Chapter 15

Maps

Concepts:
- ▷ Maps
- ▷ Hash tables
- ▷ Tables

X is very useful
if your name is
Nixie Knox.
It also
comes in handy
spelling ax
and extra fox.
—Theodor Seuss Geisel

WE HAVE SEEN THAT AN ASSOCIATION ESTABLISHES A LINK between a *key* and a *value*. An *associative array* or *map* is a structure that allows a disjoint set of keys to become associated with an arbitrary set of values. The convenience of an associative array is that the values used to index the elements need not be comparable and their range need not be known ahead of time. Furthermore, there is no upper bound on the size of the structure. It is able to maintain an arbitrary number of different pieces of information simultaneously. The analogy with a mathematical map or function stems from the notion that every key has at most associated value. Maps are sometimes called *dictionaries* because of the uniqueness of the association of words and definitions in a household dictionary. Needless to say, a map structure would nicely support the storage of dictionary definitions.

15.1 Example Revisited: The Symbol Table

In Chapter 14 we stored the words and their translations (name-alias pairs) in a structure called a `SymTab`. This structure forms a good basis for a more general-purpose approach. Here, we suggest a slightly modified program to accomplish exactly the same task. The names of the methods, however, have been changed to suggest slight improvements in the *semantics* of structure:

`SymMap`

```
public static void main(String args[])
{
    Map<String,String> table = new MapList<String,String>();
    Scanner s = new Scanner(System.in);
    String alias, name;

    // read in the alias-name database
```

```
        do
        {
            alias = s.next();
            if (!alias.equals("END"))
            {
                name = s.next();
                table.put(alias,name); // was called add, but may modify
            }
        } while (!alias.equals("END"));

        // enter the alias translation stage
        do
        {
            name = s.next();
            while (table.containsKey(name)) // was contains; more explicit
            {
                name = table.get(name); // translate alias
            }
            System.out.println(name);
        } while (s.hasNext());
    }
```

The differences between this implementation and that of Section 14.3 involve improvements in clarity. The method `add` was changed to `put`. The difference is that `put` suggests that the key-value pair is replaced if it is already in the `Map`. We also check for a value in the domain of the `Map` with `containsKey`. There might be a similar need to check the range; that would be accomplished with `containsValue`. Finally, we make use of a method, `keySet`, that returns a `Set` of values that are possible keys. This suggests aliases that might be typed in during the translation phase. Other methods might return a collection of `values`.

Thus we see that the notion of a `Map` formalizes a structure we have found useful in the past. We now consider a more complete description of the interface.

15.2 The Interface

In Java, a `Map` can be found within the `java.util` package. Each `Map` structure must have the following interface:

Map

```
public interface Map<K,V>
{
    public int size();
    // post: returns the number of entries in the map

    public boolean isEmpty();
    // post: returns true iff this map does not contain any entries

    public boolean containsKey(K k);
```

```
    // pre: k is non-null
    // post: returns true iff k is in the domain of the map

    public boolean containsValue(V v);
    // pre: v is non-null
    // post: returns true iff v is the target of at least one map entry;
    // that is, v is in the range of the map

    public V get(K k);
    // pre: k is a key, possibly in the map
    // post: returns the value mapped to from k, or null

    public V put(K k, V v);
    // pre: k and v are non-null
    // post: inserts a mapping from k to v in the map

    public V remove(K k);
    // pre: k is non-null
    // post: removes any mapping from k to a value, from the mapping

    public void putAll(Map<K,V> other);
    // pre: other is non-null
    // post: all the mappings of other are installed in this map,
    // overriding any conflicting maps

    public void clear();
    // post: removes all map entries associated with this map

    public Set<K> keySet();
    // post: returns a set of all keys associated with this map

    public Structure<V> values();
    // post: returns a structure that contains the range of the map

    public Set<Association<K,V>> entrySet();
    // post: returns a set of (key-value) pairs, generated from this map

    public boolean equals(Object other);
    // pre: other is non-null
    // post: returns true iff maps this and other are entry-wise equal

    public int hashCode();
    // post: returns a hash code associated with this structure
}
```

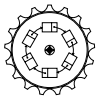
The put method places a new key-value pair within the Map. If the key was already used to index a value, that association is replaced with a new association between the key and value. In any case, the put method returns the value replaced or null. The get method allows the user to retrieve, using a key, the value from the Map. If the key is not used to index an element of the Map, a null value is returned. Because this null value is not distinguished from a stored value that is null, it is common to predicate the call to get with a call to the containsKey method. This method returns true if a key matching the parameter can be found within the Map. Sometimes, like human associative memory, it is useful to check to see if a *value* is found in the array. This can be accomplished with the containsValue method.

Aside from the fact that the keys of the values stored within the Map should be distinct, there are no other constraints on their type. In particular, the keys of a Map need only be accurately compared using the equals method. For this reason, it is important that a reasonable key equality test be provided.

There are no iterators provided with maps. Instead, we have a Map return a Set of keys (a keySet as previously seen), a Set of key-value pairs (entrySet), or any Structure of values (values). (The latter must not be a Set because values may be duplicated.) Each of these, in turn, can generate an Iterator with the iterator method. Because keys might not implement the Comparable class, there is no obvious ordering of the entries in a Map. This means that the keys generated from the keySet and the values encountered during an iteration over the values structure may appear in different orders. To guarantee the correct association, use the Iterator associated with the entrySet method.

15.3 Simple Implementation: MapList

One approach to this problem, of course, is to store the values in a List. Each mapping from a key to a value is kept in an Association which, in turn, is stored in a List. The result is what we call a MapList; we saw this in Section 15.2, though we referred to it as a generic Map structure. The approach is fairly straightforward. Here is the protected data declaration and constructors:



MapList

```
public MapList()
// post: constructs an empty map, based on a list
{
    data = new SinglyLinkedList<Association<K,V>>();
}

public MapList(Map<K,V> source)
// post: constructs a map with values found in source
{
    this();
    putAll(source);
}
```

It is conventional for complex structures to have a *copy constructor* that generates a new structure using the entries found in another `Map`. Notice that we don't make any assumptions about the particular *implementation* of the `Map` we copy from; it may be a completely different implementation.

Most of the other methods are fairly straightforward. For example, the `put` method is accomplished by finding a (possible) previous `Association` and replacing it with a fresh construction. The previous value (if any) is returned.

```
public V put(K k, V v)
// pre: k and v are non-null
// post: inserts a mapping from k to v in the map
{
    Association<K,V> temp = new Association<K,V>(k,v);
    Association<K,V> result = data.remove(temp);
    data.add(temp);
    if (result == null) return null;
    else return result.getValue();
}
```

The `Set` constructions make use of the `Set` implementations we have discussed in passing in our discussion of `Lists`:

```
public Set<K> keySet()
// post: returns a set of all keys associated with this map
{
    Set<K> result = new SetList<K>();
    Iterator<Association<K,V>> i = data.iterator();
    while (i.hasNext())
    {
        Association<K,V> a = i.next();
        result.add(a.getKey());
    }
    return result;
}

public Set<Association<K,V>> entrySet()
// post: returns a set of (key-value) pairs, generated from this map
{
    Set<Association<K,V>> result = new SetList<Association<K,V>>();
    Iterator<Association<K,V>> i = data.iterator();
    while (i.hasNext())
    {
        Association<K,V> a = i.next();
        result.add(a);
    }
    return result;
}
```

(We will discuss the implementation of various `Iterators` in Section 15.4; they are filtering iterators that modify `Associations` returned from subordinate iterators.) Notice that the uniqueness of keys in a `Map` suggests they form a `Set`,

yet this is checked by the Set implementation in any case. The values found in a Map are, of course, not necessarily unique, so they are stored in a general Structure. Any would do; we make use of a List for its simplicity:

```
public Structure<V> values()
// post: returns a structure that contains the range of the map
{
    Structure<V> result = new SinglyLinkedList<V>();
    Iterator<V> i = new ValueIterator<K,V>(data.iterator());
    while (i.hasNext())
    {
        result.add(i.next());
    }
    return result;
}
```

Exercise 15.1 *What would be the cost of performing a* containsKey *check on a* MapList*? How about a call to* containsValue*?*

Without giving much away, it is fairly clear the answers to the above exercise are not constant time. It would seem quite difficult to get a $O(1)$ performance from operators like contains and remove. We discuss the possibilities in the next section.

15.4 Constant Time Maps: Hash Tables

Clearly a collection of associations is a useful approach to filling the needs of the map. The costs associated with the various structures vary considerably. For Vectors, the cost of looking up data has, on average, $O(n)$ time complexity. Because of limits associated with being linear, all the $O(n)$ structures have similar performance. When data can be ordered, sorting the elements of the Linear structure improves the performance in the case of Vectors: this makes sense because Vectors are random access structures whose intermediate values can be accessed given an index.

When we considered binary search trees—a structure that also stores Comparable values—we determined the values could be found in logarithmic time. At each stage, the search space can be reduced by a factor of 2. The difference between logarithmic and linear algorithms is very dramatic. For example, a balanced BinarySearchTree or an ordered Vector might find one number in a million in 20 or fewer compares. In an unordered Vector the expected number of compares increases to 500,000.

Is it possible to improve on this behavior? With hash tables, the answer is, amazingly, *yes*. With appropriate care, the hash table can provide access to an arbitrary element in roughly constant time. By "roughly," we mean that as long as sufficient space is provided, each potential key can be reserved an undisturbed location with probability approaching 1.

How is this possible? The technique is, actually, rather straightforward. Here is an example of how hashing occurs in real life:

I was just going to say that.

> We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for *the last two digits* of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow.

The technique used by the appliance store was *hashing*. The "bin" or *bucket* that contains the object is identified by the last two digits of the phone number of the future owner. If two or more items were located in the bin, the name could be used to further distinguish the order.

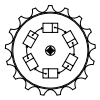
An alternative approach to the "addressing" of the bins might be to identify each bin with the first letter of the name of the customer. This, however, has a serious flaw, in that it is likely that there will be far more names that begin with S than with, say, K. Even when the entire name is used, the names of customers are unlikely to be evenly distributed. These techniques for addressing bins are less likely to uniquely identify the desired parcel.

That would be a large number of bins!

The success of the phone number technique stems from generating an identifier associated with each customer that is both random and evenly distributed.[1]

15.4.1 Open Addressing

We now implement a hash table, modeled after the `Hashtable` of Java's `java.util` package. All elements in the table are stored in a fixed-length array whose length is, ideally, prime. Initialization ensures that each slot within the array is set to `null`. Eventually, slots will contain references to associations between keys and values. We use an array for speed, but a `Vector` would be a logical alternative.



`Hashtable`

```
protected static final String RESERVED = "RESERVED";
protected Vector<HashAssociation<K,V>> data;
protected int count;

protected final double maximumLoadFactor = 0.6;

public Hashtable(int initialCapacity)
// pre: initialCapacity > 0
// post: constructs a new Hashtable
//       holding initialCapacity elements
{
    Assert.pre(initialCapacity > 0, "Hashtable capacity must be positive.");
```

[1] Using the last two digits of the telephone number makes for an evenly distributed set of values. It is *not* the case that the first two digits of the exchange would be useful, as that is not always random. In our town, where the exchange begins with 45, no listed phones have extensions beginning with 45.

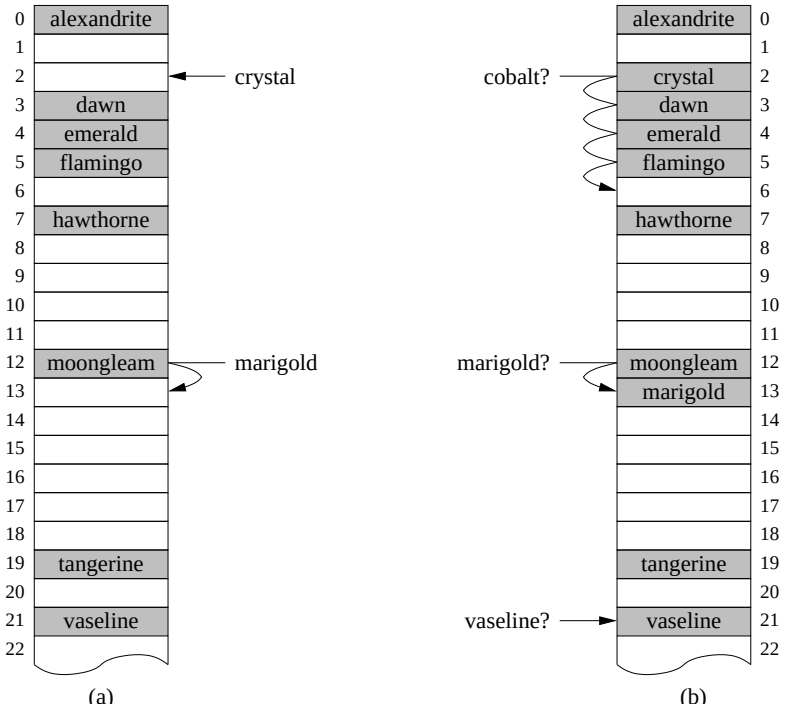


Figure 15.1 Hashing color names of antique glass. (a) Values are hashed into the first available slot, possibly after rehashing. (b) The lookup process uses a similar approach to possibly find values.

```
    data = new Vector<HashAssociation<K,V>>();
    data.setSize(initialCapacity);
    count = 0;
}

public Hashtable()
// post: constructs a new Hashtable
{
    this(997);
}
```

The key and value management methods depend on a function, `locate`, that finds a good location for a value in the structure. First, we use an index-producing function that "hashes" a value to a slot or bucket (see Figure 15.1). In Java, every `Object` has a function, called `hashCode`, that returns an integer to be used for precisely this purpose. For the moment, we'll assume the hash code is the alphabet code ($a = 0$, $b = 1$, etc.) of the first letter of the word. The

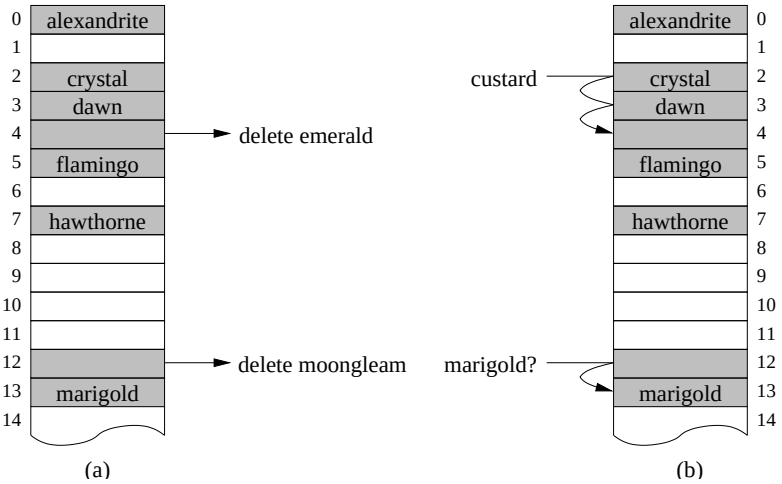


Figure 15.2 (a) Deletion of a value leaves a shaded reserved cell as a place holder. (b) A reserved cell is considered empty during insertion and full during lookup.

hash code for a particular key (2 for the word "crystal") is used as an index to the first slot to be considered for storing or locating the value in the table. If the slot is empty, the value can be stored there. If the slot is full, it is possible that another value already occupies that space (consider the insertion of "marigold" in Figure 15.1). When the keys of the two objects do not match, we have a *collision*. A *perfect hash function* guarantees that (given prior knowledge of the set of potential keys) no collisions will occur. When collisions do occur, they can be circumvented in several ways. With *open addressing*, a collision is resolved by generating a new hash value, or *rehashing*, and reattempting the operation at a new location.

Slots in the hash table logically have two states—empty (`null`) or full (a reference to an object)—but there is also a third possibility. When values are removed, we replace the value with a *reserved* value that indicates that the location potentially impacts the lookup process for other cells during insertions. That association is represented by the empty shaded cell in Figure 15.2a. Each time we come across the reserved value in the search for a particular value in the array (see Figure 15.2b), we continue the search as though there had been a collision. We keep the first reserved location in mind as a possible location for an insertion, if necessary. In the figure, this slot is used by the inserted value "custard."

When large numbers of different-valued keys hash or rehash to the same locations, the effect is called *clustering* (see Figure 15.3). *Primary clustering* is when several keys hash to the same initial location and rehash to slots with potential collisions with the same set of keys. *Secondary clustering* occurs when

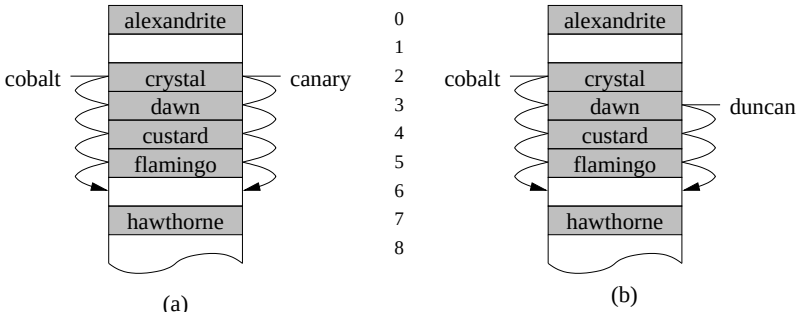


Figure 15.3 (a) *Primary clustering* occurs when two values that hash to the same slot continue to compete during rehashing. (b) Rehashing causes keys that initially hash to different slots to compete.

keys that initially hash to different locations eventually rehash to the same sequence of slots.

In this simple implementation we use *linear probing* (demonstrated in Figures 15.1 to 15.3). Any rehashing of values occurs a constant distance from the last hash location. The linear-probing approach causes us to wrap around the array and find the next available slot. It does not solve either primary or secondary clustering, but it is easy to implement and quick to compute. To avoid secondary clustering we use a related technique, called *double hashing*, that uses a second hash function to determine the magnitude of the constant offset (see Figure 15.4). This is not easily accomplished on arbitrary keys since we are provided only one `hashCode` function. In addition, multiples and factors of the hash table size (including 0) must also be avoided to keep the `locate` function from going into an infinite loop. Still, when implemented correctly, the performance of double hashing can provide significant improvements over linear-probing.

We now discuss our implementation of hash tables. First, we consider the `locate` function. Its performance is important to the efficiency of each of the public methods.

```
protected int locate(K key)
{
    // compute an initial hash code
    int hash = Math.abs(key.hashCode() % data.size());
    // keep track of first unused slot, in case we need it
    int reservedSlot = -1;
    boolean foundReserved = false;
    while (data.get(hash) != null)
    {
        if (data.get(hash).reserved()) {
```

```
            // remember reserved slot if we fail to locate value
            if (!foundReserved) {
                reservedSlot = hash;
                foundReserved = true;
            }
        } else  {
            // value located? return the index in table
            if (key.equals(data.get(hash).getKey())) return hash;
        }
        // linear probing; other methods would change this line:
        hash = (1+hash)%data.size();
    }
    // return first empty slot we encountered
    if (!foundReserved) return hash;
    else return reservedSlot;
}
```

To measure the difficulty of finding an empty slot by hashing, we use the *load factor*, α, computed as the ratio of the number of values stored within the table to the number of slots used. For open addressing, the load factor cannot exceed 1. As we shall see, to maintain good performance we should keep the load factor small as possible. Our maximum allowable load factor is a constant `maximumLoadFactor`. Exceeding this value causes the array to be reallocated and copied over (using the method `extend`).

When a value is added, we simply locate the appropriate slot and insert a new association. If the ideal slot already has a value (it must have an equal key), we return the replaced association. If we replace the reference to an empty cell with the `reserved` association, we return `null` instead.

```
public V put(K key, V value)
// pre: key is non-null object
// post: key-value pair is added to hash table
{
    if (maximumLoadFactor*data.size() <= (1+count)) {
        extend();
    }
    int hash = locate(key);
    if (data.get(hash) == null || data.get(hash).reserved())
    {   // logically empty slot; just add association
        data.set(hash,new HashAssociation<K,V>(key,value));
        count++;
        return null;
    } else {
        // full slot; add new and return old value
        HashAssociation<K,V> a = data.get(hash);
        V oldValue = a.getValue();
        a.setValue(value);
        return oldValue;
    }
}
```

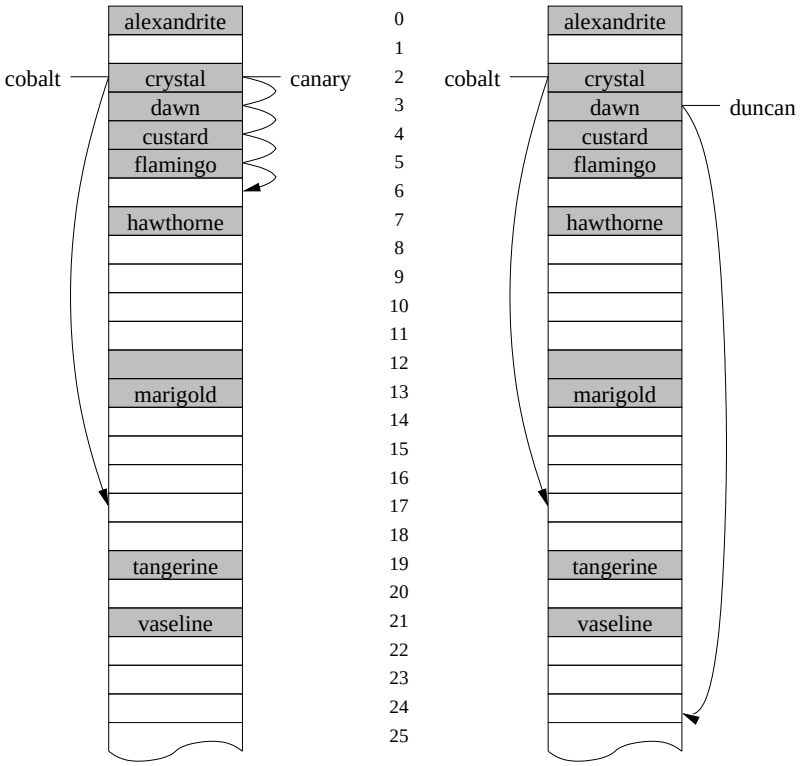


Figure 15.4 The keys of Figure 15.3 are rehashed by an offset determined by the alphabet code ($a = 1$, $b = 2$, etc.) of the *second* letter. No clustering occurs, but strings must have two letters!

The get function works similarly—we simply return the value from within the key-located association or null, if no association could be found.

```
public V get(K key)
// pre: key is non-null Object
// post: returns value associated with key, or null
{
    int hash = locate(key);
    if (data.get(hash) == null ||
        data.get(hash).reserved()) return null;
    return data.get(hash).getValue();
}
```

The containsKey method is similar. To verify that a value is within the table we build contains from the elements iterator:

```
public boolean containsValue(V value)
// pre: value is non-null Object
// post: returns true iff hash table contains value
{
    for (V tableValue : this) {
        if (tableValue.equals(value)) return true;
    }
    // no value found
    return false;
}

public boolean containsKey(K key)
// pre: key is a non-null Object
// post: returns true if key appears in hash table
{
    int hash = locate(key);
    return data.get(hash) != null && !data.get(hash).reserved();
}
```

The containsValue method is difficult to implement efficiently. This is one of the trade-offs of having a structure that is fast by most other measures.

To remove a value from the Hashtable, we locate the correct slot for the value and remove the association. In its place, we leave a reserved mark to maintain consistency in locate.

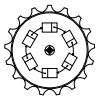
```
public V remove(K key)
// pre: key is non-null object
// post: removes key-value pair associated with key
{
    int hash = locate(key);
    if (data.get(hash) == null || data.get(hash).reserved()) {
        return null;
    }
    count--;
```

```
        V oldValue = data.get(hash).getValue();
        data.get(hash).reserve(); // in case anyone depends on us
        return oldValue;
    }
```

Hash tables are not made to be frequently traversed. Our approach is to construct sets of keys, values, and `Associations` that can be, themselves, traversed. Still, to support the `Set` construction, we build a single iterator (a `HashtableIterator`) that traverses the `Hashtable` and returns the `Associations`. Once constructed, the association-based iterator can be used to generate the key- and value-based iterators.

The protected iterator is similar to the `Vector` iterator. A current index points to the cell of the current non-`null` (and nonreserved) association. When the iterator is incremented, the underlying array is searched from the current point forward to find the next non-`null` entry. The iterator must eventually inspect every element of the structure, even if very few of the elements are currently used.[2]

Given an iterator that returns `Associations`, we can construct two different public filtering iterators, a `ValueIterator` and a `KeyIterator`. Each of these maintains a protected internal "slave" iterator and returns, as the iterator is incremented, values or keys associated with the respective elements. This design is much like the design of the `UniqueFilter` of Section 8.5. The following code, for example, implements the `ValueIterator`:



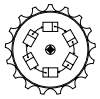
ValueIterator

```
class ValueIterator<K,V> extends AbstractIterator<V>
{
    protected AbstractIterator<Association<K,V>> slave;

    public <T extends Association<K,V>> ValueIterator(Iterator<T> slave)
    // pre: slave is an iterator returning Association elements
    // post: creates a new iterator returning associated values
    {
        this.slave = (AbstractIterator<Association<K,V>>)slave;
    }

    public boolean hasNext()
    // post: returns true if current element is valid
    {
        return slave.hasNext();
    }

    public V next()
    // pre: hasNext()
    // post: returns current value and increments iterator
    {
```

[2] The performance of this method could be improved by linking the contained associations together. This would, however, incur an overhead on the `add` and `remove` methods that may not be desirable.

```
            Association<K,V> pair = ((AbstractIterator<Association<K,V>>)slave).next();
            return pair.getValue();
        }
    }
```

Once these iterators are defined, the `Set` and `Structure` returning methods are relatively easy to express. For example, to return a `Structure` that contains the values of the table, we simply construct a new `ValueIterator` that uses the `HashtableIterator` as a source for `Associations`:



Hashtable

```
public Structure<V> values()
// post: returns a Structure that contains the (possibly repeating)
// values of the range of this map.
{
    List<V> result = new SinglyLinkedList<V>();
    Iterator<V> i = new ValueIterator<K,V>(new HashtableIterator<K,V>(data));
    while (i.hasNext())
    {
        result.add(i.next());
    }
    return result;
}
```

It might be useful to have direct access to iterators that return keys and values. If that choice is made, the `keys` method is similar but constructs a `KeyIterator` instead. While the `ValueIterator` and `KeyIterator` are protected, they may be accessed publicly when their identity has been removed by the `elements` and `keys` methods, respectively.

This is a form of identity laundering.

15.4.2 External Chaining

Open addressing is a satisfactory method for handling hashing of data, if one can be assured that the hash table will not get too full. When open addressing is used on nearly full tables, it becomes increasingly difficult to find an empty slot to store a new value.

One approach to avoiding the complexities of open addressing—reserved associations and table extension—is to handle collisions in a fundamentally different manner. *External chaining* solves the collision problem by inserting all elements that hash to the same bucket into a single collection of values. Typically, this collection is a singly linked list. The success of the hash table depends heavily on the fact that the average length of the linked lists (the *load factor* of the table) is small and the inserted objects are uniformly distributed. When the objects are uniformly distributed, the *deviation* in list size is kept small and no list is much longer than any other.

The process of locating the correct slot in an externally chained table involves simply computing the initial `hashCode` for the key and "modding" by the table size. Once the appropriate bucket is located, we verify that the collection is constructed and the value in the collection is updated. Because our `List`

classes do not allow the retrieval of internal elements, we may have to remove and reinsert the appropriate association.

ChainedHashTable

```
public V put(K key, V value)
// pre: key is non-null object
// post: key-value pair is added to hash table
{
    List<Association<K,V>> l = locate(key);
    Association<K,V> newa = new Association<K,V>(key,value);
    Association<K,V> olda = l.remove(newa);
    l.addFirst(newa);
    if (olda != null)
    {
        return olda.getValue();
    }
    else
    {
        count++;
        return null;
    }
}
```

Most of the other methods are implemented in a similar manner: they locate the appropriate bucket to get a `List`, they search for the association within the `List` to get the association, and then they manipulate the key or value of the appropriate association.

One method, `containsValue`, essentially requires the iteration over two dimensions of the hash table. One loop searches for non-`null` buckets in the hash table—buckets that contain associations in collections—and an internal loop that explicitly iterates across the `List` (the `containsKey` method can directly use the `containsValue` method provided with the collection). This is part of the price we must pay for being able to store arbitrarily large numbers of keys in each bucket of the hash table.

```
public boolean containsValue(V value)
// pre: value is non-null Object
// post: returns true iff hash table contains value
{
    for (V v : this) {
        if (value.equals(v)) return true;
    }
    return false;
}
```

At times the implementations appear unnecessarily burdened by the interfaces of the underlying data structure. For example, once we have found an appropriate `Association` to manipulate, it is difficult to modify the key. This is reasonable, though, since the value of the key is what helped us locate the

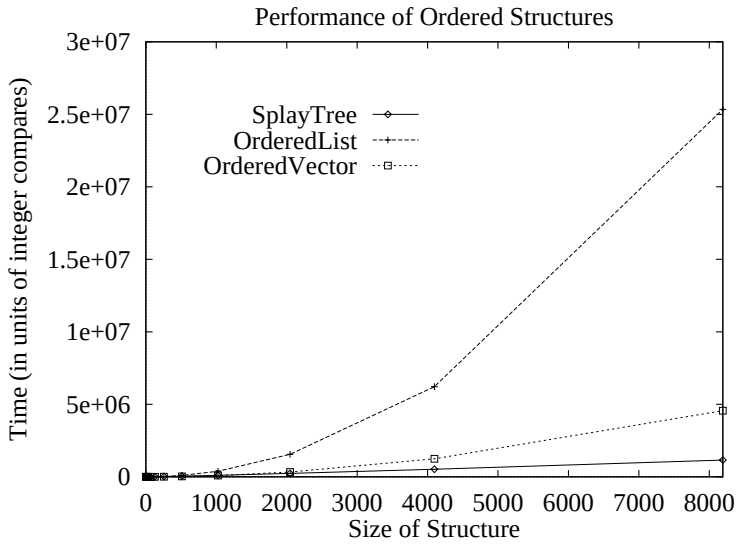


Figure 15.5 The time required to construct large ordered structures from random values.

bucket containing the association. If the key could be modified, we could insert a key that was inconsistent with its bucket's location.

Another subtle issue is the selection of the collection class associated with the bucket. Since linked lists have poor linear behavior for most operations, it might seem reasonable to use more efficient collection classes—for example, tree-based structures—for storing data with common hash codes. The graph of Figure 15.5 demonstrates the performance of various ordered structures when asked to construct collections of various sizes. It is clear that while `SplayTrees` provide better ultimate performance, the simple linear structures are more efficient when the structure size is in the range of expected use in chained hash tables (see Figure 15.6). When the average collection size gets much larger than this, it is better to increase the size of the hash table and re-insert each of the elements (this is accomplished with the `Hashtable` method, `extend`).

15.4.3 Generation of Hash Codes

Because any object might eventually be stored within a hash table, and because data abstraction hides the details of implementation, it is important for implementors to provide a `hashCode` method for their classes whenever possible.

Principle 24 *Provide a method for hashing the objects you implement.*

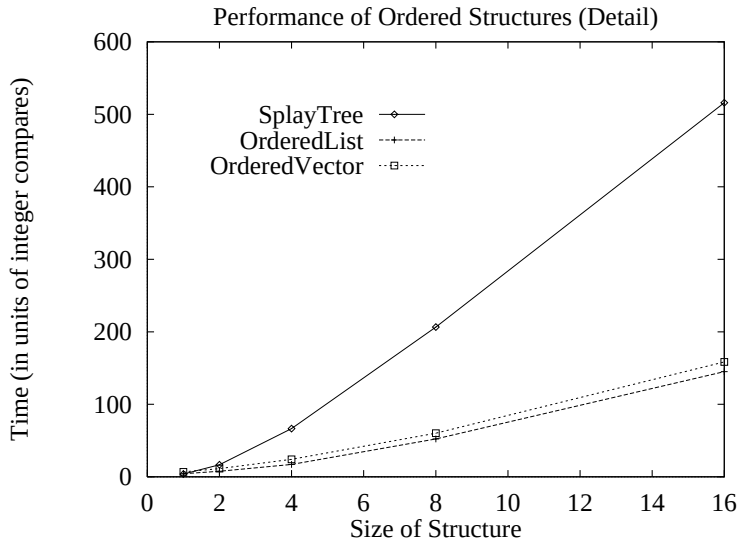


Figure 15.6 The time required to construct small ordered structures from random values.

When a `hashCode` method *is* provided, it is vital that the method return the same `hashCode` for any pair of objects that are identified as the same under the `equals` method. If this is not the case, then values indexed by equivalent keys can be stored in distinct locations within the hash table. This can be confusing for the user and often incorrect.

Principle 25 *Equivalent objects should return equal hash codes.*

The generation of successful hash codes can be tricky. Consider, for example, the generation of hash codes for `Strings`. Recall that the purpose of the hash code generation function is to distribute `String` values uniformly across the hash table.

Most of the approaches for hashing strings involve manipulations of the characters that make up the string. Fortunately, when a character is cast as an integer, the internal representation (often the ASCII encoding) is returned, usually an integer between 0 and 255. Our first approach, then, might be to use the first character of the string. This has rather obvious disadvantages: the first letters of strings are not uniformly distributed, and there isn't any way of generating hash codes greater than 255.

Our next approach would be to sum all the letters of the string. This is a simple method that generates large-magnitude hash codes if the strings are long. The main disadvantage of this technique is that if letters are transposed,

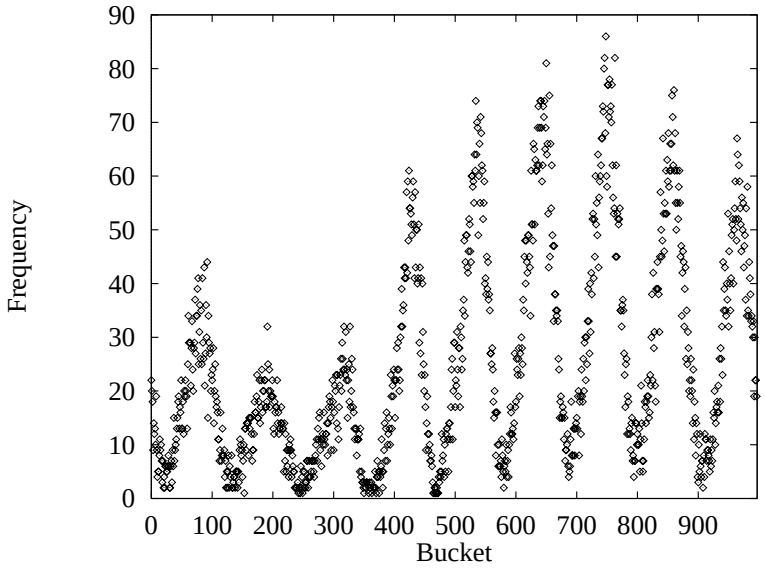


Figure 15.7 Numbers of words from the UNIX spelling dictionary hashing to each of the 997 buckets of a default hash table, if sum of characters is used to generate hash code.

---

then the strings generate the same hash values. For example, the string `"dab"` has $100 + 97 + 98 = 295$ as its sum of ASCII values, as does the string `"bad"`. The string `"bad"` and `"bbc"` are also equivalent under this hashing scheme. Figure 15.7 is a histogram of the number of words that hash, using this method, to each slot of a 997 element hash table. The periodic peaks demonstrate the fact that some slots of the table are heavily preferred over others. The performance of looking up and modifying values in the hash table will vary considerably, depending on the slot that is targeted by the hash function. Clearly, it would be useful to continue our search for a good mechanism.

Another approach might be to weight each character of the string by its position. To ensure that even very short strings have the potential to generate large hash values, we can provide exponential weights: the hash code for an l character string, s, is

$$\sum_{i=0}^{l-1} s[i]c^i$$

where c is usually a small integer value. When c is 2, each character is weighted by a power of 2, and we get a distribution similar to that of Figure 15.8. While this is closer to being uniform, it is clear that even with exponential behavior, the value of $c = 2$ is too small: not many words hash to table elements with

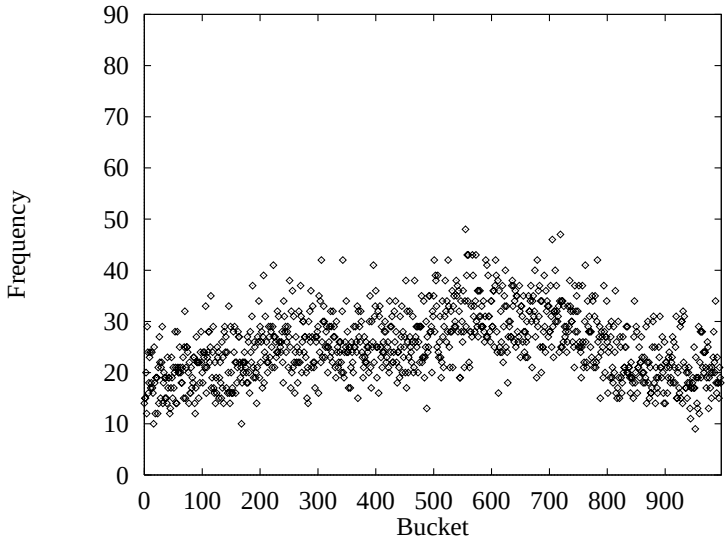


Figure 15.8 Frequency of dictionary words hashing to each of 997 buckets if characters are weighted by powers of 2 to generate hash code.

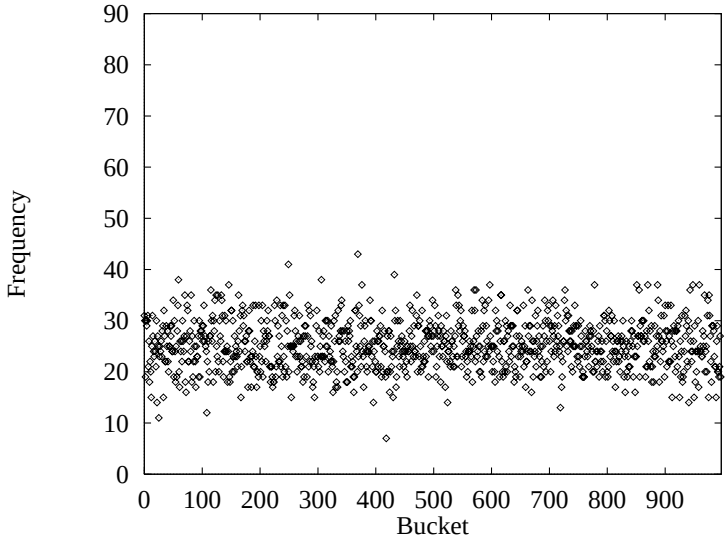


Figure 15.9 Frequency of words from dictionary hashing to each of 997 buckets if hash code is generated by weighting characters by powers of 256.

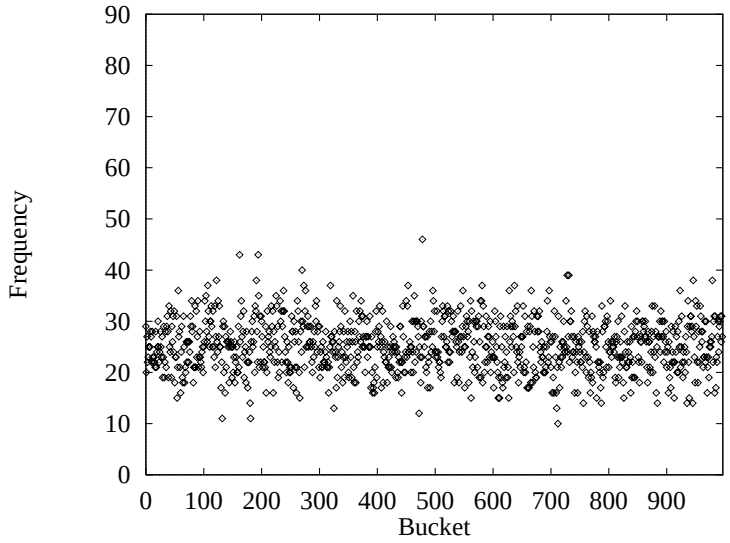


Figure 15.10 Frequency of words from dictionary hashing to each of 997 buckets, using the Java `String` hash code generation.

large indices. When $c = 256$, the hash code represents the first few characters of the string exactly (see Figure 15.9). Java currently hashes with $c = 31$.

The hashing mechanism used by Java `Strings` in an early version of Java's development environment (see Figure 15.10) used a combination of weightings that provided a wide range of values for short strings and was efficient to compute for longer strings. Unfortunately, the constant-time algorithm was not suitable for distinguishing between long and nearly identical strings often found, say, in URLs.

Method	**Successful**	**Unsuccessful**
Linear probes	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$
Double hashing	$\frac{1}{\alpha} \ln \frac{1}{(1-\alpha)}$	$\frac{1}{1-\alpha}$
External chaining	$1 + \frac{1}{2}\alpha$	$\alpha + e^{-\alpha}$

Figure 15.11 Expected theoretical performance of hashing methods, as a function of α, the current load factor. Formulas are for the number of association compares needed to locate the correct value or to demonstrate that the value cannot be found.

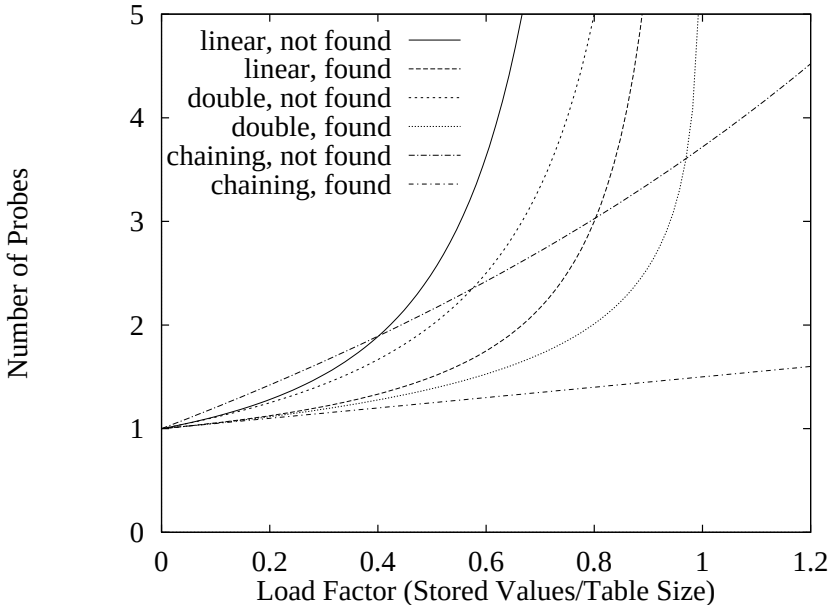


Figure 15.12 The shape of the theoretical performance curves for various hashing techniques. (These graphs demonstrate *theoretical predictions* and not *experimental results* which are, of course, dependant on particular data and hashing functions.) Our hash table implementation uses linear probing.

Many of the data structures we have investigated are classes that contain multiple objects of unspecified type. When hashing entire container classes, it can be useful to compose the codes of the contained elements.

15.4.4 Hash Codes for Collection Classes

Each of the techniques used to generate hash codes from a composition of characters of `Strings` can be used to compose hash codes of objects in collection classes. The features of primary importance for the construction of hash codes are:

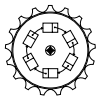
1. Whenever two structures are equal, using the `equals` methods, the `hashCode` method should return the same value.

2. For container structures—structures whose only purpose is to hold values—the state of the structure itself should be transparent; the state of the structure should not be included in the `hashCode`.

The first item was discussed before, but it is the most common error leading to difficulties with the use of `Hashtables`. When the `hashCodes` do not match for objects that are logically equal, the objects have a high probability of entering into different locations of the table. Accesses that should interact do not.

The second consideration is understood if we consider self-modifying structures like `SplayTrees` or `Hashtables` whose external state may be modeled by several distinct internal states. The construction of the hash code should consider those bits of information that enter into identifying equal structures. In the case of the `SplayTree`, for example, we might use the sum of the hash codes of the values that appear within the tree.

In general, the following first attempt at a `hashCode` method, taken from the `AbstractStructure` type, is reasonable:



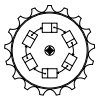
`Abstract-Structure`

```
public int hashCode()
// post: generate a hashcode for the structure: sum of
// all the hash codes of elements
{
    Iterator<E> i = iterator();
    int result = 0;
    while (i.hasNext())
    {
        E o = i.next();
        result = result * 31;
        if (o != null) result += o.hashCode();
    }
    return result;
}
```

As we can see here, we must constantly be on the watch for values that may potentially be `null` references. For some structures, of course, such an approach

may lead to intolerable amounts of time computing `hashCode`s for large structures.

One last point concerns the `hashCode`s associated with recursively defined structures. If the recursive structure is visible externally, that is, the structure could be referenced at several points, it may be suitable to define the `hashCode` to be the value contained within a single node of the structure. This certainly fulfills the primary obligation of a hashing function, but it also serves to separate the structure from the hash code. In our case, we choose to make a recursive definition, similar to the following definition found in `BinaryTree`:



BinaryTree

```
public int hashCode()
// post: return sum of hashcodes of the contained values
{
    if (isEmpty()) return 0;
    int result = left().hashCode() + right().hashCode();
    if (value() != null) result += value().hashCode();
    return result;
}
```

15.4.5 Performance Analysis

For open addressing, the load factor α obviously cannot exceed 1. As the load factor approaches 1, the performance of the table decreases dramatically. By counting the number of *probes* or association compares needed to find a value (a successful search) or to determine that the value is not among the elements of the map (an unsuccessful search), we can observe the relative performance of the various hashing techniques (see Figures 15.11 and 15.12). Notice that the number of probes necessary to find an appropriate key is a function of the load factor, and not directly of the number of keys found in the table.

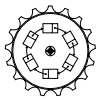
When a hash table exceeds the maximum allowable load factor, the entire table is forced to expand, and each of the keys is rehashed. Looking at the graph in Figure 15.12, we select our threshold load factor to be 60 percent, the point at which the performance of linear probing begins to degrade. When we expand the hash table, we make sure to at least double its size. For the same reasons that doubling is good when a `Vector` is extended, doubling the size of the hash table improves the performance of the hash table without significant overhead.

15.5 Ordered Maps and Tables

In fact, better hash functions probably avoid order!

A significant disadvantage of the `Map` interface is the fact that the values stored within the structure are not kept in any particular order. Often we wish to efficiently maintain an ordering among key-value pairs. The obvious solution is to construct a new `OrderedMap` that builds on the interface of the `Map`, but where methods may be allowed to assume parameters that are `Comparable`:

```
public interface OrderedMap<K extends Comparable<K>,V> extends Map<K,V>
{
}
```

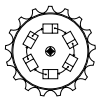


OrderedMap

When we do this, the methods of the Map are inherited. As a result, the types of the key-based methods manipulate Objects and not Comparables. Because we desire to maintain order among comparable keys, we have a general precondition associated with the use of the data structure—that keys provided and returned must be objects supporting the Comparable interface.

Even with comparable keys, it is not easy to construct a Hashtable whose keys iterator returns the keys in order. The hash codes provided for Comparable objects are not required (and unlikely) to be ordered in a way consistent with the compareTo function. We therefore consider other OrderedStructures to maintain the order among ComparableAssociations.

We will call our implementation of the OrderedMap a Table. As a basis for the implementation, we depend on the SplayTree class. OrderedLists and OrderedVectors could also provide suitable implementations for small applications. The Table maintains a single protected data item—the SplayTree. The constructor is responsible for allocating the SplayTree, leaving it initialized in its empty state:



Table

```
protected OrderedStructure<ComparableAssociation<K,V>> data;

public Table()
// post: constructs a new table
{
    data = new SplayTree<ComparableAssociation<K,V>>();
}

public Table(Table<K,V> other)
{
    data = new SplayTree<ComparableAssociation<K,V>>();
    Iterator<Association<K,V>> i = other.entrySet().iterator();
    while (i.hasNext())
    {
        Association<K,V> o = i.next();
        put(o.getKey(),o.getValue());
    }
}
```

When a key-value pair is to be put into the Table, a ComparableAssociation is constructed with the key-value pair, and it is used to look up any previous association using the same key. If the association is present, it is removed. In either case, the new association is inserted into the tree. While it seems indirect to remove the pair from the table to update it, it maintains the integrity of the ComparableAssociation and therefore the SplayTree. In addition, even though two keys may be logically equal, it is possible that they may be distinguishable. We insert the actual key-value pair demanded by the user, rather

than perform a partial modification. Theoretically, removing and inserting a value into the SplayTree costs the same as finding and manipulating the value in place. Next, we see the method for put (get is similar):

```
public V put(K key, V value)
// pre: key is non-null object
// post: key-value pair is added to table
{
    ComparableAssociation<K,V> ca =
        new ComparableAssociation<K,V>(key,value);
    // fetch old key-value pair
    ComparableAssociation<K,V> old = data.remove(ca);
    // insert new key-value pair
    data.add(ca);
    // return old value
    if (old == null) return null;
    else return old.getValue();
}
```

While most of the other methods follow directly from considering Hashtables and SplayTrees, the contains method—the method that returns true exactly when a particular value is indexed by a key in the table—potentially requires a full traversal of the SplayTree. To accomplish this, we use an Iterator returned by the SplayTree's elements methods. We then consider each association in turn, returning as soon as an appropriate value is found:

```
public boolean containsValue(V value)
// pre: value is non-null object
// post: returns true iff value in table
{
    Iterator<V> i = iterator();
    while (i.hasNext())
    {
        V nextValue = i.next();
        if (nextValue != null &&
            nextValue.equals(value)) return true;
    }
    return false;
}
```

Next, our Table must provide an Iterator to be used in the construction of the keySet and entrySet. The approach is similar to the Hashtable—we construct a private Association-returning Iterator and then return its KeyIterator or ValueIterator. Because every value returned from the SplayTree's iterator is useful,[3] we need not implement a special-purpose iterator for

[3] Compare this with, perhaps, a Vector iterator that might be used to traverse a Vector-based Hashtable.

Tables; instead, we use the SplayTree's iterator directly. Since Comparable-Associations extend Associations, the KeyIterator generates an Iterator that returns the comparable keys as Objects to be cast later.

```
public Set<K> keySet()
// post: returns a set containing the keys referenced
// by this data structure.
{
    Set<K> result = new SetList<K>();
    Iterator<K> i = new KeyIterator<K,V>(data.iterator());
    while (i.hasNext())
    {
        result.add(i.next());
    }
    return result;
}

public Set<Association<K,V>> entrySet()
// post: returns a structure containing all the entries in
// this Table
{
    Set<Association<K,V>> result = new SetList<Association<K,V>>();
    Iterator<ComparableAssociation<K,V>> i = data.iterator();
    while (i.hasNext())
    {
        result.add(i.next());
    }
    return result;
}
```

Previous hard work greatly simplifies this implementation! Since no hashing occurs, it is not necessary for any of the keys of a Table to implement the hashCode method. They must, though, implement the compareTo method since they are Comparable. Thus, each of the methods runs in amortized logarithmic time, instead of the near-constant time we get from hashing.

Exercise 15.2 *Modify the* Table *structure to make use of* RedBlackTrees*, instead of* SplayTrees*.*

Exercise 15.3 *It is common to allow ordered structures, like* OrderedMap*, to use a* Comparator *to provide an alternative ordering. Describe how this approach might be implemented.*

15.6 Example: Document Indexing

Indexing is an important task, especially for search engines that automatically index keywords from documents retrieved from the Web. Here we present the

skeleton of a document indexing scheme that makes use of a `Map` to keep track of the vocabulary.

Given a document, we would like to generate a list of words, each followed by a list of lines on which the words appear. For example, when provided Gandhi's seven social sins:

```
 politics without principle
 pleasure without conscience
   wealth without work
knowledge without character
 business without morality
  science without humanity
            and
  worship without sacrifice
```

(It is interesting to note that `programming without comments` is not among these!) The indexing program should generate the following output:

```
and: 7
business: 5
character: 4
conscience: 2
humanity: 6
knowledge: 4
morality: 5
pleasure: 2
politics: 1
principle: 1
sacrifice: 8
science: 6
wealth: 3
without: 1 2 3 4 5 6 8
work: 3
worship: 8
```

In this program we make use of Java's `StreamTokenizer` class. This class takes a stream of data and converts it into a stream of tokens, some of which are identified as words. The process for constructing this stream is a bit difficult, so we highlight it here.

Index

```
public static void main(String args[])
{
    try {
        InputStreamReader isr = new InputStreamReader(System.in);
        java.io.Reader r = new BufferedReader(isr);
        StreamTokenizer s = new StreamTokenizer(r);
          ...
    } catch (java.io.IOException e) {
        Assert.fail("Got an I/O exception.");
    }
}
```

Each of the objects constructed here provides an additional layer of filtering on the base stream, `System.in`. The body of the main method is encompassed by the `try` statement in this code. The `try` statement catches errors generated by the `StreamTokenizer` and rewraps the exception as an assertion failure.

We begin by associating with each word of the input an initially empty list of line numbers. It seems reasonable, then, to use the vocabulary word as a key and the list of lines as the value. Our `Map` provides an ideal mechanism to maintain the data. The core of the program consists of reading word tokens from the stream and entering them into the `Map`:

```
// allocate the symbol table (uses comparable keys)
Map<String,List<Integer>> t = new Table<String,List<Integer>>();
int token;
// we'll not consider period as part of identifier
s.ordinaryChar('.');
// read in all the tokens from file
for (token = s.nextToken();
     token != StreamTokenizer.TT_EOF;
     token = s.nextToken())
{
    // only tokens we care about are whole words
    if (token == StreamTokenizer.TT_WORD)
    {
        // each set of lines is maintained in a List
        List<Integer> l;

        // look up symbol
        if (t.containsKey(s.sval))
        {   // symbol is there, get line # list
            l = t.get(s.sval);
            l.addLast(s.lineno());
        } else {
            // not found, create new list
            l = new DoublyLinkedList<Integer>();
            l.addLast(s.lineno());
            t.put(s.sval,l);
        }
    }
}
```

Here, we use a `Table` as our `Map` because it is important that the entries be sorted alphabetically. As the tokens are read from the input stream, they are looked up in the `Map`. Since the `Map` accepts comparable keys, it is important to use a (comparable) `String` to allow the words to index the structure. If the key is within the `Map`, the value associated with the key (a list) is updated by appending the current line number (provided by the stream's `lineno` method) to the end of the list. If the word is not found, a new list is allocated with the current line appended, and the fresh word–list pair is inserted into the table.

The next section of the program is responsible for generating the output:

```
// printing table involves tandem key-value iterators
Iterator<List<Integer>> ki = t.values().iterator();
for (String sym : t.keySet())
{
    // print symbol
    System.out.print(sym+": ");
    // print out (and consume) each line number
    for (Integer lineno : ki.next())
    {
        System.out.print(lineno+" ");
    }
    System.out.println();
    // increment iterators
}
```

Here, two iterators—one for keys and one for values—are constructed for the `Map` and are incremented in parallel. As each word is encountered, it is printed out along with the list of line numbers, generated by traversing the list with an iterator.

Because we used a `Table` as the underlying structure, the words are kept and printed in sorted order. If we had elected to use a `Hashtable` instead, the output would appear shuffled. The order is neither alphabetical nor the order in which the words are encountered. It is the result of the particular hash function we chose to locate the data.

15.7 Conclusions

In this chapter we have investigated two structures that allow us to access values using a key or index from an arbitrary domain. When the keys can be uniformly distributed across a wide range of values, hashing is an excellent technique for providing constant-time access to values within the structure. The cost is extra time necessary to hash the value, as well as the extra space needed to keep the load factor small enough to provide the expected performance.

When the keys are comparable, and order is to be preserved, we must depend on logarithmic behavior from ordered structures we have seen before. In our implementation of `Tables`, the `SplayTree` was used, although any other `OrderedStructure` could be used instead.

Because of the nonintuitive nature of hashing and hash tables, one of the more difficult tasks for the programmer is to generate useful, effective hash code values. Hash functions should be designed specifically for each new class. They should be fast and deterministic and have wide ranges of values. While all `Objects` inherit a `hashCode` function, it is important to update the `hashCode` method whenever the `equals` method is changed; failure to do so leads to subtle problems with these useful structures.

Self Check Problems

Solutions to these problems begin on page 450.

15.1 What access feature distingishes `Map` structures from other structrues we have seen?

15.2 What is the load factor of a hash table?

15.3 In a hash table is it possible to have a load factor of 2?

15.4 Is a constant-time performance guaranteed for hash tables?

15.5 What is a hash collision?

15.6 What are the qualities we seek in a hash function?

15.7 Under what condition is a `MapList` preferable to a `Hashtable`?

15.8 Many of our more complex data structures have provided the underpinnings for efficient sorts. Is that the case for the `Hashtable`? Does the `Table` facilitate sorting?

Problems

Solutions to the odd-numbered problems begin on page 484.

15.1 Is it possible for a hash table to have two entries with equal keys?

15.2 Is it possible for a hash table to have two entries with equal values?

15.3 Suppose you have a hash table with seven entries (indexed 0 through 6). This table uses open addressing with the hash function that maps each letter to its alphabet code ($a = A = 0$, etc.) modulo 7. Rehashing is accomplished using linear-probing with a jump of 1. Describe the state of the table after each of the letters `D`, `a`, `d`, `H`, `a`, and `h` are added to the table.

15.4 Suppose you have a hash table with eight entries (indexed 0 through 7). The hash mechanism is the same as for Problem 15.3 (alphabet code modulo 8), but with a linear probe jump of 2. Describe what happens when one attempts to add each of the letters `A`, `g`, `g`, `a`, and `g`, in that order. How might you improve the hashing mechanism?

15.5 When using linear probing with a rehashing jump size of greater than 1, why is it necessary to have the hash table size and jump size be relatively prime?

15.6 Design a `hashCode` method for a class that represents a telephone number.

15.7 Design a `hashCode` method for a class that represents a real number.

15.8 Suppose two identifiers—`Strings` composed of letters—were considered equal even if their cases were different. For example, `AGEdwards` would be equal to `AgedWards`. How would you construct a hash function for strings that was “case insensitive”?

15.9 When 23 randomly selected people are brought together, chances are greater than 50 percent that two have the same birthday. What does this tell us about uniformly distributed hash codes for keys in a hash table?

15.10 Write a `hashCode` method for an `Association`.

15.11 Write a `hashCode` method for a `Vector`. It should only depend on hash codes of the `Vector`'s elements.

15.12 Write a `hashCode` method for a `BinaryTree`. Use recursion.

15.13 Write a `hashCode` method for a `Hashtable`. (For some reason, you'll be hashing hash tables into other hash tables!) Must the hashing mechanism look at the value of *every* element?

15.14 The Java hash function for `String`s computes a hash code based on a fixed maximum number of characters of the string. Given that `String`s have no meaningful upper bound in length, describe how an effective, constant-time hashing algorithm can be constructed. (Hint: If you were to pick, say, eight characters to represent a string of length l, which would you choose?)

15.15 Since URLs differ mostly toward their end (at high indices), write code that efficiently computes a hash code based on characters $l - x_i$ where $x_i = 2^i$ and $i = 0, 1, 2, \ldots$ How fast does this algorithm run? Is it better able to distinguish different URLs?

15.16 A hash table with *ordered linear probing* maintains an order among keys considered during the rehashing process. When the keys are encountered, say, in increasing order, the performance of a failed lookup approaches that of a successful search. Describe how a key might be inserted into the ordered sequence of values that compete for the same initial table entry.

15.17 Isn't the hash table resulting from Problem 15.16 just an ordered `Vector`? (Hint: No.) Why?

15.18 If we were to improve the iterators for `Map`s, we might add an iterator that returned key-value pairs. Is this an improvement in the interface?

15.19 Design a hash function for representing the state of a checkerboard.

15.20 Design a hash function for representing the state of a tic-tac-toe board. (It would—for strategy reasons—be useful to have mirror images of a board be considered equal.)

15.21 One means of potentially reducing the complexity of computing the hash code for `String`s is to compute it once—when the `String` is constructed. Future calls to `hashCode` would return the precomputed value. Since the value of a `String` never changes, this has potential promise. How would you evaluate the success of such a method?

15.22 Explain how a `Map` might be useful in designing a spelling checker. (Would it be useful to have the words `bible` and `babble` stored near each other?)

15.8 Laboratory: The Soundex Name Lookup System

Objective. To use a `Map` structure to keep track of similar sounding names.

Discussion. The United States National Archives is responsible for keeping track of the census records that, according to the Constitution, must be gathered every 10 years. After a significant amount of time has passed (70 or more years), the census records are made public. Such records are of considerable historical interest and a great many researchers spend time looking for lost ancestors among these records.

To help researchers find individuals, the censuses are often indexed using a phonetic system called *Soundex*. This system takes a name and produces a short string called the Soundex key. The rules for producing the Soundex key of a name are precisely:

1. The entire name is translated into a series of digit characters:

Character	Letter of name
`'1'`	b, p, f, v
`'2'`	c, s, k, g, j, q, x, z
`'3'`	d, t
`'4'`	l
`'5'`	m, n
`'6'`	r
`'7'`	*all other letters*

   For example, `Briggs` would be translated into the string `167222`.

2. All double digits are reduced to single digits. Thus, `167222` would become `1672`.

3. The first digit is replaced with the first letter of the original name, in uppercase. Thus, `1672` would become `B672`.

4. All 7's are removed. Thus, `B672` becomes `B62`.

5. The string is truncated to four characters. If the resulting string is shorter than four characters, it is packed with enough `'0'` characters to bring the length to four. The result for `Briggs` would be `B620`. Notice that, for the most part, the nonzero characters represent the significant sounded letters of the beginning of the name.

Other names translate to Soundex keys as follows:

Bailey	becomes	B400
Ballie	becomes	B400
Knuth	becomes	K530
Scharstein	becomes	S623
Lee	becomes	L000

Procedure. You are to write a system that takes a list of names (the UNIX spelling dictionary is a good place to find names) and generates an ordered map whose entries are indexed by the Soundex key. The values are the actual names that generated the key. The input to the program is a series of names, and the output is the Soundex key associated with the name, along with all the names that have that same Soundex key, in alphabetical order.

Pick a data structure that provides performance: the response to the query should be nearly instantaneous, even with a map of several thousand names.

Thought Questions. Consider the following questions as you complete the lab:

1. What is the Soundex system attempting to do when it encodes many letters into one digit? For example, why are 'd' and 't' both encoded as `'3'`?

2. Why does the Soundex system ignore any more than the first four sounds in a name?

Notes: