CMPS 101
Algorithms and Abstract Data Types
Winter 2006

ADTs and Modules in Java and ANSI C

Introduction

This document introduces the concepts of Modules and ADTs, and describes how to implement them in both Java and ANSI C. Informally, an *Abstract Data Type* (ADT) is a collection of mathematical objects of some kind, together with some associated operations on those objects. When an ADT is used in a program, it is usually implemented in its own *module*. Each module should be self-contained and have a well defined *interface* detailing what the module does and how it can be used.

Why are ADTs necessary? The standard data types provided by many 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 the 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. 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* or *values*.
(2) An associated set of operations which can be applied to the states in *S*.

Each ADT *object* or *instance* has a state which is one of the mathematical structures in the set *S*. The operations on *S* fall (roughly) into two classes. *Manipulation procedures* are operations which cause an ADT instance to change it's state. *Access functions* are operations which return information about an ADT instance, without altering it's state. In this course we will maintain a clear distinction between the two types of operations. We will also from time to time consider operations which don't fall into either category, but we will not use operations which belong to both categories.

An ADT (or ADT instance) is an abstract mathematical entity which exists apart from any computing device. On the other hand, ADTs are frequently implemented by a program module. We will distinguish between the mathematical ADT and it's implementation in a programming language. In fact, a single ADT could have many different implementations, all with 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 meaning of these operations is 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.)

Manipulation procedures

Enqueue	Insert a new integer at the back of the queue
Dequeue	Remove an integer from the front of the queue

Access functions

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.

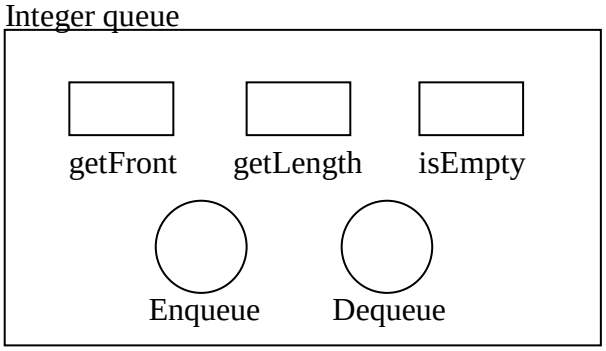
An ADT *object* or *instance* is always associated with a particular sequence, or history, of states, brought about by the application of ADT operations. In our queue example we could have the following sequence starting with the empty state $\varnothing$:

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 it's 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. Even those operations whose preconditions are always met should document that fact. Good ADTs 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 can sometimes be thought of as functions in the mathematical sense. Preconditions and postconditions 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 queues. 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 operation called Concatenate which operates on two queues by emptying one queue and placing its contents at the end of the other queue.

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



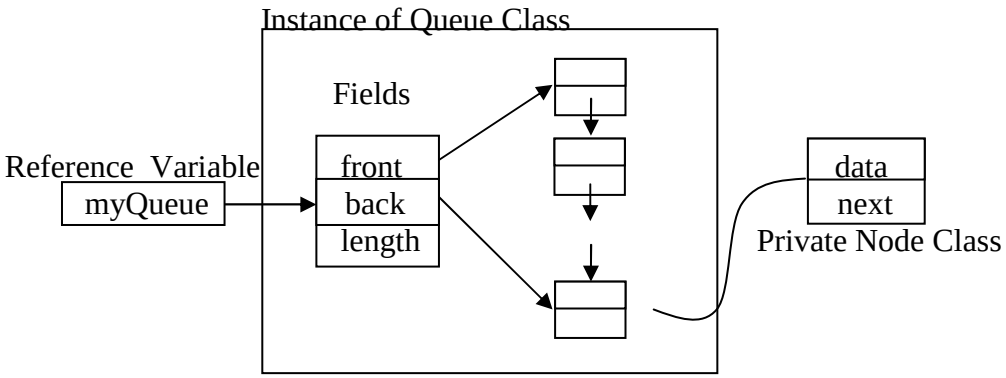
Note that in this example the Dequeue operation deletes the element at the front of the queue, but doesn't return a value. Some texts (including our own) define Dequeue so as to return the front element, as well as to alter the state of the queue. We adopt this particular definition in order to maintain the distinction between Access functions and Manipulation Procedures. Note that such a change in the set of ADT operations results in a *different* ADT. If for instance we implement our queue using an array of fixed size, then we should add an access function called isFull which reports on whether there is room left in the array for another element. Enqueue would then have the precondition 'not isFull'. Although this ADT and our original ADT can both be legitimately called queues, they are different ADTs.

Implementing ADTs

There is a straightforward way of implementing an ADT in both Java and ANSI C, once it has been specified. The implementation strategies in these two languages may look different on the surface, but conceptually they are quite similar, the differences being mostly syntactical.

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 which represents an instance of the ADT.

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



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

```
myQueue.getFront()
```

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 effect 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, the *toString* method which provides a string representation of the class, and the *main* method which we will use as a test driver for the ADT implementation. Our integer queue ADT in Java might look something like:

```
// 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() {...}

    // main
    // Used as a test driver for the Queue class.
    public static void main(String[] args) {...}
}
```

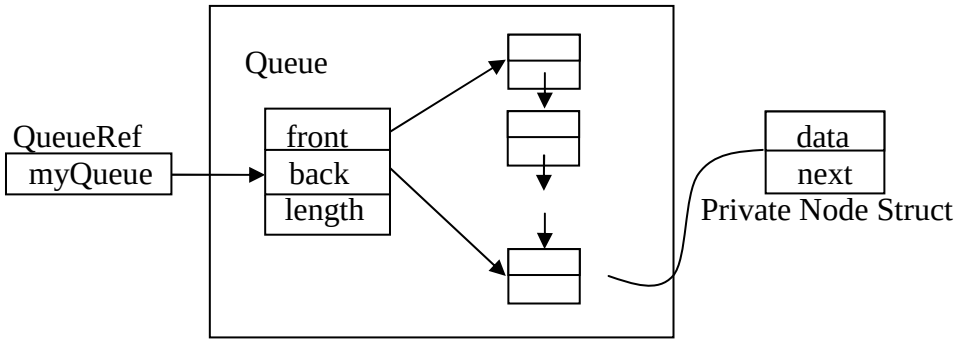
Exercise
Fill in the definitions of all of these methods, i.e. replace `{...}` where it appears by some appropriate Java source code.

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, with `main` as the final method. This convention is by no means universal, but we adopt it in this course for the sake of uniformity. Note that all ADT operations should 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.

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 modules as providing services (e.g. functions, data types, ..) to *Clients*. A client is anything (program, person, another module) which uses a module's services. These services are said to be *exported to* the client or *imported from* the 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 will be embodied in a single .java file containing a single top level class, and possibly some private inner classes. The module interface will consist of all variables and methods in that class which are not declared private.

In ANSI C the situation is somewhat different. Each ADT object consists of a struct which provides access to the 'mathematical structure'. The user of the ADT (i.e. client) 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. This reference is defined in such a way that the client cannot follow 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 objects 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 interface consists of exactly that which appears in the .h file. Functions whose prototypes do not appear in this file are not exported.

Example

```
/* File: Queue.h */
typedef struct Queue* QueueRef;

/* Constructor-destructor */
QueueRef newQueue(void);
void freeQueue(QueueRef* pQ);

/* Access functions */
int getFront(QueueRef Q);
int getLength(QueueRef Q);
int isEmpty(QueueRef Q);

/* Manipulation procedures */
void Enqueue(QueueRef Q, int data);
void Dequeue(QueueRef Q);
```

Notice Queue.h defines a pointer called QueueRef to a struct called Queue, which is not defined in this file. This is how data hiding is implemented in C. The client will #include Queue.h so the compiler recognizes calls to the exported functions. The client can also declare QueueRef objects and define functions which take QueueRef arguments and have QueueRef return values. Note however that the client cannot reference through the pointer QueueRef since it does not have the definition of the struct Queue. That definition appears in the next file.

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

```
typedef struct Node{ /* private inner struct, not exported */
   int data;
   struct Node* next;
} Node;

typedef Node* NodeRef;

typedef struct Queue{
   NodeRef front;
   NodeRef back;
   int length;
} Queue;

/* Constructor-Destructor */
QueueRef newQueue(void){
   QueueRef Q;
   Q = malloc(sizeof(Queue));
   Q->front = Q->back = NULL;
   Q->length = 0;
   return(Q);
}
void freeQueue(QueueRef* pQ) {...}

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

/* Manipulation procedures */
void Enqueue(QueueRef Q, int data) {...}
void Dequeue(QueueRef Q) {...}
```

Exercise Complete the definitions of these functions by replacing { . . .} by appropriate C code.

You should also write a constructor and destructor for the private Node struct to be used by Enqueue and Dequeue. These functions must not be exported since they manipulate Nodes directly, and therefore no prototype will appear in the .h file. (Exporting these functions would give the client access to the inside of the black box, which would violate the principle of modularity.)

```
NodeRef newNode(void) {...}
void freeNode(Node* N) {...}
```

Another function called printQueue which prints out the complete state of a Queue object should be included for diagnostic purposes. This function corresponds roughly to the toString() function in Java. It should be exported, and therefore its prototype should be added to the file Queue.h.

```
void printQueue(QueueRef Q) {...}
```

To complete the ADT module one should create yet another file called QueueClient.c, which acts as a test client for the Queue module. The purpose of this program is to thoroughly test the Queue module in isolation before using it in a real application. It should contain test calls to each of the functions

which implement ADT operations. We accomplished this in Java by including a main method in our Queue class definition. The complete code for both the Java and C implementations of Integer Queue will be placed on the class webpage. Follow the link to 'Examples'.

Some people may (correctly) argue that our ANSI 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
```

You can then use the type `QueueElement` whenever you want 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.

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 dangerous) 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, both methods are considered acceptable in this course, but 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.