CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 8

February 27, 2014

Rehashing: Resizing the Hash Table

- It is important to keep hash tables less than half full for good performance.
- This can be quite critical for performance.
- With Java, its particularly easy to resize a hash table because you dont have to carefully dismantle the old one – you can point to a new table and just leave the old table to the garbage collector to clean up.

Rehashing

```
private void rehash( )
{
    HashEntry [ ] oldArray = array;
    // Create a new, empty table
    allocateArray( nextPrime( 4 * size( ) ) );
    currentSize = 0;
    occupied = 0;
    // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( isActive( oldArray, i ) )
            add( (AnyType) oldArray[ i ].element );
}
```

Performance of Rehashing

- Typically, we rehash every time the number of entries exceeds 2n, starting at the original half-full point.
- Consider doing a sequence of inserts to an empty hash table of size 64+, prime (M = first prime over 64, that would be 67):
- insert 1, 2, 3, ..., 31 – still less than half full.
- insert 32 – first rehash, to M=128+, prime
- insert 33, 34, 35, ..., 63 – still less than half full ($32 = 2^5$ ops)
- insert 64: second rehash, to M=256+, prime ($64 = 2^6$ ops)

Performance of Rehashing

- In general, during inserts 2n to 2n+1, we do 2n simple inserts, each $O(1)$, plus 2n+1 relocations into a new hash table, each also $O(1)$.
- This totals $O(n)$ inserts.
- If we amortize this over 2n inserts, that is, average the total time over the total number of inserts between rehashes, 2n, we get O(1) per insert.
- Naturally there's a "hiccup" when the expansion happens, and this can be unacceptable if it exceeds some threshold time such as human-perceivable time in interactive programs.
- There are more sophisticated extensible hashing methods that don't hiccup so badly.

Hashing in Memory and on Disk

- The hash table may be located in memory, supporting fast lookup to records on disk, or even on disk, supporting fast access to further disk.
- In fact, a disk-resident hash table that is in frequent use ends up being in memory because of the memory "caching" of disk pages in the file system.

keys	hash table	Data records	Example
memory	memory	memory	typical HashMap apps
memory	memory	disk	use HashMap to hold disk record locations as values
memory	disk	disk	hashed files, some database tables

Hashing Packages in Programming Language libraries

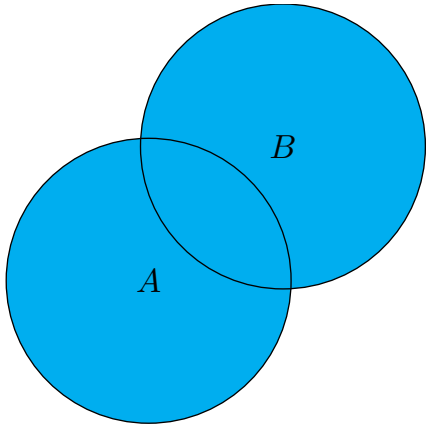
- The C library sticks to basics and has no collection support.
- We've looked at the Java Collections HashMap and HashSet.
- The STL (Standard Template Library) of C++ provides Maps, but their performance (HP implementation) is logarithmic, and thus indicates they are implemented with trees, not hash tables.
- Microsoft's .NET/C# provides collection classes: see `http://msdn.microsoft.com/en-us/library/0ytkdh4s(VS.85).aspx`.

Set Operations

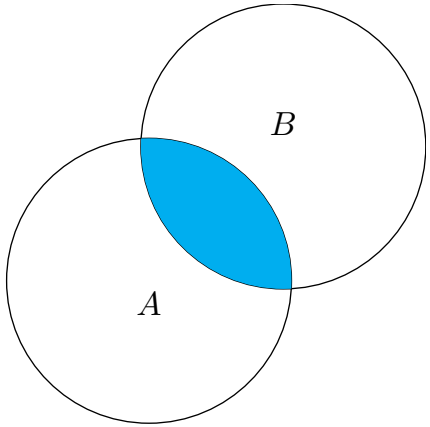
- Consider set of integers: A = {1, 5, 3, 96}, or B = {17, 5, 1, 96} and a set of strings, {"Mary", "contrary", "quite"}.
- Sets have no duplicates and order doesn't count. (List allows duplicates and order does count.)
- The Set interface contains just the basic set operations. We would like to use Union, Intersection, and Difference.

Basic Set Operations - Reminder

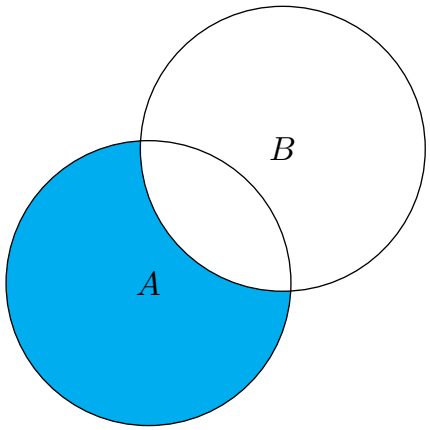
$A \cup B = \{1, 3, 5, 17, 96\}$



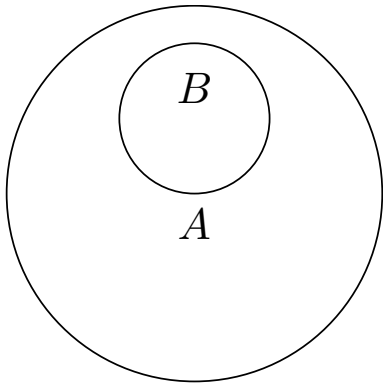
$A \cap B = \{1, 5, 96\}$



$A - B = \{3\}$, $B - A = \{17\}$



$B \subset A$



Set Operations – Examples

- Consider Sets made with HashSet:
  `Set a = new HashSet(); ...`
- Union of Set a and Set b: we can addAll of b to a or vice versa, to form the union set:

```
a.addAll(b) // a is the union of a and b
b.addAll(a) // turns b into the union
```

- If we don't want to change a or b, make a new set from one of them first:

```
Set union = new HashSet(a);
union.addAll(b); // form a UNION b in own set
```

- For subset containment, we just use containsAll:

```
a.containsAll(b); //returns true if b is a subset
of a
```

Intersection Example

```
a.retainAll(b);
// turns a into intersection of a and b
b.retainAll(a); /// turns b into intersection
Set intersection = new HashSet<T>(a);
intersection.retainAll(b); // form a INTERSECT b in
own set
```

Difference Example

Notice that $A - B$ and $B - A$ are different.

```
a.removeAll(b);
// turns a into a - b
b.removeAll(a);
// turns b into b - a
Set<T> diffAB = new HashSet<T>(a);
diffAB.removeAll(b);
// form a - b in own set
Set<T> diffBA = new HashSet<T>(b);
diffBA.removeAll(a);
// form b - a in own set
```

Implementation of Union, Intersection

- UNION via a.addAll(b):
  - Iterate through b, checking elements for membership "contains()" in a, an O(1) op.
  - Any that aren't there, add (by ref copy, as usual) to result set, also O(1).
  - For N elements of B, O(N) operation.
- INTERSECTION via a.retainAll(b):
  - Iterate through a, checking elements for membership in b.
  - If not in b, remove it (O(1)). For N elements of A, an O(N) operation.

Implementation of Difference

For a.removeAll(b) it is possible to be a little smarter.

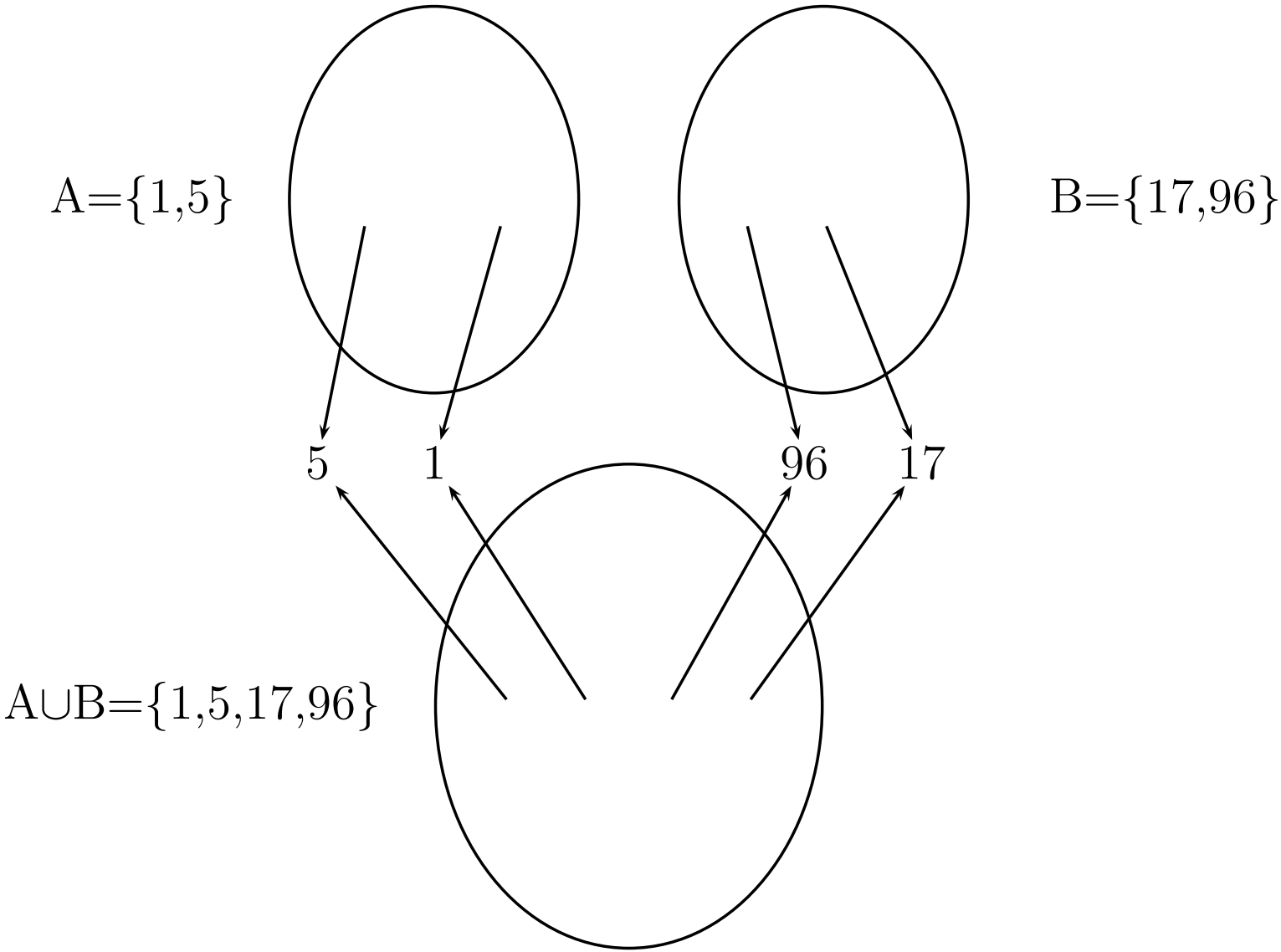
if (a.size() ¡ b.size())
    iterate through a, removing (from a) those that are in b
else
    iterate through b, removing those from a

This way, only the smaller set is scanned.

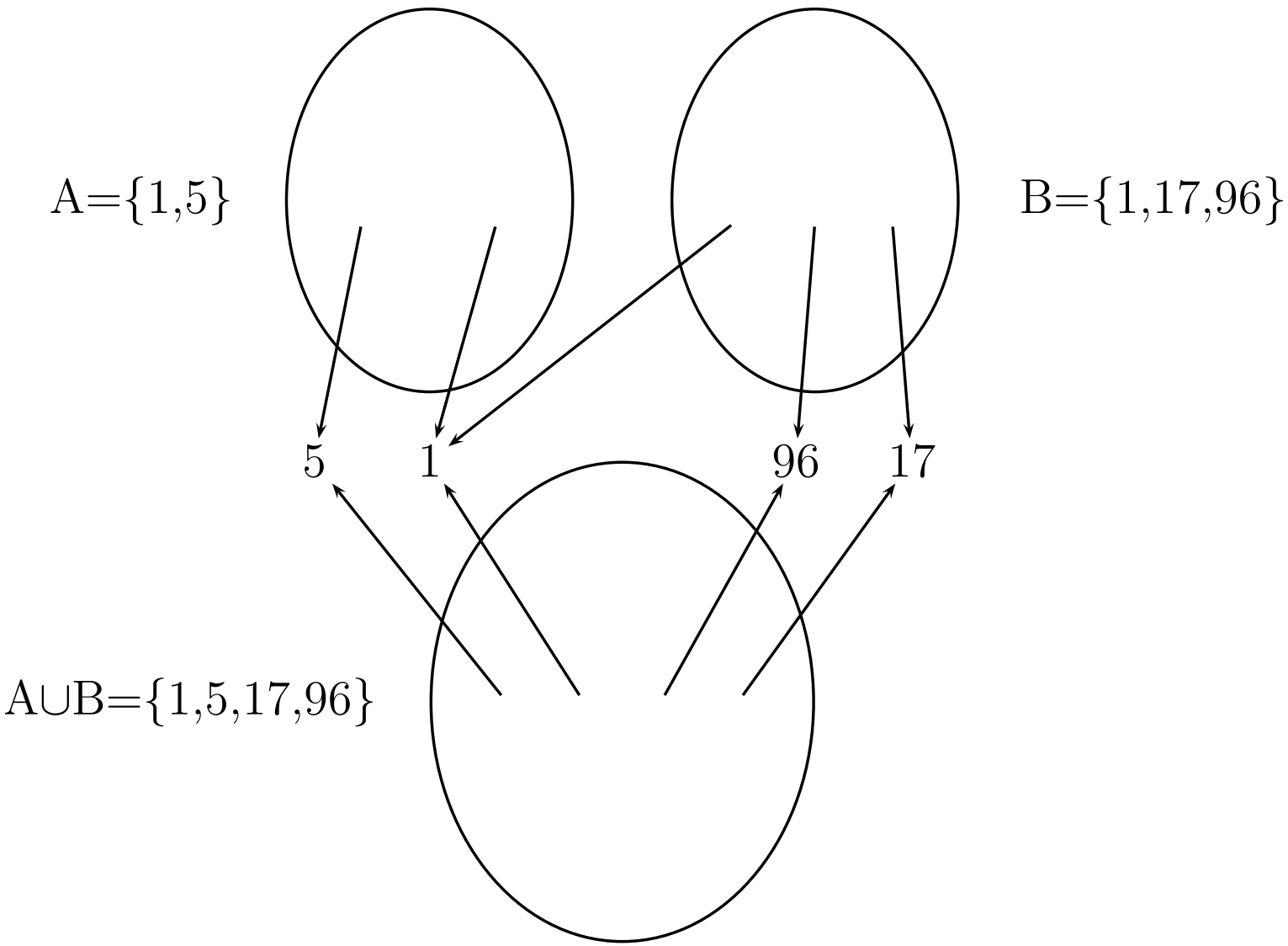
How Does It Look With the Element Objects?

- In Java we allow many ref's to one object, and when we call "new Hashset(a)" we are not copying the element objects but just their refs, a "shallow" copy.
- Then the addAll copies more element refs to the new set.
- Element matching is done via element.equals().
- And if we're using HashSet, we need a consistent hashCode for the elements as well.

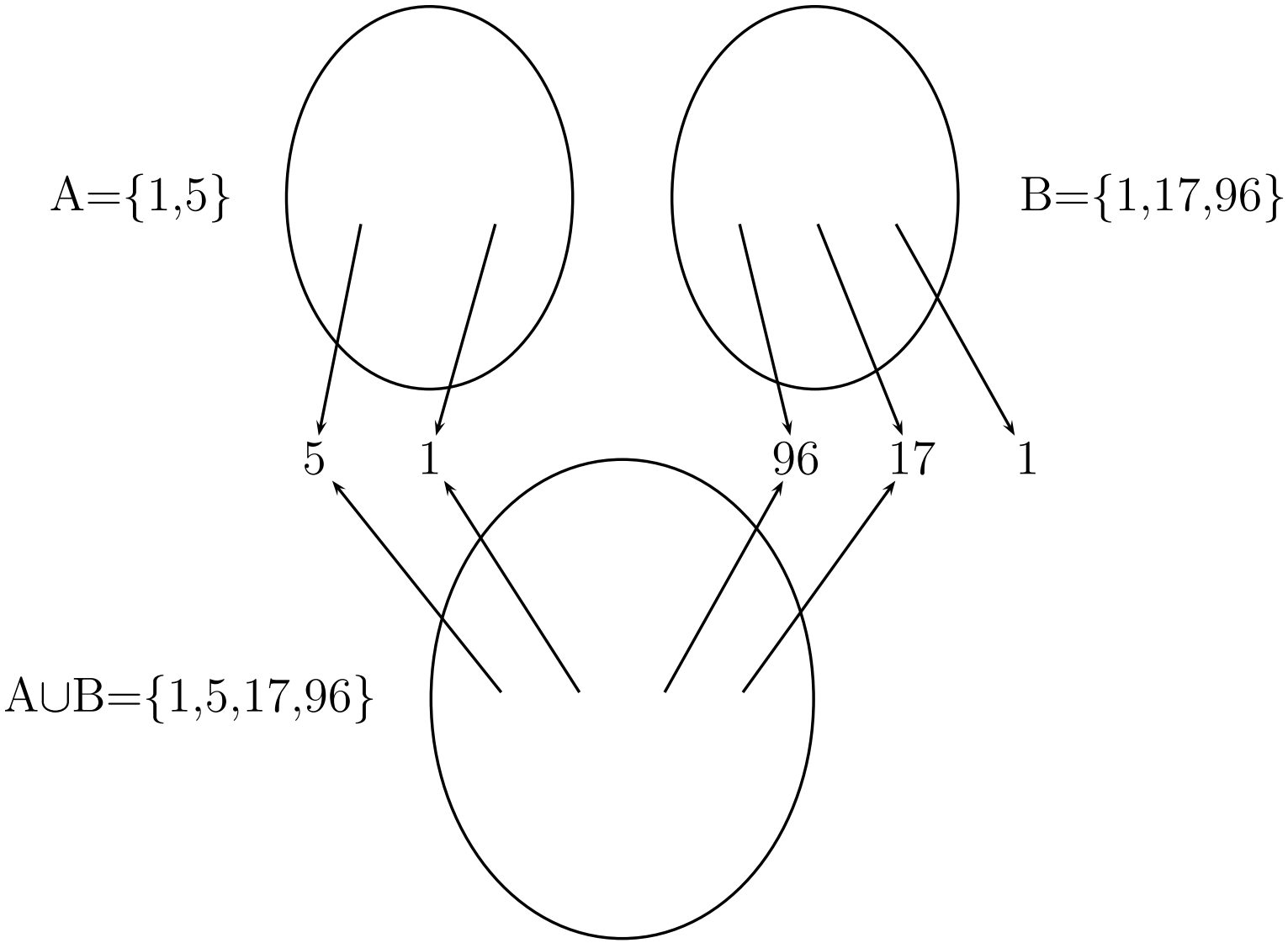
Disjoint Sets and Their Unions



Sets With Shared Elements and Their Union



Sets With Equal But Not Shared Elements and Their Union



Toy Example Revisited: Vowels

- The original pseudocode said

  if (c is a vowel)

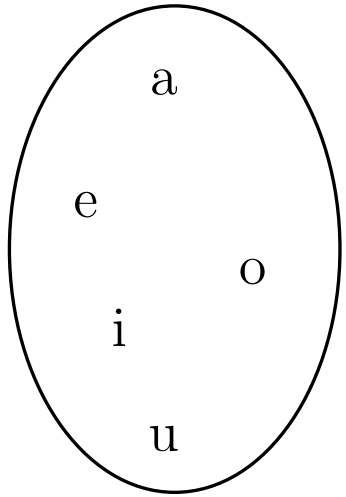
  count it

- How did we code the condition here?

```
String vowelStr = ‘‘aeiou’’;
Set vowels = new HashSet();
for (int i=0;i<vowelStr.length(); i++)
    vowels.add(new
    Character(vowelStr.charAt(i)));
```

- Later:

```
if (vowels.contains(c))
    // c a char:  O(1) lookup
    count++;
```



Toy Example Revisited – Vowels

- Of course for 5 items this is not important, but for large sets, it really matters.
- We have gone from an O(N) search to an O(1) lookup, a big savings.
- So when you see a loop of tests, think could I use a Set here?
- We just saw how set-thinking allowed us to replace an O(N) search with an O(1) lookup when we needed to classify something.
- But to see the full pay-off of set-thinking we need to use the set-crunching operations of union, intersection, etc.

Example from Qualifying Exam

- 500 terminal lines, each terminal line has a Set of usernames seen on the line, so this defines 500 Sets of username.
- For example, the set of users seen on line 1, set of users seen on line 2, etc.
- We could hold this all in an ArrayList<Set<String>> if we wanted. (But not in an array of Set<String>, unfortunately, by an annoying rule of generics.)

Example from Qualifying Exam

- How do you find all users ever seen on any line?
  - **Answer:** take the union of all the sets.
- How do you find if anyone (or several users) logged in on all lines?
- Suppose one line is for a public terminal suspected of break-in attempts.
- The set of users who logged in on that line are considered suspicious.
- How do you find all the lines that any of these users logged in on?

More on Inner Classes

- Handout "Example of a private inner class providing an iterator"
- For hw3, you tried out some uses of nested classes.
- Why do we need private inner classes?
- Answer: We could live without them, but they are convenient if the detail they represent has a close relationship to the outer class, and is suited to share the outer class type parametrization.
- For an iterator over HashSet<T> for example, we need an Iterator<T> object to return to the client, so we set up the private inner class HashSetIterator.
- But we need to return this Iterator object to the caller **outside HashSet**, so how can we restrict ourselves to a private inner class?

More on Inner Classes

- Answer: With the power of Java interfaces. The Iterator<T> interface is public. We define a private object of type that ISA Iterator<T>, and return it to the caller as an Iterator<T>.
- The caller never knows the real type of the object, and doesn't care either. It works fine as an iterator.
- Look at MyContainer, pg. 578(522). Not parametrized, but we can fairly easily fix it. See the handout. Add <T> to lines 3, 10, end of 14, and on next page, end of line 2.
- Change Object to T on lines 21 and 9 on the next page, and a cast to T, (T), on lines 22 and 10 on the next page.
- Note that LocalIterator is written two ways, the second one using the special relation an inner class has to the outer class fields.

More on Inner Classes

- It needs a "remove" method to implement the JDK Iterator<T> interface. See the handout.
- We added <T> to MyContainer's declaration, but not to LocalIterator. LocalIterator gets the parameter automatically because it's non-static.
- In fact, adding <T> to LocalIterator causes an unwanted parametrization shadowing the original one. Nasty bug. Need a good IDE to show warning right away.
- Now understand line 6, pg. 789(728) HashSetIterator: an inner class of HashSet<AnyType>, so itself parametrized by <AnyType>.

More on Inner Classes

- Note that inner class code can use outer class fields as easily as its own fields. Look at the line of code in LocalIterator's hasNext():
  current $<$ size
- Here current is a field of the inner class and size is a field of the outer class. It's that easy.
- This is implemented by the inner class having a "secret" reference to the outer class object that it must have. See pg. 577.

More on Inner Classes

- What about nested classes – can we refer to the outer object's fields?
- Yes, if we have a ref to the outer object, but that requires some programming on our part.
- For Book of SimpleStudent, we could add a field outerRef and invent a constructor for Book that set it.
  public Book (SimpleStudent ss) { outerRef = ss; }
- Then the SimpleStudent object code that created a Book could use Book(this) and get it set up.
- The Book code could then use outerRef.name to access the student's name. (Probably should call the outer ref "student" here.)

More on Inner Classes

- Don't forget that static inner classes don't get outer class type parameters–
- HashEntry is a static inner class, so lack of <> after HashEntry means it has no type parametrization.
- The code in HashEntry can use no outer-class parametric types, and has none of its own.

More on Inner Classes

- In general, take union of parameters of the inner class and its outer class:
  `public class Foo<X,Y> ...`
  `private class Bar<Z> ...`
  This means types X,Y, and Z are usable in Bar's code.
- Bar can implement or extend Mumble<X>– Mumble<X>'s API is followed–some types show up in the API, others are involved in implementation of Foo, or both.
- More challenging case – in TreeMap code, but useful for 5b, the challenge option of pa2.
- Cases involving multiple parametric types:
- pg. 753 (695) ViewClass<AnyType>, inner class of MapImpl<KeyType, ValueType>
- This class is parametrized 3 ways! That means the code in this class is allowed to use these 3 types.

Application – Movie Rental

- Movie Rentals, inspired by sign in a store: "If you liked X, try Y"
- Suppose we get our patrons to mark watched movies as one of two options: "liked movie" and "disliked movie".
- Then our media handler could enter this data into the customer-movie database.
- Movies have well-defined identifiers, movie titles, or title-year if needed.
- We want our computer to print out these prediction signs, based on our like-dislike data from our own customers.

Application – Movie Rental

- After a while we would have access to the movie like-sets and dislike sets of our customers.
- This is now happening in a lot of industries, and is summarized by the phrase "market of one" data.
- Looking from the movie angle, we have sets of customers who liked it and disliked it.
- The simplest analysis is simple movie popularity: popularity = Size(likeset)/[Size(likeset) + Size(dislikeset)].

Application – Movie Rental

- But what about these predictions from X to Y?
- In the population, some people liked both X and Y, some liked X only and some liked Y only and some disliked both.
- If the first and last of these are relatively large compared to the middle ones, we have correlation between them: they tend to go together, like or dislike.
- But we don't actually care about the dislike-X case for our prediction signs.
- We are interested in all the cases where people liked X – then what about Y?The chance that such a person will like Y is: probability(like Y | like X) = Size(like(X) $\cap$ like(Y)) / Size(like(X))

Application – Movie Rental

- Similarly if an individual customer wants a recommendation for a movie they haven't yet seen we could use these ratios to reach out from each movie that the customer has seen and liked and find the most predictably likable movie.
- We could also use dislike prediction as well. At some point we would need to bring in a real statistician!
- After a while our customers think we're great and really come to depend on our predictions.Then they become more predictable themselves.
- Although it is impossible to keep enough movies on hand to keep everything available, we now have a notion of a good substitute: if X predicts Y, then Y is a good substitute for X.
- Each movie has a substitute list based on some predictability level we've established.

How Do We Implement This?

- Each movie has a substitute set.movie$\rightarrow$ {substitute movies} i.e., a Map.
- We also have movie quantity-on-hand, its inventory.
- We can combine these into one master Map,using notation (x,y) for an object with fields x,y:
  movie $\rightarrow$ (qoh, {substitutes})
- Here the domain element is a movie, id'd by String movie title. The range element is an object with an int qoh and a Set of movies, i.e., Set of Strings.
- Pretty easy to work with since String comes with equals and hashCode, ready for HashMap.
- Implementation: `HashMap<String, InventoryData>` where InventoryData is a class which HASA Set<String> as well as an int "qoh".