CHAPTER 21

Hashing: Implementing Dictionaries and Sets

Objectives

- To know what hashing is for (§21.2).
- To use the hash function to obtain a hash code (§21.2).
- To handle collisions using open addressing (§21.3).
- To know the differences among linear probing, quadratic probing, and double hashing (§21.3).
- To handle collisions using separate chaining (§21.4).
- To understand the load factor and the need for rehashing (§21.5).
- To implement Map using hashing (§21.6).
- To implement Set using hashing (§21.7).
21.1 Introduction

Key Point: Hashing is super efficient. It takes \( O(1) \) time to search, insert, and delete an element using hashing.

The preceding chapters introduced search trees. An element can be found in \( O(\log n) \) time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called hashing. You can use hashing to implement a map or a set to search, insert, and delete an element in \( O(1) \) time.

Recall that a map is a data structure that stores entries. Each entry contains two parts: key and value. The key is also called a search key, which is used to search for the corresponding value.

NOTE:
A map is also called a dictionary, a hash table, or an associative array.

In this chapter, you will learn the concept of hashing and use it to implement a map and a set.

21.2 What is Hashing?

Key Point: Hashing uses a hashing function to map a key to an index.

If you know the index of an element in the list, you can retrieve the element using the index in \( O(1) \) time.

So, can we store the values in a list and use the key as the index to find the value? The answer is yes—if you can map a key to an index. The list that stores the values is called a hash table. The function that maps a key to an index in the hash table is called a hash function. As shown in Figure 21.1, a hash function obtains an index from a key and uses the index to retrieve the value for the key. Hashing is a technique that retrieves the value using the index obtained from the key without performing a search.
A hash function maps a key to an index in the hash table.

How do you design a hash function that produces an index from a key? Python provides a built-in hash function that returns an integer hash code for an object. For example,

```
Sample output
>>> hash(35)
35
>>> hash(3.5))
1073741827
>>> hash("Welcome")
-1542878549
```

This function is well-designed to minimize the possibility that two different objects have the same value. You can use `hash(key) % N` to map the object to an index in the hash table, where N is the size of the table.

Ideally, we would like to design a function that maps each search key to a different index in the hash table. Such a function is called a perfect hash function. However, it is difficult to find a perfect hash function. When two or more keys are mapped to the same index, we say that a collision has occurred. We now discuss how to deal with collisions.

### 21.3 Handling Collisions Using Open Addressing

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

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

### 21.3.1 Linear Probing

When a collision occurs during the insertion of an entry to a hash table, linear probing finds the next available location sequentially. For example, if a collision occurs at \(\text{hashTable}[k \% N]\), check whether \(\text{hashTable}[(k+1) \% N]\) is available. If not, check \(\text{hashTable}[(k+2) \% N]\) and so on, until an available cell is found, as shown in Figure 21.2.

**Figure 21.2**

Linear probe finds the next available location sequentially.

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

**NOTE:**

When probing reaches the end of the table, it goes back to the beginning of the table.

Thus, the hash table is treated as if it were circular.
To search for an entry in the hash table, obtain the index, say $k$, from the hash function for the key. Check whether $hashTable[k \mod n]$ contains the entry. If not, check whether $hashTable[(k+1) \mod n]$ contains the entry, and so on, until it is found, or an empty cell is reached.

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

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

Pedagogical NOTE

Follow the link

www.cs.armstrong.edu/liang/animation/HashingLinearProbing

Animation.html to see how to hashing with linear probing works, as shown in Figure 21.3.
21.3.2 Quadratic Probing

Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index $k$. Quadratic probing, on the other hand, looks at the cells at indices $(k + j^2) \mod n$, for $j \geq 0$, i.e., $k$, $(k + 1) \mod n$, $(k + 4) \mod n$, $(k + 9) \mod n$, ..., and so on, as shown in Figure 21.4.
Quadratic probing works in the same way as linear probing except for the change of search sequence. Quadratic probing avoids the clustering problem in linear probing, but it has its own clustering problem, called secondary clustering; i.e., the entries that collide with an occupied entry use the same probe sequence.

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

21.3.3 Double Hashing

Another open addressing scheme that avoids the clustering problem is known as double hashing. Starting from the initial index k, both linear probing and quadratic probing add an increment to k to define a search sequence. The increment is 1 for linear probing and \( j^2 \) for quadratic probing. These increments are independent of the keys. Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

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

![Figure 21.4](image)

**Figure 21.4**

*Quadratic probe increases the next index in the sequence by \( j^2 \) for \( j = 1, 2, 3, \ldots \).*
\[ h(k) = k \% 11 \]
\[ h'(k) = 7 - k \% 7 \]

For a search key of 12, we have

\[ h(12) = 12 \% 11 = 1 \]
\[ h'(k) = 7 - 12 \% 7 = 2 \]

The probe sequence for key 12 starts at index 1 with an increment 2, as shown in Figure 21.5.

Figure 21.5

The secondary hash function in a double hashing determines the increment of the next index in the probe sequence.

The indices of the probe sequence are as follows: 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10. This sequence reaches the entire table. You should design your functions to produce the probe sequence that reaches the entire table.

Note that the second function should never have a zero value, since zero is not an increment.

Check point
21.1 What is a hash function? What is a perfect hash function? What is a collision?

21.2 What is a hash code? What is the hash code for an integer in Python?

21.3 How is a hash code compressed to an integer representing the index in a hash table?

21.4 What is open addressing? What is linear probing? What is quadratic probing? What is double hashing?

21.5 Describe the clustering problem for linear probing.

21.6 What is the secondary clustering?

21.7 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using linear probing.

21.8 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using quadratic probing.

21.9 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using double hashing with the following functions:

\[
\begin{align*}
h(k) &= k \mod 11 \\
h'(k) &= 7 - k \mod 7
\end{align*}
\]

21.10 Suppose the size of the table is 10. What is the probe sequence for a key 12 using the following double hashing functions?

\[
\begin{align*}
h(k) &= k \mod 10 \\
h'(k) &= 7 - k \mod 7
\end{align*}
\]

21.4 Handling Collisions Using Separate Chaining
Key Point: The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a bucket. A bucket is a container that holds multiple entries.

The preceding section introduced handling collisions using open addressing. The open addressing scheme finds a new location when a collision occurs. This section introduces handling collisions using separate chaining. The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a bucket. A bucket is a container that holds multiple entries.

You may implement a bucket using a list or a LinkedList. We will use LinkedList for demonstration. You can view each cell in the hash table as the reference to the head of a linked list, and elements in the linked list are chained starting from the head, as shown in Figure 21.6.

![Figure 21.6](image)

*Figure 21.6*

Separate chaining chains the entries with the same hash index in a bucket.

### 21.5 Load Factor and Rehashing
Key Point: *Load factor measures how full the hash table is. If the load factor is exceeded, increase the hash table and reload the entries into the new hash table. This is called rehashing.*

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

Note that $\lambda$ is zero if the map is empty. For the open addressing scheme, $\lambda$ is between 0 and 1; $\lambda$ is 1 if the hash table is full. For the separate chaining scheme, $\lambda$ can be any value. As $\lambda$ increases, the probability of collision increases. Studies show that you should maintain the load factor under 0.5 for the open addressing scheme and under 0.9 for the separate chaining scheme.

Keeping the load factor under a certain threshold is important for the performance of hashing. Typically, the threshold 0.75 is recommended. Whenever the load factor exceeds the threshold, you need to increase the hash-table size and rehash all the entries in the map to the new hash table. Notice that you need to change the hash functions, since the hash-table size has been changed. To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even with periodic rehashing, hashing is an efficient implementation for map.

Pedagogical NOTE

Follow the link


...to see how to hashing with linear probing works, as shown in Figure 21.7.
Figure 21.7

The animation tool shows how separate chaining works.

Check point

21.11 Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using separate chaining.

21.12 What is load factor?

21.6 Implementing a Map Using Hashing

Key Point: A map can be implemented using hashing.

Now you know the concept of hashing. You know how to use a hash function to map a key to an index in a hash table, how to measure performance using the load factor, and how to increase the table size and rehash to maintain the performance. This section demonstrates how to implement a map using separate chaining.
We design our custom `Map` class to mirror Python’s dictionary class with some minor variations. The `Map` class is defined in Figure 21.8.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Map()</code></td>
<td>Constructs an empty map.</td>
</tr>
<tr>
<td><code>put(key: object, value: object)</code>: None</td>
<td>Adds an entry to this map.</td>
</tr>
<tr>
<td><code>remove(key: object)</code>: None</td>
<td>Removes an entry for the specified key.</td>
</tr>
<tr>
<td><code>containsKey(key: object)</code>: bool</td>
<td>Returns true if this map contains an entry for the specified key.</td>
</tr>
<tr>
<td><code>containsValue(value: object)</code>: bool</td>
<td>Returns true if this map maps one or more keys to the specified value.</td>
</tr>
<tr>
<td><code>items()</code>: list</td>
<td>Returns a list consisting of the entries in this map.</td>
</tr>
<tr>
<td><code>get(key: object)</code>: V</td>
<td>Returns a value for the specified key in this map.</td>
</tr>
<tr>
<td><code>getAll(key: object)</code>: list</td>
<td>Returns all values for the specified key in this map.</td>
</tr>
<tr>
<td><code>keys()</code>: list</td>
<td>Returns a list consisting of the keys in this map.</td>
</tr>
<tr>
<td><code>values()</code>: list</td>
<td>Returns a list consisting of the values in this map.</td>
</tr>
<tr>
<td><code>clear()</code>: None</td>
<td>Removes all entries from this map.</td>
</tr>
<tr>
<td><code>getSize()</code>: int</td>
<td>Returns the number of mappings in this map.</td>
</tr>
<tr>
<td><code>isEmpty()</code>: bool</td>
<td>Returns true if this map contains no mappings.</td>
</tr>
<tr>
<td><code>toString()</code>: str</td>
<td>Returns the hash table as a string.</td>
</tr>
<tr>
<td><code>setLoadFactorThreshold(threshold: int)</code>: None</td>
<td>Sets a new load factor threshold.</td>
</tr>
<tr>
<td><code>toString()</code>: str</td>
<td>Returns the entries in the map as a string.</td>
</tr>
<tr>
<td><code>getTable()</code>: str</td>
<td>Returns the internal hash table as a string.</td>
</tr>
</tbody>
</table>

The number of entries in the map. Each element in the list is a bucket to hold a list of entries.

Figure 21.8

The `Map` class stores key/value pairs.

In Python’s dictionary class, the keys are distinct. However, a map may allow duplicate keys. Our map interface allows duplicate keys. The `get(key)` method gets one of the values that match the key. The `getAll(key)` method retrieves all values that match the key. In the exercise, you will revise the class to disallow duplicate keys.

How do you implement `Map`? If you use an `list` and store a new entry at the end of the list, the search time will be $O(n)$. If you implement `Map` using an AVL tree, the search time will be $O(\log n)$. 
Nevertheless, you can implement `Map` using hashing to obtain an $O(1)$ time search algorithm. Listing 21.1 implements the `Map` class using separate chaining.

**Listing 21.1 Map.py**

```python
# Define the default hash-table size
DEFAULT_INITIAL_CAPACITY = 4

# Define default load factor
DEFAULT_MAX_LOAD_FACTOR = 0.75

# Define the maximum hash-table size to be 2 ** 30
MAXIMUM_CAPACITY = 2 ** 30

class Map:
    def __init__(self, capacity = DEFAULT_INITIAL_CAPACITY,
                 loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR):
        # Current hash-table capacity. Capacity is a power of 2
        self.capacity = capacity

        # Specify a load factor used in the hash table
        self.loadFactorThreshold = loadFactorThreshold

        # Create a list of empty buckets
        self.table = []
        for i in range(self.capacity):
            self.table.append([])

        self.size = 0  # Initialize map size

    # Add an entry (key, value) into the map
    def put(self, key, value):
        if self.size >= self.capacity * self.loadFactorThreshold:
            if self.capacity == MAXIMUM_CAPACITY:
                raise RuntimeError("Exceeding maximum capacity")

            self.rehash()

        bucketIndex = hash(key) % self.capacity

        # Add an entry (key, value) to hashTable[index]
        self.table[bucketIndex].append([key, value])

        self.size += 1  # Increase size

    # Remove the entry for the specified key
    def remove(self, key):
        bucketIndex = hash(key) % self.capacity

        # Remove the first entry that matches the key from a bucket
        if len(self.table[bucketIndex]) > 0:
            bucket = self.table[bucketIndex]

            for entry in bucket:
                if entry[0] == key:
                    bucket.remove(entry)

                    self.size -= 1  # Decrease size
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
break  # Remove just one entry that matches the key

# Return true if the specified key is in the map
def containsKey(self, key):
    if self.get(key) != None:
        return True
    else:
        return False

# Return true if this map contains the specified value
def containsValue(self, value):
    for i in range(self.capacity):
        if len(self.table[i]) > 0:
            bucket = self.table[i]
            for entry in bucket:
                if entry[1] == value:
                    return True
    return False

# Return a set of entries in the map
def items(self):
    entries = []
    for i in range(self.capacity):
        if self.table[i] != None:
            bucket = self.table[i]
            for entry in bucket:
                entries.append(entry)
    return tuple(entries)

# Return the first value that matches the specified key
def get(self, key):
    bucketIndex = hash(key) % self.capacity
    if len(self.table[bucketIndex]) > 0:
        bucket = self.table[bucketIndex]
        for entry in bucket:
            if entry[0] == key:
                return entry[1]
    return None

# Return all values for the specified key in this map
def getAll(self, key):
    values = []
    bucketIndex = hash(key) % self.capacity
    if len(self.table[bucketIndex]) > 0:
        bucket = self.table[bucketIndex]
        for entry in bucket:
            if entry[0] == key:
                values.append(entry[1])
    return tuple(values)

# Return a set consisting of the keys in this map
def keys(self):
    keys = []
    for i in range(0, self.capacity):
        if len(self.table[i]) > 0:
            bucket = self.table[i]
            for entry in bucket:
keys.append(entry[0])

    return keys

# Return a set consisting of the values in this map
def values(self):
    values = []
    for i in range(self.capacity):
        if len(self.table[i]) > 0:
            bucket = self.table[i]
            for entry in bucket:
                values.append(entry[1])
    return values

# Remove all of the entries from this map
def clear(self):
    self.size = 0 # Reset map size
    self.table = [] # Reset map
    for i in range(self.capacity):
        self.table.append([])

# Return the number of mappings in this map
def getSize(self):
    return self.size

# Return true if this map contains no entries
def isEmpty(self):
    size == 0

# Rehash the map
def rehash(self):
    temp = self.items() # Get entries
    self.capacity *= 2 # Double capacity
    self.table = [] # Create a new hash table
    self.size = 0 # Clear size
    for i in range(self.capacity):
        self.table.append([])

    for entry in temp:
        self.put(entry[0], entry[1]) # Store to new table

# Return the entries as a string
def str_(self):
    return str(self.items())

# Return a string representation for this map
def setLoadFactorThreshold(self, threshold):
    self.loadFactorThreshold = threshold

# Return the hash table as a string
def getTable(self):
    return str(self.table)

The Map class is implemented using a hash table. The default initial size, default max load
factor, and max table capacity are defined in lines 1-8.
The constructor creates a Map object with the specified initial capacity and load factor threshold (lines 14-17). The hash table is initialized with empty buckets (lines 20-22). Each bucket is a list to hold entries.

The put(key, value) method adds a new entry into the map. The method first checks whether the size exceeds the load-factor threshold (line 8). If so, invoke rehash() (line 32) to increase the capacity and store entries into the new hash table.

The remove(key) method removes all entries with the specified key in the map (lines 42–52). This method takes $O(1)$ time.

The get(key) method (lines 84-90) returns the value of the first entry with the specified key. The method first locates the bucket index (line 85) and then checks if there is any entry with the matching key in the bucket. If so, the first such element is returned. This method takes $O(1)$ time, since the individual bucket size is typically small.

The getAll(key) method returns the value of all entries with the specified key (lines 95-106). Similar to the get method, this method takes $O(1)$ time.

The containsKey(key) method checks whether the specified key is in the map by invoking the get method (line 56). Since the get method takes O(1) time, the containsKey(key) method takes $O(1)$ time.

The containsValue(value) method checks whether the value is in the map (lines 62-70). This method takes $O(capacity + size)$ time. It is actually $O(capacity)$, since $capacity > size$. 
The \texttt{items()} method returns a list that contains all entries in the map (lines 73-81). This method takes $O(capacity)$ time.

The \texttt{keys()} method returns all keys in the map as a set. The method finds the keys from each bucket and adds them to a list (lines 107-116). This method takes $O(capacity)$ time.

The \texttt{values()} method returns all values in the map. The method examines each entry from all buckets and adds it to a list (lines 119–127). This method takes $O(capacity)$ time.

The \texttt{clear} method removes all entries from the map (lines 130-135). It simply resets the table with empty buckets. The old hash table becomes garbage and will be automatically collected by Python runtime system. This method takes $O(capacity)$ time.

The \texttt{isEmpty()} method simply returns true if the map is empty (lines 120-122). This method takes $O(1)$ time.

The \texttt{getSize()} method simply returns the size of the map (lines 138–139). This method takes $O(1)$ time.

The \texttt{rehash()} method first copies all entries in a temporary list (line 147), doubles the capacity (line 148), resets the size (line 150), and creates a new hash table with empty buckets (lines 151-152). The method then copies the entries into the new hash table (lines 154–155). The \texttt{rehash} method takes $O(capacity)$ time. If no rehash is performed, the \texttt{put} method takes $O(1)$ time to add a new entry.

Table 21.1 summarizes the time complexities of the methods in \texttt{Map}.
Table 21.1

Time Complexities for Methods in Map

<table>
<thead>
<tr>
<th>Methods</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>containsKey(key: Key)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>containsValue(value: V)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>items()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get(key: K)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getAll(key: K)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>keys()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>put(key: K, value: V)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(key: K)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getSize()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>values()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>rehash()</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Since rehashing does not happen very often, the time complexity for the put method is $O(1)$.

Note that the complexities of the clear, items, keys, values, and rehash methods depend on capacity, so to avoid poor performance for these methods you should choose an initial capacity carefully.

Listing 21.2 gives a test program that uses Map.

Listing 21.2 TestMap.py

```python
from Map import Map

def main():
    # Create a map
    map = Map()
    map.put("Smith", 30)
    map.put("Anderson", 31)
    map.put("Lewis", 29)
    map.put("Cook", 29)
    map.put("Cook", 129)
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
print("Entry set in map: " + str(map.items()))
print("The age for Lewis is " + str(map.get("Lewis")))
print("Is Smith in the map? " + str(map.containsKey("Smith")))
print("Is Johnson in the map? " + str(map.containsKey("Johnson")))
print("Is value 30 in the map? " + str(map.containsValue(30)))
print("Is value 33 in the map? " + str(map.containsValue(33)))
print("Is age 33 in the map? " + str(map.containsValue(33)))
print("All values for Cook? " + str(map.getAll("Cook")))
print("Keys are " + str(map.keys()))
print("Values are " + str(map.values()))
map.remove("Smith")
print("The map is " + map.getTable())
map.clear()
print("The map is " + map.getTable())
main()

Sample output
Entries in map: [[Anderson, 31], [Smith, 30], [Lewis, 29], [Cook, 29]]
Entry set in map: (['Cook', 29], ['Cook', 129], ['Lewis', 29], ['Anderson', 31], ['Smith', 30])
The age for Lewis is 29
Is Smith in the map? True
Is Johnson in the map? False
Is value 30 in the map? True
Is value 33 in the map? False
Is age 33 in the map? False
All values for Cook? (29, 129)
keys are ['Cook', 'Cook', 'Lewis', 'Anderson', 'Smith']
values are [29, 129, 29, 31, 30]
The map is [[['Cook', 29], ['Cook', 129]], [['Lewis', 29]], [['Anderson', 31]], [], [], [], [], []]
The map is [[]], [], [], [], [], [], [], []

The program creates a map using Map (line 5), adds entries to the map (lines 6–10), displays the entries (line 12), gets a value for a key (line 13), checks whether the map contains the key (line 14) and a value (line 17), removes an entry with the key “Smith” (line 24), and display the hash table (line 25). The program removes all entries (line 24) and redisplays the hash table (line 28).

21.7 Implementing Set Using Hashing

Key Point: A hash set can be implemented using a hash map.
A set is a data structure that stores distinct values. Python supports the set type. This section discusses how to implement a Set class. The Set class can be implemented using the same approach for implementing Map. The only difference is that key/value pairs are stored in the map, while elements are stored in the set.

We design our Set class to mirror Python’s set type with some minor variations, as shown in Figure 21.9.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: int</td>
<td>The number of elements in the set.</td>
</tr>
<tr>
<td>table: list</td>
<td>The hash table for storing set elements.</td>
</tr>
<tr>
<td>Set()</td>
<td>Creates an empty set.</td>
</tr>
<tr>
<td>add(e: object): bool</td>
<td>Adds the element to the set and returns true if the element is added successfully.</td>
</tr>
<tr>
<td>remove(e: object): bool</td>
<td>Removes the element from the set and returns true if the set contained the element.</td>
</tr>
<tr>
<td>clear(): void</td>
<td>Removes all elements from this set.</td>
</tr>
<tr>
<td>contains(e: object): bool</td>
<td>Returns true if the element is in the set.</td>
</tr>
<tr>
<td>isEmpty(): bool</td>
<td>Returns true if this set contains no elements.</td>
</tr>
<tr>
<td>getSize(): int</td>
<td>Returns the number of elements in this set.</td>
</tr>
<tr>
<td>union(s: Set): Set</td>
<td>Set union.</td>
</tr>
<tr>
<td>difference(s: Set): Set</td>
<td>Set difference.</td>
</tr>
<tr>
<td>intersect(s: Set): Set</td>
<td>Set intersection.</td>
</tr>
<tr>
<td>toString(): str</td>
<td>Returns a string representation for the set.</td>
</tr>
<tr>
<td>getTable(): str</td>
<td>Returns the internal hash table as a string.</td>
</tr>
</tbody>
</table>

Figure 21.9

The Set class defines a set.

Listing 21.3 implements the Set class.

Listing 21.3 Set.py

```python
1  # Define the default hash-table size
2  DEFAULT_INITIAL_CAPACITY = 4
3
4  # Define default load factor
5  DEFAULT_MAX_LOAD_FACTOR = 0.75
6
7  # Define the maximum hash-table size to be 2 ** 30
8  MAXIMUM_CAPACITY = 2 ** 30
9
class Set:
10  def __init__(self, capacity = DEFAULT_INITIAL_CAPACITY,
11                  loadFactorThreshold = DEFAULT_MAX_LOAD_FACTOR):
12      # Current hash-table capacity. Capacity is a power of 2
13      self.capacity = capacity
14
15      # Specify a load factor used in the hash table
16      self.loadFactorThreshold = loadFactorThreshold
17
18
19  ```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
# Create a list of empty buckets
self.table = []
for i in range(self.capacity):
    self.table.append([])

self.size = 0  # Initialize set size

# Add an entry (key, value) into the map
def add(self, key):
    if self.size >= self.capacity * self.loadFactorThreshold:
        if self.capacity == MAXIMUM_CAPACITY:
            raise RuntimeError("Exceeding maximum capacity")

        self.rehash()

    bucketIndex = hash(key) % self.capacity

    # Add an entry (key, value) to hashTable[index]
    if not (key in self.table[bucketIndex]):
        self.table[bucketIndex].append(key)
        self.size += 1  # Increase size

# Remove the entry for the specified key
def remove(self, key):
    bucketIndex = hash(key) % self.capacity

    # Remove the first entry that matches the key from a bucket
    if len(self.table[bucketIndex]) > 0:
        bucket = self.table[bucketIndex]
        for e in bucket:
            if e == key:
                bucket.remove(e)
                self.size -= 1  # Decrease size
                break  # Remove just one entry that matches the key

    # Return true if the specified key is in the map
    def contains(self, key):
        if self.get(key) != None:
            return True
        else:
            return False

    # Return the first value that matches the specified key
    def get(self, key):
        bucketIndex = hash(key) % self.capacity
        if len(self.table[bucketIndex]) > 0:
            bucket = self.table[bucketIndex]
            for e in bucket:
                if e == key:
                    return e

        return None

# Return all keys in a list
def keys(self):
    keys = []

    for i in range(self.capacity):
        if len(self.table[i]) > 0:
            bucket = self.table[i]
            for e in bucket:
                keys.append(e)

    return keys

    # Return a string representation for the keys in this set

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.
def __str__(self):
    return str(self.keys())

def clear(self):
    self.size = 0 # Reset map size
    self.table = [] # Reset map
    for i in range(self.capacity):
        self.table.append([])

# Return the number of mappings in this map
def getSize(self):
    return self.size

# Return true if this map contains no entries
def isEmpty(self):
    return size == 0

# Rehash the map
def rehash(self):
    temp = self.keys() # Get elements
    self.capacity *= 2 # Double capacity
    self.table = [] # Create a new hash table
    self.size = 0 # Clear size
    for i in range(self.capacity):
        self.table.append([])

    for e in temp:
        self.add(e) # Store to new table

# Return the hash table as a string
def getTable(self):
    return str(self.table)

# Return a string representation for this map
def setLoadFactorThreshold(self, threshold):
    self.loadFactorThreshold = threshold

# The union, difference, and intersect methods are left as exercise

Implementing Set is very similar to implementing Map except for the following differences:

1. The elements are stored in the hash table for Set, but the entries (key/value pairs) are stored in the hash table for Map.

2. The elements are all distinct in Set, but two entries may have the same keys in Map.

The program defines constants for default initial capacity, default max load factor, and maximum capacity (lines 1-8).
The constructor creates a set with the specified capacity and load factor threshold. If these parameters are not given, the default values are used (lines 13-17). A hash table is created with empty buckets (lines 20-24). Each bucket will hold the keys with the same hash code index.

The `add(element)` method (lines 27-39) adds a new element into the set. The method first checks whether the size exceeds the load-factor threshold (line 28). If so, invoke `rehash()` (line 32) to increase the capacity and store entries into the new hash table.

The `remove(element)` method removes the specified element in the set (lines 42–52). This method takes $O(1)$ time.

The `contains(element)` method checks whether the specified element is in the set by examining whether the designated bucket contains the element (lines 55–59). This method takes $O(1)$ time.

The `clear` method removes all entries from the map (lines 89–94). It resets size and creates a new hash table with empty buckets.

The `rehash()` method (lines 105-114) first copies all elements in a list (line 106), doubles the capacity (line 107), and creates a new hash table with empty buckets (lines 110-111), and clears the size (line 109). The method then copies the entries into the new hash table (lines 113–114). The `rehash` method takes $O(capacity)$ time. If no rehash is performed, the `add` method takes $O(1)$ time to add a new element. Since rehash does not happen very often, the `add` method takes $O(1)$ time.

The `getSize()` method simply returns the size of the set (lines 97–98). This method takes $O(1)$ time.
Table 21.2 summarizes the time complexity of the methods in Set.

Table 21.2
Time Complexities for Methods in Set

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear()</td>
<td>$O(\text{capacity})$</td>
</tr>
<tr>
<td>contains(e: E)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add(e: E)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(e: E)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>size()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>iterator()</td>
<td>$O(\text{capacity})$</td>
</tr>
<tr>
<td>rehash()</td>
<td>$O(\text{capacity})$</td>
</tr>
</tbody>
</table>

Listing 21.4 gives a test program that uses Set.

Listing 21.4 TestSet.py

```python
1  from Set import Set
2
3  set = Set()  # Create an empty set
4  set.add(45)
5  set.add(13)
6  set.add(43)
7  set.add(43)
8  set.add(1)
9  set.add(2)
10
11  print("Elements in set: "+ str(set))
12  print("Number of elements in set: "+ str(set.getSize()))
13  print("Is 1 in set? "+ str(set.contains(1)))
14  print("Is 11 in set? "+ str(set.contains(11)))
15
16  set.remove(2)
17  print("After deleting 2, the set is "+ str(set))
18  print("The internal table for set is "+ set.getTable())
19
20  set.clear()
21  print("After deleting all elements")
22  print("The internal table for set is "+ set.getTable())
```

Sample output
Elements in set: [1, 2, 43, 45, 13]
Number of elements in set: 5
Is 1 in set? True
Is 11 in set? False
After deleting 2, the set is [1, 43, 45, 13]
The internal table for set is [[]], [1], [], [43], [], [45, 13], [], []
After deleting all elements
The internal table for set is [[]], [[]], [[]], [[]], [[]], [[]], [[]], [[]]

The program creates a set (line 3), adds elements to the set (lines 4–9), displays the elements (line 11) and the set size (line 12), checks whether the set contains the element (lines 13-14), removes an element (line 16), and clears the set (line 20). Note that the elements in the set are unique. So, when you add 43 twice (lines 6-7), only one is stored in the set.

Key Terms

- associative array
- clustering
- dictionary
- double hashing
- hash code
- hash function
- hash map
- hash set
- hash table
- linear probing
- load factor
- open addressing
- perfect hash function
- rehashing
- separate chaining

Chapter Summary
1. A map is a data structure that stores entries. Each entry contains two parts: key and value. The key is also called a search key, which is used to search for the corresponding value. You can implement a map to obtain \(O(1)\) time complexity on search, retrieval, insertion, and deletion, using the hashing technique.

2. A set is a data structure that stores elements. You can use the hashing technique to implement a set to achieve \(O(1)\) time complexity on search, insertion, and deletion for a set.

3. Hashing is a technique that retrieves the value using the index obtained from key without performing a search. A typical hash function first converts a search key to an integer value called a hash code, then compresses the hash code into an index to the hash table.

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

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

6. The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a bucket. A bucket is a container that holds multiple entries.

Multiple-Choice Questions

See multiple-choice questions online at

liang.armstrong.edu/selftest/selftestpy?chapter=21.

Programming Exercises
**21.1 (Implement Map using open addressing with linear probing) Implement the Map class using open addressing with linear probing. For simplicity, use \( f(key) = key \mod size \) as the hash function, where size is the hash-table size. Initially, the hash-table size is 4. The table size is doubled whenever the load factor exceeds the threshold (0.5).

**21.2 (Implement Map using open addressing with quadratic probing) Implement Map using open addressing with quadratic probing. For simplicity, use \( f(key) = key \mod size \) as the hash function, where size is the hash-table size. Initially, the hash-table size is 4. The table size is doubled whenever the load factor exceeds the threshold (0.5).

**21.3 (Implement Map using open addressing with double hashing) Implement Map using open addressing with double probing. For simplicity, use \( f(key) = key \mod size \) as the hash function, where size is the hash-table size. Initially, the hash-table size is 4. The table size is doubled whenever the load factor exceeds the threshold (0.5).

**21.4 (Modify Map with distinct keys) Modify Map so that all entries in it have different keys.

**21.5 (Implement Set) Implement the union, difference, and intersect methods in the Set class.

**21.6 (Animating linear probing) Write a program that animates linear probing as shown in Figure 21.3. You can change the initial size of the hash-table in the applet. Assume the load-factor threshold is 0.75.

**21.7 (Animating separate chaining) Write a program that animates Map as shown in Figure 21.7. You can change the initial size of the table. Assume the load-factor threshold is 0.75.