CMPSCI 187: Programming With Data Structures

Lecture 4: Data Structures Overview
12 September 2012

Data Structures Overview

- Containers and Collections
- Ways to Arrange a Collection
- The Stack: Data and Operations
- Separating Specification From Implementation
- Inheritance: Distinguishable Objects

The Container Metaphor

- Imagine a container terminal in a seaport. (There is a nice picture on the cover of last year’s text by Lewis and Chase.)
- Containers come off of ships, for example, and are put on trucks or trains.
- The containers are identical on the outside, different on the inside, but identifiable by labels.
- Between the unloading from ships and loading onto something else we have a **collection** of containers.
- This is a good metaphor for a collection of pieces of data, such as an array. We often want to move the objects around without looking at their contents.

Collections of Data

- A Collection is a set of objects from the same class.
- We can create Collections, insert elements, and remove elements.
- We might want to remove an element that is “best” in some way, such as the container that needs to be shipped out next.
- How the Collection is arranged internally might affect how quickly or easily we can find and remove that element.
- Different kinds of Collections have different sets of operations.
- We will use **generics** (Java 5.0) so that from type `T` we get type `Collection<T>`, and so forth.

How might we arrange our containers?

- If they come off the ship in the right order, we could put them in a line where we add elements to one end and take them off the other -- a **queue**.
- If they come off the ship in reverse order, we could put them on a train siding where we can take out the last one we put in -- a **stack**, as in the discussion.
- If they might come off in any order, what we do depends on the memory we have.
- Can we keep them in a sorted array? What happens if we get a new one that belongs between two that are already adjacent?
- If we have multiple "buffer areas", managing each one might be easier as they would on average have fewer elements in them.

A More Formal Definition of a Stack

- A `Stack<T>` object is a set of `T` objects.
- We can create an empty `Stack<T>`, with no elements.
- With an existing `Stack<T>`, we can **push** a new `T` element, inserting it.
- If a `Stack<T>` has one or more elements, we can **pop** the last element that was pushed, removing it.
- If a `Stack<T>` has one or more elements, we can **peek** at the last element pushed, looking at it without removing it.
- We can find out how many elements are in a `Stack<T>`, in particular, we can test whether there are any at all.

Separating Specification From Implementation

- Each of our standard data structures is specified by an **abstract data type (ADT)**, a list of the permissible operations as we've given for the Stack.

- We should be able to write code for a Stack that works however it is implemented. But a Java class fixes a particular implementation, with its fields and code for the methods.

- So "Stack" should not be a class! In Java, it ought to be an **interface**, which is a list of methods that specifies ways we can use a class. A class **implements** the interface if all those ways to use the class are enabled.

- Actually in the Collections package in Java there *is* a class called Stack, in fact a generic definition that gives a class `Stack<T>` for every class `T`. DJW have an interface called `StackInterface<T>`.

Implementations of a Data Structure

- Stacks, queues, lists, and graphs are **conceptual** ways to group multiple objects into a single object. But in our actual code we will need to group pieces of data (objects) into larger objects, using the rules of Java. There are two basic ways to do this.

- We know about **arrays**, single objects that **contain** an linear sequence of other objects of a particular type. We can easily retrieve a particular element `x[i]` of an array `x`.

- The other basic way to group objects is a **linked structure** using **pointers**. Our linked structure "contains" the first object, which "contains" the second, which contains the third, and so on until one object "contains" the virtual object `null` and the sequence ends. Since Java objects are always **called by reference**, our pointers are **implicit** -- we point to an object by saying that we contain the object itself. This can be confusing if you are used to a language like C or C++ with **explicit pointers**.

Different Implementations of the Same Structure

- The first data structure that DJW studies in detail is a StringLog, a collection of `String` objects. We can insert strings into a StringLog, test whether a particular string is a member of it, find out how many strings it contains, and get a single string representing the whole StringLog.

- One way to implement StringLog is with an array of strings, an object of type `String [ ]`. Of course arrays have a fixed length, where the length of a StringLog varies with time.

- We could also make a linked structure where each **node** of the structure contains a `String` object and a pointer to the next node.

- If we write code that only uses the given operations of a StringLog, it should work equally well whichever of the two implementations is used.

Inheritance: Distinguishable Objects

- The container metaphor is a good way to think about **inheritance**.

- Some containers may need refrigeration, some may have hazardous chemicals, some may be empty, etc., but all are still containers and can be offloaded, moved around in the port, and sent out like any other.

- If a Container is actually a RefrigContainer, there may be other operations that we can perform on it. So the class RefrigContainer will **extend** Container, and be a **subclass** of it. A RefrigContainer object can have either Container or RefrigContainer instance methods run from it.

- We can't say too many times that **is-a** and **has-a** relationships are different. In Project 1, a StringLog has-a String, but in the Lecture 3 example a Terrier is-a Dog.