CMPS 101
Algorithms and Abstract Data Types

ADTs and Modules in Java and C

Introduction

This document introduces the concepts of Modules and ADTs, and describes how to implement them in both Java and C. Informally, an *Abstract Data Type* (ADT) is a collection of mathematical objects, together with some associated operations on those objects. When an ADT is used in a program, it is usually implemented in its own *module*. A module is a self-contained component of a program having a well-defined *interface* that details its role and relationship to the rest of the program.

Why are ADTs necessary? The standard data types provided by most programming languages are not powerful enough to capture the way we think about the higher level objects in our programs. This is why most languages have a type declaration mechanism that allows the user to create high level types as desired. Often the implementation of these high level types gets spread throughout the program, creating complexity and confusion. Errors may occur when the legal operations on the high level types are not well defined or are not consistently used.

The term Abstract Data Type can mean different things to different people, but for the purposes of this course, an ADT consists of two things:

> (1) A set *S* of "mathematical structures", the elements of which are called *states*.
> (2) An associated set of operations which can be applied to the states in *S*.

Each ADT *instance* or *object* has a current state which is one of the elements of the set *S*. The operations on *S* fall (roughly) into two categories. *Manipulation procedures* are operations which cause an ADT object to change its state. *Access functions* are operations which return information about an ADT object, without altering its state. We will from time to time consider operations which fall into both categories, or to neither. An ADT is an abstract mathematical entity which exists apart from any program or computing device. On the other hand, ADTs are frequently implemented by a program module. We will distinguish between the mathematical ADT and its implementation in a programming language. A single ADT could have many different implementations, all having various advantages and disadvantages.

Example Consider an *integer queue*. In this case *S* is the set of all finite sequences of integers, and the associated operations are: Enqueue, Dequeue, getFront, getLength, and isEmpty. The meanings of these operations are given below. One possible state for this ADT is (5, 1, -7, 2, -3, 4, 2). (It is recommended that the reader who is unfamiliar with elementary data structures such as queues, stacks, and lists, review sections 10.1 and 10.2 of the text.)

> <u>Manipulation procedures</u>
>   Enqueue   Insert a new integer at the back of the queue
>   Dequeue   Remove an integer from the front of the queue
> <u>Access functions</u>
>   getFront   Return the integer at the front of the queue
>   getLength   Return the number of integers in the queue
>   isEmpty   Return true if length is zero, false otherwise

Other examples of mathematical structures which could form the basis for an ADT are: sets, graphs, trees, matrices, polynomials, or finite sequences of such structures. In principle, the underlying set S could be anything, but typically it is a set of discrete mathematical objects of some kind.
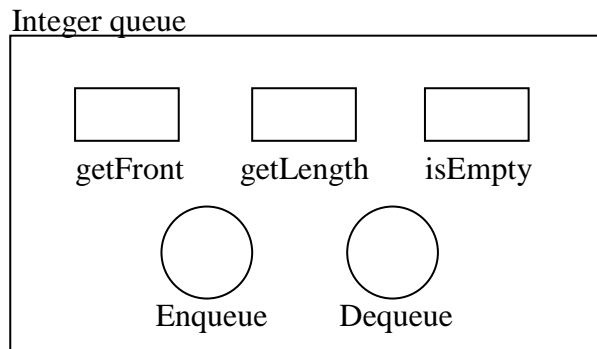
An ADT instance is always associated with a particular sequence, or history, of states, brought about by the application of ADT operations. In our queue example we could have the following sequence starting with the empty state $\varnothing$:

Operation	State
	$\varnothing$
Enqueue(5)	(5)
Enqueue(1)	(5, 1)
Enqueue(7)	(5, 1, 7)
Dequeue()	(1, 7)
Enqueue(3)	(1, 7, 3)
getLength()	(1, 7, 3)

Observe that if isEmpty is true for some state, then getFront and Dequeue are undefined on that state. One option to deal with this situation would be to make special definitions for Dequeue and getFront on an empty queue. We could for instance define getFront to return zero on an empty queue, and define Dequeue to not change its state. Unfortunately, these special cases complicate the ADT and can easily lead to errors. A better solution is to establish *preconditions* for each operation indicating exactly when (i.e. to which states) that operation can be applied. Thus a precondition for both getFront and Dequeue is: 'not isEmpty'. In order for an ADT to be useful, the user must be able to determine if the preconditions for each operation are satisfied. Good ADTs clearly indicate the preconditions for each operation, usually as a sequence of access function calls. Good ADTs may also document their *postconditions*, i.e. conditions which *will be* true after an operation is performed. For example, a postcondition of Enqueue is 'not isEmpty'. ADT operations are analogous to mathematical functions. Preconditions and postconditions in this analogy then define the function's domain and codomain. Only when all operations have been defined, along with all relevant preconditions and postconditions, can we say that an ADT has been fully specified.

We often consider multiple instances of the same ADT. For example, we may speak of several simultaneous integer queue objects. ADT operations should therefore specify which object is being operated on. It is also possible for some operations to refer to multiple objects. We could for instance have an access function called Equals which operates on two queues and returns true if they contain the same elements in the same order and false otherwise, or a manipulation procedure called Concatenate which empties one queue and places its contents at the end of the another queue.

It is sometimes helpful to think of an ADT object as being a 'black box' equipped with a control panel containing buttons that can be pushed (manipulation procedures), and indicator lights to be read (access functions).

Integer queue

```
┌─────────────────────────────────────────────┐
│  ┌────────┐    ┌────────┐    ┌────────┐       │
│  │        │    │        │    │        │       │
│  └────────┘    └────────┘    └────────┘       │
│   getFront     getLength      isEmpty         │
│      ○              ○                          │
│   Enqueue       Dequeue                        │
└─────────────────────────────────────────────┘
```

Note that some texts (including our own) define Dequeue so as to return the front element, as well as to alter the state of the queue. In our example, the Dequeue operation deletes the element at the front of the queue, but doesn't return a value, making it a pure manipulation procedure rather than an operation of mixed type. Such a change in the set of ADT operations results in a *different* ADT, although both have a right to be called a "queue". As a further example suppose we implement our queue by storing integers in an array of fixed size. We should then add an access function called isFull which reports whether there is room left in the array for another integer. Enqueue would then have the precondition 'not isFull'. This queue ADT and our original queue ADT can both be called queues, but they are different ADTs.
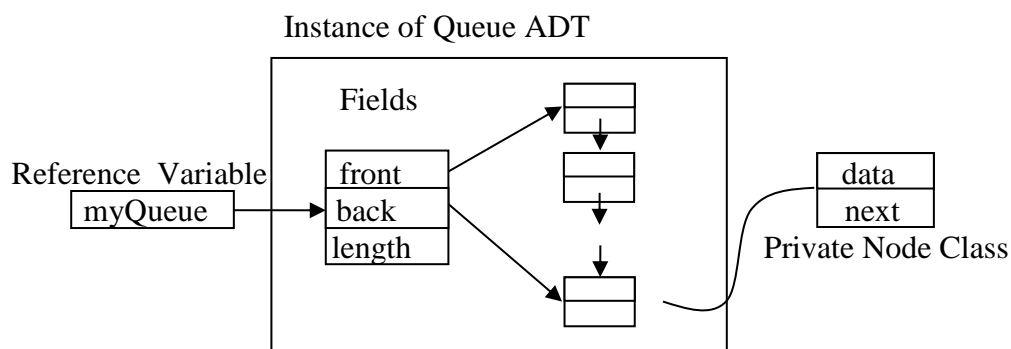
Implementing ADTs
There is a straightforward way of implementing an ADT in both Java and in C, once it has been specified. The implementation strategies in these two languages look different on the surface, but conceptually they are very similar, the differences being mostly syntactical.

Java Implementation
In Java an ADT is embodied in a *class*. A class contains *fields* (or *member variables*) which form the 'mathematical structure', and *methods* which implement the ADT operations. Such a class may also contain some (private) inner classes as part of the 'mathematical structure'. An instance of this class is accessed by a reference variable that represents an instance of the ADT.

Example The *inside* of our integer queue 'black box' can be pictured as

Instance of Queue ADT

```
                    ┌────────────────────────────────────┐      ┌────────┐
                    │  Fields                  ┌────┐     │      │  data  │
                    │                          └─┬──┘     │      ├────────┤
Reference  Variable │         ┌────────┐   ┌────▼┐        │      │  next  │
┌──────────┐        │         │ front  │   └─┬───┘        │      └────────┘
│ myQueue  │────────┼────────▶│ back   │     ▼            │   Private Node Class
└──────────┘        │         │ length │   ┌────┐         │
                    │         └────────┘   └────┘         │
                    └────────────────────────────────────┘
```

The user of the ADT should never be allowed to directly access the 'structure' inside the 'box'. Instead a reference variable (myQueue) points to an instance of the class and is passed as an (implicit) first argument to the methods of the class. For example, the call

3

would return the front element in the ADT object referenced by myQueue. For this reason (instance) variables should always be declared as private. This is the idea of *information hiding*: the user of the ADT cannot see or directly affect anything inside the 'black box' except through the official ADT operations. In the implementation depicted above, a queue consists of a singly linked list of private Node objects which cannot be directly accessed by the user of the Queue class. The purpose of this restriction is to free the user from the responsibility of knowing the internal details of a Queue object, which reduces the complexity of the user's task. To the user, a Queue is simply a sequence of integers which can be manipulated in certain ways.

Other methods are also needed but which do not correspond to access functions or manipulation procedures. Among these are the *constructors* which create new ADT instances, and the *toString* method which provides a string representation of the class. Our integer queue ADT in Java might look something like:

```java
// Queue.java
// An integer queue ADT

class Queue {
   private class Node {
      // Fields
      int data;
      Node next;
      // Constructor
      Node(int data) {...}
      // toString: overrides Object's toString method
      public String toString() {...}
   }

   // Fields
   private Node front;
   private Node back;
   private int length;

   // Constructors
   Queue() {...}

   // Access functions
   int getFront() {...}
   int getLength() {...}
   boolean isEmpty() {...}

   // Manipulation procedures
   void Enqueue(int data) {...}
   void Dequeue() {...}

   // other methods
   // toString:  Overrides Object's toString method.
   public String toString() {...}
}
```

An ADT implementation should be fully tested in isolation before it is used in a larger program. The following program serves this purpose.

```
// QueueTest.java

class QueueTest {
   public static void main(String[] args){
       // Allocate several Queue Objects, and manipulate
       // them in various ways.  Call each of the above
       // ADT operations at least once.
       ... ... ...
   }
}
```

Exercise Fill in the definitions of all of these methods, i.e. replace `{...}` where it appears by some appropriate Java source code. A solution to this exercise will be posted on the webpage.

By convention, all of our ADT implementations in Java should follow this same pattern: private inner classes, followed by fields, then constructors, access methods, manipulation procedures, then all other methods. This convention is by no means universal, but we adopt it in this course for the sake of uniformity.

Note that all ADT operations must state their own preconditions in a comment block, and then check that those preconditions are satisfied before proceeding. If a precondition is violated, the program should quit with a useful error message. This can be done efficiently in Java by throwing a RuntimeException. (It is not necessary and not recommended, that you write new Exception classes to be thrown when preconditions are violated.)

As mentioned previously, a *module* is a part of a program which is isolated from the rest of the program by a well-defined interface. We think of ADT modules as providing services (e.g. functions, data types, etc.) to *clients*. A client is anything (program, person, computer, another module) which uses an ADT module's services. These services are said to be *exported to* the client or *imported from* the ADT module.
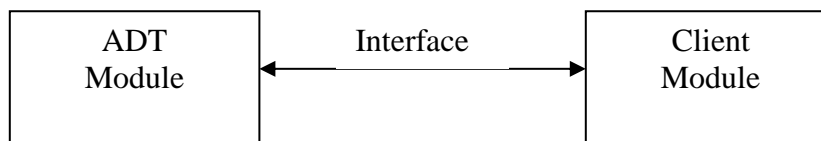
The module concept supports the idea of *information hiding*, i.e. clients have no access to a module's implementation details (inside the black box). The client can only access the services exported through the interface. Generally we will have a separate module for each ADT. As we've seen, an ADT module in Java is embodied in a single .java file containing a single top level class, and possibly some private inner classes. The module interface consists of all variables and methods in that class that are not declared private.

C Implementation

In C the situation is somewhat different since modularity is not directly supported by the language. An ADT implementation contains a struct which provides access to the 'mathematical structure' underlying the ADT. The user of the ADT (i.e. the client module) is given a *reference* which is a pointer to this struct. One C function is defined for each of the ADT operations. Each such function takes an ADT reference as argument, specifying which instance of the ADT to operate on. This reference is defined in a way that prevents the client from following the pointer to access the interior of the 'black box', thus enforcing the information hiding paradigm.
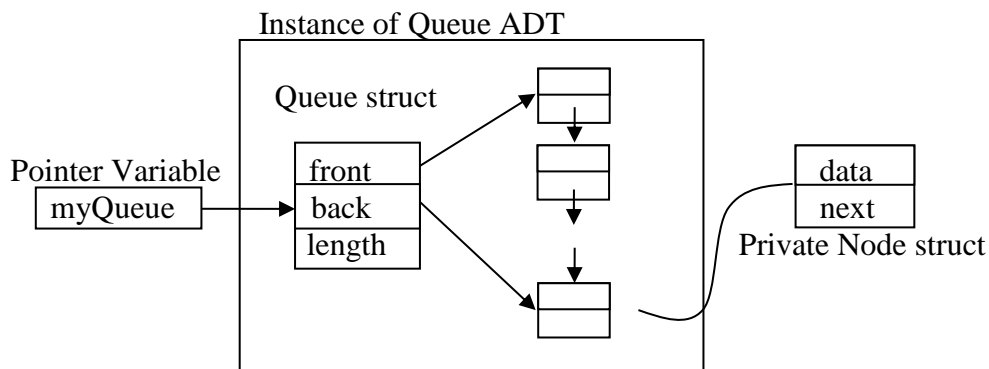
Two more C functions are necessary. One to create new objects (constructor) and one to free memory associated with ADT instances no longer in use (destructor). It is the responsibility of these functions to manage all of the memory inside the 'black box', balancing calls to malloc (or calloc) and free.

In C, we split our ADT module implementation into a .c file containing struct and function definitions, and a .h file containing typedefs and prototypes of exported functions. The ADT interface by definition consists of exactly that which appears in the .h file. Functions whose prototypes do not appear in this file cannot be accessed from outside the ADT implementation, and are therefore effectively private.

Example

```c
// Queue.h
typedef struct QueueObj* Queue;

// Constructor-Destructor
Queue newQueue(void);
void freeQueue(Queue* pQ);

// Access functions
int getFront(Queue Q);
int getLength(Queue Q);
int isEmpty(Queue Q);

// Manipulation procedures
void Enqueue(Queue Q, int data);
void Dequeue(Queue Q);

// Other functions
void printQueue(Queue Q, FILE* out);
```

In this example, Queue.h defines a pointer called Queue to a struct called Queue, which is not defined in this file. This is how data hiding is implemented in C. The client module will #include Queue.h so that the compiler recognizes calls to the exported functions. The client can also declare variables of type Queue and define functions which take Queue arguments. Notice however that the client cannot dereference through a Queue variable, since the Queue struct is not defined in the file Queue.h. That definition appears in the next file.

```c
// Queue.c
#include<stdio.h>
#include<stdlib.h>
#include "Queue.h"

// Private inner NodeObj struct, reference type Node, and constructor-
// destructor pair.  Not exported.
typedef struct NodeObj{
   int data;
   struct NodeObj* next;
} NodeObj;

typedef NodeObj* Node;

Node newNode(int data) {...}
void freeNode(Node* pN) {...}

// Private QueueObj struct, constructor-destructor
typedef struct QueueObj{
   Node front;
   Node back;
   int length;
} QueueObj;

Queue newQueue(void){
   Queue Q = malloc(sizeof(QueueObj));
   Q->front = Q->back = NULL;
   Q->length = 0;
   return(Q);
}

void freeQueue(Queue* pQ) {...}

// Access functions
int getFront(Queue Q) {...}
int getLength(Queue Q) {...}
int isEmpty(Queue Q) {...}

// Manipulation procedures
void Enqueue(Queue Q, int data) {...}
void Dequeue(Queue Q) {...}

// Other functions
void printQueue(Queue Q, FILE* out) {...}
```

Notice that the types `NodeObj` and `Node` as well as functions `newNode()` and `freeNode()`, do not appear in the file Queue.h, and are therefore not available to the client. Exporting these items would give the client access to the inside of the black box, violating the modularity principle. Notice also that another public function called `printQueue()` is included in both Queue.h and Queue.c. This function prints the state of a Queue object to a FILE handle (which may be `stdout`.) Function `printQueue()` corresponds roughly to the `toString()` function in Java.

As before, it is necessary to test the ADT implementation in isolation before it is used in a larger application.

```
// QueueTest.c
#include<stdio.h>
#include<stdlib.h>
#include "Queue.h"

int main(int argc, char* argv[]){
   // Call all of the above functions at least once
   return(0);
}
```

Exercise Complete the definitions of these functions by replacing { . . .} by appropriate C code. The solution to this exercise will be posted on the webpage. Also read the handout entitled "Additional Remarks on ADTs in C".

Some may (correctly) argue that our C Integer Queue is not really a general purpose queue, and that we should really write a queue of 'anythings'. The problem is that C's type mechanism is not advanced enough to properly deal with this issue. There are two possible solutions. The safer solution is to simply edit your Integer Queue to be a queue of whatever you need a queue of. Simply changing the appropriate `ints` to the new type will create a ready-made queue. This change can be easily accomplished by defining the type `QueueElement` in the .h file as

```
typedef int QueueElement
```

The type `QueueElement` is used to refer to the things that are stored in a Queue. This methodology lets you change the element type by editing a single line of code. (We follow this procedure in the exercise solution posted on the webpage.)

This simple fix has the drawback that if you want `int` queues and `double` queues in the same program, then you need two different Queue modules. A more powerful (and difficult) technique is to make `QueueElement` a generic pointer, by doing

```
typedef void* QueueElement
```

Now the Queue module can handle Queues which hold any kind of pointer. The danger is that a client might get confused and call `getFront()` or `Enqueue()` on the wrong kind of pointer. Using `void*` means that you will not find out about this problem until you run the program and get a segmentation fault. These types of pointer errors can be very difficult to debug. Given these warnings, I would recommend the safer solution for those students who do not have extensive C experience.

The equivalent queue of 'anything' in Java is accomplished by simply defining the `data` field in the private node class to be `Object` rather than `int`. This is essentially the same as using `void*` in C, but without the same danger of runtime errors. If such an error does occur, Java's exception handling mechanism should make it easier to track down. Better yet, starting with JDK 1.5, Java offers a new mechanism for abstracting over data types called *generics*, which is similar to the notion of a Template in C++. See http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf for a nice tutorial on the subject. Also see https://classes.soe.ucsc.edu/cmps012b/Spring15/lab6.pdf from for a project using generics.