Data Structure:

- Concepts of Data Structure
- Why are they required?
- Study of algorithms

Data Structure Definition:

1. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a Data Structure.

2. In computer science, a **Data Structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.

3. A Data Structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Why are Data Structures required?

The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

There are two types of Data Structures:

- Primitive Data structure
- Non-Primitive Data structure

Primitive Data Structure:

Primitive Data Structure is a basic data structure which can be directly operated by the machine instructions.
eg: int, float, double, character, pointer, Boolean.

Non-Primitive Data Structure:

Non-Primitive Data structure emphasis on structuring of a group of homogenous or heterogeneous data item.

There are two type of Non-Primitive Data Structure:

- Linear Data Structure
- Non-Linear Data Structure

Linear Data Structure:

A Data Structure which contains the linear arrangement of elements in the memory.This is known as Linear Data Structure.
e.g. Array, Stack, Queue, Linked List

A list which displays the relationship of adjacency between elements is said to be ***linear***.

Non-Linear Data Structure:

A data structure which represents a hierarchical arrangements of elements.
e.g. Trees and Graphs

What are the applications of Data Structure?

Some of the applications of data structure are

- Operating system
- Compiler design
- Statistical and Numerical analysis
- Database Management system
- Expert System
- Network analysis
- Artificial Intelligence

DATA STRUCTURE OPERATIONS:

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed.

The following are the major operations performed on data structures:

(1) **Traversing**: Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called visiting" the record.)

(2) **Searching**: Finding the location of the record with a given key value, or finding the locations of all records that satisfy one or more conditions.

(3) **Inserting**: Adding a new record to the structure.

(4) **Deleting**: Removing a record from the structure.

Sometimes two or more of the operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record. The following two operations, which are used in special situations, are also be considered:

(1) **Sorting**: Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

(2) **Merging**: Combining the records in two different sorted files into a single sorted file Other operations, e.g., **copying** and **concatenation**, are also used.

EXAMPLE :

An organization contains a membership file in which each record contains the following data for a given member:

Name, Address, Telephone Number, Age, Sex

(a) Suppose the organization wants to announce a meeting through a mailing. Then one would traverse the file to obtain Name and Address for each member.

(b) Suppose one wants to find the names of all members living in a certain area. Again one would traverse the file to obtain the data.

(c) Suppose one wants to obtain Address for a given Name. Then one would search the file for the record containing Name.

(d) Suppose a new person joins the organization. Then one would insert his or her record into the file.

(e) Suppose a member dies. Then one would delete his or her record from the file.

(f) Suppose a member has moved and has a new address and telephone number. Given the name of the member, one would first need to search for the record in the file. Then one would perform the "update"- i.e., change items in the record with the new data.

(g) Suppose one wants to find the number of members 65 or older. Again one would traverse the file, counting such members.

Arrays, and how different Data Structures implement same ADT.

- Concept of Array
- Different operations performed using Arrays

This lesson discusses a very common linear structure called all array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.

LINEAR ARRAY:

A ***linear array*** is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:

(a) The elements of the array are referenced respectively by an *index set* consisting of n Consecutive numbers.

(b) The elements of the array are stored respectively in successive memory locations. The number n of elements is called the *length* or *size* of the array. If not explicitly stated, we will assume the index set consists of the integers 1, 2, . . . , n. In general, the length or the number of data elements of the array can be obtained from the index set by the formula

Length = UB - LB + 1

Where UB is the largest index, called the *upper bound*, and LB is the smallest index, called the *lower bound*, of the array. Note that length = VB when LB = 1.

The elements of an array A may be denoted by the subscript notation

$A_1 A_2, A_3, \ldots, A_n$

or by the parentheses notation (used in FORTRAN, PL11 and BASIC)

A(l), A(2), . . . , A(N)

Or by the bracket notation (used in Pascal)

A[l], A[2], A[3], . . . , A[N]

We will usually use the subscript notation or the bracket notation. Regardless of the notation, the number K in A[K] is called a *subscript* or an *index* and A[K] is called a; ***subscripted variable***.

Note that subscripts allow any element of A to be referenced by its relative position in A.

EXAMPLE

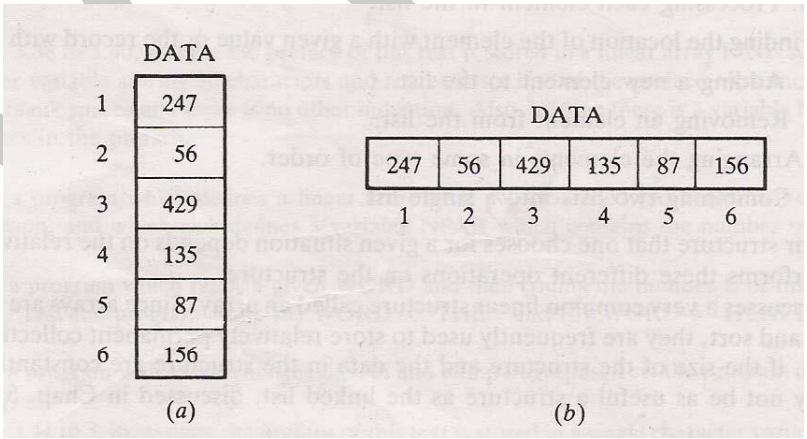
(a) Let DATA be a 6-element linear array of integers such that

DATA[1] =247 DATA[2] = 56 DA T A[3] = 429 DATA[4] = 135 DATA[5] = 87 DATA[6] = 156

Sometimes we will denote such an array by simply writing

DATA: 247, 56, 429, 135, 87, 156

The array DATA is frequently pictured as in Fig. (a) or Fig. (b).



(b) An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through 1984. Rather than beginning the index set with 1, it is more useful to begin the index set with 1932 so that.

AUTO[K] = number of automobiles sold in the year K

Then LB = 1932 is the lower bound and UB = 1984 is the upper bound of AUTO.

Length = UB - LB + 1 = 1984 -1930 + 1 = 55

That is, AUTO contains 55 elements and its index set consists of all integers from 1932 through 1984.

Each programming language has its own rules for declaring arrays. Each such declaration must give, implicitly or explicitly, three items of information: (1) the name of the array, (2) the data type of the array and (3) the index set of the array.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:

Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations as pictured in Fig. below. Let us use the notation

LOC (LA [K]) = address of the element LA [K] of the array LA

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base (LA)

and called the **base address** of LA. Using this address Base (LA), the computer calculates the address of any element of LA by the following formula:

LOC (LA [K]) = Base(LA) + w(K - lower bound)

where w is the number of. words per memory cell for the array _A. Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

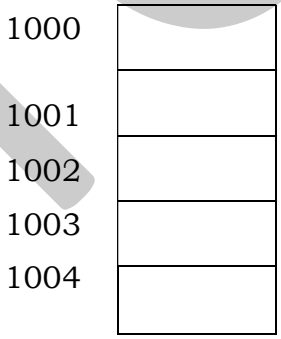


Fig: Computer memory

EXAMPLE

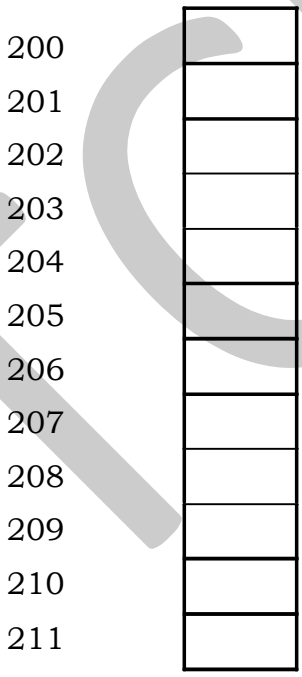
Consider the array AUTO in previous example, which records the number of automobiles, sold

each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. below. That is, Base (AUTO) = 200, and w = 4 words per memory cell for AUTO. Then

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, . . .
The address of the array element for the year K = 1965 can be obtained as:

LOC (AUTO [1965]) = Base (AUTO) + w (1965 -lower bound) = 200 + 4(1965 - 1932) = 332

Thus, the contents of this element can be obtained without scanning any other element in array AUTO.



.

Remark: A collection A of data elements is said to be indexed if any element of A, which we shall call A_K, can be located and processed in a time that is independent of K., The above discussion indicates that linear arrays can be indexed. This is a very important property of linear arrays. In fact, linked lists, which are covered in the next section, do not have this property.

TRAVERSING LINEAR ARRAYS:

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of A exactly once.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked lists, can also be easily traversed. On the other hand, the traversal of nonlinear structures, such as trees and graphs, is considerably more complicated.

Algorithm: (Traversing a Linear Array)

Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter]
   Set K ← LB
2. Repeat Steps 3 and 4 while K≤UB.
3. [Visit element]
   Apply PROCESS to LA[K]
4. [Increase counter]
   Set K ← K + 1
   [End of Step 2 loop]
5. Exit

INSERTING AND DELETING:

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation 'of adding another element to the collection A, and "deleting" refers to the operation of removing one of the elements from A. This section discusses inserting and deleting when A is a linear array.

Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.

EXAMPLE

Suppose TEST has been declared to be a 5-element array but data have been recorded only for TEST[l], TEST[2] and TEST[3]. If X is the value of the next test, then one simply assigns

TEST[4]:= X

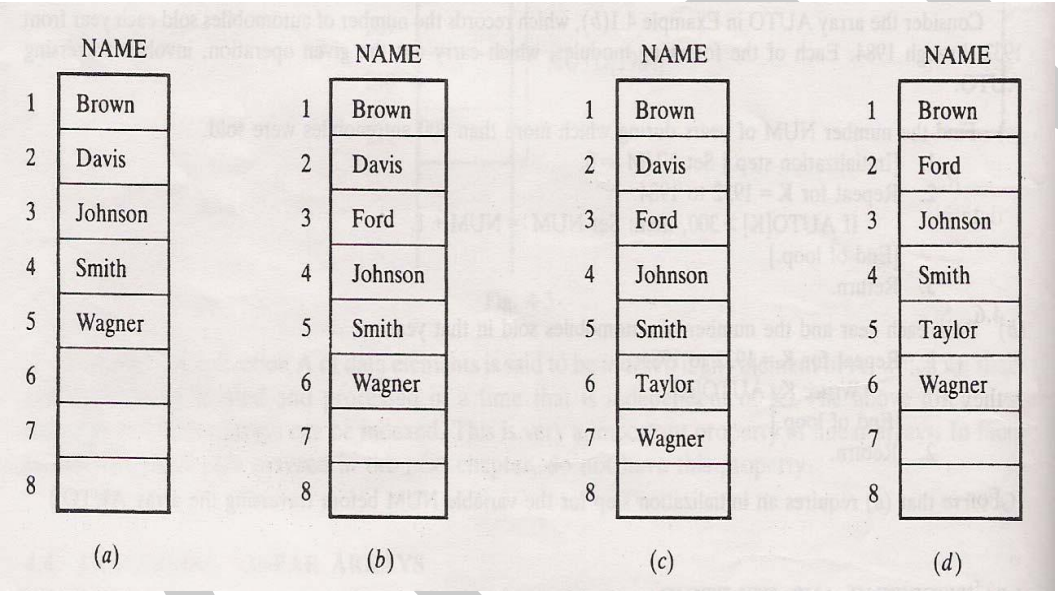
to add X to the list. Similarly, if Y is the value of the subsequent test, then we simply assign

TEST[5]:= Y

to add Y to the list. Now, however, we cannot add any new test scores to the list.

EXAMPLE

Suppose NAME is an 8-element linear array, and suppose five names are in the array, as in Fig. (a). Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose Ford is added to the array. Then Johnson, Smith and Wagner must each be moved downward one location, as in Fig. (b). Next suppose Taylor is added to the array; then Wagner must be moved, as in Fig. (c). Last, suppose Davis is removed from the array. Then the five names Ford, Johnson, Smith, Taylor and Wagner must each be moved upward one location, as in Fig. (d). Clearly such movement of data would be very expensive if thousands of names were in the array.



The following algorithm inserts a data element ITEM into the Kth position in a linear array LA with N elements. The first four steps create space in LA by moving downward one location each element from the Kth position on. We emphasize that these elements are moved in reverse order-i.e. first LA[N], then LA[N - 1], . . . , and last LA[K];-otherwise data might be erased.

In more detail, we first set J:= N and then, using J as a counter, decrease J each time the loop is executed until J reaches K. The next step, Step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

Algorithm: (Inserting into a Linear Array)

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤ N. This algorithm inserts an element ITEM into the K^{th} position in LA.

1. [Initialize counter]
   Set J ← N
2. Repeat Steps 3 and 4 while J >= K
3. [Move J^{th} element downward]
   Set LA [J + 1] ← LA[J]
4. [Decrease counter]
   Set J ← J - 1
   [End of Step 2 loop]
5. [Insert element]
   Set LA [K] ← ITEM
6. [Reset N]
   Set N ← N +1
7. Exit

Algorithm: (Deleting from a Linear Array)

DELETE(LA, N, K ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K≤ N. This algorithm deletes the K^{th}element from LA.

1 Set ITEM ← LA[K]
2. Repeat for J ← K to N - 1:
   [Move J + 1st element upward] Set LA[J] ← LA[J + 1]
   [End of loop]
3. [Reset the number N of elements in LA]
   Set N ← N - 1
4. Exit

STACK:-

- What is a Stack?
- What operations can be performed on it?
- Applications

The linear lists and linear arrays 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 situations are stacks and queues.

A stack is a linear structure in which items may be added or removed only at one end.

Figure below pictures three everyday examples of such a structure: a stack of dishes, a stack of pennies and a stack of folded towels. Observe that an item may be added or removed only from the top of any of the stacks. This means, in particular, that the last item to be added to a stack is the first item to be removed. Accordingly, stacks are also called last-in first-out (LIFO) lists. Other names used for stacks are "piles" and "push- down lists." Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.

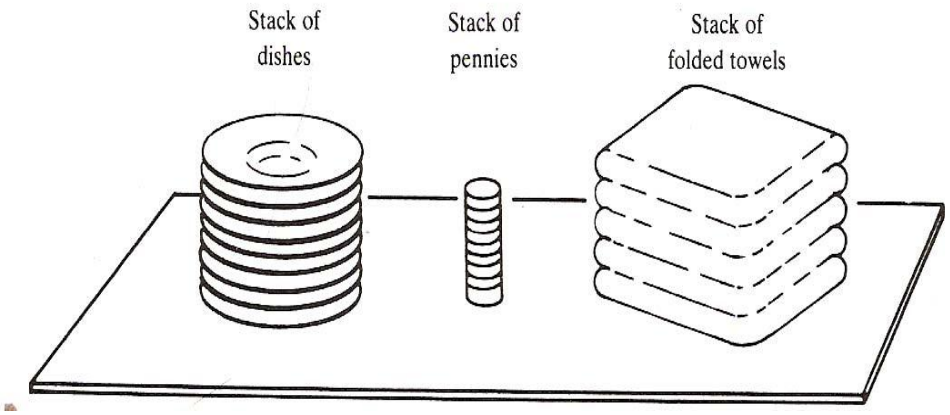


Figure- Stack

Definitions:

A **Stack** is a list of elements in which an element may be inserted or deleted only at one end, called "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. It is also called **L**ast **I**n **F**irst **O**ut (LIFO).

Special terminology is used for Four 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.

(c) "PEEP" is used to return the value of the Ith element from the top of the stack.

(d) "CHANGE" is used to change the value of Ith element from the top of the stack.

EXAMPLE:

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Figure below shows three ways of picturing such a stack. For notational convenience, we will frequently designate the stack by writing:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

The implication is that the right-most element is the top element. We emphasize that, regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted, and so on. Consequently, the elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

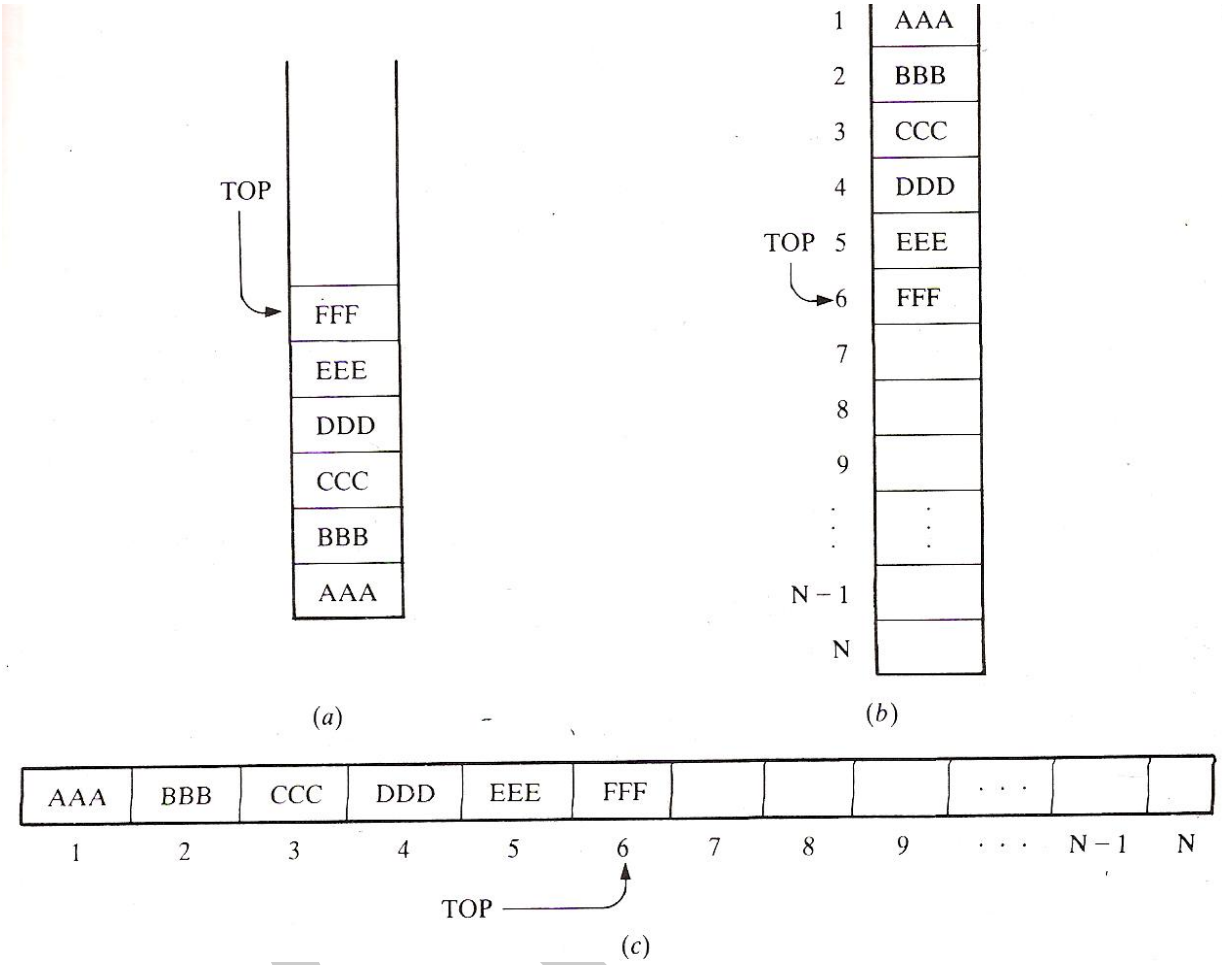
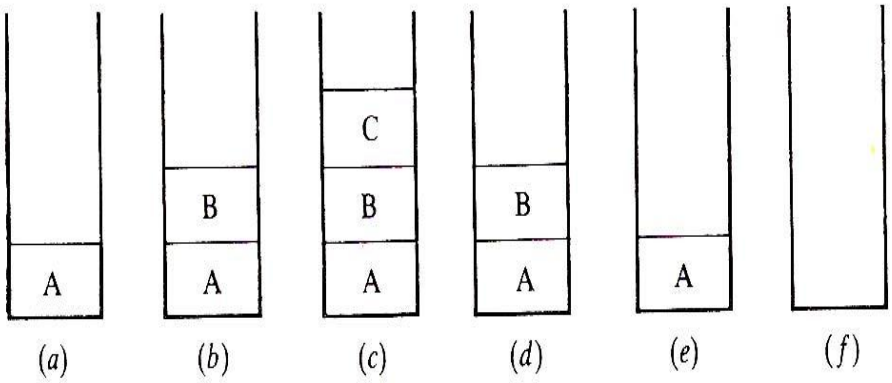


Fig: Diagrams of Stacks

Postponed Decisions

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. (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. (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. (c), and begin to process D.



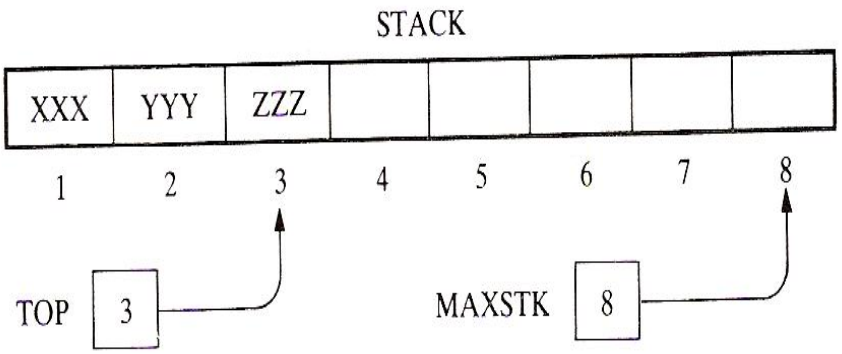
On the other hand, suppose we are able to complete the processing of project D. Then the only project we may continue to process is project C, which is on top of the stack. Hence we remove folder C from the stack, leaving the stack as pictured in Fig. (d), and continue to process C. Similarly, after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

ARRAY REPRESENTATION OF STACKS

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Unless otherwise stated or implied, each of our stacks will be maintained by a linear array STACK; a pointer Variable TOP, which contains the location of the top element of the stack; and a variable MAXSTK which gives the maximum number of elements that can be held by the stack. The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.

Figure on next page pictures such an array representation of a stack. (For notational convenience, the array is drawn horizontally rather than vertically.) Since TOP = 3, the stack has three-elements, XXX, YYY and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack.



The operation of adding (pushing) an item onto a stack and the operation of removing (popping) an item from a stack may be implemented, respectively, by the following procedures, called PUSH and POP. In executing the procedure PUSH, one must first test whether there is room in the stack for the new item; if not, then we have the condition known as **overflow**.

Analogously, in executing the procedure POP, one must first test whether there is an element in the stack to be deleted; if not, then we have the condition known as **underflow**.

Algorithm: **PUSH (STACK, TOP, MAXSTK, ITEM)**
STACK is a Linear Array.
TOP is a Pointer which contains the location of the top element of the stack.
MAXSTK which gives the maximum number of elements held by Stack.
ITEM is element insert onto Stack.

This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP ← MAXSTK then
   PRINT : "STACK IS OVERFLOW"
   Return.
2. [Increases TOP by 1]
   Set TOP ← TOP + 1
3. [Inserts ITEM in new TOP position]
   Set STACK [TOP] ← ITEM
4. Exit

Algorithm: **POP (STACK, TOP, ITEM)**
STACK is a Linear Array.
TOP is a Pointer which contains the location of the top element of the stack.
ITEM is element Delete from Stack.

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP ← 0 then
   PRINT : "STACK IS UNDERFLOW"
   Return.
2. [Assigns TOP element to ITEM]
   Set ITEM ← STACK [TOP]
3. [Decreases TOP by 1]
   Set TOP ← TOP - 1
4. Exit

EXAMPLE

(a) Consider the stack in previous figure. We simulate the operation PUSH (STACK, WWW):

1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK [TOP] = STACK [4] = WWW.
4. Return.

Note that WWW is now the top element in the stack.

(b) Consider again the same stack. This time we simulate the operation POP (STACK, ITEM):

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 - 1 = 2.
4. Return.

Observe that STACK [TOP] = STACK [2] =YYY is now the top element in the stack.

Algorithm: PEEP (STACK, TOP, I)

STACK is a Linear Array.
TOP is a Pointer which contains the location of the top element of the stack.
I is element to display from Stack.

This Function returns the value of the Ith element from the top of the stack.

1. [Check for Stack underflow]
   If TOP – I + 1 <= 0 then
   PRINT: " STACK IS UNDERFLOW ON PEEP"
   Return
2. [Return Ith element from the top of the stack]
   Return(STACK[TOP – I + 1])
3. [Finished]
   EXIT

Algorithm: CHANGE (STACK, TOP, X, I)

STACK is a Linear Array.
TOP is a Pointer which contains the location of the top element of the stack.
I is element to Change from Stack.
This Function Changes the value of the Ith element from the top of the stack to the value contained in X

1. [Check for Stack underflow]
   If TOP – I + 1 <= 0 then
   PRINT: "STACK IS UNDERFLOW ON CHANGE"
   Return
2. [Return Ith element from the top of the stack]
   STACK[TOP – I + 1] ← X
3. [Finished]
   EXIT

Applications of Stacks

- Polish Notations
- Recursion

POLISH NOTATION

For most common arithmetic operations,the operator symbol is placed between its two Operands. For example,

A+B C-D E*F G/H

This is called infix notation. With this notation, we must distinguish between

(A+B)*C and A + (B * C)

by using either parentheses or some operator-precedence convention such as the usual precedence levels discussed above. Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the **Polish mathematician Jan Lukasiewicz**, refers to the notation in which the operator symbol is placed before its two operands. For example,

+AB -CD *EF /GH

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

(A + B) *C= [+AB]*C = *+ABC

A + (B*C) = A + [*BC] = +A*BC

(A + B) / (C - D) = [+ AB] / [- CD] = / + AB – CD

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing, expressions in Polish notation.

Reverse Polish notation(Postfix or Suffix) refers to the analogous notation in which the operator symbol is placed after its two operands:

AB+ CD- EF* GH/

Again,one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called postfix (or suffix) notation, whereas prefix notation is the term used for Polish notation, discussed in the preceding paragraph.

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task.

Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator x is encountered, then:
    a. Remove the two top elements of STACK, where A is the top element and B is the next-top element.
    b. Evaluate B⊗A.
    c. Place the result of (b) back on STACK.

   [End of If structure]
   [End of Step 2 loop]
5. Set VALUE equal to the top element to STACK
6. Exit

We note that, when Step 5 is executed, there should be only one number on STACK.

Example:

Consider the following arithmetic expression P written in postfix notation:

P : 5, 6, 2, +, *, 12, 4, /, -,)

(Commas are used to separate the elements of P so that, 5,6,2 is not interpreted as the number 562.)

Symbol Scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	−	37
(10)	)	

Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation.Besides operands & operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations (↑), multiplications (*), divisions (/), additions (+) and subtractions (-), and that they have the usual three levels of precedence as given above.

We also assume that operators on the same level, including exponentiations, are performed from left to right unless otherwise indicated by parentheses. (This is not standard, since expressions may contain unary operators and some languages perform the exponentiations from right to left. However, these assumptions simplify our algorithm.)

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators, which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm:POLISH(Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q
3. until the STACK is empty:
4. If an operand is encountered, add it to P.
5. If a left parenthesis is encountered, push it onto STACK.
   If an operator is encountered, then:
   (a)Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than Operator.
   (b) Add operator to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
   7.Exit.

Consider the following arithmetic infix expression Q:

Q: A+(B*C- (D/E↑F)*G)*H

Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) −	(+ (−	A B C *
(8) (	(+ (− (	A B C *
(9) D	(+ (− (	A B C * D
(10) /	(+ (− (/	A B C * D
(11) E	(+ (− (/	A B C * D E
(12) ↑	(+ (− (/ ↑	A B C * D E
(13) F	(+ (− (/ ↑	A B C * D E F
(14))	(+ (−	A B C * D E F ↑ /
(15) *	(+ (− *	A B C * D E F ↑ /
(16) G	(+ (− *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * −
(18) *	(+ *	A B C * D E F ↑ / G * −
(19) H	(+ *	A B C * D E F ↑ / G * − H
(20))		A B C * D E F ↑ / G * − H * +

P: A B C * D E F ↑ / G * - H * +

RECURSION:

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion.

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.

Function Factorial (*n*)

The above Function obtains the factorial of a given number n in a recursive manner. If the number is positive and other than zero then factorial is computed otherwise for zero the Function returns 1.

Step 1 If number is zero
if (n = 0) then
return 1.

Step 2 If number is greater than zero.
if (n > 0) then
return (n * Factorial (n – 1))

Step 3 END

Function Fibseq (*n*)

The above Function generates the fibonacci sequence of a given number, in a recursive manner.

Step 1 Is zero ?
if (n = 0) then
return 0

Step 2 Is one ?
if (n = 1) then
return 1

Step 3 number is greater than 1, recursive call if (n > 1) then
return (Fibseq (n – 1) + Fibseq (n – 2))

Procedure move (n, X, Z, Y).

The above Procedure moves '*n*' disks from peg *X* to peg *Z* using temporary peg Y. In this, '*n*' denotes the number of disk in peg *X*. Here move function is defined in a recursive manner.

Step 1 $n = 1$, one disk to move
if ($n = 1$) then
call to move (n, X, Z, Y)
Step 2 if n is greater than one
call to move ($n - 1$, X, Y, Z)
moving ($n - 1$) disks from peg X to peg Y, using peg Z
call to move ($n - 1$, Y, Z, X)
moving ($n - 1$) disk from peg Y to peg Z, using peg X.
Step 3 return at the point of call
return.

QUEUE

- What is a Queue?
- What operations can be performed on it?

A **queue** is a linear list in which items may be added only at one end and items may be removed-only at the other end.

The name "queue" likely comes from the everyday use of the term. Consider: queue of people waiting at a bus stop, as pictured in fig. below. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first Clearly, the first person in the line is the first person to leave. Thus queues are also called **first-in first-out (FIFO)** lists.

Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others..

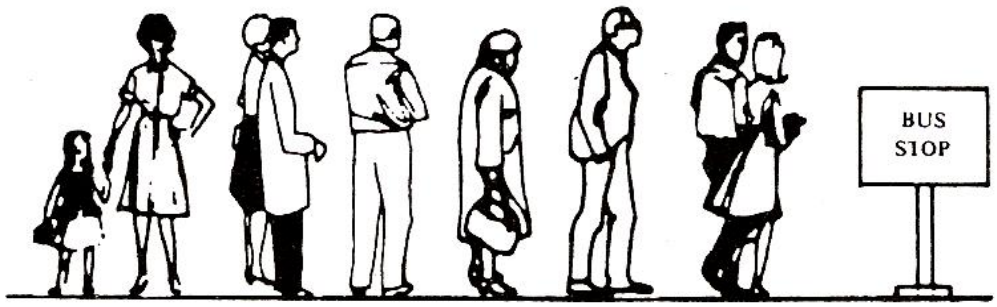


Figure: People waiting for a bus

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.

Representation of Queues

Queues may be represented in the computer in various ways, usually by means of one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty. Figure below indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

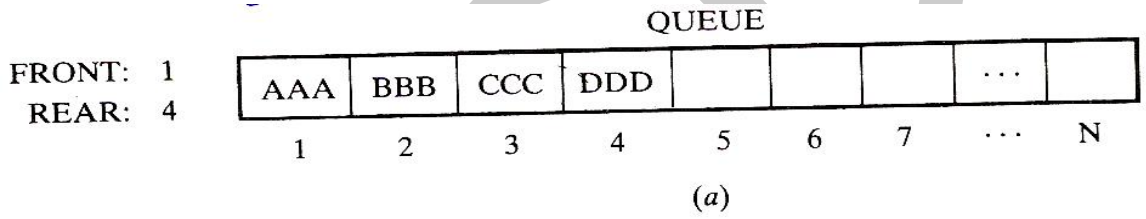
FRONT = FRONT + 1

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

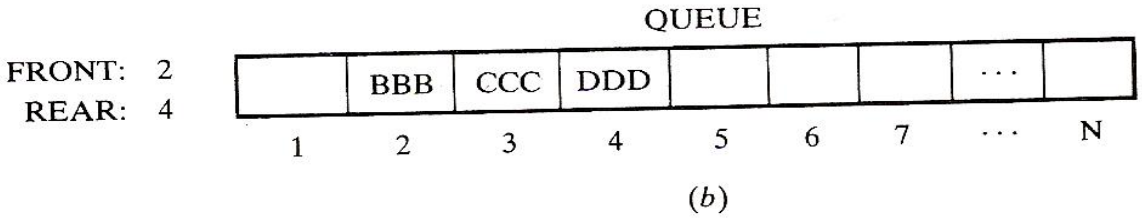
REAR: = REAR + 1

This means that after N insertions, the rear element of the queue will occupy QUEUE [N] or, in other words, eventually the queue will occupy the last part of the array. This occurs even through the queue itself may not contain many elements.

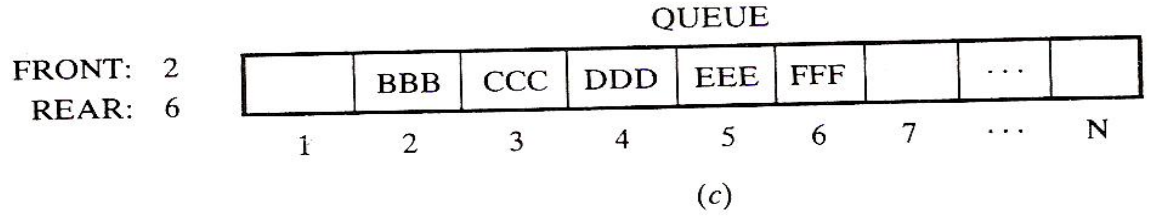
Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array, i.e., when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array



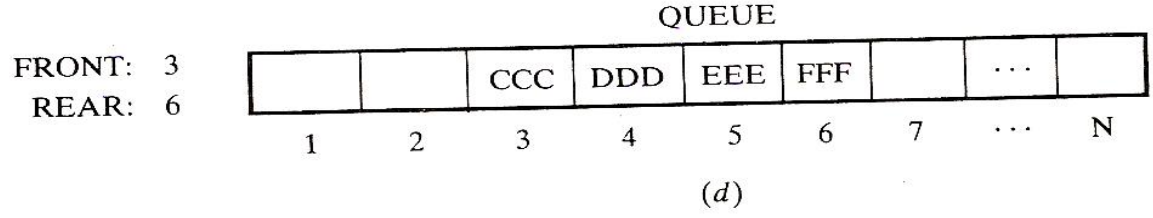
(a)



(b)



(c)



(d)

Algorithm:(Insertion into Queue)

QINSERT (Q, FRONT, REAR, ITEM, N)

FRONT and REAR are pointers element of a Queue Q consisting of N elements.

1. [Check for Queue Overflow]
   IF REAR >= N then
   PRINT: "QUEUE IS OVERFLOW"
   Return
2. [Increment REAR Pointer]
   REAR ← REAR + 1
3. [Insert an element at REAR of Queue]
   Q[REAR] ← ITEM
4. [Set the FRONT Pointer]
   IF FRONT ← -1 then
   FRONT = 0
5. [Finished]
   Exit

Algorithm:(Deletion from Queue)

QDELETE (Q, FRONT, REAR, ITEM)

FRONT and REAR are pointers element of a Queue Q.

1. [Check for Queue 'Underflow']
   IF FRONT ← 0 then
   PRINT: " QUEUE IS UNDERFLOW"
   Exit
2. [Remove an element from Queue]
   ITEM ← Q[FRONT]
3. [Print the poped element]
   PRINT: "ITEM"
4. [Check for empty queue]
   IF FRONT = REAR then
   FRONT ← 1
   REAR ← 1
   Else
   FRONT ← FRONT + 1
5. [Finished] Exit

DEQUEUE:

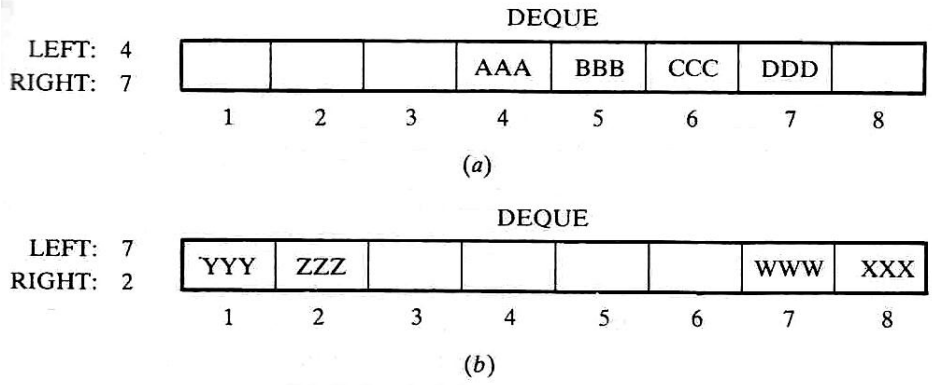
A dequeue (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term dequeue is a contraction of the name double-ended queue.

There are various ways of representing a dequeue in a computer. Unless it is –otherwise stated or implied, we will assume our dequeue is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the dequeue. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE [I] comes after DEQUE [N] in the array. Figure below pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two kinds of a dequeue:

- **Input-restricted deque**- Input-restricted dequeue is a dequeue, which allows insertions at only one end of the list but allows deletions at both ends of the list.
- **Output-restricted deque**- Output-restricted dequeue is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

The procedures, which insert and delete elements in dequeues and the variations on those procedures, are given as supplementary problems. As with queues, a complication may arise (a) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (b) when there is underflow, that is, when an element is to be deleted from a deque,which is empty. The procedures must consider these possibilities.



Algorithm:(Insertion into DQueue from right)

QINSERT RIGHT(DQ, FRONT, REAR, ITEM, N)

FRONT and REAR are pointers element of a DQueue DQ consisting of N elements.

1. [Check for Queue Overflow]
   IF REAR >= N then
   PRINT: "DQUEUE IS OVERFLOW"
   Return
2. [Increment REAR Pointer]
   REAR ← REAR + 1
3. [Insert an element at REAR of Queue]
   DQ [REAR] ← ITEM
4. [Set the FRONT Pointer]
   IF FRONT ← -1 then
   FRONT = 0
5. [Finished]
   Exit

Algorithm:(Insertion into DQueue from left)

QINSERT LEFT(DQ, FRONT, REAR, ITEM, N)

FRONT and REAR are pointers element of a Deueue DQ consisting of N elements.

1. [Check for Queue Overflow]
   If FRONT = 1 then
   PRINT: ”DQUEUE IS OVERFLOW”
   Return
2. [check FRONT and REAR]
   If FRONT = 0 then
   FRONT ← REAR ← N
   Else
   FRONT ← FRONT - 1
3. [Insert an element at REAR of Queue]
   DQ [FRONT] ← ITEM
4. [Finished]
   Exit

Algorithm:(Deletion from DQueue from left)

QDELETE LEFT(DQ, FRONT, REAR, ITEM)

FRONT and REAR are pointers element of a Queue DQ.

1. [Check for Queue ‘Underflow’]
   IF FRONT ← 0 then
   PRINT: “ DQUEUE IS UNDERFLOW”
   Exit
2. [Remove an element from Queue]
   ITEM ← DQ [FRONT]
3. [Print the poped element]
   PRINT: “ITEM”
4. [Check for empty queue]
   IF FRONT = REAR then
   FRONT ← 1
   REAR ← 1
   Else
   FRONT ← FRONT + 1
5. [Finished] Exit

Algorithm:(Deletion from Queue from right)
QDELETE RIGHT(DQ, FRONT, REAR, ITEM)
FRONT and REAR are pointers element of a Queue DQ.
1. [Check for Queue 'Underflow']
IF FRONT ← 0 then
PRINT: " QUEUE IS UNDERFLOW"
Exit
2. [Remove an element from Queue]
ITEM ← Q [FRONT]
3. [Print the poped element]
PRINT: "ITEM"
4.[Check for empty queue]
IF FRONT = REAR then
FRONT ← 1
REAR ← 1
Else
FRONT ← FRONT + 1
5. [Finished]
Exit

Circular Queue:

From above discussion in a linear queue, we face the problem of overflow of a queue frequently. If queue contain total maximum element and we want to insert a new element, this type problems can be solved by using circular queue instead of Linear queue.

CQINSERT (FRONT, REAR, CQ, N, ITEM)

FRONT and REAR are pointers element of a Circular Queue CQ consisting of N elements.

1. [Check for Queue Overflow]
   If FRONT = 1 and REAR = N then
   PRINT: ”CIRCULAR QUEUE IS OVERFLOW”
   Return
2. [Find the new value of REAR Pointer]
   If REAR = N
   REAR ← 1
   Else
   REAR ← REAR + 1
3. [Insert an element at REAR of Queue]
   CQ [REAR] ← ITEM
4. [Set the FRONT Pointer]
   IF FRONT ← 0 then
   FRONT ← 1
5. [Finished]
   Exit

CQDELETE (FRONT, REAR, CQ, N, ITEM)

FRONT and REAR are pointers element of a Circular Queue CQ consisting of N elements.

1. [Check for Queue Empty]
   If FRONT = 0 then
   PRINT: ”CIRCULAR QUEUE IS EMPTY”
   Return
2. [Delete an element at FRONT of Queue]
   ITEM ← CQ [FRONT]
3. [Set the FRONT Pointer]
   IF FRONT = REAR then
   FRONT ← REAR ← 0
   Else
   FRONT ← FRONT + 1
4. [Finished]
   Return

PRIORITY QUEUE:

A ***priority queue*** is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.
(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

LINKED LISTS

A ***linked list***, or ***one-way list***, is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***.

That is, each node is divided into two parts: the first part contains the ***information*** of the element, and the second part, called the ***link field* or *next pointer*** field, contains the address of the next node in the list.

Figure below is a schematic diagram of a linked list with 6 nodes. Each node is pictured with two parts. The left part represents the information part of the node, which may contain an entire record of data items (e.g. NAME, ADDRESS, . . .). The right part represents the next pointer field of the node, and there is an arrow drawn from it to the next node in the list. This follows the usual practice of drawing an arrow from a field to a node when the address of the node appears in the given field. The pointer of the last node contains a special value, called the null pointer, which is any invalid address.

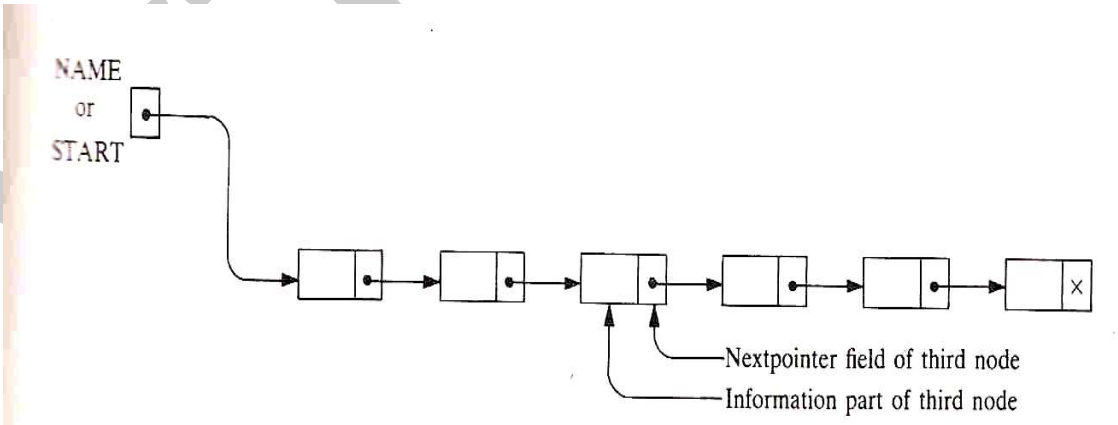
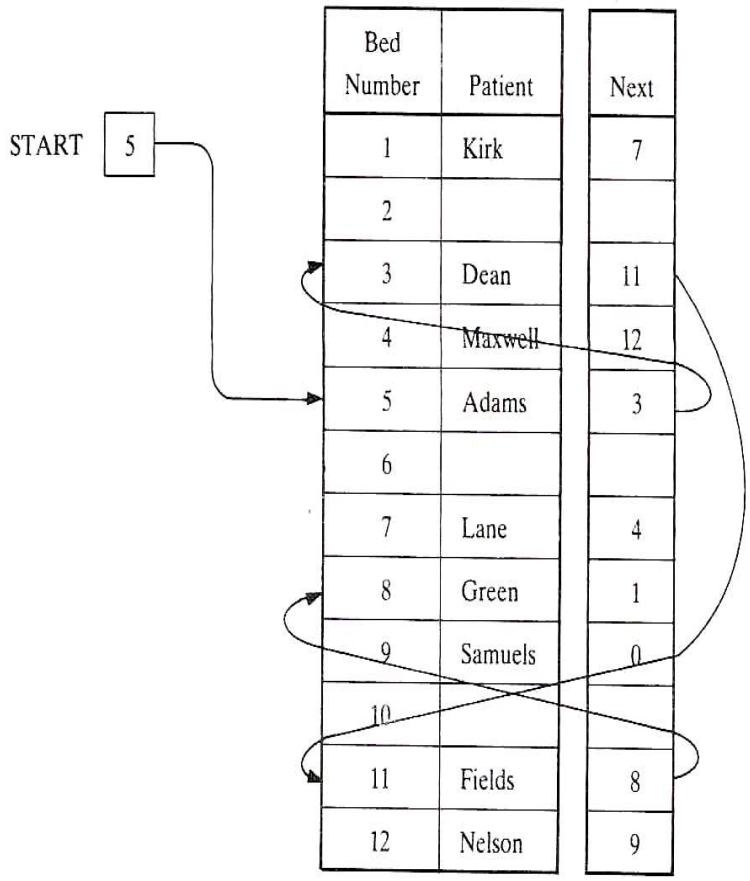


Fig: Linked list with 6 nodes

(In actual practice, 0 or a negative number is used for the null pointer.) The null pointer, denoted by x in the diagram, signals the end of the list. The linked list also contains a *list pointer variable*-called START or NAME-which contains the address of the first node in the list; hence there is an arrow drawn from START to the first node. Clearly, we need only this address in START to trace through the list. A special case is the list that has no nodes. Such a list is called the **null list** or **empty list** and is denoted by the null pointer in the variable START.

EXAMPLE

A hospital ward contains 12 beds, of which 9 are occupied as shown in Fig below. Suppose we want an alphabetical listing of the patients. This listing may be given by the pointer field, called Next in the figure. We use the variable START to point to the first patient. Hence START contains 5, since the first patient, Adams, occupies bed 5. Also, Adams's pointer is equal to 3, since Dean, the next patient, occupies bed 3; Dean's pointer is 11, since Fields, the next patient, occupies bed 11; and so on. The entry for the last patient (Samuels) contains the null pointer, denoted by O. (Some arrows have been drawn to indicate the listing of the first few patients.)



REPRESENTATION OF LINKED LISTS IN MEMORY:

Let LIST be a linked list. Then LIST will be maintained in memory, unless otherwise specified or implied, as follows. First of all, LIST requires two linear arrays-we will call them here INFO and LINK-such that INFO[K] and LINK[K] contain, respectively, the information part and the next pointer field of a node of LIST. As noted above, LIST also requires a variable name-such as START-which contains the location of the beginning of the list, and a next pointer sentinel-denoted by NULL-which indicates the end of the list. Since the subscripts of the arrays INFO and LINK will usually be positive, we will

choose NULL = 0, unless otherwise stated.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

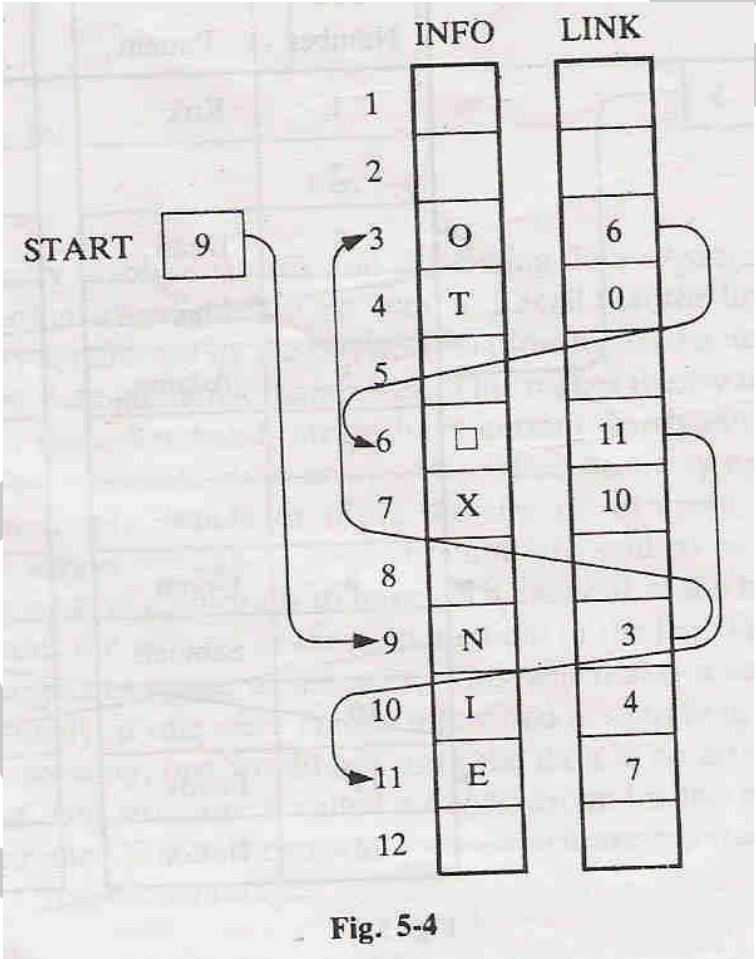


Fig. 5-4

EXAMPLE:

Figure pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = 0 is the second character.
LINK[3] = 6, so INFO[6] = 0 (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7]= X is the fifth character.
LINK[7] = 10, so INFO 0] = I is the sixth character.
LlNK[10] = 4, so INFO] = T is the seventh character.
LlNK[4] = 0, the NULL value, so the list has ended.

In other words, NO EXIT is the character string.