CSE 101
Introduction to Data Structures and Algorithms
ADTs in C

Introduction

Informally, an *Abstract Data Type* (ADT) is a set of mathematical objects, together with a set of operations on those objects. When an ADT is used in a program, it is implemented in a *module*, which is a self-contained component of a program, usually a separate source code file. An ADT module should have a well-defined *interface* detailing its relationship to the rest of the program, and often this interface is also a separate file. In the C language, an ADT called `Blah` for instance, would consist of a source file called `Blah.c` and an interface file called `Blah.h`.

Why are ADTs necessary? The built-in 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. Therefore most languages have a type declaration mechanism that allows the user to create high level types as desired. If care is not taken, the implementation of these high level types may be spread throughout the program, creating complexity and confusion. Errors occur if the legal operations on the high level types are not sufficiently well defined, or are not consistently applied. The ADT concept was developed as a way to cope with these, and other problems related to program complexity. The youtube link below goes to an interesting talk by Barbara Liskov, who is the inventor of the ADT concept. In it, she discusses early events in the study of programming methodology, and the evolution of the notion of an ADT.

https://www.youtube.com/watch?v=qAKrMdUycb8&t=3714s

Definition

An Abstract Data Type consists of two things:

(1) A set *S* of "mathematical structures", the elements of which are called *states*.

(2) An associated set of *operations* that can be applied to the states in *S*.

Each ADT *instance* or *object* has a particular history of states, brought about by application of the various ADT operations. The operations *S* fall into (roughly) two categories: Manipulation Procedures and Access Functions. *Manipulation procedures* are operations that cause an ADT object to change its state. *Access functions* are operations that return information about the state without altering it. From time to time we will consider operations that belong to both categories, or to neither. An ADT is an abstract mathematical construct existing apart from any program or computing device. On the other hand, ADTs are frequently implemented in a program module. We will distinguish between the mathematical ADT, and its implementation in a programming language. A single ADT may have many different implementations, in the same language or in different languages, all with various advantages and disadvantages.

The Integer Queue ADT

In an Integer Queue, a state is a finite sequence of integers, and the set *S* consists of all such sequences. The associated operations are: Enqueue(), Dequeue(), getFront(), getLength(), and isEmpty(), whose meanings are given below. 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 Cormen, Leirserson, Rivest & Stein (CLRS), 4th edition.

Access functions	
getFront()	Return the integer at the front of the queue
getLength()	Return the length of the sequence
isEmpty()	Return true if the length is zero, and false otherwise

Manipulation procedures	
Enqueue()	Insert a new integer at the back of the queue
Dequeue()	Remove an integer from the front of the queue

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.

Below we trace a possible history of the Integer Queue, starting with the empty state ().

Operation	State	Return Value
	( )	
isEmpty()	( )	true
Enqueue(5)	(5)	----
Enqueue(1)	(5, 1)	----
Enqueue(-7)	(5, 1, -7)	----
getFront()	(5, 1, -7)	5
Dequeue()	(1, -7)	----
Enqueue(3)	(1, -7, 3)	----
getLength()	(1, -7, 3)	3
isEmpty()	(1, -7, 3)	false

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. For instance, we could define getFront() to return zero, and define Dequeue() to not change its state on an empty queue. Unfortunately, such special definitions like tend to complicate the ADT and can easily lead to errors. A better solution is to establish *preconditions* for each operation defining precisely to which states that operation can be applied. Thus, a precondition for both getFront() and Dequeue() is: "not isEmpty()". An ADT should clearly document all preconditions for each of its operations. The program module that uses an ADT, often called its *client module*, must be able to establish, for each operation, that its preconditions are satisfied. Therefore preconditions should be given in terms of calls to access functions. With this information, the programmer of the client module can logically exclude the possibility that a precondition will be violated.

ADTs should also document what data will be returned or how the state will be changed by each operation. This information is sometimes referred to as a *postcondition* of an operation. ADT operations are analogous in some sense to functions in mathematics. Preconditions and postconditions in this analogy define the function's domain and codomain, respectively. Only when all operations have been defined, along with all relevant preconditions and postconditions, can we say that an ADT has been fully specified.

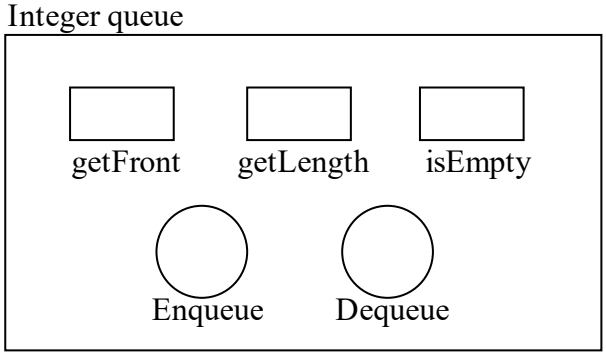
Since we often work with multiple instances of an ADT, operations should specify which object is being operated on. Also, some operations may refer to multiple objects, as for instance an Equals() operation that returns true if its two Queue arguments are in the same state, and false otherwise.

Some texts (including our own), define Dequeue() so as to return the front element, as well as to alter the state of the Queue, making it an operation of mixed type, both an access function and a manipulation procedure. In our example, we defined Dequeue() as a pure manipulation procedure. Such changes in the set of ADT operations result in a *different* ADT. As a further example, suppose we implement our Queue by storing integers in an array of fixed size. We might then add an access function called isFull() returning true if there is no room left in the array, and false otherwise. Enqueue() would then have the precondition "not isFull()". All of these variations can legitimately be called Queues, but they would be considered different as ADTs.

When implementing an ADT, it is usually necessary to write constructors and destructors to allocate and deallocate memory to be used by ADT objects. These operations are usually left out of the discussion when dealing with an ADT at a purely abstract level, since the details are heavily dependent on specific features of the implementation language.

The Black Box

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).

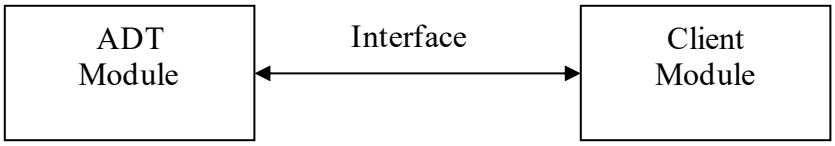


This metaphorical front panel of the black box is called the *interface* of the ADT. It is this interface that defines how an ADT will interact with other parts of a program.

Implementing ADTs in C

There is a straightforward way to implement an ADT in C, once it has been specified. Many modern programming languages (like C++, Java or Python) were in a sense, made for this purpose. The C language however pre-dates the ADT concept, so implementing an ADT requires some effort.

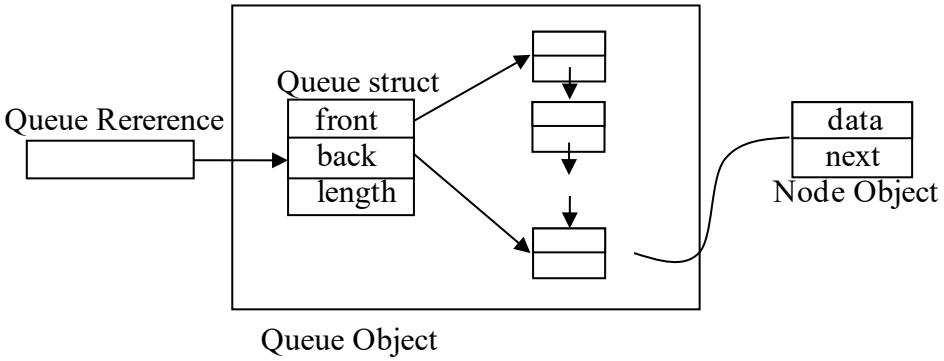
We think of ADT modules as providing services (like functions or data types) to *clients*. A client is another module that uses the ADT's services. These services are said to be *exported to* the client or *imported from* the ADT module.



The module concept includes the idea of *information hiding*: clients have no access to the internal state of a module, or any of its implementation details (i.e. what is inside the black box). The client can access a module's services only through its interface. The purpose of this principle is to reduce the complexity of the client's task by freeing it from the responsibility of knowing *how* an operation is performed. The client only knows that a given operation *will be* performed. To a client of the Queue ADT, a Queue object is just a sequence of integers that can be manipulated in certain ways. This modularity principle is actually a basic concept in all engineering disciplines. No comercially available automobile requires the driver to know how its engine works in order to drive it, and likewise for an ADT module.

An ADT implementation in C should contain a struct defining its "mathematical structure". The client is then given a pointer to this struct. We define one C function for each of the ADT operations, and each such function takes this pointer as an input argument. The pointer type 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 principle.

In our Queue example, we choose the underlying data structure to be a singly linked list. (Other choices are possible, such as an array.) Thus our C implementation will contain a private struct defining a Node object. The whole thing can be pictured in memory as follows.



Two more C functions are necessary. One to create new objects (constructor) and one to free heap 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(), calloc() and free().

The Queue ADT module is split into two files. A `.h` file containing typedefs and prototypes of exported functions, and a `.c` file containing struct and function definitions. The module interface consists of exactly that which appears in the `.h` file. Functions and types in the `.c` file whose prototypes do not appear in the `.h` file, cannot be accessed from outside the module, and are therefore effectively private.

```
// Queue.h
typedef int QueueElement;
typedef struct QueueObj* Queue;

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

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

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

// Other functions
void printQueue(Queue Q, FILE* out);
bool equals(Queue A, Queue B);
```

The file Queue.h defines a pointer type called Queue, to a struct called QueueObj, which is not defined in this file. The client module will `#include` Queue.h, so the compiler will recognize calls to the exported functions. The client can also declare variables of type Queue and define functions that take Queue arguments. Notice however that the client cannot dereference a Queue variable, since the struct to which it points is not defined in Queue.h. This is how data hiding is accomplished in C. The definition of QueueObj appears in Queue.c. Observe also that we define a typedef called QueueElement, so that it will be easier to change this Queue ADT into a queue of something other than an int.

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

// private Node type
typedef struct Node{
   QueueElement data;
   struct Node* next;
} Node;

// private QueueObj type
typedef struct QueueObj{
   Node* front;
   Node* back;
   int length;
} QueueObj;

// newNode()
// Returns a pointer to new Node object. Initializes next and data fields.
Node newNode(int data) {...}   // fill in

// newQueue()
// Returns reference to new empty Queue object.
Queue newQueue(){
   Queue Q;
   Q = malloc(sizeof(QueueObj));
   assert( Q!=NULL );
   Q->front = Q->back = NULL;
   Q->length = 0;
   return(Q);
}
```

```
// freeQueue()
// Frees all heap memory associated with Queue *pQ, and sets *pQ to NULL.
void freeQueue(Queue* pQ) {...}       // fill in

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

// Manipulation procedures
void Enqueue(Queue Q, QueueElement data) {...}  // fill in
void Dequeue(Queue Q) {...}   // fill in

// Other functions
void printQueue(Queue Q, FILE* out) {...}       // fill in
bool equals(Queue A, Queue B) {...}             // fill in
```

Notice that the type `Node` and function `newNode()`, 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 data hiding principle. Note also that public functions `printQueue()` and `equlals()` are included in both Queue.h and Queue.c.

We have left aside the question of what shall be done if the client calls an ADT operation on an argument that violates one of its preconditions. We adopt the following policy in all ADT implementations.

(1) All ADT operations must state their preconditions in a comment block appearing both before the function prototype in the `.h` file, and before the function definition in the `.c` file. These comment blocks should be identical.

(2) All ADT operations must check that those preconditions are satisfied before proceeding with nominal execution. Each precondition should be tested in a stand-alone `if` statement.

(3) If a precondition is violated, the operation will terminate with an error message giving: the name of the ADT module, the name of the ADT operation, and the particular precondition that was violated.

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

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

int main(int argc, char* argv[]){
   // Call each of the above functions at least once, exercising
   // every possible logical pathway through the function, including
   // error states.
   return(EXIT_SUCCESS);
}
```

Exercise Complete the function definitions above by replacing empty braces `{...}` with appropriate C code. A solution to this exercise is posted on the webpage. See the section "Naming Conventions for ADTs in C" below before you do this exercise.

Some may (correctly) argue that our Integer Queue in C is not really a general purpose queue at all, and we should really write a Queue of "anythings". The problem is that C's type declaration mechanism is not advanced enough to adequately deal with this issue. There are two possible solutions. The safer one is to edit your Integer Queue to be a Queue of whatever you need a Queue of. This change can be accomplished efficiently by defining the type `QueueElement` in the `.h` file, as was done in the above example. 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.

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 less safe) technique is to make `QueueElement` a generic pointer, by doing

```
typedef void* QueueElement
```

Now the Queue module can handle Queues holding any kind of pointer. The danger here 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 client program and get a segmentation fault.

Naming Conventions for ADTs in C

Suppose you wish to implement an ADT in C. The particular ADT is unimportant, so let's just call it a "Blah". You should create the following files at minimum: Blah.c, Blah.h, BlahTest.c. The file Blah.c represents the 'inside' of the black box, Blah.h represents the interface, and BlahTest.c is a kind of dummy client module or test harness used to wring the bugs out of the Blah ADT.

File Blah.h will contain prototypes for all exported ADT operations. It will also contain the line.

```
typedef struct BlahObj* Blah;
```

This defines `Blah` to be a pointer to some struct called `BlahObj`. Blah.c will `#include` Blah.h, and will contain definitions of all exported functions, as well as definitions of private functions and structs as well. Blah.c will also contain the following typedef statement.

```
typedef struct BlahObj{
   // code that defines fields for the Blah ADT
} BlahObj;
```

A client module can then `#include` Blah.h giving it the ability to declare variables of type `Blah`, as well as functions that take `Blah` parameters. However, the client cannot dereference a `Blah`, since the object it points to is not defined in Blah.h. The ADT operations take `Blah` arguments, so the client does not need to (and is in fact unable to) directly access the struct these references point to. Therefore the client can interact with a Blah object only through the exported ADT operations, thereby enforcing the information hiding principle.

Blah.c also contains a constructor:

```
Blah newBlah(...){
   Blah B;
   // code that initializes B in heap memory
   return(B);
}
```

and a destructor

```
void freeBlah(Blah* pB){
   if(pB!=NULL && *pB!=NULL){
      // free all heap memory associated with *pB
      free(*pB);
      *pB = NULL;
   }
}
```

Notice that the destructor is defined in a strange way. It's argument is not a `Blah`, but a pointer to a `Blah` (i.e. a pointer to a pointer to `BlahObj`.) Therefore the destructor is called by passing the address of a `Blah` reference. Each Blah object is explicitly created and destroyed as follows.

```
Blah B = newBlah(...);
// do something with B
freeBlah(&B);
```

Function `freeBlah()` must be defined in this way (i.e. taking a pointer to a `Blah` reference rather than a simple `Blah` reference) since it is the one ADT operation that changes the `Blah` reference itself (in addition to the `BlahObj` it points to) by setting it to NULL.

Recall that all ADT operations must check their own preconditions and exit with a useful error message when one of them is violated. The error message should state the module in which the error occurred (i.e. Blah), the operation in which it occurred, and exactly which precondition was violated. The purpose of this message is to provide diagnostic assistance to whoever is programming a client of the Blah ADT. In this course, that person is you the student, but in a the real world, it may well be another programmer, so you must make the error message as helpful as possible.

In the C language however, each ADT operation has at least one precondition that should be checked before all others, namely that its main reference argument is not NULL. This check must come first since any attempt to dereference a NULL pointer will result in a segmentation fault.

```
void some_op(Blah B){
   if(B==NULL){
      printf("Blah Error: some_op: NULL Blah reference");
      exit(EXIT_FAILURE);
   }
   // check other preconditions
   // proceed with nominal execution
}
```

In some other programming classes you may have used names like `BlahHndl`, `BlahRef` or `BlahPtr` instead of just `Blah`. We have chosen this convention in order to emulate the syntax of other OOP

languages (like Python) as closely as possible. Obviously one name is not intrinsically better than another, but for the sake of consistency, you are required to adhere to the naming conventions outlined here. In particular, the file names Blah.c, Blah.h, BlahTest.c; the function names `newBlah()`, `freeBlah()`, `printBlah()`; and the type names `BlahObj`, and `Blah` are not open to modification.