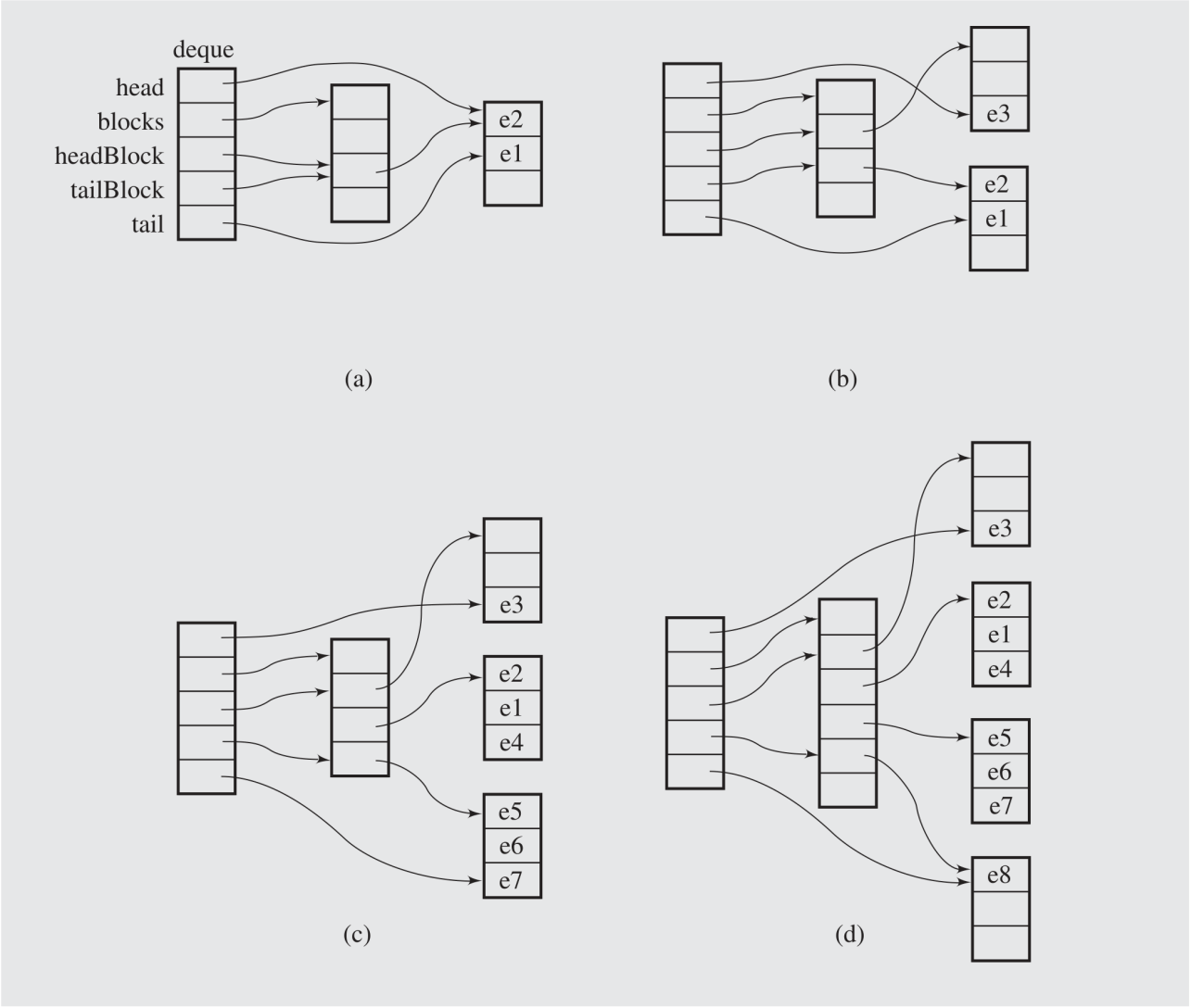
FIGURE 4.20 A program demonstrating the operation of `deque` member functions.

```
#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.