Fig. 6.2 Queue Waiting for a Bus

6.2 STACKS

A *stack* is a list of elements in which an element may be inserted or deleted only at one end, called the *top* of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Special terminology is used for two basic operations associated with stacks:

(a) "Push" is the term used to insert an element into a stack.
(b) "Pop" is the term used to delete an element from a stack.

We emphasize that these terms are used only with stacks, not with other data structures.

6.1 INTRODUCTION

The linear lists and linear arrays discussed in the previous chapters allowed one to insert and delete elements at any place in the list—at the beginning, at the end, or in the middle. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of the data structures that are useful in such

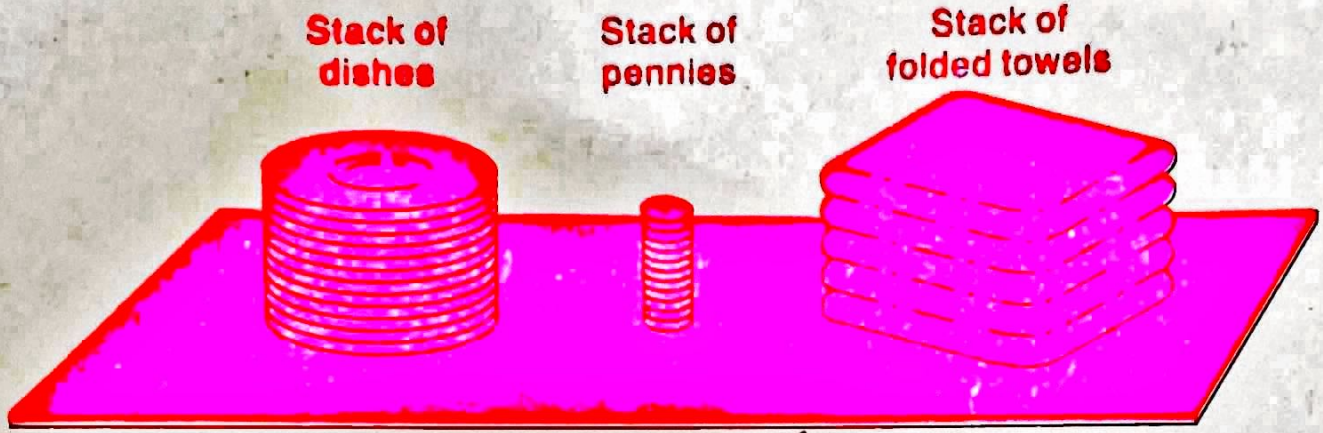


Fig. 6.1

Postponed Decisions — Real Life Example

Stacks are frequently used to indicate the order of the processing of data when certain steps of the processing must be postponed until other conditions are fulfilled. This is illustrated as follows.

Suppose that while processing some project A we are required to move on to project B, whose completion is required in order to complete project A. Then we place the folder containing the data of A onto a stack, as pictured in Fig. 6.4(a), and begin to process B. However, suppose that while processing B we are led to project C, for the same reason. Then we place B on the stack above A, as pictured in Fig. 6.4(b), and begin to process C. Furthermore, suppose that while processing C we are likewise led to project D. Then we place C on the stack above B, as pictured in Fig. 6.4(c), and begin to process D.

On the other hand, suppose we are able to complete the processing of project D. Then the only project we may continue to process is project C, which is on top of the stack. Hence we remove folder C from the stack, leaving the stack as pictured in Fig. 6.4(d), and continue to process C.

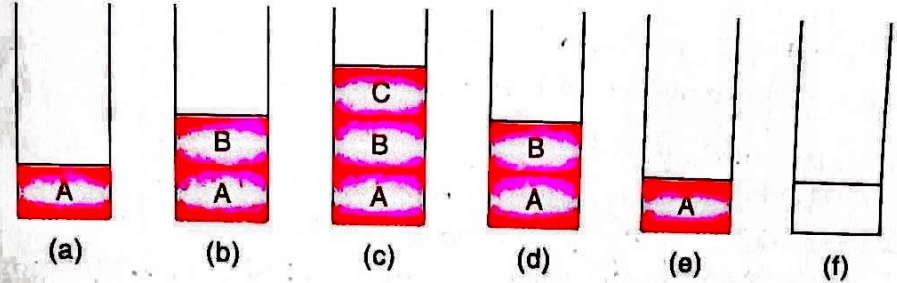


Fig. 6.4

Similarly, after completing the processing of C, we remove folder B from the stack, leaving the stack as pictured in Fig. 6.4(e), and continue to process B. Finally, after completing the processing of B, we remove the last folder, A, from the stack, leaving the empty stack pictured in Fig. 6.4(f), and continue the processing of our original project A.

Observe that, at each stage of the above processing, the stack automatically maintains the order that is required to complete the processing. An important example of such a processing in computer science is where A is a main program and B, C and D are subprograms called in the order given.

We use the term *container* to denote a data structure that permits storage and retrieval of data items *independent of content*. By contrast, dictionaries are abstract data types that retrieve based on key values or content, and will be discussed in Section 3.3 (page 72).

Containers are distinguished by the particular retrieval order they support. In the two most important types of containers, this retrieval order depends on the insertion order:

- *Stacks* – Support retrieval by last-in, first-out (LIFO) order. Stacks are simple to implement and very efficient. For this reason, stacks are probably the right container to use when retrieval order doesn't matter at all, such as when processing batch jobs. The *put* and *get* operations for stacks are usually called *push* and *pop*:

- *Push(x,s)*: Insert item x at the top of stack s.
- *Pop(s)*: Return (and remove) the top item of stack s.

LIFO order arises in many real-world contexts. People crammed into a subway car exit in LIFO order. Food inserted into my refrigerator usually exits the same way, despite the incentive of expiration dates. Algorithmically, LIFO tends to happen in the course of executing recursive algorithms.

Questions on stacks and queues will be much easier to handle if you are comfortable with the ins and outs of the data structure. The problems can be quite tricky, though. While some problems may be slight modifications on the original data structure, others have much more complex challenges.

Implementing a Stack

The stack data structure is precisely what it sounds like: a stack of data. In certain types of problems, it can be favorable to store data in a stack rather than in an array.

A stack uses LIFO (last-in first-out) ordering. That is, as in a stack of dinner plates, the most recent item added to the stack is the first item to be removed.

It uses the following operations:

- `pop()`: Remove the top item from the stack.
- `push(item)`: Add an item to the top of the stack.
- `peek()`: Return the top of the stack.
- `isEmpty()`: Return true if and only if the stack is empty.

Unlike an array, a stack does not offer constant-time access to the ith item. However, it does allow constant-time adds and removes, as it doesn't require shifting elements around.

One case where stacks are often useful is in certain recursive algorithms. Sometimes you need to push temporary data onto a stack as you recurse, but then remove them as you backtrack (for example, because the recursive check failed). A stack offers an intuitive way to do this.

A stack can also be used to implement a recursive algorithm iteratively. (This is a good exercise! Take a simple recursive algorithm and implement it iteratively.)

STACKS

6.1 Consider the following stack of characters, where STACK is allocated N = 8 memory cells:

STACK: A, C, D, F, K, ___, ___, ___,

(For notational convenience, we use "___" to denote an empty memory cell.) Describe the stack as the following operations take place:

(a) POP(STACK, ITEM)
(b) POP(STACK, ITEM)
(c) PUSH(STACK, L)
(d) PUSH(STACK, P)
(e) POP(STACK, ITEM)
(f) PUSH(STACK, R)
(g) PUSH(STACK, S)
(h) POP(STACK, ITEM)

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly:

(a) STACK: A, C, D, F, ___, ___, ___, ___
(b) STACK: A, C, D, ___, ___, ___, ___, ___
(c) STACK: A, C, D, L, ___, ___, ___, ___
(d) STACK: A, C, D, L, P, ___, ___, ___
(e) STACK: A, C, D, L, ___, ___, ___, ___
(f) STACK: A, C, D, L, R, ___, ___, ___
(g) STACK: A, C, D, L, R, S, ___, ___
(h) STACK: A, C, D, L, R, ___, ___, ___

6.9 IMPLEMENTATION OF RECURSIVE PROCEDURES BY STACKS

The preceding sections showed how recursion may be a useful tool in developing algorithms for specific problems. This section shows how stacks may be used to implement recursive procedures. It is instructive to first discuss subprograms in general.

Recall that a subprogram can contain both parameters and local variables. The parameters are the variables which receive values from objects in the calling program, called arguments, and which transmit values back to the calling program. Besides the parameters and local variables, the subprogram must also keep track of the return address in the calling program. This return address is essential, since control must be transferred back to its proper place in the calling program. At the time that the subprogram is finished executing and control is transferred back to the calling program, the values of the local variables and the return address are no longer needed.

Suppose our subprogram is a recursive program. Then each level of execution of the subprogram may contain different values for the parameters and local variables and for the return address. Furthermore, if the recursive program does call itself, then these current values must be saved, since they will be used again when the program is reactivated.

Suppose a programmer is using a high-level language which admits recursion, such as Pascal. Then the computer handles the bookkeeping that keeps track of all the values of the parameters, local variables and return addresses. On the other hand, if a programmer is using a high-level language which does not admit recursion, such as FORTRAN, then the programmer must set up the necessary bookkeeping by translating the recursive procedure into a nonrecursive one. This bookkeeping is discussed below.

A recursive approach to solving a problem can only be applied if ther underlying programming language supports recursion. But, what happens when the programming language (e.g. FORTRAN and COBOL) does not support recursion? For such cases, we need to simulate recursion using iterative means.

Apart from the lack of support to recursion by a programming language, there could be other reasons for adopting the simulation approach. The execution of recursive procedures in a programming language has associated time and space overheads. A dedicated control stack is required to store the reurn addresses and data sets of the recursively called procedures. To develop high-performance programs, it may be required to avoid such expenses even though the underlying language may support recursion. This is achieved by way of simulating recursion. The following recursion-specific elements are simulated and managed using iterative means:

- Function arguments
- Control stack
- Return addresses

6.7 RECURSION

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. This section introduces this powerful tool, and Sec. 6.9 will show how recursion may be implemented by means of stacks.

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a *recursive procedure*. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

(1) There must be certain criteria, called *base criteria*, for which the procedure does not call itself.

(2) Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

A recursive procedure with these two properties is said to be *well-defined.*

Similarly, a function is said to be *recursively defined* if the function definition refers to itself. Again, in order for the definition not to be circular, it must have the following two properties:

(1) There must be certain arguments, called *base values*, for which the function does not refer to itself.

(2) Each time the function does refer to itself, the argument of the function must be closer to a base value

A recursive function with these two properties is also said to be well-defined.

The following examples should help clarify these ideas.

10 Elementary Data Structures

In this chapter, we examine the representation of dynamic sets by simple data structures that use pointers. Although we can construct many complex data structures using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees. We also show ways to synthesize objects and pointers from arrays.

10.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. In this section we show how to use a simple array to implement each.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As Figure 10.1 shows, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $S.top$ that indexes the most recently

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

6.10 QUEUES

A *queue* is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

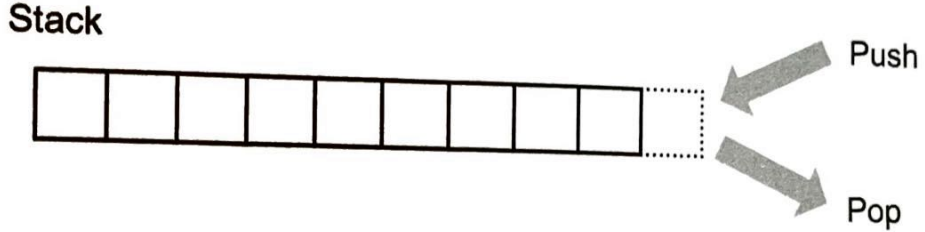
Real-life example

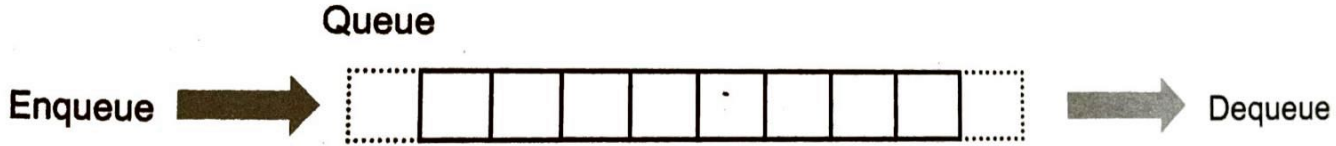
Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed. (Another structure, called a priority queue, is discussed in Sec. 6.13.)

h3 ## Stacks & Queues

A stack is a data structure that allows adding and removing elements in a last-in, first-out (LIFO) order. This means the element that is added last is the first element to be removed. Another name for adding and removing elements from a stack is pushing and popping. Stacks are often implemented using an array or linked list.

A queue is a data structure that allows adding and removing elements in a first-in, first-out (FIFO) order. Queues are also typically implemented using an array or linked list.





The main difference between a stack and a queue is the removal order: in the stack, there is a LIFO order, whereas in a queue it's a FIFO order. Stacks are generally used in recursive operations, whereas queues are used in more iterative processes.

Implementing a Queue

A queue implements FIFO (first-in first-out) ordering. As in a line or queue at a ticket stand, items are removed from the data structure in the same order that they are added.

It uses the operations:

- `add(item)`: Add an item to the end of the list.
- `remove()`: Remove the first item in the list.
- `peek()`: Return the top of the queue.
- `isEmpty()`: Return true if and only if the queue is empty.

A queue can also be implemented with a linked list. In fact, they are essentially the same thing, as long as items are added and removed from opposite sides.