Chapter 21 Hashing: Implementing Dictionaries and Sets

1.

If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index. The array that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*.

How do you design a hash function that produces an index from a key? Ideally, we would like to design a function that maps each search key to a different index in the hash table. Such a function is called a *perfect hash function*. However, it is difficult to find a perfect hash function. When two or more keys are mapped to the same hash value, we say that a *collision* has occurred.
2.
A typical hash function first converts a search key to an integer value called a *hash code*, and then compresses the hash code into an index to the hash table.
For a search key of the type byte, short, int, and char, simply cast it to int. So two different search keys of any one of these types will have different hash codes.

3.
The hash code for a key can be a large integer that is out of the range for the hash table index. You need to scale it down to fit in the range of the index. Assume the index for a hash table is between 0 and N-1. The most common way to scale an integer to between 0 and N-1 is to use

```
h(hashCode) = hashCode % N
```

To ensure that the indices are spread evenly, choose N to be a prime number greater than 2.
8.
Yes.

4.
Open addressing is to find an open location in the hash table in the event of collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and *double hashing*.
When a collision occurs during the insertion of an entry to a hash table, linear probing finds the next available location sequentially.

Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \% n$, for $j \geq 0$, i.e., k, (k + 1) % n, (k + 4) % n, (k + 9) % n, ..., and so on.
Another open addressing scheme that avoids the clustering problem is known as *double hashing*. Starting from the initial index k, both linear probing and quadratic probing add an increment to k to define a search sequence. The increment is 1 for linear probing and j^2 for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash

function on the keys to determine the increments to avoid the clustering problem.

5.
Linear probing tends to cause groups of consecutive cells in the hash table to be occupied. Each group is called a *cluster*. Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even larger clusters, further slowing down the search time. This is a big disadvantage of linear probing.

6.
Quadratic probing works in the same way as linear probing except for the change of search sequence. Quadratic probing avoids the clustering problem in linear probing, but it has its own clustering problem, called *secondary clustering*, i.e., the entries that collide with an occupied entry use the same probe sequence.

7.
Omitted

8.
Omitted

9.
Omitted

10.
Omitted

11.
Omitted

12.
Load factor λ measures how full the hash table is. It is the ratio of the size of the map to the size of the hash table, i.e., $\lambda = \frac{n}{N}$, where *n* denotes the number of elements and *N* the number of locations in the hash table.