LECTURE NOTES

ON

DATA STRUCTURE

3RD SEMESTER

Sunanda Kumar Sahoo

ASST. PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

In Search of Platonic Excellence

GANDHI INSTITUTE OF TECHNOLOGY AND MANAGEMENT (GITAM)

Affiliated to BPUT & SCTE&VT, Govt. of Odisha

Approved by AICTE, New Delhi

OM.

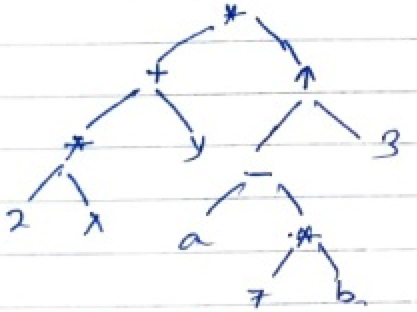
Data are simple values or set of values. Meaningful or processed data is Information. (data with given attribute)

The logical or mathematical model of a particular Organization of data is called datastructure. The datastructure must be rich enough in structure to mirror the actual relationships of the data in real world. On the other hand. the structure should be simple enough that one can effectively process the data when necessary.

One dimensional Array.
Two " "
linked list

Stack.
Queue
Tree
$(2x+y)(a-7b)^3$.

Graph.
~~[illegible]~~
Airline flies only set of cities connected by line.



Primitive Vs Non primitive ⇒

Linear Vs Nonlinear ⇒

Static Vs Dynamic Datastructure ⇒

Arrays.

A linear array is a list of a finite number 'n' of homogeneous data elements.

LOC(LA[K]) = Base (LA) + w (K− Lower bound).

Operations on Array

Traversal. Deletion.
Search. sorting.
Insertion. Merging.

Linear Search.

Algo: (Linear Search) LINEAR(DATA, N, ITEM, LOC)

Here DATA is a linear array with 'N' elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA or sets LOC:=0 if the search is unsuccessful.

```
1. [INSERT ITEM at the end of DATA.]
      Set DATA[N+1] := ITEM.
2. [Initialize Counter] Set LOC := 1.
3. [Search for ITEM]
      Repeat while DATA[LOC] ≠ ITEM:      [LOC ≤ N]
         Set LOC := LOC+1.
      [End of Loop.]
4. [Successful?] If LOC = N+1, then: Set LOC := 0.
5. Exit.
```

Binary Search.

Algo: (Binary Search) BINARY (DATA, LB, UB, ITEM, LOC)
Here DATA is a Sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote respectively, the beginning, end and middle Locations of a Segment of elements of DATA. This algorithm finds the Location LOC of ITEM in DATA or sets LOC = NULL.

```
1. [Initialize Segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2
2. Repeat steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM
3.     If ITEM < DATA[MID], then:
           Set END := MID - 1
       Else:
           Set BEG := MID + 1.
       [End of If Structure.]
4.     Set MID := INT ((BEG + END) / 2).
   [End of Step 2 Loop]
5. If DATA[MID] = ITEM, then:
       Set LOC := MID.
   Else:
       Set LOC := NULL.
   [End of If Structure.]
6. Exit
```

11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

Representation of Linear Arrays in Memory.

⇒ $LOC(LA[K]) = Base(LA) + w(K - \text{Lower bound})$

Multidimensional Array:

Two dimensional array

⇒

Column-major Order :- If the 2-dimensional array is represented in memory column by column, known as column-major order.

Row-major Order :- If 2-dimensional array is arranged row by row in memory, known as row-major order.

11	12	13
21	22	23
31	32	33

~~[illegible]~~ M×N.

column major
11
21
31
12
22
32
31
32
33

row major.
11
12
13
21
22
23
31
32
33

Column-major Order

$$LOC(A[J,K]) = Base(A) + w[M(K-\overset{LB}{1}) + (J-\overset{LB}{1})]$$

Row-major Order.

$$LOC(A[J,K]) = Base(A) + w[N(J-\overset{LB}{1}) + (K-\overset{LB}{1})]$$

M → No. of rows

N → No of column.

3×3.

00	01	02
10	11.	12
20	21	22

$C.M.O = (1,1) = 1000 + 2[3(0) + 0]$

Upper triangle

$$L = \left(\frac{n(n+1)}{2}\right) - \left(\frac{(n-j)(n-j+1)}{2} + (n-k)\right)$$

SPARSE MATRICES

Matrices with a relatively high proportion of zero entries called Sparse matrices.

→ Triangular matrix → Tridiagonal matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 3 & 5 & 6 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & - & - \\ 3 & 4 & 5 & - \\ - & 6 & 7 & 8 \\ - & - & 9 & 10 \end{pmatrix}$$

Lower Upper

B will contain $1+2+3+\cdots+n = \frac{n(n+1)}{2}$ elements

$B[L] = a_{JK}$

no. of elements upto $j-1$ rows

$= 1+2+3+\cdots+j-1 = \frac{(j-1)j}{2}$

$$L = \boxed{\frac{j(j-1)}{2} + K}$$

Tridiagonal Matrix.

There are $3(J-2)+2$ elements above $A[J,K]$ and $K-J+1$ elements to the left of $A[J,K]$.

Hence $L = [3(j-2)+2] + [K-j+1] + 1$

$= 2j+K-2.$

Primitive DataStructures :-

These are basic structures and are directly operated upon by the machine instructions. Integer, Float, character, pointer etc. are primitive datastructure.

Non-primitive DataStructures :- These are more Sophisticated datastructures. These are derived from the primitive datastructures. The non-primitive datastructures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (diff. type) data items.

eg: Arrays, Lists, Stack, Queue, Tree, Graph.

Algo: (Bubble Sort) BUBBLE (DATA, N)

```
Here DATA is an array with N elements. this algorithm Sorts
the elements in DATA.
1. Repeat steps 2 and 3 for K = 1 to N-1
2.        Set PTR := 1. [Initializes pass pointer PTR.]
3.        Repeat while PTR ≤ N-K: [Executes pass.]
              (a) If DATA[PTR] > DATA[PTR+1], then:
                    Interchange DATA[PTR] and DATA[PTR+1].
                    [End of If structure.]
              (b) Set PTR := PTR+1.
          [End of inner Loop.]
        [End of step 1 outer Loop.]
4. Exit.
```

Algo: (Insertion Sort) INSERTION(A,N)

```
     This algorithm Sorts the array A with N elements.
* [1. Set A[0] := -∞. [Initializes sentinel element.]
  1. Repeat steps 2 to 4 for K = 2,3. ...., N:
  2.      Set TEMP := A[K] and PTR := K-1
  3.      Repeat while TEMP < A[PTR] and PTR ≥ 1
              a) Set A[PTR+1] := A[PTR]. [Moves element forward]
              b) Set PTR := PTR-1.
          [End of Loop.]
  4.  Set A[PTR+1] := TEMP. [Insert element in proper Place.]
      [End of Step 2 Loop.]
  5.  Return/Exit
```

77, 33, 44, 11, 88, 22, 66, 55

33, 77, 44, 11, 88, 22, 66, 55	K = 2
33, 44, 77, 11, 88, 22, 66, 55	K = 3
11, 33, 44, 77, 88, 22, 66, 55	K = 4

The insertion Sort algorithm Scans the Array A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted Subarray A[1], A[2], ― A[K-1].

Storage of Sparse Matrix:

$$\begin{bmatrix} 7 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{c|ccc} & \text{No. Rows} & \text{No. col} & \text{No. non-zero} \\ & 6 & 6 & 8 \\ i= & 0 & 0 & 7 \\ & 0 & 3 & 1 \\ & 0 & 5 & 2 \\ & 1 & 1 & 1 \\ & 1 & 2 & 9 \\ & 2 & 3 & 7 \\ & 4 & 0 & 8 \\ 8 & 5 & 2 & 3 \end{array}$$

Transpose of a Sparse Matrix.

```
ALgo: (Transpose Matrix) TRANSMAT( MAT1[ ][3], MAT2[ ][3
```

$$\begin{matrix} 6 & 6 & 8 \\ 0 & 0 & 7 \\ 0 & 4 & 8 \\ 1 & 1 & 1 \\ 2 & 1 & 9 \end{matrix}$$

```
m = MAT1[1][1]
n = MAT1[1][2]
t = MAT1[1][3].
MAT2[1][1] = n.
MAT2[1][2] = m
MAT2[1][3] = t;
K=2;
for (j=1 ; j<=n ; j++)
      for (i=2; i<=t+1 ; i++)
            if( MAT1[i][2] == j)
            {
               MAT2[k][1] = MAT1[i][2]
               MAT2[k][2] = MAT1[i][1]
               MAT2[k][3] = MAT1[i][3]
               K++ ;
            }.
```

Storage of Sparse Matrix.

$$\begin{bmatrix} 7 & 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{array}{c|ccc} & \text{No. Rows} & \text{No. col} & \text{No. of non-zero} \\ & 6 & 6 & 8 \\ i= & 0 & 0 & 7 \\ & 0 & 3 & 1 \\ & 0 & 5 & 2 \\ & 1 & 1 & 1 \\ & 1 & 2 & 9 \\ & 2 & 3 & 7 \\ & 4 & 0 & 8 \\ 8 & 5 & 2 & 3 \end{array}$$

Transpose of a Sparse Matrix.

Algo: (Transpose Matrix) TRANSMAT(MAT1[ ][3], MAT2[ ][3]

$$\begin{array}{ccc} 6 & 6 & 8 \\ 0 & 0 & 7 \\ 0 & 4 & 8 \\ 1 & 1 & 1 \\ 2 & 1 & 9 \end{array}$$

```
m = MAT1[1][1].
n = MAT1[1][2]
t = MAT1[1][3].
MAT2[1][1] = n.
MAT2[1][2] = m
MAT2[1][3] = t ;
K = 2;
for (j = 1 ; j <= n ; j++)
        for (i = 2; i <= t+1 ; i++)
            if ( MAT1 [i] [2] == j)
            {
                MAT2[k][1] = MAT1[i][2]
                MAT2[k][2] = MAT1[i][1]
                MAT2[k][3] = MAT1[i][3]
                K++ ;
            }.
```

Selection Sort

~~Procedures MIN (A, K, N, LOC)~~

Algorithm : (Selection Sort) SELECTION (A, N)

This algorithm sorts an array A with N element. It first find the smallest element in the array and put it in it's place.

```
1. Repeat steps 2 to 4 for K = 1, 2, ....., N-1
2.   Set MIN := A[K] and LOC := K
3.   Repeat for J = K+1, K+2, ..., N :
         If MIN > A[J], then: Set MIN := A[J] ~~and~~
                                  ~~LOC := A[J]~~ and LOC := J.
     [end of Loop]
4.   [Interchange A[K] and A[LOC].]
         Set TEMP := A[K], A[K] := A[LOC] and A[LOC] := TEMP.
     [End of Step 1 loop.]
5. Exit
```

77, 33, 44, 11, 88, 22, 66, 55

Selection Sort first finds the smallest element in the list and put it in first place. Then finds the second smallest element in the list and put it in the second position and so on.

```
     n = 1;
for(i=0; i<row; i++)
     for(j=0; j<col; j++)
     { scanf("%d", ~~mat[i][j]~~ &item);
        if(item != 0)
        { spa[n][0] = i;
          spa[n][1] = j;
          spa[n][2] = item;
                    n++; }
     }
  spa[0][0] = row;
  spa[0][1] = col;
  spa[0][2] = n-1;
```

```
void transpose(int mat1[][3], mat2[][3])
{ int row = mat1[0][0];
  int col = mat1[0][1];
  int n = mat1[0][2]; int i, j, k=1;
 for( j=0; j<col; j++)
    for(i=1; i<=n; i++)
       if( mat1[i][1] == j )
       { mat2[k][0] = mat1[i][1];
         mat2[k][1] = mat1[i][0];
         mat2[k][2] = mat1[i][2];
           k++;
       }
```

STACKS

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

Operations on STACK.

a) Push : Inserts elements into stack.

b) Pop : Delete " " " .

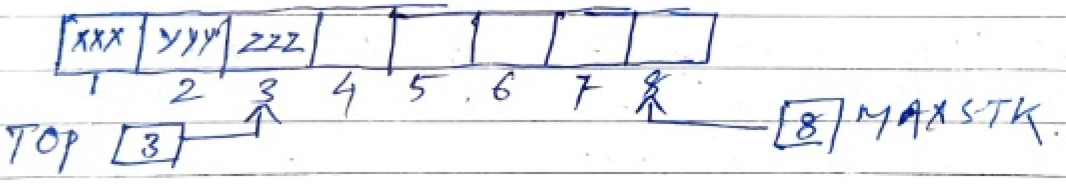
Array Representation of STACK:-

TOP:- a pointer variable which points to the top element of stack.

MAXSTK :- Maximum no. of elements can be held by the stack.

If TOP:= 0 or NULL → Stack is empty.

TOP := MAXSTK → Stack is full.



```
Procedure PUSH (STACK, TOP, MAXSTK, ITEM)
    This procedure pushes an ITEM onto a stack.
  1. [Stack already filled?]
     If TOP = MAXSTK, then: Print: OVERFLOW, and Return
  2. Set TOP:= TOP+1. [Increases TOP by 1.]
  3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position]
  4. Return.
```

```
Procedure: POP( STACK, TOP, ITEM)
    This procedure deletes the top element of STACK and
    assigns it to ~~the~~e variable ITEM.
    1. [Stack has an item to be removed?]
           If TOP = 0, then: Print: UNDERFLOW, and Return.
    2. Set ITEM := STACK[TOP]. [Assigns TOP element to
                                                     ITEM.]
    3. Set TOP := TOP - 1
    4. Return.
```

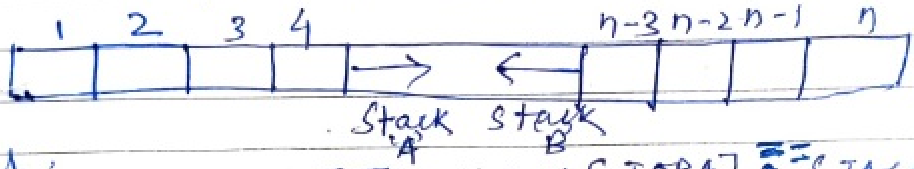
Minimizing Overflow.

overflow → ~~Dess no~~ Small Stacksize

Underflow → More Stack Size.

To minimize the ~~so~~ overflow, we can take 2 stack in a single array.

A → n1 , B → n2 . n = n1 + n2.



Push: If ~~TOP~~ STACK[TOPA] = STACK[TOPB] - 1
then full.

POP: If ~~STACK~~ TOPA = 0 and TOPB > n
then underflow.

As the push operation push the new element on top of old element, Stack also known as pushdown List.

Arithmatic Expression [Infix, Prefix, Postfix]

Infix → operand operator operand.

Prefix → polish notation → operator operand operand.

Postfix → Reverse polish → operand operand operator.

Infix.	Polish. Prefix.	Reverse polish. Post fix.
A+B	+AB	AB+
C-D	-CD	CD-
E*F	*EF	EF*
(A+B)*C	*+ABC	AB+C*
A+(B*C)	+A*BC	A(BC*)+
(A+B)/(C-D)	/+AB-CD.	AB+CD-/

Advantage :- One never need parantheses to evaluate Reverse polish expre

For our purpose we only assume the below three levels of precedance.

Highest :- Exponentiation (↑)

Next highest :- Multiplication (*) and division (/)

Lowest :- Addition (+) and Subtraction (-).

no unary operators and Evaluation as from left to right even for exponentiations.

5*(6+2)-12/4 | -,*5,+,6,2,/,12,4

post fix 5(62+)* -12,4/

5,6,2+*124/-

A + (B * C - (D / E ↑ F) * G) * H

A + (BC* - (D/EF↑) * G) * H

A + (BC* - (DEF↑/) * G) * H

A + (BC* DEF↑/G*-) * H

A + (BC* DEF↑/G*- H*)

ABC*DEF↑/G*-H*+

12, 7, 3, -, /, 2, 1, 5, +, *, +

12, [7-3], /, 2, 1, 5, +, *, +

(12/[7-3]), 2, [1+5] * +

(12/(7-3)) + 2 * (1+5)

Evaluation of a postfix Expression

Suppose P is an arithmatic 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 act as sentin
2. Scan P from left to right and repeat steps 3 and 4 for
   element of P until the sentinel ")" is encountered.
3.     If an operand is encountered, put it on STACK
4.     If an operator ⊗ is encountered, then:
       a) Remove the two top elements of STACK,
          where A is the top element and B is the
          next-to-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 on STAC
6. Exit.
```

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

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)	)	End

Transforming Infix Expressions into postfix Exp.

```
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 until the STACK is empty.
3.     If an operand is encountered, add it to P.
4.     If a left parenthesis is encountered, push it onto STACK.
5.     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 ⊗
         b) Add ⊗ to STACK.
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, ~~[illegible]~~.
        b) ~~Repeatedly pop from~~ Remove the left
           parenthesis. [Do not add the left parenthesis
           to P.]
        [End of If Structure].
    [End of step 2 Loop.]
7. Exit.
```

Q : A + (B * C - (D / E ↑ F) * G) * H) ← Added in Algorithm.

Symbol Scanned	Stack	Symbol Scanned	Stack	Symbol Scanned	Stack
A	('A' Added to 'P'	(	(+ (- (	G	(+ (- * Added to 'P'
+	(+	D	(+ (- (Added to 'P'	)	(+ *, - added to 'P'
(	(+ (	/	(+ (- (/	*	(+ *
B	(+ (Added to 'P'	E	(+ (- (/ Added to 'P'	H	(+ * added to 'P'
*	(+ (*	↑	(+ (- (/ ↑	)	Empty *, + added to 'P'.
C	(+ (* Added to 'P'	F → Added to 'P'			
-	(+ (- '*' added to 'P'	)	(+ (- ↑, / to 'P'		
		*	(+ (- *		

P :- A, B, C, *, D, E, F, ↑, /, G, *, - H + *

QuickSort, An Application of STACKS

(QuickSort) This algorithm Sorts an array A with N elements.

```
1. [Initialize.] TOP := NULL.
2. [Push boundary values of A onto stacks when A has 2
   or more elements]
      If N>1, then: TOP:=TOP+1, LOWER[1]:=1, UPPER[1]:=N.
3. Repeat steps 4 to 7 while TOP ≠ NULL.
4.    [Pop sublist from stacks.]
      Set BEG:= LOWER[TOP], END:= UPPER[TOP],
         TOP := TOP-1.
5.    Call QUICK (A,N, BEG, END, LOC).
6.    [Push left sublist onto stacks when it has 2 or more elements]
      If BEG < LOC-1, then:
         TOP := TOP+1, LOWER[TOP] := BEG,
          UPPER[TOP] = LOC-1,
        [End of If Structure.]
7.    [Push right sublist onto stacks when it has 2 or more elements.
        If LOC+1 < END, then:
         TOP:= TOP+1, LOWER[TOP] := LOC+1,
          UPPER[TOP] := END.
      [End of If structure].
      [End of step 3 Loop]
8. Exit.
```

QUICK (A,N, BEG, END, LOC)

[Here A is an array with N elements, Parameters BEG and END Contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will Contain the boundary values of the list of elements that have not been scanned.]

```
1. [Initialize.] Set LEFT:= BEG, RIGHT:= END and LOC := BEG.
```

```
2. [Scan from right to left.]
   a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT
          RIGHT := RIGHT-1
        [End of loop]
   b) If LOC = RIGHT, then: Return.
   c) If A[LOC] > A[RIGHT], then:
        i) [Interchange A[LOC] and A[RIGHT]]
         TEMP := A[LOC], A[LOC] := A[RIGHT]
          A[RIGHT] := TEMP.
        ii) ~~A[RIGHT] := TEMP~~ set LOC := RIGHT.
        iii) Go to step 3.
3. [Scan from left to right.]
   a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC
             LEFT := LEFT+1.
   b) If LOC = LEFT, then: Return.
   c) If A[LEFT] > A[LOC], then
        i) [Interchange A[LEFT] and A[LOC].]
           TEMP := A[LOC], A[LOC] := A[LEFT]
            A[LEFT] := TEMP.
        ii) Set LOC := LEFT.
        iii) Go to step 2.
```

Quick Sort is an algorithm of the divide and Conquer type.

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

22, 33, 11, 55, 77, 90, 40 60, 99, 44, 88, 66

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

```
void quick_sort (int a[], int l,
                          int h)
{  int temp, key, low, high;
   low = l ; high = h;
   key = a[(low+high)/2];
   do
   {   while (key > a[low])
             low++;
       while (key < a[high])
             high--;
     if (low <= high)
     {   temp = a[low];
         a[low++] = a[high];
         a[high--] = temp;
     }
   } while (low <= high);
   if (l < high)
   quickSort (a, l, high);
    if (low < h)
      quick_sort (a, low, h);
}
```

Circular

QUEUES

A Queue is a linear list of elements in which deletion can take place only at one end ~~[illegible]~~ called front, and insertion can take place only at the other end, called the rear. 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.

Representation of Queues

Unless untill specified Queues are represented by linear Array in computer, with two pointer variable FRONT & REAR.

1	2	3	4		
A	B	C	D		F = 1 R = 4
	B	C	D		F = 2, R = 4

Item Insterted REAR = REAR + 1.

Item Deleted FRONT = FRONT + 1.

```
QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR+1,
   then: write OVERFLOW, and Return.
2. [Find new value of REAR].
   If FRONT := NULL, then
       Set FRONT := 1 and REAR := 1
   Else if REAR = N, then:
       Set REAR := 1
   Else: Set REAR := REAR + 1
3. Set QUEUE [REAR] := ITEM
4. Return.
```

```
QDELETE (QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a
queue and assigns it to the variable ITEM.
1. [Queue already empty?]
   If FRONT := NULL, then: write: UNDERFLOW
                                and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
   If FRONT = REAR, then: [Queue has only one element to start]
      Set FRONT := NULLL
                  and REAR := NULL
   Else if FRONT = N, then
           Set FRONT := 1.
   Else: Set FRONT := FRONT + 1
4. Return.
```

A	B	C			F=1, R=3	A, B, C. inserted.
	B	C			F=2, R=3	A deleted.
	B	C	D	E	F=2, R=5	D & E inserted.
			D	E	F=4, R=5	B & C deleted.
F			D	E	F=4, R=1	F inserted.
F				E	F=5, R=1	D deleted.
						G & H inserted.
						E deleted.
						F deleted.
						K inserted.
						G & H deleted.
						K deleted.

DEQUES.

A deque ~~[illegible]~~ is a Linear list in which elements can be added or removed at either end but not in the middle.

There are two variations of a deque.

✓ Input Restricted deque :- a deque which allows insertions at only one end of the list but allows deletions at both ends of the list.

✓ Output Restricted deque :- a deque which allows deletions at only one end of the list but allows insertions at both end of the list.

LEFT = 4.
RIGHT = 7

1	2	3	4	5	6	7	8
			AAA	BBB	CCC	DD	

LEFT = 7
RIGHT = 2.

y	z					w	x

Priority Queues.

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 rule.

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.

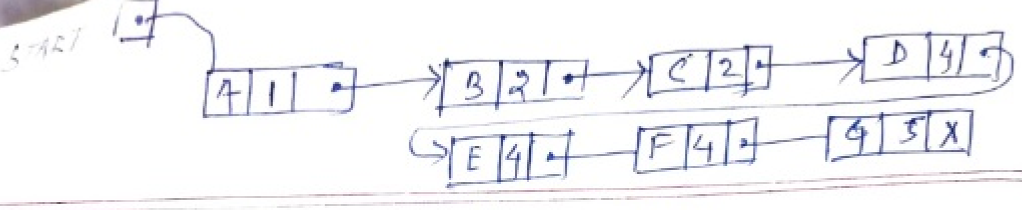
One-way List Representation of a priority Queue.

✓ an information field INFO
a priority number PRN.
a link number LINK
} Each node.

✓ A node X precedes a node Y in the list

1) when X has higher priority than Y or

2) when both have the same priority but X was added to the list before Y.



AVAIL $\boxed{2}$ → 2

STATRT $\boxed{5}$ → 5

	INFO	PRN	LINK
1	BBB	2	6
2			7
3	DDD	4	4
4	EEE	4	9
5	AAA	1	1
6	CCC	2	3
7			10
8	GGG	5	0
9	FFF	4	8
10			11
11			12
12			0

Array Representation of a Priority Queue

Use a Separate Queue for each level of priority. Each queue will appear in it's own Circular array and must have it's own pair of pointers. If each queue having the Same Size then Can be represented by matrix.

	F	R
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	22	3	4	5	6
1		AA				
2	BB	CC	XX			
3						
4	FF				DD	EE
5				GG		

No. of Elements in a Queue

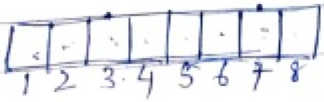
$F \leq R$ = $NE = R - F + 1$

$F > R$ = $N - (F - R - 1) = N + R - F + 1$

$NE = (R - F + 1) \bmod N$

$F = 3,\ R = 7 = 7 - 3 + 1 \bmod 8$

$= 5 \bmod 8 = 5$

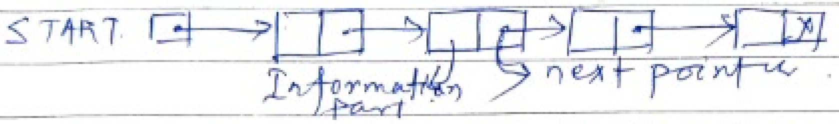


$R = 3,\ F = 7 = 3 - 7 + 1 \bmod 8$

$= -3 \bmod 8$

LinkedList.

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. 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 nextpointer field, contains the address of the next node in the list.



Next pointer of last node will contain NULL

START 9

	INFO	LINK
1		
2		
3	O	6
4	T	0
5		
6	□	11
7	X	10
8		
9	N	3
10	I	4
11	E	7
12		

START=[9]	INFO[9] = N.
LINK[9]=3	INFO[3] = O
LINK[3] = 6	INFO[6] = _
LINK[6] = 11	INFO[11] = E.
LINK[11] = 7	INFO[7] = X
LINK[7] = 10	INFO[10] = I
LINK[10] = 4	INFO[4] = T
	LINK[4] = 0 / NULL.

Algo: (Traversing a linked list)

```
1. Set PTR := START. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]
5. Exit.
```

Algorithm 5.2 SEARCH (INFO, LINK, START, ITEM, LOC)
LIST is a Linked list in memory. This algorithm finds the Location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

```
1. Set PTR := START.
2. Repeat step 3 while PTR ≠ NULL:
3.     If ITEM = INFO[PTR], then:
           Set LOC := PTR, and Exit.
       Else
           Set PTR := LINK[PTR].
4.   Set LOC := NULL.
5. Exit.
```

Self Referential Structure.

A Structure which contains a pointer variables of it's own type known as Self Referential Structure.

```
# include <stdio.h>
  typedef struct node
          { char info;
            struct node *next_node;
          } List_node;

  main()
  { List_node *list_of_char, n1, n2, n3;
      n1.info = 'x';
      n1->next_node = NULL;
      n2.info = 'y';
      n2->next_node = &n1;
      n3.info = 'z';
      n3->next_node = &n2;
      list_of_char = &n3;
printf ("node3 = %.C %.X, node 2 = %.C %.X\n", n3.info,
        n3->next_node, n2.info, n2->next_node);
Printf (" node1 = %.C %.X, header addr = %.X \n",
        n1.info, n1->next_node, list_of_char);
  }
```

Dynamic Memory allocation.

```
      ptr = (Cast-type*) malloc (byte-size)
      ptr = (Cast-type*) calloc( n, elem-size)
```

Contiguous Space for n blocks of size elem-size bytes is alloca

```
# include <stdio.h>
# include <malloc.h>
 Struct link
  { int info;
    Struct link *next;
  };
void CreateList (structlink*);
void display (struct link*);
void main()
  { struct link *node;
     clrscr();
    node = (struct link*) malloc (sizeof (struct link));
    if (node == NULL)
    { printf("\nout of memory space"); exit(0);{
  createlist(node); display(node);
  }
Void Createlist ( Struct link *node).
 {      char ch;
        int i=1;
        Printf ("\n Enter the Value for %d node:", i);
        Scanf ("%d", &node->info);
        node->next = NULL;
        i++;
      Printf(("\n Press 'n' to Quit, any other to continu
        fflush (stdin);
        ch = getchar();
      while ( ch != 'n')
   {    node->next =(struct link*) malloc (sizeof (struc
                                                    link))
        if (node->next == NULL)
        { printf ("\n out of memory Space");
          exit(0);
          }
    node = node -> next;
    printf("\n Enter the value for %d node:", i);
    Scanf("%d", &node -> info);
     node -> next = NULL;
      i++;
     printf("\n press 'n' to quit any other key to continue
      fflush(stdin);
       ch = getchar();
   }.
}
void display (struct link *node)
      {  struct link *ptr;
         printf ("\n Value of nodes in the list are
            are follows : \n\n");
         ptr = node;
      while (ptr != NULL)
            {  printf("%d", node -> info);
                 node = node -> next;
            }.
     }.
```

Advantages of linked lists.

→ Linked list are dynamic in nature, i.e they can grow or shrink during execution of a program.

→ Efficient memory utilization : Here memory is not pre-allocated. Memory is allocated whenever it is required and it is deallocated when it is no longer needed.

→ Insertion and deletions are easier and efficient : Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

→ Many complex applications can be easily carried out with linked lists.

Disadvantages

→ For a node we need extra memory spaces for pointers.

→ Access to an arbitary data item is little bit cumbersome and also time-consuming.

Types of Linked list.

✓ Singly-Link List ✓ Circular link list

✓ Doubly link list. ✓ Circular doubly linked list

Operation on Linked list.

→ Creation. → Insertion → Deletion → Traversing

→ Searching → Concatenation → Display.

INSERTION.

Insertion at beginning.

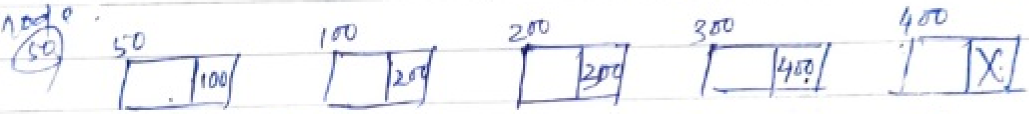
```
void insertatbegin(struct link *node, int item)
{   struct link * ptr;
    ptr = (struct link*) malloc(sizeof(struct link));
    ptr → info = item;
      [illegible] ptr → next = ~~node~~ start;
          ~~node~~ start = ptr;
}
```

Insertion at end.

```
void insertatend(struct link *node, int item)
 { struct link *ptr, *loc;
   ptr = (struct link*) malloc(sizeof(struct link));
    ptr → info = item;
     ptr → next = NULL;
    if ([illegible] start == NULL)
       { [illegible] ~~node~~ start = ptr;}
   else {   loc = ~~node~~ start;
          while(loc → next != NULL)
            { loc = loc → next;}.
              loc → next = ptr;
   }    }
```

Insert after specified location

```
→ void insert_loc(struct link* node, int item, int loc)
  { struct link* ptr, *temp;
    int k;
                          start
    for ( k=1, temp = node ; k<loc ; k++)
   {   ~~temp = temp -> next;~~
       if ( temp == NULL )
        { printf ("There are less no. of nodes
                    than %d", loc); return }
       temp = temp -> next;
   }  ptr = (struct link*) malloc(sizeof(struct link));
      ptr -> info = item;   ptr -> next = temp -> next;
                            temp -> next = ptr; }
```



```
loc = 5     1   temp = 100    3 = 300
            2   temp = 200 ,  4 = 400
                              5 = NULL
```

```
void Rev_List ( )
  {  node* p, *c, *n;
     if (start == NULL)
        return;
     if (start -> next == NULL)
        return;
   p = start;   c = start -> next;
      p -> next = NULL;
   while (  c -> next != NULL)
        {   n = c -> next;
            c -> next = p;
             p = c;
             c = n;
        }
      c -> next = p;
      start = c;
  }
```

Deletion of nodes in Single linkedlist

Deleting the first node of the linked list.

```
Void delete_beg (Struct link *node)
                                   struct link *ptr;
  {   if ( node == NULL)
          return;
      else                ptr = node;
        {  node = node ->next; } free(ptr)
  }
```

Deleting the Last Node.

```
 void delete_end( Struct link *node)
  {   struct link * ptr, * temp;
     if ( node ==NULL)      | else if (node ->next == NULL)
       {return;}            |   {  ptr = node;
     else                   |      node = NULL;
                            |      free(ptr);
       {   ptr = node;  }
          while( ptr ->next != NULL)
            {  temp = ptr;
                  ptr = ptr ->next;
            }
        temp ->next = NULL;
          free(ptr)
       }
```

Deleting the node from Specified Position

```
Void delete-spe-loc(struct link* node, int Loc)
     { Struct link* ptr ; *temp;
         int i ;
         if ( node == NULL)
           {  printf (empty list) ; return ;
           }
        else
         {   ptr = node;
            for ( i=1 ; i< LOC ; i++)
                {  ptr = ptr -> next;
                         if ( ptr == NULL)
                   {printf( ,        );
                     return ;
                   }
                }
          temp = ptr -> next ;
            ptr ->next = ptr -> next -> next ;
          free ( temp) ;
```

Circular linklist.

The ~~last node~~ link part of last node will contain the first nodes address rather than NULL;

It will be better if we maintain two pointer FIRST & LAST to point to the first node and last node.

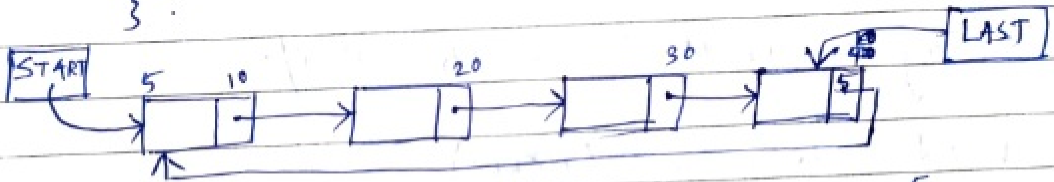
```
typedef Struct Node
        { int num;
          Struct node *next;
        } node;
    node *start = NULL;
    node *last = NULL;

void main()
  {   node *temp;
      temp = (node *) malloc (sizeof(node));
      ~~start = temp;~~
      ~~last = temp;~~
      ~~start -> next = start;~~
```

Inserting a node at the beginning.

```
  void insertfirst ( )
    {   node *ptr;
        ptr = (node *) malloc (size of (node));
        printf ("Enter the no");
        scanf ("%d", &ptr->num);
        if (start == NULL)
          {  ptr->next = ptr;
             start = ptr;
             last = ptr;
          }
        else
           { ptr->next = start;
             start = ptr;
             last->next = ptr;
           }
    }
```

Inserting node at the end

```
  void insertlast ( )
   {    node *ptr;
    ptr = (node *) malloc( sizeof (node));
    Printf (" Enter the number");
    Scanf ("%d", &p → num);
   if (Start == NULL)
       {  ptr → next = ptr;
          Start = last = ptr;
       }
    else
       {  last → next = ptr;
          last = ptr;
          Ptr → next = Start;  / last → next = Start
       }
   }.
```



Inserting in the middle.

```
  for (i = 1 ; i < loc-1; i++)
     {  ptr = ptr → next;
        if (ptr == start)
          exit
     }
  ~~if → ptr = start~~
           if ( ptr → next == start)
              {  ptr → next = temp
                 temp → next = start
                 last = temp;
              }
          else
             {  temp → next = ptr → next
                ptr → next = temp
             }
  }
```

5

1 - 4

1 - ptr = 10

2 - ptr = 20

3 - ptr = 30

~~40 - ptr = 5~~

Deleting node from Beginning.

```
void delete_first( )
  {   node  *p;
    if( start == NULL)
      { printf("List empty");
      }                          start
    else  if( start → next == NULL) { start = last = NUL)
      { p = start;
        start = start → next;
        last → next = start;  free(p);
      }
  }
```

Deleting a node from the end.

```
void  del_last (    )
   {  node  *p, *q;
     if( start = NULL)
        { print ("list empty"); }
     else
        {  p = start;
         if ( P → next == NULL)
           {  start = NULL;
              last = NULL;
              free(P);
           }
         else
           { while ( P → next != last)
               { p
                   p = p → next;
               }
     printf(  ,  -  ,       )
       q = p → next;
      p → next = start;
      last = p;
       free (q); }
```

Deleting node from Beginning.

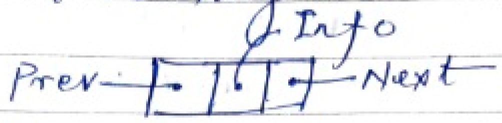
```
void delete_first( )
   {   node  *p;
     if ( start == NULL)
        { printf("List empty");
       }                                    Start
    else   if( start -> next == NULL)){ start
                                           = last
                                           = NUL)
       {  p = start;
          start = start -> next;
          last -> next = start;  free(p);
       }
   }
```

Deleting a node from the end.

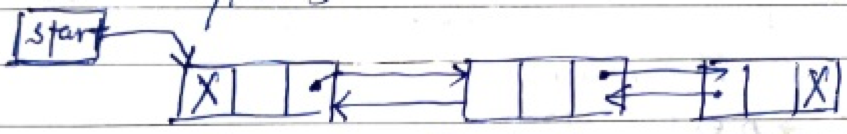
```
    void   del_last ( )
       {  node  *p, *q;
        if ( start = NULL)
           { print ("list empty"); }
        else
          {    p = start;
           if ( p -> next == NULL)
              {  start = NULL;
                 last = NULL;
                 free(P);
              }
           else
              {  while ( p -> next != last)
                    {
                         p = p -> next;
                    }
        printf (  ,  ,  ,        )
          q = p -> next;
         p -> next = start;
         last = p;
          free (q); }
```

Doubly linked list

A doubly linked list is one in which all nodes are linked together by two links which help in accessing both the successor node and predecessor node from the given node position. It provides bi-directional traversing.



```
struct node
  { int num;
    struct node *prev;
    struct node *next;
  };   typedef struct node NODE;
```



```
NODE * start = NULL;
```

Inserting a node at beginning

```
void insert_beg( int item)
  {   NODE *ptr;
     ptr = (NODE *) malloc (sizeof(NODE));
     ptr -> num = item;
      if( start == NULL)
       {  ptr -> prev = ptr -> next = NULL;
              start = ptr;
       }
      else {   ptr -> prev = NULL;
               ptr -> next = start;
               start -> prev = ptr;
               start = ptr;
           }
  }
```

Inserting a node at the end.

```
void insert_end (int item)
{ NODE *ptr, temp;
   ptr = (NODE *) malloc (sizeof(NODE));
  ptr -> num = item;
  if ( ~~tail = = (node *)~~ start == NULL)
   { ptr -> prev = ptr -> next = NULL;
      start = ptr;
   }
 else
    { ptr -> next = NULL;
      temp = start;
      while ( temp -> next != NULL)
            temp = temp -> next;
      ptr -> prev = temp;
      temp -> next = ptr;
    }
}
```

Deleting a node from the beginning

```
void delete_beg( )
  {    NODE *ptr;
     if ( start == NULL)
       {  ...           , return() }
     else if ( start -> next == NULL)
       {      ptr = start;
             start = NULL;
             free(ptr)
        }
        else
         {   ptr = start;
            start = start -> next;
            start -> prev = NULL;
            free(ptr);
         }
  }
```

Deleting A Node from the End.

```
void delete_end ()
  {   node *ptr; ~~*temp~~
    if ( start == NULL)
        return;
    else if( start -> next == NULL)
          {  ptr = start;
             start = NULL;
             free(ptr);
          }
         else {  ptr = start;
                 while  ( ptr -> next != NULL)
                      ptr = ptr -> next;
                 ptr -> prev -> next  = NULL
                    free(ptr)
              }
  }
```

Adv: Birectional Traversing is possible, which helps in easy accessibility of nodes.

Disadv: uses of more pointers leads to more memory requirement.