Remark

These notes are from the textbook Algorithms in C++ by Sedgewick. The important thing is the discussion of the elementary data structures. You may not understand all the code (C++) as it uses pointers and some C++ as opposed to C syntax.

3

Elementary Data Structures

In this chapter, we discuss basic ways of organizing data for processing by computer programs. For many applications, the choice of the proper data structure is really the only major decision involved in the implementation: once the choice has been made, only very simple algorithms are needed. For the same data, some data structures require more or less space than others; for the same operations on the data, some data structures lead to more or less efficient algorithms than others. This theme will recur frequently throughout this book, as the choice of algorithm and data structure is closely intertwined, and we continually seek ways of saving time or space by making the choice properly.

A data structure is not a passive object: we also must consider the operations to be performed on it (and the algorithms used for these operations). This concept is formalized in the notion of an *abstract data type*, which we discuss at the end of this chapter. But our primary interest is in concrete implementations, and we'll focus on specific representations and manipulations.

We're going to be dealing with arrays, linked lists, stacks, queues, and other simple variants. These are classical data structures with widespread applicability: along with trees (see Chapter 4), they form the basis for virtually all of the algorithms considered in this book. In this chapter, we consider basic representations and methods for manipulating these structures, work through some specific examples of their use, and discuss related issues such as storage management.

Arrays

Perhaps the most fundamental data structure is the *array*, which is defined as a primitive in C++ and most other programming languages. An array is a fixed number of data items that are stored contiguously and that are accessible by an index. We refer to the `i`th element of an array `a` as `a[i]`. It is the responsibility of the programmer to store something meaningful in an array position `a[i]` before referring to it; neglecting this is one of the most common programming mistakes.

A simple example of the use of an array is given by the following program, which prints out all the prime numbers less than 1000. The method used, which dates back to the 3rd century B.C., is called the "sieve of Eratosthenes":

```
const int N = 1000;
main()
  {
    int i, j, a[N+1];
    for (a[1] = 0, i = 2; i <= N; i++) a[i] = 1;
    for (i = 2; i <= N/2; i++)
      for (j = 2; j <= N/i; j++)
        a[i*j] = 0;
    for (i = 1; i <= N; i++)
      if (a[i]) cout << i << ' ';
    cout << '\n';
  }
```

This program uses an array consisting of the very simplest type of elements, boolean (0-1) values. The goal of the program is to set `a[i]` to `1` if `i` is prime, `0` if it is not. It does so by, for each `i`, setting the array element corresponding to each multiple of `i` to `0`, since any number that is a multiple of any other number cannot be prime. Then it goes through the array once more, printing out the primes. The array is first "initialized" to indicate that no numbers are known to be nonprime: the algorithm sets to `0` array elements corresponding to indices that are known to be nonprime. The program can be made somewhat more efficient by testing `a[i]` before the `for` loop involving `j`, since if `i` is not prime, the array elements corresponding to all of its multiples must already have been marked. It could be made more space-efficient by explicitly using an array of bits rather than integers.

The sieve of Eratosthenes is typical of algorithms that exploit the fact that any item of an array can be efficiently accessed. The algorithm also accesses the items of the array sequentially, one after the other. In many applications, sequential ordering is important; in other applications sequential ordering is used because it is as good as any other. But the primary feature of arrays is that *if the index is known*, any item can be accessed in constant time.

The size of the array must be known beforehand: to run the above program for a different value of `N`, it is necessary to change the constant `N`, then compile and execute. In some programming environments, it is possible to declare the size of an array at execution time (so that one could, for example, have a user type in the value of `N`, and then respond with the primes less than `N` without wasting memory by declaring an array as large as any value the user is allowed to type). In C++ it is possible to achieve this effect through proper use of the storage allocation mechanism, but it is still a fundamental property of arrays that their sizes are fixed and must be known before they are used.

Here, in case of confusion I will give the C version:

```
#include <stdio.h>
#define N 1000

void main()
{
     int a[N+1], i, j;  //declare array

     // assume all numbers are prime to begin with
     for(i=1; i<=N; ++i)
          a[i] = 1;

     // use the sieve to to filter out non primes
     for(i = 2; i <= N/2; ++i)
          for(j = 2; j <= N/i; ++j)
               a[i * j] = 0;

     int count = 0;
     printf("\nPrimes between 1 and %d are :", N);
     for(i=2; i<=N; ++i)
          if(a[i] == 1) {
               ++count;
               printf(" %d", i);
          }
     printf("\n\nThere are %d primes\n\n", count);

}
```

Remarks:

- The algorithm uses an array of size N+1 to find all prime numbers between 2 and N. This means it is impractical or impossible to use it for finding large prime numbers as it the array would be impossibly large. So the largest prime number it can find depends on the amount of RAM in the computer and the ability of the programming language to support very large arrays.

- The above program uses an int array which means 4 bytes per element. All that is required is a single bit (1 – prime, 0 – not a prime). So one improvement would be to use a char array. Then for the same amount of RAM the algorithm could search 4 times as many integers.

Arrays are fundamental data structures in that they have a direct correspondence with memory systems on virtually all computers. To retrieve the contents of a word from the memory in machine language, we provide an address. Thus, we could think of the entire computer memory as an array, with the memory addresses corresponding to array indices. Most computer language processors translate programs that involve arrays into rather efficient machine-language programs that access memory directly.

Another familiar way to structure information is to use a table of numbers organized into rows and columns. A table of students' grades in a course might have one row for each student, one column for each assignment. On a computer, such a table would be represented as a *two-dimensional* array with two indices, one for the row and one for the column. Various algorithms on such structures are straightforward: for example, to compute the average grade on an assignment, we sum together the elements in a column and divide by the number of rows; to compute a particular student's average grade in the course, we sum together the elements in a row and divide by the number of columns. Two-dimensional arrays are widely used in applications of this type. On a computer, it is often convenient and rather straightforward to use more than two dimensions: an instructor might use a third index to keep student grade tables for a sequence of years.

Arrays also correspond directly to *vectors*, the mathematical term for indexed lists of objects. Similarly, two-dimensional arrays correspond to *matrices*. We study algorithms for processing these mathematical objects in Chapters 36 and 37.

Linked Lists

The second elementary data structure to consider is the *linked list*, which is defined as a primitive in some programming languages (notably in Lisp) but not in C++. However, C++ does provide basic operations that make it easy to use linked lists.

The primary advantage of linked lists over arrays is that linked lists can grow and shrink in size during their lifetime. In particular, their maximum size need not be known in advance. In practical applications, this often makes it possible to have several data structures share the same space, without paying particular attention to their relative size at any time.

A second advantage of linked lists is that they provide flexibility in allowing the items to be rearranged efficiently. This flexibility is gained at the expense of quick access to any arbitrary item in the list. This will become more apparent below, after we have examined some of the basic properties of linked lists and some of the fundamental operations we perform on them.

A linked list is a set of items organized sequentially, just like an array. In an array, the sequential organization is provided implicitly (by the position in the array); in a linked list, we use an explicit arrangement in which each item is part of

Figure 3.1 A linked list.

a "node" that also contains a "link" to the next node. Figure 3.1 shows a linked list, with items represented by letters, nodes by circles and links by lines connecting the nodes. We look in detail below at how lists are represented within the computer; for now we'll talk simply in terms of nodes and links.

Even the simple representation of Figure 3.1 exposes two details we must consider. First, every node has a link, so the link in the last node of the list must specify some "next" node. Our convention will be to have a "dummy" node, which we'll call `z`, for this purpose: the last node of the list will point to `z`, and `z` will point to itself. In addition, we normally will have a dummy node at the other end of the list, again by convention. This node, which we'll call `head`, will point to the first node in the list. The main purpose of the dummy nodes is to make certain manipulations with the links, especially those involving the first and last nodes on the list, more convenient. Other conventions are discussed below. Figure 3.2 shows the list structure with these dummy nodes included.

Now, this explicit representation of the ordering allows certain operations to be performed much more efficiently than would be possible for arrays. For example, suppose that we want to move the T from the end of the list to the beginning. In an array, we would have to move every item to make room for the new item at the beginning; in a linked list, we just change three links, as shown in Figure 3.3. The two versions shown in Figure 3.3 are equivalent; they're just drawn differently. We make the node containing T point to A, the node containing S point to `z`, and `head` point to T. Even if the list was very long, we could make this structural change by changing just three links.

More important, we can talk of "inserting" an item into a linked list (which makes it grow by one in length), an operation that is unnatural and inconvenient in an array. Figure 3.4 shows how to insert X into our example list by putting X in a node that points to S, then making the node containing I point to the new node. Only two links need to be changed for this operation, no matter how long the list.

Similarly, we can speak of "deleting" an item from a linked list (which makes it shrink by one in length). For example, the third list in Figure 3.4 shows how to delete X from the second list simply by making the node containing I point to

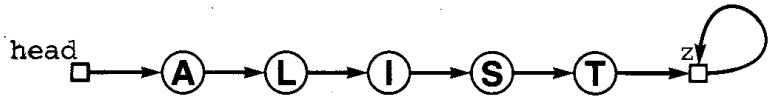

Figure 3.2 A linked list with its dummy nodes.

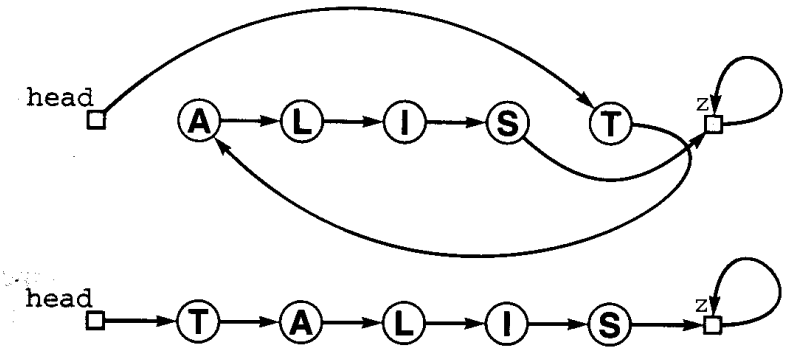

Figure 3.3 Rearranging a linked list.

S, skipping X. Now, the node containing X still exists (in fact it still points to S), and perhaps should be disposed of in some way—the point is that X is no longer part of this list, and cannot be accessed by following links from `head`. We will return to this issue below.

On the other hand, there are other operations for which linked lists are *not* well-suited. The most obvious of these is "find the kth item" (find an item given its index): in an array this is done simply by accessing `a[k]`, but in a list we have to travel through k links. Another operation that is unnatural on linked lists is "find the item *before* a given item." If all we have is the link to T in our sample list, then the only way we can find the link to S is to start at `head` and travel

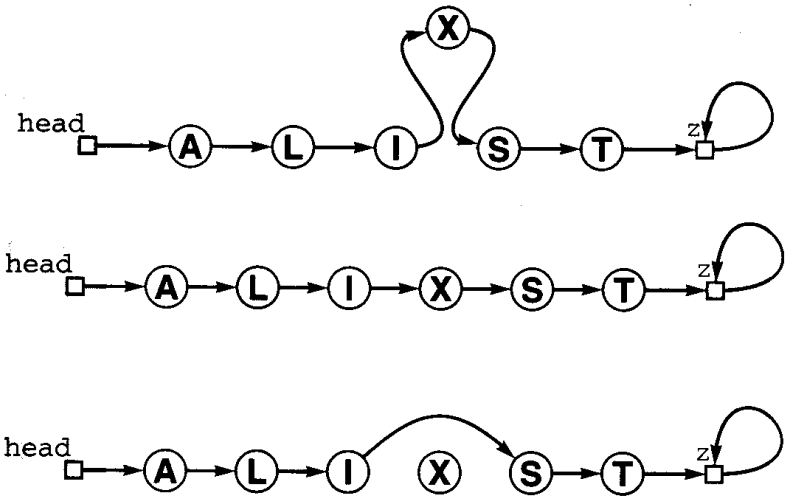

Figure 3.4 Insertion into and deletion from a linked list.

through the list to find the node that points to T. As a matter of fact, this operation is necessary if we want to be able to delete a given node from a linked list: how else do we find the node whose link must be changed? In many applications, we can get around this problem by redesigning the deletion operation to be "delete the *next* node". A similar problem can be avoided for insertion by making the insertion operation "insert a given item *after* a given node" in the list.

To illustrate how basic linked list primitives might be implemented in C++, we begin by precisely specifying the format of the list nodes and building an empty list, as follows:

```
struct node
  { int key; struct node *next; };
struct node *head, *z;
head = new node; z = new node;
head->next = z; z->next = z;
```

The `struct` declaration specifies that lists are made up of `node`s, each node containing an integer and a pointer to the `next` node on the list. The `key` is an integer here only for simplicity, and could be any type—the `next` pointer is the key to the list. The asterisks indicate that the variables `head` and `z` are declared to be pointers to nodes. Nodes are actually created only when the built-in function `new` is called. This hides a complex mechanism that is intended to relieve the programmer of the burden of "allocating" storage for the nodes as the list grows. We discuss this mechanism in some detail below. The "arrow" (minus sign followed by a greater-than sign) notation is used in C++ to follow pointers through structures. We write a reference to a link followed by this symbol to indicate a reference to the node pointed to by that link. Thus, the above code creates two new nodes referred to by `head` and `z` and sets both of them to point to `z`.

To insert a new node with key value `v` into a linked list at a point following a given node `t`, we create the node (`x = new node`) and put in the key value (`x->key = v`), then copy `t`'s link into it (`x->next = t->next`) and make `t`'s link point to the new node (`t->next = x`).

To extract the node following a given node `t` from a linked list, we get a pointer to that node (`x = t->next`), copy its pointer into `t` to take it out of the list (`t->next = x->next`), and return it to the storage allocation system using the built-in procedure `delete`, unless the list was empty (`if (x!=z) delete x`).

The reader is encouraged to check these C++ implementations against Figure 3.4. Note that the `head` node makes it possible to avoid having a special test for insertion at the beginning of the list, and the `z` node provides a convenient way to test for an empty list. We will see another use for `z` in Chapter 14.

We'll see many examples of applications of these and other basic operations on linked lists in later chapters. Since the operations involve only a few statements, we

Josephus Problem

often manipulate the lists directly rather than through data types. As an example, we consider next a program for solving the so-called "Josephus problem" in the spirit of the sieve of Eratosthenes. We imagine that N people have decided to commit mass suicide by arranging themselves in a circle and killing the Mth person around the circle, closing ranks as each person drops out of the circle. The problem is to find out which person is the last to die (though perhaps that person would have a change of heart at the end!), or, more generally, to find the order in which the people are executed. For example, if $N = 9$ and $M = 5$, the people are killed in the order 5 1 7 4 3 6 9 2 8. The following program reads in N and M and prints out this ordering:

```
struct node
  { int key; struct node *next; };
main()
  {
    int i, N, M;
    struct node *t, *x;
    cin >> N >> M;
    t = new node; t->key = 1; x = t;
    for (i = 2; i <= N; i++)
      {
        t->next = new node;
        t = t->next; t->key = i;
      }
    t->next = x;
    while (t != t->next)
      {
        for (i = 1; i < M; i++) t = t->next;
        cout << t->next->key << ' ';
        x = t->next; t->next = x->next;
        delete x;
      }
    cout << t->key << '\n';
  }
```

The program uses a "circular" linked list to simulate the sequence of executions directly. First, the list is built with keys from 1 to N: the variable `x` holds onto the beginning of the list as it is built, then the pointer in the last node in the list is set to `x`. Then, the program proceeds through the list, counting through $M - 1$ items and deleting the next, until only one is left (which then points to itself). Note the call to `delete` for the delete, which corresponds to an execution: this is the opposite of `new` as mentioned above.

At this point Sedgewick just presented a solution to the Josephus problem using a circular linked list. Here we will look at a version that uses an aray to simulate a linked list which in turn simulates the people commiting suicide.

```
#include <stdio.h>
#define Nmax 100

void main()
{
    int N, M;
    int curr, prev, i;
    int a[Nmax+1];

    printf("Input values for N and M for the Josephus problem: ");
    scanf("%d %d", &N, &M);

    // form a circle with each array element 'pointing' to the next
    for(i=1; i<N; ++i)
        a[i] = i+1;
    a[N] = 1;

    printf("\nOrder of death is: ");
    prev = N;
    curr = 1;

    while(a[curr] != curr) {  // starting from first node, follow
        for(i=1; i<M; ++i) { // M-1 links to get to the M th node.
            prev = curr;     // Moves M-1 steps thru the circular list
            curr = a[curr];
        }
        printf("%d  ", curr);     // kill person number M
        a[prev] = a[curr];        // i.e. remove from circle
        curr = a[curr];    }

    printf("\n\nLast one left is number %d\n\n", curr);

}
```


Circular lists are sometimes used as an alternative to having the the dummy nodes `head` or `z`, with one dummy node to mark the beginning (and the end) of the list and to help handle the case of an empty list.

It is possible to support the operation "find the item *before* a given item" by using a *doubly linked list* in which we maintain two links for each node, one to the item before, one to the item after. The cost of providing this extra capability is doubling the number of link manipulations per basic operation, so it is not normally used unless specifically called for. As mentioned above, however, if a node is to be deleted and only a link to the node is available (perhaps it is also part of some other data structure), double linking may be called for.

Pushdown Stacks

We have been concentrating on structuring data in order to insert, delete, or access items arbitrarily. Actually, it turns out that for many applications, it suffices to consider various (rather stringent) restrictions on how the data structure is accessed. Such restrictions are beneficial in two ways: first, they can alleviate the need for the program using the data structure to be concerned with its details (for example, keeping track of links to or indices of items); second, they allow simpler and more flexible implementations, since fewer operations need be supported.

The most important restricted-access data structure is the *pushdown stack.* Only two basic operations are involved: one can `push` an item onto the stack (insert it at the beginning) and `pop` an item (remove it from the beginning). A stack operates somewhat like a busy executive's "in" box: work piles up in a stack, and whenever the executive is ready to do some work, he takes it off the top. This might mean that something gets stuck in the bottom of the stack for some time, but a good executive would presumably manage to get the stack emptied periodically. It turns out that sometimes a computer program is naturally organized in this way, postponing some tasks while doing others, and thus pushdown stacks appear as the fundamental data structure for many algorithms.

Figure 3.7 shows how a sample stack evolves through the series of `push` and `pop` operations represented by the sequence:

$$A\ *\ S\ A\ *\ M\ *\ P\ *\ L\ *\ E\ S\ *\ T\ *\ *\ *\ A\ *\ C\ K\ *\ *.$$

Each letter in this list means "push" (the letter); each asterisk means "pop".

C++ provides excellent support for defining and using fundamental data structures like pushdown stacks. The following implementation of the basic stack operations is prototypical of many implementations that we will see later in the book. The stack is represented with an array `stack` and pointer `p` to the top of the stack—the functions `push`, `pop`, and `empty` are straightforward implementations of the basic stack operations. This code does not check whether the user is trying to push onto a full stack or pop from an empty one, though the function `empty` is a way to check the latter.

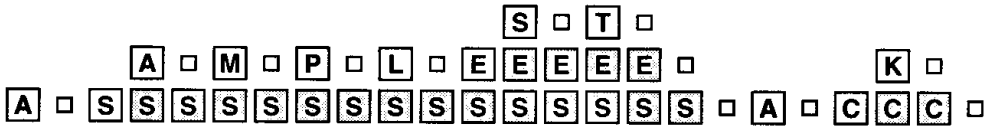

Figure 3.7 Dynamic characteristics of a stack.

```
class Stack
  {
    private:
      itemType *stack;
      int p;
    public:
      Stack(int max=100)
        { stack = new itemType[max]; p = 0; }
      ~Stack()
        { delete stack; }
      inline void push(itemType v)
        { stack[p++] = v; }
      inline itemType pop()
        { return stack[--p]; }
      inline int empty()
        { return !p; }
  };
```

The type of the items held on the stack is left unspecified with the name `itemType` to give some flexibility. For example, the statement `typedef int itemType;` could be used to have the stack hold integers. A C++ alternative to this is discussed at the end of this chapter.

The above code exhibits several C++ constructs, which we now consider. The implementation is a `class`, a C++ "user defined type". With this definition, a `Stack` has the same status as an `int`, `char` or any of the other builtin types. The implementation is divided into two parts, a `private` part that specifies how the data is organized and a `public` part that defines the operations allowed on the data. Programs that use stacks need only reference the public part and need not be concerned with how the stacks are implemented. The function `Stack` is a "constructor" that creates and initializes the stack when it is first defined, and the function `~Stack` is a "destructor" that deletes the stack when it is no longer needed. The keyword `inline` means that function call overhead is avoided, which is appropriate for short functions such as these.

We'll see a great many applications of stacks in the chapters that follow: for an introductory example, let's look at using stacks in evaluating arithmetic expressions. Suppose that one wants to find the value of a simple arithmetic expression involving multiplication and addition of integers, such as

$$5 * (((9 + 8) * (4 * 6)) + 7).$$

The calculation involves saving away some intermediate results: for example, if one calculates 9+8 first, then the 17 must be saved away somewhere while, say, 4*6

is computed. It turns out that a stack is the ideal mechanism for saving intermediate results in such a calculation.

To begin, we systematically rearrange the expression so that each operator appears *after* its two arguments, not between. The example above corresponds to the expression

5 9 8 + 4 6 * * 7 + *.

This is called *reverse Polish* notation (because it was introduced by a Polish logician), or *postfix*. The customary way of writing arithmetic expressions is called *infix*. One interesting property of postfix is that parentheses are not required; in infix they are needed to distinguish, for example, 5*(((9+8)*(4*6))+7) from ((5*9)+8)*((4*6)+7). A more interesting property of postfix is that it provides a simple way to perform the calculation, saving intermediate results on a stack. The following program reads a postfix expression, interpreting each operand as a command to "push the operand onto the stack" and each operator as a command to "pop the two operands from the stack, perform the operation, and push the result."

```
char c; Stack acc(50); int x;
while (cin.get(c))
  {
    x = 0;
    while (c == ' ') cin.get(c);
    if (c == '+') x = acc.pop() + acc.pop();
    if (c == '*') x = acc.pop() * acc.pop();
    while (c>='0' && c<='9')
      { x = 10*x + (c-'0'); cin.get(c); }
    acc.push(x);
  }
cout << acc.pop() << '\n';
```

The pushdown stack `save` is declared and defined along with other program variables, and the `push` and `pop` operations for `save` are invoked precisely in the same way as `get` for the input stream `cin`. This program reads any postfix expression involving multiplication and addition of integers, then prints the value of the expression. Blanks are ignored, and the `while` loop converts integers from character format to numbers for calculation. Otherwise, the operation of the program is straightforward. In C++, the order in which the two `pop()` operations is performed is unspecified, so slightly more complicated code would be needed for noncommutative operators such as subtract and divide.

The following program converts a legal fully parenthesized infix expression into a postfix expression:

```
char c; Stack save(50);
while (cin.get(c))
  {
    if (c == ')') cout.put(save.pop());
    if (c == '+') save.push(c);
    if (c == '*') save.push(c);
    while (c>='0' && c<='9')
      { cout.put(c); cin.get(c); }
    if (c != '(') cout << ' ';
  }
cout << '\n';
```

Operators are pushed on the stack, and arguments are simply passed through. Thus, arguments appear in the postfix expression in the same order as in the infix expression. Then each right parenthesis indicates that both arguments for the last operator have been output, so the operator itself can be popped and output. It is amusing to note that, since we use only operators with exactly two operands, the left parentheses are not needed in the infix expression (and this program skips them). For simplicity, this program does not check for errors in the input and requires spaces between operators, parentheses and operands.

The "save intermediate results" paradigm is fundamental, and pushdown stacks arise frequently. Many machines implement basic stack operations in hardware because they naturally implement function call mechanisms: save the current environment on entry to a procedure by pushing information onto a stack; resore the environment on exit by using information popped from the stack. Some calculators and some computing languages base their method of calculation on stack operations explicitly: every operation pops its arguments from the stack and returns its results to the stack. As we'll see in Chapter 5, stacks often arise implicitly even when not used explicitly.

Linked List Implementation of Stacks

Figure 3.7 illustrates the typical case where a small stack suffices even when there are a large number of stack operations. If one is confident this is the case, then the array representation given above is appropriate. Otherwise, a linked list can allow the stack to grow and shrink gracefully, which is especially useful if it is one of many such data structures. To implement the basic stack operations using linked lists, we begin by defining the interface:

```
class Stack
  {
    public:
      Stack(int max);
     ~Stack();
      void push(itemType v);
      itemType pop();
      int empty();
    private:
      struct node
       { itemType key; struct node *next; };
      struct node *head, *z;
  };
```

In C++, such an interface serves two purposes: to *use* a stack, one need only consult the `public` section of the interface to learn what operations are supported; and to *implement* a stack routine, one consults the `private` section to learn the basic data structures associated with the implementation. The implementations of the stack procedures are separated from the class declaration, even included in a separate file. This ability to separate implementations from interfaces and therefore easily experiment with different implementations is a very important feature of C++ that we will discuss in more detail at the end of this chapter.

Next we need the "constructor" function to create the stack when it is declared and the "destructor" function to delete it when it is no longer needed (goes out of scope):

```
Stack::Stack(int max)
  {
    head = new node; z = new node;
    head->next = z;  z->next = z;
  }
Stack::~Stack()
  {
    struct node *t = head;
    while (t != z)
      { head = t; t = t->next; delete head; }
  }
```

Finally, we have the actual implementation of the pushdown stack operations:

```
void Stack::push(itemType v)
  {
    struct node *t = new node;
    t->key = v; t->next = head->next;
    head->next = t;
  }
itemType Stack::pop()
  {
    itemType x;
    struct node *t = head->next;
    head->next = t->next; x = t->key;
    delete t; return x;
  }
int Stack::empty()
  { return head->next == z; }
```

The reader is advised to study this code carefully to reinforce understanding of both linked lists and pushdown stacks.

Queues

Another fundamental restricted-access data structure is called the *queue*. Again, only two basic operations are involved: one can `insert` an item into the queue at the beginning and `remove` an item from the end. Perhaps our busy executive's "in" box *should* operate like a queue, since then work that arrives first would get done first. In a stack, something can get buried at the bottom, but in a queue everything is processed in the order received.

Figure 3.8 shows how a sample queue evolves through the series of `get` and `put` operations represented by the sequence

$$A * S\ A * M * P * L\ E * Q * * * U * E\ U * * E * .$$

where each letter in this list means "put" (the letter) and the asterisk means "get".

Although stacks are encountered more often than queues because of their fundamental relationship with recursion (see Chapter 5), we will encounter algorithms for which the queue is the natural data structure. In Chapter 20 we encounter a `deque` (or "double-ended queue"), which is a combination of a stack and a queue, and in Chapters 4 and 30 we discuss rather fundamental examples involving the application of a queue as a mechanism to allow exploration of trees and graphs. Stacks are sometimes referred to as obeying a "last in, first out" (LIFO) discipline; queues obey a "first in, first out" (FIFO) discipline.

The linked-list implementation of the queue operations is straightforward and left as an exercise for the reader. As with stacks, an array can also be used if one

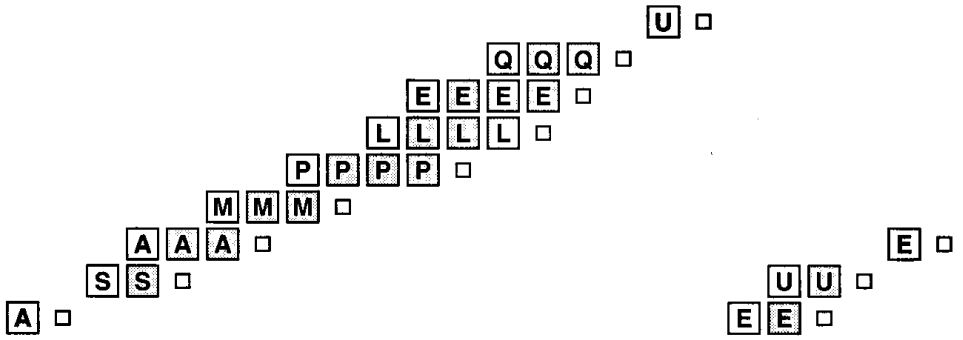

Figure 3.8 Dynamic characteristics of a queue.

can estimate the maximum size, as in the following implementation of the `put`, `get`, and `empty` functions (we omit the code for the interface and the construct and destruct functions here because they are so similar to the code given above for the array implementation of pushdown stacks):

```
void Queue::put(itemType v)
  {
    queue[tail++] = v;
    if (tail > size) tail = 0;
  }
itemType Queue::get()
  {
    itemType t = queue[head++];
    if (head > size) head = 0;
    return t;
  }
int Queue::empty()
  { return head == tail; }
```

There are three class variables: the `size` of the queue and two indices, one to the beginning of the queue (`head`) and one to the end (`tail`). The contents of the queue are all the elements in the array between `head` and `tail`, taking into account the "wraparound" back to 0 when the end of the array is encountered. If `head` and `tail` are equal, then the queue is defined to be empty; but if `put` would make them equal, then it is defined to be full (though, again, we do not include this check in the code above). This requires that the size of the array be one greater than the maximum number of elements one wishes to fit in the queue: a full queue contains one empty array position.

Abstract and Concrete Data Types

We've seen above that it is often convenient to describe algorithms and data structures in terms of the operations performed, rather than in terms of details of implementation. When a data structure is defined in this way, it is called an *abstract data type*. The idea is to separate the "concept" of what the data structure should do from any particular implementation.

The defining characteristic of an abstract data type is that nothing outside of the definitions of the data structure and the algorithms operating on it should refer to anything inside, except through function and procedure calls for the fundamental operations. The main motivation for the development of abstract data types has been as a mechanism for organizing large programs. Abstract data types provide a way to limit the size and complexity of the interface between (potentially complicated) algorithms and associated data structures and (a potentially large number of) programs that use the algorithms and data structures. This makes it easier to understand the large program, and more convenient to change or improve the fundamental algorithms.

Stacks and queues are classic examples of abstract data types: most programs need be concerned only about a few well-defined basic operations, not details of links and indices.

Arrays and linked lists can in turn be thought of as refinements of a basic abstract data type called the *linear list*. Each of them can support operations such as *insert*, *delete*, and *access* on a basic underlying structure of sequentially ordered items. These operations suffice to describe the algorithms, and the linear list abstraction can be useful in the initial stages of algorithm development. But as we've seen, it is in the programmer's interest to define carefully which operations will be used, for the different implementations can have quite different performance characteristics. For example, using a linked list instead of an array for the sieve of Eratosthenes would be costly because the algorithm's efficiency depends on being able to get from any array position to any other quickly, and using an array instead of a linked list for the Josephus problem would be costly because the algorithm's efficiency depends on the disappearance of deleted elements.

Many more operations suggest themselves on linear lists that require much more sophisticated algorithms and data structures to support efficiently. The two most important are *sorting* the items in increasing order of their keys (the subject of Chapters 8–13), and *searching* for an item with a particular key (the subject of Chapters 14–18).

One abstract data type can be used to define another: we use linked lists and arrays to define stacks and queues. Indeed, we use the "pointer" and "record" abstractions provided by C++ to build linked lists, and the "array" abstraction provided by C++ to build arrays. In addition, we saw above that we can build linked lists with arrays, and we'll see in Chapter 36 that arrays should sometimes be built with linked lists! The real power of the abstract data type concept is that it

allows us conveniently to construct large systems on different levels of abstraction, from the machine-language instructions provided by the computer, to the various capabilities provided by the programming language, to sorting, searching and other higher-level capabilities provided by algorithms as discussed in this book, to the even higher levels of abstraction that the application may suggest.

In this book, we deal with relatively small programs that are rather tightly integrated with their associated data structures. While it is possible to talk of abstraction at the interface between our algorithms and their data structures, it is really more appropriate to focus on higher levels of abstraction (closer to the application): the concept of abstraction should not distract us from finding the most efficient solution to a particular problem. We take the view here that performance does matter! Programs developed with this in mind can then be used with some confidence in developing higher levels of abstraction for large systems.

Our implementations of the stack and queue operations in this chapter are examples of *concrete data types*, which package together data structures and the algorithms that operate on them. We will use this paradigm frequently throughout this book, for it is a very convenient way to describe basic algorithms while at the same time developing broadly useful code for use in applications. C++ does provide a way, using the "class hierarchy" and "virtual functions", to implement true abstract data types, where an interface consisting only of the functions (*not* the data representation) is provided—we refrain from using this because our focus is on an awareness of performance characteristics, which is difficult to maintain when using true abstract data types.

As mentioned above, real data structures rarely consist simply of integers and links. Nodes often contain a great deal of information and may belong to multiple independent data structures. For example, a file of personnel data may contain records with names, addresses, and various other pieces of information about employees, and each record may need to belong to one data structure for searching for particular employees, and to another data structure for answering statistical queries, etc. C++ has a general mechanism, called the *template* facility, that provides an easy way to extend simple algorithms to work on complex structures. Rather than using `typedef`, the code `template <class itemType>` placed just before the `class` definitions in this chapter turn them into definitions of structures that work with any type. For example, this would allow a declaration like `Stack<float> acc()` to be used to build a stack of `floats`. In this book, we usually will be working with integers, but will keep the types unspecified as in this chapter with the understanding that `typedefs` or `templates` can easily be used in applications as appropriate.