ABSTRACTION and ADTs

Capturing General Concepts

Problem: Given a map, and a set of cities to visit, find the shortest route that will take you to every city once and return you back home at the end.

Problem: Given a board layout in checkers, find the best next move. (Assume we have defined some measure of goodness for moves.)

What do these problems have in common?

Over many years, computer scientists have found that certain very general structures occur repeatedly in all sorts of problem domains. E.g.:

- a "graph" (like a map)
- a stack (like a pile)
- a queue (like a lineup)
- a computer file
- a computer window

Such a structure is called an Abstract Data Type (ADT).

You've already seen two ADTs: stack and queue.

Advantage of keeping an ADT general

If we define the ADT in a *general* way, then:

- whatever algorithms we derive for it (*e.g.*, for finding the shortest path between two nodes in a graph), and
- whatever we prove about it (*e.g.*, you can tour a graph, visiting each edge exactly once and returning to the starting node, iff every node has an even number of neighbours)

can be applied to many problem domains. This is tremendously valuable!

Example: Shortest path.

- If the graph represents a map, the shortest path between two nodes tells you what route to take from one place to the other.
- If the graph represents a computer network, the shortest path between two nodes tells you how to direct an email message from one machine to another.

What's "abstract" about ADTs?

From the Oxford and Webster's dictionaries:

"Abstract" comes from the Latin "abstractus", meaning to draw away.

Abstraction is

> the act or process of separating the inherent qualities or properties of something from the actual physical object or concept to which they belong.

E.g., the gas gauge on your car is an abstraction. It displays only information that is necessary to the driver (how full the tank is) and abstracts away from everything else about the tank (shape, colour, type of metal, etc.).

An ADT is an abstraction in this sense. It defines what is essential and ignores everything else.

Defining an ADT

To define an ADT, we must specify 2 things:

- The $\boxed{\text{data}}$ (and any particular properties it must have).
- The $\boxed{\text{operations}}$ one can perform on it.

The definition should be complete and unambiguous, so that someone can implement it using only the definition.

This is all about "what." For example:

- what a queue is, and
- what you can do to it.

It is not about "how." For example:

- how the queue will be used,
- how to store the queue, or
- how to perform the operations.

Another ADT: Text File

Data: A sequence of characters. A file has one of two states: open or closed. If it is open, there is a current position within the file.

Operations:

open(): Prepare the file and set the current position to the beginning.

close(): Finish using the file.

write(c): Write character c at the current position in the file and advance the current position. The file must already be open.

read(): Read and return the character at the current position in the file and advance the current position. The file must already be open.

go(p): Go to position p within the file and make that the current position.

What's abstract about the Text File ADT?

It seems to say everything about how files work. But here are some things that it doesn't say:

You may have used files in some programming language and not even realized that any of this goes on behind the scenes.

That's the whole point of abstraction: You don't *have to* know! Imagine how much harder it would be to use a file if you did have to.

Implementing an ADT

To use an ADT in a program, we must "implement" it, that is, write code for 2 things:

- Variables that can represent the data using some data structure.
- Methods that can perform the operations.

There are *many* different ways to implement any ADT.

For example, the queue ADT can be implemented using an array, a circular array, a vector, and many other structures.

Each implementation has pros and cons. We must be aware of these so that we can recognize tradeoffs and make informed choices.

About our queue implementation

We used a Java `interface` to separate how to use a queue from how it works.

We made the data members in class `ArrayQueue` that store the queue itself `private`. This is called **information-hiding** or **encapsulation**.

Important practical advantages:

- The data members are protected from harm. The only way we can change them is through public methods, which guarantee to keep them consistent.
- We are free to change the implementation (e.g., to improve efficiency) without affecting any code that uses it.
- To use a class, we need to know only its interface.

- We can write client code even before the class is implemented.
- Client code that uses the interface works with any class that implements it.

Another ADT: graph

Data:

- A set of nodes (or "vertices"), and
- A set of undirected edges, each defined by the pair of nodes that it connects.

Operations:

addNode(n): Add node n to the graph. Return false iff n is already in the graph.

removeNode(n): Remove node n and its edges from the graph. Return false iff n is not in the graph.

addEdge(n1, n2): Add to the graph an edge between nodes n1 and n2. Return false iff n is already in the graph.

removeEdge(n1, n2): Remove from the graph the edge between nodes n1 and n2. Return false iff that edge is not in the graph.

nodes(): Return a list of all the nodes in the graph.

edges(): Return a list of all the edges in the graph.

A Good ADT Definition

You can define your own ADTs.

Starting from the concept (*e.g.*, a stack is like a pile), there are many ways one could define the ADT formally.

A good ADT definition is complete

The operations should let you do anything you need. Generally there should be operations to insert, remove, and query the ADT.

But is also minimal

It should have a *minimal* set of operations that is complete. If you want to do something fancy, you may have to write your own method (outside the class) that uses a series of the basic operations.

Examples:

- To clear a stack, you can repeatedly pop the stack until it is empty.
- To find all the neighbours of a given node in a graph, you can get the list of all edges and look through it yourself.

When you implement an ADT, you may add extra operations to the class for the convenience of the client code or for efficiency. For example:

- A stack clearing operation would be a convenience.
- An operation to return the neighbours of a given node may be much more efficient if performed *within* the graph class. Why?

But these do not belong in the ADT or its `interface`.

Implementing the Graph ADT

Question: Describe some alternative data structures for implementing the graph ADT.

Abstract Data Types vs Data Structures

An **ADT** describes *what* you want to do with your data.

A **data structure** is part of the implementation, which describes *how* you do it.

An ADT can be viewed as a specification that the chosen implementation must meet.

Classes and interfaces support the distinction between an implementation and an ADT.