Chapter 5

Hash Tables

In previous chapters, the ideas of analysis of algorithms and abstract data types were introduced and illustrated using data structures and algorithms with which you were already familiar, at least in a general way. From now on, we will be working mostly with new data structures and algorithms and applying the ideas we have learned to them.

In this chapter, we look at two related abstract data types, *Set* and *Map*, and at a concrete data structure, the *hash table*, that can be used to implement them efficiently.

Mathematically, a set is just a collection of elements, with no duplicates allowed. The elements are in no particular order. The primary operators on a set are adding and removing elements and testing whether a given item is an element of the set. There are also set-theoretical operators such as union and intersection, but we will not include these in our definition of the *Set* ADT:

Definition 5.1. Let *BaseType* be a data type. We define the abstract data type **Set of BaseType** as follows. The possible values of this type are unordered collections of items of type *BaseType*, containing no duplicate items. The operations of the ADT are:

- **add(x):** Adds an item, x, of type *BaseType* to the set, if it is not already in the set. Adding an item that is already in the set is not an error but has no effect on the set.
- **remove(x):** Removes an item, x, of type *BaseType* from the set, if it is in the set. Removing an item that is not in the set is not an error and has no effect on the set.
- **contains(x):** Returns a boolean value indicating whether x is contained in the set.
- **size():** Returns an integer value equal to the number of items in the set.

For example, a set of strings could be used to implement a spelling dictionary where the main operations are to check whether a given word is in the dictionary and to add new words to the dictionary. Another typical application occurs when a task requires searching for and processing items. If an item can be "encountered" several times during the search but should only be processed once, a set can be used to hold items that have already been processed. When an item is encountered, it can be tested for membership in the set. If it is in the set, it has already been processed and can be ignored. If not, it should be processed and added to the set.

In many applications, however, in addition to a set of items, there is a *value* associated to each item in the set. For example, an ordinary dictionary is a set of words, where each word has an associated definition. A phone directory is a set of names, where each name has an associated phone number. An Internet DNS database contains a set of domain names, such as *math.hws.edu*, and their associated IP addresses. All these are examples of the *Map* ADT. A Map consists of a set of items, where each items has an associated value. We refer to the items in this context as **keys**. A map is a collection of key/value pairs. The keys form a set; that is, no duplicate keys are allowed. The basic operation is, given a key, look up its associated value. We also want to be able to set the value associated with a given key and to remove a key and its associated value from the map. This leads us to our Map ADT:

Definition 5.2. Let *KeyType* and *ValueType* be data types. We define the abstract data type **Map of (KeyType, ValueType)** as follows. The possible values of this type are unordered collections of pairs (k, v), where k is of type *KeyType* and v is of type *ValueType*. Among the pairs in the collection, no duplicate key values are allowed. The operations of the ADT are as follows (with k standing for a value of type *KeyType* and v for a value of type *ValueType*):

get(k): Returns the value associated to the key k. That is, if there is a pair (k, v) in the map, then v is the return value of this function. It is an error to call this function when there is no such pair.[1]

put(k,v): Set the associated value for k to v. If no pair with key value k exists in the map, then the pair (k, v) is added to the map. If a pair with key value k already exists, then v replaces the value in the existing pair.

remove(k): Remove the pair with key value k from the map, if there is such a pair. If there is no such pair in the map, this has no effect.

containsKey(k): Returns a boolean value indicating whether the map contains a pair with key value k.

size(): Returns an integer value equal to the number of pairs in the map.

There are many ways to implement sets and maps. In the remainder of this chapter, we will look at hash tables, which can be used to implement both sets and maps. We will encounter some alternative implementations in the next chapter.

5.1 Open Hash Tables

The basic idea of a hash table is that each item that might be stored in the table has a natural location in the table. This natural location can be computed as a function of the item itself. When searching for an item, it is not necessary to go through all the possible locations in the table; instead, you can go directly to the natural location of the item. The function that computes the natural location of an item in a hash table is called a **hash function**. A hash table is in fact an array, so a location in a hash table is specified by an integer index. When a hash table is used to implement sets, the hash function takes a possible element of the set and returns an integer value that specifies the index in the array where that element should be stored. When a hash table is

[1]In many implementations, it is not an error to call *get(k)* when k has no associated value. Instead, a special value (such as `NULL`), is returned to represent "no value."

used to implement maps, the hash function is applied to a key and it returns the expected position of the pair with that key. The integer returned by a hash function is referred to as the **hash code** of the function's input value. That is, the hash code of a value is the natural location for that value in the hash table.

The obvious problem that arises is that an array is a finite data structure with a relatively small number of locations. On the other hand, the number of different values that are candidates for storage in the table is typically very large or even (as in the case of strings) infinite. This means that it is not possible to assign a different hash code to each possible value. When we try to store two values that have the same hash code in a hash table, the result is what we call a **collision**. That is, a collision occurs when two items that we want to store in a hash table both have the same natural location in the table. The question is, where do we put the second item?

It is tempting to think that we could avoid collisions by making the table large enough. Surely, if the size of the table is significantly larger than the number of items that we want to store in the table, then the chance of a collision should be negligible. However, this is not the case. This is related to what is known in probability theory as the "birthday problem": In a group of m people, what is the probability that there are two people who have the same day of the year as their birthday? When asked to guess, most people would severely underestimate the probability. For a group of 30 people, the chance that two of them have the same birthday is already 70%. For 50 people, it is 97%. Even for a group of 10 people, the chance that two of them will have the same birthday is already greater than 10%. In terms of hash tables, we might ask this type of question: If we want to store 100 items in the table, how large should the table be if we want to keep the chance of a collision less than 1%. The answer is that the table size must be greater than about 490,000. Clearly, the table would contain a ridiculous amount of wasted space. For a more reasonable table size—say, 200 spaces for 100 items—collisions are a virtual certainty.[2]

Since we can't avoid collisions, we have to decide what to do when one occurs. That is, suppose that we want to insert an item into a hash table, and the natural location of that item is already occupied by another item. What do we do with the new item?

One obvious solution is to allow a location in the hash table to hold more than one item. To accomplishes this, the hash table is implemented as an array of *lists*. When an empty hash table is first created, each location in the array contains an empty list. To add an item to the hash table, simply go to the location given by the hash code of that item and add the item to the list stored at that location. A hash table that stores a list of items at each location is called an **open hash table**, and the technique is referred to as "open hashing."

The lists used in an open hash table are typically simple linked lists. Extra features such as header nodes, tail pointers, and pointers to previous nodes are overhead that are not needed to implement hash table operations. The hash table itself is implemented as an array of pointers, and an empty location in the array is represented by a `NULL` pointer. When a hash table is used to implement a Set, each list node will contain one of the elements of the set and a pointer for linking to the next item in the list. For a map, the node would contain a key, its associated value, and a pointer. Without writing any actual code, let's look at how each operation of the Map ADT would

[2]These calculations assume that each location in the table, or each possible birthday, is equally likely to be chosen. If this is not the case, then collisions are even more likely.

be implemented in an open hash table. Assume that *table* is the name of the array of list pointers and that $hash(k)$ is the hash function that takes a key as input and returns its hash code. Then we can implement the operations as follows:

- **get(k)**: Let $loc = hash(k)$. For each node in the list $table[loc]$, if the key in that node is k, then return the value of the node.
- **put(k,v)**: Let $loc = hash(k)$. For each node in the list $table[loc]$, if the key in that node is k, then set the value in that node to v and return. If k is not found in any node on the list, add a new node containing k and v to the beginning of the list.
- **containsKey(k)**: Let $loc = hash(k)$. For each node in the list $table[loc]$, if the key in that node is k, then return true. If k is not found in any node on the list, return false.
- **remove(k)**: Let $loc = hash(k)$. For each node in the list $table[loc]$, if the key in that node is k, then remove that node from the list and return.

Where these algorithms say "for each node in the list," a loop would be used to move a pointer along the list from one node to the next in the usual way.

Provided that certain conditions are met, all these hash-table operations are very efficient. First, we want to assume that the size of the hash table is somewhat larger than the number of items that are stored in the table. Typically, we would like to assume that the number of items is no more than 75% of the array size. This means that the average length of the lists in the hash table is less than one. However, this is not enough to guarantee that the operations will be efficient: If most of the keys hash to just a few locations, then the lists at those locations will be very long, and for most keys, we will end up doing inefficient linear searches on these long lists. So, as a second condition, we would like the items to be evenly distributed among the possible array locations. That is, for a given item the probability that that item has a certain hash code should be the same for every possible hash code. Under these assumptions, collisions will still occur, but the large majority of the lists in the hash table will have length zero or one, and none will be very long. In fact, the average run time for each of the hash table operations will be $\Theta(1)$.

In practice, it's not always easy to ensure that these conditions are met. Since the number of items in the table can vary over time, the optimum size for the hash table cannot usually be specified in advance. So, in many implementations, hash tables are resizable. When the number of items in the table exceeds some threshold, such as 75% of the array size, a new larger array is created and the items from the old array are moved to the new array. Since the range of possible locations has changed, the hash function will also have to change, and a new hash code will have to be computed for each item to determine its location in the new array. This process of increasing the table size and putting all items into their correct locations in the new table is sometimes referred to as **re-hashing**. Re-hashing is an expensive operation, but in a typical application it is one that will only rarely occur.

To achieve the second condition—that items are distributed evenly over the available array locations—we just need a good hash function.

5.2 Hash Functions

A hash function takes a key (or one of the possible elements of a set) as input, and it computes an integer value. The return value is the hash code, which is the index of the array location where that key should be stored. The hash code must lie in the range 0 to $n-1$ where n is the size of the array. Furthermore, a good hash function will distribute the keys evenly over the range of possible hash codes. To some extent, of course, this depends on which keys actually occur among the items that we what to store in the table. However, we will assume that all the keys are equally likely so that the only question is how well the hash function distributes the keys.

If the keys are integers, then a natural choice for the hash function is $hash(k) = k \mod n$; that is, $hash(k)$ is the remainder when the integer k is divided by the array size. For other key types, we first compute some integer function, $h(k)$, from the key and then compute the hash code as $h(k) \mod n$. It is the function $h(k)$ that we will consider in the rest of this section.[3]

Essentially, we want to choose a function $h(k)$ that will make a "hash" of k. If k consists of several 32-bit numerical values, for example, they might be combined using the bitwise exclusive or operation, which is represented in C++ by the operator `^`. Even better, apply a circular shift to mix things up a bit more. Since any value can be considered to be a collection of bits, ideas like these can be applied to produce hash functions for any type of key. Here, for example, is a function that might be used if the key value is a string (which in C++ is essentially just an array of integers):

```
unsigned int h( string str ) {
   unsigned int x = 0x11111111;  // start with a mix of 0's and 1's
   for (int i = 0; i < str.length(); i++) {
      x = x ^ str[i];   // combine next char with x
      x = (x >> 5) | (x << 27);  // circular right-shift by 5 bits
   }
   return x;
}
```

This function is likely to give a better result than, for example, simply adding up the individual character values, which would tend to give a small number as output unless the string was very long and would tend to cluster the values to some extent since some characters are so much more likely to occur than others.

5.3 Closed Hash Tables

In an open hash table, the table is implemented as an array of pointers. The actual items in the table are stored outside this array, in linked list nodes that are allocated and deallocated dynamically.

[3]In fact, it is $h(k)$ that is often considered to be the hash function, and the reduction modulo n is assumed. If we use this terminology, then the same hash function is used regardless of the array size; only the value of n will be different. Note in particular that re-hashing a table to increase the array size will not involve selecting a new hash function.

In some situations, we might want to avoid the use of dynamic memory, or we might want to avoid using the extra memory that is required for storing the linked list pointers. In such cases, we can use a **closed hash table**. In closed hashing, all the items are stored in the array itself rather than in linked lists attached to the array. Since each location of the array can accommodate only one item, we need a new strategy for handling collisions.

The simplest strategy is called **linear probing**. Suppose that we want to store an item in the table. Let h be the hash code for the item. Suppose that location h in the array is already occupied. With linear probing, we simply look at following locations $h+1$, $h+2$, and so on, until we find an empty location. (If we reach the end of the array without finding an empty spot, we loop back to the beginning of the array.) When we find an empty spot, we store the item there. When we search for the item, we just follow the same strategy, looking first in the location given by its hash code and then in following locations. Either we find the item, or we come to an empty spot, which tells us that the item is not in the table.[4]

A problem with linear probing is that it leads to excessive clustering. When an item is bumped from location x, it goes into location $x+1$. Now, any item that is bumped from either location x or location $x+1$ is bumped into location $x+2$. Then, any item that belongs in any of these three locations is bumped into location $x+3$. All these items in consecutive locations form a cluster. The larger a cluster grows, the more likely it is that more items will join the cluster. And the more items there are in the cluster, the less efficient searching for those items will be. Note that not all items in the cluster have the same hash code—when an item with *any* hash code hits the cluster, it will join the cluster.

There are other probing strategies that are less prone to clustering. In **quadratic probing**, the sequence of locations that we probe for hash code h is h, $h+1^2$, $h+2^2$, $h+3^2$, $h+4^2$, $\ldots$ (with wrap-around if we get to the end of the array). The point is not that these locations are not physically next to each other—all items with hash code h will follow the same probe sequence and will form a sort of physically spread-out cluster. The point is that when another item with a *different* hash code hits this cluster, it will not join the cluster. For example, if location $h+2^2$ is occupied by an item with hash code h, and we try to insert an item with hash code $h+2^2$ into the table, it will be bumped out of its place. However, the next probe location for the new item will be location $(h+2^2)+1$, not location $h+3^2$, so it will not join the cluster of items with hash code h. Because it is less prone to clustering, a closed hash table that uses quadratic probing can be more efficient than one that uses linear probing.

[4]The remove operation gets complicated for closed hash tables. You can't simply remove an item by deleting it from the array. Suppose that when we try to store item B, its spot is already occupied by item A. We then store B in a different location. Now suppose that we remove A and leave an empty spot in the array. Now, when we search for B, we will see that empty spot and stop searching.