# 3

# Stacks and Queues

Queues and stacks are two special list types of particular importance. They can be implemented using any list implementation, but arrays are a more practical solution for these structures than for general purpose lists.

## 3.1 Stacks

A stack is a list with the operations insert and remove restricted to one end of the list, called the top; insert and remove are usually called push and pop respectively. The natural analogy is to a stack of books: it is easy to add or retrieve a book from the top of a pile, but more difficult in the middle of the pile. Another good model is the spring loaded stack of dishes found in many dining halls and cafeterias; 'push' and 'pop' seem particularly appropriate in this context. The properties of a stack are often referred to as LIFO (Last In First Out), which is sometimes also used as a noun, that is, a stack is a lifo.

Stacks are often useful in systems programming and organization. A simple example of a use for a stack is as a text buffer in a line editor. As a line of text is typed in, it is stored in a stack. When a mistake is noticed and the delete key is pressed the editor merely pops the stack, thus eliminating the most recently typed character.

Stacks are usually implemented as arrays. In any given application there will typically be only a few stacks in use, so the problem of wasted space is not important. If stacks are expected to be highly variable in size, it may be appropriate to use a pointer based approach.

For the array implementation we can borrow directly the naive array based implementation of lists. In the record declaration the `lastentry` field would usually be called `top`, and the `push` and `pop` routines would of course not allow for a parameter indicating position in the stack. Typically the `pop` function will not only remove the top item from the stack but return the value of the item. Sometimes this is the only way to examine a value, and there is no separate `retrieve` function. On the other hand, in some modified implementations, it may be possible to examine any item on the stack, but only to insert and remove at the top. In this case, the `get_top` function might be declared as `get_top(int&,int)`, and the call `get_top(m,-2)` would retrieve the value stored in the second item below the top of the stack.

```
class int_stack {
 public:
  int_stack();    // initialize a stack with default capacity
  int_stack(int); // initialize a stack with capacity at least n

  // The next 4 functions return true on success, false on failure

  bool push(int n);     // push n on the stack
  bool pop(int& n);     // put the top value in n and pop the stack
  bool get_top(int& n); // put the top value in n
  bool clear();         // reinitialize the stack so it is empty

  bool empty();         // return true if the stack is empty, false otherwise
  bool full();          // return true if the stack is full, false otherwise
  int size();           // return number of items currently on the stack

  void dump();          // dump the stack to cerr

 private:
  int top,capacity;
  int* stk;
}
```

**Figure 3.1** Array based stack

### 3.1.1 Arithmetic evaluation

A good illustration of the use of stacks is in the evaluation of arithmetic expressions. We will be concerned with two standard ways to write arithmetic expressions called postfix

(or reverse Polish) and infix. An example of the same expression written in both forms:

$$4 + 5 * 3 - 8$$
$$4\ 5\ 3 * +\ 8\ -$$

To evaluate an expression in postfix you read left to right and perform an operation as soon as it is encountered, using the two operands immediately to the left; these operands are then deleted and replaced by the result. There is never a need for parentheses in postfix; these are equivalent expressions:

$$(4 + 5) * 3 - 8$$
$$4\ \ 5\ +\ 3 * 8\ -$$

You should be able to see quite easily that a stack is just the ticket for keeping track of the evaluation of a postfix expression. The symbols are read left to right; each value is pushed onto a stack; when an operator is read, the action it denotes is performed using the operands at the top of the stack, those operands are popped and the new result is pushed. Let's follow the stack through the evaluation of a simple expression, $945+3*8--$, shown below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 5 | | 3 | | 8 | | |
| | 4 | 4 | 9 | 9 | 27 | 27 | 19 | |
| 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | −10 |

There are some very definite advantages to postfix notation, but infix will probably continue to be much more widely known and used. An obvious approach to evaluating infix notation automatically is to convert to postfix and then use the algorithm outlined above to evaluate. Figure 3.2) shows a step by step conversion of an expression from infix to postfix. When we encounter a number, we can immediately write it in the output postfix expression; we must hold on to operators for a while until it's appropriate to write them. Notice that in postfix both operands must come before the operator, so each operator will have to be held at least one step before it can be written. When an operator is seen in the input, we have to decide whether it indicates that some waiting operators can now be written to output; for example, when the first '−' is seen, we can tell that the waiting '∗' and '+' can be written, because multiplication is done before subtraction in infix expressions, and since the '+' has the same precedence as '−', but occured first, it too can be written.

Except possibly for the parentheses, this should look fairly straightforward, and you might be able to throw together an ad hoc program to handle these operators. Notice

| Remaining Input | Output | Ops Waiting |
|---|---|---|
| $1+2*3-4+5/6/(7+8)$ | | |
| $*3-4+5/6/(7+8)$ | 1 2 | + |
| $-4+5/6/(7+8)$ | 1 2 3 | + * |
| $-4+5/6/(7+8)$ | 1 2 3 * | + |
| $-4+5/6/(7+8)$ | 1 2 3 * + | |
| $+5/6/(7+8)$ | 1 2 3 * + 4 | − |
| $+5/6/(7+8)$ | 1 2 3 * + 4 − | |
| $/6/(7+8)$ | 1 2 3 * + 4 − 5 | + |
| $/(7+8)$ | 1 2 3 * + 4 − 5 6 | + / |
| $(7+8)$ | 1 2 3 * + 4 − 5 6 / | + / |
| ) | 1 2 3 * + 4 − 5 6 / 7 8 | + / + |
| | 1 2 3 * + 4 − 5 6 / 7 8 + | + / |
| | 1 2 3 * + 4 − 5 6 / 7 8 + / | + |
| | 1 2 3 * + 4 − 5 6 / 7 8 + / + | |

**Figure 3.2** Infix to postfix

especially that the operators waiting for action are behaving as if they are on a stack. The key to programming this whole process in a uniform manner is the idea of operator precedence. Normally '$*$' takes precedence over '+'; it is this which governs whether an operator should be written out or saved. A simple precedence scheme might assign to each operator an integer; then the ordering of the integers would reflect the ordering of the operators in terms of precedence. Now when we read an operator with high precedence, say '$*$', and compare it to an operator with low precedence, say '+', waiting on the top of the stack, we know by comparing the precedence numbers that the '$*$' should be saved without first writing out the '+'. Parentheses temporarily disturb the precedences, allowing a '+' to be placed on the stack above a '/'.

The problem with this for more sophisticated use is that some symbols need to have different precedences at different times. It turns out that a slick way to handle parentheses is to treat '(' as an operator, i.e., push it on the 'waiting' stack and use it to control, via its precedence, the order in which other operators are written. The problem is that its precedence must be different when it has just been encountered than when it is on the waiting stack. When we first see '(' it must be put onto the stack immediately, before any waiting symbols are written out; thus it must behave as if it had high precedence. Once on the stack, however, it should remain until a ')' comes along; thus it must behave as if it had low precedence. We can take all this into account by keeping precedence information in a two-dimensional array of boolean values that indicate whether one operator has precedence over another. Each pair of operators can be associated with two spots in this array depending on which is listed first. This allows us to have separate entries for operators

| | | *onstack* | | | | |
|---|---|---|---|---|---|---|
| | | + | − | ∗ | / | ( |
| | + | F | F | F | F | T |
| | − | F | F | F | F | T |
| *incoming* | ∗ | T | T | F | F | T |
| | / | T | T | F | F | T |
| | ( | T | T | T | T | T |
| | ) | F | F | F | F | T |

**Figure 3.3** Operator precedence

when first encountered and when already on the stack. A portion of the precedence array might look like figure 3.3.

Here an entry of F means that the 'incoming' operator should cause the operator on top of the stack to be written as part of the postfix expression, because it is false that the incoming operator takes precedence over the top operator. Notice that I have not bothered to list ')' among the 'onstack' operators, since it will in no case be pushed. The 'driver' routine for the translation using this table would look something like figure 3.4.

```
while ( !prec[token][top_item(op_stk)] ) {
  if (!op_stk.pop(op)) {
    cerr << "op_stk underflow\n";
  } else {
    cout << op;
  }
}
if ( token != RPAREN ) {
  op_stk.push(token);
} else {
  op_stk.pop(op);
}
```

**Figure 3.4** Infix to postfix code

You should work through the above example using this routine and the precedence table. The '(' will now appear on the stack at the appropriate point and will insure that the '+' (adding 7 and 8) gets written before the '/'. You will also notice that the algorithm is not correct as it stands, because there is no provision for stopping the while loop when there is no top operator. One solution is to introduce an 'imaginary' symbol, often denoted '$', which is pushed on the stack at the beginning of processing. This symbol has very low precedence so that nothing will force it to be popped: it will stop the while loop whenever it becomes the top operator. Notice that this '$' need only appear on the 'onstack' side

of the table; since it is not a 'real' symbol we will never actually read one as an incoming symbol. Also there is no automatic way to signal the end of an expression so that all waiting operators can be written out. For this we can introduce an end-of-expression marker ';' with low precedence which will force everything (except '$') to be popped. This marker will only appear on the incoming side of the table.

Now notice that there is really no need to evaluate infix expressions in two stages (translation to postfix and then evaluation). Whenever it is time to write out an operator to the postfix expression, it is time to *do* the operation. That is, if we keep the operands on a separate stack, then we can do the postfix evaluation simultaneously with the translation. This is not only quicker, it eliminates the storage necessary to hold the postfix expression until evaluation. In this case the body of the while loop above will become more complex; it will consist of a switch statement with a label corresponding to each operator that might be on the stack.

Once the framework is in place and working it becomes quite easy to add more operators, pretty much just by adding a new case label with statements that know how to perform the new operation. The unary minus, assignment and exponential operators are all quite easy to add. The exponential operator ($**$) is usually assumed to associate to the right instead of the left. This can easily be handled by setting the table entry `prec[**,**]` equal to T, unlike other diagonal entries which are F.

### 3.1.2 Recursion and Run-time Stacks

Another common and very important use for stacks is the run-time stack. A compiler for a language that allows recursive functions creates this stack when it translates a program into machine language. The stack is in operation when the translated program (*not* the compiler) is running. The translated program uses this stack to keep track of the order in which functions are called and the variables to which each function has access.

When a function is called, whatever local variables it owns become active. If there were no recursive procedures the memory locations for these local variables could be set aside for the duration of the program; the compiler, as it generates code for the procedure body, would know where these values could be found. In a recursive call of a procedure, however, we need to keep track of more than one copy of every local variable: one copy for each time the procedure has been called. Moreover, we can not in general predict how many copies will be needed.

The run-time stack is a stack of **activation records** which contain enough space to store all the local variables (including parameters) of the function. When the compiler needs to generate code involving the use of a local variable, it uses the appropriate loca-

tion in the activation record; at run-time the program knows how to find the top activation record, so it automatically gets the right copy of the local variable. In addition the activation records contain a field indicating where control should be passed when the procedure finishes, i.e., where the next program step can be found. Recursive procedures are not always the best way to solve a problem, but often a recursive process will be very easy to program and understand. The recursive structure of the program can do some very complex bookkeeping almost as if by magic; underlying this is really the run-time stack.

The knapsack problem is an example of one that is particularly easy to solve recursively; a non-recursive solution would be much more difficult to organize. A simple version of the problem goes like this:

> *We have a number, $n$, of objects with weights $w_1, w_2, \ldots, w_n$ (not necessarily distinct). The problem is to determine whether we can choose some subset of the objects with a total weight equal to some specified target weight $W$.*

A solution is simple to state: try all combinations until the target weight is made or until we run out of combinations. On a small set of weights this is easy to do by inspection, but the programming of such a task for a large number of weights is not trivial. (Unfortunately, this is an example of a problem for which the best known solution is of the 'try all possibilities' variety.)

Let's state the proposed solution with a bit more precision and see how this restatement leads almost directly to the recursive procedure to solve the problem. Given a target weight $W$, we first guess that we can use the first weight $w_1$. In that case we have a new version of the knapsack problem to deal with: the target weight is now $(W - w_1)$ and the list of available weights is $w_2, w_3, \ldots, w_n$. We solve this smaller problem; if the answer is *yes* we're in business, because then the answer to the original problem is also *yes*. If the answer to the smaller problem is *no* then we decide that the weight $w_1$ is of no use to us, and instead ask whether we can make the target $W$ out of $w_2, w_3, \ldots, w_n$. We proceed in this manner until we have tried all possible combinations. The only thing left is to examine a couple of special cases in which we can give an immediate answer. If at any point we are trying to solve any version of the knapsack problem with target weight 0 then we can answer *yes* immediately, since we can always make a target of 0 by choosing no weights at all. If we have a target weight which is less than 0, the answer is clearly *no*, since a negative weight makes no sense. Finally, if we have a non-zero target weight but no more weights to consider, we return the answer *no*. This has a nearly direct translation into a recursive procedure, shown in figure 3.5 along with a simple main program to test it. When this program runs, it produces the following output:

```
Target : 10
Weight number: 4   Value: 1
Weight number: 3   Value: 4
Weight number: 1   Value: 5
Success
Target : 23
Failure
```

```
#include <iostream>
using namespace std;
#define MAX 5
bool knapsack (int target, int candidate, int wt[MAX]) {
  if (target == 0)
    return (true);
  else if (target < 0 || candidate >= MAX)
    return (false);
  else if ( knapsack(target-wt[candidate],candidate+1,wt) ) {            // *
    // At this point we have a 'yes' answer to the sub-problem using
    // 'target-weight[candidate]' and 'candidate + 1'. Hence we know
    // that we can use the current weight 'candidate' in a solution to
    // the problem.
    cout << "Weight number: " << candidate;
    cout << "   Value: " << wt[candidate] << endl;
    return (true);
  } else {
    // At this point we know that weight[candidate] won't help,
    // so we try the problem with the same target but starting with
    // the next weight, 'candidate + 1'. }
    return(knapsack(target, candidate + 1,wt));                          // **
  }
}

main() {
int weight[MAX] = {7,5,8,4,1};
int t = 23;
 cout << "Target : 10" << endl;
 cout <<  (knapsack (10,0,weight) ? "\nSuccess" : "\nFailure") << endl;
 cout << "\nTarget : " << t << endl;
 cout <<  (knapsack (t,0,weight) ? "\nSuccess" : "\nFailure") << endl;
}
```

**Figure 3.5** Knapsack function

Let's follow how the run-time stack can keep track of the whole mess. The activation records look like this:

| target |
|---|
| candidate |
| return to |

The target and candidate fields are initialized to the value of the parameters when the function is called. The 'return to' field will have the value '$*$' or '$**$' corresponding to the lines marked in the procedure, except for the first call which is from the main program. Below is a trace of the run-time stack as knapsack works on the problem with target 10 and weights 7, 5, 8, 4, 1. The stack is shown after each push and pop, i.e., as each call to knapsack is about to start and when control is passed back to the calling location. Remember that knapsack returns a boolean value to the point of call where it either controls an if-statement (at $*$) or simply passes the value on ($**$).

| 10 |
|---|
| 0 |
| |

| 10 | 3 |
|---|---|
| 0 | 1 |
| | $*$ |

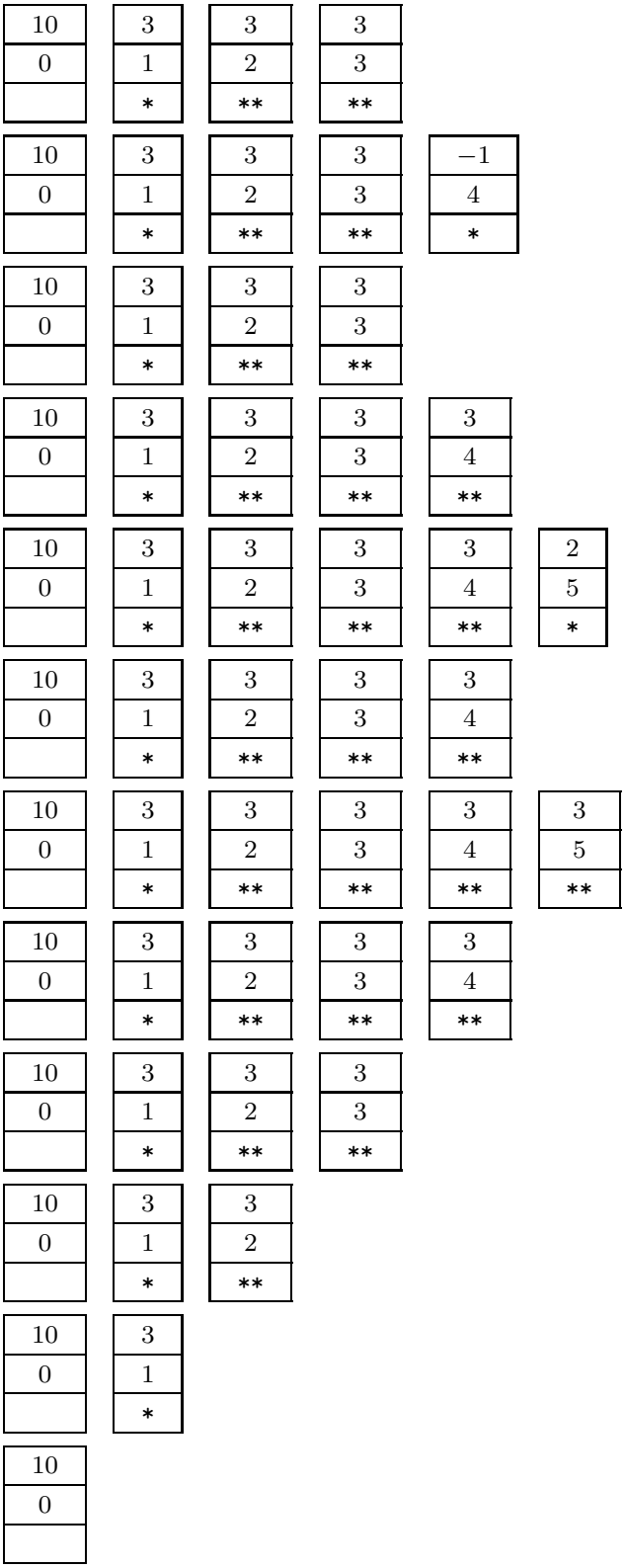
| 10 | 3 | $-2$ |
|---|---|---|
| 0 | 1 | 2 |
| | $*$ | $*$ |

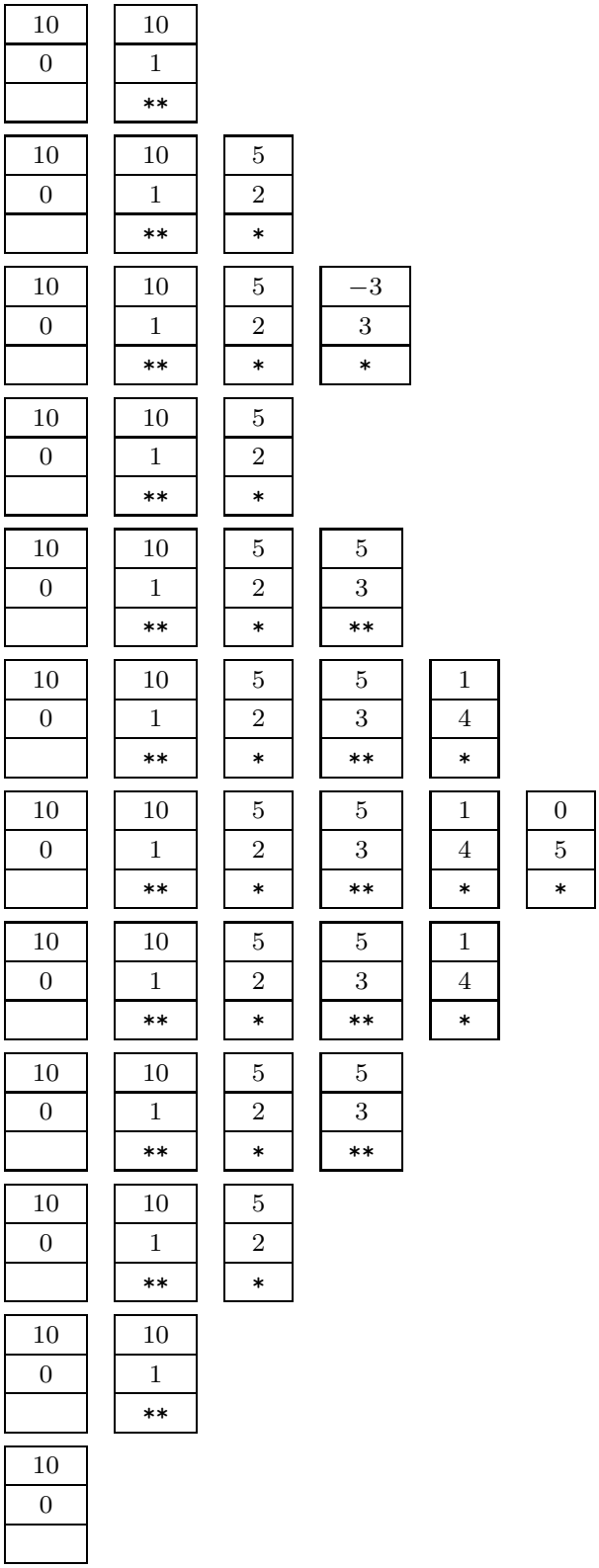
| 10 | 3 |
|---|---|
| 0 | 1 |
| | $*$ |

| 10 | 3 | 3 |
|---|---|---|
| 0 | 1 | 2 |
| | $*$ | $**$ |

| 10 | 3 | 3 | $-5$ |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| | $*$ | $**$ | $*$ |

| 10 | 3 | 3 |
|---|---|---|
| 0 | 1 | 2 |
| | $*$ | $**$ |


10 3 3 3
0 1 2 3
* ** **
10 3 3 3 −1
0 1 2 3 4
* ** ** *
10 3 3 3
0 1 2 3
* ** **
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3 3 2
0 1 2 3 4 5
* ** ** ** *
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3 3 3
0 1 2 3 4 5
* ** ** ** **
10 3 3 3 3
0 1 2 3 4
* ** ** **
10 3 3 3
0 1 2 3
* ** **
10 3 3
0 1 2
* **
10 3
0 1
*
10
0


10
0
10
1
**
10
0
10
1
**
5
2
*
10
0
10
1
**
5
2
*
−3
3
*
10
0
10
1
**
5
2
*
10
0
10
1
**
5
2
*
5
3
**
10
0
10
1
**
5
2
*
5
3
**
1
4
*
10
0
10
1
**
5
2
*
5
3
**
1
4
*
0
5
*
10
0
10
1
**
5
2
*
5
3
**
1
4
*
10
0
10
1
**
5
2
*
5
3
**
10
0
10
1
**
5
2
*
10
0
10
1
**
10
0

## 3.2 Queues

A queue is a list in which insertions can be made only at one end (the rear) and deletions at the other (the front). The properties of a queue are often summarized as FIFO (First In, First Out); the queue is useful whenever some sort of waiting is involved. Queues are widely used in writing system software because of the competition for scarce resources. When a process wants to use the CPU, the printer, the disk or any other system device it might well be placed on a queue to wait its turn. (Frequently this will be some sort of modified queue since queues are a bit inflexible. For example, you can remove an item from the printer queue before the system 'deletes' it (i.e., prints it), and there is a scheme of priorities so that even if you 'get in line' first, some job with higher priority might get printed first.)

Sometimes what is called a buffer is a queue. For example, when you 'type ahead' at a terminal without seeing the characters you are typing, they are being stored in some sort of text buffer. When the computer gets around to you again, it takes the waiting characters from the front of the queue, while you may continue to type and add them at the rear.

You should always try the simplest possible implementation of a new structure, both because it's often better to have something that works than nothing at all and because the simplest way may be the best if your application is not particularly demanding. The simplest possible way to organize a queue is in an array with the rear 'growing' toward the higher indexed slots in the array and the front moving behind it. This is easy to implement and quite efficient. The problem is that when the rear hits the top of the array no more items can be added, even if there are empty spots at the bottom of the array. The obvious solution (if one is needed) is to move everything down whenever a deletion is made; this probably conforms closely to your idea of a 'queue' in the normal sense of 'waiting line.' We could implement this by using the naive array implementation of lists and modifying insert and delete to conform to queue specifications (this just amounts to removing one parameter, making it impossible to specify where to do the insert or delete). Alternately, we might move the whole queue down only when we actually run out of room at the top. Then we get the benefit of many incremental moves for the cost of only one (notice that it is not more expensive to move a block of items all the way down the array than it is to move everything one position down; this is because arrays are implemented as random access structures). For lists we needed to move items after every delete because the items on the list had to occupy adjacent spots in the array (otherwise the bookkeeping would get out of hand). The items on the queue will always be adjacent so the move is not crucial

to the function of the queue but rather a way to make more efficient use of the allotted space.

Either of the modified versions above will work and will make the best possible use of space, that is, will not crash for lack of space until the array is really full. Nevertheless, queues are often quite active beasts and the cost of repeated moves of long blocks of items may add up. Suppose we start with the simplest version again (just let the rear go until it hits the top) and modify it in a different way. Instead of moving the whole queue we can simply make our addition in an existing empty position. We have to do this in some reasonable way so that it is easy to tell who is next in line. Fortunately there is a natural answer: we simply let the queue 'wrap around' to the beginning of the array again, and grow toward the retreating front. This implementation is referred to as a 'circular array' implementation and is the one most frequently encountered. It makes very good use of space, it is efficient in operation; its main drawback is the possibility that the array will become full, a property shared by all simple array implementations.

It should be obvious that a queue can be implemented using any linked list implementation, but this is not often necessary. As a rule only some small number of queues will be needed in any particular environment, so the lack of flexibility in the use of space is rarely a problem. The necessity of declaring the space for the queue in advance is also not usually a problem. Queues are used for waiting; if a queue grows until it fills the array it is in, then of course you should try increasing the size of the array. If the problem persists, it probably means that the resource being waited for is badly overloaded and increasing the size of the queue without bound will not solve the problem. You should usually consider using linked lists to implement queues only when the size of the queue in normal operation varies considerably or the space an array would occupy when the queue is nearly empty is needed elsewhere. In some situations it may be deemed unacceptable ever to get a full queue, in which case a linked implementation would be appropriate.

The details of the circular queue implementation deserve some discussion. We will need two pieces of information for each queue: where in the array we can find the front and the rear of the queue. The action of the insert and remove routines should be fairly clear: to insert we find the rear of the queue, insert in the appropriate spot and update the value of the rear variable. If the queue is full, i.e. size=capacity, then instead of failing a good implementation would use "new" to get a larger array, transfer the data to the new array, delete the old array, and then add the item to the queue. Typical functions are shown in figure 3.6. I have really combined the `remove` and `retrieve` operations since in many uses of a queue retrieve without a remove is not done; of course they could be implemented separately.

```
class cell_queue {
private:
  int front,rear,capacity,size;
  cell* the_queue;
public:
  cell_queue();
  cell_queue(int n);
  ~cell_queue();
  bool enqueue (cell n);
  bool dequeue(cell& n);
};

bool cell_queue::enqueue (cell c) {
  if (size < capacity) {
    rear = (rear + 1) % (capacity);
    the_queue[rear] = c;
    size++;
    return true;
  } else {
    return false;
  }
}

bool cell_queue::dequeue(cell& c) {
  if (size > 0) {
    c = the_queue[front];
    front = (front + 1) % (capacity);
    size--;
    return true;
  } else {
    return false;
  }
}
```

**Figure 3.6** Circular queue