What: WiCS CS Courses: Inside Scoop
When: Monday, Nov 19th from 5-7pm
Where: SEO 1000

Having trouble figuring out what classes to take next semester? Wish you had information on what CS course to take when? Want to know more about the professors who teach these classes?
WiCS (Women in Computer Science) is hosting an event where Undergrad and Graduate students can get the inside scoop on computer science courses. We will have a board of upperclassman and graduate students who can give details and tips about the classes you want to take in the future.
****This event is open to any students interested in taking CS course.****

...and Free Food!!!!
So please come and find out all you need to know to prepare your classes for the coming semesters. And it's students only - so you can ask ANY questions!

> "Love, like a chicken salad or restaurant hash, must be taken with blind faith or it loses its flavor"
>
> **Helen Rowland** (1875-1950)

CS202 Fall 2012 Lecture – 11/15

Hashing

Prof. Tanya Berger-Wolf

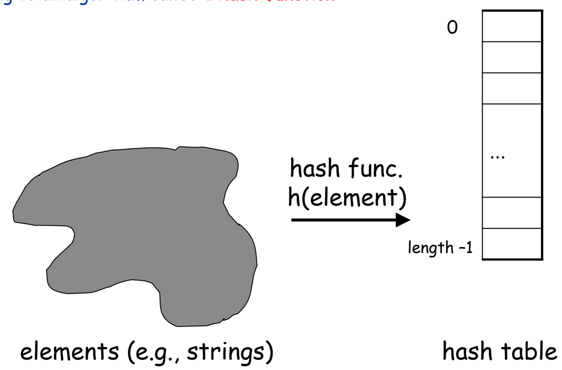
Remember? Word search

Write an application that reads in the text of all the web pages on the web and then lets the user type words, and tells whether those words are contained in WWW or not.

1. How would we implement this with a List?
2. Would this be a good or bad implementation?
3. Does the ordering of the elements in the List affect the algorithm? Could we use this information to our advantage?

Hash tables

hash table: an array of some fixed size, that positions elements according to an algorithm called a **hash function**



Hashing, hash functions

The idea: somehow we map every element into some index in the array ("hash" it);
this is its one and only place that it should go

1. Lookup becomes *constant-time*: simply look at that one slot again later to see if the element is there
2. add, remove, contains all become O(1) !

For now, let's look at integers (`int`)

1. a "hash function" *h* for `int` is trivial:
   store `int i` at index `i` (a direct mapping)
   - if `i` >= `array.length`, store i at index
     `(i % array.length)`

2. *h(i) = i % array.length*

Hash function example

elements = Integers
h(i) = i % 10
add 41, 34, 7, and 18
constant-time lookup:

1. just look at *i % 10* again later

We lose all ordering information:

1. getMin, getMax, removeMin, removeMax
2. the various ordered traversals
3. printing items in sorted order

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Hash collisions

collision: the event that two hash table elements map into the same slot in the array

example: add 41, 34, 7, 18, then 21

1. 21 hashes into the same slot as 41!
2. 21 should not replace 41 in the hash table;
   they should both be there

collision resolution: means for fixing collisions in a hash table

0	
1	21
2	
3	
4	34
5	
6	
7	7
8	18
9	

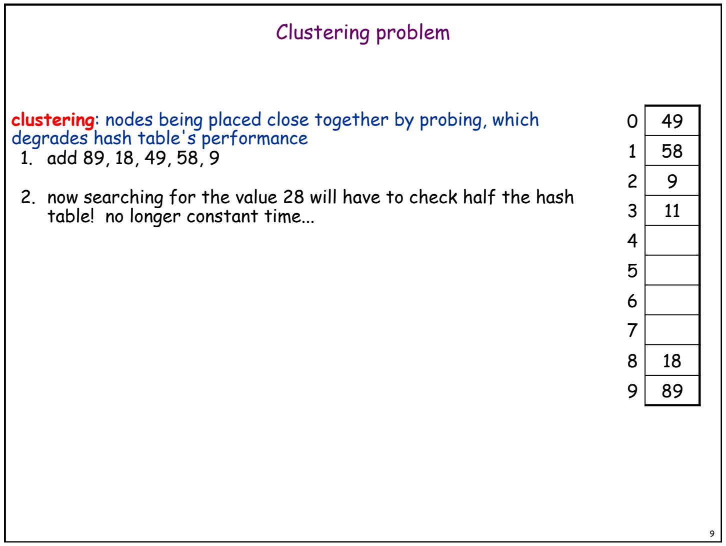
Linear probing

linear probing: resolving collisions in slot *i* by putting the colliding element into the next available slot (i+1, i+2, ...)

1. add 41, 34, 7, 18, then 21, then 57
   - 21 collides (41 is already there), so we search ahead until we find empty slot 2
   - 57 collides (7 is already there), so we search ahead twice until we find empty slot 9

2. lookup algorithm becomes slightly modified; we have to loop now until we find the element or an empty slot
   - what happens when the table gets mostly full?

0	20
1	41
2	21
3	58
4	34
5	
6	
7	7
8	18
9	57



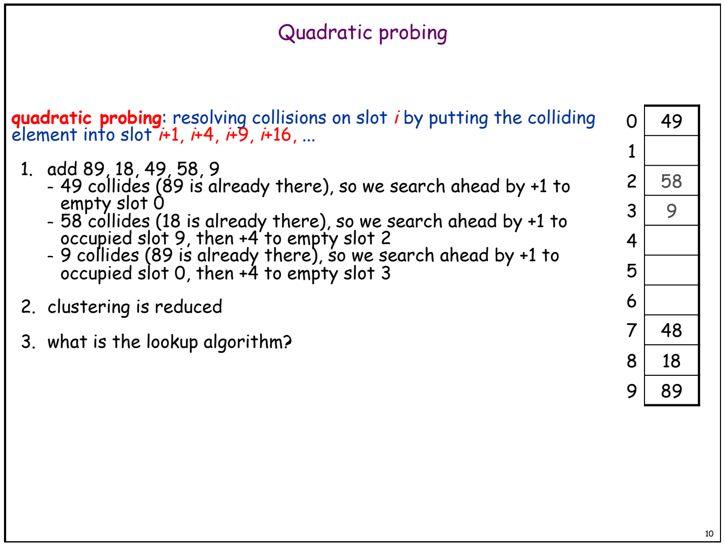
Clustering problem

clustering: nodes being placed close together by probing, which degrades hash table's performance

1. add 89, 18, 49, 58, 9
2. now searching for the value 28 will have to check half the hash table! no longer constant time...

0	49
1	58
2	9
3	11
4	
5	
6	
7	
8	18
9	89

9



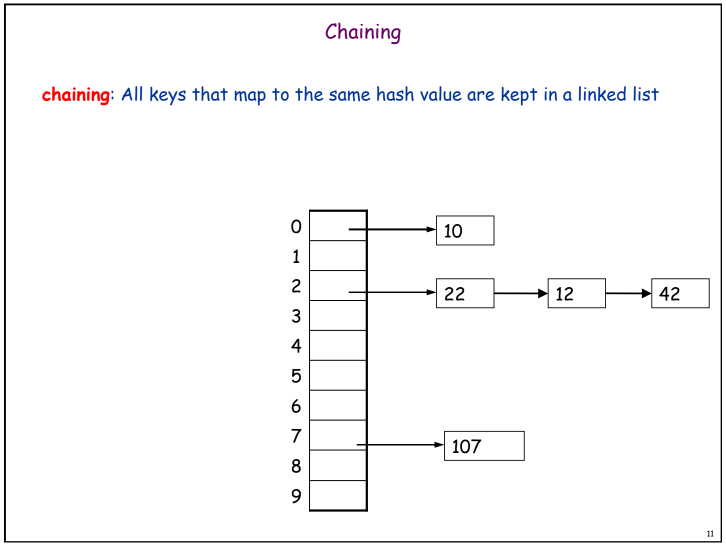
Quadratic probing

quadratic probing: resolving collisions on slot *i* by putting the colliding element into slot *i*+1, *i*+4, *i*+9, *i*+16, ...

1. add 89, 18, 49, 58, 9
   - 49 collides (89 is already there), so we search ahead by +1 to empty slot 0
   - 58 collides (18 is already there), so we search ahead by +1 to occupied slot 9, then +4 to empty slot 2
   - 9 collides (89 is already there), so we search ahead by +1 to occupied slot 0, then +4 to empty slot 3
2. clustering is reduced
3. what is the lookup algorithm?

0	49
1	
2	58
3	9
4	
5	
6	
7	48
8	18
9	89

10



Chaining

chaining: All keys that map to the same hash value are kept in a linked list

0 → 10

1

2 → 22 → 12 → 42

3

4

5

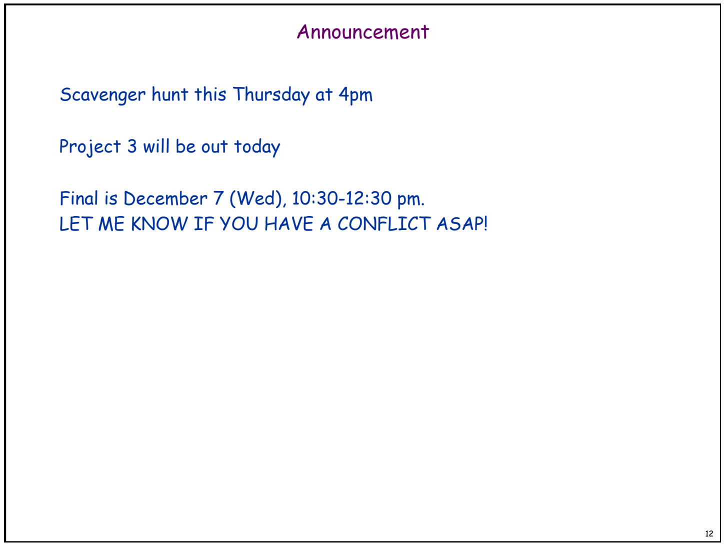
6

7 → 107

8

9

11



Announcement

Scavenger hunt this Thursday at 4pm

Project 3 will be out today

Final is December 7 (Wed), 10:30-12:30 pm.
LET ME KNOW IF YOU HAVE A CONFLICT ASAP!

12

Writing a hash function

If we write a hash table that can store objects, we need a hash function for the objects, so that we know what index to store them

We want a hash function to:

1. be simple/fast to compute
2. map equal elements to the same index
3. map different elements to different indices
4. have keys distributed evenly among indices

Hash function for strings

elements = Strings

let's view a string by its letters:

1. String s : $s_0, s_1, s_2, \ldots, s_{n-1}$

how do we map a string into an integer index?

(how do we "hash" it?)

one possible hash function:

1. treat first character as an `int`, and hash on that
   - *h(s) =* s_0 *% array.length*
   - is this a good hash function? When will strings collide?

Better string hash functions

view a string by its letters:

String s : $s_0, s_1, s_2, \ldots, s_{n-1}$

another possible hash function:

treat each character as an `int`, sum them, and hash on that

$$h(s) = \left(\sum_{i=0}^{n-1} s_i\right) \% \text{ array.length}$$

what's wrong with this hash function? When will strings collide?

a third option:

perform a *weighted sum* of the letters, and hash on that

$$h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i\right) \% \text{ array.length}$$

Analysis of hash table search

load: the load λ of a hash table is the ratio:

$$\frac{N}{M} \quad \begin{matrix} \leftarrow \text{no. of elements} \\ \leftarrow \text{array size} \end{matrix}$$

analysis of search, with linear probing:

1. unsuccessful: ≈ $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

2. successful: ≈ $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$

Analysis of hash table search

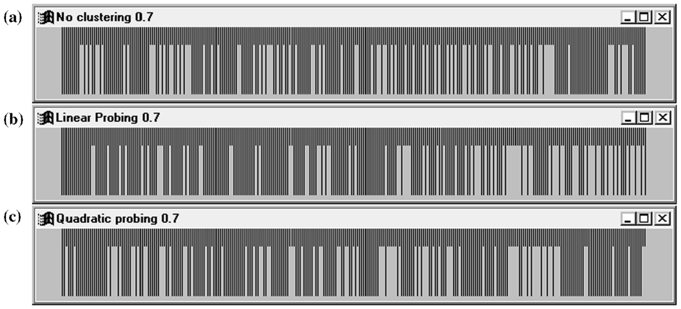
analysis of search, with chaining:

1. unsuccessful: λ
   (the average length of a list at hash(*i*))

2. successful: 1 + (λ/2)
   (one node, plus half the avg. length of a list
   (not including the item))

Analysis of linear, quadratic

graphical representation of hash table elements with load of 0.7



Rehashing, hash table size

rehash: increasing the size of a hash table's array, and re-storing all of the items into the array using the hash function
can we just copy the old contents to the larger array?

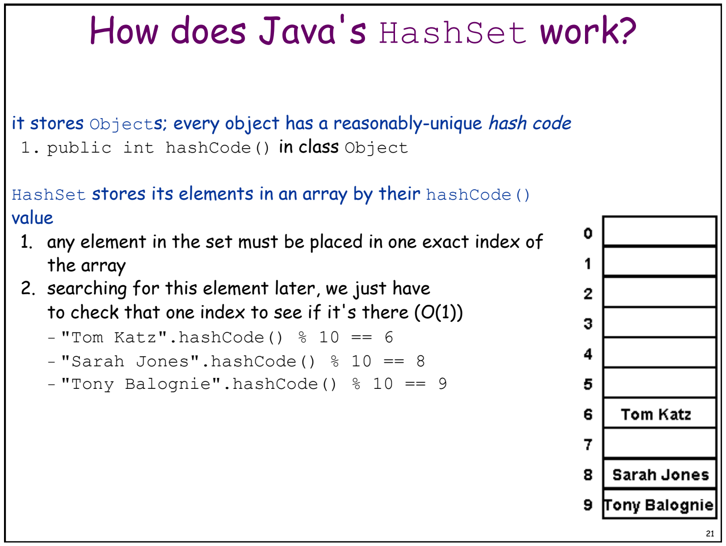
When should we rehash? Some options:
when load reaches a certain level (e.g., λ = 0.5)
when an insertion fails

What is the cost (Big-Oh) of rehashing?
what is a good hash table array size?
how much bigger should a hash table get when it grows?

Hash versus tree

Which is better, a hash set or a tree set?

Hash	Tree
O(1) search time	O(log n) search time
O(1) insertion time	O(log n) insertion
Collision	No collision
Unsorted	Sorted
O(n) rank order stats	O(1) rank order stats
1/load memory	O(n) memory



How does Java's `HashSet` work?

it stores `Objects`; every object has a reasonably-unique *hash code*

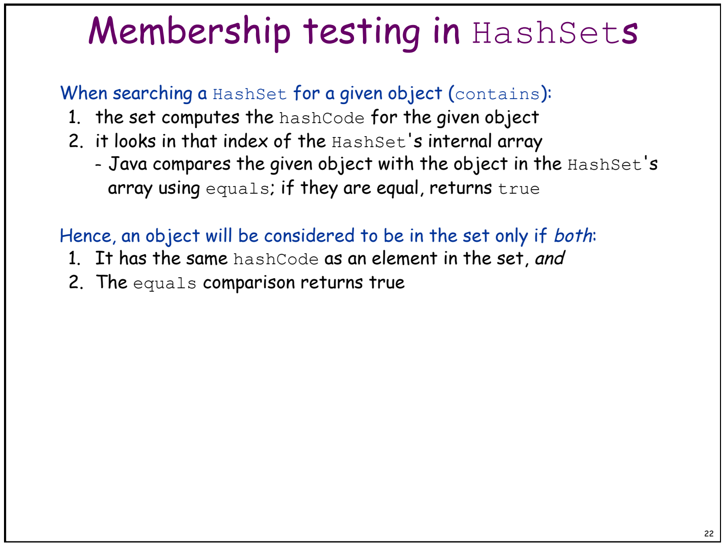
1. `public int hashCode()` in class `Object`

`HashSet` stores its elements in an array by their `hashCode()` value

1. any element in the set must be placed in one exact index of the array
2. searching for this element later, we just have to check that one index to see if it's there (*O*(1))
   - `"Tom Katz".hashCode() % 10 == 6`
   - `"Sarah Jones".hashCode() % 10 == 8`
   - `"Tony Balognie".hashCode() % 10 == 9`

0	
1	
2	
3	
4	
5	
6	Tom Katz
7	
8	Sarah Jones
9	Tony Balognie

21



Membership testing in `HashSets`

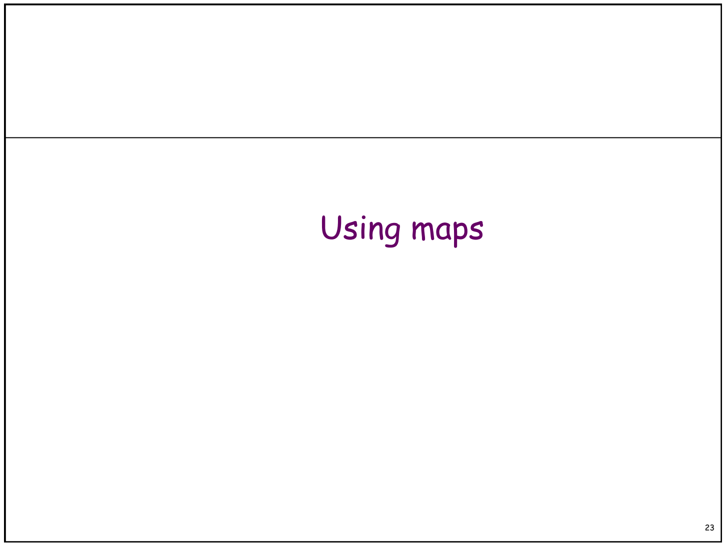
When searching a `HashSet` for a given object (`contains`):

1. the set computes the `hashCode` for the given object
2. it looks in that index of the `HashSet`'s internal array
   - Java compares the given object with the object in the `HashSet`'s array using `equals`; if they are equal, returns `true`

Hence, an object will be considered to be in the set only if *both*:

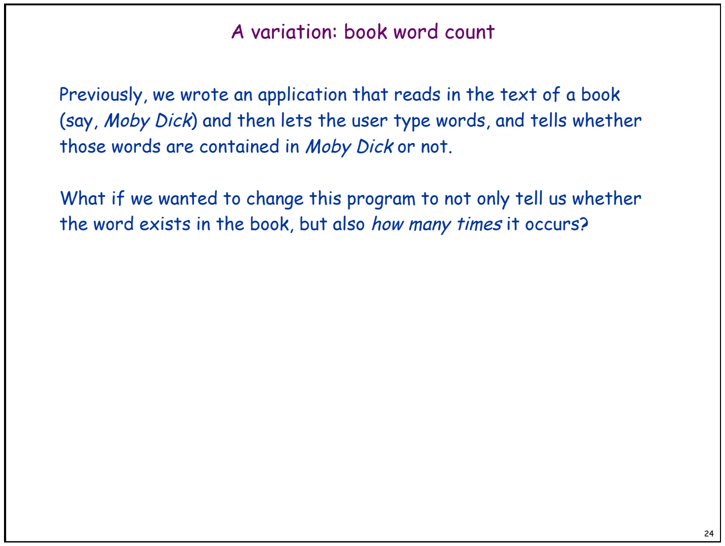
1. It has the same `hashCode` as an element in the set, *and*
2. The `equals` comparison returns true

22



Using maps

23



A variation: book word count

Previously, we wrote an application that reads in the text of a book (say, *Moby Dick*) and then lets the user type words, and tells whether those words are contained in *Moby Dick* or not.

What if we wanted to change this program to not only tell us whether the word exists in the book, but also *how many times* it occurs?

24

Mapping between sets

sometimes we want to create a mapping between elements of one set and another set

1. example: map words to their count in the book
   - "the" --> 325
   - "whale" --> 14
2. example: map people to their phone numbers
   - "Marty Stepp" --> "692-4540"
   - "Jenny" --> "867-5309"

How would we do this with a list (or list(s))?

1. A list doesn't map people to phone numbers; it maps `int`s from 0 .. *size* - 1 to objects
2. Could we map some `int` to a person's name, and the same `int` to the person's phone number?
3. How would we find a phone number, given the person's name? Is this a good solution?

25

A new ADT: **Map**

map: an unordered collection that associates a collection of element values with a set of keys so that elements they can be found very quickly

1. Each key can appear at most once (no duplicate keys)
2. A key maps to at most one value
3. the main operations:
   - **put**(key, value)
     "Map this *key* to that *value*."
   - **get**(key)
     "What value, if any, does this key map to?"
4. maps are also called:
   - dictionaries
   - associative arrays
   - (depending on implementation) tables, hashes, hash tables

26

Java's `Map` interface

Basic ops

```
public interface Map<K, V> {
public V put(K key, V value);
public V get(Object key);
public V remove(Object key);
public boolean containsKey(Object key);
public boolean containsValue(Object value);
public int size();
public boolean isEmpty();
```

Bulk ops

```
public void putAll(Map<K, V> map);
public void clear();
```

Collection views

```
public Set<K> keySet();
public Collection<V> values();
}
```

27

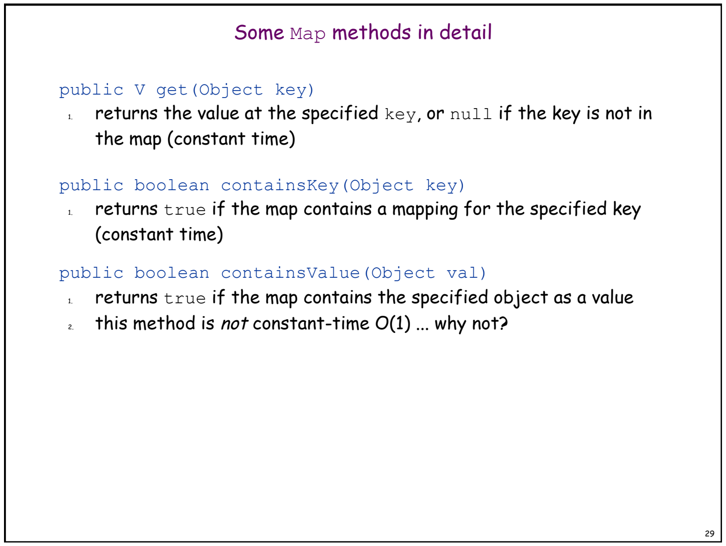
`Map` example

```
import java.util.*;

public class Birthday {
    public static void main(String[] args){
        Map<String, Integer> m = new HashMap<String, 
Integer>();
        m.put("Newton", 1642);
        m.put("Darwin", 1809);
        System.out.println(m);
    }
}

Output:
{Darwin=1809, Newton=1642}
```

28



Some Map methods in detail

`public V get(Object key)`

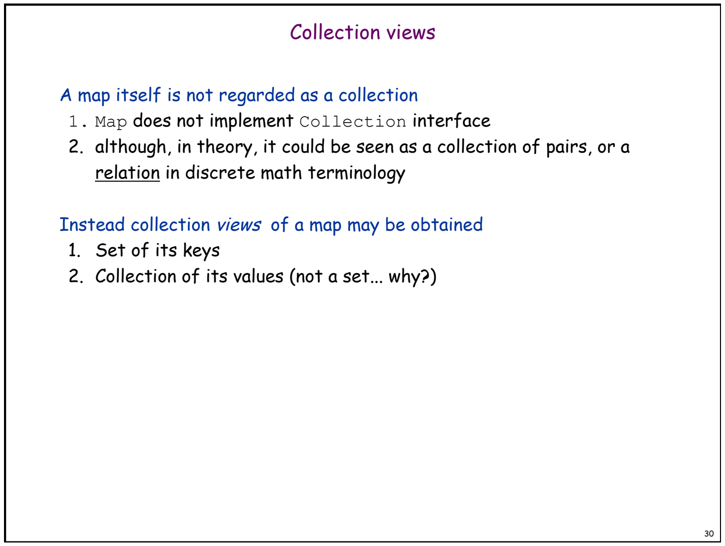
1. returns the value at the specified `key`, or `null` if the key is not in the map (constant time)

`public boolean containsKey(Object key)`

1. returns `true` if the map contains a mapping for the specified key (constant time)

`public boolean containsValue(Object val)`

1. returns `true` if the map contains the specified object as a value
2. this method is *not* constant-time O(1) ... why not?



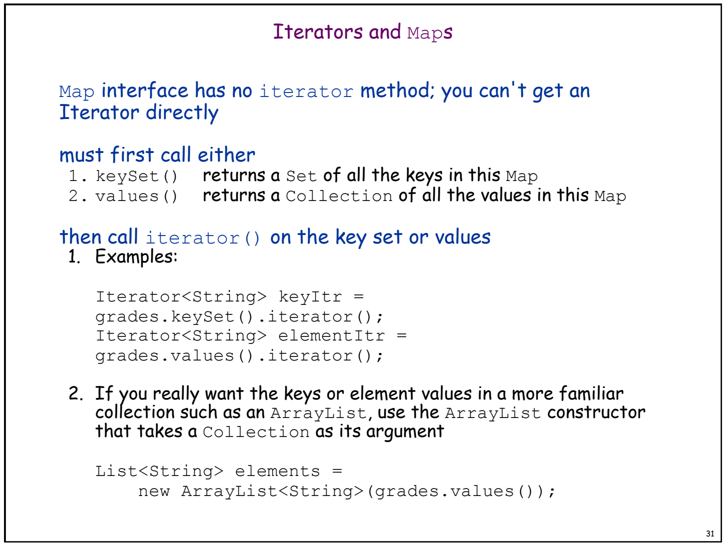
Collection views

A map itself is not regarded as a collection

1. `Map` does not implement `Collection` interface
2. although, in theory, it could be seen as a collection of pairs, or a relation in discrete math terminology

Instead collection *views* of a map may be obtained

1. Set of its keys
2. Collection of its values (not a set... why?)



Iterators and Maps

`Map` interface has no `iterator` method; you can't get an Iterator directly

must first call either

1. `keySet()` returns a `Set` of all the keys in this `Map`
2. `values()` returns a `Collection` of all the values in this `Map`

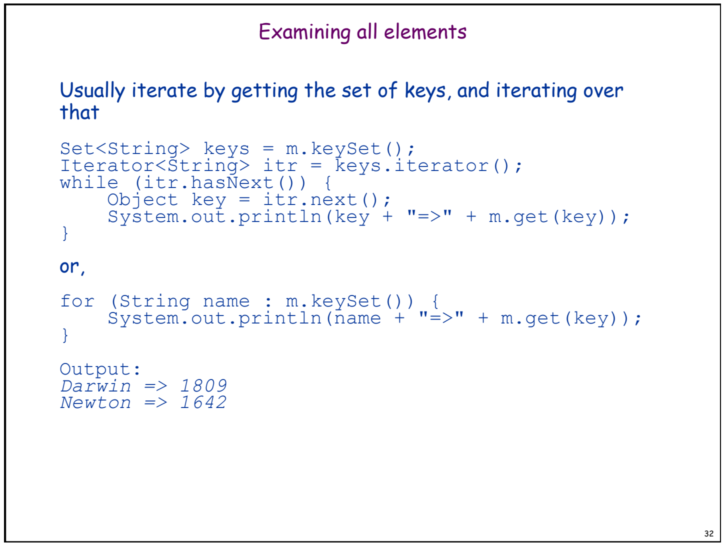
then call `iterator()` on the key set or values

1. Examples:

```
Iterator<String> keyItr =
grades.keySet().iterator();
Iterator<String> elementItr =
grades.values().iterator();
```

2. If you really want the keys or element values in a more familiar collection such as an `ArrayList`, use the `ArrayList` constructor that takes a `Collection` as its argument

```
List<String> elements =
    new ArrayList<String>(grades.values());
```



Examining all elements

Usually iterate by getting the set of keys, and iterating over that

```
Set<String> keys = m.keySet();
Iterator<String> itr = keys.iterator();
while (itr.hasNext()) {
    Object key = itr.next();
    System.out.println(key + "=>" + m.get(key));
}
```

or,

```
for (String name : m.keySet()) {
    System.out.println(name + "=>" + m.get(key));
}
```

```
Output:
Darwin => 1809
Newton => 1642
```

Map practice problems

Write code to invert a Map; that is, to make the values the keys and make the keys the values.

```
Map<String, String> byName =
    new HashMap<String, String>();
byName.put("Darwin", "748-2797");
byName.put("Newton", "748-9901");

Map<String, String> byPhone = new HashMap<String,
String>();
// ... your code here!
System.out.println(byPhone);
```

Output:

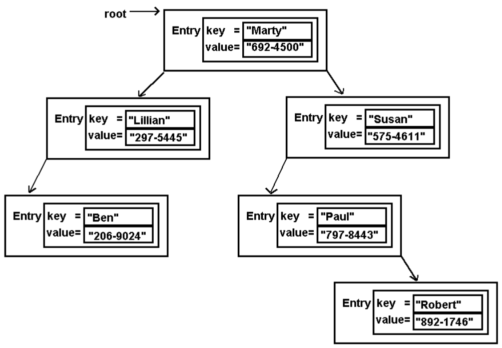
```
{748-2797=Darwin, 748-9901=Newton}
```

Write a program to count words in a text file, using a hash map to store the number of occurrences of each word.

Implementing maps using trees and hash tables

Implement Map with a tree

Make a BST of entries, sorted on the keys. Each entry also contains the associated value

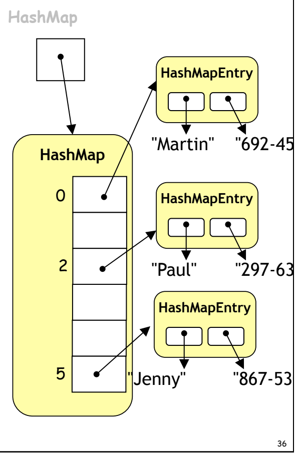

root
Entry key = "Marty" value= "692-4500"
Entry key = "Lillian" value= "297-5445"
Entry key = "Susan" value= "575-4611"
Entry key = "Ben" value= "206-9024"
Entry key = "Paul" value= "797-8443"
Entry key = "Robert" value= "892-1746"

Implement Map with hash table

make a hash table of entries, where each key's hash code determines the position

the entry also contains the associated value

search for the key using the standard O(1) hash table lookup algorithm, then retrieve the associated value



`Map` implementations in Java

`Map` is an interface; you can't say `new Map()`

There are two implementations:

1. `TreeMap`: a (balanced) BST storing entries
2. `HashMap`: a hash table storing entries

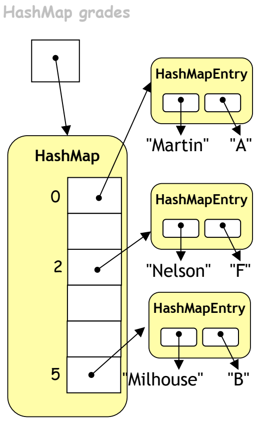
`HashMap` example

```
Map<String, String> grades =
    new HashMap<String, String>();
grades.put("Martin", "A");
grades.put("Nelson", "F");
grades.put("Milhouse", "B");

// What grade did they get?
System.out.println(
  grades.get("Nelson"));
System.out.println(
  grades.get("Martin"));

grades.put("Nelson", "W");
grades.remove("Martin");

System.out.println(
  grades.get("Nelson"));
System.out.println( grades.get("Martin
"));
```



Compound collections

Collections can be nested to represent more complex data

example: A person can have one or many phone numbers

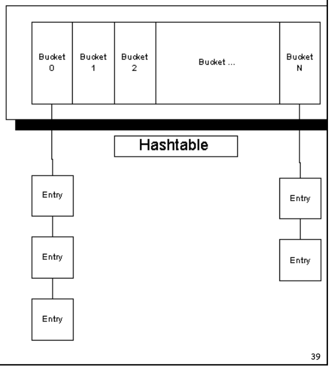
1. want to be able to quickly find all of a person's phone numbers, given their name

implement this example as a HashMap of Lists

1. keys are Strings (names)
2. values are Lists (e.g ArrayList) of Strings, where each String is one phone number

String List<String>
name --> list of phone numbers

"Phil" --> ["234-8793", "439-8575", ...]



Compound collection code 1

```
// map names to list of phone numbers
Map<String, List<String>> m = new HashMap<String,
List<String>>();
m.put("Marty", new ArrayList<String>());
...

List<String> list = m.get("Marty");
list.add("253-692-4540");
...

list = m.get("Marty");
list.add("206-949-0504");

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

```
[253-692-4540, 206-949-0504]
```

Compound collection code 2

```
// map names to set of friends
Map<String, Set<String>> m = new HashMap<String,
Set<String>>();
m.put("Marty", new HashSet<String>());
...
Set set = m.get("Marty");
set.add("James");
...
set = m.get("Marty");
set.add("Mike");
System.out.println(set);
if (set.contains("James"))
  System.out.println("James is my friend");

{Mike, James}
James is my friend
```

Objects and Hashing: `hashCode`

`HashMap` uses `hashCode` method on objects to store them efficiently (O(1) lookup time)

1. `hashCode` method is used by `HashMap` to partition objects into buckets and only search the relevant bucket to see if a given object is in the hash table

If objects of your class could be used as a hash key, you should override `hashCode`

1. `hashCode` is already implemented by most common types: String, Double, Integer, List

Overriding `hashCode`

General contract: if `equals` is overridden, `hashCode` should be overridden also

Conditions for overriding `hashCode`:

1. should return same value for an object whose state hasn't changed since last call
2. if `x.equals(y)`, then `x.hashCode() == y.hashCode()`
3. (if `!x.equals(y)`, it is not necessary that `x.hashCode() != y.hashCode()` … why?)

Advantages of overriding `hashCode`

1. your objects will store themselves correctly in a hash table
2. distributing the hash codes will keep the hash balanced: no one bucket will contain too much data compared to others

Overriding `hashCode`, cont'd.

Things to do in a good `hashCode` implementation

1. make sure the hash code is same for equal objects
2. try to ensure that the hash code will be different for different objects
3. ensure that the hash code value depends on every piece of state that is important to the object
4. preferrably, weight the pieces so that different objects won't happen to add up to the same hash code

```
public class Employee {
    public int hashCode() {
        return  7 * this.name.hashCode()
             + 11 * new Double(this.salary).hashCode()
             + 13 * this.employeeID;
  }
```