[**https://www.geeksforgeeks.org/advanced-data-structures/**](https://www.geeksforgeeks.org/advanced-data-structures/)

**Hashing –** General Idea, Hash Function,

Separate Chaining, Linear Probing, Quadratic Probing, Double Hashing,

Rehashing, Universal Hashing, Extendible Hashing.

**Skip Lists:** Skip list representation, Search and Update Operations on skip lists.

Hashing

The implementation of hash tables is frequently called **hashing**. Hashing is a technique used for performing insertions, deletions, and finds in constant average time. Tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations such as findMin, findMax, and the printing of the entire table in sorted order in linear time are not supported.

**5.1 General Idea**

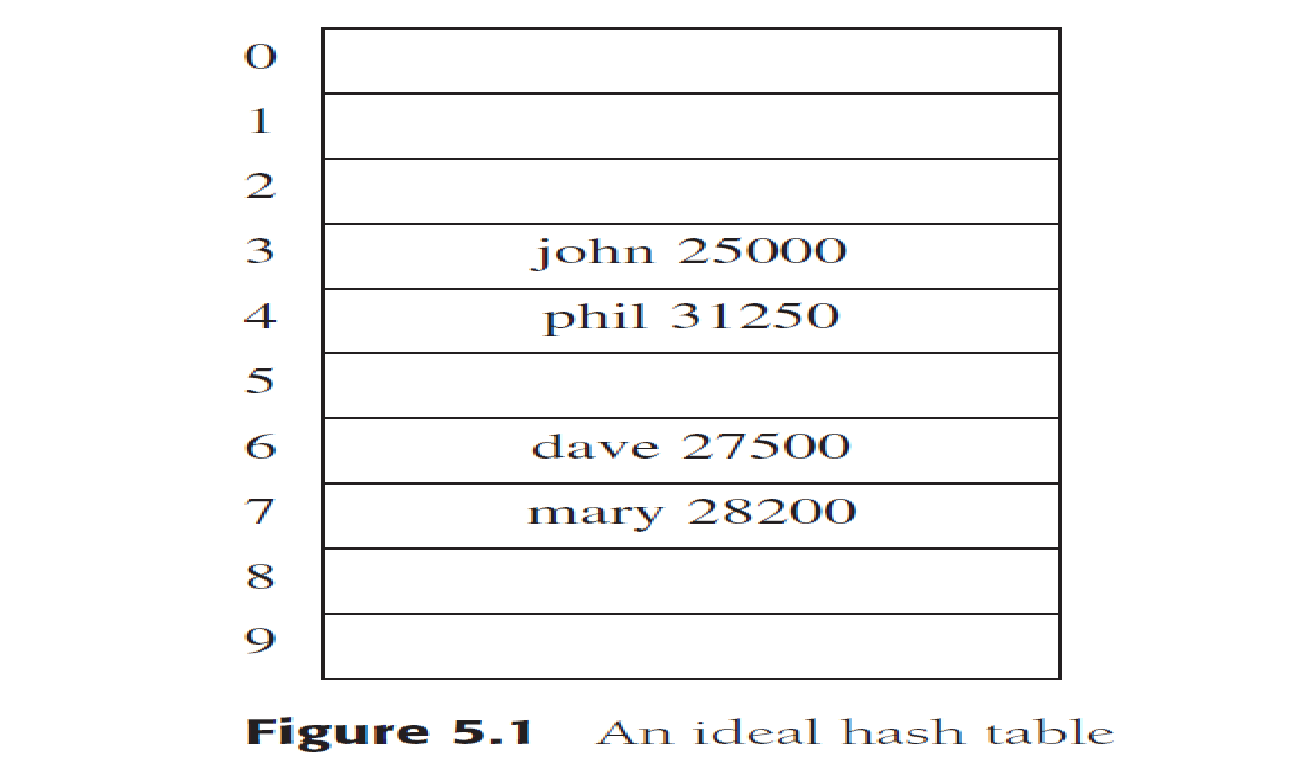
1. The ideal hash table data structure is merely an array of some fixed size containing the

items.

2. Generally a search is performed on some part of the item. This is called the key. For instance, an item could consist of a string (that serves as the key) and additional data members (for instance, a name that is part of a large employee structure).

3. We will refer to the table size as TableSize, The common convention is to have the table run from 0 to TableSize − 1;

.

. 

4. Each key is mapped into some number in the range 0 to TableSize − 1 and placed in the appropriate cell.

5. The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells.

6. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a hash function that distributes the keys evenly among the cells.

7. Figure 5.1 is typical of a perfect situation. In this example, john hashes to 3, phil hashes to 4, dave hashes to 6, and mary hashes to 7.

**5.2 Hash Function**

1. If the input keys are integers, then simply returning Key mod TableSize is generally a reasonable strategy, unless Key happens to have some undesirable properties.
2. In this case, the choice of hash function needs to be carefully considered.

For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is a bad choice.

1. it is often a good idea to ensure that the table size is prime. When the input keys are random integers, then this function is not only very simple to compute but also distributes the keys evenly.
2. Usually, the keys are strings; in this case, the hash function needs to be chosen carefully.
3. One option is to add up the ASCII values of the characters in the string.
4. The routine in Figure 5.2 implements this strategy.

The hash function depicted in Figure 5.2 is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute the keys well.

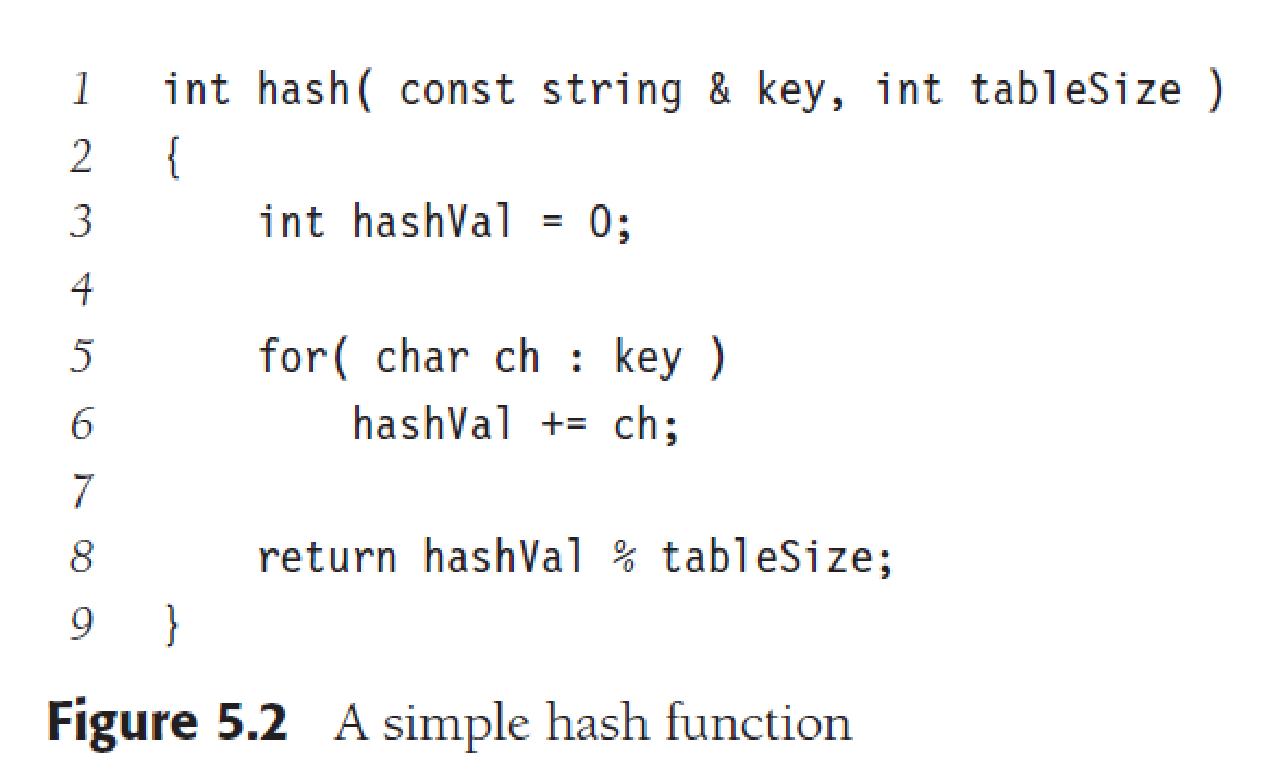
* For instance, suppose that TableSize = 10,007 (10,007 is a prime number). Suppose all the keys are eight or fewer characters long. Since an ASCII character has an integer value that is always at most 127, the hash function typically can only assume values between 0 and 1,016, which is 127 ∗ 8. This is clearly not an equitable distribution!
* Another hash function is shown in Figure 5.3. This hash function assumes that Key has

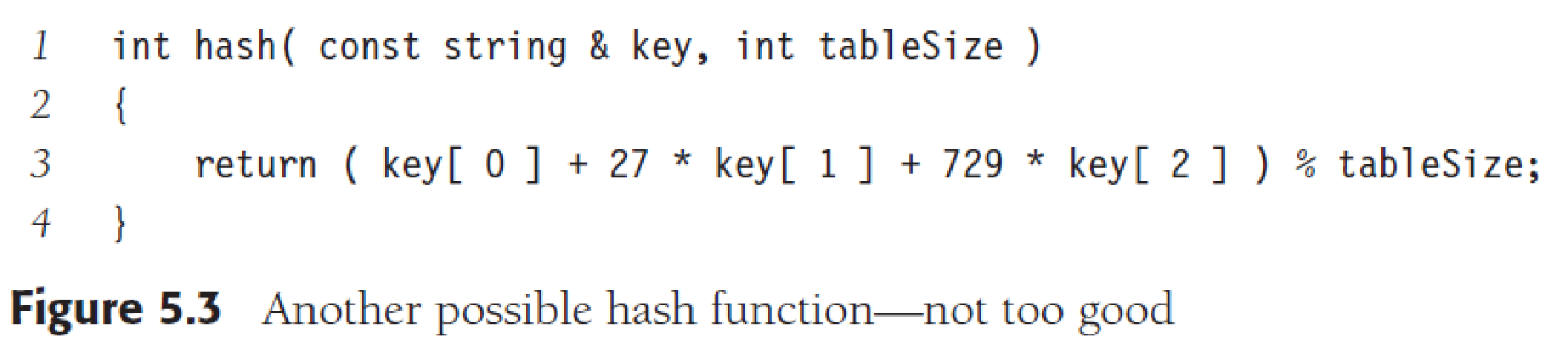
at least three characters. The value 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 272. This function examines only the first three characters,but if these are random and the table size is 10,007, as before, then we would expect a reasonably equitable distribution. Unfortunately, English is not random. Although there are 263 = 17,576 possible combinations of three characters (ignoring blanks), a check of a reasonably large online dictionary reveals that the number of different combinations is actually only 2,851. Even if none of these combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.

* Figure 5.4 shows a third attempt at a hash function. This hash function involves

all characters in the key and can generally be expected to distribute well (it computes  · and brings the result into proper range). The code computes a polynomial function (of 37) by use of Horner’s rule. For instance, another way of computing hk = k0 + 37k1 + 372k2 is by the formula hk = ((k2) ∗ 37 + k1) ∗ 37 + k0 Horner

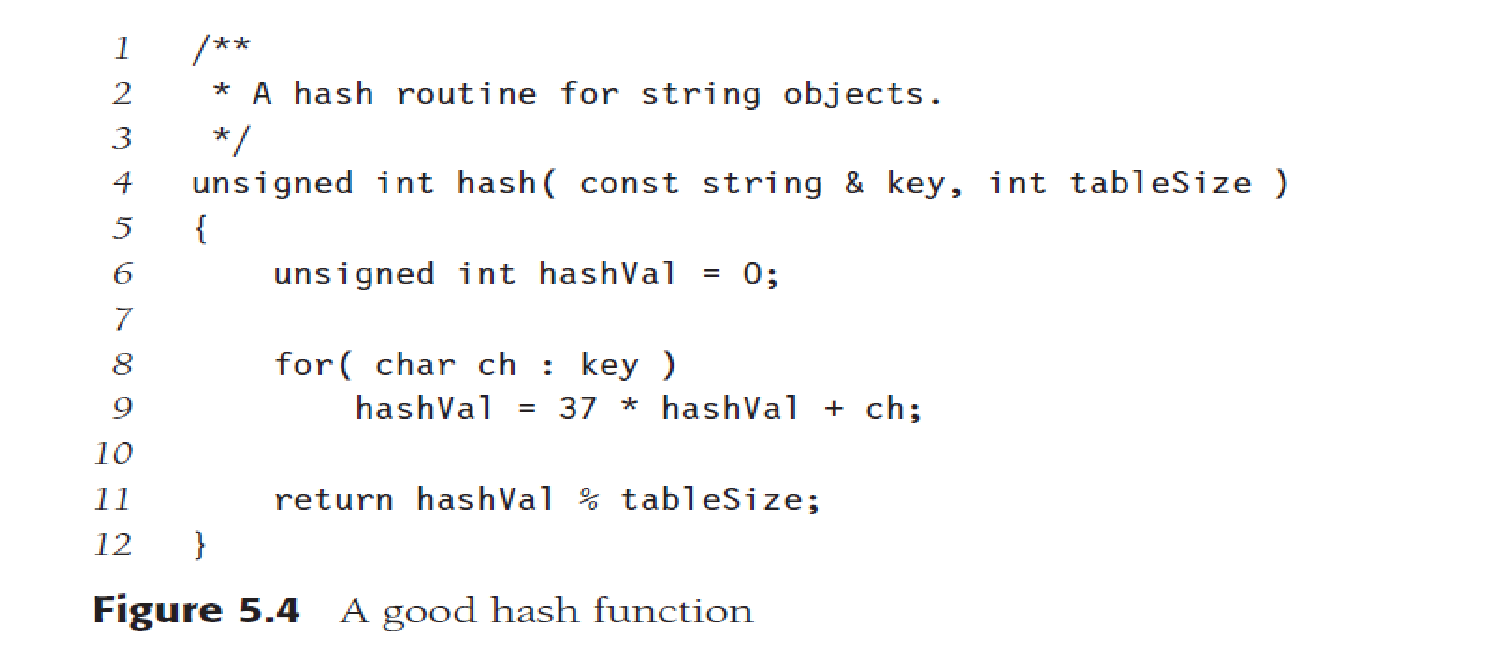
’s rule extends this to an nth degree polynomial. reasonably equitable distribution.





* Unfortunately, English is not random. Although there are 263 = 17,576 possible combinations of three characters (ignoring blanks), a check of a reasonably large online dictionary reveals that the number of different combinations is actually only 2,851.
* Even if none of these combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.
* Figure 5.4 shows a third attempt at a hash function. This hash function involves a\_ll characters in the key and can generally be expected to distribute well (it computes KeySize−1. i=0 Key[KeySize − i − 1] · 37i and brings the result into proper range). The code computes a polynomial function (of 37) by use of Horner’s rule. For instance, another way of computing hk = k0 + 37k1 + 372k2 is by the formula hk = ((k2) ∗ 37 + k1) ∗ 37 + k0. Horner’s rule extends this to an nth degree polynomial.
* The hash function takes advantage of the fact that overflow is allowed and uses unsigned int to avoid introducing a negative number.
* The hash function described in Figure 5.4 is not necessarily the best with respect to

table distribution, but it does have the merit of extreme simplicity and is reasonably fast. If the keys are very long, the hash function will take too long to compute. A common practice in this case is not to use all the characters. The length and properties of the keys would then influence the choice. For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code. Some programmers implement their hash function by using only the characters in the odd spaces, with the idea that the time saved computing the hash function will make up for a slightly less evenly distributed function.



* If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it. There are several methods for dealing with this. Two of them are : separate chaining and open addressing;

**5.3 Separate Chaining**

The first strategy, commonly known as **separate chaining,** is to keep a list of all elements

that hash to the same value. We can use the Standard Library list implementation. If space

is tight, it might be preferable to avoid their use (since these lists are doubly linked and

waste space). We assume for this section that the keys are the first 10 perfect squares and

that the hashing function is simply *hash*(*x*) = *x* mod 10. (The table size is not prime but

is used here for simplicity.) Figure 5.5 shows the resulting separate chaining hash table.

To perform a search, we use the hash function to determine which list to traverse. We

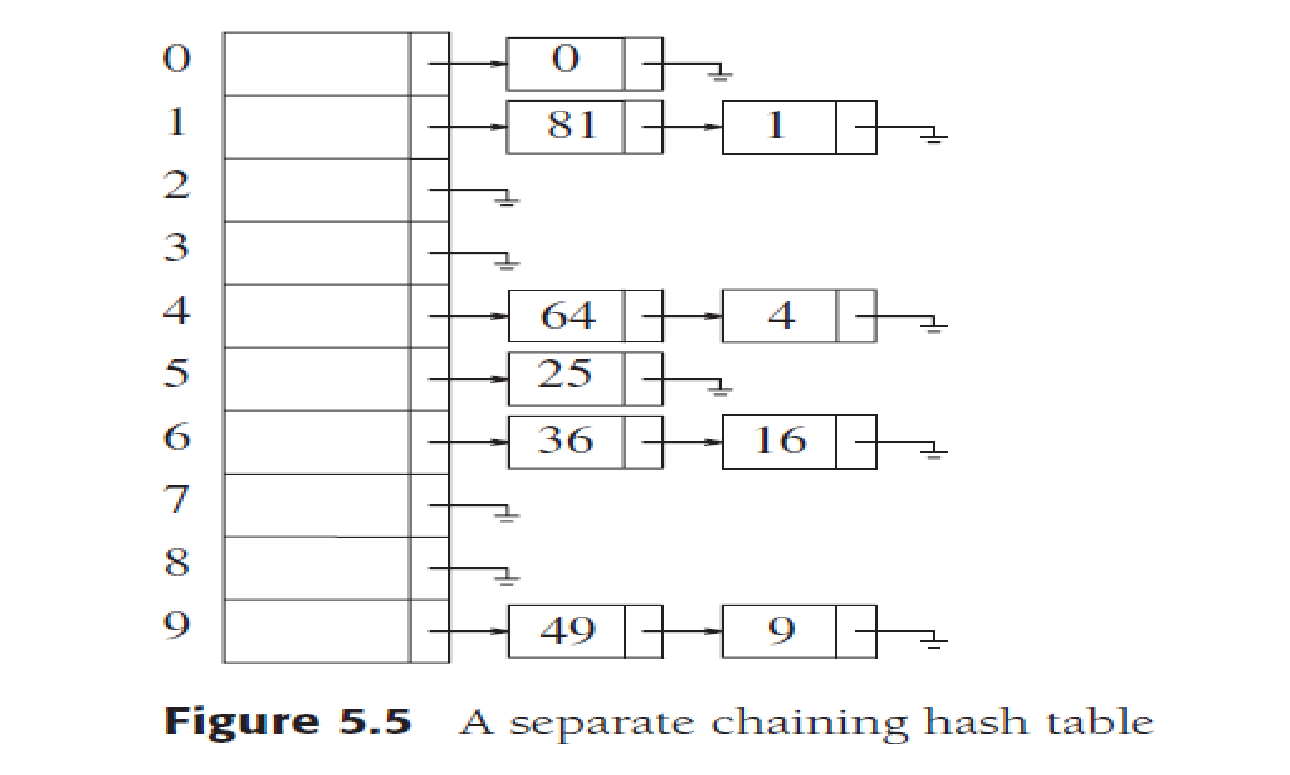
then search the appropriate list. To perform an insert, we check the appropriate list to see

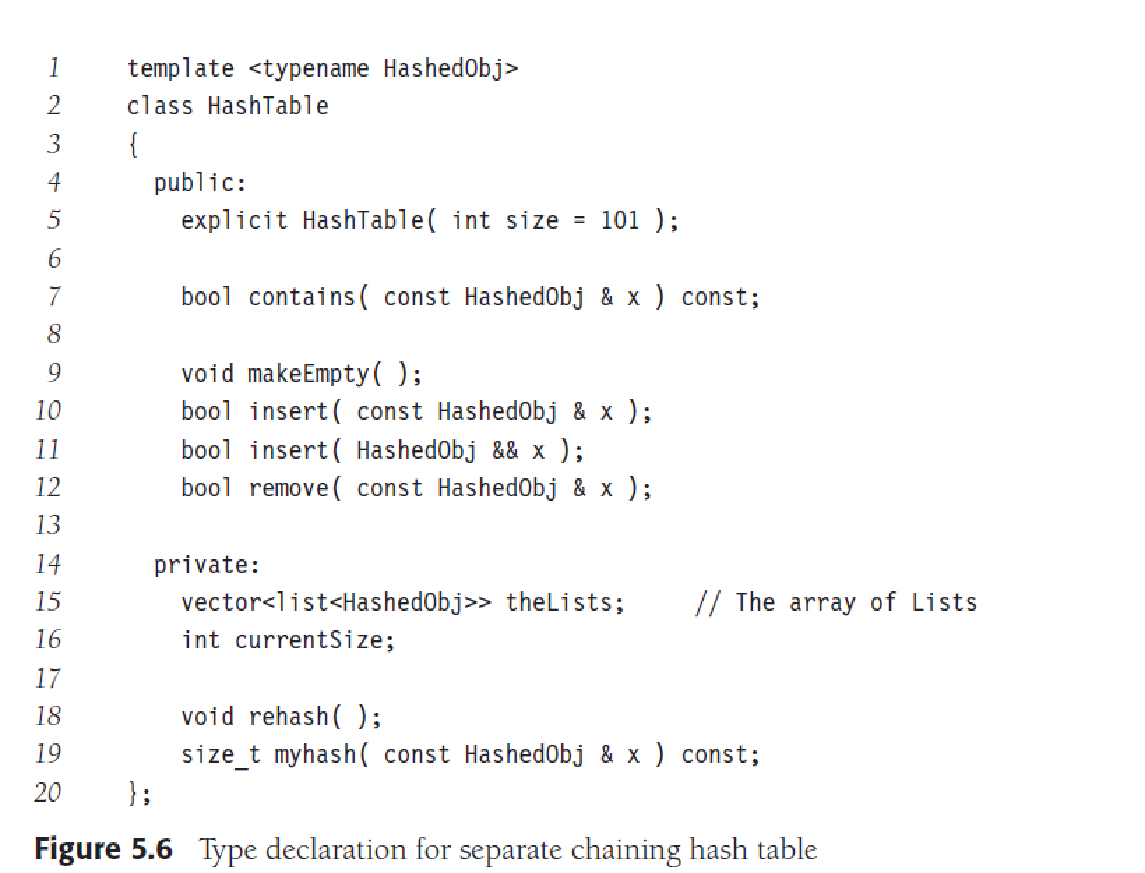
whether the element is already in place (if duplicates are expected, an extra data member isusually kept, and this data member would be incremented in the event of a match). If the

element turns out to be new, it can be inserted at the front of the list, since it is convenient

and also because frequently it happens that recently inserted elements are the most likely

to be accessed in the near future.





The class interface for a separate chaining implementation is shown in Figure 5.6. The

hash table stores an array of linked lists, which are allocated in the constructor.

The class interface illustrates a syntax point: Prior to C++11, in the declaration of

theLists, a space was required between the two >s; since >> is a C++ token, and because it

is longer than >, >> would be recognized as the token. In C++11, this is no longer the case.

Just as the binary search tree works only for objects that are Comparable, the hash tables

in this chapter work only for objects that provide a hash function and equality operators

(operator== or operator!=, or possibly both).

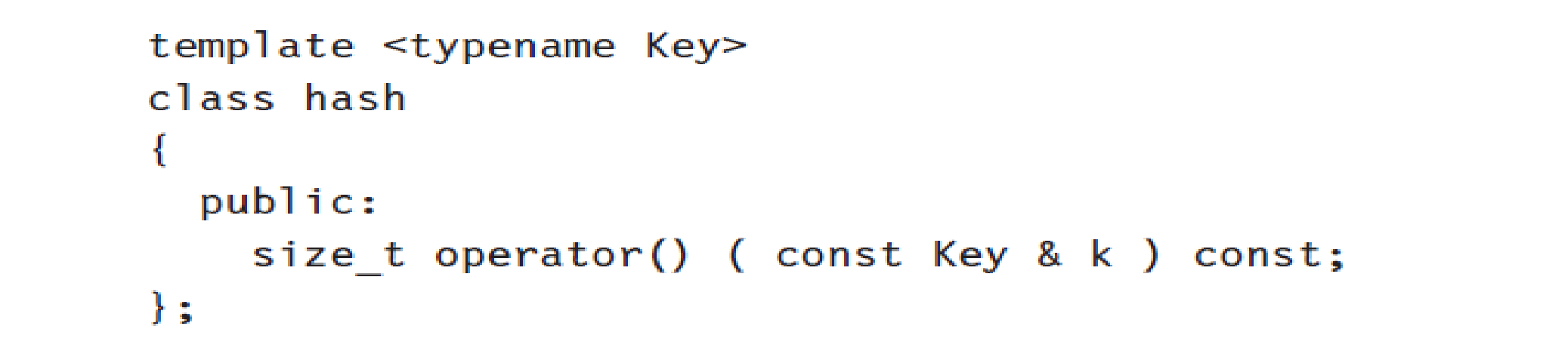
Instead of requiring hash functions that take both the object and the table size as

parameters, we have our hash functions take only the object as the parameter and return

an appropriate integral type. The standard mechanism for doing this uses function objects,

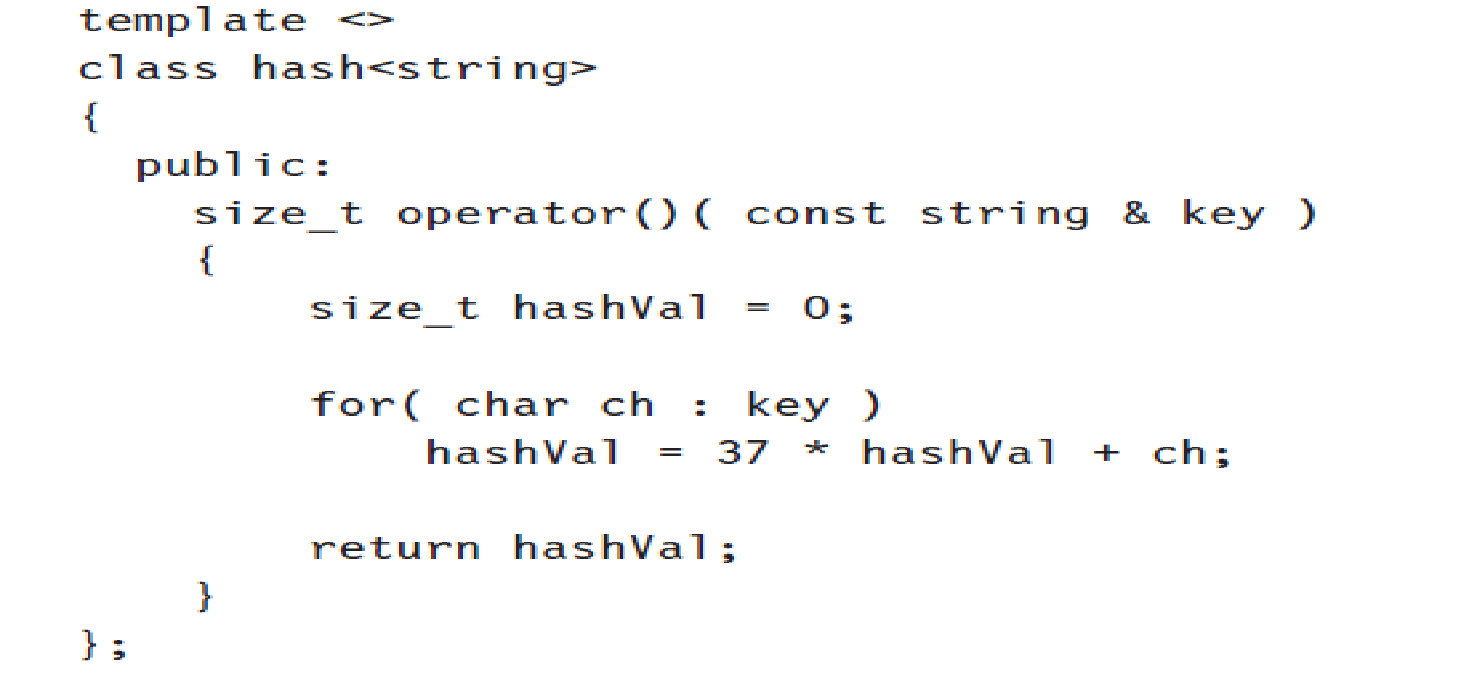
and the protocol for hash tables was introduced in C++. pecifically, in C++11, hash

functions can be expressed by the function object template:



Default implementations of this template are provided for standard types such as int

and string; thus, the hash function described in Figure 5.4 could be implemented as



The type size\_t is an unsigned integral type that represents the size of an object;

therefore, it is guaranteed to be able to store an array index. A class that implements a

hash table algorithm can then use calls to the generic hash function object to generate an

integral type size\_t and then scale the result into a suitable array index.

In our hash tables,

this is manifested in private member function myhash, shown in Figure 5.7.

Figure 5.8 illustrates an Employee class that can be stored in the generic hash

table, using the name member as the key. The Employee class implements the HashedObj

requirements by providing equality operators and a hash function object.

The code to implement makeEmpty, contains, and remove is shown in Figure 5.9.

Next comes the insertion routine. If the item to be inserted is already present, then we

do nothing; otherwise, we place it in the list (see Fig. 5.10). The element can be placed

anywhere in the list; using push\_back is most convenient in our case. whichList is a reference

variable; see Section 1.5.2 for a discussion of this use of reference variables.

Any scheme could be used besides linked lists to resolve the collisions; a binary search

tree or even another hash table would work, but we expect that if the table is large and the

hash function is good, all the lists should be short, so basic separate chaining makes no

attempt to try anything complicated.

We define the load factor, λ, of a hash table to be the ratio of the number of elements

in the hash table to the table size. In the example above, λ = 1.0. The average length of a

list is λ. The effort required to perform a search is the constant time required to evaluate

the hash function plus the time to traverse the list. In an unsuccessful search, the number

of nodes to examine is λ on average. A successful search requires that about 1 + (λ/2)

links be traversed. To see this, notice that the list that is being searched contains the one

node that stores the match plus zero or more other nodes. The expected number of “other

nodes” in a table of N elements and M lists is (N−1)/M = λ−1/M, which is essentially λ,

since M is presumed large. On average, half the “other nodes” are searched, so combined

with the matching node, we obtain an average search cost of 1 + λ/2 nodes. This analysis

shows that the table size is not really important but the load factor is. The general rule

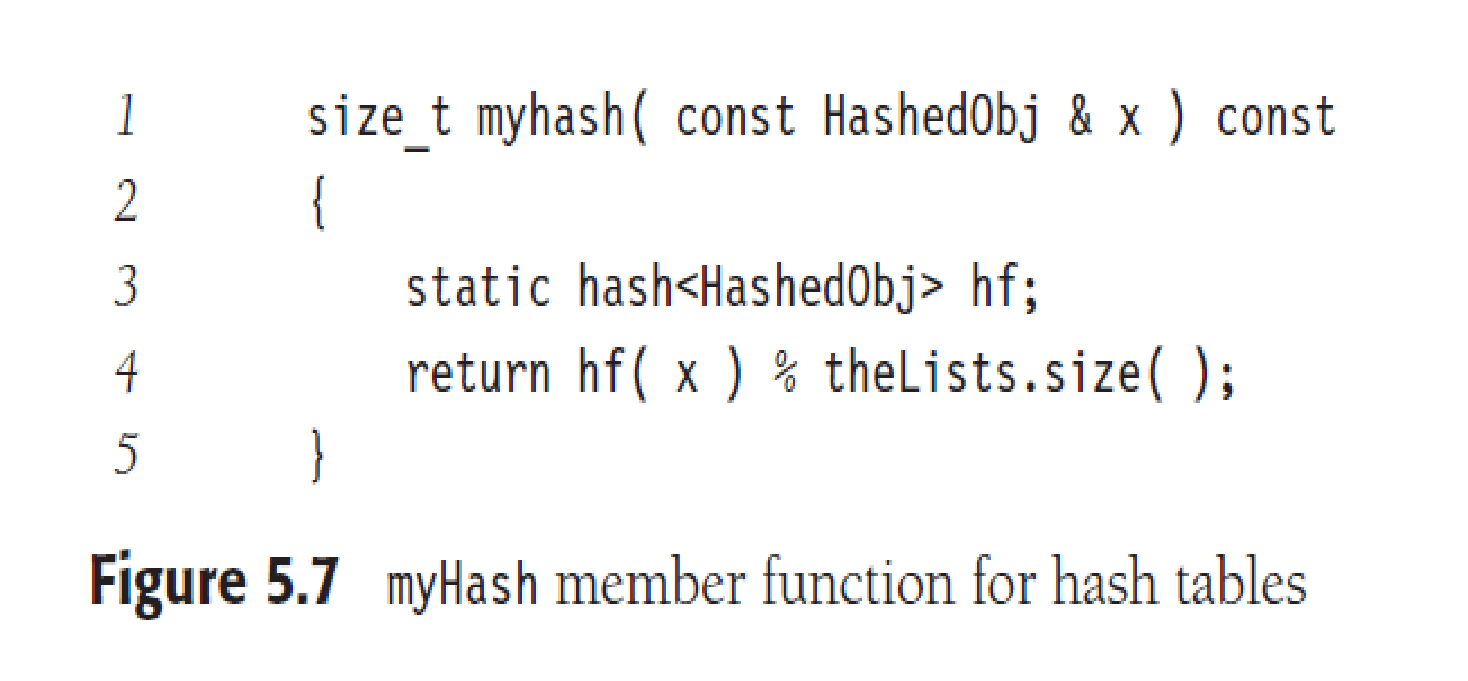
for separate chaining hashing is to make the table size about as large as the number of

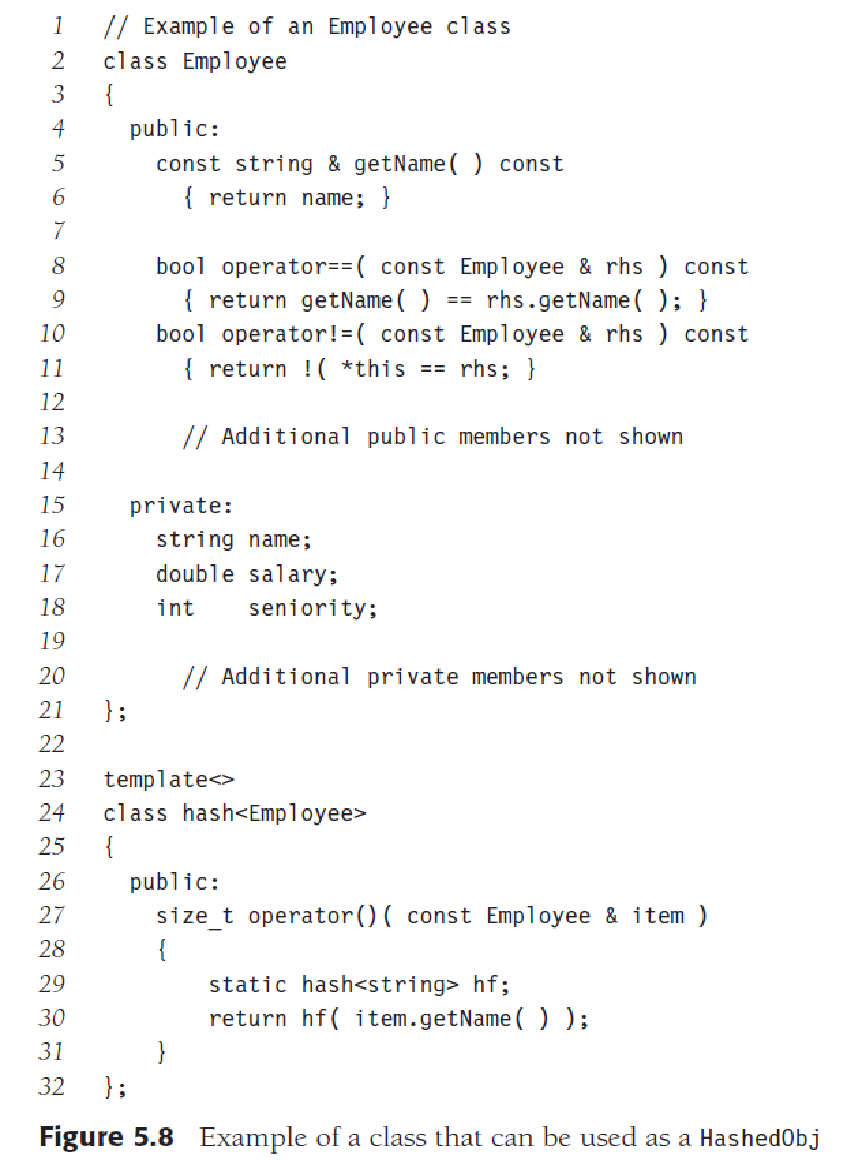
elements expected (in other words, let λ ≈ 1). In the code in Figure 5.10, if the load

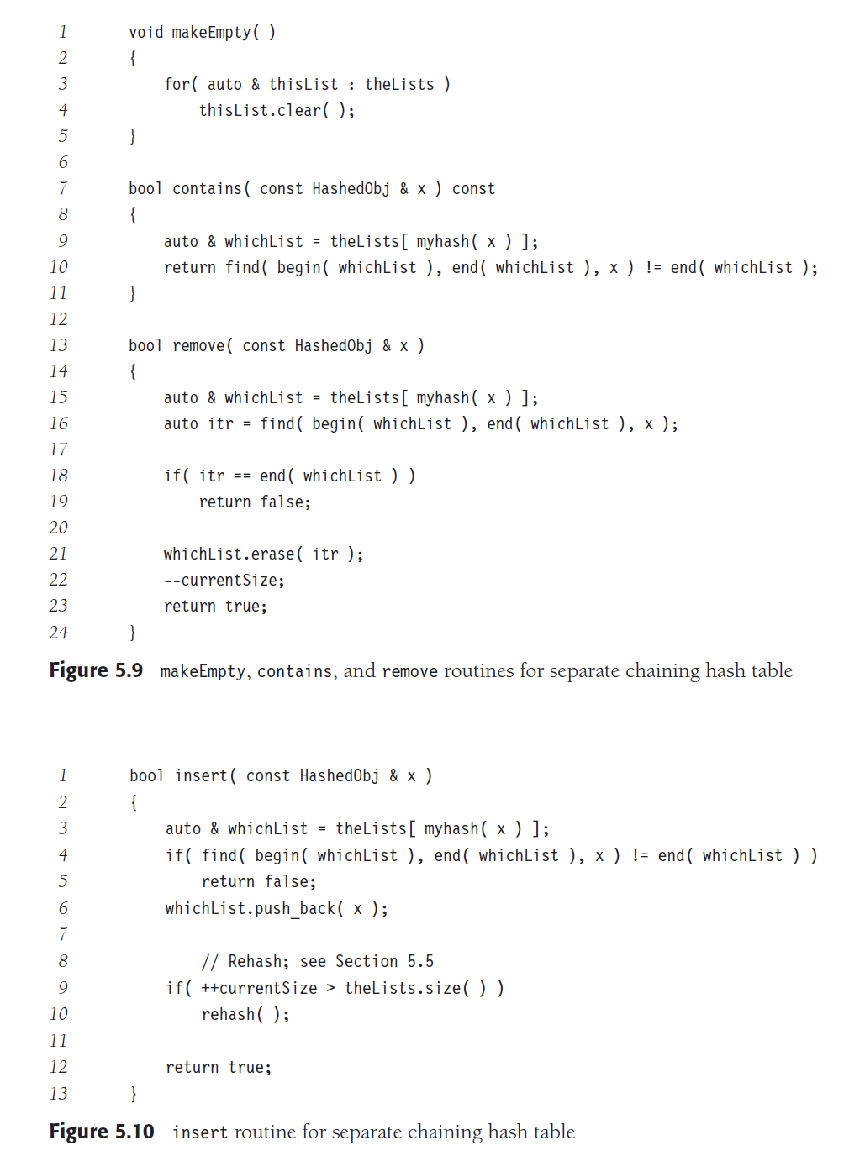
factor exceeds 1, we expand the table size by calling rehash at line 10. rehash is discussed

in Section 5.5. It is also a good idea, as mentioned before, to keep the table size prime to

ensure a good distribution.







| Chaining (Open Hashing) | Linear Probing (Open Addressing) |
| --- | --- |
| Key values can be stored outside of the table using a separate linked list. | Key values should be stored inside the table only. |
| The number of elements in the hash table may exceed the size of the hash table. | The number of elements present in the hash table will not exceed the number of indices in the hash table. |
| Deletion is efficient in chaining technique. | Deletion can be cumbersome. Can be avoided if not required. |
| Since a separate linked list is maintained for each location, the space taken is large. | Since all entries are accommodated in the same table, space taken is lesser. |

| S.No. | Separate Chaining | Open Addressing |
| --- | --- | --- |
| 1. | Chaining is Simpler to implement. | Open Addressing requires more computation. |
| 2. | In chaining, Hash table never fills up, we can always add more elements to chain. | In open addressing, table may become full. |
| 3. | Chaining is Less sensitive to the hash function or load factors. | Open addressing requires extra care to avoid clustering and load factor. |
| 4. | Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted. | Open addressing is used when the frequency and number of keys is known. |
| 5. | Cache performance of chaining is not good as keys are stored using linked list. | Open addressing provides better cache performance as everything is stored in the same table. |
| 6. | Wastage of Space (Some Parts of hash table in chaining are never used). | In Open addressing, a slot can be used even if an input doesn’t map to it. |
| 7. | Chaining uses extra space for links. | No links in Open addressing |

**5.4 Hash Tables without Linked Lists**

Separate chaining hashing has the disadvantage of using linked lists. This could slow

the algorithm down a bit because of the time required to allocate new cells (especially

in other languages) and essentially requires the implementation of a second data structure.

An alternative to resolving collisions with linked lists is to try alternative cells until

an empty cell is found. More formally, cells *h*0(*x*), *h*1(*x*), *h*2(*x*), *. . .* are tried in succession,

where *hi*(*x*) = (*hash*(*x*) + *f* (*i*)) mod *TableSize*, with *f*(0) = 0. The function, *f*, is the collision

resolution strategy. Because all the data go inside the table, a bigger table is needed

in such a scheme than for separate chaining hashing. Generally, the load factor should be

below *λ* = 0.5 for a hash table that doesn’t use separate chaining. We call such tables

**probing hash tables**. We now look at three common collision resolution strategies.

5.4.1 Linear Probing

In linear probing, *f* is a linear function of *i*, typically *f* (*i*) = *i*. This amounts to trying cells

sequentially (with wraparound) in search of an empty cell. Figure 5.11 shows the result of

inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before

and the collision resolution strategy, *f* (*i*) = *i*.

The first collision occurs when 49 is inserted; it is put in the next available spot, namely,

spot 0, which is open. The key 58 collides with 18, 89, and then 49 before an empty cell

is found three away. The collision for 69 is handled in a similar manner. As long as the

table is big enough, a free cell can always be found, but the time to do so can get quite

large. Worse, even if the table is relatively empty, blocks of occupied cells start forming.

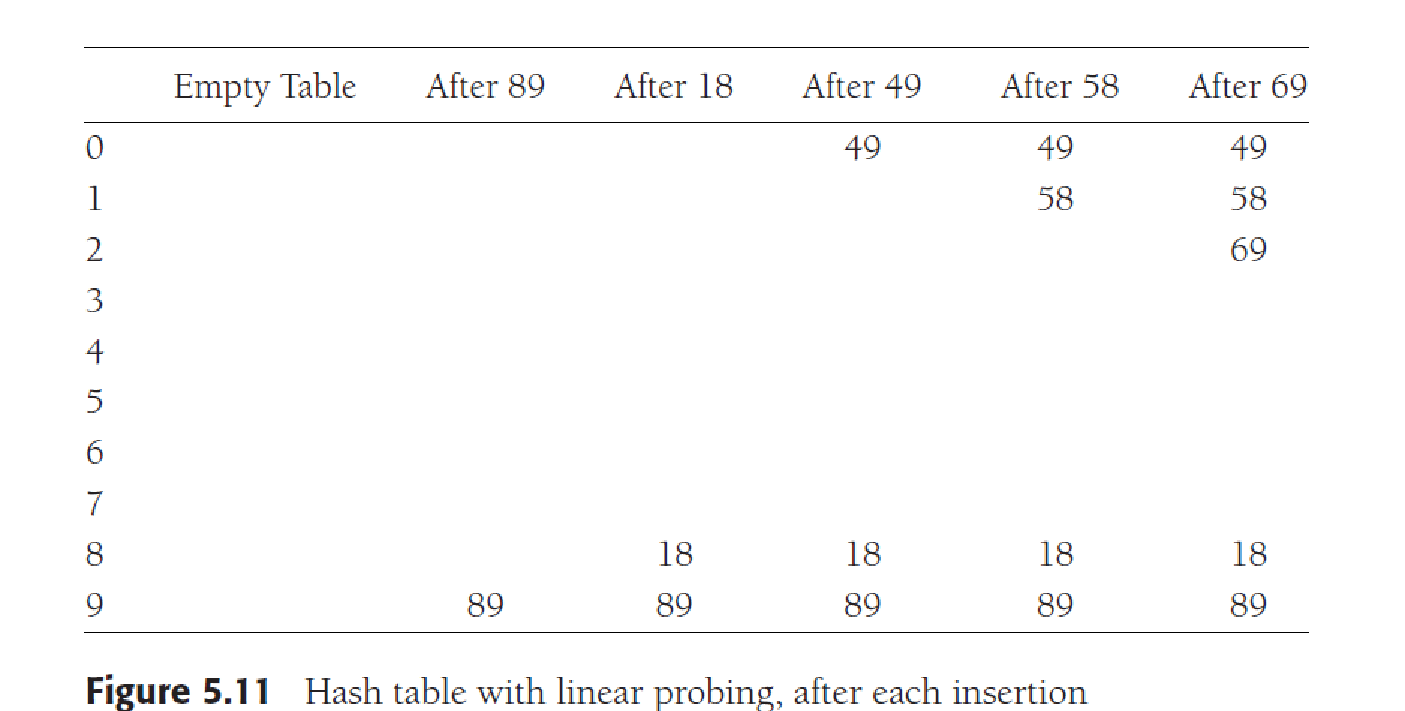
This effect, known as **primary clustering,** means that any key that hashes into the cluster

will require several attempts to resolve the collision, and then it will add to the cluster.

Although we will not perform the calculations here, it can be shown that the expected

number of probes using linear probing is roughly 12

(1 + 1*/*(1 − *λ*)2) for insertions and



unsuccessful searches, and 12

(1 + 1/(1 − λ)) for successful searches. The calculations

are somewhat involved. It is easy to see from the code that insertions and unsuccessful

searches require the same number of probes. A moment’s thought suggests that, on average,

successful searches should take less time than unsuccessful searches.

The corresponding formulas, if clustering is not a problem, are fairly easy to derive.

We will assume a very large table and that each probe is independent of the previous

probes. These assumptions are satisfied by a random collision resolution strategy and are

reasonable unless λ is very close to 1. First, we derive the expected number of probes in

an unsuccessful search. This is just the expected number of probes until we find an empty

cell. Since the fraction of empty cells is 1 − λ, the number of cells we expect to probe is

1/(1 − λ). The number of probes for a successful search is equal to the number of probes

required when the particular element was inserted. When an element is inserted, it is done

as a result of an unsuccessful search. Thus, we can use the cost of an unsuccessful search

to compute the average cost of a successful search.

The caveat is that λ changes from 0 to its current value, so that earlier insertions are

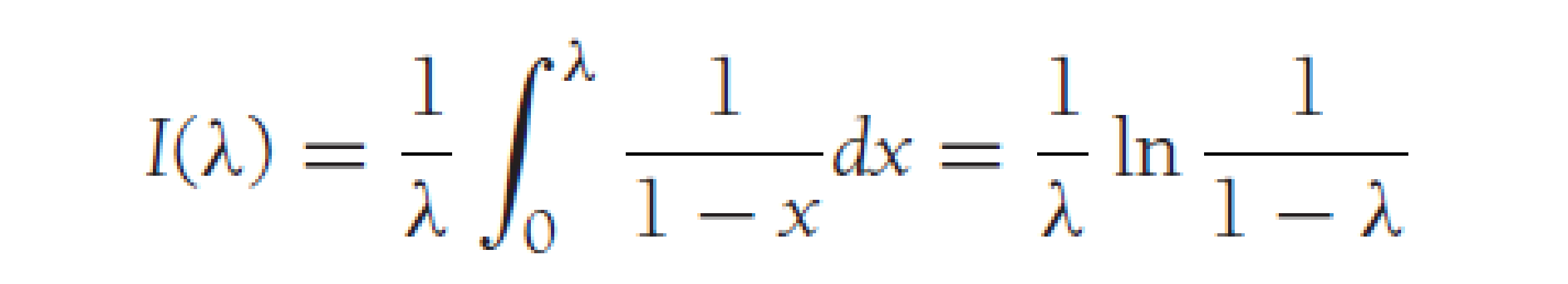
cheaper and should bring the average down. For instance, in the table in Figure 5.11,

λ = 0.5, but the cost of accessing 18 is determined when 18 is inserted. At that point,

λ = 0.2. Since 18 was inserted into a relatively empty table, accessing it should be easier

than accessing a recently inserted element, such as 69. We can estimate the average by

using an integral to calculate the mean value of the insertion time, obtaining



These formulas are clearly better than the corresponding formulas for linear probing.

Clustering is not only a theoretical problem but actually occurs in real implementations.

Figure 5.12 compares the performance of linear probing (dashed curves) with what

would be expected from more random collision resolution. Successful searches are indicated

by an S, and unsuccessful searches and insertions are marked with U and I,

respectively.

If λ = 0.75, then the formula above indicates that 8.5 probes are expected for an

insertion in linear probing. If λ = 0.9, then 50 probes are expected, which is unreasonable.

This compares with 4 and 10 probes for the respective load factors if clustering were not

a problem. We see from these formulas that linear probing can be a bad idea if the table is

expected to be more than half full. If λ = 0.5, however, only 2.5 probes are required on

average for insertion, and only 1.5 probes are required, on average, for a successful search.

**Challenges in Linear Probing :**

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering*:***Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

5.4.2 Quadratic Probing

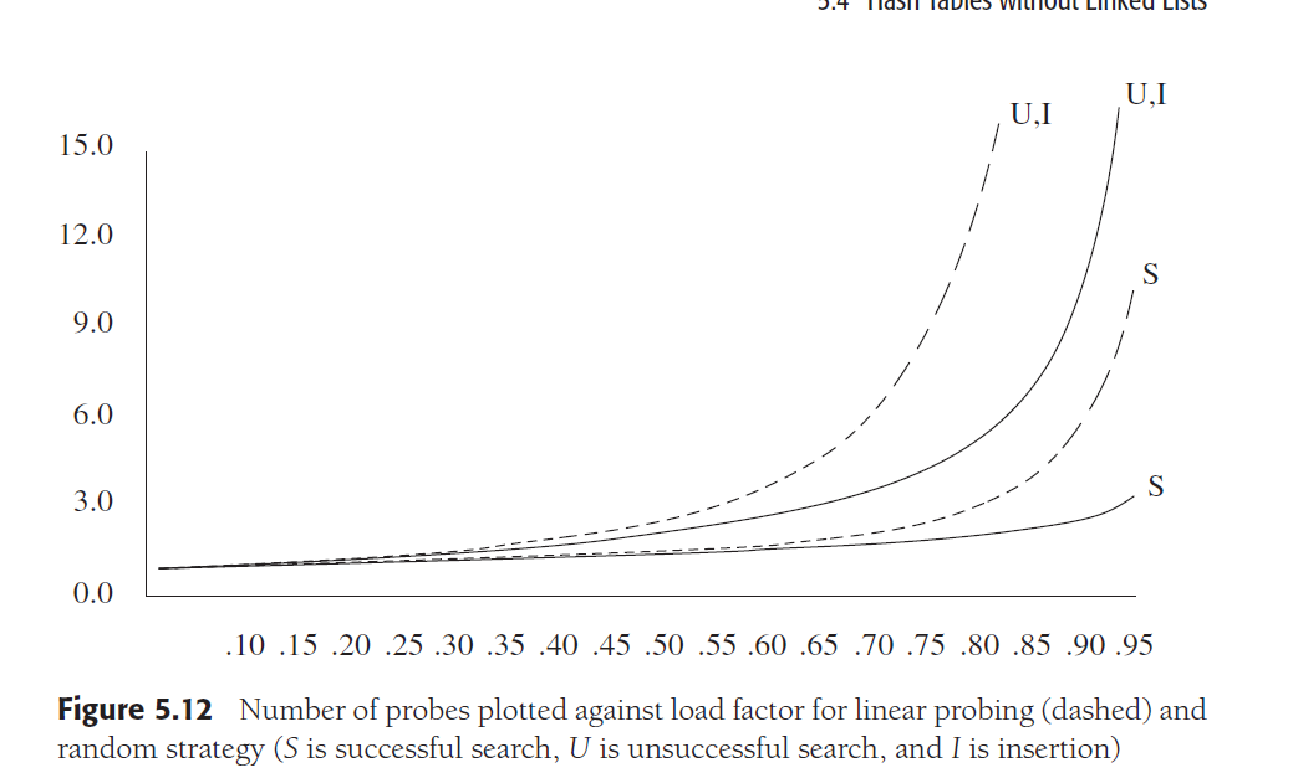
Quadratic probing is a collision resolution method that eliminates the primary clustering

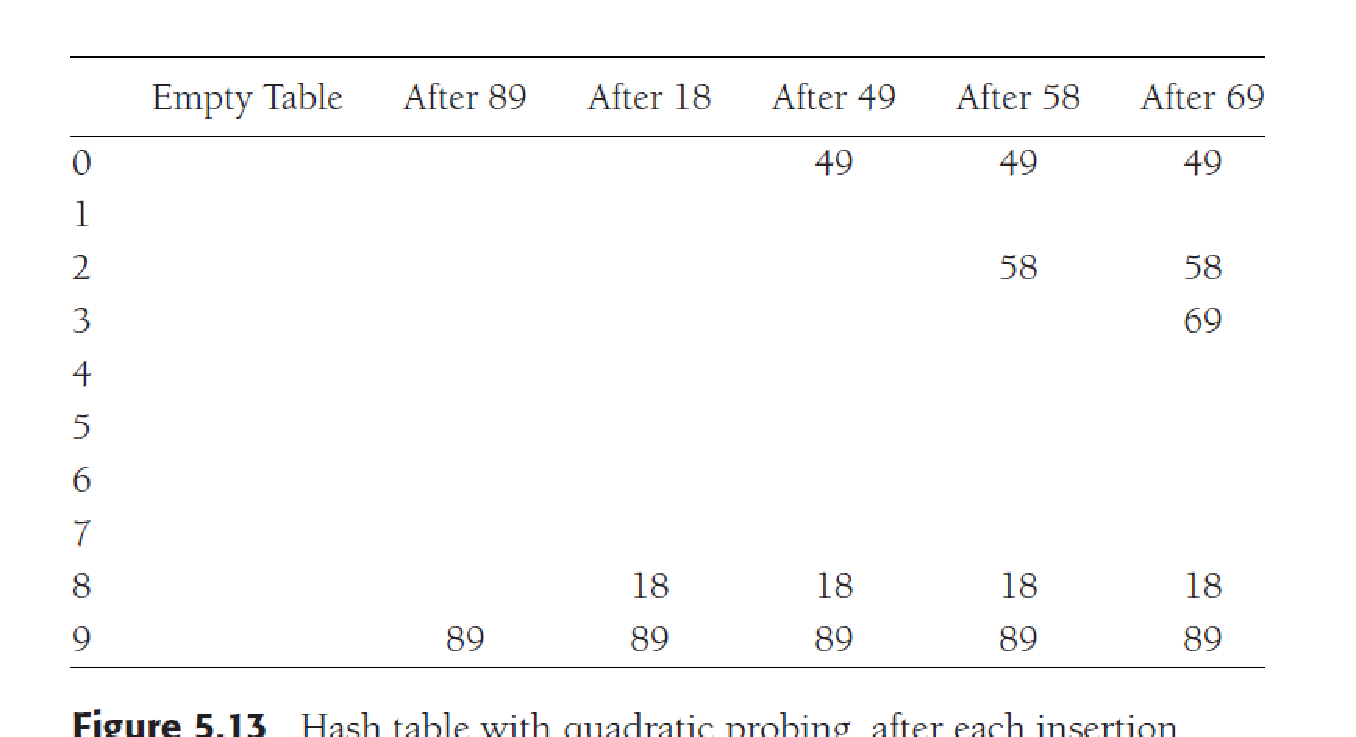
problem of linear probing. Quadratic probing is what you would expect—the collision

function is quadratic. The popular choice is f (i) = i2. Figure 5.13 shows the resulting hash

table with this collision function on the same input used in the linear probing example.

When 49 collides with 89, the next position attempted is one cell away. This cell is





tried, but another collision occurs. A vacant cell is found at the next cell tried, which is

22 = 4 away. 58 is thus placed in cell 2. The same thing happens for 69.

For linear probing, it is a bad idea to let the hash table get nearly full, because performance

degrades. For quadratic probing, the situation is even more drastic: There is no

guarantee of finding an empty cell once the table gets more than half full, or even before

the table gets half full if the table size is not prime. This is because at most half of the table

can be used as alternative locations to resolve collisions.

Indeed, we prove now that if the table is half empty and the table size is prime, then

we are always guaranteed to be able to insert a new element.

**Extra material for your reference**

**Comparison of above three:**

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.   
Quadratic probing lies between the two in terms of cache performance and clustering.   
Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

**Quadratic Probing and Double Hashing**

Quadratic Probing and Double Hashing attempt to find ways to reduce the size of the clusters that are formed by linear probing.

**Quadratic Probing**

Quadratic Probing is similar to Linear probing. The difference is that if you were to try to insert into a space that is filled you would first check 1^2 ==1 element away then 2^2 = 4 elements away, then 3^2 =​​=9 elements away then 4^2=16 elements away and so on.

With linear probing we know that we will always find an open spot if one exists (It might be a long search but we will find it). However, this is not the case with quadratic probing unless you take care in the choosing of the table size. For example consider what would happen in the following situation:

Table size is 16. First 5 pieces of data that all hash to index 2

First piece goes to index 2.

Second piece goes to 3 ((2 + 1)%16

Third piece goes to 6 ((2+4)%16

Fourth piece goes to 11((2+9)%16

Fifth piece dosen't get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven't searched all empty spots.

In order to guarantee that your quadratic probes will hit every single available spots eventually, your table size must meet these requirements:

Be a prime number

never be more than half full (even by one element)

**Double Hashing**

Double Hashing is works on a similar idea to linear and quadratic probing. Use a big table and hash into it. Whenever a collision occurs, choose another spot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. For example, given hash function H1 and H2 and key. do the following:

Check location hash1(key). If it is empty, put record in it.

If it is not empty calculate hash2(key).

check if hash1(key)+hash2(key) is open, if it is, put it in

repeat with hash1(key)+2hash2(key), hash1(key)+3hash2(key) and so on, until an opening is found.

like quadratic probing, you must take care in choosing hash2. hash2 CANNOT ever return 0. hash2 must be done so that all cells will be probed eventually.

Given the two hash functions, h1​**(k) = k mod 23** and h2​**(k) = 1 + k mod 19**. Assume the table size is 23. Find the address returned by double hashing after 2nd collision for the key = 90.

**h(k,i) = (**h1​**(k) + i \***h2​**(k) )%m**

**problems on linear probing , quadratic probing and double hashing**

In this section we will see what is linear probing technique in open addressing scheme. There is an ordinary hash function h´(x) : U → {0, 1, . . ., m – 1}. In open addressing scheme, the actual hash function h(x) is taking the ordinary hash function h’(x) and attach some another part with it to make one linear equation.

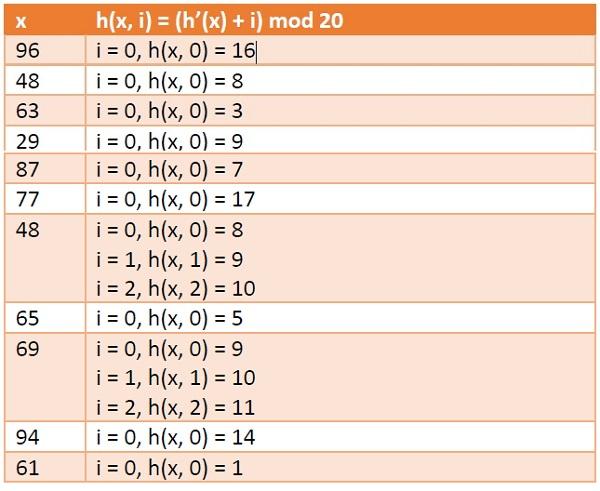
h´(𝑥) = 𝑥 𝑚𝑜𝑑 𝑚

ℎ(𝑥, 𝑖) = (ℎ´(𝑥) + 𝑖)𝑚𝑜𝑑 𝑚

The value of i| = 0, 1, . . ., m – 1. So we start from i = 0, and increase this until we get one freespace. So initially when i = 0, then the h(x, i) is same as h´(x).

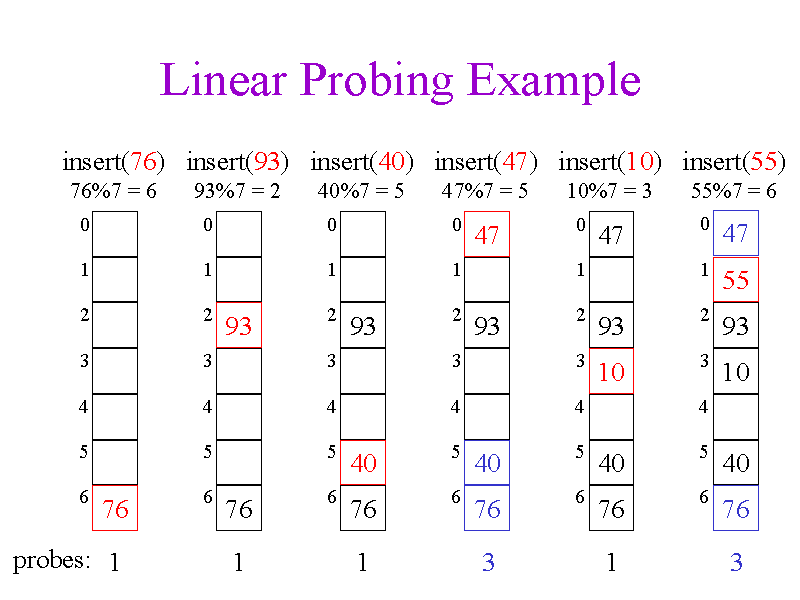
## Example

Suppose we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}



**Hash Table**

https://www.tutorialspoint.com/assets/questions/media/41070/linear_probing1.jpg



# Avoid collision using linear probing

#### Collision

While hashing, two or more key points to the same hash index under some modulo M is called as collision.

In this tutorial, we will learn how to avoid collison using linear probing technique.

## Linear Probing

Calculate the hash key. key = data % size;

If hashTable[key] is empty, store the value directly. hashTable[key] = data.

If the hash index already has some value, check for next index.

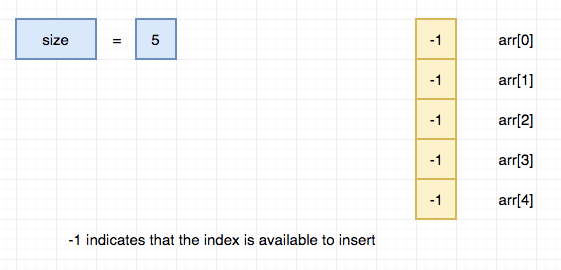
**key = (key+1) % size;**

If the next index is available hashTable[key], store the value. Otherwise try for next index.

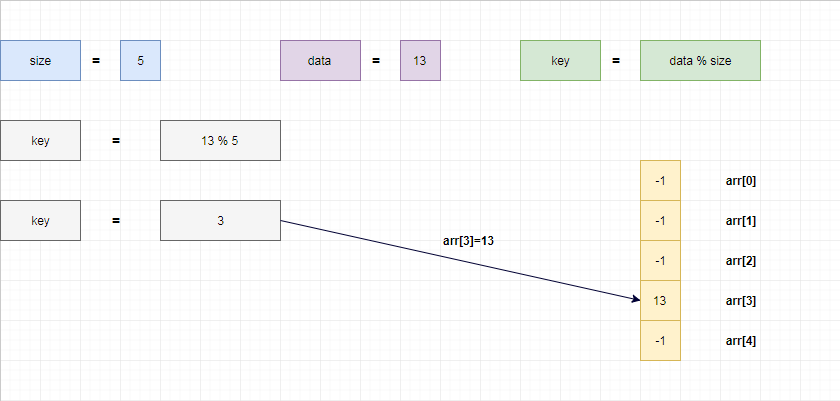
Do the above process till we find the space.

## Linear Probing Procedure

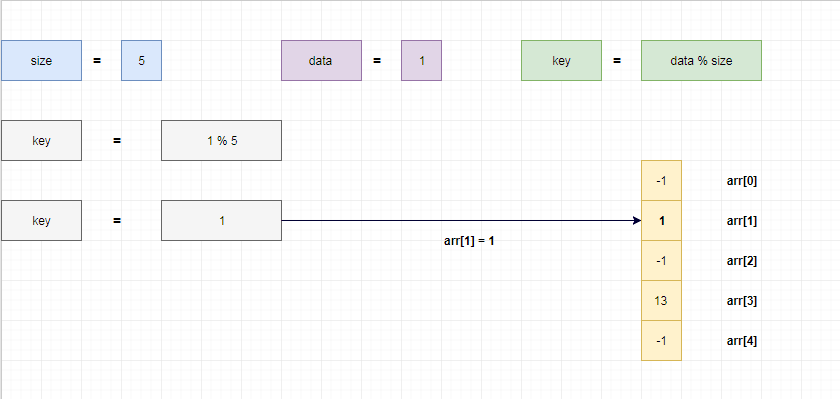
#### Initial Hash Table



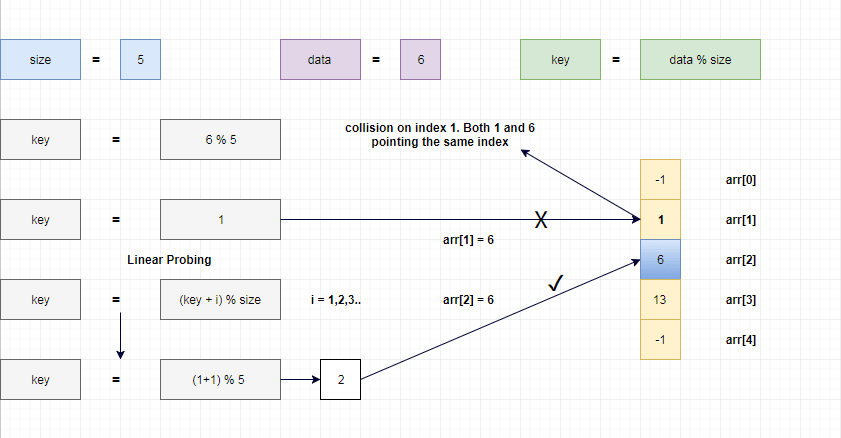
#### Insert 13



#### insert 1



#### Insert 6



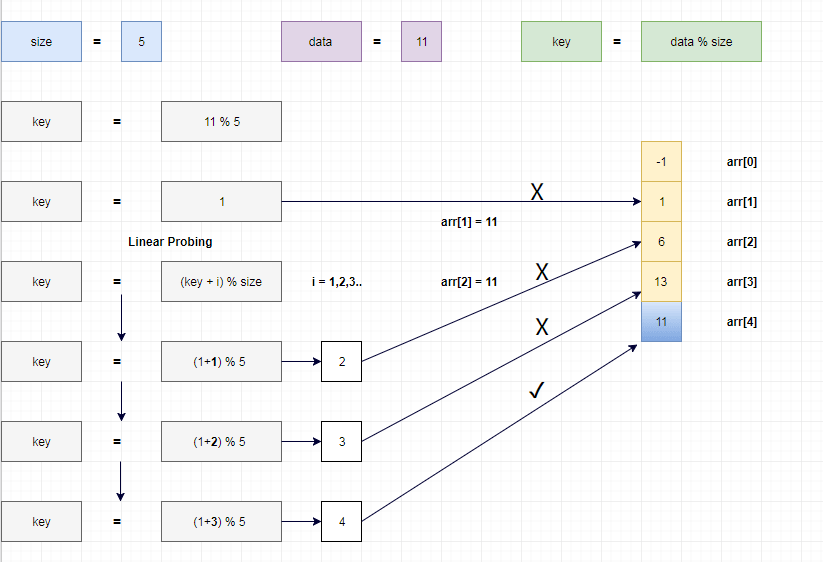
1 % 5 = 1.

6 % 5 = 1.

Both 1 and 6 points the same index under modulo 5.

So that we have placed 6 in arr[2] which is next available index.

#### Insert 11



1 % 5 = 1.

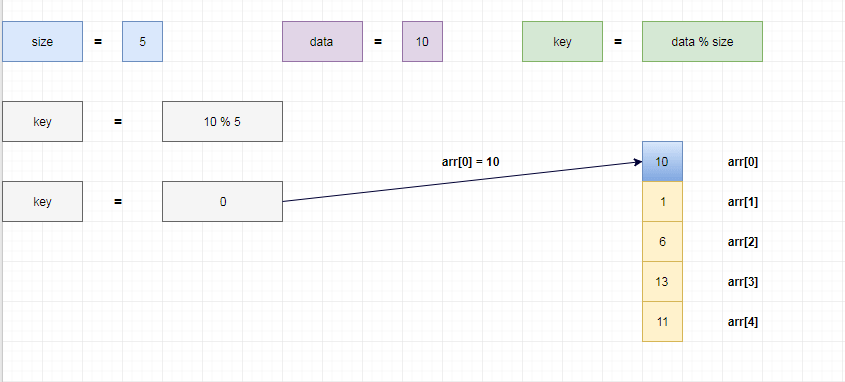
6 % 5 = 1.

11 % 5 = 1.

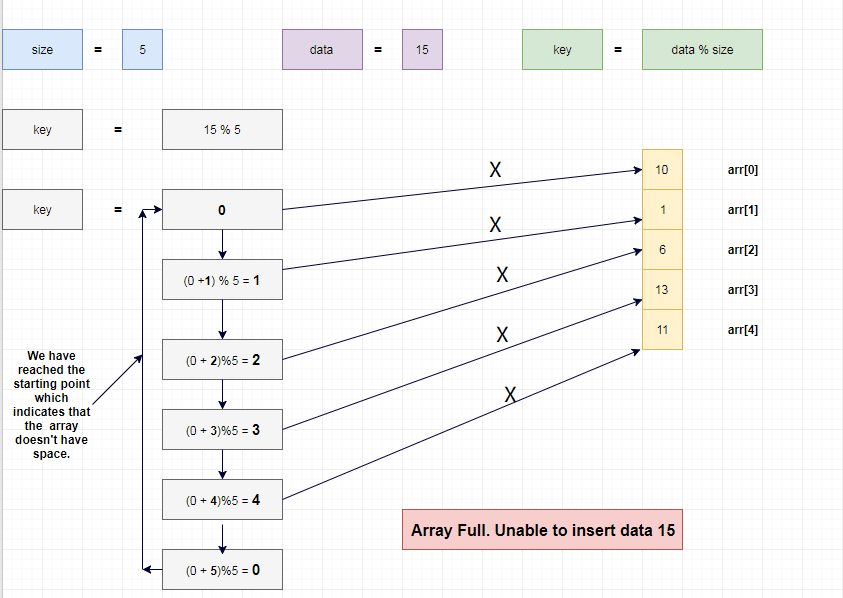
Both 1, 6 and 11 points the same index under modulo 5.

So that we have placed 11 in arr[4] which is next available index.

#### Insert 10



#### Insert 15



15 % 5 = 0.

Hash table don't have any empty index. So, we can't insert the data.

**Quadratic probing**

In this section we will see what is quadratic probing technique in open addressing scheme. There is an ordinary hash function h’(x) : U → {0, 1, . . ., m – 1}. In open addressing scheme, the actual hash function h(x) is taking the ordinary hash function h’(x) and attach some another part with it to make one quadratic equation.

h´ = (𝑥) = 𝑥 𝑚𝑜𝑑 𝑚

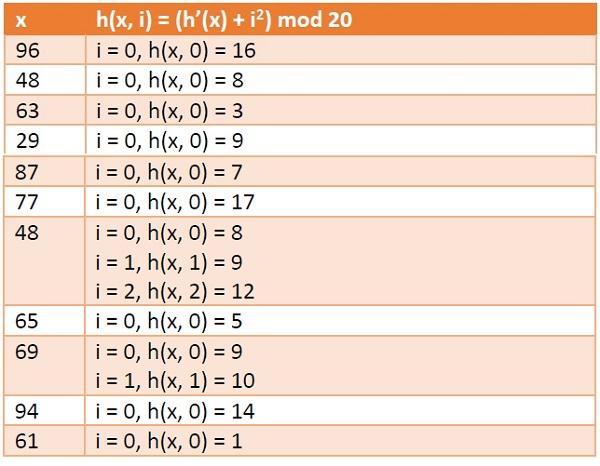
ℎ(𝑥, 𝑖) = (ℎ´(𝑥) + 𝑖2)𝑚𝑜𝑑 𝑚

We can put some other quadratic equations also using some constants

The value of i = 0, 1, . . ., m – 1. So we start from i = 0, and increase this until we get one free space. So initially when i = 0, then the h(x, i) is same as h´(x).

## Example

we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}



**Hash Table**

https://www.tutorialspoint.com/assets/questions/media/41071/quadratic_probing1.jpg

#### Quadratic Probing

Quadratic probing is an open addressing method for resolving collision in the hash table. This method is used to eliminate the primary clustering problem of linear probing. This technique works by considering of original hash index and adding successive value of an arbitrary quadratic polynomial until the empty location is found. In linear probing, we would use **H+0, H+1, H+2, H+3,.....H+K** hash function sequence. Instead of using this sequence, the quadratic probing would use the another sequence is that **H+12, H+22, H+32,....H+K2**. Therefore, the hash function for quadratic probing is  
hi(X) = ( Hash(X) + F(i)2) % TableSize **for i = 0, 1, 2, 3,...etc**.  
Let us examine the same example that is given in linear probing:  
  
**Solution:**

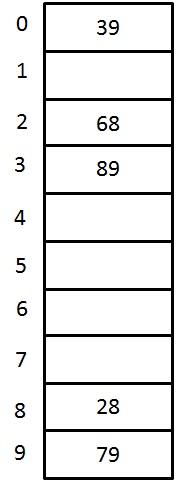
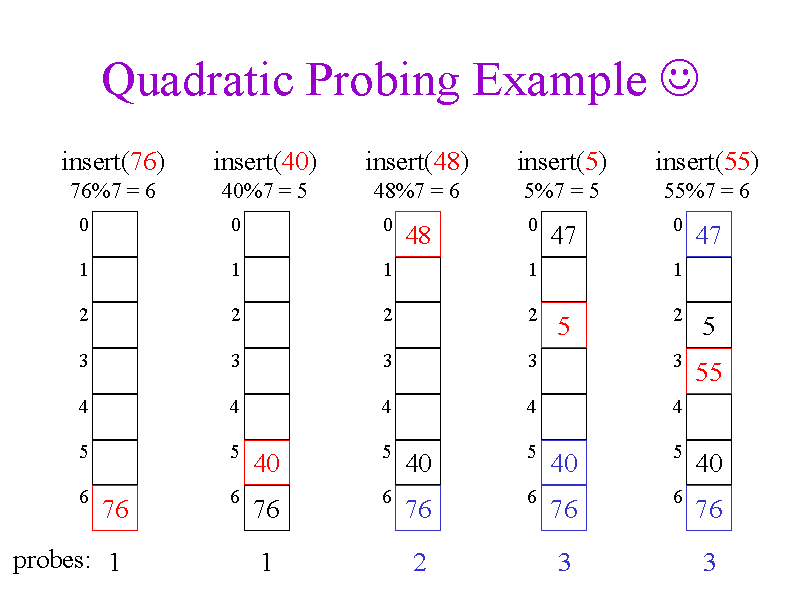


Fig 4. A closed Hash Table using Quadratic Probing

| **Key** | **Hash Function h(X)** | **Index** | **Collision** | **Alt Index** |
| --- | --- | --- | --- | --- |
| **79** | h0(79) = ( Hash(79) + F(0)2) % 10  = ((79 % 10) + 0) % 10 | 9 |  |  |
| **28** | h0(28) = ( Hash(28) + F(0)2) % 10  = ((28 % 10) + 0) % 10 | 8 |  |  |
| **39** | h0(39) = ( Hash(39) + F(0)2) % 10  = ((39 % 10) + 0) % 10 | 9 | The first collision occurs |  |
| h1(39) = ( Hash(39) + F(1)2) % 10  = ((39 % 10) + 1) % 10 | 0 |  | 0 |
| **68** | h0(68) = ( Hash(68) + F(0)2) % 10  = ((68 % 10) + 0) % 10 | 8 | The collision occurs |  |
| h1(68) = ( Hash(68) + F(1)2) % 10  = ((68 % 10) + 1) % 10 | 9 | Again collision occurs |  |
| h2(68) = ( Hash(68) + F(2)2) % 10  = ((68 % 10) + 4) % 10 | 2 |  | 2 |
| **89** | h0(89) = ( Hash(89) + F(0)2) % 10  = ((89 % 10) + 0) % 10 | 9 | The collision occurs |  |
| h1(89) = ( Hash(89) + F(1)2) % 10  = ((89 % 10) + 1) % 10 | 0 | Again collision occurs |  |
| h2(89) = ( Hash(89) + F(2)2) % 10  = ((89 % 10) + 4) % 10 | 3 |  | 3 |

Although, the quadratic probing eliminates the primary clustering, it still has the problem. When two keys hash to the same location, they will probe to the same alternative location. This may cause secondary clustering. In order to avoid this secondary clustering, double hashing method is created where we use extra multiplications and divisions



## Double Hashing

Double Hashing is a hashing collision resolution technique where we use 2 hash functions.

## Double Hashing - Hash Function 1

hi = ( Hash(X) + F(i) ) % Table Size

where

* F(i) = i \* hash2(X)
* X is the Key or the Number for which the hashing is done
* i is the ith time that hashing is done for the same value. Hashing is repeated only when collision occurs
* Table size is the size of the table in which hashing is done

This F(i) will generate the sequence such as hash2(X), 2 \* hash2(X) and so on.

## Double Hashing - Hash Function 2

We use second hash function as

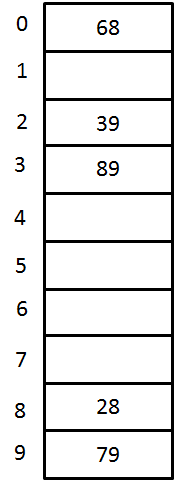
hash2(X) = R - (X mod R)

where

* R is the prime number which is slightly smaller than the Table Size.
* X is the Key or the Number for which the hashing is done

## Double Hashing Example - Closed Hash Table

Let us consider the same example in which we choose R = 7.



A Closed Hash Table using Double Hashing

| **Key** | **Hash Function h(X)** | **Index** | **Collision** | **Alt Index** |
| --- | --- | --- | --- | --- |
| **79** | h0(79) = ( Hash(79) + F(0)) % 10 = ((79 % 10) + 0) % 10    = 9 | 9 |  |  |
| **28** | h0(28) = ( Hash(28) + F(0)) % 10 = ((28 % 10) + 0) % 10     = 8 | 8 |  |  |
| **39** | h0(39) = ( Hash(39) + F(0)) % 10 = ((39 % 10) + 0) % 10     = 9 | 9 | first collision occurs |  |
| h1(39) = ( Hash(39) + F(1)) % 10 = ((39 % 10) + 1(7-(39 % 7))) % 10 = (9 + 3) % 10 =12 % 10     = 2 | 2 |  | 2 |
| **68** | h0(68) = ( Hash(68) + F(0)) % 10 = ((68 % 10) + 0) % 10     = 8 | 8 | collision occurs |  |
| h1(68) = ( Hash(68) + F(1)) % 10 = ((68 % 10) + 1(7-(68 % 7))) % 10 = (8 + 2) % 10 =10 % 10     =0 | 0 |  | 0 |
| **89** | h0(89) = ( Hash(89) + F(0)) % 10 = ((89 % 10) + 0) % 10     = 9 | 9 | collision occurs |  |
| h1(89) = ( Hash(89) + F(1)) % 10 = ((89 % 10) + 1(7-(89 % 7))) % 10 = (9 + 2) % 10 =10 % 10     = 0 | 0 | Again collision occurs |  |
| h2(89) = ( Hash(89) + F(2)) % 10 = ((89 % 10) + 2(7-(89 % 7))) % 10 = (9 + 4) % 10=13 % 10     =3 | 3 |  | 3 |

In this section we will see what is Double Hashing technique in open addressing scheme. There is an ordinary hash function h´(x) : U → {0, 1, . . ., m – 1}. In open addressing scheme, the actual hash function h(x) is taking the ordinary hash function h’(x) when the space is not empty, then perform another hash function to get some space to insert.

h1(x)=xmodmh1(x)=xmodm

h2(x)=xmodm′h2(x)=xmodm′

h(x,i)=(h1(x)+ih2)modmh(x,i)=(h1(x)+ih2)modm

The value of i = 0, 1, . . ., m – 1. So we start from i = 0, and increase this until we get one free space. So initially when i = 0, then the h(x, i) is same as h´(x).

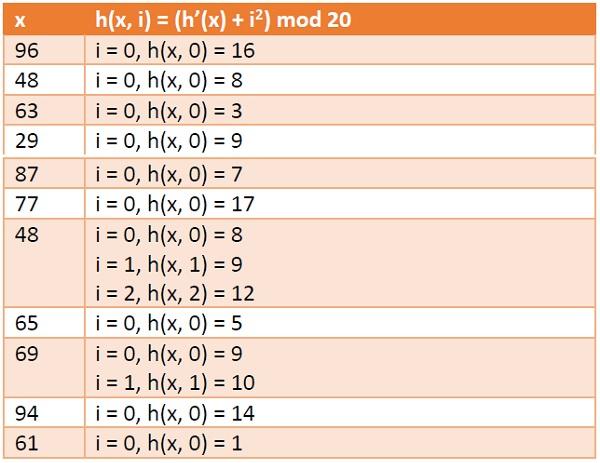
**Example**

Suppose we have a list of size 20 (m = 20). We want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}

h1(x)=xmod20h1(x)=xmod20

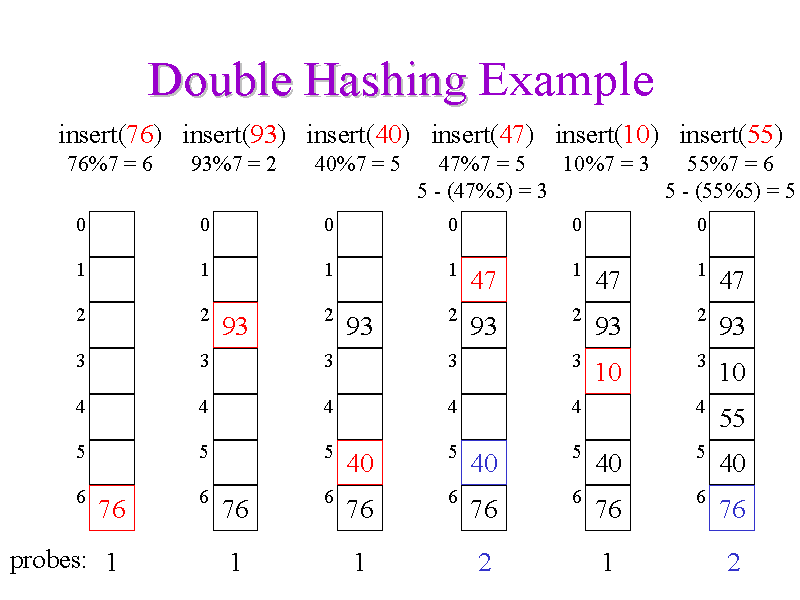
h2(x)=xmod13h2(x)=xmod13

x h(x, i) = (h1 (x) + ih2(x)) mod 20



**Hash Table**

https://www.tutorialspoint.com/assets/questions/media/41072/double_hashing1.jpg



Double hashing is the most efficient collision technique, when the size of the table is prime number and it avoids clustering. Quadratic probing is also efficient but only when the records to be stored are not greater than the half of the table. It has problem of secondary clustering where two keys with the same hash value probes the same position. Linear probing is easier to implement and work with, but its efficiency tends to reduce drastically as the number of records approaches the size of the array.

**Summary Description**

* A Hash Table is an array that, given a key *key*, computes a hash value from *key* (using a hash function) and then uses the hash value to compute an index at which to store *key*
* A Hash Map is the exact same thing as a Hash Table, except instead of storing just *keys*, we store (*key*, *value*) pairs
* The capacity of a Hash Table should be prime (to help reduce collisions)
* When discussing the time complexities of Hash Tables/Maps, we typically ignore the time complexity of the hash function
* The load factor of a Hash Table, \alpha = \frac{N}{M}*α*=*MN*​ (*N* = size of Hash Table and *M* = capacity of Hash Table), should remain below ~0.75 to keep the Hash Table fast (a smaller load factor means better performance, but it also means more wasted space)
* For a hash function *h* to be valid, given two equal keys *k* and *l*, *h*(*k*) must equal *h*(*l*)
* For a hash function *h* to be good, given two unequal keys *k* and *l*, *h*(*k*) should ideally (but not necessarily) not equal *h*(*l*)
* A good hash function for a collection that stores *k* items (e.g. a string storing *k* characters, or a list storing *k* objects, etc.) should perform some non-commutative arithmetic that utilizes each of the *k* elements
* In Linear Probing (a form of Open Addressing), collisions are resolved by simply shifting over to the next available slot
* In Double Hashing (a form of Open Addressing), collisions are resolved in a way similar to Linear Probing, except instead of only shifting over one slot at a time, the Hash Table has a second hash function that it uses to determine the "skip" for the probe
* In Random Hashing (a form of Open Addressing), for a given key *key*, a pseudorandom number generator is seeded with *key*, and the possible indices are given by the sequence of numbers returned by the pseudorandom number generator
* In Separate Chaining (a form of Closed Addressing), each slot of the Hash Table is actually a data structure itself (typically a Linked List), and when a key hashes to a given index in the Hash Table, simply insert it into the data structure at that index
* In Cuckoo Hashing (a form of Open Addressing), the Hash Table has two hash functions, and in the case of a collision, the new key pushes the old key out of its slot, and the old key uses the other hash function to find a new slot

**Time/Space Complexities of a Hash Table/Map with Linear Probing**

**Worst-Case Time Complexity (Linear Probing)**

* **Find: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements
* **Insert: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements
* **Remove: O(*n*)** — If all the keys mapped to the same index, we would need to probe over all *n* elements

**Average-Case Time Complexity (Linear Probing)**

* **Find: O(1)** — The formal proof is too complex for a summary slide
* **Insert: O(1)** — The formal proof is too complex for a summary slide
* **Remove: O(1)** — The formal proof is too complex for a summary slide

**Best-Case Time Complexity (Linear Probing)**

* **Find: O(1)** — No collisions
* **Insert: O(1)** — No collisions
* **Remove: O(1)** — No collisions

**Space Complexity (Linear Probing)**

* **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

**Time/Space Complexities of a Hash Table/Map with Double Hashing**

**Worst-Case Time Complexity (Double Hashing)**

* **Find: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements
* **Insert: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements
* **Remove: O(*n*)** — If we are extremely unlucky, we may have to probe over all *n* elements

**Average-Case Time Complexity (Double Hashing)**

* **Find: O(1)** — The formal proof is too complex for a summary slide
* **Insert: O(1)** — The formal proof is too complex for a summary slide
* **Remove: O(1)** — The formal proof is too complex for a summary slide

**Best-Case Time Complexity (Double Hashing)**

* **Find: O(1)** — No collisions
* **Insert: O(1)** — No collisions
* **Remove: O(1)** — No collisions

**Space Complexity (Double Hashing)**

* **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

**Time/Space Complexities of a Hash Table/Map with Random Hashing**

**Worst-Case Time Complexity (Random Hashing)**

* **Find: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers
* **Insert: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers
* **Remove: O(*n*)** — If each number generated by our generator mapped to an occupied slot, we would need to generate *n* numbers

**Average-Case Time Complexity (Random Hashing)**

* **Find: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)
* **Insert: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)
* **Remove: O(1)** — The formal proof is too complex for a summary slide (ignoring the time complexity of the pseudorandom number generator)

**Best-Case Time Complexity (Random Hashing)**

* **Find: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)
* **Insert: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)
* **Remove: O(1)** — No collisions (ignoring the time complexity of the pseudorandom number generator)

**Space Complexity (Random Hashing)**

* **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined)

**Time/Space Complexities of a Hash Table/Map with Separate Chaining**

**Worst-Case Time Complexity (Separate Chaining)**

* **Find: O(*n*)** — If all the keys mapped to the same index (assuming Linked List)
* **Insert: O(*n*)** — If all the keys mapped to the same index (assuming Linked List) and we check for duplicates
* **Remove: O(*n*)** — If all the keys mapped to the same index (assuming Linked List)

**Average-Case Time Complexity (Separate Chaining)**

* **Find: O(1)** — The formal proof is too complex for a summary slide
* **Insert: O(1)** — The formal proof is too complex for a summary slide
* **Remove: O(1)** — The formal proof is too complex for a summary slide

**Best-Case Time Complexity (Separate Chaining)**

* **Find: O(1)** — No collisions
* **Insert: O(1)** — No collisions
* **Remove: O(1)** — No collisions

**Space Complexity (Separate Chaining)**

* **O(*n*)** — Hash Tables typically have a capacity that is at most some constant multiplied by *n* (the constant is predetermined), and each of our *n* nodes occupies O(1) space