8
Hash Tables

A hash table is a data structure that implements an associative array in which the data is stored by mapping the keys to the values as `key-value` pairs. In many applications, we mostly require different operations such as insert, search, and delete in a dictionary data structure. For example, a symbol table is a data structure based on a hash table that is used by the compiler. A compiler that translates a programming language maintains a symbol table in which keys are character strings that are mapped to the identifiers. In such situations, a hash table is an effective data structure since we can directly compute the index of the required record by applying a hash function to the key. So, instead of using the key as an array index directly, the array index is computed by applying the hash function to the key. It makes it very fast to access an element from any index from the hash table. The hash table uses the hashing function to compute the index of where the data item should be stored in the hash table.

While looking up an element in the hash table, hashing of the key gives the index of the corresponding record in the table. Ideally, the hash function assigns a unique value to each of the keys; however, in practice, we may get hash collisions where the hash function generates the same index for more than one key. In this chapter, we will be discussing different techniques that deal with such collisions.

In this chapter, we will discuss all the concepts related to these, including:

- Hashing methods and hash table techniques
- Different collision resolution techniques in hash tables

Introducing hash tables

As we know, **arrays** and **lists** store the data elements in sequence. As in an array, the data items are accessed by an index number. Accessing array elements using index numbers is fast. However, they are very inconvenient to use when it is required to access any element when we can't remember the index number. For example, if we wish to extract the phone number for a person from the address book at index 56, there is nothing to link a particular contact with number 56. It is difficult to retrieve an entry from the list using the index value.

Hash tables are a data structure better suited to this kind of problem. A **hash table** is a data structure where elements are accessed by a keyword rather than an index number, unlike in **lists** and **arrays**. In this data structure, the data items are stored in key-value pairs similar to dictionaries. A hash table uses a hashing function in order to find an index position where an element should be stored and retrieved. This gives us fast lookups since we are using an index number that corresponds to the hash value of the key.

An overview of how the hash table stores the data is shown in *Figure 8.1*, in which key values are hashed using any hash function to obtain the index position of the record in the hash table.

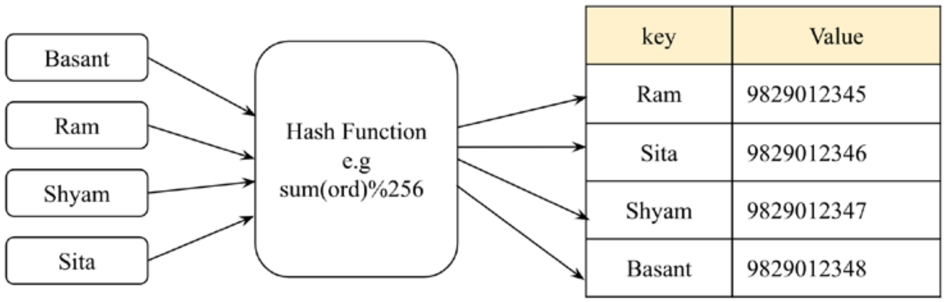


Figure 8.1: An example of a hash table

Dictionaries are a widely used data structure, often built using hash tables. A dictionary uses a keyword instead of an index number, and it stores data in (`key`, `value`) pairs. That is, instead of accessing the contact with the index value, we use the *key* value in the dictionary data structure.

The following code demonstrate the working of dictionaries that store the data in (`key`, `value`) pairs:

```
my_dict={"Basant" : "9829012345", "Ram": "9829012346", "Shyam":
"9829012347", "Sita": "9829012348"}
```

```
print("All keys and values")
for x,y in my_dict.items():
    print(x, ":" , y)       #prints keys and values
my_dict["Ram"]
```

The output of the preceding code is as follows:

```
Basant : 9829012345
Ram : 9829012346
Shyam : 9829012347
Sita : 9829012348

'9829012346'
```

Hash tables stores the data in a very efficient way so that retrieval can be very fast. Hash tables are based on a concept called hashing.

Hashing functions

Hashing is a technique in which, when we provide data of arbitrary size to a function, we get a small, simplified value. This function is called a hash function. Hashing uses a hash function to map the keys to another range of data in a way that a new range of keys can be used as an index in the hash table; in other words, hashing is used to convert the key values to integer values, which can be used as an index in the hash table.

In our discussions in this chapter, we are using hashing to convert strings into integers. We could have used any other data type in place of strings that can be converted into integers. Let's take an example. Say, we want to hash the expression `hello world`, that is, we want to get a numeric value corresponding to this string that can be used as an index in the hash table.

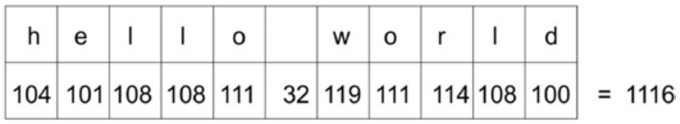
In Python, the `ord()` function returns a unique integer value (known as ordinal values) that is mapped to a character in the Unicode encoding system. The ordinal values map the Unicode character to a unique numerical representation provided the character is Unicode-compatible, for example, numbers 0-127 are mapped to ASCII characters, which also correspond to the ordinal values within Unicode systems. However, the range of Unicode encoding may be larger. So, Unicode encoding is a superset of ASCII. For example, in Python, we get a unique ordinal value `102` for character '`f`' by using `ord('f')`. Further, to get the hash of the whole string, we could just sum the ordinal numbers of each character in the string. See the following code snippet:

```
sum(map(ord, 'hello world'))
```

The output of the above is as follows:

```
1116
```

In the above output, we obtain a numeric value, `1116`, for the `hello world` string, which is the **hash** of the given string. Consider the following *Figure 8.2* to see the ordinal values of each character in the string that results in the hash value `1116`:



h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116

Figure 8.2: Ordinal values of each character for the hello world string

The preceding approach used to obtain the hash value for a given string has the problem that more than one string can have the same hash value; for example, when we change the order of the characters in the string and we have the same hash value. See the following code snippet where we get the same hash value for the `'world hello'` string:

```
sum(map(ord, 'world hello'))
```

The output of the above is as follows:

```
1116
```

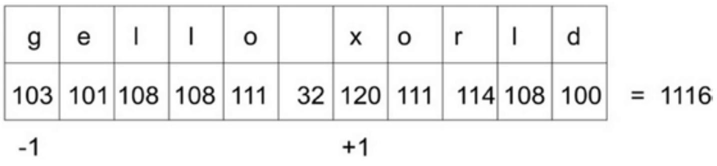
Again, there would be the same hash value for the `'gello xorld'` string, as the sum of the ordinal values of the characters for this string would be the same since g has an ordinal value that is one less than that of `h`, and x has an ordinal value that is one greater than that of `w`. See the following code snippet:

```
sum(map(ord, 'gello xorld'))
```

The output of the above is as follows:

```
1116
```

Look at the following *Figure 8.3*, where we can see that the hash value for this `'gello xorld'` string is again `1116`:



g	e	l	l	o		x	o	r	l	d	
103	101	108	108	111	32	120	111	114	108	100	= 1116
-1						+1					

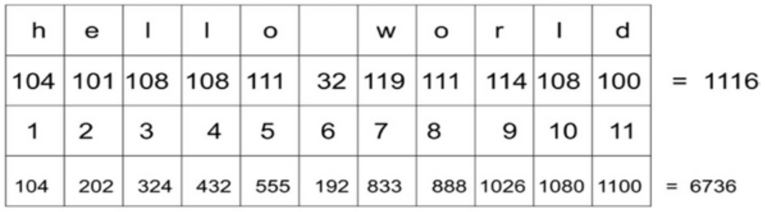
Figure 8.3: Ordinal values of each character for the gello xorld string

In practice, most of the hashing functions are imperfect and face collisions. This means that a hash function gives the same hash value to more than one string. Such collisions are undesirable for implementing the hash table.

Perfect hashing functions

A perfect **hashing function** is one by which we get a unique hash value for a given string (it can be any data type; here, we are using a string data type as an example). Our aim is to create a hash function that minimizes the number of collisions, is fast, easy to compute, and distributes the data items equally in the hash table. But, normally, creating an efficient hash function that is fast as well as providing a unique hash value for each string is very difficult. If we try to develop a hash function that avoids collisions, this becomes very slow, and a slow hash function does not serve the purpose of the hash table. So, we use a fast hash function and try to find a strategy to resolve the collisions rather than trying to find a perfect hash function.

To avoid the collisions in the hash function discussed in the previous section, we can add a multiplier to the ordinal value of each character that continuously increases as we progress in the string. Furthermore, the hash value of the string can be obtained by adding the multiplied ordinal value of each character. To better understand the concept, refer to the following *Figure 8.4*:



h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116
1	2	3	4	5	6	7	8	9	10	11	
104	202	324	432	555	192	833	888	1026	1080	1100	= 6736

Figure 8.4: Ordinal values multiplied by numeric values for each character of the hello world string

In the preceding *Figure 8.4*, the ordinal value of each character is progressively multiplied by a number. Note that row two has the ordinal values of each character; row three shows the multiplier value; and, in row four, we get values by multiplying the values of rows two and three so that `104 x 1` equals `104`. Finally, we add all of these multiplied values to get the hash value of the `hello world` string, that is, `6736`.

The implementation of this concept is shown in the following function:

```
def myhash(s):
        mult = 1
        hv = 0
        for ch in s:
            hv += mult * ord(ch)
            mult += 1
        return hv
```

We can test this function on the strings that we used earlier, shown as follows:

```
for item in ('hello world', 'world hello', 'gello xorld'):
        print("{}: {}".format(item, myhash(item)))
```

When we execute the preceding code, we get the following output:

```
hello world: 6736
world hello: 6616
gello xorld: 6742
```

We can see that this time, we get different hash values for these three strings. Still, this is not a perfect hash. Let's now try the strings `ad` and `ga`:

```
for item in ('ad', 'ga'):
        print("{}: {}".format(item, myhash(item)))
```

The output of the preceding code snippet is as follows:

```
ad: 297
ga: 297
```

So, we still do not have a perfect hash function since we get the same hash values for these two different strings. Therefore, we need to devise a strategy for resolving such collisions. We will discuss more strategies to resolve collisions in the next sections.

Resolving collisions

Each position in the hash table is often called a **slot** or **bucket** that can store an element. Each data item in the form of a (`key`, `value`) pair is stored in the hash table at a position that is decided by the hash value of the key. Let's take an example in which firstly we use the hashing function that computes the hash value by summing up the ordinal values of all the characters. Then, we compute the final hash value (in other words, the index position) by computing the total ordinal values of module 256. Here, we use 256 slots/buckets as an example. We can use any number of slots depending upon how many records we require in the hash table. We show a sample hash in *Figure 8.5*, which has key strings corresponding to data values, for example, the `eggs` key string has the corresponding data value `123456789`.

This hash table uses a hashing function that maps the input string `hello world` to a hash value of `92`, which finds a slot position in the hash table:

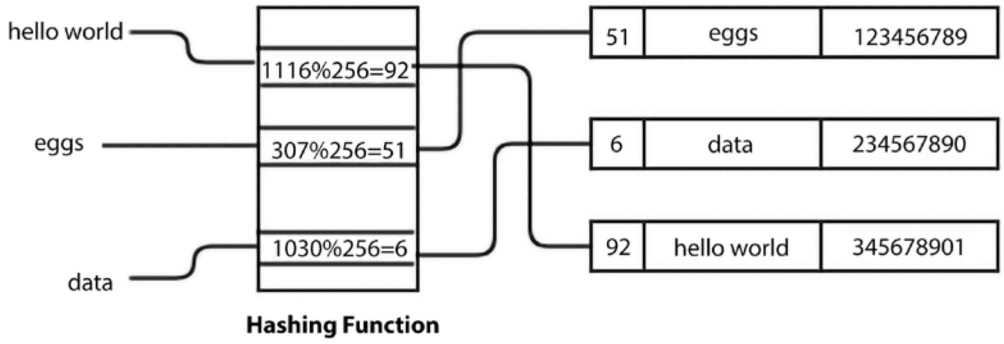


Figure 8.5: A sample hash table

Once we know the hash value of the key, it will be used to find the position where the element should be stored in the hash table. So, we need to find an empty slot. We start at the slot that corresponds to the hash value of the key. If that slot is empty, we insert the data item there. And, if the slot is not empty, that means we have a collision. It means that we have a hash value for the item that is the same as a previously stored item in the table. We need to ascertain a strategy to avoid such collisions or conflicts.

For example, in the following diagram, the key string `hello world` is already stored in the table at index position `92`, and with a new key string, for example, `world hello`, we get the same hash value of `92`. This means that there is a collision. Refer to the following *Figure 8.6* depicting this concept:

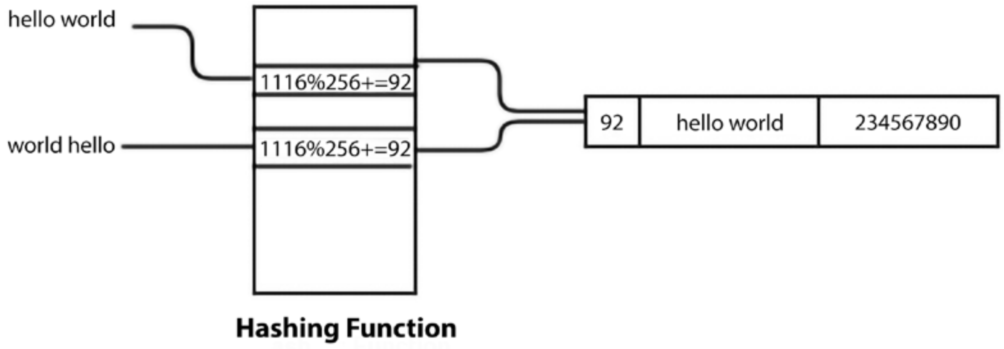


Figure 8.6: Hash values of two strings are the same

One way of resolving this kind of collision is to find another free slot from the position of the collision. This collision resolution process is called **open addressing**.

Open addressing

In open addressing, the key values are stored in the hash table, and collisions are resolved using the probing technique. Open addressing a collision resolution technique used in hash tables. The collision is resolved by searching (also called probing) an alternate position until we get an unused slot in the hash table to store the data item.

There are three popular approaches for an open addressing-based collision resolution technique:

1. Linear probing
2. Quadratic probing
3. Double hashing

Linear probing

The systematic way of visiting each slot is a linear way of resolving collisions, in which we linearly look for the next available slot by adding `1` to the previous hash value where we get the collision. This is known as linear probing. We can resolve the conflict by adding `1` to the sum of the ordinal values of each character in the key string, which is further used to compute the final hash value by taking its modulo according to the size of the hash table.

Let's consider an example. First, compute the hash value of the key. If the position is occupied, we check the hash table sequentially for the next free slot. Let's use this to resolve a collision, as shown in the following *Figure 8.7*, wherein, for the key string egg, the sum of ordinal values comes to `307`, and then we compute the hash value by taking the module `256`, which gives the hash value for the egg key string as `51`. However, data is already stored at this position, so this means a collision. Therefore, we add `1` to the hash value that is computed by the sum of the ordinal values of each character of the string. In this way, we obtain a new hash value, `52`, for this key string to store the data. Refer to the following *Figure 8.7*, which depicts the above process:

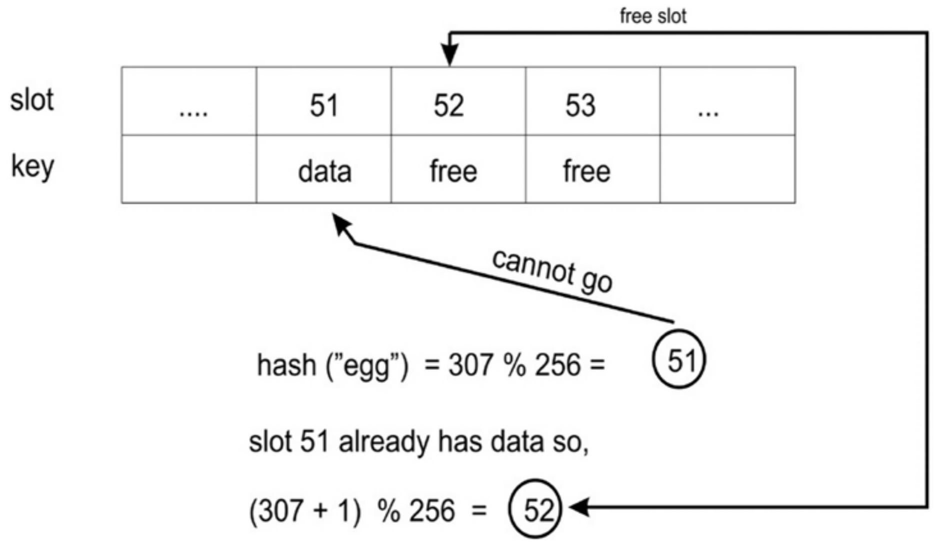


Figure 8.7: An example of collision resolution

In order to find the next free slot in the hash table, we increment the hashing value, and this increment is fixed in the case of linear probing. Due to a fixed increment in the hashing value when we get collisions, the new data element is always stored at the next available index position given by the hash function. This creates a continuous cluster of occupied index positions, with this cluster growing whenever we get another data element that has a hash value anywhere within the cluster.

So, one major drawback of this approach is that the hash table can have consecutive occupied positions that are called clusters of items. In this case, one portion of the hash table may become dense, with the other part of the table remaining empty. Because of these limitations, we may prefer to use a different strategy to resolve collisions such as quadrant probing or double hashing, which we will discuss in forthcoming sections. Let us first discuss the implementation of the hash table with linear probing as a collision resolution technique, and after understanding the concept of linear probing, we will discuss other collision resolution techniques.

Implementing hash tables

To implement the hash table, we start by creating a class to hold hash table items. These need to have a key and a value since the hash table is a `{key-value}` store:

```
class HashItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

Next, we start working on the hash table class itself. As usual, we start with a constructor:

```
class HashTable:
     def __init__(self):
         self.size = 256
         self.slots = [None for i in range(self.size)]
         self.count = 0
```

Standard Python lists can be used to store data elements in a hash table. Let's set the size of the hash table to 256 elements to start with. Later, we will look at strategies for how to grow the hash table as we begin filling it up. We will now initialize a list containing `256` elements in the code. These are the positions where the elements are to be stored—the slots or buckets. So, we have `256` slots to store elements in the hash table. It is important to note the difference between the size and count of a table. The size of a table refers to the total number of slots in the table (used or unused). The count of the table refers to the number of slots that are filled, meaning the number of actual `(key-value)` pairs that have been added to the table.

Now, we have to decide on a hashing function for the table. We can use any hash function. Let's take the same hash function that returns the sum of ordinal values for each character in the strings with a slight modification. Since this hash table has `256` slots, that means we need a hashing function that returns a value in the range of `0` to `255` (the size of the table). A good way of doing it is to return the remainder of dividing the hash value by the size of the table since the remainder would surely be an integer value between `0` and `255`.

Since the hashing function is only meant to be used internally by the class, we put an underscore (_) at the beginning of the name to indicate this. This is a Python convention for indicating that something is intended for internal use. Here is the implementation of the `hash` function, which should be defined in the `HashTable` class:

```
def _hash(self, key):
        mult = 1
        hv = 0
        for ch in key:
            hv += mult * ord(ch)
            mult += 1
        return hv % self.size
```

For the time being, we are assuming that keys are strings. We will discuss how non-string keys can be used later. For now, the `_hash()` function is going to generate the hash value for a string.

Storing elements in a hash table

To store the elements in the hash table, we add them to the table with the `put()` function and retrieve them with the `get()` function. First, we will look at the implementation of the `put()` function. We start by adding the key and the value to the `HashItem` class and then compute the hash value of the key. The `put()` method should be defined in the `HashTable` class:

```
def put(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + 1) % self.size
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

After obtaining the hash value of the key and if the slot is not empty, the next free slot is checked by adding `1` to the previous hash value by applying the linear probing technique. Consider the following code:

```
while self.slots[h] != None:
      if self.slots[h].key == key:
          break
      h = (h + 1) % self.size
```

If the slot is empty, then we increase the count by one and store the new element (meaning the slot contained `None` previously) in the list at the required position. Refer to the following code:

```
if self.slots[h] is None:
     self.count += 1
self.slots[h] = item
self.check_growth()
```

In the above code, we have created a hash table and discussed the `put()` method for storing the data element in the hash table with the linear probing technique at the time of the collision.

In the last line of the preceding code, we call a `check_growth()` method, which is used to expand the size of the hash table when we have a very limited number of empty slots remaining in the hash table. We will discuss this in more detail in the next section.

Growing a hash table

In the example that we have discussed, we have fixed the hash table size at 256. It is obvious that, when we add the elements to the hash table, the hash table starts filling up, and at some point, all of the slots would be filled up and the hash table will be full. To avoid such a situation, we can grow the size of the table when it is starting to get full.

To grow the size of the hash table, we compare the size and the count in the table. `size` is the total number of slots, and `count` denotes the number of slots that contain elements. So, if `count` is equal to `size`, this means we have filled up the table. The load factor of the hash table is generally used to expand the size of the table; that gives us an indication of how many available slots of the table have been used. The load factor of the hash table is computed by dividing the number of **used** slots by the **total** number of slots in the table. It is defined as follows:

$$\text{Load factor} = n/k$$

Here, n is the number of used slots, and k is the total number of slots. As the load factor value approaches 1, this means that the table is going to be filled, and we need to grow the size of the table. It is better to grow the size of the table before it gets almost full, as the retrieval of elements from the table becomes slow when the table fills up. A value of 0.75 for the load factor may be a good value to grow the size of the table. Another question is how much we should increase the size of the table. One strategy would be to simply double its size.

The problem of linear probing is that as the load factor increases, it takes a long time to find the insertion point for the new element. Moreover, in the case of the open addressing collision resolution technique, we should grow the size of the hash table depending upon the load factor to reduce the number of collisions.

The implementation of growing the hash table when the load factor increases more than the threshold is as follows. First, we redefine the `HashTable` class that includes one more variable, `MAXLOADFACTOR`, that is used to ensure that the load factor of the hash table is always below the predefined maximum load factor. The `HashTable` class is defined as follows:

```
class HashTable:
     def __init__(self):
         self.size = 256
         self.slots = [None for i in range(self.size)]
         self.count = 0
         self.MAXLOADFACTOR = 0.65
```

Next, we check the load factor of the hash table after adding any record to the hash table using the following `check_growth()` method, which should be defined in the `HashTable` class:

```
    def check_growth(self):
        loadfactor = self.count / self.size
        if loadfactor > self.MAXLOADFACTOR:
            print("Load factor before growing the hash table", self.count
/ self.size )
            self.growth()
            print("Load factor after growing the hash table", self.count /
self.size )
```

In the preceding code, we compute the load factor of the table, and then we check if it is more than the set threshold (in other words, `MAXLOADFACTOR` is a variable that we initialize at the time of creating a hash table). In that case, we call the `growth()` method that increases the hash table size (in this example, we are doubling the hash table size). The `growth()` method, which should be defined in the `HashTable` class, is implemented as follows:

```
    def growth(self):
        New_Hash_Table = HashTable()
        New_Hash_Table.size = 2 * self.size
        New_Hash_Table.slots = [None for i in range(New_Hash_Table.size)]

        for i in range(self.size):
             if self.slots[i] != None:
                New_Hash_Table.put(self.slots[i].key, self.slots[i].value)

        self.size = New_Hash_Table.size
        self.slots = New_Hash_Table.slots
```

In the preceding code, we firstly create a new hash table double the size of the original hash table and then we initialize all of its slots to be `None`. Next, we check all the filled slots in the original hash table where we have the data, since we have to insert all these existing records into the new hash table, hence, we call the `put()` method with all the key-value pairs of the existing hash table. Once we copy all the records to the new hash table, we replace the size and slots of the existing table with the new hash table.

Let's create a hash table with a maximum capacity of 10 records and a threshold load factor of 65% by defining `self.size = 10` in the `__init__` method in the `HashTable` class, meaning whenever a seventh record is added to the hash table, we call a `check_growth()` method:

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")
ht.put("awd", "do not")
ht.put("add", "do not")
ht.checkGrow()
```

In the above code, we add seven records using the `put()` method. The output of the preceding code is as follows:

```
Load factor before growing the hash table 0.7
Load factor after growing the hash table 0.35
```

In the above output, we can see that the load factor before and after adding the seventh record became half of the load factor before growing the hash table.

In the next section, we will discuss the `get()` method for retrieving the data element that we have stored in the hash table.

Retrieving elements from the hash table

To retrieve the elements from the hash table, the value stored corresponding to the key would be returned. Here, we discuss the implementation of the retrieval method—the `get()` method. This method returns the value stored in the table corresponding to the given key.

Firstly, we compute the hash of the given key corresponding to the value that is to be retrieved. Once we have the hash value of the key, we look up the hash table at the position of the hash value. If the key item is matched with the stored key value at that location, the corresponding value is retrieved.

If that does not match, then we add 1 to the sum of the ordinal values of all the characters in the string, similar to what we did at the time of storing the data, and we look at the newly obtained hash value. We keep searching until we get the key element, or we check all the slots in the hash table.

Here, we used the linear probing technique to resolve the collision, and hence we use the same technique when retrieving the data element from the hash table. Hence, if we were to use a different technique, let's say double hashing or quadratic probing at the time of storing the data element, we should use the same method to retrieve the data element. Consider an example to understand the concept in *Figure 8.8*, and in the following four steps:

1. We compute the hash value for the given key string, egg, which turns out to be 51. Then, we compare this key with the stored key value at location 51, but it does not match.
2. As the key does not match, we compute a new hash value.
3. We look up the key at the location of the newly created hash value, which is 52; we compare the key string with the stored key value and, here, it matches, as shown in the following diagram.
4. The stored value is returned corresponding to this key value in the hash table. See the following *Figure 8.8*:

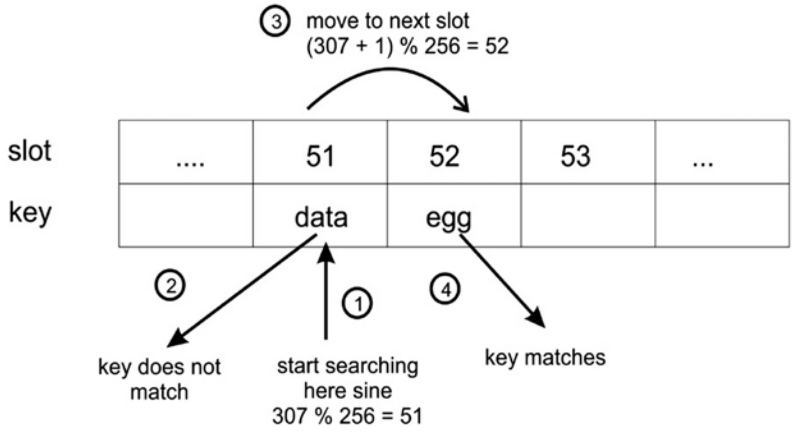


Figure 8.8: Four steps are demonstrated for retrieving an element from the hash table

To implement this retrieval method, that is, the `get()` method, we start by calculating the hash of the key. Next, we look up the computed hash value in the table. If there is a match, we return the corresponding stored value. Otherwise, we keep looking at the new hash value location computed as described. Here is the implementation of the `get()` method, which should be defined in the `HashTable` class:

```
def get(self, key):
    h = self._hash(key)    # computed hash for the given key
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ 1) % self.size
    return None
```

Finally, we return `None` if the key was not found in the table; we could have printed the message that the key is not found in the hash table.

Testing the hash table

To test the hash table, we create `HashTable` and store a few elements in it, and then try to retrieve them. We can use `get()` method to find out if a record exists for a given key. We also use the two strings, `ad` and `ga`, that had the collision and returned the same hash value with our hashing function. To evaluate the work of the hash table, we throw this collision as well, just to see that the collision is properly resolved. Refer to the example code, as follows:

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")

for key in ("good", "better", "best", "worst", "ad", "ga"):
        v = ht.get(key)
        print(v)
```

After executing the above code, we get the following output:

```
eggs
ham
```

```
spam
none
do not
collide
```

As you can see, looking up the `worst` key returns `None`, since the key does not exist. The `ad` and `ga` keys also return their corresponding values, showing that the collision between them is handled properly.

Implementing a hash table as a dictionary

Using the `put()` and `get()` methods to store and retrieve elements in the hash table may look slightly inconvenient. However, we can also use the hash table as a dictionary, as it would be easier to use. For example, we would like to use `ht["good"]` instead of `ht.get("good")` to retrieve elements from the table.

This can easily be done with the special methods, `__setitem__()` and `__getitem__()`, which should be defined in the `HashTable` class.

See the following code for this:

```
def __setitem__(self, key, value):
     self.put(key, value)

def __getitem__(self, key):
     return self.get(key)
```

Now, our test code would be like the following:

```
ht = HashTable()
ht["good"] = "eggs"
ht["better"] = "ham"
ht["best"] = "spam"
ht["ad"] = "do not"
ht["ga"] = "collide"
for key in ("good", "better", "best", "worst", "ad", "ga"):
     v = ht[key]
     print(v)
print("The number of elements is: {}".format(ht.count))
```

The output of the preceding code is as follows:

```
eggs
ham
spam
none
do not
collide
The number of elements is: 5
```

Notice that we also print the number of elements already stored in the hash table using the count variable. The above code does the same thing as we did in the previous section, but it is just more convenient to use.

In the next section, we discuss the quadratic probing technique for collision resolution.

Quadratic probing

This is also an open addressing scheme for resolving collisions in hash tables. It resolves the collision by computing the hash value of the key and adding successive values of a quadratic polynomial; the new hash is iteratively computed until an empty slot is found. If a collision occurs, the next free slots are checked at the locations $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$, and so on. Hence, the new hash value is computed as follows:

```
new-hash(key) = (old-hash-value + i²)
Here, hash-value = key mod table_size
```

When we have a key as strings, we compute the hash value using the sum of the ordinal values multiplied by numeric values for each character, and then we pass it the hash function to finally obtain the hash of the key string. However, in the case of non-string key elements, we can use the hash function directly to compute the hash of the key.

Let us take a simple example of a hash table in which we have seven slots and assume that the hash function is `h(key) = key mod 7`. To understand the concept of quadratic probing, let's assume that we have key element values that are the hash of the given key strings.

So, whenever we use the quadratic probing technique to ascertain the next index positions to store a data element when we have a collision, we should perform the following steps to resolve the collision:

1. Initially, since we have an empty table, when we get a key element of `15` (assuming it is a hash of the given string), we compute the hash value using our given hash function, in other words, `15 mod 7= 1`. So, the data element is stored at index position `1`.
2. Then, let's say we get a key element of `22` (assuming it is a hash of the next given string), we use the hash function to compute the hash value, in other words, `22 mod 7 = 1`, it gives the index position `1`. Since index position `1` is already occupied, there is a collision, so we compute a new hash value using quadratic probing, which is $(1+ 1^2 = 2)$. The new index position is `2`. Therefore, the data element is stored at index position `2`.
3. Next, assuming that we get a data element of `29` (assuming it is a hash of the given string), we compute the hash value `29 mod 7 = 1`. Since we have a collision here, we compute the hash value again as in *step 2*, but we get another collision here, so we have to recompute the hash value once more, in other words $(1+2^2 = 5)$, so the data is stored at that location.

The above example of resolving the process using the quadratic probing technique is shown in *Figure 8.9*:

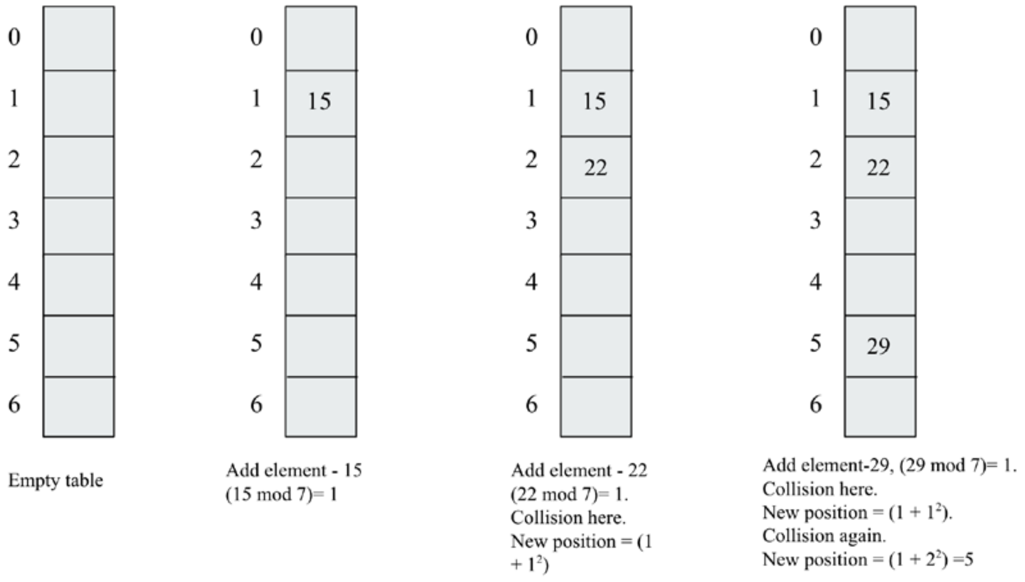


Figure 8.9: Example of collision resolution using quadratic probing

The quadratic probing technique for collision avoidance does not suffer from the formation of clusters of items in the same way as linear probing; however, it does suffer from secondary clustering. Secondary clustering creates a long run of filled slots since the data elements that have the same hash value will also have the same probe sequence.

We discussed the implementation of a hash table in the previous section with the addition and retrieval of data elements, and we used the linear probing technique to resolve the collision. Now, we can update the implementation of the hash table if we want to use any other collision resolution technique, such as the quadratic probing technique. All the methods will be the same in the `HashTable` class except the following two methods, which should be defined in the `HashTable` class:

```
    def get_quadratic(self, key):
        h = self._hash(key)
        j = 1
        while self.slots[h] != None:
            if self.slots[h].key == key:
                return self.slots[h].value
            h = (h+ j*j) % self.size
            j = j + 1
        return None

    def put_quadratic(self, key, value):
        item = HashItem(key, value)
        h = self._hash(key)
        j = 1
        while self.slots[h] != None:
            if self.slots[h].key == key:
                break
            h = (h + j*j) % self.size
            j = j+1
        if self.slots[h] == None:
            self.count += 1
        self.slots[h] = item
        self.check_growth()
```

The above code of the `get_quadratic()` and `put_quadratic()` methods are similar to the implementation of the `get()` and `put()` methods that we discussed earlier, except for the fact that the code statements are in bold in the preceding codes. The bold statements are indicating that at the time of the collision, we check the next empty slot using the quadratic probing formula:

```
ht = HashTable()
ht.put_quadratic("good", "eggs")
```

```python
ht.put_quadratic("ad", "packt")
ht.put_quadratic("ga", "books")

v = ht.get_quadratic("ga")
print(v)
```

In the above code, we first add three data elements along with their associated values, and then we search for a data item with the key `"ga"` in the hash table. The output of the preceding code is as follows:

```
books
```

The above output corresponds to the key string `"ga"`, which is correct as per the input data stored in the hash table. Next, we will discuss another collision resolution technique – double hashing.

Double hashing

In the double hashing collision resolution technique, we use two hashing functions. This technique works as follows. Firstly, the primary hash function is used to compute the index position in the hash table, and whenever we get a collision, we use another hash function to decide the next free slot to store the data by incrementing the hashing value.

In order to find the next free slot in the hash table, we increment the hashing value, and this increment is fixed in the case of linear probing and quadratic probing. Due to a fixed increment in the hashing value when we get collisions, the record is always moved to the next available index position given by the hash function. It creates a continuous cluster of occupied index positions. This cluster grows whenever we get another record that has a hash value anywhere within the cluster.

However, in the case of the double hashing technique, the probing interval depends on the key data itself, meaning that we always map to the different index positions in the hash table whenever we get a collision, which, in turn, helps in avoiding the formation of clusters.

The probing sequence for this collision resolving technique is as follows:

$$(h^1(key)+i*h^2(key))mod\ table\_size$$
$$h^1(key)\ =\ key\ mod\ table\_size$$

It is important to note here that the second hash function should be fast, easy to compute, should not evaluate to 0, and should be different from the first hash function.

One choice for the second hash function can be defined as follows:

```
h2(key) = prime_number - (key mod prime_number)
```

In the above hash function, the prime number should be less than the table size.

For example, let's say we have a hash table that can have a maximum of seven slots when we add data elements `{15, 22, 29}` to this table in sequence. The following steps are performed to store these data elements in the hash table using the double hashing technique when we get a collision:

1. Firstly, we have data element `15`, and we compute the hash value using the primary hash function, in other words, `(15 mod 7 = 1)`. Since the table is empty initially, we store the data at index position `1`.
2. Next, the data element is `22`, and we compute the hash value using the primary hash function, in other words, `(22 mod 7 = 1)`. Since the index position 1 is already filled, this means there is a collision. Next, we use the secondary hashing function defined above as `h2(key) = prime_number - (key mod prime_number)` to ascertain the next index positions in the hash table. Here, we assume that the prime number less than the table size is 5. This means that the next index position in the hash table will be `(1 + 1*(5 - (22 mod 5))) mod 7`, which is equivalent to 4. So, we store this data element at index position 4.
3. Next, we have data element 29, so we compute the hash value using the primary hashing function, in other words, `(29 mod 7 =1)`. We get a collision, and now we use the secondary hash function to establish the next index position for storing the data element, in other words, `(1 + 1*(5 - (29 mod 5))) mod 7`, which turns out to be 2, so we store this data element at location 2.

The above example of the process of resolving the collision using double hashing is shown in *Figure 8.10*:

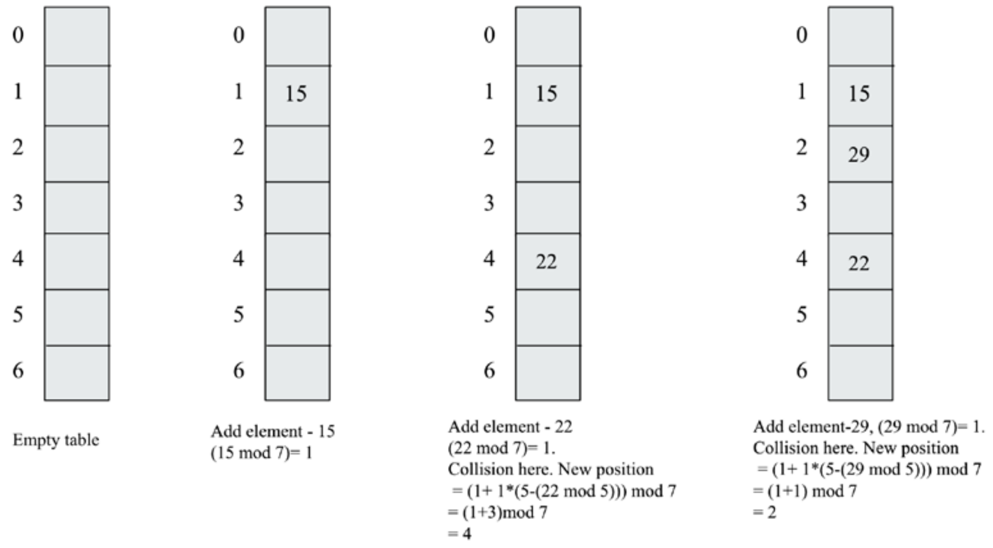

0
1
2
3
4
5
6
Empty table
15
Add element - 15
(15 mod 7)= 1
22
Add element - 22
(22 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(22 mod 5))) mod 7
= (1+3)mod 7
= 4
29
Add element-29, (29 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(29 mod 5))) mod 7
= (1+1) mod 7
= 2

Figure 8.10: Example of collision resolution using double hashing

Let us now see how we can implement the hash table with the double hashing technique to resolve the collision. The `put_double_hashing()` and `get_ double_hashing ()` methods are given as follows, which should be defined in the `HashTable` class.

The following `h2()` method is used to compute the sum of the ordinal values since, in our examples, we have strings as a key element:

```
    def h2(self, key):
        mult = 1
        hv = 0
        for ch in key:
            hv += mult * ord(ch)
            mult += 1
        return hv
```

Furthermore, we should redefine the hash table to include a prime number as a variable that will be used in computing the secondary hash function:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
        self.MAXLOADFACTOR = 0.65
        self.prime_num = 5
```

The following code is designed to insert a data element and associated value in the hash table and use the double hashing technique at the time of collision:

```
    def put_double_hashing(self, key, value):
        item = HashItem(key, value)
        h = self._hash(key)
        j = 1
        while self.slots[h] != None:
            if self.slots[h].key == key:
                break
            h = (h + j * (self.prime_num - (self.h2(key) % self.prime_
num))) % self.size
            j = j+1
        if self.slots[h] == None:
            self.count += 1
        self.slots[h] = item
        self.check_growth()

    def get_double_hashing(self, key):
        h = self._hash(key)
        j = 1
        while self.slots[h] != None:
            if self.slots[h].key == key:
                return self.slots[h].value
            h = (h + j * (self.prime_num - (self.h2(key) % self.prime_
num))) % self.size
            j = j + 1
        return None
```

The above code of the `get_doubleHashing()` and `put_doubleHashing()` methods are very similar to the implementation of the `get()` and `put()` methods that we discussed earlier, except for the statements that are in bold in the preceding codes. The statements in bold are showing that at the time of the collision, we use the double hashing technique formula to get the next empty slot in the hash table:

```
ht = HashTable()
ht.put_doubleHashing("good", "eggs")
ht.put_doubleHashing("better", "spam")
ht.put_doubleHashing("best", "cool")
ht.put_doubleHashing("ad", "donot")
ht.put_doubleHashing("ga", "collide")
ht.put_doubleHashing("awd", "hello")
ht.put_doubleHashing("addition", "ok")

for key in ("good", "better", "best", "worst", "ad", "ga"):
        v = ht.get_doubleHashing(key)
        print(v)
print("The number of elements is: {}".format(ht.count))
```

In the above code, we first insert seven different data elements along with their associated values, and then we search and check a few random data items in the hash table. The output of the preceding code is as follows:

```
eggs
spam
cool
none
donot
collide
The number of elements is: 7
```

In the above output, we can observe that the key string `worst` is not present in the hash table, meaning the output corresponding to this is `None`.

Linear probing leads to primary clustering, while quadratic probing may lead to secondary clustering, whereas the double hashing technique is one of the most effective methods for collision resolution since it does not yield any clusters. The advantage of this technique is that it produces a uniform distribution of records in the hash table.

In open addressing collision resolution techniques, we search for another empty slot within the hash table, as we did in linear probing, quadratic probing, and double hashing. "closed" in "closed hashing" refers to the fact that we do not leave the hash table, and every record is stored at an index position given by the hash function, hence "closed hashing" and "open addressing" are synonyms.

On the other hand, when a record is always stored at an index position given by the hash function, this is known as the "closed addressing," or "open hashing," technique. Here, "open" in "open hashing" refers to the fact that we are open to leaving the hash table through a separate list where the data elements can be stored; for example, separate chaining is a closed addressing technique.

In the next section, we will discuss another collision resolution technique – the chaining technique.

Separate chaining

Separate chaining is another method to handle the problem of collision in hash tables. It solves this problem by allowing each slot in the hash table to store a reference to many items at the position of a collision. So, at the index of a collision, we are allowed to store multiple items in the hash table.

In chaining, the slots in the hash table are initialized with empty lists. When a data element is inserted, it is appended to the list that corresponds to that element's hash value. For example, in the following *Figure 8.11*, there is a collision for the key strings `hello world` and `world hello`. In the case of chaining, both data elements are stored using a list at the index position given by the hash function, in other words, `92` in the example shown in *Figure 8.11*. Here is an example to show collision resolution using chaining:

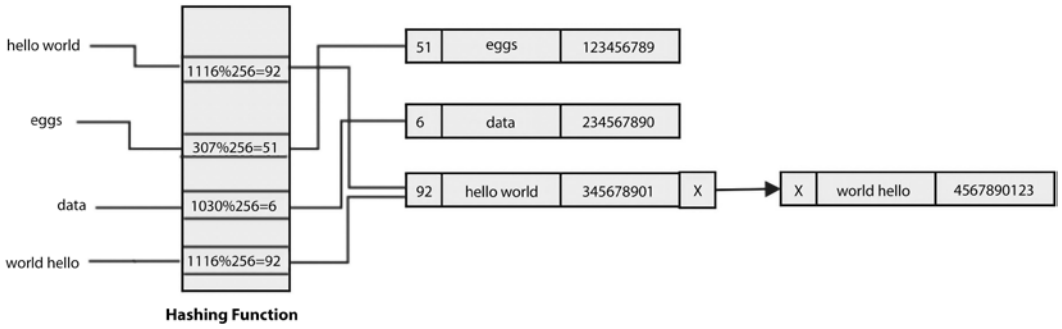


Figure 8.11: Example of collision resolution using chaining

One more example is shown in *Figure 8.12*, wherein if we have many data elements that have a hash value of 51, all of these elements would be added to the list that exists in the same slot of the hash table:

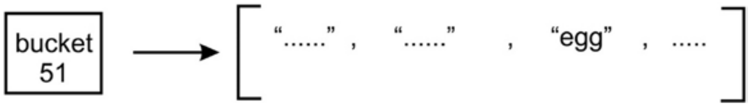


Figure 8.12: More than one element having the same hash value stored in a list

Chaining then avoids conflict by allowing multiple elements to have the same hash value. Hence, there is no limit in terms of the number of elements that can be stored in a hash table, whereas, in the case of open addressing collision resolution techniques, we had to fix the size of the table, which we need to later grow when the table is filled up. Moreover, the hash table can hold more values than the number of available slots, since each slot holds a list that can grow.

However, there is a problem with chaining—it becomes inefficient when a list grows at a particular hash value location. As a particular slot has many items, searching them can become very slow since we have to do a linear search through the list until we find the element that has the key we want. This can slow down retrieval, which is not good since hash tables are meant to be efficient. Hence, the worst-case time complexity for searching in a separate chaining algorithm using linked lists is O(n), because in the worst case, all the items will be added to only one index position in the hash table, and searching an item will work just similar to a linked list. The following *Figure 8.13* demonstrates a linear search through list items until we find a match:

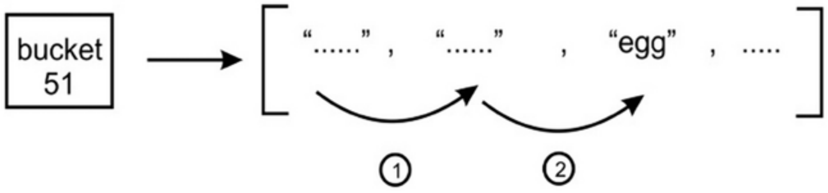


Figure 8.13: Demonstration of a linear search for the hash value of 51

So, there is a problem with the slow retrieval of items when a particular position in a hash table has many entries. This problem can be resolved using another data structure in place of using a list that can perform fast searching and retrieval. There is a nice choice of using **binary search trees** (**BSTs**), which provide fast retrieval, as we discussed in the previous chapter.

We could simply insert an (initially empty) BST into each slot, as shown in the following *Figure 8.14*:

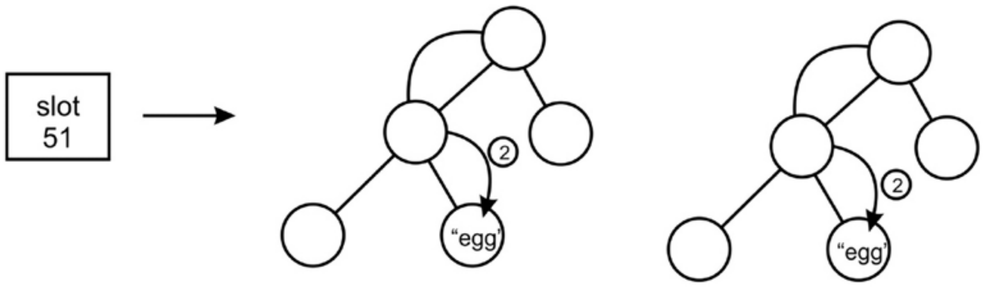


Figure 8.14: BST for a bucket for the hash value of 51

In the preceding diagram, the `51` slot holds a BST, which we use to store and retrieve the data items. However, we would still have a potential problem—depending on the order in which the items were added to the BST, we could end up with a search tree that is as inefficient as a list. That is, each node in the tree has exactly one child. To avoid this, we would need to ensure that our BST is self-balancing.

Here is the implementation of the hash table with separate chaining. Firstly, we create a `Node` class to store the key-value pairs and one pointer for pointing to the next node in the linked list:

```
class Node:
    def __init__(self, key=None, value=None):
        self.key = key
        self.value = value
        self.next = None
```

Next, we define the singly linked list, the details of which are provided in *Chapter 4*, *Linked Lists*. Here, we have defined the `append()` method for adding a new data record to the linked list:

```
class SinglyLinkedList:
    def __init__ (self):
        self.tail = None
        self.head = None

    def append(self, key, value):
        node = Node(key, value)
        if self.tail:
            self.tail.next = node
            self.tail = node        else:
```

```
            self.head = node
            self.tail = node
```

Next, we define the `traverse()` method, which prints all the data records with `key-value` pairs. The `traverse()` method should be defined in the `SinglyLinkedList` class. We start from the head node, and move the next nodes while iterating through the `while` loop:

```
    def traverse(self):
        current = self.head
        while current:
            print("\"", current.key, "--", current.value, "\"")
            current = current.next
```

Next, we define a `search()` method that matches the key that we want to search in the linked list. If the key matches any of the nodes, the corresponding key-value pair is printed. The `search()` method should be defined in the `SinglyLinkedList` class:

```
    def search(self, key):
        current = self.head
        while current:
            if current.key == key:
                print("\"Record found:", current.key, "-", current.value, 
"\"")
                return True
            current = current.next
        return False
```

Once, we have defined the linked list and all the required methods, we define the `HashTableChaining` class, in which we initialize the hash table with its size and all the slots with an empty linked list:

```
class HashTableChaining:
    def __init__(self):
        self.size = 6
        self.slots = [None for i in range(self.size)]
        for x in range(self.size) :
            self.slots[x] = SinglyLinkedList()
```

Next, we define the hash function, in other words, `_hash()`, similar to what we have discussed in previous sections:

```
    def _hash(self, key):
        mult = 1
        hv = 0
        for ch in key:
            hv += mult * ord(ch)
            mult += 1
        return hv % self.size
```

Then, we define the `put()` method to insert a new data record in the hash table. Firstly, we create a node with key-pair pairs and then compute the index position based on the hash function. Then, we append the node at the end of the linked list associated with the given index position. The `put()` method should be defined in the `HashTableChaining` class:

```
    def put(self, key, value):
        node = Node(key, value)
        h = self._hash(key)
        self.slots[h].append(key, value)
```

Next, we define the `get()` method to retrieve the data elements given the key value from the hash table. Firstly, we compute the index position using the same hash function that we used at the time of adding the records to the hash table, and then we search the required data record in the linked list associated with the given index position computed. The `get()` method should be defined in the `HashTableChaining` class:

```
    def get(self, key):
        h = self._hash(key)
        v = self.slots[h].search(key)

```

Finally, we can define the `printHashTable()` method, which prints the complete hash table showing all the records of the hash table:

```
    def printHashTable(self) :
        print("Hash table is :- \n")
        print("Index \t\tValues\n")
        for x in range(self.size) :
            print(x,end="\t\n")
            self.slots[x].traverse()
```

We can use the following code to insert a few sample data records in the hash table and we use the chaining technique to store the data. Then, we search a data record with the key string best, and we also print the complete hash table:

```
ht = HashTableChaining()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")
ht.put("awd", "do not")

ht.printHashTable()
```

The output of the preceding code is as follows:

```
Hash table is :-
Index              Values
0
1
2
" good - eggs "
3
" better - ham "
" ad - do not "
" ga - collide "
4
5
" best - spam "
" awd - do not "
```

The above output shows how all the data records are stored at each index position in the hash table. We can observe that multiple data records are stored at the same index position given by the hash function.

Hash tables are important data structures for storing data in key-value pairs, and we can use any of the collision resolution techniques, in other words, open addressing or separate chaining. Open addressing techniques are very fast when the keys are uniformly distributed in the hash table, but there is a possible complication of cluster formation.

The separate chaining technique does not have the problem of clustering, but it may become slower when all the data records are hashed to a very few index positions in the hash table.

Symbol tables

Symbol tables are used by compilers and interpreters to keep track of the symbols and different entities, such as objects, classes, variables, and function names, that have been declared in a program. Symbol tables are often built using hash tables since it is important to efficiently retrieve a symbol from the table.

Let's look at an example. Suppose we have the following Python code in the `symb.py` file:

```
name = "Joe"
age = 27
```

Here, we have two symbols, `name` and `age`. Each symbol has a value; for example, the `name` symbol has the value `Joe`, and the `age` symbol has the value `27`. A symbol table allows the compiler or the interpreter to look up these values. So, the `name` and `age` symbols become keys in the hash table. All of the other information associated with them becomes the `value` of the symbol table entry.

In compilers, symbol tables can have other symbols as well, such as functions and class names. For example, the `greet()` function and two variables, in other words, `name` and `age`, are stored in the symbol table as shown in *Figure 8.15*:

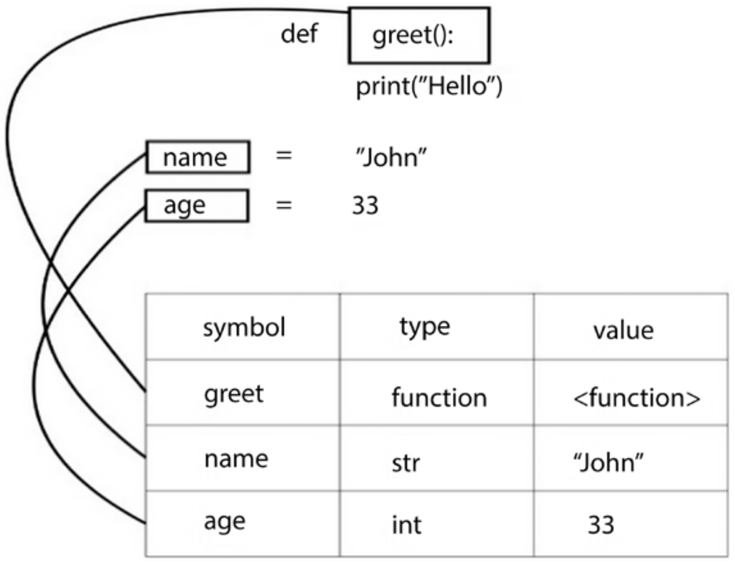


Figure 8.15: Example of a symbol table

The compiler creates a symbol table for each of its modules that are loaded in memory at the time of its execution. Symbol tables are one of the important applications of hash tables, which are mostly used in the compilers and interpreters to efficiently store and retrieve the symbols and associated values.

Summary

In this chapter, we discussed hashing techniques and the data structure of hash tables. We learned about the implementation and concepts of different operations performed on hash tables. We also discussed several collision resolution techniques, including open addressing techniques, namely, linear probing, quadratic probing, and double hashing. Furthermore, we discussed another kind of collision resolution method – separate chaining. Finally, we looked at symbol tables, which are often built using hash tables. Symbol tables allow a compiler or an interpreter to look up a symbol (such as a variable, function, or class) that has been defined and retrieve all the information about it. In the next chapter, we will discuss graph algorithms in detail.

Exercise

1. There is a hash table with 40 slots and there are 200 elements stored in the table. What will be the load factor of the hash table?
2. What is the worst-case search time of hashing using a separate chaining algorithm?
3. Assume a uniform distribution of keys in the hash table. What will be the time complexities for the Search/Insert/Delete operations?
4. What will be the worst-case complexity for removing duplicate characters from an array of characters?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`