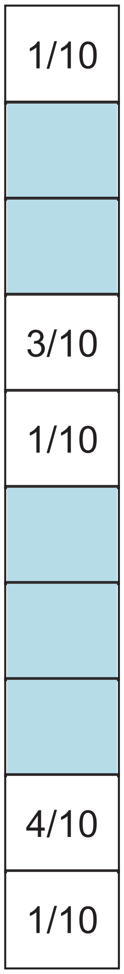
2. (The birthday Paradox) The frequency of collisions when performing hashing is related to a well known mathematical puzzle. How many randomly chosen people need be in a room before it becomes likely that two people will have the same birth date? Most people would guess the answer would be in the hundreds, since there are 365 possible birthdays (excluding leap years). In fact, the answer is only 24 people.

To see why, consider the opposite question. With n randomly chosen people in a room, what is the probability that no two have the same birth date? Imagine we take a calendar, and mark off each individual's birth date in turn. The probability that the second person has a different birthday from the first is 364/365, since there are 364 different possibilities not already marked. Similarly the probability that the third person has a different birthday from the first two is 363/365. Since these two probabilities are independent of each other, the probably that they are *both* true is their product. If we continue in this fashion, if we have n-1 people
all with different birthdays, the probability that individual n has a different birthday is: 364/365 * 363/365 * 363/365 * … * 365-n+1/365. When n >= 24 this expression

becomes less than 0.5. This means that if 24 or more people are gathered in a room the odds are better than even that two individuals have the same birthday.

The implication of the birthday paradox for hashing is to tell us that for any problem of reasonable size we are almost certain to have some collisions. Functions that avoid duplicate indices are surprisingly rare, even with a relatively large table.

2. (Clustering) Imagine that the colored squares in the ten-element table at right indicate values in a hash table that have already been filled. Now assume that the next value will, with equal probability, be any of the ten values. What is the probability that each of the free squares will be filled? Fill in the remaining squares with the correct probability.



Here is a hint: since both positions 1 and 2 are filled, any value that maps into these locations must go into the next free location, which if 3. So the probability that square 3 will be filled is the sum of the probabilities that the next item will map into position 1 (1/10) plus the probability that the next item will map into position 2 (which is 1/10) plus the probability that the next item will map into position 3 (also 1/10). So what is the final probability that position 3 will be filled? Continue with this type of analysis for the rest of the squares.

This phenomenon, where the larger a block of filled cells becomes, the more likely it is to become even larger, is known as clustering. (A similar phenomenon explains why groups of cars on a freeway tend to become larger).

Clustering is just one reason why it is important to keep the load factor of hash tables low.

Simply moving to the next free location is known as linear probing. Many alternatives to linear probing have been studied, however as open address hash tables are relatively rare we will not examine these alternatives here.

Show that probing by any constant amount will not reduce the problem caused by clustering, although it may make it more difficult to observe since clusters are not adjacent. To do this, assume that elements are uniformly inserted into a seven element hash table, with a linear probe value of 3. Having inserted one element, compute the probability that any of the remaining empty slots will be filled. (You can do this by simply testing the values 0 to 6, and observing which locations they will hash into. If only one element will hash into a location, then the probability is 1/7, if two the probability is 2/7, and so on). Explain why the empty locations do not all have equal probability. Place a value in the location with highest probability, and again compute the likelihood that any of the remaining empty slots will be filled. Extrapolate from these observations and explain how clustering will manifest itself in a hash table formed using this technique.

3. You can experimentally explore the effect of the load factor on the efficiency of a hash table. First, using an open address hash table, allow the load factor to reach 0.9 before you reallocate the table. Next, perform the same experiment, but reallocate as soon as the table reaches 0.7. Compare the execution times for various sets of operations. What is the practical effect? You can do the same for the hash table with bucket abstraction, using values larger than 1 for the load factor. For example, compare using the limit of 5 before reallocation to the same table where you allow the lists to grow to length 20 before reallocation.

4. To be truly robust, a hash table cannot perform the conversion of hash value into index using integer operations. To see why, try executing using the standard abs function, and compute and print the absolute value of the smallest integer number. Can you explain why the absolute value of this particular integer is not positive? What happens if you negate the value v? What will happen if you try inserting the value v into the hash table containers you have created in this chapter?

To avoid this problem the conversion of hash value into index must be performed first as a long integer, and then converted back into an integer.

```
Long longhash = abs((Long) hash(v));
int hashindex = (int) longhash;
```

Verify that this calculation results in a positive integer. Explain how it avoids the problem described earlier.

Programming Projects

1. Sometimes the dictionary abstraction is defined using a number of additional functions. Show how each of these can be implemented using a dynamic array. What is the big-oh complexity of each?

   a. int containsValue (struct dyArray *v, ValueType testValue)
   b. void clear (struct dyArray *v) // empty all entries from map
   c. boolean equals (struct dyArray *left, struct dyArray *right)
   d. boolean isEmpty (struct dyArray * v)
   e. void keySet (struct dyArray *v, struct dyArray *e) /* return a set containing all keys */
   f. void values (struct dyArray *v, struct dyArray *e) // return a bag containing all values

1. All our Bag abstractions have the ability to return an iterator. When used in the fashion of Worksheet D1, these iterators will yield a value of type Association. Show how to create a new inner class that takes an iterator as argument, and when requested

returns only the key porition of the association. What should this iterator do with the remove operation?

2. Although we have described hash tables in the chapter on dictionaries, we noted that the technique can equally well be applied to create a Bag like container. Rewrite the Hashtable class as a Hashbag that implements Bag operations, rather than dictionary operations.

3. An iterator for the hash table is more complex than an iterator for a simple bag. This is because the iterator must cycle over two types of collections: the array of buckets, and the linked lists found in each bucket. To do this, the iterator will maintain two values, an integer indicating the current bucket, and a reference to the link representing the current element being returned. Each time hasMore is called the current link is advanced. If there are more elements, the function returns true. If not, then the iterator must advance to the next bucket, and look for a value there. Only when all buckets have been exhausted should the function hasNext return false. Implement an iterator for your hash table implementation.

4. If you have access to a large online dictionary of English words (on unix systems such a dictionary can sometimes be found at /usr/lib/words) perform the following experiment. Add all the words into an open address hash table. What is the resulting size of the hash table? What is the resulting load factor? Change your implementation so that it will keep track of the number of probes used to locate a value. Try searching for a few words. What is the average probe length?

5. Another approach to implementing the Dictionary is to use a pair of parallel dynamic arrays. One array will maintain the keys, and the other one will maintain the values, which are stored in the corresponding positions.

picture

If the key array is sorted, in the fashion of the SortedArrayBag, then binary search can be used to quickly locate an index position in the key array. Develop a data structure based on these ideas.

6. Individuals unfamiliar with a foreign language will often translate a sentence from one language to another using a dictionary and word-for-word substitution. While this does not produce the most elegant translation, it is usually adequate for short sentences, such as ``Where is the train station?'' Write a program that will read from two files. The first file contains a series of word-for-word pairs for a pair of languages. The second file contains text written in the first language. Examine each word in the text, and output the corresponding value of the dictionary entry. Words not in the dictionary can be printed in the output surrounded by square brackets, as are [these] [words].

7. Implement the bit set spell checker as described earlier in this chapter.

On the Web

The dictionary data structure is described in wikipedia under the entry “Associative Array”. A large section of this entry describes how dictionaries are written in a variety of languages. A separate entry describes the Hash Table data structure. Hash tables with buckets are described in the entry “Coalesced hashing”.