CMPT 126: Lecture 11
Collections: Abstract Data Types and Dynamic Representation (Chapt. 12)

Tamara Smyth, tamaras@cs.sfu.ca
School of Computing Science,
Simon Fraser University

October 30, 2007

Collections and Data Structures

- A data structure is any programming construct, either defined in the language or by a programmer, used to organize data into a format to facilitate access and processing.
  - Eg.: arrays, linked lists, and stacks can all be considered data structures.
- A *collection* is an object that serves as a repository for other objects. They may be implemented in a variety of ways (which is why they are often referred to as abstract data types (ADTs), determined by the data structure.
- The role of a collection is generally to provide the following services:
  - add
  - remove,
  - manage the contained elements

array vs `ArrayList`

- We already saw the array, which is limited in that:
  - once created, its size cannot be changed, making it difficult to add or remove new elements.
  - it has no build-in utility methods (though the `Array` class in Java does provide a few)
  - they are not very useful for more complex data structures, such as sets or trees (to be discussed).
- The Java `ArrayList` is implemented using an array, but is dynamic, that is, it can shrink and grow, and includes helpful support methods.
- Drawbacks of `ArrayList`
  - Slightly less efficient than arrays because of the extra work required to keep track of their size and to shrink and grow.
  - They use more memory than arrays. They reserve extra memory, at worst twice as much as an array storing the same number of elements, so adding can be done quickly.
  - "Unusual" syntax.

Using a collection or ADT

- If you want to read every line of a text file into memory, which data structure would you use?
- Problem: Write a complete method that takes a sorted array of ints and returns an array of ints that is identical to the input, except that it does not contain any duplicates. The size of the array should be equal to the number of int it contains.

$$2, 2, 4, 5, 6, 7, 8, 8 \longrightarrow 2, 4, 5, 6, 7, 8$$

Solution using an array

```
public int[] removeDuplicates(int[] ar)
{
   int[] tmp = new int[ar.length];
   tmp[0] = ar[0];
   int next = 1;
   for (int i=1; i<ar.length; i++) {
      if (ar[i] != ar[i-1]) {
         tmp[next] = ar[i];
         next++;
      }
   }
   //Create a new array of the correct size
   int[] result = new int[next];
   for (int j=0; j<result.length; j++)
      result[j] = tmp[j]; //copy

   return result;
}
```

- Required the creating of 2 new arrays: `tmp` and `result`.

Solution Using `ArrayList`

```
static ArrayList<Integer> removeDuplicates2(int[] ar)
{
   ArrayList<Integer> result = new ArrayList<Integer>();
   result.set(0, ar[0]);
   for (int i=1; i<ar.length; i++) {
      if (ar[i] != ar[i-1]) {
         result.add(ar[i]);
      }
   }
   return result;
}
```

- and using `HashSet`...

```
static HashSet<Integer> removeDuplicates3(int[] ar)
{
   HashSet<Integer> result = new HashSet<Integer>();
   for (int i=0; i<ar.length; i++)
      result.add(ar[i]);

   return result;
}
```


Sets

- A set is a collection of items which contains no duplicates, that is, no two different items are equal.
- Not used, perhaps, as frequently as arrays, but still very useful.
- Example: Given a jpeg image, determine exactly how many different colours of pixels there are.
  - Solution 1: Read colours into an `ArrayList` as `Color` objects. This would work, but then you would need to write an algorithm to remove duplicates and this could get messy.
  - Solution 2: Read all the pixel colours into a set. A set won't let you add any object that is already present in the set. The number of distinct pixel colours is simply the number of elements in the set.

    ```
    initialize 'set' to be empty
    for each pixel (x,y) in the image:
       get the color C of (x,y)
       add C to the set
    ```


Java sets

- Java has two sets:
  - **HashSet** is faster, but does not actually store elements in any particular order.
  - **TreeSet** is slower, but they are ordered (which may make it faster depending on what it's used for). Elements are ordered, but this means they must be *comparable*, i.e., for any two objects, Java must be able to determine if $x \leq y$.
- If you can, use a `HashSet`.

ColorCounter.java

```
/*
 * Uses HashSet to count the number of distinct pixel colors
 * in an image.
*/
import java.awt.Color;
import java.util.HashSet;
import csimage.*;

public class ColorCount {
   public static void main(String[] args) {
      String fname = EasyInput.chooseFile();
      UberImage m = UberImage.fromFile(fname);
      show.inFrame(m);

      HashSet<Color> set = new HashSet<Color>();
      for (int x = 0; x < m.getWidth(); x++) {
         for (int y = 0; y < m.getHeight(); y++) {
            Color c = m.getColor(x, y);
            set.add(c);
         }
      }
      int numPixels = m.getHeight() * m.getWidth();
      int numColors = set.size();
      double pct = 100 * (numColors / (double) numPixels);
      System.out.printf(fname + ``: '' + pct +
                  ``\% of pixels are uniquely coloured.'');
   }
}
```


Maps

- A more advanced data structure.
- Also called a dictionary or an associated array, it is essentially a set of (key, value) pairs.
- Both the key and value are objects.
- A map may never have two pairs with the same key, i.e., **keys must be unique**.
- Example: Create a glossary (like the one in the back of your textbook), that is, a special-purpose dictionary that lists words and their definitions.

animal:	Food.
vegetable:	What food eats.
fruit:	What food eats.
apple:	A red fruit.

- A map is the natural data structure for storing a glossary: the words are they keys, and the definitions are the values.

Java maps

- Java has two maps:
  - **HashMap** is more efficient, but does not guarantee any particular order for storing its pairs.
  - **TreeMap** is a little slower (but not much), but keeps things in order.
- `HashMaps` are more frequently used.
- When you declare a map, you must include the types of the key and the values:

```
HashMap<Integer, String> hmap = new HashMap<Integer, String>();
hmap.put(1, "one");
hmap.put(2, "two");
hmap.put(3, "three");
hmap.put(4, "four");
hmap.put(5, "five");

for (int i=1; i<=5; i++)
   System.out.println(hmap.get(i));
```

- This creates a new map containers where the keys are Integers and the values are Strings.

MapTest.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;

public class MapTest {

   public static void main(String[] args) {
      HashMap<String, String> map = new HashMap<String, String>();
      map.put("animal", "Food.");
      map.put("vegetable", "What food eats.");
      map.put("fruit", "What food eats.");
      map.put("apple", "A red fruit.");

      System.out.println("" + map);
      ArrayList<String> terms = new ArrayList<String>(map.keySet());
      Collections.sort(terms); //sorts in ascending order
      for(String word: terms) {
         String def = map.get(word);
         System.out.printf("%s - %s\n", word, def);
      }
}
```

Dynamic Representation

- An array and an arraylist is limited because it has a fixed size during its existence. Even the `ArrayList` has overhead because it has to create new arrays when the size of the collection changes.
- Another way to store a collection is by using a *linked list*, where each element is "linked" to the next element in the list.
- The objects in this collection must therefore store both data and a link which is a reference to the next element. Their class may look something like this

```
class Node
{
   int info;
   Node next;
}
```

- The list may then be managed by storing and updating references to objects of type `Node`.

Linked Lists

- What are the limitations of a linked list?
- Each time you want to add a node, you must traverse to the end.
- You could have a `ListHeader` class that contains a reference to both the front and the end of the list.

```
class ListHeader
{
  int count;
  Node front, rear;
}
```

- In this case, the header node is not the same as the element nodes.
- You may also want to hold a counter to the number of elements in the list.

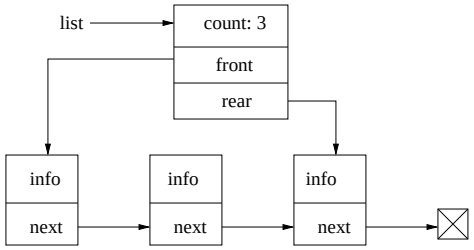

Figure 1: A list with references both at the front and the end.

Doubly Linked Lists

- It may be necessary to have an Abstract Data Type that allows you to go back and forth between nodes, rather than returning to the beginning of the list to access the previous element.
- How would you modify your `Node` class?

```
public class Node
{
   int info;
   Node next, prev;
}
```


Queues

- Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.
- A queue is similar to a list, except that it has restrictions on the way you add and remove items.
- A queue uses *first-in, first-out* (FIFO) processing.
- Think of a waiting line: the first person in line is the first one into the show.
- A queue has the following operations:
  - **enqueue**: adds an item to the rear of the queue
  - **dequeue**: removes an item from the front of the queue
  - **empty**: returns true is the queue is empty

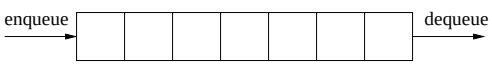

Figure 2: A queue data structure.

Stacks

- A *stack* is similar to a queue except that its elements go on and come off at the same end. Therefore, the last item on is the first item off.
- A stack uses *last-in, last-out* (LIFO) processing.

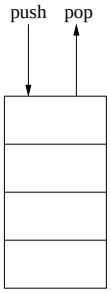

Figure 3: A stack data structure.

- Analogies: a stack of dishes: the last one on the stack is the first one off. A crowded elevator, where begin the first one on likely means you will be the last one off!

Stacks in Java

- A stack ADT contains the following operations:
  - push: pushed an item onto the top of the stack
  - pop: removes an item onto the top of the stack
  - peek: retrieves information from the top item of the stack without removing it.
  - empty: returns true if the stack is empty.
- Java has a class called `Stack` which implements that stack data structure.

```
import java.util.Stack;

public class TestStack
{
    public static void main(String[] args)
    {
        Stack wordStack = new Stack();

        wordStack.push("apple");
        wordStack.push("orange");
        wordStack.push("banana");

        while (!wordStack.empty())
           System.out.println((String)wordStack.pop());
    }
}
```


Trees

- A tree is a non-linear data structure that organizes data into a hierarchy.
- It consists of a *root node* and possible many additional levels of nodes with *internal nodes*.
- Nodes without children are called leaf nodes.

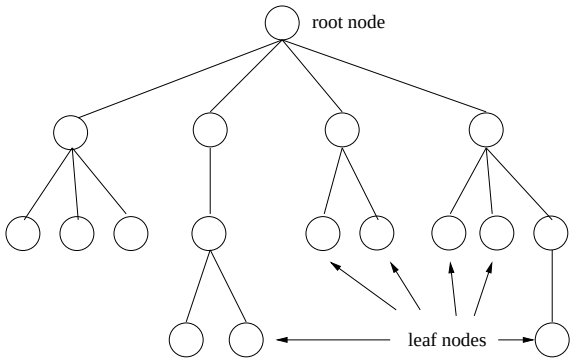

Figure 4: A tree data structure.

Binary Tree

- A binary tree is a tree data structure in which each node has at most two children.
- Typically, child nodes are called left and right.

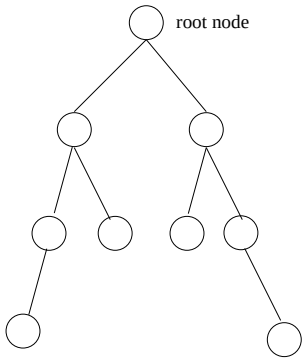

Figure 5: A Binary Tree.

Graphs

- A graph is also a non linear data structure. Unlike a tree, it does not have a primary entry point like the *root node*.
- A graph is linked to another node using an *edge*. Generally the number of edges is unrestricted.
- Edges are usually bi-directional, meaning they can be followed to or from the node to which it is connected.

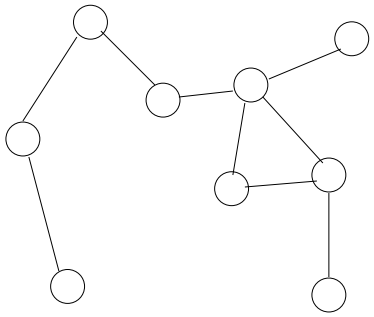

Figure 6: A graph data structure.

- In a *digraph*, or *directed graph*, the edge has a single direction.
- Graphs are useful for representing non linear, non-hierarchical relationships, such as the highway system connecting cities on a map, or airline connection between airports.
- They may be implemented using both arrays and dynamic links.

The Java Collections API

- The *Java Collections API* contains several classes that represent collections of various types.
- Class names usually indicate the type of collection (e.g., `ArrayList`, `LinkedList`).
- Several interfaces are used to define the collection operations:
  - List
  - Set
  - SortedSet
  - Map
  - SortedMap
- Classes are usually implemented as *generic types*, meaning the type of object in the collection can be established upon instantiation.

```
LinkedList<String> myStringList = new LinkedList<String>();
```

- This ensures compatibility among the objects of the collection.