Module 11

Sets, Maps, and Hashtables

Objectives:

1. Understand the concept of the Dictionary structure, and the notion of key-value pairs.
2. Understand the differences between the structures, set, map, and hashtable.
3. Know what a hashing function does, and some of the ways to create a hashing function.
4. Understand what collisions are, and three ways they can be resolved: 1) linear probing, 2) quadratic probing, and 3) separate chaining hashing
5. Be able to apply the three collision resolution mechanisms listed above to an actual problem.

Reading:

Chapter 20 (Dictionaries)

(Optional) Chapter 21 (Dictionary Implementations)

Chapter 22 (Hashing)

Chapter 23 (Hashing as a Dictionary Implementation)

Assignment:

Complete the assignment for Module 11.

Quiz:

Take the quiz for Module 11.

Lab:

No lab for this module.

Dictionaries

The dictionary is a type of data structure in which each item, or **value**, is associated with a **key**. Thus, in a dictionary entries consist of a collection of key-value pairs. A specific item is retrieved by searching for its key rather than by searching for the item itself. In this module we'll talk about three types of dictionary data structures: 1) hashtables, 2) sets, and 3) maps. I will also introduce you to the map and set classes provided by the JDK library.

Hashtables

Hashtables are dictionaries in which the key for an item is an array index (typically of type `int`), which is derived from the item itself using what is known as a **hashing function**. Hashtables use arrays internally to store data, so typically the running time to add, retrieve, or remove an item is O(1). Since a hashing function is not always guaranteed to return a value that falls within the valid range of array indexes for the internal array, the hashcodes must be mapped to valid array indexes using a modulus function, which is accomplished using the remainder division operator (%) in Java.

Ensuring O(1) Running Times

The only way to ensure O(1) average time for retrievals, insertions and removals is to use an array. Bear in mind that in a hashtable, when an item is added or removed it is not necessary to shift items as it is in an array-based list - gaps are permitted. Ideally the array should be the same size as the number of items it contains to ensure efficient use of space. For a hashtable, items don't need to be stored contiguously—gaps between items are acceptable. There are several problems with using an array, however:

1. Since arrays require blocks of contiguous memory, they can be expensive in terms of space consumption.

2. When the array becomes full, it must be resized before more items can be added; for a hashtable we want to avoid resizing the array as much as possible.

3. Array cells are only accessible by integers, so all keys must be ultimately be integers; Strings or other Objects can be more useful, but they can't be used to reference an array cell.

Keys and Hash Functions

In a hashtable the keys themselves are used to compute the position of the items in the hashtable. If the keys are integers, there is no problem — arrays can easily be accessed using integers. If the keys are some other object (e.g., a string), we have to somehow convert the key to a non-negative integer value that is a valid index in the array. Often the keys themselves can be used to obtain integer values. For example, for a string one could compute the sum of the ASCII codes for each character in the string. A function that does this is known as a **hash function**, (sometimes called hashing function). The integer value produced by a hash function is called a **hashcode**.

What if the resulting hashcode is too large to be a valid array index? For example, if we have a 100 cell array with indexes from 0 to 99, how do we deal with an item whose key turns out to be 437? We need some way to transform the larger integer 437 into a smaller integer that falls into the range 0 to 99. Usually this is done using a modulus function against the size of the underlying array.

Operations that can be used in a hashing function include:

1. arithmetic operations

2. polynomial operations

3. bit mixing, bit shifting

4. folding — the key is divided into parts which are combined or transformed in various ways

5. extraction — only a portion of the key is actually used to generate the hashcode

6. radix transformation — the key is transformed using another number base (e.g., base 7 instead of base 10)

7. mod functions — can be used to ensure the hashcode is within the valid range of indexes

Some hash functions are better than others. The important things to consider when creating or choosing a hash function are:

1. It should generate evenly distributed hashcodes; i.e., if the range of valid indexes is 0 to 99, each index should have equal probability of being generated by the hash function. If certain indexes are chosen more frequently than others, items will be clustered in those areas of the array. This problem can often be alleviated by using only prime numbers for the size of the array.

2. It should not generate any negative values for hashcodes.

3. It should be able to cover every cell in the array (i.e., no cells should remain forever empty because the hash function never generates keys to those cells).

Collisions

One problem with hash functions is that they aren't guaranteed to generate a unique value for every key. Sometimes a hash function will generate the same hashcode for 2 keys. This is known as a **collision**, since it means that two items will be assigned to occupy the same array cell. Obviously this is unacceptable, but since we can't always guarantee a hash function that will always generate a unique value, we have to find some way of dealing with collisions.

Dealing with Collisions in Hashtables

We will cover three ways of dealing with collisions: linear probing, quadratic probing and separate chaining hashing.

Linear Probing

Simply stated, when a collision occurs the array is scanned sequentially one cell at a time, beginning at the index where the collision occurred, to find the next empty cell. Since the index where the collision occurs could be in the middle of the array, the scanning process should be capable of wrapping around to the beginning of the array, similarly to what was done in the ArrayQueue class using a circular array.

The primary disadvantages of linear probing are:

1. If the array is heavily populated, finding the first empty cell takes longer, increasing the running time.

2. If the array is full, it must be resized.

3. Finding or deleting one of several items that have the same hashcode can be problematic—how do you know which of the items you really want?

4. A phenomenon known as **primary clustering** can occur. If a hash function frequently produces the same hashcode, those items will tend to be clustered around the array index to which that hashcode maps, rather than being evenly distributed in the array. The running times for the various operations will be dominated by collision resolution, which is a linear

process, and performance will decrease.

5. There is an additional performance decrease caused by linear probing. For example, assume an item hashes out to array index 5, but there is already an item in cell 5. Assume the item is placed in cell 6 using linear probing. If a subsequent item is added that hashes out to array index 6, another collision results. The point is that placing items into array cells they normally would not hash out to sets the stage for inadvertent collisions when future items are added.

Load Factor and Threshold

The amount of scanning that must be done depends on how full the hashtable is. The fraction of the hashtable that is full is referred to as the **load factor** (*LF*). If the probability of examining each cell is independent of the other cells, the average number of cells that must be examined using linear probing could be represented as the probability: $p = 1/(1 - LF)$.

However, with linear probing the probability of examining each cell is not independent. The rule for linear probing imposes a dependency, since it specifies the order in which the cells will be examined. Therefore, the average number of cells that must be examined must be estimated taking into account this dependence. The approximate result is: $p = (1 + 1/(1 - LF)2)/2$. That is, more cells must be examined than would normally be expected. The result is a decrease in performance.

Typical hashtable implementations do not allow the underlying array to be completely filled before it is resized. A threshold is usually established to determine when to resize the array. For example, a threshold value of 75% (0.75) might be established. When the load factor reaches the threshold, the array is resized. There is no single, optimal value for this threshold. The better the hash function is at evenly distributing items, the higher the threshold that can be used. When the array is resized, new index values must be determined for all the items in the hashtable, since the index depends on the table size. The process of recomputing the index values is known as rehashing. Rehashing is a O(*N*) operation, so it's best if resizing doesn't happen too frequently.

Quadratic Probing

In quadratic probing, collisions are not handled by scanning the array sequentially (i.e., $H + 1$, $H + 2$, $H + 3$, ...), where H is the original array index specified by the hashcode. Instead, the scanning sequence used is: $H + 1^2$, $H + 2^2$, $H + 3^2$, ...

As a result, the primary clustering effect is eliminated. Some secondary clustering does occur, since the same alternative cells are always examined when a collision occurs at a given array index, but it is generally not as extensive as primary clustering. Some tricks can be used to ensure that virtually no clustering will occur:

1. Make sure the size of the array is a prime number.

2. Make sure the load factor never exceeds 0.5 (i.e., the array is never more than half full).

Doing so will ensure that no cell is examined twice during an access. Ensuring the array is never more than half full requires being able to dynamically resize the array, and when the array is large there is a lot of unused space. Placing the items in the new array requires that a different hash function be used. This process is called rehashing.

Separate Chaining Hashing

This method of handling collisions avoids the wasted space problem caused by quadratic probing. The idea behind separate chaining hashing is simple. In each array cell, maintain a linked list. The items that hash to that cell are all stored in the linked list. Thus, each array cell is able to store multiple items. Using this method it is not uncommon to have a load factor of 1.0 (i.e., the array is full). But since the linked lists can be easily expanded, this doesn't present a problem until the linked lists become too long.

Collisions and O(1) Running Times

Technically, handling collisions means the running time for an operation may actually be somewhat greater than O(1). However, as long as the effect of handling the collisions is relatively small - i.e., it happens infrequently or only a small amount of probing is required, the cost of the collision handling can be amortized, and the resulting running time is close enough to O(1) to still be considered O(1).

The `Hashtable` class in the JDK Library

The JDK Library does provide a **`Hashtable`** class, as an extension to the obsolete class `Dictionary`. Although you can still use it, there is the possibility that it will eventually become slated for deprecation and thus no longer be available. If you need to use a hashtable it is recommended that you use either `HashMap` or `HashSet`.

Maps

In a map, items are mapped to keys and are only accessible through those keys. Maps are similar to hashtables, with the keys of a map corresponding to the hashcodes of a hashtable. One important difference, though, is that duplicate keys are not allowed in a map, whereas in a hashtable duplicate hashcodes are allowed. A summary of the characteristics of a map is as follows:

1. Each item in a map has an associated key; the item is only accessible through its key.
2. Duplicate items are allowed.
3. Duplicate keys are not allowed.
4. Each key can map to only one element.
5. Each element can be mapped to by more than one key.

Maps (and hashtables) are often used when building symbol tables for compilers, and in database management applications.

Maps and the JDK Library

The JDK library provides two basic implementations for a map:

1. `HashMap`, which is based on a hashtable
2. `TreeMap`, which is based on a red-black tree

These implementations have no relationship to JDK's `Collection` interface.

Implementation of the Map Interface

Note: You will need to refer to the source code I have provided for the remainder of this lecture.

The majority of the functionality for a map is defined by the `Map` interface. The methods specified by this interface are listed below for the sake of convenience:

```
/**
  * interface Map
  *
  * Encapsulates the basic functionality for a data structure which maps values
  * to keys. A value can only be added, removed, or retrieved via its key.
  *
  * Duplicate values are allowed in a Map, but duplicate keys are not allowed.
  *
  * This interface does not provide iterators; rather, the keys or values in the
  * Map can be returned in the form of a List, which can be traversed.
  */
public interface Map
{

    public boolean add(Object key, Object value);

    public boolean remove(Object key);

    public void clear();

    public int size();

    public boolean isEmpty();

    public Object get(Object key);

    public boolean containsKey(Object key);

    public boolean containsValue(Object value);

    public List keyList();

    public List valueList();

    public Iterator iterator();

    public MapIterator mapIterator();

}
```

The major difference between the `Map` and the other data structures we have covered is the requirement for each value to have an associated key, which must be specified when adding, removing, or retrieving values. Items are accessed strictly by their associated keys. The `Map` is traversable, so methods are provided for returning an iterator. Additional methods are provided for querying whether the `Map` contains a specified value or key, and for returning the collection of keys or the collection of values as a `List`.

The implementations I use for the map data structures are not derived from the `TraversableCollection` interface. This is done because values in maps must be associated with a key, which is a requirement unique to maps. It is not ideal in the purist sense since a map is also a collection, but I've done it this way because the add, remove, and retrieve methods that would be inherited from `TraversableCollection` interface would have to be included in the API for the map, but would be useless since those methods do not account for keys.

Implementation of the HashMap Class

The `HashMap` class is a `Map` whose underlying implementation is based on a hashtable. Although other structures could be used, using a hashtable will ensure that on average the running time for adding, removing, or retrieving a value will be close to O(1). For the `HashMap` implementation I am providing, collisions are resolved using the separate chaining hashing strategy.

Implementation of the `HashMap` Class

The **`HashMap`** class is a Map whose underlying implementation is based on a hashtable. Although other structures could be used, using a hashtable will ensure that on average the running time for adding, removing, or retrieving a value will be close to O(1). For the `HashMap` implementation I am providing, collisions are resolved using the separate chaining hashing strategy.

Class `HashFunction`

The **`HashFunction`** class is not part of the `HashMap` class, but is used by it. Since there are many ways to define a hash function, it makes sense to have a `HashMap` class that can use any one of a number of different hash functions. Encapsulating the hash function as a class gives us this capability. The class is abstract, meaning that it cannot be instantiated. Instead, a subclass must be created for each desired hash function. I have only provided one such class, `DefaultHashFunction`, which simply uses takes the value returned by an object's `hashCode` method and logically AND's that value with a mask to guarantee the result will be positive.

Inner Class `HashMapEntry`

Recall that values, or values, are added to a map as key-value pairs. The **`HashMapEntry`** class models this notion. It contains a reference to a value and its associated key. It also contains the hashcode that is generated for the key by whatever hash function is used. Finally, it contains a reference to the next entry in the chain, should a collision result in more than one value being placed in the same location in the hashtable.

The `HashMapEntry` class contains only one method: an overridden `equals` method. Note that two `HashMapEntry` instances are equal if and only if both instances contain the same key and the same value. This allows duplicate values to be added without allowing duplicate keys.

Attributes and Constants of the `HashMap` class

Since the **`HashMap`** is implemented using a hashtable, we need to have an array in which to store the values. The table attribute serves this purpose. Note that table contains references to `HashMapEntry` objects, not the values themselves.

Recall that the efficiency of a hash function depends in part on the array size that is used. Ultimately, all hashcodes must be converted to array indexes, usually by doing modular division by the array size. Prime numbers tend to give the best results for table sizes since they reduce the chances of collisions caused by hashcodes that map to a given array index or one of its factors. When the array needs to be resized, we still want to approximately double the size of the array. We can't exactly double its size, or we would no longer have a size that is a prime number. An algorithm could be used to compute the next largest array size, but it is usually more efficient to explicitly store all possible array sizes. The `tableSizes` array stores these values. For Java, we only need to store 28 table sizes, since the 29th size would exceed Java's maximum allowable integer value.

The values for the load factor and threshold the hashtable uses are stored in the attributes `loadFactor` and `threshold`, respectively. There is also an attribute, `hashFunction`, that stores a reference to the `HashFunction` class used to generate hashcodes for the keys.

As with the other data structures we have seen, we also have attributes to keep track of the size of the data structure and the number of modifications that have been made, in `theSize` and `modCount`, respectively.

Constructors

Three constructors are provided. The first constructor creates a new `HashMap` using default values for the hash function, load factor, and the set of table sizes that will be used. The second constructor allows the default hash function to be replaced with a user-defined hash function. The last constructor is the most versatile. It allows the hash function, the load factor, and even the set of table sizes to be specified. This versatility is important, since the combination of these three attributes can make a hashtable very efficient or very inefficient, depending on the values of these attributes and the data being stored in the hashtable.

Retrieving an Item Using `get(Object key)`

Retrieving a value is relatively easy. Since values are only accessed by their keys, it's necessary to query using the key, not the value itself. We first compute the key's hashcode and map the hashcode to an array index, then search through all entries in that index for an entry with the specified key. If a match is found, we return the value in the matching entry; otherwise we return null.

Adding an Item Using `add(Object key, Object value)`

The **`HashMap`** contains only one method for adding values. To add a value it is necessary to specify both the value and its associated key. Note that both the value and the key are references to `Object`. As with all array-based data structures, we must first check to see if it's necessary to resize the array. In the case of `HashMap` (and hashtable in general), the effective capacity is given by the value of threshold, which in most cases will be smaller than the array's actual capacity. The next step is to compute the hashcode and map the hashcode to a valid array index.

Recall that in a map, duplicate values are allowed, but duplicate keys are not allowed. Also, remember in this case we are resolving collisions using separate chaining hashing, so it's possible to have more than one value in one array index. We must search through all entries that are in the index to which the hash-code maps for duplicate keys. If a duplicate key is found, we simply replace the value in that entry with the new value, rather than throw an exception. If the key is not a duplicate, we add a new entry containing the key-value pair **to the beginning** of the chain. In the implementation I am using it is easier to add to the beginning of the chain than it is to add to the end of the chain, which is why we do it that way.

Finally, we update the size of the `HashMap`, as well as `modCount`.

Removing an Item Using `remove(Object key)`

To remove a value, we follow mostly the same logic as when trying to retrieve a value, since we must first find the value before we can remove it. If the value is found, we need to remove the entire **`HashMapEntry`** from the array, not just the key or value. As with a linked list, we have to make sure the two parts of the chain on either side of the removed entry are rejoined. Additionally, we have to ensure we maintain the reference to the beginning item of the chain in the array index. If the operation was successful we return `true`. If it failed, or if there is no entry with the specified key, we return `false`.

Using the `containsKey(Object key)` and `containsValue(Object value)` Methods

These methods provide simple true-false queries to determine whether a specified key or value is in the **`HashMap`**. When querying for a key, we can simply compute the hashcode for the key and go directly to the index where that key should be, if it's in the `HashMap`. Querying for a value is not as efficient. It requires searching the entire **`HashMap`** until either the value is found or it is determined the value is not in the **`HashMap`** — a O(N) procedure. There is no way to effectively use a binary search strategy since values in a **`HashMap`** are not guaranteed to be ordered.

The `rehash()` Method

When the threshold for the **`HashMap`** has been reached, two things must happen. The array must be resized, and the hashcodes for all the values that were previously in the **`HashMap`** must be remapped to the larger array. This procedure is typically called rehashing, although the term is somewhat of a misnomer, since new hashcodes are not computed - only new array indexes are. Since we are storing the various table sizes in the `tableSizes` array, we just need to consult tableSizes for the new size. In the event we are already at the maximum size, we throw a `CapacityExceededException`. Otherwise, the original array, table, is copied and a new, larger array is created. One by one, each entry in the old array is remapped to an index in the new array. The order of the entries will not necessarily be preserved; i.e., they may be in a different order in the new array than they were in the old array. This is of no concern, though, since by definition the **`HashMap`** does not make any guarantees with respect to ordering.

Traversing a `HashMap` Using `HashMapIterator`

`Traversing a HashMap` will use the same set of methods as the other iterators we have seen: `getCurrent()`, `hasNext()`, and `next()`. It also uses a method, `getCurrentKey()`, which is inherited from the `MapIterator` interface. This method returns the key of an entry, while `getCurrent()` returns the value. A `HashMapIterator` begins its traversal at the first entry in the `HashMap`, which may or may not be located in the first index of the array, table. The only tricky part of the implementation is that in order to advance to the next entry we need to consider three possible cases:

1. The next entry is in the same array index as the current entry, and is the next entry in a chain of entries.

2. The next entry is the first entry in the array cell immediately after the array cell containing the current entry.

3. There is a gap of empty array cells between the current entry and the next entry.

Running Times of the Methods

The table on the next page shows the Big-Oh running times for the methods of the `HashMap` and `HashMapIterator` classes. Note that the running time for adding, retrieving, or removing a value will depend on the following factors:

1. the running time of the hash function (for the default hash function I have provided, that running time is O(1))

2. how frequently a chain will need to be traversed to find a value (if separate chaining hashing is used to resolve collisions)

3. how many positions must be checked by probing (if one of the probing mechanisms is used to resolve collisions)

4. whether or not rehashing is required

A good hash function should produce hashcodes in O(1) time. Additionally, a good hash function will produce minimal collisions, so there should be few array cells with a chain of length > 1. In the event a chain must be traversed to find a value, the number of values that must be traversed should never be more than a fraction of the total number of values in the HashMap. If a probing mechanism is used to resolve collisions there is no way to know for certain how many indexes must be checked before an open one is found, but on average this should only take O(1) time. Rehashing will occur more frequently when the array is small, and less frequently when it is large. The effects of these factors are amortized over time, similarly to the way resizing an array is amortized for an ArrayList, so on average the running times for the add, remove, and retrieve oeprations is about O(1).

The expensive methods are `rehash` and `containsValue`. Unfortunately, there is little that can be done to bring the running times for these methods below O(N) using a simple hashtable implementation.

Class	**Method**	**Running Time Complexity (assumes `DefaultHashFunction` used for deriving hashcodes and separate chaining hashing used to resolve collisions)**	
`HashMap`	`public boolean add(Object key, Object value)`	Ω(1) (no other entries at index; no rehashing necessary)	O(N) (rehashing required)
	`public void clear()`	Θ(1)	
	`public boolean containsKey(Object key)`	Ω(1) (key is not in HashMap or key is found in first item tested)	O(N) (very unlikely except in a very small Hash-Map; requires every item to be in same chain && key is associated with last item in chain)
	`public boolean containsValue(Object value)`	Ω(1) (value is found in first item tested)	O(N + 1) (value not in Hash-Map)
	`public Object get(Object key)`	Ω(1) (key is not in HashMap or key is found in first item tested)	O(N) (very unlikely except in a very small Hash-Map; requires every item to be in same chain && key is associated with last item in chain)
	`public boolean isEmpty()`	Θ(1)	
	`private void rehash()`	Θ(N)	
	`public boolean remove(Object key)`	Ω(1) (key is found in first item tested)	O(N + 1) (key not in HashMap)
	`public int size()`	Θ(1)	
`HashMapIterator`	`public Object getCurrent()`	Θ(1)	
	`public Object getCurrentKey()`	Θ(1)	
	`public boolean hasNext()`	Θ(1)	
	`public void next()`	Ω(1) (next item to process is next item in current chain)	O(<table size> - 1) (current position is at first array index, and next value is at last array index)

Sets

A set is simply a collection that contains no duplicate items. It otherwise has all the functionality of a collection. Unlike maps, items in a set are not associated with keys. Sets are often used in database queries, where it might be necessary to retrieve the intersection of two or more resultsets obtained from different queries. For example, suppose you needed to know which students were enrolled in both CSC385 — Data Structures, AND CSC478 — Software Engineering. You can perform a separate query for each course easily enough, but if a student is enrolled in both courses you only need to see that student listed once, not twice.

Mutable Elements

Mutable elements are elements whose value can be changed (elements whose values cannot be changed are said to be immutable). Since a Set cannot contain duplicate elements, when modifying an element in a `Set` care must be taken not to inadvertently introduce a duplicate item.

Sets and the JDK Library

The class hierarchy for sets in the JDK library is shown below, in relation to the classes derived from `Collection`. The JDK library provides two basic implementations for a set:

1. `HashSet`, which is based on a hashtable
2. `TreeSet`, which is based on a red-black tree

Implementation of the `Set` Interface

Since items in a set do not need to be associated with keys, we can derive the **`Set`** interface from `TraversableCollection`. We only need to add a remove method to the interface to get the functionality we need. The code for the `Set` interface is shown below.

```
/**
 * interface Set
 *
 * Encapsulates the functionality for a set, which is defined as a type of
 * collection that does not allow duplicate items.
 *
 * For now, this interface simply serves as a basis for deriving any class
 * that is-a Set.
 */
public interface Set extends TraversableCollection
{

    /**
     * remove the specified item from the Set
     *
     * returns true if removal was successful, false otherwise
     */
    public boolean remove(Object obj);

}
```

Implementation of the `HashSet` Class

As with a **`Map`**, there are several ways to provide the underlying implementation for a `Set`. Since a hashtable proved effective for implementing a `Map`, we will also use a hashtable to implement a `Set`. We can create a `HashSet` by simply reusing the `HashMap` methods, and incorporating a test for duplicates in the implementation for adding items.

Attributes

We only need a single attribute: `hashMap`, a reference to the `HashMap` that will provide the underlying implementation. All other needed attributes can be retrieved from this `HashMap` instance.

Constructors

I have only provided a single constructor. This constructor initializes `hashMap` by calling the default constructor from the `HashMap` class. This `HashMap` instance will use the default hash function, the default table sizes, and the default load factor.

Retrieving an Item Using `get()` and `get(Object obj)`

To retrieve a specific item, we use the **`get(Object obj)`** method, which simply makes a call to the `get(Object key)` method of the `HashMap` class. We can do this since we are using the item as its own key.

The `get()` method by convention retrieves the first item in the `HashSet`. This item will be the first item in the first array cell of the underlying `HashMap` that contains one or more items.

Adding an Item Using `add(Object obj)`

To add an item, we can call the add method of `HashMap`. We use the item itself as both the key and the value to satisfy the requirements of the `HashMap` add method. First, though, we must make sure the item doesn't already exist in the **`HashSet`**. We need to call the `containsValue` method of `HashMap` to accomplish this. If the item does exist, the method returns `false`.

Removing an Item Using `remove()` and `remove(Object obj)`

The **`remove(Object obj)`** method allows removal of a specified item. To do this we simply need to call the `remove(Object key)` method from the `HashMap` class.

The `remove()` method by convention removes the first item in the `HashSet`. This item will be the first item in the first array cell of the underlying `HashMap` that contains one or more items.

Using the `contains(Object obj)` Method

Using this method means we are interested in knowing whether a particular value exists in the **`HashSet`**. This can be accomplished easily by returning the return value of the `containsValue` method of `HashMap`.

Running Times of the Methods

Class	Method	Running Time Complexity	
`HashSet`	`public boolean add(Object obj)`	$\Omega(1)$ (duplicate item found in first item tested via containsValue)	$O(N)$ (rehashing required || all items must be checked via containsValue to ensure no duplicates)
	all other methods	refer to corresponding HashMap method	

JDK Classes `LinkedHashMap` and `LinkedHashSet`

Due to the random-access manner in which items are added to the standard **`HashMap`** or `HashSet`, when traversing these structures, the order in which items are processed is not guaranteed. In many cases, this lack of deterministic ordering does not matter; for example, if all items need to be processed, or if the structure is being searched for a particular item. There can be cases, however, when it is desirable to be able to process items in the order in which they were added, similarly to how a list is traversed. The JDK has provided two classes with such a hashing/list hybrid structure: the `LinkedHashMap` and the `LinkedHashSet`. In these structures, the inner class that represents an entry contains references to the entry of the item that was added immediately prior to that item, as well as the entry of the item that was added immediately after that item. The result is a structure that still allows O(1) time retrievals, but also allows deterministic traversal of the items, with the order of traversal being the order in which the items were added.