

---

# Hashing

Reading:  
L&C 17.1, 17.3  
Eck 10.3

---

# Objectives

- Define hashing
- Discuss the problem of collisions in hash tables
- Examine Java's **HashMap<K, V>** implementation of hashing
- Look at **HashMap** example
- Save *serializable* objects to binary files

# Hashing

So far, all of the collections that we have seen have made one of the following assumptions about the order of the elements in the collection:

- order is unimportant (sets)
- order is determined by the way elements are added (stacks, queues)
- order is determined by comparing the elements (sorted lists)

---

# Hashing

- We looked at ways to search for a particular element in a collection, such as linear search and binary search.
- The efficiency of both of these search algorithms depends on the number of elements in the collection.

---

# Hashing

- **Hashing** is a very clever way of determining the location of an element based on some function of the element itself.
- This way, the time it takes to find an element is independent of the number of elements being stored in the collection.

---

# Hashing

- Elements are stored in a **hash table** (or **hash map**), with their locations in the table determined by a **hashing function**.
- Each location in the hash table is called a **bucket**, or a **cell**.
- Elements are mapped to a bucket using a hashing function that calculates a hash code.
- To find an element in a hash table, you just have to compute the **hash code** of the key and go directly to the table location given by that hash code.

---

# Hashing Example

- Consider an example where we want to store names.
- Our hash table will be an array.
- The buckets are indexes into the array.
- The elements are the names we want to store.

# Hashing Example

- Suppose the hashing function uses the first letter of each name.
- A is mapped to position 0 (hash code 0), B is mapped to position 1, and so on up to Z, which is mapped to 25.

Ann
Doug
Elizabeth
Hal
Mary
Tim
Walter
Young



---

# Hashing Example

- To locate elements in the hash table, we perform the hashing function to find the position.
- To find “Tim” in our example, we perform the hashing function

**`name.charAt(0)` → 'T'**

and get a hash code of 19 (remember that every character is represented as an **`int`**).

---

# Hashing

- With hashing, we don't have to compare elements, we can find them directly, with each access taking a constant amount of time.
- However, this efficiency is only realized if each element maps to a unique position in the table.
- When two or more elements map to the same position, it is called a **collision**.
- E.g., “Andrew” also maps to position 0.

---

# Perfect Hashing Function

- A hashing function that maps each element to a unique position in the table is called a **perfect hashing function**.
- It is not necessary to develop a perfect hashing function to use a hash table.
- A function that does a good job of distributing the elements in the table will still be more efficient than a linear search or binary search.

# Hashing – Table Size

- Let's examine the issue of how to decide how large the table should be.
- Table size depends on two factors:
  - the number of elements to store ( $n$ )
  - whether or not we have a perfect hashing function
- Assuming that we know that  $n$  elements will be stored in the table:
  - if we have a perfect hashing function, the table size should be  $n$ .
  - if our hashing function is not perfect, it is recommended to make the table size  $n * 1.5$

---

# Hashing – Table Size

- Usually we don't know how many elements we will need to store in the table.
- In that case, we rely on **dynamic resizing**.
- Like resizing an array, resizing a hash table involves creating a larger table and inserting all of the elements of the old table into the new one.

---

# Hashing – Table Size

- Luckily, the table resizing is done for us.
- We can, however, indicate when it should be done.
- When resizing an array, we wait until it is completely full.
- It is not a good idea to let a hash table get too full because the more full it is the less efficient it becomes.

---

# Hashing – Load Factor

- A **load factor** is used to determine when to resize the table.
- The load factor indicates at what percentage occupancy the table should be resized.
  - a load factor of 0.5 indicates that the table should be resized when it is 50% full.
  - a load factor of 0.75 indicates that the table should be resized when it is 75% full.

---

# Hashing – Load Factor

- The classes in the java library that implement hash tables have constructors that allow you to give an initial capacity and (optionally) a load factor as arguments.
- The default initial capacity is very small (16 for the **HashMap** class), so if you want to store more than 10 elements you should give a better initial capacity.
- The default load factor is 0.75, which should be ok in most cases.



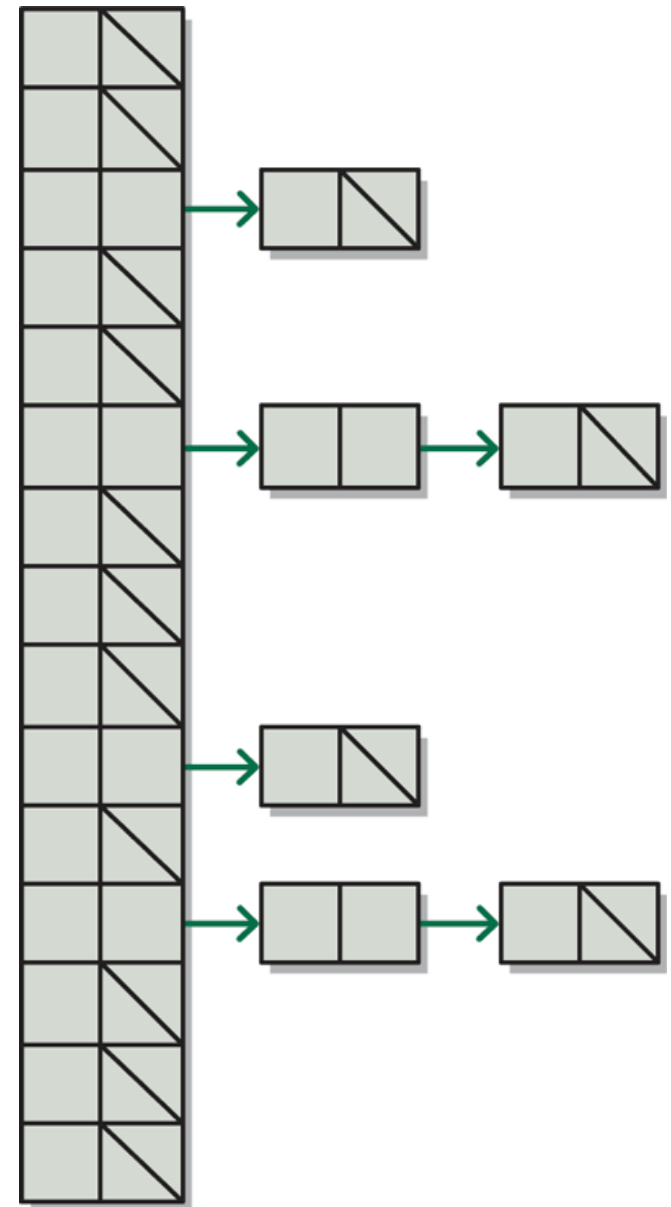
---

# Resolving Collisions – Chaining

- When more than one element map to the same table location (i.e., have the same hashcode), a collision occurs.
- The most common way of resolving collisions is by **chaining**.
- With chaining, each location in the table has a linked list of elements to be stored at that location.
- When an element is added to a location, it is simply appended to the linked list.

# Resolving Collisions – Chaining

- When searching for an element, first the cell is identified by the hashing function, then the list at that location is searched.
- A good hashing function will keep the size of the linked lists small.



---

# Resolving Collisions – Chaining

- Suppose we have a hashing function that just returns some constant, zero for example.
- Is this a good idea?
- Why or why not?

---

# Resolving Collisions – Chaining

- It's NOT a good idea.
- All of the elements will be inserted at position 0.
- In effect we have turned our hash table into a linked list.
- Searching the linked lists is done linearly, using the **equals** method to determine if the element has been found.
- As we know, linear searching is not very efficient.

---

# Hashing Functions

- There is a wide variety of hashing functions that provide good distribution of various types of data.
- Lewis&Chase discuss several ways of developing a hashing function.
- We will try to simplify our hash functions by combining existing hash codes.

---

# equals and hashCode Methods

- Every object in Java has a hash code.
- The **Object** class defines the method **hashCode()**, which computes the hashcode and returns it as an **int**.
- All objects inherit the **hashCode** method from the **Object** class.

---

# **equals and hashCode Methods**

- The **hashCode** method in the **Object** class, however, returns an integer based on the memory address of the object.
- Remember that the **equals** method in the **Object** class is also based on the memory address.
- For hashing to work, two objects that are considered equal by the **equals** method must also have the same hashcode.

---

# **equals and hashCode Methods**

- Most of our classes override the **equals** method in the **Object** class, so that we can determine if the contents of the objects are equal, rather than just checking if they are aliases of each other.
- If we want to use these classes in a hash table, we must also override the **hashCode** method, so that objects that are equal also have the same hashcode.



---

# **equals and hashCode Methods**

- Most of the classes included in the Java library have correctly defined **equals** and **hashCode** methods. In particular, the **String** class has well-defined **equals** and **hashCode** methods.
- If **Strings** are being used, you don't need to do anything extra.

---

# hashCode()

- The **hashCode** method in a class you define yourself should include all of the instance variables that the **equals** method uses.
- Let's look at some examples using a **Word** class.

# hashCode() for Word1

```
public class Word1 {
    private String form;
    private int frequency;
    < constructors, etc >

    public boolean equals(Object otherObj) {
        // two words are equal if they have same form
        if (otherObj == null)
            return false;

        if (getClass() != otherObj.getClass())
            return false;

        Word1 otherWord = (Word1) otherObj;
        return form.equals(otherWord.form);
    }
}
```

---

# hashCode() for Word1

```
public class Word1 {  
    private String form;  
    private int frequency;  
    < constructors, etc >  
  
    < equals method based on form only >  
  
    public int hashCode() {  
        return form.hashCode();  
    }  
}
```

# hashCode() for Word2

```
public class Word2 {
    private String form;
    private int frequency;
    < constructors, etc >

    public boolean equals(Object otherObj) {
        // two words are equal if they have same form
        // and the same frequency
        if (otherObj == null)
            return false;

        if (getClass() != otherObj.getClass())
            return false;

        Word2 otherWord = (Word2) otherObj;
        return (form.equals(otherWord.form) &&
                (frequency == otherWord.frequency));
    }
}
```

# hashCode() for Word2

```
public class Word2 {  
    private String form;  
    private int frequency;  
    < constructors, etc >
```

*< equals method based on form AND frequency >*

```
    public int hashCode() {  
        return form.hashCode() + frequency;  
        // or: form.hashCode() * frequency;  
    }  
}
```

# Maps

- A **map** is a kind of generalized array, sometimes also called an **associative array**.
- In an array, we use an integer index to access elements: `int x = myArray[5];`
- In a map, we use an object (a **key**) to access elements.
- Each **key** is associated with a **value**.

---

# Maps

- The keys are used to determine a position in the map.
- Objects used as keys must have well-defined **equals** and **hashCode** methods.
- A value is associated with a key.
- Objects used as values do not necessarily need to override the **equals** and **hashCode** methods.



---

# HashMap<K,V>

- Let's take a look at Java's **HashMap<K,V>** class (in package **java.util**) and an example of how it can be used.
- Suppose we want to store some names and phone numbers in a hash map.
- We will use a person's name as the key and their phone number as the value.
- We will use strings for both the name and the phone number.

---

# HashMap<K,V>

- Our phonebook will most likely contain about 100 entries.
- Let's look at our choices for constructors in the **HashMap** class and create an empty **HashMap**.

# HashMap<K,V> - Constructors

- **HashMap()**

Constructs an empty **HashMap** with the default initial capacity (16) and the default load factor (0.75).

- **HashMap(int initialCapacity)**

Constructs an empty **HashMap** with the specified initial capacity and the default load factor (0.75).

- **HashMap(int initialCapacity,  
float loadFactor)**

Constructs an empty **HashMap** with the specified initial capacity and load factor.

# HashMap<K,V> - Example

- A load factor of 75% is usually ok, but the default initial capacity of 16 is too small.
- We'll use the second constructor with an initial capacity of 150, which is 150% of our estimated data size of 100:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

- A **HashMap** has two *type parameters* (both of type **String** in this example):
  - the first one specifies the type of the keys
  - the second specifies the type of the values

---

# HashMap<K,V> - put

- Let's add some entries to our phonebook:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

# HashMap<K,V> - get

- Get Sam's and Tim's number:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
String samsNumber = phonebook.get("Sam");  
// samsNumber is now "54321"
```

```
String timsNumber = phonebook.get("Tim");  
// timsNumber is now null
```

---

# HashMap<K,V> - containsKey

- Find out if “Mike” is in phonebook:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
if (phonebook.containsKey("Mike"))  
    // do something
```

---

# HashMap<K,V> - remove

- Remove “Joe” from the phonebook:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
phonebook.remove("Joe");
```



# HashMap<K,V> - change a value

- Change Kathy's phone number:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
phonebook.put("Kathy", "222222");  
// Kathy's phone number is replaced
```

# HashMap<K,V> - change a value

- Add an extension to Kathy's number:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
String value = phonebook.get("Kathy");  
if (value != null) {  
    value += "-32";  
    phonebook.put("Kathy", value);  
    // Kathy's number is now "11111-32"  
}
```

# HashMap<K,V> - toString

- We could use **toString** to print:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
System.out.println(phonebook);
```

- The output is not well formatted.
- It is better to print each entry on a separate line.
- We'll do this with iteration.

# HashMap<K,V> - Iterating

- The **HashMap** class does not provide an **iterator** method.
- There are three common ways of iterating a **HashMap**:
  - **keySet()**: get a **Set** of the keys in the map and iterate it.
  - **entrySet()**: get a **Set** of the entries in the map and iterate it.
  - **values()**: get a **Collection** of the values and iterate it.
- Either way is fine – just pick one.

---

# HashMap<K,V> - Iterating

- Note that the order in which we get the elements back from either of these methods is unpredictable.
- This is because an element's position depends on the hashCode and the table size.
- The table size can change as we add elements.

# HashMap<K,V> - Iterating (keySet)

- Print all entries:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);  
String value;
```

```
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");
```

```
// for each key in the map's keyset  
for (String key : phonebook.keySet()) {  
    value = phonebook.get(key);  
    System.out.println(key + " " + value);  
}
```

# HashMap<K,V> - Iterating (entrySet)

- Print all entries:

```
HashMap<String, String> phonebook =  
    new HashMap<String, String>(150);  
String key, value;  
  
phonebook.put("Joe", "12345");  
phonebook.put("Sam", "54321");  
phonebook.put("Kathy", "11111");  
  
// for each entry in the map  
for (Map.Entry<String,String> entry : phonebook.entrySet()) {  
    key = entry.getKey();  
    value = entry.getValue();  
    System.out.println(key + " " + value);  
}
```

---

# Saving to a Binary File

- In Java, it is possible to save any *serializable* object to a binary file.
- **ArrayLists, LinkedLists, and HashMaps** are all serializable (to name a few), i.e., they implement the **Serializable** interface.
- The **writeObject** method of the **ObjectOutputStream** class is used to store the data



---

# Saving to a Binary File

```
try {  
    ObjectOutputStream out = new ObjectOutputStream(  
        new FileOutputStream("myFile"));  
    out.writeObject(myHashMap);  
    out.close();  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

---

# Retrieving the Object from a File

- To retrieve the object from the file, use the **readObject** method of the **ObjectInputStream** class.

# Retrieving the Object from a File

```
File inputFile = new File("myFile");
try {
    if (inputFile.exists()) {

        ObjectInputStream in = new ObjectInputStream(
                                new FileInputStream(inputFile));

        myHashMap = (HashMap<String,String>) in.readObject();
        in.close();

    }
} catch (IOException e) {
    System.out.println(e.getMessage());
} catch (ClassNotFoundException e) {
    System.out.println(e.getMessage());
}
```

---

# Serializable

- You can make your own classes serializable by having them implement the **Serializable** interface:

```
public class MyClass implements Serializable { ... }
```

- The **Serializable** interface has no methods that need to be implemented, so that's it!

# HashMap FactSheet

HashMaps are storing data in the form of maps. A map consists of a key and a value:



The key has to be always a Java object (String, Integer or your own objects). It is important that the key objects are providing a unique hashCode. This is the case for every built-in object in Java. If you want to use your own objects as keys, make sure that they are providing a method hashCode and that this method returns a unique (integer-) value!

## Creating a HashMap:

```
HashMap<String, String> hm = new HashMap<>(<String, String>);
```

This creates a HashMap where the key and the value are Strings. Another common HashMap declaration would be <Integer, String> which creates a HashMap with an Integer as key and a String as value.

## Putting maps into the HashMap:

```
hm.put(„ABC“, „XYZ“);
```

The keys of a HashMap are always unique, so it is not possible to put a second map with the same key into a HashMap. In this case, the already existing map will be overwritten.

## Getting a map out of a HashMap:

```
String s = hm.get(„ABC“);
```

This makes use of the key („ABC“) to retrieve the map from the HashMap and store the value of the Map („XYZ“) into the String s.

## Iterating over HashMaps:

Iterating over a HashMap can be done in three ways: over the keys (keySet), the values (values) or the whole maps (entrySet):

- **keySet:** iterates over all keys in the HashMap. Gives back a set of the keys. For example, if the keys are Strings, keySet gives back a Set of type String
- **values:** gives back a collection with all values in the HashMap. The collection has the same type as the values
- **entrySet:** gives back a set of maps. Every map consists of a key and its corresponding value

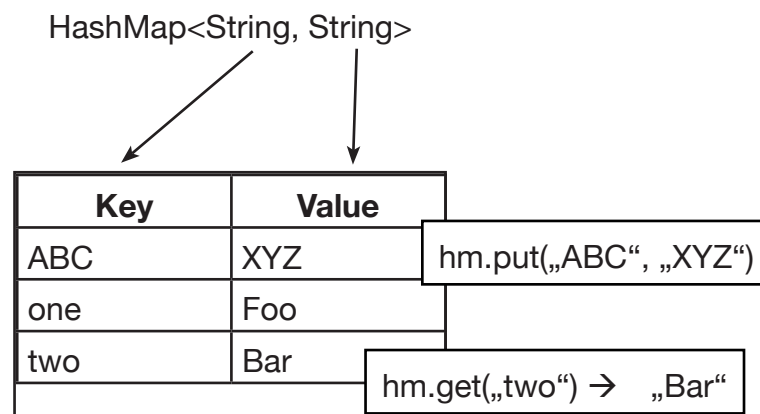
Example: iterating over the maps of a HashMap:

```
for (String key : hm.keySet()) {  
    value = hm.get(key);  
    System.out.println(key + " " + value);  
}
```

*Please mention: the order in which you'll get the elements from the HashMap is absolutely unpredictable!*

Iterating over a HashMap is only useful if

1. you don't know the key of the map you are looking for
2. you have to touch all maps in the HashMap



## Further Reading:

<http://math.hws.edu/javanotes/c10/s3.html>