

## Stacks and Queues (Continued)

As a further application of the ADT Stack we give a simple algorithm to evaluate an arithmetic expression in postfix notation. Informally the algorithm reads the input onto a stack, applying each operator to the operands currently in the top two items of the stack. Providing the expression was well-formed there will always be sufficient operands to allow each operator to be applied, and there should be exactly one value on the stack when the input is exhausted.

### Value (P)

Suppose P is an expression in postfix notation. The algorithm outputs the value of P.

1. Add a right parenthesis at the end of P (as sentinel).
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ')' is reached.
3. If an operand is read, put it on the stack.
4. If an operator OP is read then
  - (a) remove the top two elements of the stack, where a is the top element and b is the next-to-top element.
  - (b) Evaluate b OP a and place the result back on the stack.

5. Set Value equal to the top element on the stack (there should only be one number on the stack).

For example, consider the evaluation of the expression

$P = 5\ 6\ 2\ +\ *\ 12\ 4\ /\ -$

Following the algorithm Value(P) then gives the sequence:

Symbol	Stack contents
5	5
6	5 6
2	5 6 2
+	5 8
*	40
12	40 12
4	40 12 4
/	40 3
-	37
)	

As useful exercises try implementing the algorithms Polish (P,Q) and Value(P) using the Stack implementation in `java.util.Stack` or the (three) versions in the `structure` package.

## Queue

The queue is a computing abstraction from the familiar structure of people waiting for a service and ‘time-stamped’ in some way (e.g. position in physical queue, number of a ticket, arrival in a phone queue etc).

The real world queue usually has two distinct ends: one for arrival, another for serving. The person who arrived first can expect to be served first. A queue is a data type that has first-in first-out (FIFO) behaviour. As with the stack the queue structure finds many immediate applications within the computing environment. For example, in a GUI environment events such as mouse movements and keystrokes, button presses must be held in an event queue pending processing by suitable handlers. The ideas of message queues and printer queues are self-explanatory.

In the ADT Queue the operation of adding an element to the queue is often called enqueue, and the operation of leaving the queue is called serve. (Note that here the word ‘dequeue’ used by Bailey has been replaced with ‘serve’. This is to avoid confusion with a doubly-ended queue which is sometimes called a Dequeue, or Deque.)

## Specification of ADT Queue

**Domain:** A bounded linear sequence of data elements ordered by 'time of arrival'.

### Operations:

**Enqueue** (Queue Q, Object e)

pre : not Full (Q)

post: post-Q is the same as pre-Q except that element e is added to the tail of Q

**Serve** (Queue Q, Object e)

pre: Length(Q) > 0

post: the head of Q is removed and returned, stored in e

**Length** (Queue Q)

post: Length is the number of elements in Q

**Full** (Queue Q) : boolean

post: Full is true if and only if Q is full

**Create** (Queue Q, created boolean)

post: If Q can be created then created is true and Q is an empty queue.

**Destroy** (Queue Q)

post: Queue Q no longer exists

A Java interface for Queue (following W&B p.152) can then be given as follows:

```
public interface Queue {

    public boolean isEmpty();
    //post: true iff queue is empty

    public int size();
    //post: returns length of queue

    public void clear();
    //post: make the queue empty

    public void addLast(Object value);
    //post: the value is added to the tail

    public Object removeFirst();
    //pre: the queue is not empty
    //post: the head of the queue is removed
    and returned

    public Object getFirst();
    //pre: the queue is not empty
    //post: the element at the head of the
    queue is returned

}
```

Note the non-standard names here for key methods. As with the Stack this interface, together with the interfaces which it extends and standard Java methods, conforms to the abstract specification given earlier.

The specification of an ADT Queue is similar to that of Stack except that the operation `serve` will remove the least recently arrived element (instead of the most recently arrived element). When considered as a list insertions to a queue are made at one end (tail) and deletions are made at the other end (head).

Although the abstract specifications of Stack and Queue are quite similar, the need to attend to both ends of a Queue mean the implementation of Queue raises very different issues.

The Java core classes do not include an implementation of the Queue type. But the `structure` package has a `QueueArray`, `QueueList` and `QueueVector`. The two-ended view necessary for a queue makes a list a natural underlying structure. And for fast access to both head and tail a doubly-linked list is appropriate.

Some typical code from the `QueueList` source (Bailey) is as follows:

```
protected List data;

public QueueList() {

//post:  constructs a new, empty queue

    data = new DoublyLinkedList();
}
```

```

}

public void enqueue (Object value)  {

//post: the value is added to the tail

data.addToTail (value) ;

}

    public Object serve()
//pre:  the queue is not empty
//post: the element at the head of the
//queue is removed and returned

return data.removeFromHead()

}

```

Other methods to do with the size of the queue are examples of immediate code re-use (of List methods). Most of the list operations can be performed in constant time (i.e.  $O(1)$ , the time does not depend on the size of the list).

In a vector-based queue we consider the head of the queue to be in the first location of the vector, and the tail of the queue in the last location (at `size() - 1`). The `enqueue` operation is efficiently accomplished with `addElement`, but the `serve` operation requires removal from the beginning and then shifting  $n-1$  elements to the

left. This is  $O(n)$  in complexity and is a major cost of the re-use involved.

When an upper bound on the size of a queue can be determined then a more efficient technique is to use an array data structure in which both the tail and the head 'wrap around' the array. This is achieved by calculating the raw values for head and tail and then taking the modulus (%) with respect to the `data.length`. Details in the lectures and in Bailey Chapter 7 (p.145).

For an array-based version of Queue and a linked list version see W&B Ch.7.