CS382: Object Oriented Programming and Data Structures

Linked List

DemoLinkListPolynomial.java

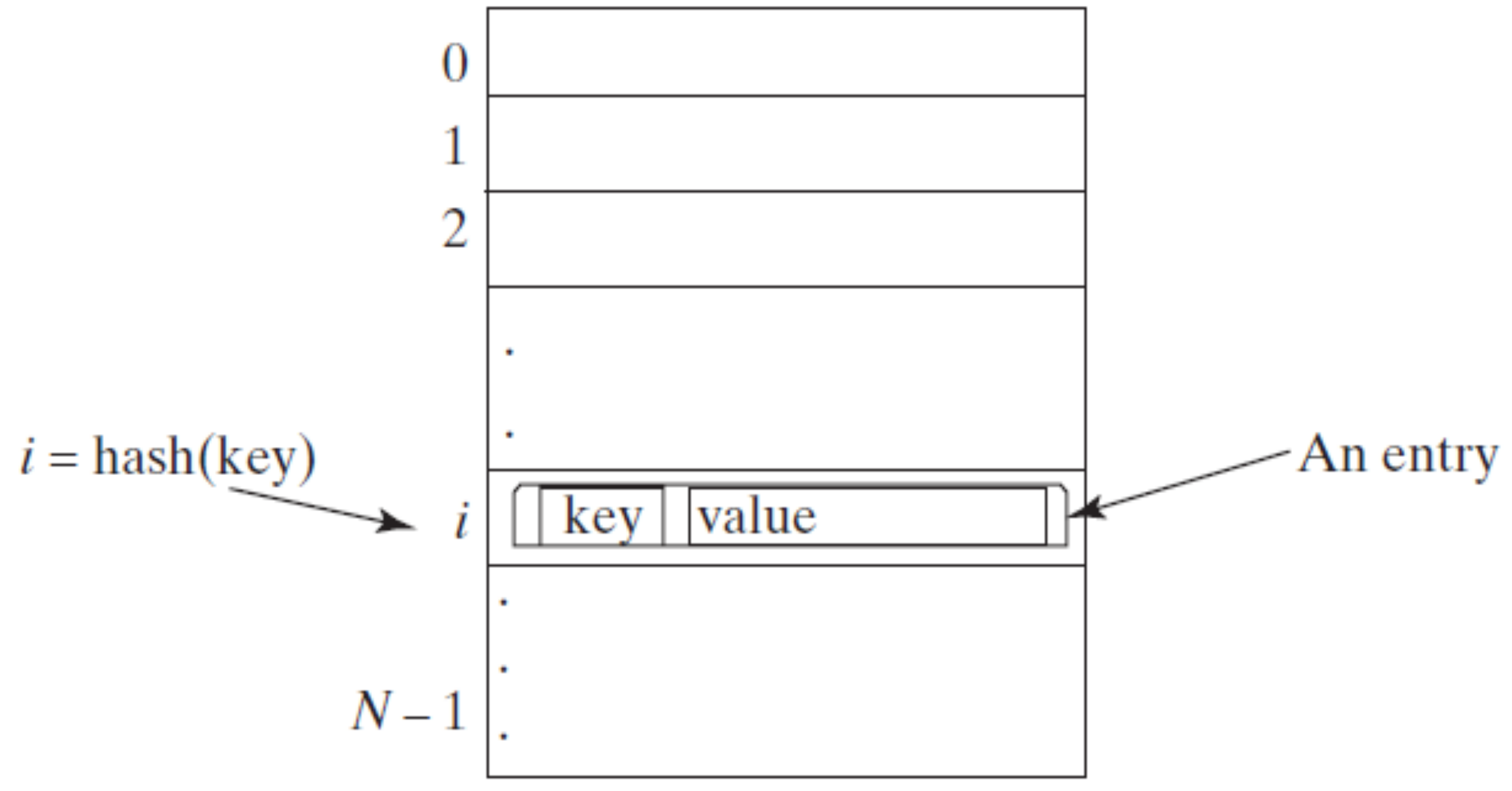
Is there a more efficient way to search for an element in a container?

Linear search: time complexity O(n)
Binary Search: time complexity O(log n)
Binary Search tree: time complexity O(log n)

Hashing technique takes O(1) time to search, insert, and delete an element using hashing.

What Is Hashing?

Hashing uses a hashing function to map a key to an index.



Hash function

A hash function maps a key to an index in the hash

Hash table and hash function

- Each entry contains two parts: a *key* and a *value*
- The key, also called a *search key, is* used to search for the corresponding value
- For example, a dictionary can be stored in a map,
in which the words are the keys and the definitions of the words are the values

Analogy: 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.*

Hash table, hash function and Collision

- 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

Hash Functions and Hash Codes

A typical hash function first converts a search key to an integer value called a hash code, then compresses the hash code into an index to the hash table.

- Java's root class **Object** has the **hashCode** method, which returns an integer hash code
- By default, the method returns the memory address for the object

Hash Functions and Hash Codes

A typical hash function first converts a search key to an integer value called a hash code, then compresses the hash code into an index to the hash table.

- Java's root class **Object** has the **hashCode** method, which returns an integer hash code
- By default, the method returns the memory address for the object
- You should override the **hashCode** method

Hash Codes for Primitive Types

For search keys of the type **byte, short, int, and char**, simply cast them to **int.**

Therefore, two different search keys of any one of these types will have different hash codes.

For a search key of the type **float,** use **Float.floatToIntBits(key)** as the hash code

floatToIntBits(float f) returns an **int** value whose bit representation is the same as the bit representation for the floating number **f**

Hash Codes for Primitive Types

Thus, two different search keys of the **float** type will have different hash codes
For a search key of the type **long**, simply casting it to **int** would not be a good choice, because all keys that differ in only the first 32 bits will have the same hash code
To take the first 32 bits into consideration, divide the 64 bits into two halves and perform the exclusive-or
operation to combine the two halves. This

Hash Codes for Primitive Types

The hash code for a **long** key is

```
int hashCode = (int)(key ^ (key >> 32));
```

Hash Codes for Strings

An intuitive approach is to sum the Unicode of all characters as the hash code for the string.

This approach may work if two search keys in an application don't contain the same letters, but it will produce a lot of collisions if the search keys contain the same letters, such as **tod** and **dot.**

Hash Codes for Strings

A better approach is to generate a hash code that takes the position of characters into considerati

$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \ \ldots + s_{n-1}$$

where s_i *is s.charAt(i)*. This expression is a polynomial for some positive *b*, so this is called a *polynomial hash code*.

Compressing Hash Codes

- The hash code for a key can be a large integer that is out of the range for the hash-table index, so you need to scale it down to fit in the index's range.
- 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

primary hash function and supplemental hash func

h(hashCode) = hashCode % N

Ideally, you should choose a prime number for **N.** However, it is time consuming to find a large prime number. In the Java API implementation for **java.util.HashMap, N** is set to a value of the power of 2

Handling Collisions Using Open Addressing

- A collision occurs when two keys are mapped to the same index in a hash table.
- Generally, there are two ways for handling collisions: open addressing and separate chaining.

  Open addressing is the process of finding an open location in the hash table in the event of a collision. Open addressing has several variations: *linear probing*, *quadratic probing*, *and double hashing*.

Linear Probing

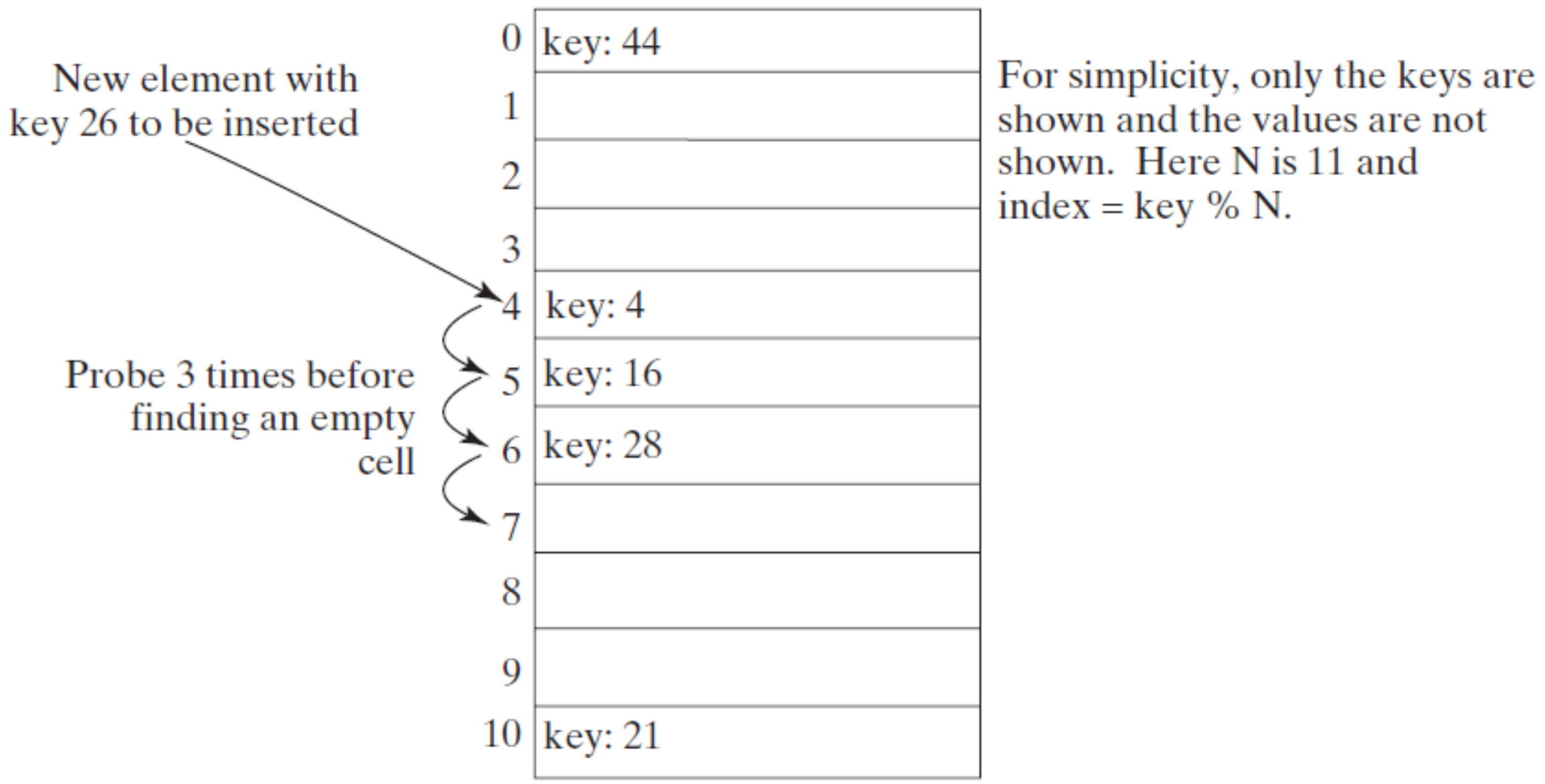
- When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially
- For example, if a collision occurs at **hashTable[k %N]**, check whether **hashTable[(k+1) % N]** is available. If not, check **hashTable[(k+2)% N]** and so on, until an available cell is found

Linear Probing

- When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds the next available location sequentially
- For example, if a collision occurs at **hashTable[k %N]**, check whether **hashTable[(k+1) % N]** is available. If not, check **hashTable[(k+2)% N]** and so on, until an available cell is found

When probing reaches the end of the table, it goes back to the beginning of the table. Thus, the hash table is treated as if it were circular.

Linear Probing


New element with key 26 to be inserted
Probe 3 times before finding an empty cell
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 16
key: 28
key: 21
For simplicity, only the keys are shown and the values are not shown. Here N is 11 and index = key % N.

Linear Probing

To search for an entry in the hash table, obtain the index, say **k,** from the hash function for the key. Check whether **hashTable[k % N]** contains the entry. If not, check whether **hashTable[(k+1) % N]** contains the entry, and so on, until it is found, or an empty cell is reached.

Linear Probing

To remove an entry from the hash table, search the entry that matches the key. If the entry is found, place a special marker to denote that the entry is available. Each cell in the hash table has three possible states: occupied, marked, or empty. Note that a marked cell is also available for insertion.

Disadvantage of Linear Probing

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

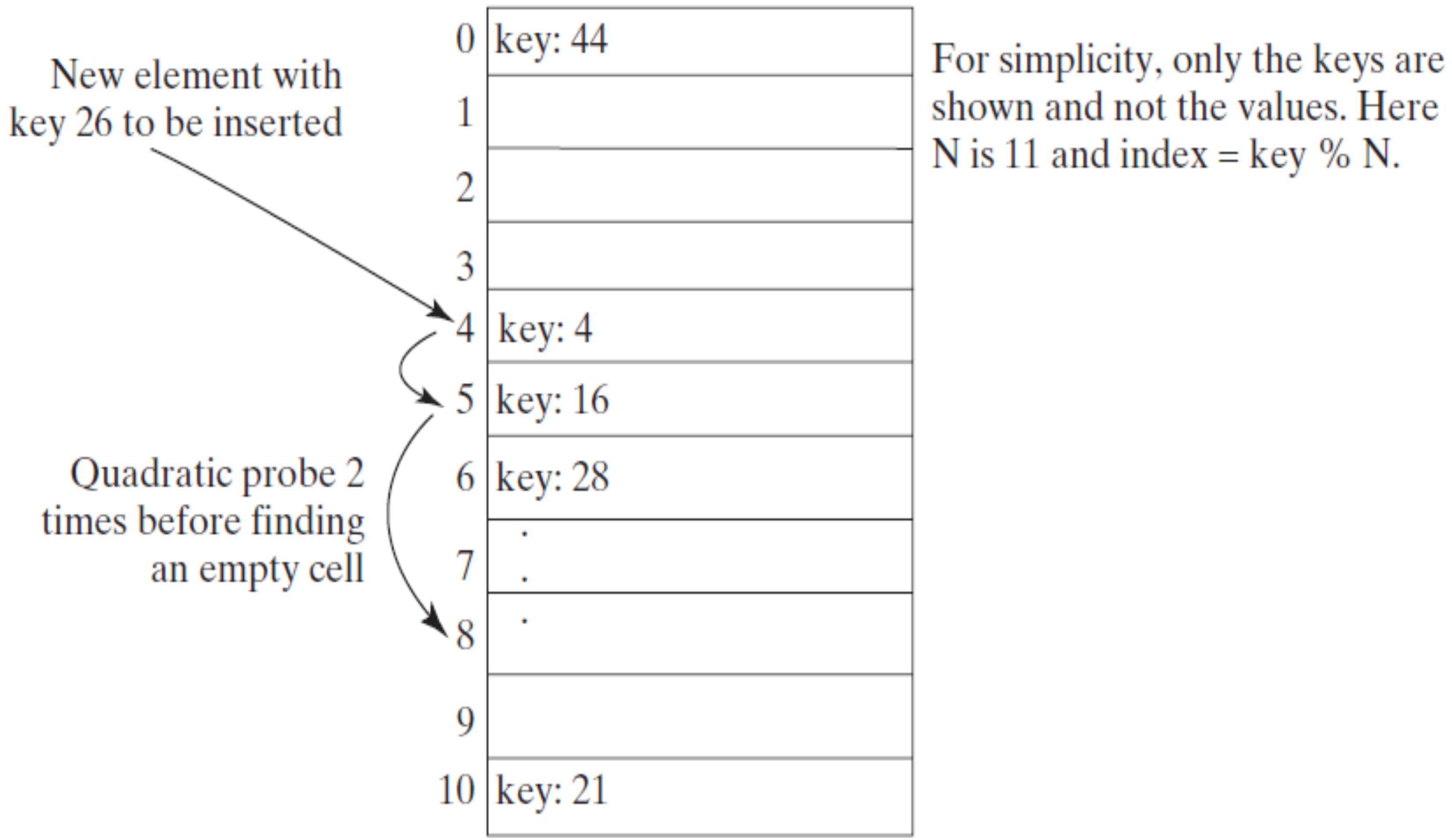
larger clusters, further slowing down the search time. This is a big disadvantage of linear

Quadratic Probing

Quadratic probing can avoid the clustering problem that can occur 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 $j \geq 0$, that is, $k \% N$, $(k + 1)\% N$, $(k + 4)\% n$, $(k + 9)\% N$, and so on.

Quadratic Probing


New element with key 26 to be inserted
0 key: 44
1
2
3
4 key: 4
5 key: 16
6 key: 28
7
8
9
10 key: 21
Quadratic probe 2 times before finding an empty cell
For simplicity, only the keys are shown and not the values. Here N is 11 and index = key % N.

Quadratic probing increases the next index in the sequence by **j²** for *j = 1, 2, 3, ...*

Quadratic Probing

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

Problem in Quadratic Probing

Linear probing guarantees that an available cell can be found for insertion as long as the table is not full. However, there is no such guarantee for quadratic probing.

Double Hashing

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

Double hashing uses a secondary hash function *h′ (key)* on the keys to determine the increments to avoid the *clustering problem*.

Specifically, double hashing looks at the cells at indices $(k + j * h'(key)) \% N$, for $j \geq 0$, that is, $k \% N$, $(k + h'(key)) \% N$, $(k + 2 * h'(key)) \% N$, $(k + 3 * h'(key)) \% N$, and so on.

Double Hashing

For example, let the primary hash function *h* and secondary hash function *h'* on a hash table of size **11** be defined as follows:

h(key) = key % **11;**

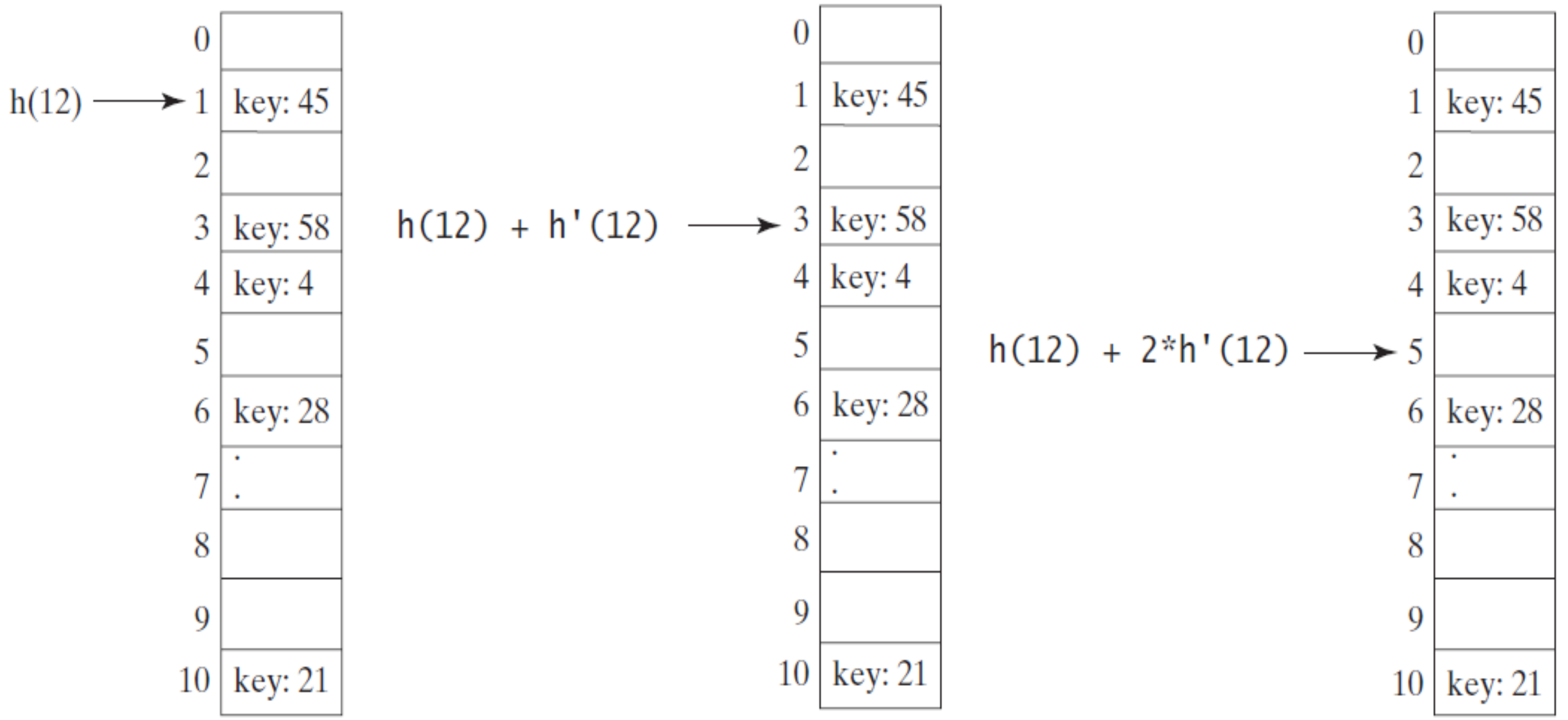
h'(key) = **7 - key % 7;**

For a search key of **12, we have**

***h*(12) = 12 % 11 = 1;**

***h'*(12) = 7 - 12 % 7 = 2;**

Double Hashing



You should design your functions to produce a probe sequence that reaches the entire table. Note that the second function should never have a zero value, since zero is not an increment.

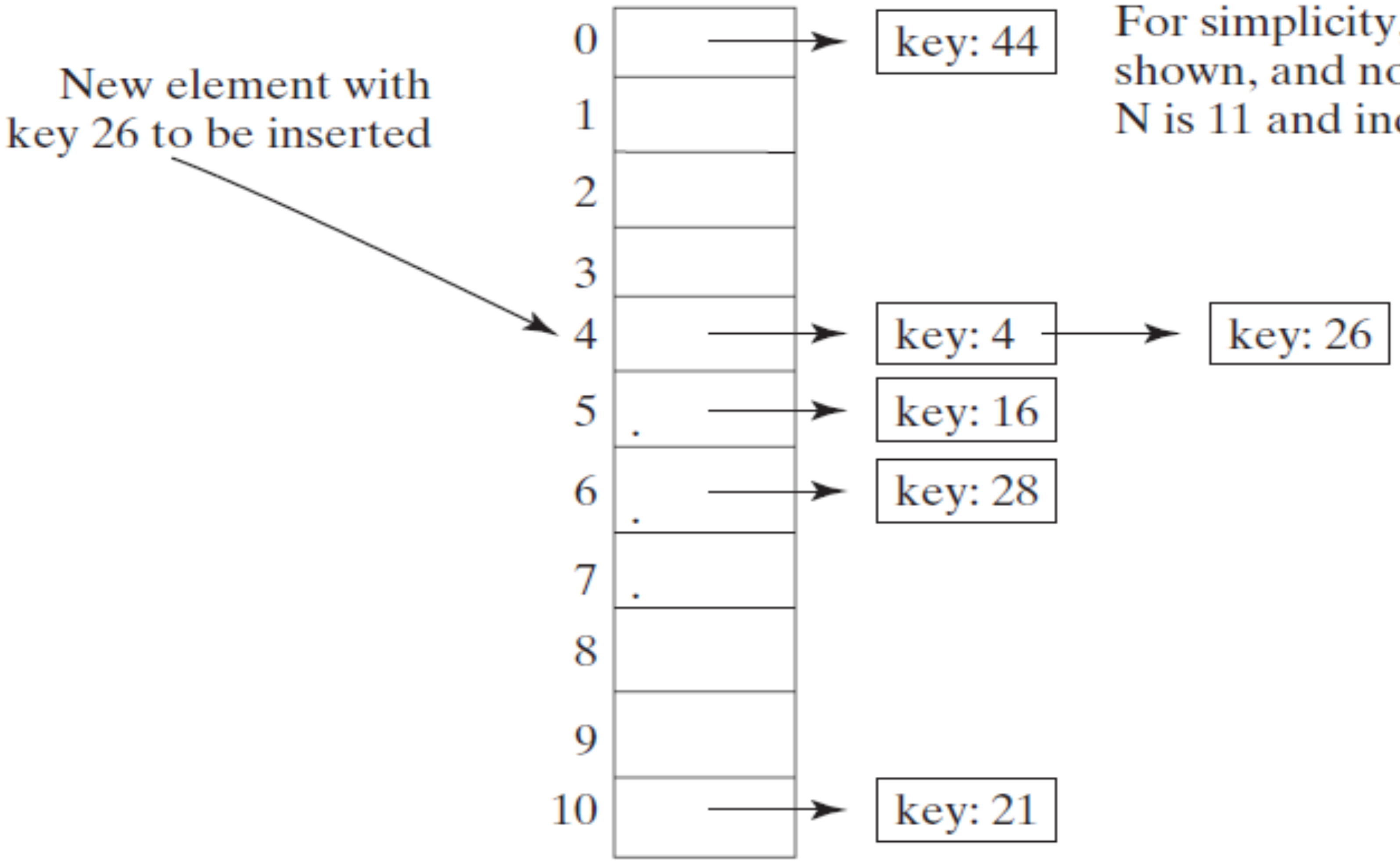
Handling Collisions Using Separate Chaining

The separate chaining scheme places all entries with the same hash index in the same location, rather than finding new locations. Each location in the separate chaining scheme uses a bucket to hold multiple entries.

Handling Collisions Using Separate Chaining

You can implement a bucket using an array, **ArrayList**, or **LinkedList.** We will use **LinkedList** for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head.

Handling Collisions Using Separate Chaining



For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

Load Factor and Rehashing

The *load factor* measures how full a hash table is. If the load factor is exceeded, increase the hash-table size and reload the entries into a new larger hash table. This is called *rehashing*.

Load facto λ *(lambda) measures how full a hash table is. It is the ratio of the number of* elements to the size of the ha $\lambda = \frac{n}{N}$ able, that is, , where *n denotes the number of elements* and *N the number of locations in the hash table*.

Load Factor and Rehashing

$$\lambda = \frac{n}{N}$$

λ is zeroif the hash table is empty

λ is **1** if the hash table is full

As λ increases, the probability of a collision increases

Studies show that you should maintain the load factor under **0.5** for the open addressing scheme and under **0.9** for the separate chaining scheme.

Load Factor and Rehashing

In the implementation of the **java.util.HashMap** class in the Java API, the threshold **0.75** is used. Whenever the load factor exceeds the threshold, you need to increase the **hashtable** size and *rehash* all the entries in the map into a new larger hash table. *Notice that you* need to change the hash functions, since the hash-table size has been changed. To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even with periodic rehashing, hashing is an efficient implementation for map.