COM1020/COM6101: Further Java Programming

AKA: Object-Oriented Programming, Advanced Java Programming

http://www.dcs.shef.ac.uk/∼sjr/com1020/

Lecture 7: Collections Accessed by Content

Steve Renals

Department of Computer Science
University of Sheffield
Sheffield S1 4DP
UK
email: s.renals@dcs.shef.ac.uk

Reading

- Horstmann and Cornell, *Core Java 2 vol. 2*, chapter 2
- Sun Java Tutorial: Java Collections Framework (http://www.javasoft.com/docs/books/tutorial/collections/)
- JavaWorld: Get started with the Java Collections Framework (http://www.javaworld.com/javaworld/jw-11-1998/jw-11-collections.html)
- Java Solutions: The Java 2 Collections (http://www.cuj.com/java/articles/a1.html)

Objectives

This lecture will consider collections in which you primarily access elements by their *content*: eg, finding words in a dictionary

Lecture Content:

- Hash Tables and Hashing
- HashSet and the Set interface
- Accessing by content with sorting for free: TreeSet
- The Comparable interface

A Dictionary

Consider this problem:

Given a text, create a set of all the words used in that text.

For example, the (short) text:

“the element in the collection in the program”

contains the set $\{$the, element, in, collection, program$\}$

To do this we need to maintain the set of words that have been observed. For each new word, we need to check if we have already seen it, and if not add it to the set.

(To avoid thinking about I/O, tokenization, etc. assume that the input text is stored as a list of tokenized strings.)

A Solution

The Collection interface has the appropriate method, contains.

Using an ArrayList implementation we have:

```
Collection inputText;
. . .
Collection wordSet = new ArrayList();
String wd;
for(Iterator i = inputText.iterator; i.hasNext();){
  wd = (String) i.next();
  if(wordSet.contains(wd) == false)
    wordSet.add(wd);
}
```

This is not efficient. ArrayList.contains must step through the ArrayList, applying method equals to each element returning when either:

1. A matching element is found
2. The end of the list is reached.

LinkedList would have similar (in)efficiency.

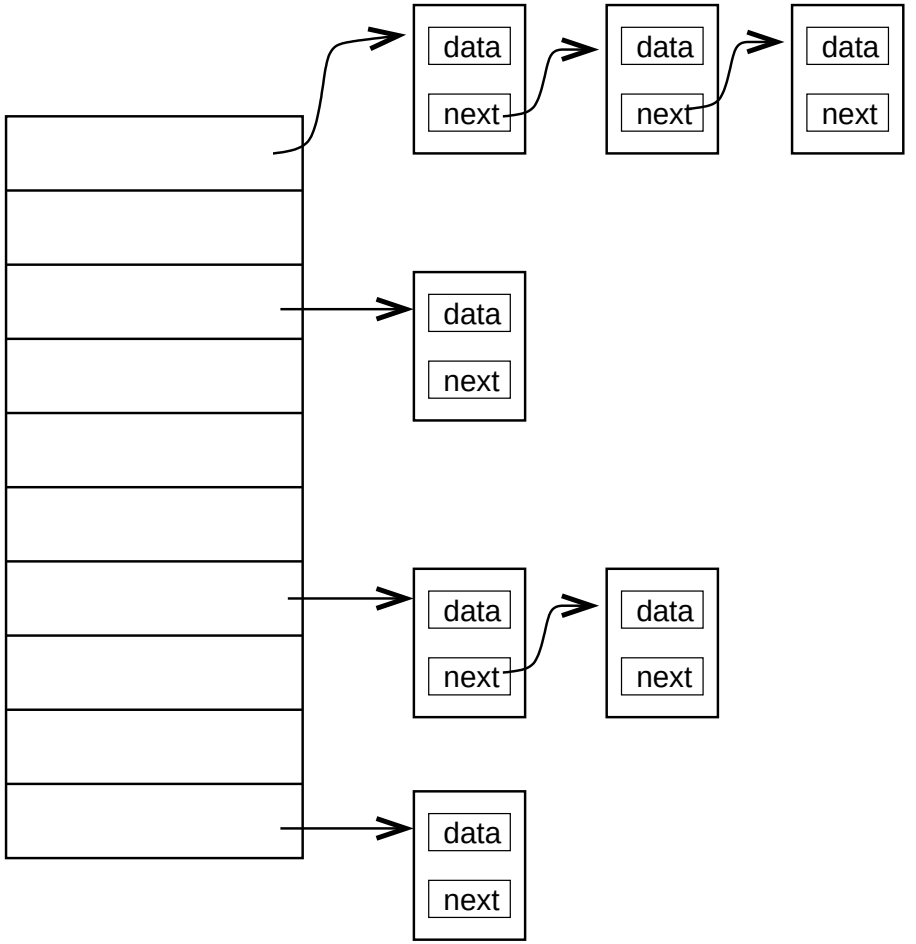
Hash Tables

A *hash table* is a data structure for access by content.

- An integer *hash code* is computed quickly for each object. For a string, the hash code would be some function of the characters that make up the string.
- A hash table is an array of linked lists, each list is called a *bucket*
- To find the index of an object (obj) in a table of N buckets:

$$\text{index} = \text{hash}(obj)\%N$$

  For example the default hash table has 101 buckets;
  `"Java".hashCode()` = 2301506
  So `"Java"` would have index $2301506\%101 = 19$
- **Insertion**: Once the index is computed, look in that bucket:
  - Bucket is empty, so insert object into bucket
  - *Hash Collision*: bucket has elements, must compare with all bucket elements to see if object is already present


data
next
data
next
data
next
data
next
data
next
data
next
data
next
data
next

Controlling Hash Tables

Bucket Count How many buckets are needed? There is a lot of theory on this; these two reasonable rules of thumb reduce the probability of collisions without wasting too much space:

- 1.5$\times$ as many buckets as expected number of elements
- Make the bucket count a prime number

Rehashing When a hash table gets too full, a new table (with more buckets) is created, all elements inserted into the new table and the old table is discarded.

Load Factor The load factor determines when the table is “too full”. Eg, when the load factor is 0.75, the table is rehashed when it is over 75% full

HashSet class The Java Collections Framework supplies a class called HashSet which does all this for you. It will provide default values for the initial capacity (101) and load factor (0.75)

Using HashSet

```
Collection inputText;
. . .
Set wordSet = new HashSet();
String wd;
for(Iterator i = inputText.iterator; i.hasNext();){
  wd = (String) i.next();
  // no need for wordSet.contains(wd)
  wordSet.add(wd);
}
```

- HashSet implements the Set interface
- Set.add adds the specified element to this set if it is not already present
- HashSet.contains is more efficient than ArrayList.contains!
- If we know we are likely to have about 20 000 elements in the HashSet we can set the initial capacity accordingly:

```
// set capacity to the smallest prime > 1.5*20,000
Set wordSet = new HashSet(30011);
```

Hashing Your Own Objects

- wordSet.add(wd) works by calling the String.hashCode method to work out where to insert the item in the hash table
- String.hashCode obtains its value by computing a function of the characters in the string
- hashCode is defined in Object (although - as usual - the default definition is not very useful.
- *Always* define a hashCode method for objects you will insert in a hash table
- hashCode should be a function that combines the (hash codes of) the relevant identifiers for the object.
- equals must also be redefined, so that the hash table knows how to tell if an object has already been inserted in the table
- These definitions must be compatible:
  If x.equals(y)
  then x.hashCode() == y.hashCode()

Hashing Example (1)

```
public class Person {
  . . .
  public int hashCode() {
    return 13*surname.hashCode() + 17*firstName.hashCode();
  }

  public boolean equals(Object obj) {
    if(obj == null || getClass() != obj.getClass())
      return false;
    Person per = (Person)obj;
    return surname.equals(per.surname) && firstName.equals(per.firstName);
  }
  . . .
  private String surname;
  private String firstName;
}
```

Hashing Example (2)

```
public class AutoPart {
 . . .
  public int hashCode() {
    return 13*partID;
  }

  public boolean equals(Object obj) {
    if(obj == null || getClass() != obj.getClass())
      return false;
    AutoPart part = (AutoPart)obj;
    return partID == part.partID;
  }
  . . .
  // unique part number
 private int partID;
}
```

TreeSet

- TreeSet also implements the Set interface, and is collection of objects that may be accessed by content
- TreeSet is a *sorted* collection — unlike HashSet:
  - Elements may be inserted in any order;
  - Iterating through the collection returns them in sorted order

```
TreeSet sortedWords = new TreeSet();
sortedWords.add("amazing");
sortedWords.add("allowable");
sortedWords.add("antimatter");
sortedWords.add("abacus");
sortedWords.add("aardvark");
for(Iterator iter = sortedWords.iterator(); iter.hasNext();)
  System.out.println(iter.next());
```

TreeSet Implementation

- The sorting is accomplished by storing things in a tree structure (typically a *red-black tree*)
- Every time an element is added to the tree it is placed in position to that the collection remains sorted
- Adding is slower than adding to a hash table, but much faster than adding to the right place in an array or linked list
- For a TreeSet of size n, add (or contains) each require about $\log_2(n)$ comparisons

```
Collection inputText;
. . .
Set wordSet = new HashSet();
String wd;
for(Iterator i = inputText.iterator; i.hasNext();){
  wd = (String) i.next();
  // no need for wordSet.contains(wd)
  wordSet.add(wd);
}
```

Timing Comparisons

The input text was the complete text of an issue of *The Guardian* newspaper, from April 1995: 119 604 words; 23 727 unique words.

Collection	Running Time/s
ArrayList	87.4
Vector	92.8
LinkedList	121.5
HashSet	0.2
TreeSet	0.4

The Comparable Interface

- Objects to be stored in TreeSet must implement the Comparable interface
- The Comparable interface defines a single method:
  int compareTo(Object obj)
- a.compareTo(b) returns:
  0 if a and b are equal;
  Negative integer if a comes before b;
  Positive integer if a comes after b.
- String implements Comparable. String.compareTo returns the strings in lexicographic (alphabetic) order
- To insert your own objects you must implement Comparable

AutoPart.compareTo and Person.compareTo

```
public class AutoPart implements Comparable {
  . . .
  public int compareTo(Object obj) {
    return partID − ((AutoPart)obj).partID;
  }
  . . .
}
```

```
public class Person implements Comparable {
  . . .
  public int compareTo(Object obj){
    Person per = (Person)obj;
    if (surname.equals(per.surname))
      return firstName.compareTo(per.firstName);
    return surname.compareTo(per.surname);
  }
. . .
}
```

Summary

- If you need to access elements of a collection by content, then HashSet and TreeSet are appropriate
- HashSet computes an index for an element based on its hash code
- TreeSet stores elements in sorted order in a tree
- Elements stored in a TreeSet should implement the Comparable interface
- Elements stored in a HashSet should define hashCode and equals appropriately
- All this is put together in the complete implementations of Person and AutoPart (on the web pages)

Next Lecture: Maps and More

Exercises

- Modify the ContactDetails class, so that ContactDetails objects may be stored in a HashSet or a TreeSet
- Write a test program that creates several ContactDetails objects and stores them in a HashSet
- Now store the ContactDetails objects in a tree set and prints them out in sorted order.