

Handed out: Thursday, February 28, 2008

Due: Wednesday, March 5, 2008, by 10PM using Web Turnin

Testing and documentation are a major focus of this assignment, but you will have to do some programming as well. This will be your first pair programming assignment. Think of it as training for the final project ;-)

Keywords using Bitstrings:

What's a bitstring? It's a sequence of bits with each bit representing something larger. A byte (eight bits in memory) is a bitstring of length 8. A bit is a binary digit: it can be either 1 or 0.

Your task is to reimplement the `icritters.base.KeywordCollection` class using a bitstring of length 63. You will use a private instance variable of type `long` as the internal representation of your new data structure. Note: Because Java uses 1 of the 64 bits of a `long` as an indicator of the sign (positive or negative) of the value, we will only use bits 0-62 for a total of 63 bits.

To represent our data, each keyword (a `String`) gets assigned a particular bit index (an `Integer` between 0 and 62, inclusive). If that bit is on (a '1' value), then the keyword exists in that `KeywordCollection`. This allows us to store a potentially large collection in a small space. Imagine having an `ArrayList` with 30+ `Strings` in every `ICritter`; you'd use a ton of memory. The bitstring implementation compresses the collection into a single `long` per collection (plus the lookup `HashMap` described below).

Basic Binary:

Before going further, it is necessary you understand the basics of binary. We typically use decimal in everyday life. Decimal is a base 10 system; meaning every digit can have 10 distinct values. As numbers grow beyond the size of one digit, they become the sum of powers of the base. The number 1987 is interpreted to mean

$$\begin{array}{cccc} \text{digit 3} & \text{digit 2} & \text{digit 1} & \text{digit 0} \\ 1000*(10^3) & + 900*(10^2) & + 8*(10^1) & + 7*(10^0) \end{array}$$

(x^y means x to the y th power)

Binary works the same way, but with a base 2. 1010 means

$$\begin{array}{cccc} \text{bit 3} & \text{bit 2} & \text{bit 1} & \text{bit 0} \\ 1*(2^3) & + 0*(2^2) & + 1*(2^1) & + 0*(2^0) \\ 1*8 & + 0*4 & + 1*2 & + 0*1 \\ 8 & + 0 & + 2 & + 0 = \text{decimal 10} \end{array}$$

Make sense? If you're still not feeling comfortable, ask your section leader for more detail, check the technical notes links below, or search the interwebs.

Using bitstrings to represent the collection:

Your implementation of KeywordCollection relies on the following pieces:

A bitstring (type long) which you will use simply as a place to mark what bits (which correspond to keywords) are on (indicating the keyword is present in the collection) or off (keyword is not present in the collection). There will be one bitstring in each KeywordCollection object.

A keywordToBit lookup table: you will use this table to maintain and lookup what bit indices correspond to a given keyword. There should only be one table for the KeywordCollection class. You could implement a private helper method that could be used like this:

```
getBitForKeyword( "Eggs" ) ----> returns 3 (given example below)
```

You must have the ability to lookup a keyword (String) given a bit index as well. You could also have another private helper method like this:

```
getKeywordForBit( 7 ) ----> returns "Zoos" (given example below)
```

Implementation details follow, but first here's an example of how these pieces interrelate.

In the example below, bits 5 and 0 are on. This means that this KeywordCollection contains the keywords "Music" (bit 5) and "Lampreys" (bit 0).

```
bit value    0 ... 0 0 1 0 0 0 0 1
bit number  62 ... 7 6 5 4 3 2 1 0
```

```
(decimal value (2^5) + (2^0) = 33)
```

```
keywordToBit {"Lampreys":0}, {"Jazz":1}, {"Art":2}, {"Eggs":3},
              {"Anime":4}, {"Music":5}, {"Manga":6}, {"Zoos":7}
```

Again, each KeywordCollection instance will have its own bitstring of type long. This bitstring tells us (via its "on" bits) what keywords the collection contains.

The keywordToBit table above gives us a way of looking up a keyword to get the bit index. There should be a single lookup table that is used by all KeywordCollections. To go backwards, (from a bit index to keyword) you could keep a reverse lookup array (it would be an array of Strings, where the String at location 0 is the keyword that corresponds to bit 0 and so on), or you can do it on the fly though that will require more work. Note that there would also only be one reverse lookup array (if you implement this) shared by all the KeywordCollection objects as well.

The keywordToBit lookup table should be implemented as a HashMap<String,Integer> as a class (not instance) variable (since there should only be one for all the KeywordCollections).

Note that in the example above the keywordToBit table is in order of the bit indices, this will probably not be the case in typical usage; HashMaps do not have an order.

Explore the HashMap API to understand how to lookup and add items to a HashMap. You will be using the table to map from keywords (Strings) to respective bit indices (Integers). You should also be able to figure out how to go backwards from bit indices (Integers) to keywords (Strings), though it requires more work. Keeping a reverse lookup array is also acceptable as mentioned above.

You may well say, "I could just keep the array of Strings and iterate down it to find the String I'm looking for, why do I need the HashMap?" The usual answer is speed. Iterating down the array of Strings is very slow compared to using the HashMap. Not only are HashMaps faster, but they are also very convenient. To know HashMaps is to love HashMaps.

The keywordToBit HashMap will have a maximum of 63 entries. You will have to keep an eye on the number of entries - if the user tries to add more than 63 keywords, your code must throw the exception detailed below. The HashMap will start out as empty, and have new keywords added to it as they are encountered. In this implementation, keywords are never removed from the HashMap, though keywords may be "removed" from a collection (which would simply turn those bits off in the collection's bitstring).

A convenient helper method for the KeywordCollection class would be the following:

```
private boolean isBitOn(int place)
```

This method determines a bit's status. It takes an int value representing which digit is to be examined (how many places from the right is the bit under question). It will return true if the corresponding bit is on (a '1' value) in this KeywordCollection object's bitstring, and false otherwise.

Before you start trying to solve this using a lot of math, start thinking in bits. We can use what's called a 'bitmask' to determine if a bit is on or off. Say you're looking for the place 1 bit, the mask is equal to the value of the place (with the decimal value of 2 ($2^1 = 2$) in this case.) If we do a bitwise AND on the bitstring and the mask, there are two outcomes: if the bit was on, the result is not zero, otherwise the result is zero. The picture form is below, and Java code is on the following page:

EX1: bit #1 off

```

                                v
bitstring  0 1 1 1 0 1 0 1
mask       0 0 0 0 0 0 1 0
-----
           0 0 0 0 0 0 0 0   Result of bitwise and
                                ^

```

EX2: bit #1 on

```

                                v
bitstring  0 1 1 1 0 1 1 1
mask       0 0 0 0 0 0 1 0
-----
           0 0 0 0 0 0 1 0   Result of bitwise and
                                ^

```

The corresponding Java for this example section is:

```
int bitIndex      = 1;
int bitMask      = Math.pow( 2,  bitIndex ); // 2^1 = 2

// or you can use bit shifting
// int bitMask    = 1 << bitIndex; // The shift operator

int ex1Bitstring = 117;           // 0 1 1 1 0 1 0 1 in binary
int ex2Bitstring = 119;           // 0 1 1 1 0 1 1 1 in binary

if(( ex1Bitstring & mask ) == 0 ) {
    System.out.println( "ex1 has bit off" );
} else {
    System.out.println( "ex1 has bit on" );
}

if(( ex2Bitstring & mask ) == 0 ) {
    System.out.println( "ex2 has bit off" );
} else {
    System.out.println( "ex2 has bit on" );
}
```

When run, this prints:

```
ex1 has bit off
ex2 has bit on
```

As we can see, the result of the AND operation is dependent on the bit in the value place. Use that knowledge.

KeywordCollection must implement the following methods:

```
public boolean addKeyword(String keyword)
    throws icritters.exceptions.TooManyKeywords
```

Add keyword to collection. If keyword already existed in the collection then true is returned, otherwise false. If a keyword already existed and is added again, it still returns true. If keyword was not seen before then a bit index is assigned in the translation map. This throws the TooManyKeywords exception if the user tries to add more than 63 keywords.

```
public boolean removeKeyword(String keyword)
```

Remove keyword from collection. Returns true if keyword existed in collection, otherwise false. If keyword was not seen before then a bit index will not be assigned in the translation map.

```
public List<String> listKeywords()
```

Return a list of the keywords in this collection.

```
public static List<String> listLookupKeywords()
```

Return a list of the keywords in the keywordToBit HashMap table. You can think of this as returning a list of all of the keywords that have ever been added to any of the collections, whereas listKeywords returns only the keywords in the collection it was called on.

```
public void clearKeywords()
```

Clear all keywords from this collection. This does not affect the keywordToBit HashMap table.

```
public boolean containsKeyword(String keyword)
```

Returns true if this set contains the specified keyword. If keyword was not seen before, then a bitnumber is not assigned.

```
public static boolean lookupContainsKeyword(String keyword)
```

Returns true if the keywordToBit HashMap table contains the specified keyword.

```
public boolean isEmpty()
```

Returns true if this collection contains no keywords.

```
public int size()
```

Returns the number of keywords in this collection. (Hint: there are many ways to do this using bitwise operations. And there are other tactics that will work fine as well.)

```
public int lookupSize()
```

Returns the number of keywords in the keywordToBit HashMap table.

```
public double correlation(KeywordCollection otherCollection):
```

Returns the percentage of keywords that match between this collection and the other collection. This should have the same results as in the last assignment.

icritters.exceptions.TooManyKeywords exception:

As mentioned above, we can't manage more than 63 keywords in this implementation, so your code must be sure to keep track of the number of keywords added, and throw a TooManyKeywords exception if we ever attempt to cross that limit. TooManyKeywords should not be of type RuntimeException, and just have a single constructor that takes a String argument. It should be in its own class, in the icritters.exceptions package.

Documentation:

You will be required to fully comment using javadoc comments. You should have an @Author tag and description at the start of every class, and you should also use, at the beginning of every method, the @param, @return, and @throws tags as appropriate. (i.e. If your method doesn't throw an exception, you don't need an @throws tag.) We will be generating HTML from your javadocs.

Testing:

You are required to test your KeywordCollection code fully using JUnit tests. We will be running our exhaustive tests on your version of KeywordCollection. This would be whitebox testing. If we find a bug, it means you didn't test enough!

Your JUnit tests will also be run against a buggy implementation of the KeywordCollection implementation, so writing good tests is crucial. This would be blackbox testing. Make sure you turn in your test classes!

Coding style will also be a grading criteria.

