Stacks and Queues

4

As the first chapter explained, abstract data types allow us to delay the specific implementation of a data type until it is well understood what operations are required to operate on the data. In fact, these operations determine which implementation of the data type is most efficient in a particular situation. This situation is illustrated by two data types, stacks and queues, which are described by a list of operations. Only after the list of the required operations is determined do we present some possible implementations and compare them.

4.1 Stacks

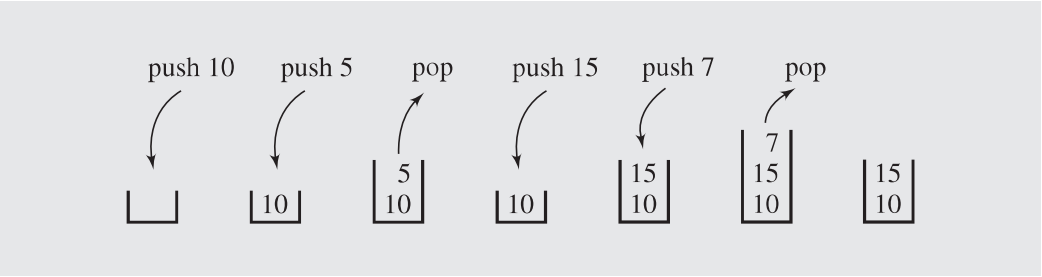
A *stack* is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: new trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called an *LIFO* structure: last in/first out.

A tray can be taken only if there is at least one tray on the stack, and a tray can be added to the stack only if there is enough room; that is, if the stack is not too high. Therefore, a stack is defined in terms of operations that change its status and operations that check this status. The operations are as follows:

- *clear()*—Clear the stack.
- *isEmpty()*—Check to see if the stack is empty.
- *push(el)*—Put the element *el* on the top of the stack.
- *pop()*—Take the topmost element from the stack.
- *topEl()*—Return the topmost element in the stack without removing it.

A series of push and pop operations is shown in Figure 4.1. After pushing number 10 onto an empty stack, the stack contains only this number. After pushing 5 on the stack, the number is placed on top of 10 so that, when the popping operation is executed, 5 is removed from the stack, because it arrived after 10, and 10 is left on

FIGURE 4.1 A series of operations executed on a stack.



the stack. After pushing 15 and then 7, the topmost element is 7, and this number is removed when executing the popping operation, after which the stack contains 10 at the bottom and 15 above it.

Generally, the stack is very useful in situations when data have to be stored and then retrieved in reverse order. One application of the stack is in matching delimiters in a program. This is an important example because delimiter matching is part of any compiler: No program is considered correct if the delimiters are mismatched.

In C++ programs, we have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and "*/". Here are examples of C++ statements that use delimiters properly:

```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
```

These examples are statements in which mismatching occurs:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

A particular delimiter can be separated from its match by other delimiters; that is, delimiters can be nested. Therefore, a particular delimiter is matched up only after all the delimiters following it and preceding its match have been matched. For example, in the condition of the loop

```
while (m < (n[8] + o))
```

the first opening parenthesis must be matched with the last closing parenthesis, but this is done only after the second opening parenthesis is matched with the next to last closing parenthesis; this, in turn, is done after the opening square bracket is matched with the closing bracket.

The delimiter matching algorithm reads a character from a C++ program and stores it on a stack if it is an opening delimiter. If a closing delimiter is found, the delimiter is compared to a delimiter popped off the stack. If they match, processing continues; if not, processing discontinues by signaling an error. The processing of the

C++ program ends successfully after the end of the program is reached and the stack is empty. Here is the algorithm:

```
delimiterMatching(file)
  read character ch from file;
   while not end of file
      if ch is '(', '[', or '{'
            push(ch);
      else if ch is ')', ']', or '}'
            if ch and popped off delimiter do not match
               failure;
      else if ch is '/'
            read the next character;
            if this character is '*'
                  skip all characters until "*/" is found and report an error
                  if the end of file is reached before "*/" is encountered;
            else ch = the character read in;
                  continue; // go to the beginning of the loop;
   // else ignore other characters;
      read next character ch from file;
   if stack is empty
       success;
   else failure;
```

Figure 4.2 shows the processing that occurs when applying this algorithm to the statement

```
s=t[5]+u/(v*(w+y));
```

The first column in Figure 4.2 shows the contents of the stack at the end of the loop before the next character is input from the program file. The first line shows the initial situation in the file and on the stack. Variable `ch` is initialized to the first character of the file, letter s, and in the first iteration of the loop, the character is simply ignored. This situation is shown in the second row in Figure 4.2. Then the next character, equal sign, is read. It is also ignored and so is the letter t. After reading the left bracket, the bracket is pushed onto the stack so that the stack now has one element, the left bracket. Reading digit 5 does not change the stack, but after the right bracket becomes the value of `ch`, the topmost element is popped off the stack and compared with `ch`. Because the popped off element (left bracket) matches `ch` (right bracket), the processing of input continues. After reading and discarding the letter u, a slash is read and the algorithm checks whether it is part of the comment delimiter by reading the next character, a left parenthesis. Because the character read is not an asterisk, the slash is not a beginning of a comment, so `ch` is set to left parenthesis. In the next iteration, this parenthesis is pushed onto the stack and processing continues, as shown in Figure 4.2. After reading the last character, a semicolon, the loop is exited and the stack is checked. Because it is empty (no unmatched delimiters are left), success is pronounced.

FIGURE 4.2 Processing the statement `s=t[5]+u/(v*(w+y));` with the algorithm `delimiterMatching()`.

Stack	Nonblank Character Read	Input Left
empty		s = t[5] + u / (v * (w + y));
empty	s	= t[5] + u / (v * (w + y));
empty	=	t[5] + u / (v * (w + y));
empty	t	[5] + u / (v * (w + y));
[	[	5] + u / (v * (w + y));
[	5	] + u / (v * (w + y));
empty	]	+ u / (v * (w + y));
empty	+	u / (v * (w + y));
empty	u	/ (v * (w + y));
empty	/	(v * (w + y));
(	(	v * (w + y));
(	v	* (w + y));
(	*	(w + y));
(		
(	(	w + y));
(		
(	w	+y));
(		
(	+	y));
(		
(	y	));
(	)	);
empty	)	;
empty	;	

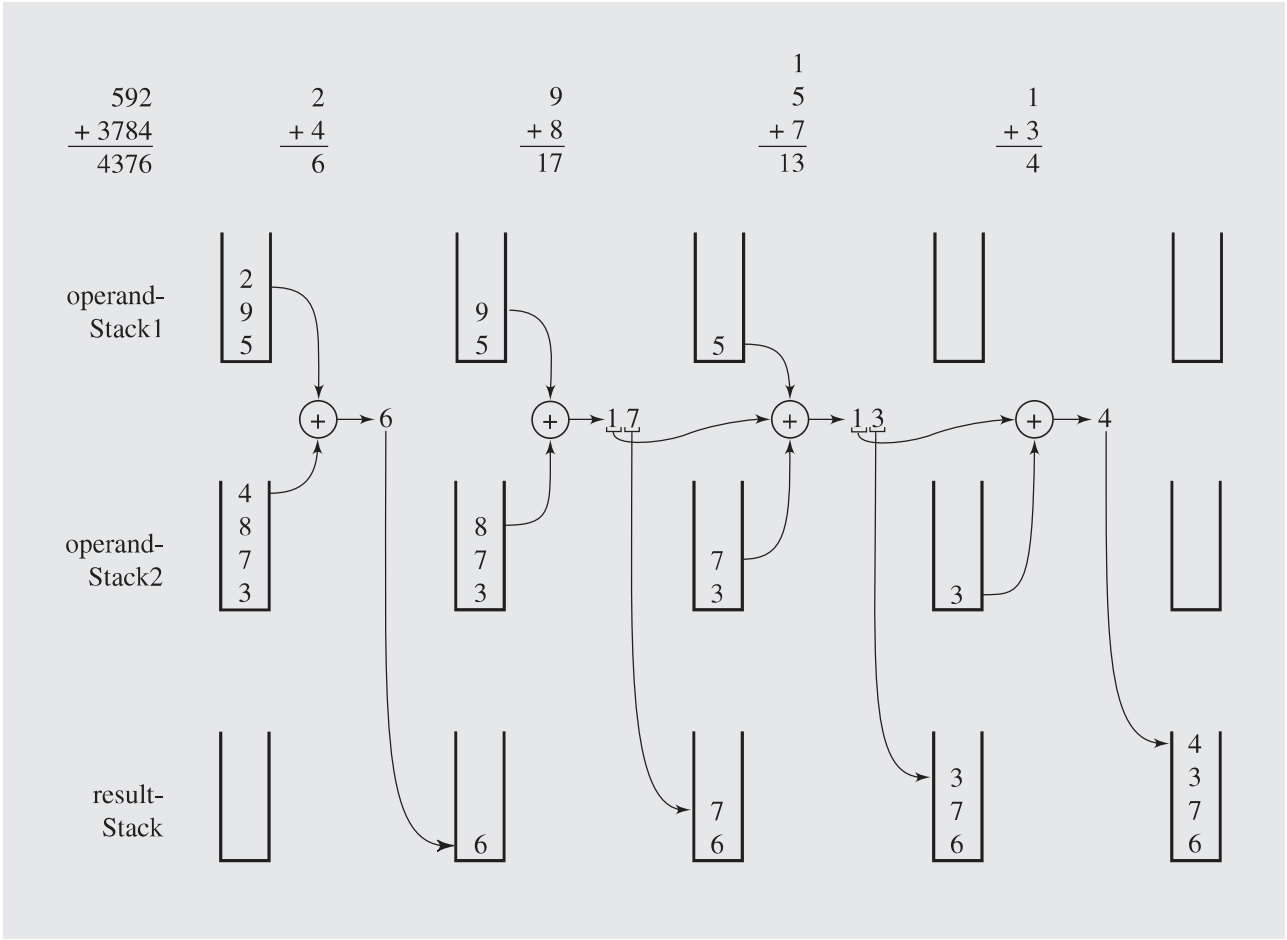
As another example of stack application, consider adding very large numbers. The largest magnitude of integers is limited, so we are not able to add 18,274,364,583,929,273,748,459,595,684,373 and 8,129,498,165,026,350,236, because integer variables cannot hold such large values, let alone their sum. The problem can be solved if we treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks. The pseudocode for this algorithm is as follows:

```
addingLargeNumbers()
    read the numerals of the first number and store the numbers corresponding to
    them on one stack;
    read the numerals of the second number and store the numbers corresponding
    to them on another stack;
    carry = 0;
    while at least one stack is not empty
        pop a number from each nonempty stack and add them to carry;
        push the unit part on the result stack;
        store carry in carry;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;
```

Figure 4.3 shows an example of the application of this algorithm. In this example, numbers 592 and 3,784 are added.

1. Numbers corresponding to digits composing the first number are pushed onto `operandStack1`, and numbers corresponding to the digits of 3,784 are pushed onto `operandStack2`. Note the order of digits on the stacks.

FIGURE 4.3 An example of adding numbers 592 and 3,784 using stacks.



2. Numbers 2 and 4 are popped from the stacks, and the result, 6, is pushed onto `resultStack`.
3. Numbers 9 and 8 are popped from the stacks, and the unit part of their sum, 7, is pushed onto `resultStack`; the tens part of the result, number 1, is retained as a carry in the variable `carry` for subsequent addition.
4. Numbers 5 and 7 are popped from the stacks, added to the carry, and the unit part of the result, 3, is pushed onto `resultStack`, and the carry, 1, becomes a value of the variable `carry`.
5. One stack is empty, so a number is popped from the nonempty stack, added to carry, and the result is stored on `resultStack`.
6. Both operand stacks are empty, so the numbers from `resultStack` are popped and printed as the final result.

Consider now implementation of our abstract stack data structure. We used push and pop operations as though they were readily available, but they also have to be implemented as functions operating on the stack.

A natural implementation for a stack is a flexible array, that is, a vector. Figure 4.4 contains a generic stack class definition that can be used to store any type of objects. Also, a linked list can be used for implementation of a stack (Figure 4.5).

FIGURE 4.4 A vector implementation of a stack.

```
//*********************  genStack.h  *************************
//      generic class for vector implementation of stack

#ifndef STACK
#define STACK

#include <vector>

template<class T, int capacity = 30>
class Stack {
public:
    Stack() {
        pool.reserve(capacity);
    }
    void clear() {
        pool.clear();
    }
    bool isEmpty() const {
        return pool.empty();
    }
    T& topEl() {
```

FIGURE 4.4 *(continued)*

```
        return pool.back();
    }
    T pop() {
        T el = pool.back();
        pool.pop_back();
        return el;
    }
    void push(const T& el) {
        pool.push_back(el);
    }
private:
    vector<T> pool;
};

#endif
```

FIGURE 4.5 Implementing a stack as a linked list.

```
//**********************  genListStack.h  *************************
//     generic stack defined as a doubly linked list

#ifndef LL_STACK
#define LL_STACK

#include <list>

template<class T>
class LLStack {
public:
    LLStack() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
```

Continues

FIGURE 4.5 *(continued)*

```
    }
    T& topEl() {
        return lst.back();
    }
    T pop() {
        T el = lst.back();
        lst.pop_back();
        return el;
    }
    void push(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

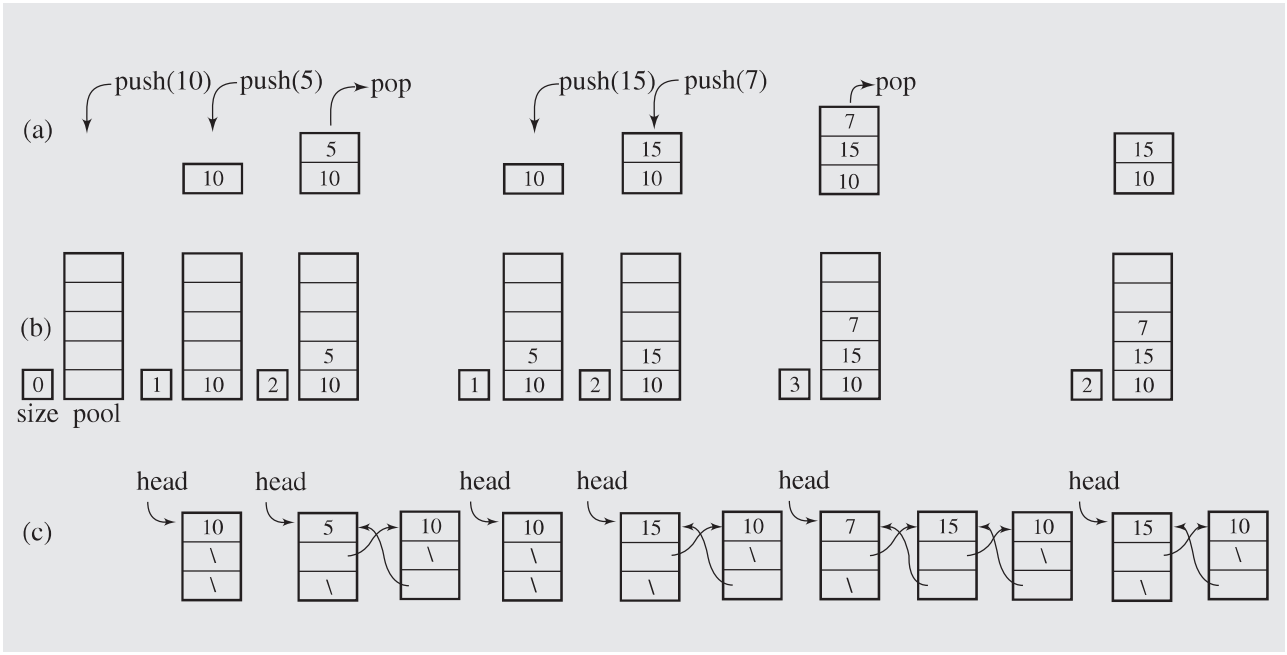
Figure 4.6 shows the same sequence of push and pop operations as Figure 4.1 with the changes that take place in the stack implemented as a vector (Figure 4.6b) and as a linked list (Figure 4.6c).

The linked list implementation matches the abstract stack more closely in that it includes only the elements that are on the stack because the number of nodes in the list is the same as the number of stack elements. In the vector implementation, the capacity of the stack can often surpass its size.

The vector implementation, like the linked list implementation, does not force the programmer to make a commitment at the beginning of the program concerning the size of the stack. If the size can be reasonably assessed in advance, then the predicted size can be used as a parameter for the stack constructor to create in advance a vector of the specified capacity. In this way, an overhead is avoided to copy the vector elements to a new larger location when pushing a new element to the stack for which size equals capacity.

It is easy to see that in the vector and linked list implementations, popping and pushing are executed in constant time $O(1)$. However, in the vector implementation, pushing an element onto a full stack requires allocating more memory and copies the elements from the existing vector to a new vector. Therefore, in the worst case, pushing takes $O(n)$ time to finish.

FIGURE 4.6 A series of operations executed on (a) an abstract stack and the stack implemented (b) with a vector and (c) with a linked list.



4.2 QUEUES

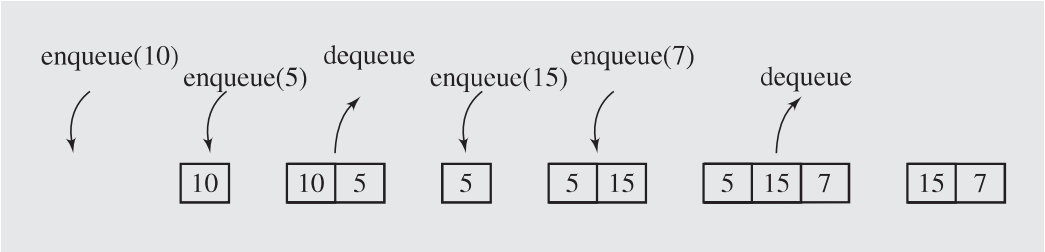
A *queue* is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed. A queue is an *FIFO* structure: first in/first out.

Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- *clear()*—Clear the queue.
- *isEmpty()*—Check to see if the queue is empty.
- *enqueue(el)*—Put the element *el* at the end of the queue.
- *dequeue()*—Take the first element from the queue.
- *firstEl()*—Return the first element in the queue without removing it.

A series of enqueue and dequeue operations is shown in Figure 4.7. This time—unlike for stacks—the changes have to be monitored both at the beginning of the queue and at the end. The elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue (Figure 4.7).

FIGURE 4.7 A series of operations executed on a queue.



For an application of a queue, consider the following poem written by Lewis Carroll:

Round the wondrous globe I wander wild,

Up and down-hill—Age succeeds to youth—

Toiling all in vain to find a child

Half so loving, half so dear as Ruth.

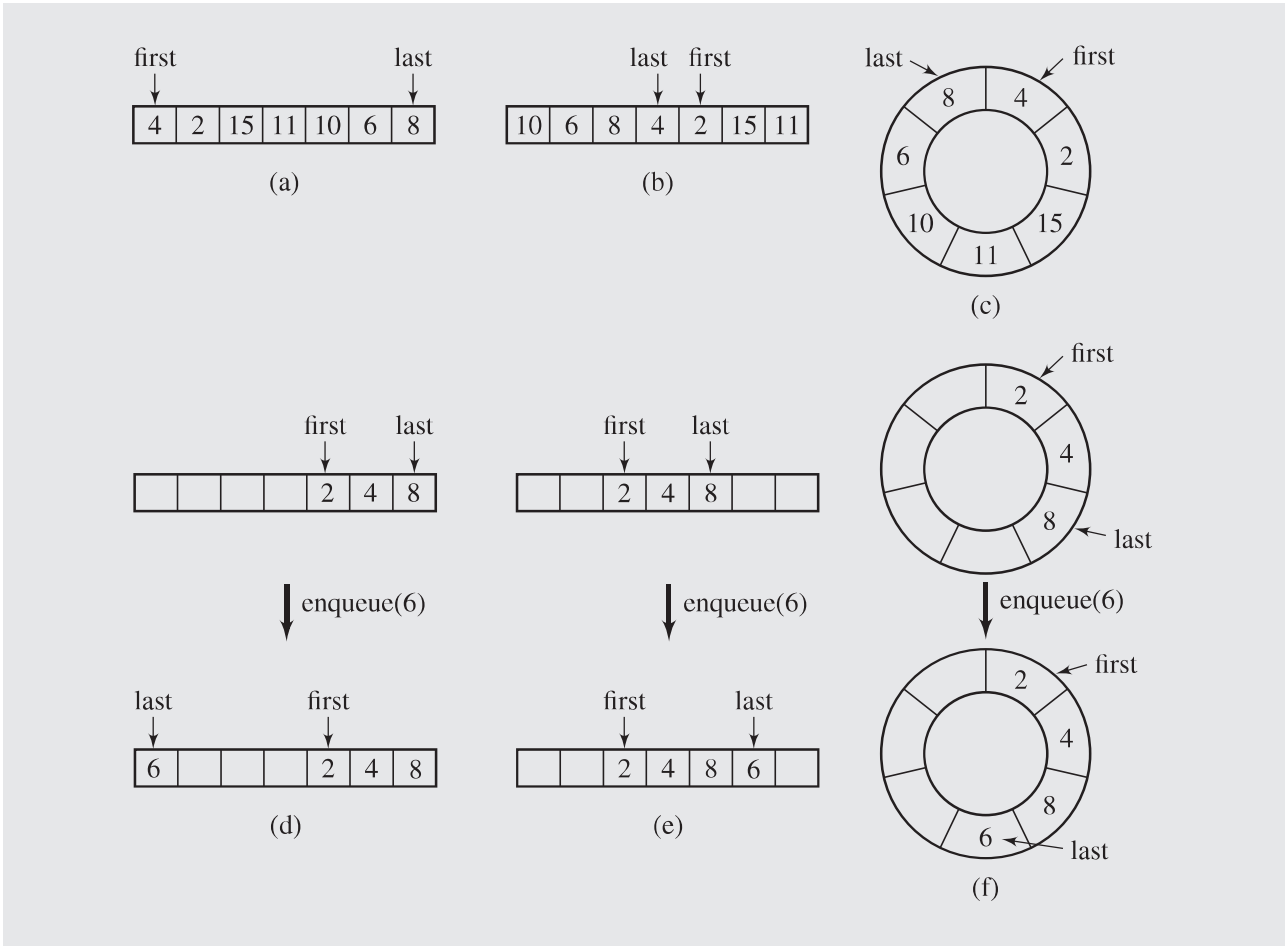
The poem is dedicated to Ruth Dymes, which is indicated not only by the last word of the poem, but also by reading in sequence the first letters of each line, which also spells Ruth. This type of poem is called an acrostic, and it is characterized by initial letters that form a word or phrase when taken in order. To see whether a poem is an acrostic, we devise a simple algorithm that reads a poem, echoprints it, retrieves and stores the first letter from each line on a queue, and after the poem is processed, all the stored first letters are printed in order. Here is an algorithm:

```
acrosticIndicator()
    while not finished
        read a line of poem;
        enqueue the first letter of the line;
        output the line;
    while queue is not empty
        dequeue and print a letter;
```

There is a more significant example to follow, but first consider the problem of implementation.

One possible queue implementation is an array, although this may not be the best choice. Elements are added to the end of the queue, but they may be removed from its beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new elements, whereby the end of the queue may occur at the beginning of the array. This situation is better pictured as a circular array, as Figure 4.8c illustrates. The queue is full if the first element immediately precedes the last element in the counterclockwise direction. However, because a circular array is implemented with a "normal" array, the queue is full if either the first element is in the first cell and the last element is in the last cell (Figure 4.8a) or if the first element is right after the last (Figure 4.8b). Similarly, *enqueue()* and *dequeue()* have to consider the possibility of wrapping around the array when adding or removing elements. For

FIGURE 4.8 (a–b) Two possible configurations in an array implementation of a queue when the queue is full. (c) The same queue viewed as a circular array. (f) Enqueuing number 6 to a queue storing 2, 4, and 8. (d–e) The same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle.



example, *enqueue()* can be viewed as operating on a circular array (Figure 4.8c), but in reality, it is operating on a one-dimensional array. Therefore, if the last element is in the last cell and if any cells are available at the beginning of the array, a new element is placed there (Figure 4.8d). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 4.8e). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 4.8f).

Figure 4.9 contains possible implementations of member functions that operate on queues.

A more natural queue implementation is a doubly linked list, as offered in the previous chapter and also in `STL`'s `list` (Figure 4.10).

In both suggested implementations enqueuing and dequeuing can be executed in constant time $O(1)$, provided a doubly linked list is used in the list implementation. In the singly linked list implementation, dequeuing requires $O(n)$ operations

FIGURE 4.9 Array implementation of a queue.

```
//********************  genArrayQueue.h  **********************
//             generic queue implemented as an array

#ifndef ARRAY_QUEUE
#define ARRAY_QUEUE

template<class T, int size = 100>
class ArrayQueue {
public:
    ArrayQueue() {
        first = last = -1;
    }
    void enqueue(T);
    T dequeue();
    bool isFull()  {
        return first == 0 && last == size-1 || first == last + 1;
    }
    bool isEmpty() {
        return first == -1;
    }
private:
    int first, last;
    T storage[size];
};

template<class T, int size>
void ArrayQueue<T,size>::enqueue(T el) {
   if (!isFull())
        if (last == size-1 || last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
   else cout << "Full queue.\n";
}

template<class T, int size>
T ArrayQueue<T,size>::dequeue() {
```

FIGURE 4.9 *(continued)*

```
    T tmp;
    tmp = storage[first];
    if (first == last)
         last = first = -1;
    else if (first == size-1)
         first = 0;
    else first++;
    return tmp;
}

#endif
```

FIGURE 4.10 Linked list implementation of a queue.

```
//**********************  genQueue.h  *************************
//     generic queue implemented with doubly linked list

#ifndef DLL_QUEUE
#define DLL_QUEUE

#include <list>

template<class T>
class Queue {
public:
    Queue() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
    }
    T& front() {
        return lst.front();
    }
```

Continues

FIGURE 4.10 *(continued)*

```
    T dequeue() {
        T el = lst.front();
        lst.pop_front();
        return el;
    }
    void enqueue(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

primarily to scan the list and stop at the next to last node (see the discussion of `deleteFromTail()` in Section 3.1.2).

Figure 4.11 shows the same sequence of enqueue and dequeue operations as Figure 4.7, and indicates the changes in the queue implemented as an array (Figure 4.11b) and as a linked list (Figure 4.11c). The linked list keeps only the numbers that the logic of the queue operations indicated by Figure 4.11a requires. The array includes all the numbers until it fills up, after which new numbers are included starting from the beginning of the array.

Queues are frequently used in simulations to the extent that a well-developed and mathematically sophisticated theory of queues exists, called *queuing theory,* in which various scenarios are analyzed and models are built that use queues. In queuing processes there are a number of customers coming to servers to receive service. The throughput of the server may be limited. Therefore, customers have to wait in queues before they are served, and they spend some amount of time while they are being served. By customers, we mean not only people, but also objects. For example, parts on an assembly line in the process of being assembled into a machine, trucks waiting for service at a weighing station on an interstate, or barges waiting for a sluice to be opened so they can pass through a channel also wait in queues. The most familiar examples are lines in stores, post offices, or banks. The types of problems posed in simulations are: How many servers are needed to avoid long queues? How large must the waiting space be to put the entire queue in it? Is it cheaper to increase this space or to open one more server?

As an example, consider Bank One which, over a period of three months, recorded the number of customers coming to the bank and the amount of time needed to serve them. The table in Figure 4.12a shows the number of customers who arrived during one-minute intervals throughout the day. For 15% of such intervals, no customers arrived, for 20%, only one arrived, and so on. Six clerks were employed, no lines were ever observed, and the bank management wanted to know whether six clerks were too

FIGURE 4.11 A series of operations executed on (a) an abstract queue and the queue implemented (b) with an array and (c) with a linked list.

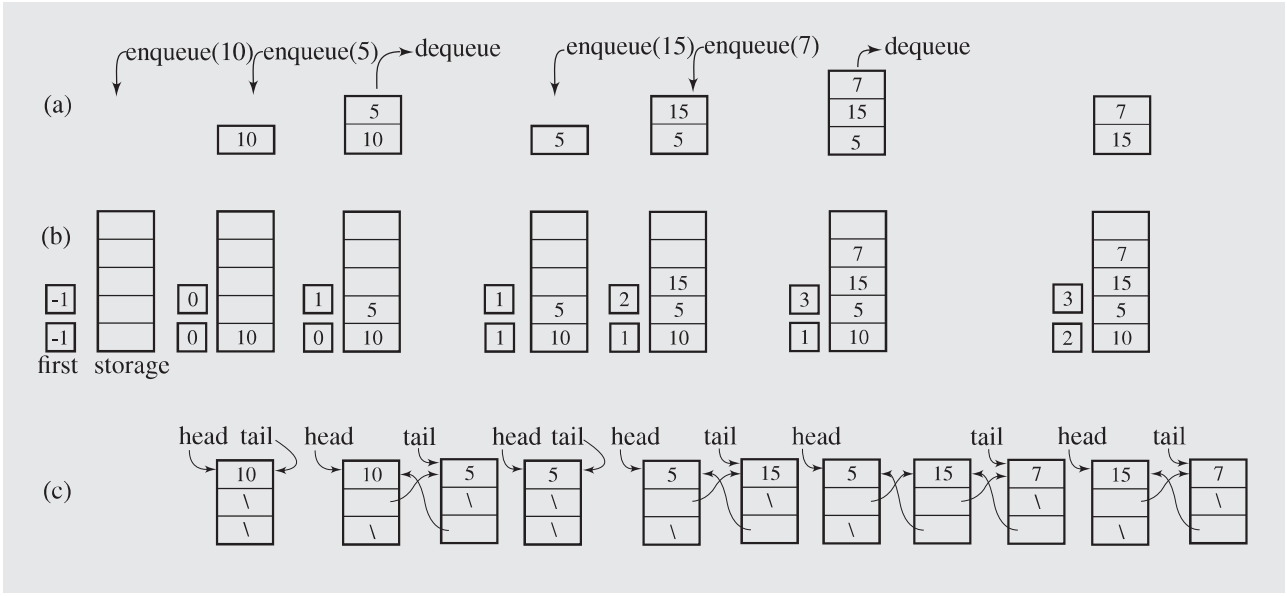


FIGURE 4.12 Bank One example: (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer.

Number of Customers Per Minute	Percentage of One-Minute Intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61–70
4	30	71–100

(a)

Amount of Time Needed for Service in Seconds	Percentage of Customers	Range
0	0	—
10	0	—
20	0	—
30	10	1–10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100

(b)

many. Would five suffice? Four? Maybe even three? Can lines be expected at any time? To answer these questions, a simulation program was written that applied the recorded data and checked different scenarios.

The number of customers depends on the value of a randomly generated number between 1 and 100. The table in Figure 4.12a identifies five ranges of numbers from 1 to 100, based on the percentages of one-minute intervals that had 0, 1, 2, 3, or 4 customers. If the random number is 21, then the number of customers is 1; if the random number is 90, then the number of customers is 4. This method simulates the rate of customers arriving at Bank One.

In addition, analysis of the recorded observations indicates that no customer required 10-second or 20-second transactions, 10% required 30 seconds, and so on, as indicated in Figure 4.12b. The table in 4.12b includes ranges for random numbers to generate the length of a transaction in seconds.

Figure 4.13 contains the program simulating customer arrival and transaction time at Bank One. The program uses three arrays. `arrivals[]` records the percentages of one-minute intervals depending on the number of the arrived customers. The array `service[]` is used to store the distribution of time needed for service. The amount of time is obtained by multiplying the index of a given array cell by 10. For example, `service[3]` is equal to 10, which means that 10% of the time a customer required $3 \cdot 10$ seconds for service. The array `clerks[]` records the length of transaction time in seconds.

FIGURE 4.13 Bank One example: implementation code.

```
#include <iostream>
#include <cstdlib>

using namespace std;

#include "genQueue.h"
int option(int percents[]) {
    register int i = 0, choice = rand()%100+1, perc;
    for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
    return i;
}

int main() {
    int arrivals[] = {15,20,25,10,30};
    int service[] = {0,0,0,10,5,10,10,0,15,25,10,15};
    int clerks[] = {0,0,0,0}, numOfClerks = sizeof(clerks)/sizeof(int);
    int customers, t, i, numOfMinutes = 100, x;
    double maxWait = 0.0, currWait = 0.0, thereIsLine = 0.0;
    Queue<int> simulQ;
    cout.precision(2);
    for (t = 1; t <= numOfMinutes; t++) {
```

FIGURE 4.13 *(continued)*

```
        cout << " t = " << t;
        for (i = 0; i < numOfClerks; i++)// after each minute subtract
            if (clerks[i] < 60)          // at most 60 seconds from time
                 clerks[i] = 0;          // left to service the current
            else clerks[i] -= 60;        // customer by clerk i;
        customers = option(arrivals);
        for (i = 0; i < customers; i++) {// enqueue all new customers
            x = option(service)*10;      // (or rather service time
            simulQ.enqueue(x);           // they require);
            currWait += x;
        }
        // dequeue customers when clerks are available:
        for (i = 0; i < numOfClerks && !simulQ.isEmpty(); )
           if (clerks[i] < 60)        {
                 x = simulQ.dequeue();  // assign more than one customer
                 clerks[i] += x;        // to a clerk if service time
                 currWait  -= x;        // is still below 60 sec;
           }
           else i++;
        if (!simulQ.isEmpty()) {
             thereIsLine++;
             cout << " wait = " << currWait/60.0;
             if (maxWait < currWait)
                 maxWait = currWait;
        }
        else cout << " wait = 0;";
        }
        cout << "\nFor " << numOfClerks << " clerks, there was a line "
             << thereIsLine/numOfMinutes*100.0 << "% of the time;\n"
             << "maximum wait time was " << maxWait/60.0 << " min.";
        return 0;
}
```

For each minute (represented by the variable `t`), the number of arriving customers is randomly chosen, and for each customer, the transaction time is also randomly determined. The function `option()` generates a random number, finds the range into which it falls, and then outputs the position, which is either the number of customers or a tenth of the number of seconds.

Executions of this program indicate that six and five clerks are too many. With four clerks, service is performed smoothly; 25% of the time there is a short line of waiting customers. However, three clerks are always busy and there is always a long line of customers waiting. Bank management would certainly decide to employ four clerks.

4.3 Priority Queues

In many situations, simple queues are inadequate, because first in/first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue. On roads with tollbooths, some vehicles may be put through immediately, even without paying (police cars, ambulances, fire engines, and the like). In a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of a system, even though P_1 was put on the queue of waiting processes before P_2. In situations like these, a modified queue, or *priority queue,* is needed. In priority queues, elements are dequeued according to their priority and their current queue position.

The problem with a priority queue is in finding an efficient implementation that allows relatively fast enqueuing and dequeuing. Because elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most likely to be dequeued and that the elements put at the end will be the last candidates for dequeuing. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as frequency of use, birthday, salary, position, status, and others. It can also be the time of scheduled execution on the queue of processes, which explains the convention used in priority queue discussions in which higher priorities are associated with lower numbers indicating priority.

Priority queues can be represented by two variations of linked lists. In one type of linked list, all elements are entry ordered, and in another, order is maintained by putting a new element in its proper position according to its priority. In both cases, the total operational times are $O(n)$ because, for an unordered list, adding an element is immediate but searching is $O(n)$, and in a sorted list, taking an element is immediate but adding an element is $O(n)$.

Another queue representation uses a short ordered list and an unordered list, and a threshold priority is determined (Blackstone et al. 1981). The number of elements in the sorted list depends on a threshold priority. This means that in some cases this list can be empty and the threshold may change dynamically to have some elements in this list. Another way is always having the same number of elements in the sorted list; the number $\sqrt{n}$ is a good candidate. Enqueuing takes on the average $O(\sqrt{n})$ time and dequeuing is immediate.

Another implementation of queues was proposed by J. O. Hendriksen (1977, 1983). It uses a simple linked list with an additional array of pointers to this list to find a range of elements in the list in which a newly arrived element should be included.

Experiments by Douglas W. Jones (1986) indicate that a linked list implementation, in spite of its $O(n)$ efficiency, is best for 10 elements or less. The efficiency of the two-list version depends greatly on the distribution of priorities, and it may be excellent or as poor as that of the simple list implementation for large numbers of elements. Hendriksen's implementation, with its $O(\sqrt{n})$ complexity, operates consistently well with queues of any size.

4.4 Stacks in the Standard Template Library

A generic stack class is implemented in the STL as a container adaptor: it uses a container to make it behave in a specified way. The stack container is not created anew; it is an adaptation of an already existing container. By default, `deque` is the underlying container, but the user can also choose either `list` or `vector` with the following declarations:

```
stack<int> stack1;                    // deque by default
stack<int,vector<int> > stack2;       // vector
stack<int,list<int> > stack3;         // list
```

Member functions in the container `stack` are listed in Figure 4.14. Note that the return type of `pop()` is void; that is, `pop()` does not return a popped off element. To have access to the top element, the member function `top()` has to be used. Therefore, the popping operation discussed in this chapter has to be implemented with a call to `top()` followed by the call to `pop()`. Because popping operations in user programs are intended for capturing the popped off element most of the time and not only for removing it, the desired popping operation is really a sequence of the two member functions from the container `stack`. To contract them to one operation, a new class can be created that inherits all operations from `stack` and redefines `pop()`. This is a solution used in the case study at the end of the chapter.

FIGURE 4.14 A list of `stack` member functions.

Member Function	Operation
`bool empty() const`	Return `true` if the stack includes no element and `false` otherwise.
`void pop()`	Remove the top element of the stack.
`void push(const T& el)`	Insert `el` at the top of the stack.
`size_type size() const`	Return the number of elements on the stack.
`stack()`	Create an empty stack.
`T& top()`	Return the top element on the stack.
`const T& top() const`	Return the top element on the stack.

4.5 Queues in the Standard Template Library

The queue container is implemented by default as the container `deque`, and the user may opt for using the container `list` instead. An attempt to use the container `vector` results in a compilation error because `pop()` is implemented as a call to `pop_front()`, which is assumed to be a member function of the underlying container, and `vector` does not include such a member function. For the list of

queue's member functions, see Figure 4.15. A short program in Figure 4.16 illustrates the operations of the member functions. Note that the dequeueing operation discussed in this chapter is implemented by `front()` followed by `pop()`, and the enqueueing operation is implemented with the function `push()`.

FIGURE 4.15 A list of `queue` member functions.

Member Function	Operation
`T& back()`	Return the last element in the queue.
`const T& back() const`	Return the last element in the queue.
`bool empty() const`	Return `true` if the queue includes no element and `false` otherwise.
`T& front()`	Return the first element in the queue.
`const T& front() const`	Return the first element in the queue.
`void pop()`	Remove the first element in the queue.
`void push(const T& el)`	Insert `el` at the end of the queue.
`queue()`	Create an empty queue.
`size_type size() const`	Return the number of elements in the queue.

FIGURE 4.16 An example application of `queue`'s member functions.

```
#include <iostream>
#include <queue>
#include <list>

using namespace std;

int main() {
    queue<int> q1;
    queue<int,list<int> > q2; //leave space between angle brackets > >
    q1.push(1); q1.push(2); q1.push(3);
    q2.push(4); q2.push(5); q2.push(6);
    q1.push(q2.back());
    while (!q1.empty()) {
        cout << q1.front() << ' ';    // 1 2 3 6
        q1.pop();
    }
    while (!q2.empty()) {
        cout << q2.front() << ' ';    // 4 5 6
        q2.pop();
    }
    return 0;
}
```

4.6 Priority Queues in the Standard Template Library

The `priority_queue` container (Figure 4.17) is implemented with the container `vector` by default, and the user may choose the container `deque`. The `priority_queue` container maintains an order in the queue by keeping an element with the highest priority in front of the queue. To accomplish this, a two-argument Boolean function is used by the insertion operation `push()`, which reorders the elements in the queue to satisfy this requirement. The function can be supplied by the user; otherwise, the operation < is used and the element with the highest value is considered to have the highest priority. If the highest priority is determined by the smallest value, then the function object `greater` needs to be used to indicate that `push()` should apply the operator > rather than < in making its decisions when inserting new elements to the priority queue. An example is shown in Figure 4.18. The priority queue `pq1` is defined as a vector-based queue that uses the operation < to determine the priority of integers in the queue. The second queue, `pq2`, uses the operation > during insertion. Finally, the queue `pq3` is of the same type as `pq1`, but it is also initialized with the numbers from the array `a`. The three `while` loops show in which order the elements from the three queues are dequeued.

FIGURE 4.17 A list of `priority_queue` member functions.

Member Function	Operation
`bool empty() const`	Return `true` if the queue includes no element and `false` otherwise.
`void pop()`	Remove an element in the queue with the highest priority.
`void push(const T& el)`	Insert `el` in a proper location on the priority queue.
`priority_queue(comp f())`	Create an empty priority queue that uses a two-argument Boolean function `f` to order elements on the queue.
`priority_queue(iterator first, iterator last, comp f())`	Create a priority queue that uses a two-argument Boolean function `f` to order elements on the queue; initialize the queue with elements from the range indicated by iterators `first` and `last`.
`size_type size() const`	Return the number of elements in the priority queue.
`T& top()`	Return the element in the priority queue with the highest priority.
`const T& top() const`	Return the element in the priority queue with the highest priority.

FIGURE 4.18 A program that uses member functions of the container `priority_queue`.

```
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

int main() {
    priority_queue<int> pq1; // plus vector<int> and less<int>
    priority_queue<int,vector<int>,greater<int> > pq2;
    pq1.push(3); pq1.push(1); pq1.push(2);
    pq2.push(3); pq2.push(1); pq2.push(2);
    int a[] = {4,6,5};
    priority_queue<int> pq3(a,a+3);
    while (!pq1.empty()) {
        cout << pq1.top() << ' ';    // 3 2 1
        pq1.pop();
    }
    while (!pq2.empty()) {
        cout << pq2.top() << ' ';    // 1 2 3
        pq2.pop();
    }
    while (!pq3.empty()) {
        cout << pq3.top() << ' ';    // 6 5 4
        pq3.pop();
    }
    return 0;
}
```

It is more interesting to see an application of the user-defined objects. Consider the class `Person` defined in Section 1.8:

```
class Person {
public:
    . . . . .
    bool operator<(const Person& p) const {
        return strcmp(name,p.name) < 0;
    }
    bool operator>(const Person& p) const {
        return !(*this == p) && !(*this < p);
    }
```

```
private:
    char *name;
    int age;
};
```

Our intention now is to create three priority queues. In the first two queues, the priority is determined by lexicographical order, but in `pqName1` it is the descending order and in `pqName2` the ascending order. To that end, `pqName1` uses the overloaded operator <. The queue `pqName2` uses the overloaded operator >, as made known by defining with the function object `greater<Person>`:

```
Person p[] = {Person("Gregg",25),Person("Ann",30),Person("Bill",20)};
priority_queue<Person> pqName1(p,p+3);
priority_queue<Person,vector<Person>,greater<Person> > pqName2(p,p+3);
```

In these two declarations, the two priority queues are also initialized with objects from the array `p`.

In Section 1.8, there is also a Boolean function `lesserAge` used to determine the order of `Person` objects by age, not by name. How can we create a priority queue in which the highest priority is determined by age? One way to accomplish this is to define a function object,

```
class lesserAge {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.age < p2.age;
    }
};
```

and then declare a new priority queue

```
priority_queue<Person,vector<Person>,lesserAge> pqAge(p,p+3);
```

initialized with the same objects as `pqName1` and `pqName2`. Printing elements from the three queues indicates the different priorities of the objects in different queues:

```
pqName1:   (Gregg,25) (Bill,20) (Ann,30)
pqName2:   (Ann,30) (Bill,20) (Gregg,25)
pqAge:     (Ann,30) (Gregg,25) (Bill,20)
```

4.7 Deques in the Standard Template Library

A *deque* (double-ended queue) is a list that allows for direct access to both ends of the list, particularly to insert and delete elements. Hence, a deque can be implemented as a doubly linked list with pointer data members `head` and `tail`, as discussed in Section 3.2. Moreover, as pointed out in Section 3.7, the container `list` uses a doubly linked list already. The STL, however, adds another functionality to the deque, namely, random access to any position of the deque, just as in arrays and vectors. Vectors, as discussed in Section 1.8, have poor performance for insertion and deletion of elements at the front, but these operations are quickly performed for doubly linked lists. This means that the STL deque should combine the behavior of a vector and a list.

The member functions of the STL container `deque` are listed in Figure 4.19. The functions are basically the same as those available for lists, with few exceptions. `deque` does not include function `splice()`, which is specific to `list`, and functions `merge()`, `remove()`, `sort()`, and `unique()`, which are also available as algorithms, and `list` only reimplements them as member functions. The most significant difference is the function `at()` (and its equivalent, `operator[]`), which is unavailable in `list`. The latter function is available in `vector`, and if we compare the set of member functions in `vector` (Figure 1.3) and in `deque`, we see only a few differences. `vector` does not have `pop_front()` and `push_front()`, as does `deque`, but `deque` does not include functions `capacity()` and `reserve()`, which are available in `vector`. A few operations are illustrated in Figure 4.20. Note that for lists only autoincrement and autodecrement were possible for iterators, but for deques we can add any number to iterators. For example, `dq1.begin()+1` is legal for deques, but not for lists.

FIGURE 4.19 A list of member functions in the class `deque`.

Member Function	Operation
`void assign(iterator first, iterator last)`	Remove all the elements in the deque and insert into it the elements from the range indicated by iterators `first` and `last`.
`void assign(size_type n, const T& el = T())`	Remove all the elements in the deque and insert into it `n` copies of `el`.
`T& at(size_type n)`	Return the element in position `n` of the deque.
`const T& at(size_type n) const`	Return the element in position `n` of the deque.
`T& back()`	Return the last element in the deque.
`const T& back() const`	Return the last element in the deque.
`iterator begin()`	Return an iterator that references the first element of the deque.
`const_iterator begin() const`	Return an iterator that references the first element of the deque.
`void clear()`	Remove all the elements in the deque.
`deque()`	Construct an empty deque.
`deque(size_type n, const T& el = T())`	Construct a deque with `n` copies of `el` of type `T` (if `el` is not provided, a default constructor `T()` is used).
`deque(const deque <T>& dq)`	Copy constructor.
`deque(iterator first, iterator last)`	Construct a deque and initialize it with values from the range indicated by iterators `first` and `last`.
`bool empty() const`	Return `true` if the deque includes no elements and `false` otherwise.
`iterator end()`	Return an iterator that is past the last element of the deque.

FIGURE 4.19 *(continued)*

`const_iterator end() const`	Return an iterator that is past the last element of the deque.
`iterator erase(iterator i)`	Remove the element referenced by iterator `i` and return an iterator referencing the element after the one removed.
`iterator erase(iterator first, iterator last)`	Remove the elements in the range indicated by iterators `first` and `last` and return an iterator referencing the element after the last one removed.
`T& front()`	Return the first element in the deque.
`const T& front() const`	Return the first element in the deque.
`iterator insert(iterator i, const T& el = T())`	Insert `el` before the element indicated by iterator `i` and return an iterator referencing the newly inserted element.
`void insert(iterator i, size_type n, const T& el)`	Insert `n` copies of `el` before the element referenced by iterator `i`.
`void insert(iterator i, iterator first, iterator last)`	Insert elements from the location referenced by `first` to the location referenced by `last` before the element referenced by iterator `i`.
`size_type max_size() const`	Return the maximum number of elements for the deque.
`T& operator[]`	Subscript operator.
`void pop_back()`	Remove the last element of the deque.
`void pop_front()`	Remove the first element of the deque.
`void push_back(const T& el)`	Insert `el` at the end of the deque.
`void push_front(const T& el)`	Insert `el` at the beginning of the deque.
`reverse_iterator rbegin()`	Return an iterator that references the last element of the deque.
`const_reverse_iterator rbegin() const`	Return an iterator that references the last element of the deque.
`reverse_iterator rend()`	Return an iterator that is before the first element of the deque.
`const_reverse_iterator rend() const`	Return an iterator that is before the first element of the deque.
`void resize(size_type n, const T& el = T())`	Make the deque have `n` positions by adding `n - size()` more positions with element `el` or by discarding overflowing `size() - n` positions from the end of the deque.
`size_type size() const`	Return the number of elements in the deque.
`void swap(deque<T>& dq)`	Swap the content of the deque with the content of another deque `dq`.

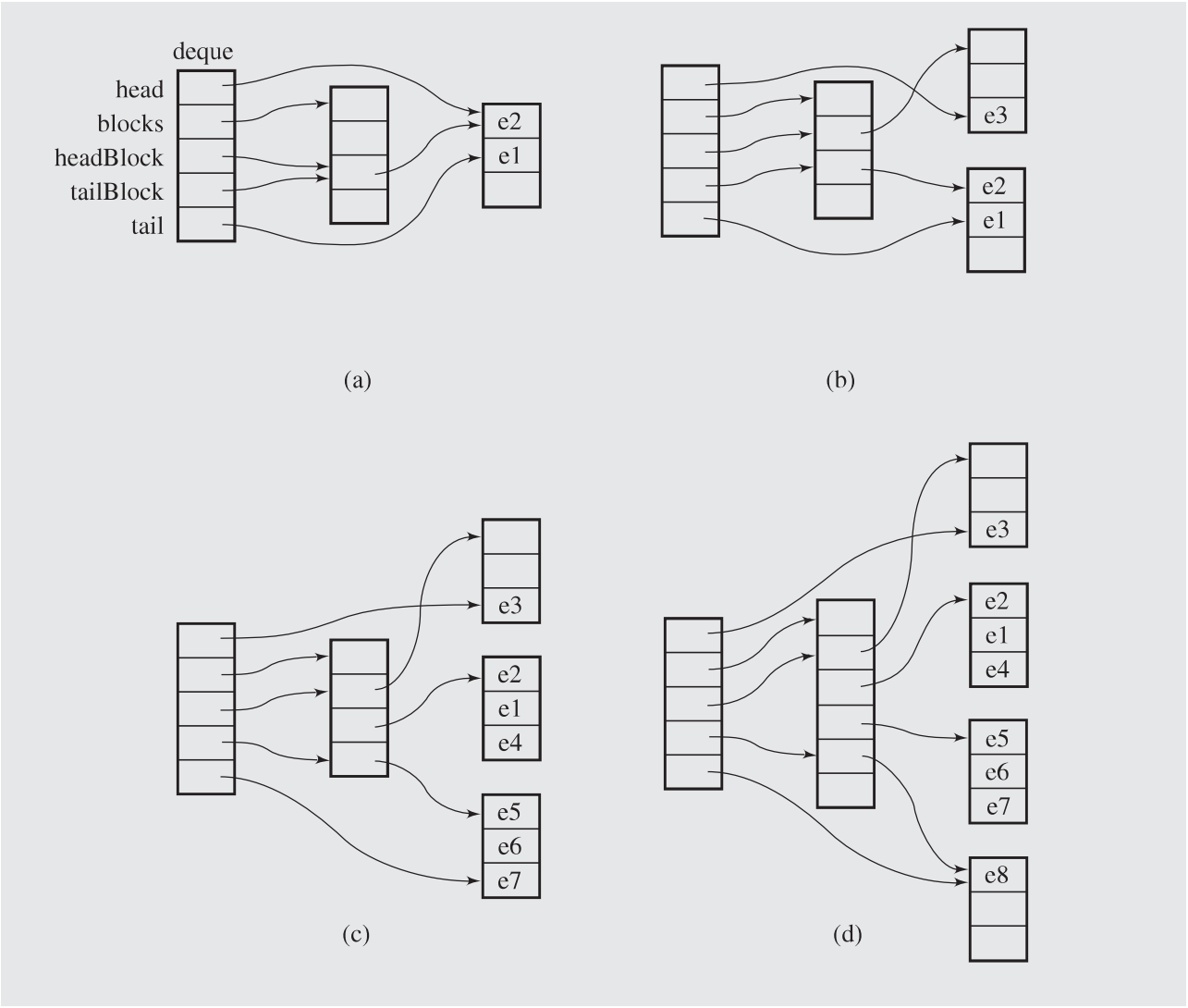
FIGURE 4.20 A program demonstrating the operation of `deque` member functions.

```
#include <iostream>
#include <algorithm>
#include <deque>

using namespace std;

int main() {
    deque<int> dq1;
    dq1.push_front(1);                          // dq1 = (1)
    dq1.push_front(2);                          // dq1 = (2 1)
    dq1.push_back(3);                           // dq1 = (2 1 3)
    dq1.push_back(4);                           // dq1 = (2 1 3 4)
    deque<int> dq2(dq1.begin()+1,dq1.end()-1); // dq2 = (1 3)
    dq1[1] = 5;                                 // dq1 = (2 5 3 4)
    dq1.erase(dq1.begin());                     // dq1 = (5 3 4)
    dq1.insert(dq1.end()-1,2,6);                // dq1 = (5 3 6 6 4)
    sort(dq1.begin(),dq1.end());                // dq1 = (3 4 5 6 6)
    deque<int> dq3;
    dq3.resize(dq1.size()+dq2.size());          // dq3 = (0 0 0 0 0 0 0)
    merge(dq1.begin(),dq1.end(),dq2.begin(),dq2.end(),dq3.begin());
    // dq1 = (3 4 5 6 6) and dq2 = (1 3) ==> dq3 = (1 3 3 4 5 6 6)
    return 0;
}
```

A very interesting aspect of the STL deque is its implementation. Random access can be simulated in doubly linked lists that have the definition of `operator[](int n)` which includes a loop that sequentially scans the list and stops at the *n*th node. The STL implementation solves this problem differently. An STL deque is not implemented as a linked list, but as an array of pointers to blocks or arrays of data. The number of blocks changes dynamically depending on storage needs, and the size of the array of pointers changes accordingly. (We encounter a similar approach applied in extendible hashing in Section 10.5.1.)

To discuss one possible implementation, assume that the array of pointers has four cells and an array of data has three cells; that is, `blockSize = 3`. An object `deque` includes the fields `head`, `tail`, `headBlock`, `tailBlock`, and `blocks`. After execution of `push_front(e1)` and `push_front(e2)` with an initially empty deque, the situation is as in Figure 4.21a. First, the array `blocks` is created, and then one data block is accessible from a middle cell of `blocks`. Next, `e1` is inserted in the middle of the data block. The subsequent calls place elements consecutively in the first half of the data array. The third call to `push_front()` cannot successfully place `e3` in the current data array; therefore, a new data array is created and `e3` is located in the last cell (Figure 4.21b). Now we execute `push_back()` four times. Element `e4` is placed

FIGURE 4.21 Changed on the deque in the process of pushing new elements.



in an existing data array accessible from `deque` through `tailBlock`. Elements `e5`, `e6`, and `e7` are placed in a new data block, which also becomes accessible through `tailBlock` (Figure 4.21c). The next call to `push_back()` affects the pointer array `blocks` because the last data block is full and the block is accessible for the last cell of `blocks`. In this case, a new pointer array is created that contains (in this implementation) twice as many cells as the number of data blocks. Next, the pointers from the old array `blocks` are copied to the new array, and then a new data block can be created to accommodate element `e8` being inserted (Figure 4.21d). This is an example of the worst case for which between n/`blockSize` and n/`blockSize` + 2 cells have to be copied from the old array to the new one; therefore, in the worst case, the pushing operation takes $O(n)$ time to perform. But assuming that `blockSize` is a large number, the worst case can be expected to occur very infrequently. Most of the time, the pushing operation requires constant time.

Inserting an element into a deque is very simple conceptually. To insert an element in the first half of the deque, the front element is pushed onto the deque, and all elements that should precede the new element are copied to the preceding cell. Then the new element can be placed in the desired position. To insert an element into the second half of the deque, the last element is pushed onto the deque, and elements that should follow the new element in the deque are copied to the next cell.

With the discussed implementation, a random access can be performed in constant time. For the situation illustrated in Figure 4.21—that is, with declarations

```
T **blocks;
T **headBlock;
T *head;
```

the subscript operator can be overloaded as follows:

```
T& operator[] (int n) {
    if (n < blockSize - (head - *headBlock))   // if n is
         return *(head + n);                   // in the first
    else {                                     // block;
         n = n - (blockSize - (head - *headBlock));
         int q = n / blockSize + 1;
         int r = n % blockSize;
         return *(*(headBlock + q) + r);
    }
}
```

Although access to a particular position requires several arithmetic, dereferencing, and assignment operations, the number of operations is constant for any size of the deque.

4.8 Case Study: Exiting a Maze

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze (Figure 4.22a). The mouse hopes to escape from the maze by systematically trying all the routes. If it reaches a dead end, it retraces its steps to the last position and begins at least one more untried path. For each position, the mouse can go in one of four directions: right, left, down, up. Regardless of how close it is to the exit, it always tries the open paths in this order, which may lead to some unnecessary detours. By retaining information that allows for resuming the search after a dead end is reached, the mouse uses a method called *backtracking*. This method is discussed further in the next chapter.

The maze is implemented as a two-dimensional character array in which passages are marked with 0s, walls by 1s, exit position by the letter e, and the initial position of the mouse by the letter m (Figure 4.22b). In this program, the maze problem is slightly generalized by allowing the exit to be in any position of the maze (picture the exit position as having an elevator that takes the mouse out of the trap) and allowing passages to be on the borderline. To protect itself from falling off the array by trying to continue its path when an open cell is reached on one of the borderlines, the mouse also has to constantly check whether it is in such a borderline position or not.