DATA STRUCTURES AND ALGORITHMS

Concepts, Techniques and Applications

GAV PAI

CORE PAPER VI.

DATA STRUCTURES.
AND ALGORITHMS

Concepts, Techniques, operations and Applications.

[For Semester 3]

B.C.A.-DATA STRUCTURES

UNIT I: Data Structures – Definition and Classification - Arrays – Array Operations – Representation of Arrays – Applications of Arrays. Stack – Operations on Stacks -Stack applications – Infix to postfix notation and Evaluation of Postfix notation Queues - Operations on the Queues - Circular queue – Dequeue - Priority queue - Applications of queue.

UNIT II: Introduction to the Linked List - Basic operations on linked list – Singly Linked Lists – Doubly Linked Lists – Circularly Linked Lists – Applications of Linked List – Addition of Polynomials – Linked Stacks – Linked Queues.

UNIT III: Trees - Basic Terminology - Binary Trees - Representation of Trees and Binary trees – Binary Tree Traversals – Threaded Binary Trees – Binary Search Tree - Insertion and Deletion operations at Binary Search Trees - Applications of Trees.

UNIT IV: Graphs – Basic Terminology – Representation of Graphs - Graph Traversals – BFS, DFS - Applications of Graphs – Minimum Cost Spanning tree and Dijkstra's Shortest Path. Searching Techniques – Linear Search – Binary Search – Fibonacci Search.

UNIT V: Sorting techniques – Bubble Sort, Selection Sort, Quick Sort, Merge Sort, and Insertion Sort – Introduction to Hashing – Hash table structure – Hash Functions – Linear Open Addressing – Chaining.

TEXT BOOKS:

1. **S. Sahni & E. Horowitz,** "*Fundamentals of Data Structure*", Second Edition, 1999, Galgotia Publications.
2. **Richard F Gilberg Behrouz A. Forouzan, Thomson,** "*Data Structure – A Pseudocode Approach with C*", First Edition, 2005, University of California.
3. **Seymour Lipschutz,** "*Schaum's Outline of Theory and Problems of Data structure*", Second Edition, Tata McGraw-Hill.
4. **Alfred V.Aho, John E.Hopcraft, Jeffrey D.Ullman,** "*Data Structures and Algorithms*", 2009, Addison-Wesley.

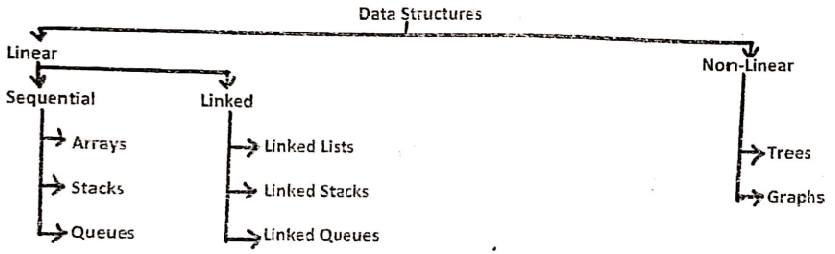
REFERENCE BOOKS:

1. **G.A. Vijayalakshmi Pai,** "*Data structures and Algorithms- Concepts, Techniques and Applications*", First Edition, 2011, Tata McGraw-Hill
2. **Dr. A. Chitra,** "*Data Structures*", 2009, Vijay Nicole Imprints Private Limited.

E- REFERENCES:

- http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-%20Guwahati/data_str_algo/frameset.htm
- http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html
- en.wikibooks.org/wiki/Data_structures

The major tasks done in any computer system are based on 3 activities storing, accessing, manipulations of data. People need to have an abstract thinking to understand the principles and techniques to solve the vast array of unfamiliar problems that arise in a rapidly changing field. Even a good programmer may unable to write new algorithms for the given problem if he has no knowledge in data structures. Hence "**Data structure is a collection of data elements organized in a specified manner and accessing functions are defined to store and retrieve individual data elements**". The classification of data structure is as given below:



A linear data structure is one in which the elements have a linear ordering between them. An element can be marked as first, second, third etc. A nonlinear data structure is one in which such an ordering is not present.

ARRAYS

An array refers to the homogeneous collection of elements stored contiguously in memory. Arrays follow static allocation of memory (ie) once the size of the array is declared it cannot be expanded nor contracted. The general features of an array are:

- A finite collection of elements.
- All elements of an array are of same data type.
- It is an ordered collection since we can label the elements as first, second, third etc.
- Each element of an array is stored adjacent to each other (ie) follows contiguous memory allocation.
- All elements are referred by same and single name.
- It is a random access data structure (ie) any element can be picked up and manipulated without sequential traversal.
- Individual element of an array is addressed by an integer called as index or subscript. It denotes the relative position of the element within the array from the starting position of the array.
- The first index of the array is called as Lower Bound while the last index of the array is called as Upper Bound. Bounds depend upon the programming language.

CIRCULARLY LINKED LIST

A variant of the singly linked list is circular linked list in which the last node's link part points to the first node.

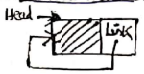
Advantages:

- Accessibility of a node- one can access any node from any given node due to the circular movement permitted by links.
- Deletion of a node – one need not specify the predecessor.
- Relative efficiency in the implementation of list based operations like merging of 2 lists, splitting of a list into two, erasing a list etc.

Disadvantages:

- During processing one has to make sure that one does not get into an infinite loop owing to the circular nature of the queue.

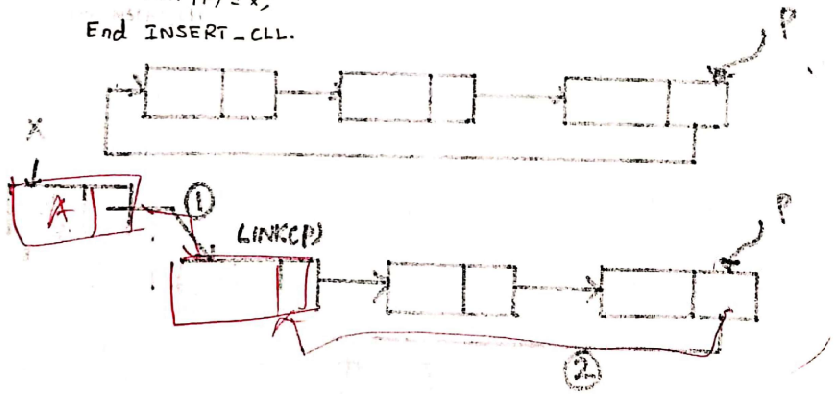
A solution to the infinite loop in circular linked list is having a special node called header nodes. So the list may be called a Headed circularly linked list or circular linked list with head node. The list can never be empty and will have a empty header node. If LINK(HEAD)=HEAD, it implies that the circular list is empty. Usually, the data field of the header node is empty, represented by a shaded portion in diagram.



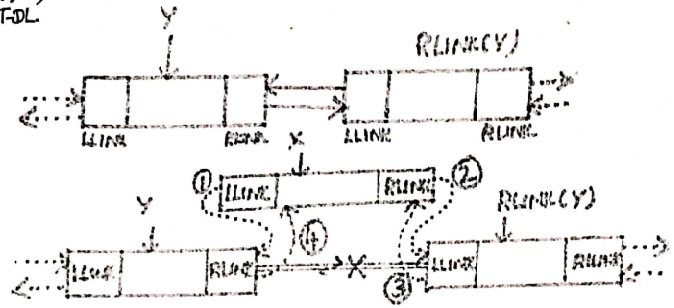
Insert Operation in Circularly Linked List

The following algorithm will insert a new node X as the left-most element in a list represented by P.

```
Procedure INSERT_CLL (P,x,A)
    Call GETNODE (x);
    DATA (x) = A;
    LINK (x) = LINK (P);
    LINK (P) = x;
End INSERT_CLL.
```



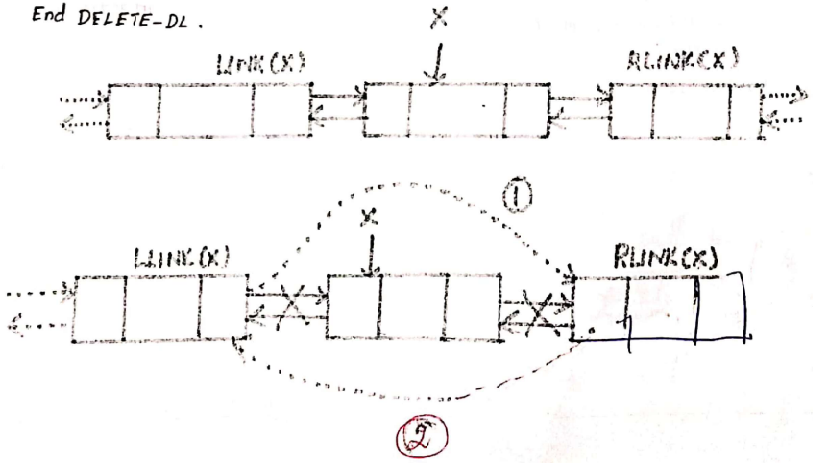
```
Procedure INSERT-DL(X,Y,ITEM)
    Call GETNODE(X);
    DATA(X)=ITEM;
    LLINK(X)=Y;
    RLINK(X)=RLINK(Y);
    LLINK(RLINK(Y))=X;
    RLINK(Y)=X;
End INSERT-DL.
```



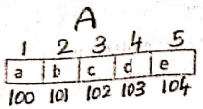
Delete Operation in Doubly Linked List

The below procedure is used to delete a node from a doubly linked list P.

```
Procedure DELETE-DL (P,X)
    IF(P=NULL) then
        Call ABONDON-DELETE;
    else
    {
        RLINK(LLINK(X)) = RLINK(X);
        LLINK(RLINK(X)) = LLINK(X);
        Call RETURN(X);
    }
End DELETE-DL.
```



Eg: char A[5]



Types of Arrays and its Representation in Memory

According to the number of index, the arrays can be broadly classified into three types as One-Dimensional Array, Two-Dimensional Array and Multi-Dimensional Array.

- One-Dimensional Array: It is the simplest form of array. It indicates a linear set of elements. It is declared as

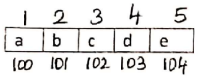
  datatype arrayname[size];

Eg:int a[100]; defines an integer array of size 100. The array declaration involves both the type of element that will be contained in the array as well as the maximum number of elements that will be stored inside the array. It is also called as Vectors. a[100] can be written as a[1:100].

Number of elements in the array=u-l+1
Address of any ith element in the array=α+(i-l)
where u=upper bound, l=lower bound and α=base or starting address of the array.

Eg:-The following figure shows the representation of one-dimensional array in memory.

char A[5];



Here u=5,l=1,α=100

Number of elements in A=5-1+1=5

Address of A[3]=100+(3-1)=100+2=102.

- Two-Dimensional Array: They are popularly called as matrices. As the name implies, this type of array is having two indices-one for row and other for column. It is usually denoted as

Arrayname[l1:u1,l2:u2] where l1=lower bound of row, u1=upper bound of row, l2=lower bound of column, u2=upper bound of column.

Eg: Representation of 3 subject marks for 4 students may be given as the following matrix.

	Col 1	Col 2	Col 3
Row 1	80	95	92
Row 2	85	76	89
Row 3	91	90	98
Row 4	77	65	35

2D arrays are logical representation while the memory storage is single dimension. Hence there should be a mapping between 2D representation and single dimension. This mapping should suggest a systematic way of storing the elements of a two-dimensional array into a one-dimensional array. This can be done in two ways as row-major representation in which the elements of the row gains priority first for storing while in column-major representation, the elements of the column gains priority first for storing in memory.

Eg: Row Major Representation of the above matrix

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)	(4,1)	(4,2)	(4,3)
80	95	92	85	76	89	91	90	98	77	65	35

Column-Major Representation of the above matrix

(1,1)	(2,1)	(3,1)	(4,1)	(2,1)	(2,2)	(2,3)	(2,4)	(3,1)	(3,2)	(3,3)	(3,4)
80	85	91	77	95	76	90	65	92	89	98	35

Number of elements in the array=(u1-l1+1)(u2-l2+1)

Address of any (i,j)th element in the array=α+(i-l1) (u2-l2+1)+(j-l2)

where α=base or starting address of the array.

Eg: int A[4,3] is a matrix with 4 rows and 3 columns. Consider the values at the matrix as,

(1,1)	(1,2)	(1,3)	(2,1)	(2,2)	(2,3)	(3,1)	(3,2)	(3,3)	(4,1)	(4,2)	(4,3)
80	95	92	85	76	89	91	90	98	77	65	35
100	101	102	103	104	105	106	107	108	109	110	111

With l1=1,u1=4,l2=1,u2=3, α=100

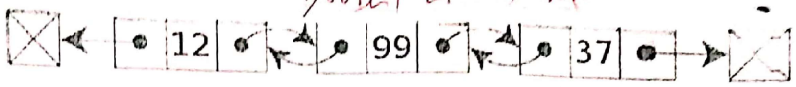
Number of elements in the above array=(4-1+1)(3-1+1)=12

Address of [2,3]rd element in the given array=100+(2-1)(3-1+1)+(3-1)=100+(1)(3)+2=105

- Multi-Dimensional Array: If the index of the array goes beyond 2, it is normally referred as multi-dimensional array.

Eg: To denote the x,y,z axis in image processing, the array can be defined as A[x.,y,z]

DOUBLY LINKED LIST



A doubly-linked list is a linked data structure consists of a set of sequentially linked records called nodes. As the name implies, each node contains two fields, called links **(LLINK, RLINK)**, which are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' previous and next links, respectively, point to some kinds of terminator, typically a sentinel node or null, to facilitate traversal of the list. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders. The two node links allow traversal of the list in either direction.

Advantages:

1. The availability of 2 links LLINK and RLINK permit forward and backward movement during the processing of list.
2. The deletion of a node X from the list only for the value X to be known.

Disadvantages:

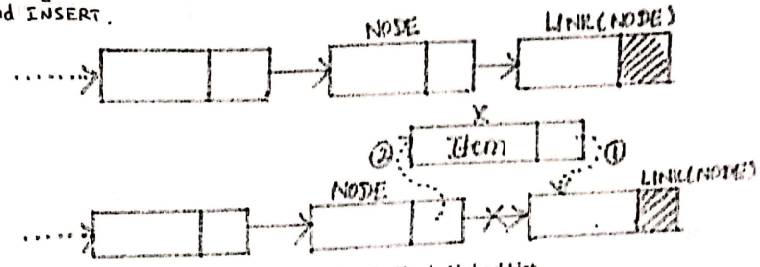
1. Obviously, we are in need of more space to construct doubly linked list as a minimum of 3 parts are needed to construct a node.

While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified. The availability of two links LLINK and RLINK permit forward and backward movement during the processing of list.

Insert Operation in Doubly Linked List

The below procedure is used to insert a new node X whose data part is ITEM to the right of an existing node Y in doubly linked list.

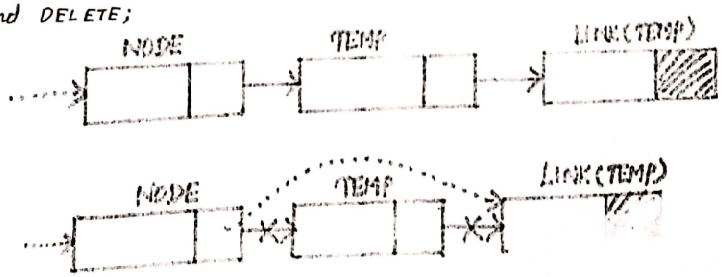
```
Procedure INSERT(START,NODE,ITEM)
call GETNODE(X);
DATA(X) = ITEM;
If (START = NULL) then
    {   LINK(X) = NULL;
        START = X;
    }
else
    {
        LINK(X) = LINK(NODE);
        LINK(NODE) = X;
    }
End INSERT.
```



Delete Operation in Singly Linked List

The below procedure is used to delete a node which is right of NODE in the existing singly linked list. It is possible to delete a node if and only if its previous node is known.

```
Procedure: DELETE(START, NODE)
    IF (START = NULL) then
        Call ABONDON_DELETE
    else
        {
            TEMP = LINK(NODE);
            LINK(NODE) = LINK(TEMP);
            Call RETURN(TEMP);
        }
End DELETE;
```



Primitive Operations on Array

The basic operations on an array are extraction (retrieval) and storing values.

- Extraction refers to getting the value of an element stored in an array as X=Data[i], where we are reading the value present at ith position in the array Data and storing it in Variable X.
- The operation of storing is just opposite to retrieval. Here a value is stored in an element of the array as Data[i]=X, where the value X is stored at ith position in the given array Data.

Applications of array

The following are the two applications of array.

1. SPARSE MATRIX: A matrix is an arrangement of m.n elements in m rows and n columns. The sparse matrix is a matrix with 0's as dominating elements. Eg: In a university of 20000 students say there are 5 electives. If all students are needed to select only 1 elective, that matrix will become sparse.

Usually a matrix consumes a lot of space in memory (ie) a 50x50 matrix needs 2500 locations-If that matrix is sparse, most of the locations will have 0's. So the representation of sparse matrix is developed as a triplet (ie)[i,j,value] used to represent the non-zero value in the sparse matrix.

If A is the given sparse matrix, then B will be the sparse matrix representing A with one row extra than that of A. Usually the first row of B will give some statistics like the total number of rows, total number of columns and total number of non-zero elements in A. The remaining row will tell about the exact row and column position of the non-zero element along with its value at A.

Eg: Consider the following sparse matrix

$$A=\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

The following is the representation of the above sparse matrix

$$B=\begin{bmatrix} 4 & 4 & 3 \\ 1 & 1 & 2 \\ 3 & 3 & 1 \\ 4 & 2 & 3 \end{bmatrix}$$

2. ORDERED LISTS: One of the simplest and useful data objects in computer science is an ordered list or linear list. It may be empty or non-empty. The elements of the list are called as atoms. Variety of

operations like retrieval, insertion, deletion, [illegible] can be done here. The most common [illegible] represent ordered list is one dimensional array. Sequential mapping is established here.

Eg. i) (sun,mon,tue,wed,thu,fri,sat)

ii) (a1,a2,a3,.....an)

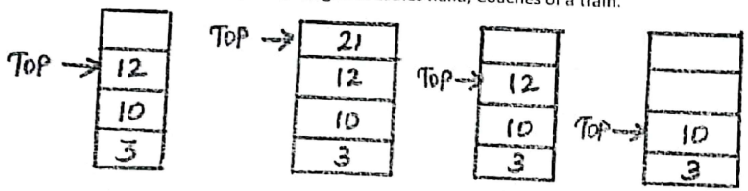
iii) (UNIX, Windows, Linux)

OPERATION	ORIGINAL LIST	RESULTANT LIST
insert a6	a1,a2,a4,a7	a1,a2,a4,a6,a7
delete a4	a1,a2,a4,a6,a7	a1,a2,a6,a7
update a6 to a9	a1,a2,a6,a7	a1,a2,a7,a9

STACKS

Linear Lists allow the insertion and deletion of elements at any position. If we put a restriction on the lists such that the insertion and deletion can occur only at one end, we get an important subclass of lists called as Stack. In stack, the user can add or delete the element only from one end called as Top of the stack. Manipulation is done only at the top item.

Eg: Pile of coins, Pile of plates, Bangles in Ladies hand, Coaches of a train.



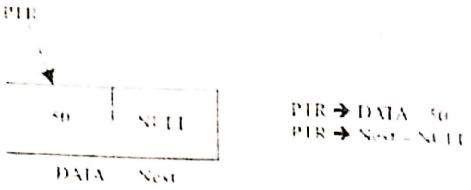
A new element 21 is added at the top of the stack. Similarly the element to be deleted is also 21 the top of the stack. During deletion, the element 12 is removed.

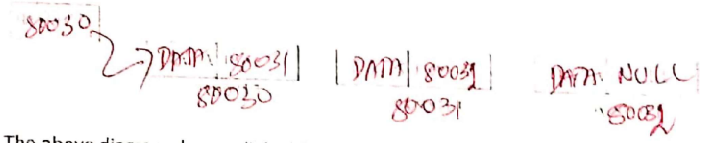
The insertion of the element is called as PUSH and the deletion of the element is called as POP. It needs to be observed that during insertion of elements into the stack it is essential that their identities are specified, whereas for removal no identity need be specified since by virtue of its functionality, the element which occupies the top of stack position is automatically removed. Since the elements inserted into the stack join last and those added last are the first to be removed, the stack belongs to LIFO (Last In First Out) data structure.

NOTE: The stack can be represented as an array STACK[n] i.e., it can store "n" elements. It is therefore becomes essential to issue a warning called STACK-FULL when we try to insert more elements (PUSH) than "n". Similarly when we try to delete an element (POP) from the stack which is already empty, a warning should be issued to indicate STACK-EMPTY. Therefore overflow and underflow should be properly indicated to the users.

SINGLY LINKED LIST

A linked list is a linear collection of specially designed data elements, called nodes linked to another by means of pointers. Each node is divided into two parts: the first part contains information of the element and the second part contains the address of the next node in the linked list. Address part of the node is also called **linked or next field**. As the name implies only one pointer arises from each node.





The above diagram shows a linked list with 3 nodes. Each node is divided into two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node (ie) NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START =NULL if there is no list (i.e.; NULL list or empty list). The data field can be integer, character, a string or even a large record.

Insert Operation in Singly Linked List

The below procedure is used to insert a new node ITEM to the right of an existing node NODE in singly linked list whose first node is pointed by START pointer. It is mandatory to tell the position after or before the insertion has to be performed.

Place the 1st node of B in the (m+1)th node in A. After concatenation A will contain (m+n) nodes.

Advantages of Linked list

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

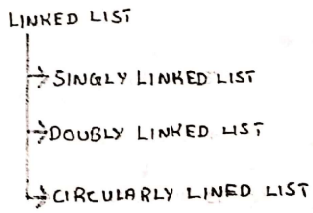
Limitations of Linked List

1. More memory: More memory space is needed even for storing a single data.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming because it purely supports sequential traversal.

Mechanisms needed for linked lists

1. A mechanism to frame chunks of memory into nodes with required number of data items and fields.
2. Determine which nodes are free and which have been allotted for use.
3. Obtain nodes from free storage area [GETNODE(X)].
4. Return or dispose of nodes again to free pool area [RETURN(X)].

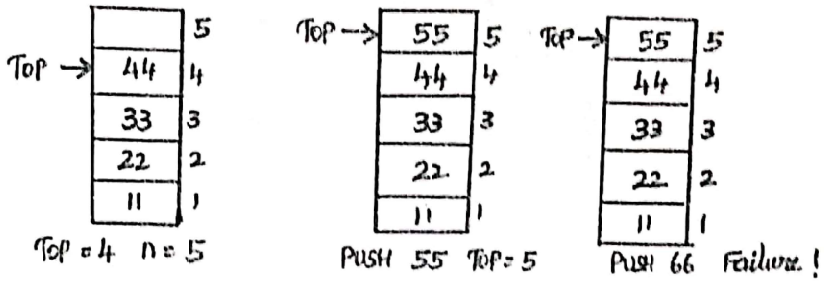
Hence dynamic allocation of memory solves the problem of static allocation of arrays. The classification of linked list done on the basis of the arrangement of nodes is as follows:



Primitive operations on Stack

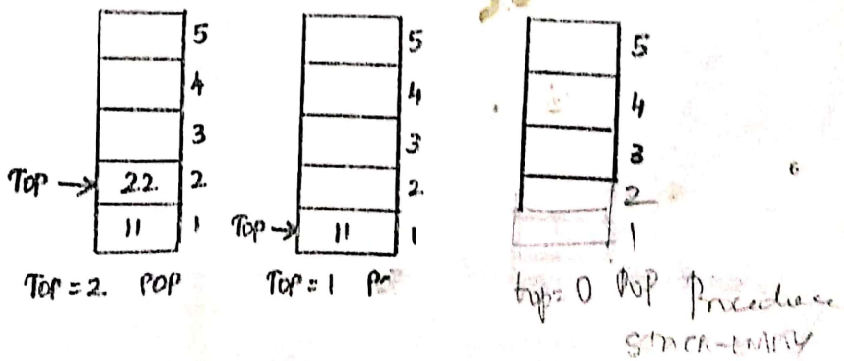
PUSH OPERATION: Let STACK [1:n] be an array representing stack and top be the pointer indicating the top element of the stack. Let top is initialized to 0. "item" is the element to be pushed into the stack and 'n' is the maximum capacity of the stack. STACK FULL is the procedure used to take remedial action for overflow of stack and print respective error message)

```
Procedure PUSH (STACK,n,top,item)
if (top=n) then
call STACK FULL;
else
{
top=top+1;
STACK[top]=item;
}
End PUSH
```



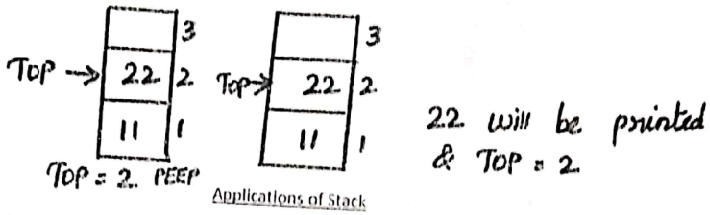
POP OPERATION: In case of POP, no element identity need be specified since by default the element occupying the top of stack position is deleted. (STACK EMPTY is the procedure used to do alternative measure to tackle underflow of stack by printing respective error message.

```
Procedure POP (STACK,top,item)
If (top=0) then
call STACK-EMPTY
else
{
item=STACK[top];
top=top-1;
}
End POP
```



PEEP OPERATION: Peep is the operation used to display the top element of the stack without removing the element from the stack.

Procedure PEEPSTACK(stack,top)
if (top=0) then
call STACK EMPTY
else
Print "The top element= ",STACK[top]
End PEEP



Applications of Stack

Stacks have been used in various computer science applications like recursion, towers of Hanoi, syntax checking, checking of well formedness of parenthesis, conversion of infix to postfix expression, evaluation of postfix expression etc.

a) Infix to Postfix Expression (Notation): An expression can be expressed in three forms as infix, prefix and postfix. They differ from each other in the position of operators. We already know the expression is the combination of operators and operands.

- If the operator is placed in between operands, it is called as Infix expression. Eg: a+b
- If the operator is placed in before operands, it is called as Prefix expression. Eg:+ ab
- If the operator is placed in after operands, it is called as Postfix expression. Eg:ab+

To convert an infix to postfix notation, we need to find out the order of execution of the operators based on their hierarchy. The precedence of operators is

1. () Parenthesis
2. ^ Exponential operator
3. * / Multiplication, Division
4. + - Addition, Subtraction

Eg: Conversion of infix notation A+ (B-C/D)*E to postfix notation

A + (B - C/D) * E

A + (B - CD/) * E

A + BCD/- * E

BCD/-E*

LINKED LISTS

The size of the array is declared and finalized at the compilation itself. We anticipate a higher bound and preserve the memory-but most of the time, we don't use all. This problem of memory under-utilization can be solved if we allocate memory only when there is a need. This is called as dynamic memory allocation. Here the elements are stored in non-contiguous fashion. So to maintain the order of elements, we need to know the address of each element. So a node will store the address of its successive node and the last node will have NULL. These types of data structure is called linked list. They are examples of linear data structures as they permit linear access of element one after other; they are also dynamic data structures as their size can grow or shrink during execution.

Limitations of arrays

1. Inefficient implementation of insertion and deletion.
2. Inefficient use of storage memory.

"Linked list is the natural data abstraction that arises in problems where a data structure maintains a collection of elements without knowing in advance the number of elements"

Operations on Linked List

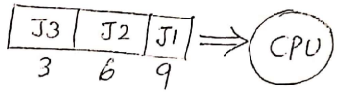
The primitive operations performed on the linked list are as follows

1. **Creation operation** is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
2. **Insertion operation** is used to insert a new node at any specified location in the linked list. A new node may be inserted.
   (a) At the beginning of the linked list
   (b) At the end of the linked list
   (c) At any specified position in between in a linked list
3. **Deletion operation** is used to delete an item (or node) from the linked list. A node may be deleted from the
   (a) Beginning of a linked list
   (b) End of a linked list
   (c) Specified location of the linked list
4. **Traversing** is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.
5. **Searching** is the process of picking up desired element from the list.
6. **Concatenation** is the process of appending the second list to the end of the first list Consider a list A having n nodes and B with m nodes. Then the operation concatenation will

Application of queue

The queue is used in the scheduling of jobs by a processor in a time sharing system. A CPU (processor) associated with memory resources are to be shared by "n" number of users. The sharing of processor and memory resources is done by allotting a definite time slice of the processor's attention on the users and in a round-robin fashion. In such a system, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU. However, to keep track of the jobs initiated by the users, the processor relies on a queue data structure recording the active user.

Eg:-



Consider the above setup of jobs whose burst times are 9, 6, 3 respectively. The time needed by the job to spent at CPU is called as burst time. The time for which CPU should at a particular job is called as time slice / time quanta. The following is the working procedure of round-robin CPU-scheduling algorithm.

Jobs	T1	T2	T3	T4
J1	9	6	3	0
J2	6	3	0	0
J3	3	0	0	0

↓ J3 quits ↓ J2 quits ↓ J1 quits

CPU spends a time quanta of 3 at each job equally.

Procedure INFIX-POSTFIX-CONV(E)

```
x=getnextchar(E);
while x≠"$" do
    Case x:
        : operand: Print "x";
        ): while (top element of stack ≠ "(") do
                Print top element of stack and Pop;
           End while
           Pop "(" from stack;
        Else: while ICP(x)<=ISP(top element of stack) do
                Print top element of stack and Pop;
           End while
           Push x onto stack;
    End case
    X=getnextchar(E);
End while
While stack is not empty do
    Print the top element of stack and pop;
End while
End INFIX-POSTFIX-CONV.
```

The above algorithm is governed by two priority factors-**ISP (In Stack Priority) and ICP (In Coming Priority)**. Apply ISP for the elements that are already residing in stack and ICP for the elements that are to be pushed into stack.

The following table will give the ISP and ICP values for operators and operands in the expression.

OPERATOR	ISP	ICP
)	NIL	NIL
^	3	4
*,/	2	2
+,-	1	1
(	0	4

NOTE: The infix expression is padded with a "$" to signal the end of input. The bottom of the stack is also with inserted with "$" with ISP($)=-1. Pop operators out of the workspace as long as the ICP of the incoming operator is <=to the ISP of the operator already existing in the stack.

Eg: Convert A*(B+C)-G into postfix.

Initialize stack as "$" and attach "$" at the end of the given expression as A*(B+C)-G $.

X	STACK	POSTFIX	REMARKS
A	$	A	Since A is an operand
*	$*	A	Since ICP(*)>ISP($)
(	$*(	A	Since ICP(()>ISP(*)
B	$*(	AB	Since B is an operand
+	$*(+	AB	Since ICP(+)>ISP(()
C	$*(+	ABC	Since C is an operand
)	$*	ABC+	Pop till "(" comes

-	$-	ABC+*	Since ICP(-)<ISP(*) pop "*" and push "-"
G	$-	ABC+*G	Since G is an operand
$	$	ABC+*G-	Since $ is encountered at expression, pop all elements from stack until "$"

Therefore the equivalent postfix notation is ABC+*G-

b) Evaluation of Postfix expression: A postfix expression has the operator immediately following its operands. When an operator is encountered reading the input from left to right, its corresponding operators are those that precede them. This matches the LIFO property of stack. Hence a stack is useful to evaluate a postfix expression.

```
Procedure POSTFIX(E)
Clear the stack
symb=getnextchar(E)
While not end of expression E
        If symb is an operand
                Push the symb into the stack
        Else
        If symb is an operator
        {
                Pop the top element and store it as operand 2;
                Pop the top element and store it as operand 1;
                Compute Result=operand1 operator operand2;
                Push the Result into the stack;
        }
        symb=getnextchar(E)
End while
End POSTFIX
```

EG: Evaluate ABC+*G- where A=2, B=5,C=12,D=18,G=10.

SYMB	OPERAND 1	OPERAND 2	RESULT	STACK
A				A
B				AB
C				ABC
+	B	C	B+C=17	A 17
*	A	17	A*17=34	34
G				34 G
-	34	G	34-10=24	24

Hence the result of the above expression is 24.

Open only 1 queue for all the elements to be manipulated. . Here the space required to build the queue is small. The time required to add a new element into the queue is O(nlogn). But deletion requires only less time since the first element at the head of the queue is deleted easily and so the time complexity is O(1).

Dequeue

A dequeue (Double Ended Queue) is a linear list of elements in which all insertions and deletions are made at the either end of the list. A Dequeue is therefore more general than stack or linear queue and is a sort of **FLIFLO (First In Last In First Out Last Out).**

There are two variants of Dequeue as,

- Dequeue
  - Input-Restricted Dequeue
  - Output Restricted Dequeue

Input-Restricted Dequeue: Here the input/insertion is restricted to the rear end and the deletion has the option of either ends (ie) the element at the front or the rear end of the queue can be deleted.

Output-Restricted Dequeue: Here the output/deletion is restricted to the front end and the insertion has the option of either ends (ie) the element at the front or the rear end of the queue can be deleted.

Dequeue is manipulated by two variables LEFT, RIGHT and their behavior during insertions and deletions are given below. During deletion, if LEFT=RIGHT, it implies that there is only one element in Dequeue and so after deletion, Dequeue becomes empty (ie) LEFT=RIGHT=NIL.

	LEFT	RIGHT
INSERTION	Decremented by 1 mod n	Incremented by 1 mod n
DELETION	Incremented by 1 mod n	Decremented by 1 mod n

Eg: DQ [1:6] be the Dequeue as a circular array.

OPERATION	DEQUEUE (1 2 3 4 5 6)	REMARKS
Initial dequeue	R T S	LEFT=3 RIGHT=5
Insert X at left and Y at right	X R T S Y	LEFT=2 RIGHT=6
Delete twice from right	X R T	LEFT=2 RIGHT=4
Insert G,Q,M at left	G X R T M Q	LEFT=5 RIGHT=4
Insert J at right	G X R T M Q	Dequeue is full and so no insertion
Delete twice from left	G X R T	LEFT=1 RIGHT=4

Eg: CQ[0...3] with n=4

Operation	Circular Queue	Values of FRONT and REAR	Remarks
------	Empty queue	FRONT=REAR=0	-----
Insert a	a	FRONT=0 REAR=1	Inserted successfully
Insert b	a b	FRONT=0 REAR=2	Inserted successfully
Insert c	a b c	FRONT=0 REAR=3	Inserted successfully
Insert d	a b c	FRONT=0 REAR=3	Insertion failure
Delete	b c	FRONT=1 REAR=3	Deleted successfully
Delete	c	FRONT=2 REAR=3	Deleted successfully
Insert d	d c	FRONT=2 REAR=0	Inserted successfully
Insert e	d e c	FRONT=2 REAR=1	Inserted successfully

Priority Queues

A priority queue is a queue in which insertion or deletion of items from any position in the queue is done based on some property called as priority of task. When elements in the queue have the same priority, then the priority queue will behave as an ordinary queue following the principle of FIFO.

Eg: Set of patients waiting for the doctor.

Representation of priority queue

Let x be the element to be inserted/deleted with priority y. The element may be represented as xy.

$a^2 b^1 c^1$

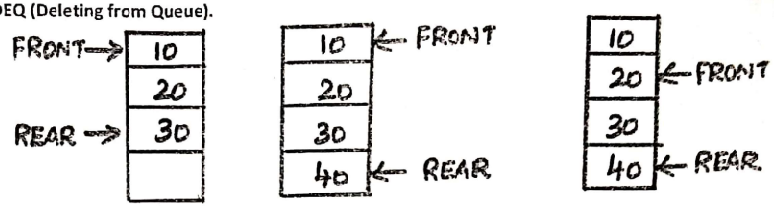
$d^4 a^2 b^1 c^1$

$a^2 b^1 c^1$

There are two types of implementation of priority queue.

1. Open as many queues as per priority. A low priority queue will be operated if and only if all high priority predecessors are empty. Here the space required to build the queue is large. The time required to add a new element into the queue is O(1). But deletion requires more time since more time is taken for checking all the queues.

A queue is an ordered group of elements in which elements are added at one end called as rear end and elements are removed from the other end called as front end. Queue is often used to simulate the real world situations. Eg: Line of students in a fees counter, Line of customers in Bank. A person who stands first leaves the queue first and if any wants to add in the queue, they have to insert themselves at the last. So queue is called as FIFO (First In First Out) data structure. The process of adding a new element is called as **ENQ (Entering into Queue)** while the process of deleting an existing element is called as **DEQ (Deleting from Queue).**



Note: When we perform a DEQ operation, we must ensure that the queue is non-empty. If we try to delete an element from an empty queue, it will be an illegal attempt indicating overflow (QUEUE-FULL). Similarly while performing ENQ operation, we must ensure that the queue is not full. If we try to add an element to an already full queue, it will be illegal attempt indicating underflow (QUEUE-EMPTY).

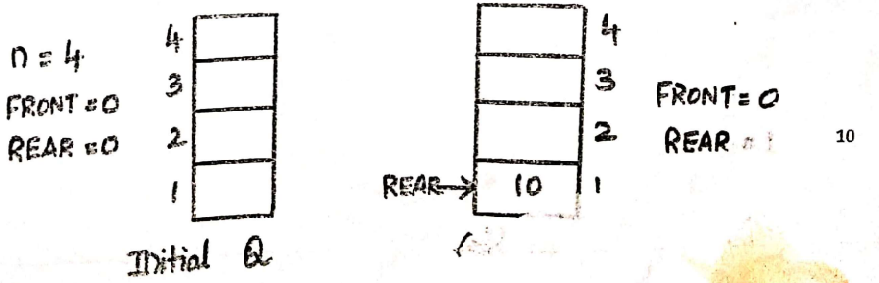
Primitive operations on Queue

Queue is also implemented by an array as Q[1:n] with a maximum capacity of n elements in the queue. Two variables front and rear are used to indicate the first and last positions in the queue respectively.

1.Insert Operation: Let item be the element inserted into the queue Q and the variables front and rear are initialized to zero.

```
Procedure ENQ(Q,n,item,REAR)
if (REAR=n) then
        Call QUEUE_FULL;
else    *
{
        REAR=REAR+1;
        Q[REAR]=item;
}
End ENQ
```

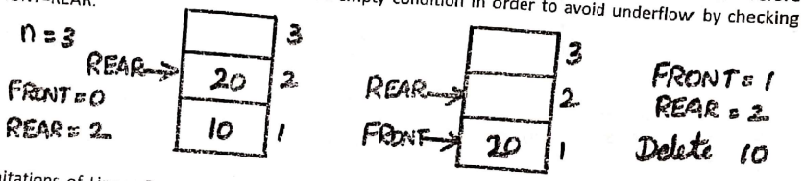
The addition of every new element into the queue increments the REAR variable and before every addition the queue is checked for its full condition in order to avoid overflow of elements.



2.Delete Operation: Though a deletion operation automatically deletes the front element of the queue, the variable item is used as an output variable to store and display the value deleted from queue.

```
Procedure DEQ(Q,FRONT,REAR,item)
if (FRONT=REAR) the
        Call QUEUE-EMPTY;
else
{
        FRONT=FRONT+1;
        item=Q[FRONT];
}
End DEQ
```

The deletion of every new element into the queue increments the FRONT variable and before every deletion the queue is checked for its empty condition in order to avoid underflow by checking FRONT=REAR.



Limitations of Linear Queue: There is a gross limitation of a linear queue since QUEUE-FULL condition checks only REAR=n condition and it does not check whether the queue is physically full or not. But in real time situations, while deletion at rear takes place, automatically the remaining elements in the queue will be moved forward towards front leaving no gap and thereby providing new rooms for new insertions. But moving the data forward will be computationally expensive and may render the process very complex and inefficient.

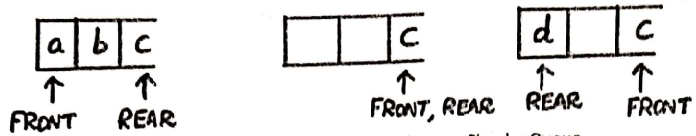
Eg:Q[1:3]

Operation	Queue	Values of FRONT and REAR	Remarks
------	Empty queue	FRONT=REAR=0	----
Insert a	a	FRONT=0 REAR=1	Inserted successfully
Insert b	a b	FRONT=0 REAR=2	Inserted successfully
Insert c	a b c	FRONT=0 REAR=3	Inserted successfully
Insert d	a b c	FRONT=0 REAR=3	Insertion failure
Delete	b c	FRONT=1 REAR=3	Deleted successfully
Delete	c	FRONT=2 REAR=3	Deleted successfully
Insert d	c	FRONT=2 REAR=3	Insertion failure though there are free space at queue

Circular Queue

In order to overcome the limitations of linear queue, we are going on for circular queue. As the name implies it is circular in nature. Here the FRONT and REAR variables are displayed in the circular movement rather than from left to right. A circular queue is also implemented as array and is conceptually viewed as a circular one. The overflow occurs only when all the locations are filled. When the rear pointer reaches the end, it is made to check the first location.

Eg: Let CIRC-Q be a circular queue with a capacity of three elements.



Primitive operations on Circular Queue

The circular movement of FRONT and REAR variables is implemented using mod function which is cyclical in nature. Also the array data structure CQ to implement the queue is declared to be CQ[0:n-1] to facilitate the circular operation of FRONT and REAR variables. As in linear queues, FRONT points to a position which is one less than the actual front of the circular queue. Both FRONT and REAR are initialized to 0. Note that (n-1) is the actual physical capacity of the queue in spite of the declaration 0 to n-1. The following are the procedure to add and delete element from a circular queue respectively.

```
Procedure:INSERT-CIRCQ(CQ,FRONT,REAR,n,item)
REAR=(REAR+1)mod n;
IF(FRONT=REAR) then
    {
        Call CIRCQ-FULL;
        Retract REAR Pointer to the old value;
    }
CQ[REAR] = item;
END INSERT-CIRCQ

Procedure DELETE-CIRCQ(CQ,FRONT,REAR,n,item)
IF(FRONT=REAR) then
    Call CIRCQ-EMPTY;
FRONT=(FRONT+1) mod n;
item=CQ[FRONT];
End (INSERT)-CIRCQ
    DELETE
```