DATA STRUCTURES

(this is an additional material and not exhaustive study material)

Data: Collection of raw facts and figures. When arranged in a proper format, it gives information

Information : Processed data which conveys useful content is called information. It gives useful meaning and helps us to make some decisions.

Algorithm : An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Data and Information - Data Structure Types - Concept of Data Types - Abstract Data Types- Pointers - Structures - Unions - Arrays - Multidimensional Arrays.

Data: Collection of raw facts and figures. When arranged in a proper format, it gives information

Information : Processed data which conveys useful content is called information. It gives useful meaning and helps us to make some decisions.

Algorithm : An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Properties of an Algorithm

1. **Finiteness:** - an algorithm terminates after a finite numbers of steps.
2. **Definiteness:** - each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.
3. **Input:-** an algorithm accepts zero or more inputs
4. **Output:-** it produces at least one output.
5. **Effectiveness:-** it consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

It is the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Time Complexity

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

The most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general it is a simple statement

statement;

Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for (i=o; i < N; i++)
{    statement;  }
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for (i=o; i < N; i++)
{   for (j=o; j < N; j++)
    {    statement;  }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
While (low <= high)
{       mid=(low+high)/2);
        if (target < list[mid])
                high=mid-1;
        else if (target > list[mid])
                low=mid+1;
        else break;     }
```

This is an algorithm to break a set of numbers into halves, to search a particular field. This algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{       int pivot = partition(list, left, right);
        quicksort(list, left, pivot-1);
        quicksort(list, pivot+1, right);  }
```

This is the slgorithm for Quick Sort. In Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Algorithm Analysis :

Time complexity and Space complexity are used to analyze the algorithm in the data structure. There are various ways of solving a problem and there exists different algorithms which can be designed to solve the problem.

Consequently, analysis of algorithms focuses on the computation of space and time complexity. Here are various types of time complexities which can be analyzed for the algorithm:

- **Best case time complexity:** The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' The running time of many algorithms varies not only for the inputs of different sizes but also for the different inputs of the same size.
- **Worst case time Complexity:** The worst case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n.' Therefore, if various algorithms for sorting are taken into account and say 'n,' input data items are supplied in reverse order for a sorting algorithm, then the algorithm will require n2 operations to perform the sort which will correspond to the worst case time complexity of the algorithm.

- **Average Time complexity Algorithm:** This is the time that the algorithm will require to execute a typical input data of size 'n' is known as the average case time complexity.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

EXAMPLES :

Big O Notation: N

```
function(n) {
   For(var a = 0; i <= n; i++) { // It's N because it's just a single loop
      // Do stuff
   }
}
```

Big O Notation: N2

```
function(n, b) {
   For(var a = 0; a <= n; a++) {
      For(var c = 0; i <= b; c++) { // It's N squared because it's two nested loops
         // Do stuff
      }
   }
}
```

Big O Notation: 2N

```
function(n, b) {
   For(var a = 0; a <= n; a++) {
      // Do stuff
   }
   For(var c = 0; i <= b; c++) { // It's 2N the loops are outside each other
      // Do stuff
   }
}
```

Big O Notation: NLogN

```
for(int i = 0; i < n; i++) //this loop is executed n times, so O(n)
{
   for(int j = n; j > 0; j/=2) //this loop is executed O(log n) times
```

```
   {

   }
}
```

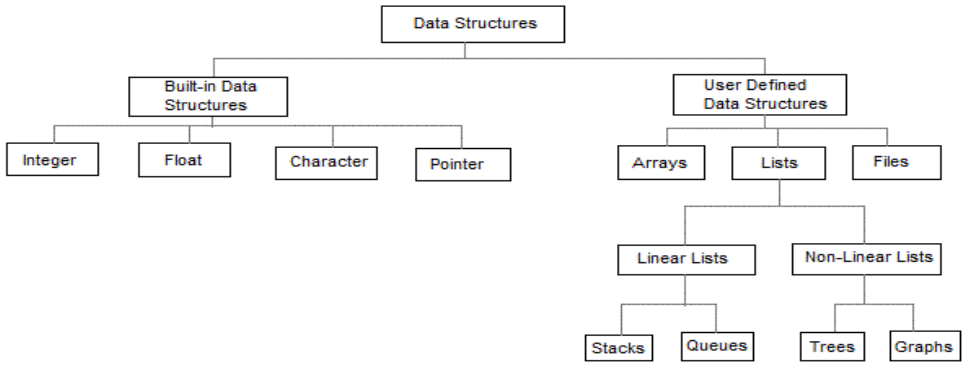
EXAMPLE

```
for (int i = 1; i <=m; i += c) {
      // some O(1) expressions
  }
  for (int i = 1; i <=n; i += c) {
      // some O(1) expressions
  }
  Time complexity of above code is O(m) + O(n) which is O(m+n)
  If m == n, the time complexity becomes O(2n) which is O(n).
```

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

Types of Data Structure Classification of Data Structures

Linear – Linear data structures, values are arranged in linear fashion. Arrays, linked lists, stacks and queues are examples of linear data structures in which the values are stored in a sequence.
Non linear – In this the values are not arranged in order. Examples trees, graphs and sets.
Homogenous – Values of the same type of data are stored. Example : Arrays
Non Nomogenous / Heterogenous – Values of different types are grouped. Example : Structures, Classes
Static – The value of the static variable remains in the memory throughout the program. Value of static variable persisits.
Dynamic – Size and memory locations can be changed during program executions. Example – Pointers.



INTRODUCTION TO DATA STRUCTURES

Primitive Data Structures :

Data structures that are directly operated upon by machine level instructions are known ass primitive data types. Example : integers, real/float, Boolean etc.

Complex Data Structures called as Abstract Data Structures are used to store large and connected data. These are complex data structures. They are derived from primitive data structures.Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.

Data Types in C :

C programming language which has the ability to divide the data into different types. The type of a variable determine the what kind of values it may take on. The various data types are

- Simple Data type
  → Integer, Real, float, Char
- Structured Data type
  →Array, Strings
- User Defined Data type
  →Enum, Structures, Unions

Four standard data types. They are int, float, char and double.

Abstract Data Types :

ADT = properties + operations

An Abstract Data Type (ADT) is:

- a set of *values*
- a set of *operations,* which can be applied uniformly to all these values

It is a data declaration packaged together with the operations that are meaningful for the data type.

Examples of ADT : Lists, Stacks, Queues, Trees, Heaps, Graphs

Structures

A structure is a user defined data type that groups logically related data items of different data types into a single unit. All the elements of a structure are stored at contiguous memory locations. A variable of structure type can store multiple data items of different data types under the one name. As the data of employee in company that is name, Employee ID, salary, address, phone number is stored in structure data type.

Defining a structure

The syntax of defining a structure is

```
struct <struct_name>
{
<data_type> <variable_name>;
<data_type> <variable_name>;
……..
<data_type> <variable_name>;
};
```

Example

The structure of Employee is declared as

```
 struct employee
{
  int  emp_id;
  char name[20];
  float salary;
```

```
    int age;
};
```

Assignment of values to structure elements

The members of individual structure variable is initialize one by one or in a single statement. The example to initialize a structure variable is

1) struct employee e1 = {1, "Alisha",12000,35);

OR

2) e1.emp_id=1;
   e1.name="Alisha";
   e1.salary=12000;
   e1.age=35;

The structure members cannot be directly accessed in the expression. They are accessed by using the name of structure variable followed by a dot and then the name of member variable. Example e1.emp_id, e1.name, e1.salary, e1.age. The data within the structure is stored and printed by this method using scanf and printf statement in c program

Structure Assignment

The value of one structure variable is assigned to another variable of same type using assignment statement. If the e1 and e2 are structure variables of type employee then the statement

e1 = e2;

assign value of structure variable e2 to e1. The value of each member of e2 is assigned to corresponding members of e1

Array of Structures

C language allows to create an array of variables of structure. The array of structure is used to store the large number of similar records. For example to store the record of 100 employees then array of structure is used. The method to define and access the array element of array of structure is similar to other array.

Example Struct employee e1[100];
Access them as e[i].emp_id, e[i].salary

Structure within a structure

C language define a variable of structure type as a member of other structure type.

Example

```
struct employee
  {  int  emp_id;
     char name[20];
     float salary;
     struct date
          {    int day;
               int month;
               int year;
          }
};
```

Arrays

- Structures of related data items
- Static entity – same size throughout program
- Group of consecutive memory locations
- Same name and type

• To refer to an element, specify
  - Array name

- Position number

- Format:

  *arrayname***[** *position number* **]**
  - First element at position **0**
  - **n** element array named **c**:
    - **c[0]**, **c[1]**...**c[n – 1]**

Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
  - Number of elements

arrayType arrayName[numberOfElements];

Examples:

int c[10];
float myArray[3284];

Declaring multiple arrays of same type. Format similar to regular variables
Example: int b[100], x[27];

Initializers int n[5] = { 1, 2, 3, 4, 5 };

- If not enough initializers, rightmost elements become **0**

int n[5] = { 0 } All elements 0
If too many a syntax error is produced syntax error
C arrays have no bounds checking
If size omitted, initializers determine it
int n[ ] = { 1, 2, 3, 4, 5 }; 5 initializers, therefore 5 element array

STACK

Stack is a LIFO (last in first out) structure. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. When elements are added to stack it grow at one end. Similarly, when elements are deleted from a stack, it shrinks at the same end.

Applications of Stack

1. **Expression Evaluation**
   Stack is used to evaluate prefix, postfix and infix expressions.
2. **Expression Conversion**
   An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
3. **Syntax Parsing**
   Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
4. **Backtracking**
   Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

5. **Parenthesis Checking**
   Stack is used to check the proper opening and closing of parenthesis.
6. **String Reversal**
   Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
7. **Function Call**
   Stack is used to keep information about the active functions or subroutines.

PROGRAM

```
#include<stdio.h>
#include<process.h>
#include<stdlib.h>
#define MAX 5          //Maximum number of elements that can be stored

int top=-1;
stack[MAX];
void push();
void pop();
void display();

void main()
{
        int ch;

        while(1)        //infinite loop, will end when choice will be 4
        {
                printf("\n*** Stack Menu ***");
                printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
                printf("\n\nEnter your choice(1-4):");
                scanf("%d",&ch);

                switch(ch)
                {
                        case 1: push();
                                        break;
                        case 2: pop();
                                        break;
                        case 3: display();
                                        break;
                        case 4: exit(0);

                        default: printf("\nWrong Choice!!");
                }
        }
}

void push()
{
```

```
        int val;

        if(top==MAX-1)
                printf("\nStack is full!!");
        else
        {
                printf("\nEnter element to push:");
                scanf("%d",&val);
                top=top+1;
                stack[top]=val;
        }
}

void pop()
{
        if(top==-1)
                printf("\nStack is empty!!");
        else
        {
                printf("\nDeleted element is %d",stack[top]);
                top=top-1;
        }
}

void display()
{
        int i;

        if(top==-1)
                printf("\nStack is empty!!");
        else
        {
                printf("\nStack is...\n");
                for(i=top;i>=0;--i)
                        printf("%d\n",stack[i]);
        }
}
```

QUEUES

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first

Applications of Queues

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
3) data packets waiting to be transmitted over the Internet.
4) There are various queues quietly doing their job in your computer's (or the network's) operating system.

5) There's a printer queue where print jobs wait for the printer to be available

Disadvantage of Linear Queue:

A major disadvantage of a classical queue is that a new element can only be inserted when *all* of the elements are deleted from the queue. Reusability of space is not possible. This can be overcome by Circular queues.

Program

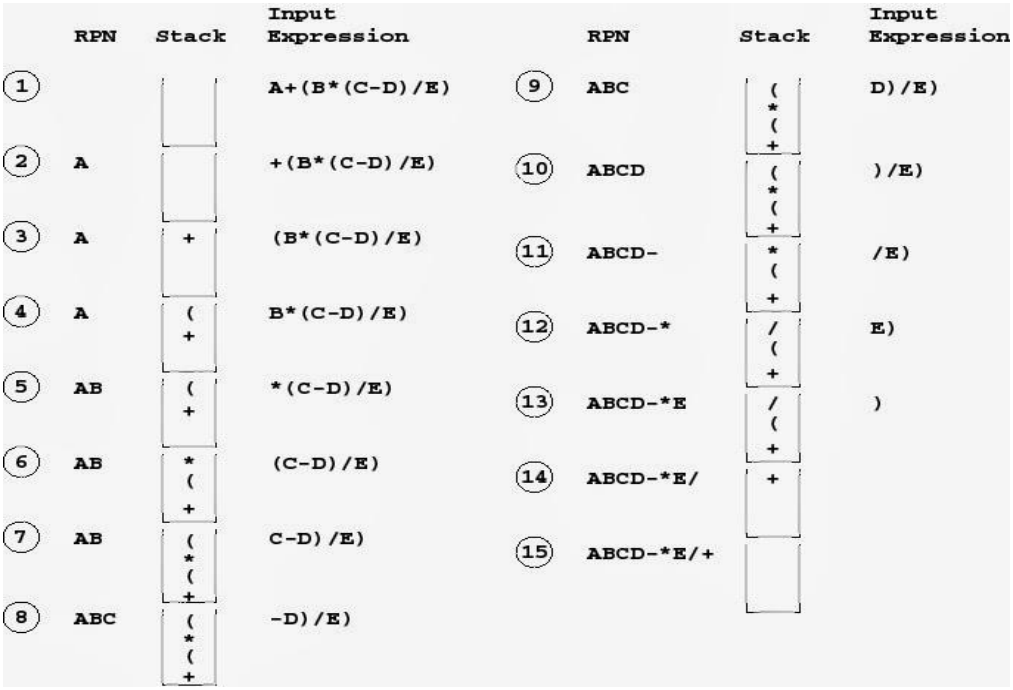
```
#include<stdio.h>
#include<conio.h>
#define n 5
void main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
   clrscr();
   printf("Queue using Array");
   printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
   while(ch)
   {
      printf("\nEnter the Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:
         if(rear==x)
            printf("\n Queue is Full");
         else
         {
            printf("\n Enter no %d:",j++);
            scanf("%d",&queue[rear++]);
         }
         break;
      case 2:
         if(front==rear)
         {
            printf("\n Queue is empty");
         }
         else
         {
            printf("\n Deleted Element is %d",queue[front++]);
            x++;
         }
         break;
      case 3:
         printf("\n Queue Elements are:\n ");
         if(front==rear)
            printf("\n Queue is Empty");
         else
         {
            for(i=front; i<rear; i++)
            {
               printf("%d",queue[i]);
               printf("\n");
```

```
            }
            break;
        case 4:
            exit(0);
        default:
            printf("Wrong Choice: please see the options");
        }
    }
  }
  getch();
}
```

Difference Between Stack and Queue

Stack	Queue
A Stack Data Structure works on Last In First Out (LIFO) principle.	A Queue Data Structure works on First In First Out (FIFO) principle.
A Stack requires only one reference pointer.	A Queue requires two reference pointers.
A Stack contains **TOP** as its reference for data processing.	A Queue contains **REAR** and **FRONT** as its reference for data processing.
The data items are inserted and deleted from the same end.	The data items in a queue are inserted and deleted from different ends.
The element to be inserted first is removed last.	The element to be inserted first is removed first.
To check if a stack is empty, following condition is used: **TOP == -1**	To check if a queue is empty, following condition is used: **FRONT == -1 || FRONT == REAR + 1**
The insertion and deletion operations occur at the TOP end of a Stack.	The insertion operation occurs at REAR end and the deletion operation occurs at FRONT end.
To check if a stack is full, following condition is used: **TOP == MAX – 1**	To check if a queue is full, following condition is used: **REAR == MAX – 1**
Used in infix to postfix conversion, scheduling algorithms, depth first search and evaluation of an expression.	A Queue offers services in operations research, transportation and computer science that involves persons, data, events and objects to be stored for later processing.

Infix to Postfix conversion :



Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
…..**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.
Dynamic Memory Allocation
Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)

Program – Infix to Postfix Conversion

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{    stack[++top] = x; }

char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}
```

```
int priority(char x)
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
}

main()
{
   char exp[20];
   char *e, x;
   printf("Enter the expression :: ");
   scanf("%s",exp);
   e = exp;
   while(*e != '\0')
   {
      if(isalnum(*e))
         printf("%c",*e);
      else if(*e == '(')
         push(*e);
      else if(*e == ')')
      {
         while((x = pop()) != '(')
            printf("%c", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(*e))
            printf("%c",pop());
         push(*e);
      }
      e++;
   }
   while(top != -1)
   {
      printf("%c",pop());
   }
}
```

Evaluation of Postfix Expression

The compiler finds it convenient to evaluate an expression in its postfix form. The advantage of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression.

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.

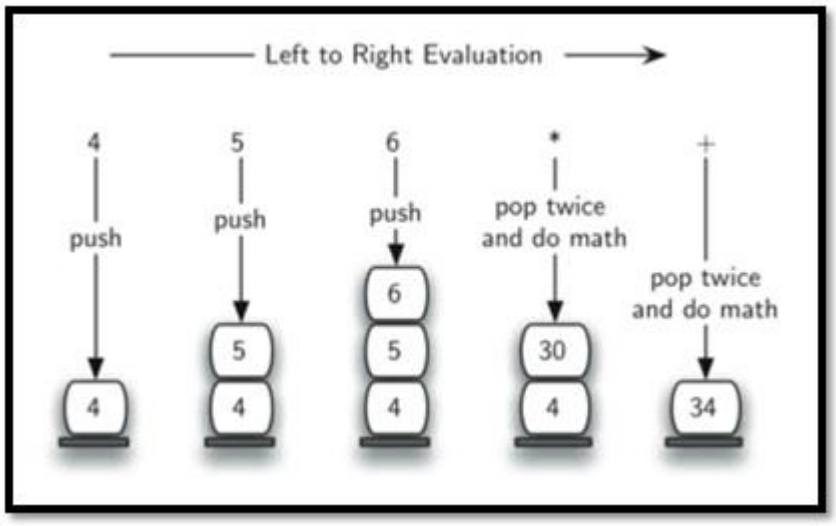
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1) Add) to postfix expression.
2) Read postfix expression Left to Right until) encountered and repeat steps 3 and 4
3) If operand is encountered, push it onto Stack
4) If operator is encountered, Pop two elements
i) A ->Top element
ii) B->Next to Top element
iii) Evaluate B operator A and Push the result onto Stack
5) Set result=pop
6) END

EXAMPLE :

Expression: 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Program

```
#include<stdio.h>
#include<ctype.h>
 # define MAXSTACK 100        /* for max size of stack */
 # define POSTFIXSIZE 100     /* define max number of charcters in postfix expression */

 /* declare stack and its top pointer to be used during postfix expression
            evaluation*/
 int stack[MAXSTACK];
 int top = -1 ;            /* because array index in C begins at 0 */
 /* can be do this initialization somewhere else */

 /* define push operation */
 void push(int item)
 {

            if(top >= MAXSTACK -1)
            {
                        printf("stack over flow");
                        return;
            }
            else
            {
                        top = top + 1 ;
                        stack[top]= item;
            }
 }

 /* define pop operation */
 int pop()
 {
            int item;
            if(top <0)
            {
                       printf("stack under flow");
            }
            else
            {
                        item = stack[top];
                        top = top - 1;
                        return item;
            }
 }

 /* define function that is used to input postfix expression and to evaluate it */
 void EvalPostfix(char postfix[])
 {

            int i ;
            char ch;
```

```
        int val;
        int A, B ;


        /* evaluate postfix expression */
        for (i = 0 ; postfix[i] != ')'; i++)
        {
                ch = postfix[i];
                if (isdigit(ch))
                {
                        /* we saw an operand,push the digit onto stack
                        ch - '0' is used for getting digit rather than ASCII code of digit */
                        push(ch - '0');
                }
                else if (ch == '+' || ch == '-' || ch == '*' || ch == '/')
                {
                        /* we saw an operator
                        * pop top element A and next-to-top elemnet B
                        * from stack and compute B operator A
                        */
                        A = pop();
                        B = pop();

                        switch (ch) /* ch is an operator */
                        {
                                case '*':
                                val = B * A;
                                break;

                                case '/':
                                val = B / A;
                                break;

                                case '+':
                                val = B + A;
                                break;

                                case '-':
                                val = B - A;
                                break;
                        }

                        /* push the value obtained above onto the stack */
                        push(val);
                }
        }
        printf( " \n Result of expression evaluation : %d \n", pop()) ;
}

void main()
{       int i ;
        /* declare character array to store postfix expression */
```

```
        char postfix[POSTFIXSIZE];
        printf("ASSUMPTION: There are only four operators(*, /, +, -) in an expression and operand is
                single digit only.\n");
        printf( " \nEnter postfix expression,\npress right parenthesis ')' for end expression : ");

        /* take input of postfix expression from user */

        for (i = 0 ; i <= POSTFIXSIZE - 1 ; i++)
        {
                scanf("%c", &postfix[i]);
                if ( postfix[i] == ')' ) /* is there any way to eliminate this if */
                { break; } /* and break statement */
        }
        /* call function to evaluate postfix expression */
        EvalPostfix(postfix);
}
```

Pointers

A *pointer* is a reference to another variable (memory location) in a program

- Used to change variables inside a function (reference parameters)
- Used to remember a particular member of a group (such as an array)
- Used in dynamic (on-the-fly) memory allocation (especially of arrays)
- Used in building complex data structures (linked lists, stacks, queues, trees, etc.)

– Variables are allocated at *addresses* in computer memory (address depends on computer/operating system)
– Name of the variable is a reference to that memory address
– A pointer variable contains a representation of an address of another variable (P is a pointer variable in the following

Need for pointers

- They allow you to refer to large data structures in a compact way
- They facilitate sharing between different parts of programs
- They make it possible to get new memory dynamically as your program is running
- They make it easy to represent relationships among data items.

Why pointers should be used carefully ?

- They are a powerful low-level device.
- Undisciplined use can be confusing and thus the source of subtle, hard-to-find bugs.
  - Program crashes
  - Memory leaks
  - Unpredictable results

Uses of Pointers

- Working with memory locations that regular variables don't give you access to
- Working with strings and arrays
- Creating new variables in memory while the program is running
- Creating arbitrarily-sized lists of values in memory

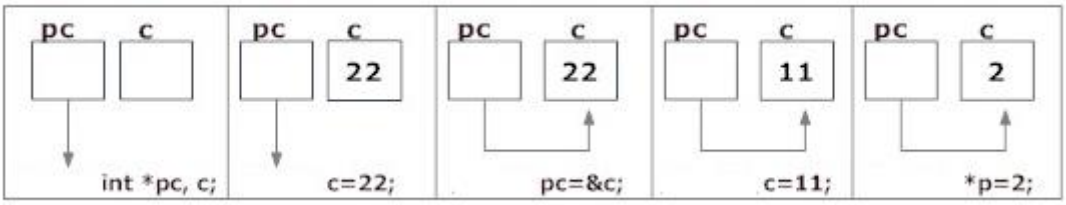
Key Points To Remember About Pointers In C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.

- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Reference operator (&) and Dereference operator (*)

& is called reference operator. It gives you the address of a variable.
Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).




```
int* pc;
  int c;
  c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

We can do integer arithmetic on a pointer:

```
float *flp, *flq;

        *flp = *flp + 10;
        ++*flp;
        (*flp)++;
        flq = flp;
```

Common mistakes when working with pointers

```
int c, *pc;

pc = c;  // Wrong! pc is address whereas, c is not an address.

*pc = &c; // Wrong! *pc is the value pointed by address whereas, &c is an address

pc = &c;   // Correct! pc is an address and, &pc is also an address.

*pc = c; // Correct! *pc is the value pointed by address and, c is also a value.
```

Example for Call by reference

```
void swap(int * n1, int * n2) // pointer n1 and n2 points to the address of num1 and num2 respectively
{  int temp;
   temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}

void main()
{    int num1 = 5, num2 = 10;
   swap( &num1, &num2);       // address of num1 and num2 is passed to the swap function
   printf("Number1 = %d\n", num1);
   printf("Number2 = %d", num2);
 }
```

Output

```
Number1 = 10
Number2 = 5
```

Example for Call by value

```
void swap(int x, int y) // a copy of num1 and num2 are made.
{  int temp;
   temp = x;
   y = x;
   x = temp;
}

void main()
{    int num1 = 5, num2 = 10;
   swap( num1, num2);       // vlues of num1 and num2 is passed to the swap function
   printf("Number1 = %d\n", num1);
```

```
   printf("Number2 = %d", num2);
}
```

Output

```
Number1 = 5
Number2 = 10
```

Dynamic Memory Allocation

In C, the exact size of array is unknown until compile time, i.e., the time when a compiler compiles your code into a computer understandable language. So, sometimes the size of the array can be insufficient or more than required. Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required. In simple terms, Dynamic memory allocation allows you to manually handle memory space for your program.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

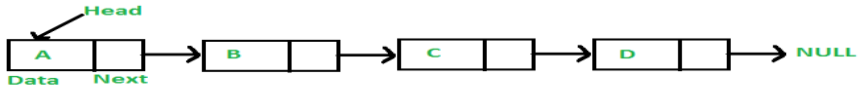
syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Linked list data structure

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

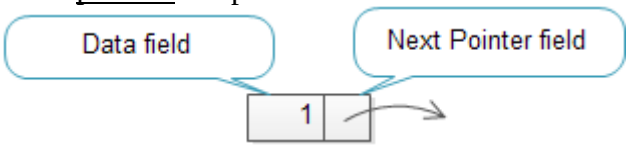
Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.

Representation in C:

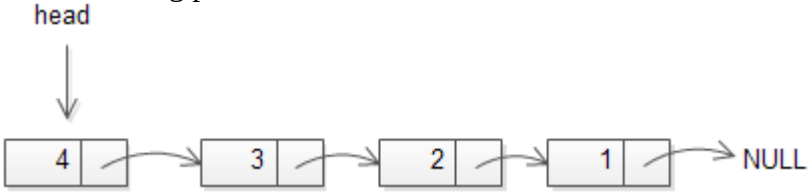
A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
Each node in a list consists of at least two parts:
1) data
2) pointer to the next node

A linked list is a data structure that consists of sequence of nodes. Each node is composed of two fields: **data field** and **reference field** which is a pointer that points to the next node in the sequence.



Linked List Node

Each node in the list is also called an element. The reference field that contains a pointer which points to the next node is called **next pointer** or **next link**.
A **head** pointer is used to track the first element in the linked list therefore it always points to the first element.
The following picture illustrates a linked list



Singly Linked List

We can model a node of the singly linked list using a structure follows:

```
typedef struct node{
   int data;
   struct node* link;
}
```

The node structure has two members:

- data stores the information
- link pointer holds the address of the next node.

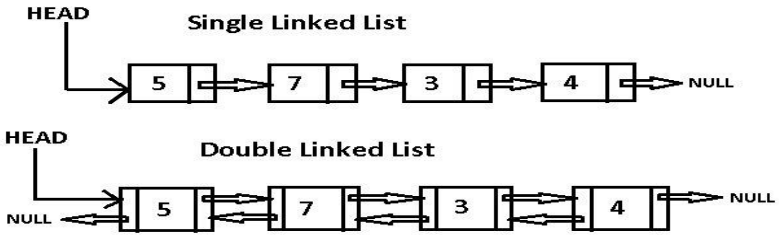
Doubly Linked List

We can model a node of the doubly linked list using a structure follows:

```
typedef struct node{
   int data;
   struct node* flink, blink;
}
```

The node structure has two members:

- data stores the information
- flink pointer holds the address of the next node.
- blink pointer holds the address of the previous node



Single Linked List	Double Linked List
Single linked list also known as one way list	Double linked list is also known as two way list
Each node is divided into two part. **Data Field-** contain the Data of node **Link Field-** Contain the address of next node	Each node is divided into three part: **Data field-** Contain the data of node. **Forward Link Field:** Contain the address of next node **Backward Link Field-** Contain the address of previous node
It can traversed only forward direction	It can be traversed both forward and backward direction.
Single linked list use less memory and less space.	Double linked list use more memory and more space because of two pointer
If we need to save memory in need to update node values frequently and searching is not required, we can use singly linked list	If we need faster performance in searching and memory is not a limitation we use Doubly Linked List

Singly Linked List

Creating a Node :

Algorithm

1. declare a structure node with two fields – data and a pointer to the next node
2. use malloc function to create a node.
3. Get the data from user and assign it to data

4. Map the link to the next node as required or assign as null.

Code

```
struct snode
{ int data;
  struct node * link; }

snode * getnode()
{ snode * newnode;
  newnode = (struct snode *) malloc(sizeof(struct snode))
  newnode->link=null;
  return newnode;
}
```

Insertion in the Beginning:

Algorithm

1. Create a new node
2. Get the data value from the user and assign to the new node. Let the link be null
3. If the list is null, head==null, then assign head to point to new node
   head = new node
4. If the list is not null, then
   - make the link of new node point to head.
   - Make the head point to new node

Code

```
void insert_beg(snode* list)
{   int x;
    snode* newnode;
    newnode=getnode();
    printf("Enter the data to be inserted");
    scanf("%d",x);
    newnode->data=x;
    if (list==null)
       list=newnode;
    else
        { newnode->link=list;
          list=newnode;  }
}
```

Insertion at the end:

Algorithm

1. Create a new node
   - Get the data value from the user and assign to the new node. Let the link be null
2. If the list is null, head==null, then assign head to point to new node
   head = new node
3. If the list is not null, then
   - Traverse till the end of the list using trav, let trav point to the last node.
   - trav->link should point to new node

Code

```
void insert_end(snode* list)
{   int x;
```

```
    snode* newnode, * trav;
    newnode=getnode();
    printf("Enter the data to be inserted");
    scanf("%d",x);
    newnode->data=x;
    if (list==null)
       list=newnode;
    else
        { trav=list;
          while (trav->link != null)
                  trav=trav->link;  // jump till last node
          trav->link=newnode;     // attach new node to last node
        }
}
```

Insertion after a node whose value is given:

Algorithm

1. If the list is null, head==null, then "ERROR – List is Empty"
2. If the list is not null, then
   - Get the node value after which insertion has to take place, say y.
   - Traverse the list with trav, till the node with value y is found or till trav becomes null.
   - If trav=null, then "ERROR – No node with value y – insertion not possible"
   - If trav!=null, then attach the newnode after trav
     - Create a new node, get the data value from the user and assign to the new node
     - newnode->link=trav->link
     - trav->link=newnode

Code

```
void insert_after(snode* list)
{   int x,y;
    snode* newnode, * trav;
    if (list==null)
       printf("ERROR – List Empty)
    else
        { printf("Enter value of node after which insertion to take place");
          Scanf("%d", &y);
          trav=list;
          while (trav != null || trav->data !=y)
                  trav=trav->link;  // jump till last node or till node with value y found
          if (trav==null)        // no node with value y
             printf("no node with value y after which insertion to be done")
          else                          // node with value y found and pointed by trav
             {   newnode=getnode();
                 printf("Enter the data to be inserted");
                 scanf("%d",x);
                 newnode->link=trav->link;    //attach the new node after trav, the node with value y
                 trav->link=newnode;
             }
          }
}  // function ends
```

Deletion in the Beginning:

Algorithm

1. If the list is nul print "ERROR – Empty list, Deletion not done"
2. Let temp point to first node pointed by head.
   temp=head
3. Make head point to the next node
   Head=head->link
4. Free temp

Code

```
void delete_beg(snode* list)
{   snode *temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done")
    else
       {  temp=list;
          list=list->link;
          free(temp);     //return the deleted node to memory
       }
}
```

Deletion at the end:

Algorithm

1. If the list is empty print "ERROR – Empty list, Deletion not done"
2. If the list is not null, then
   - Traverse till the end of the list using trav, **let trav point node previous to last**
   - Make temp point to last node (as pointed by trav)
     temp=trav->link
   - Disconnect last nnode
     trav->link = null;
   - Free temp

Code

```
void delete_end(snode* list)
{   snode* temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done")
    else
        { trav=list;
          while (trav->link->link != null)    // move on till you reach one node before last node
                trav=trav->link;  // jump till last node
          temp=trav->link;     // last node pointed by temp
          trav->link=null;       //disconnect the last node
          free(temp);             // return the deleted node to memory
        }
}
```

Deletion of a node whose value is given:

Algorithm

1. If the list is null, head==null, then "ERROR – List is Empty"
2. if the list is not null, then
   - Get the value of the node to be deleted.
3. If that is the only node in the list, then make head null.
   If (head->data==y && head->link==null)
      head=null

4. If there are more than one node, start traversing to the node with value y.
5. Traverse the list with trav, till the node with value y is found or till trav becomes null.
   Keep prev to keep track of previous node
   - If trav=null, then "ERROR – No node with value y – deletion not possible"
   - If trav!=null, then
     - Make temp point to the node to be deleted
       temp=trav;
     - prev->link = trav->link // Disconnect temp from list
     - free(temp); // Return the deleted node to memory

Code

```
void delete-middle(snode* list)
{   int y;
    snode* temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done");
    if (list!==null)
       {
        printf("enter the node value to be deleted");
        scanf("%d",&y)
        if (list->data==y)
          {  temp=trav;
             list=list->link;            // happened to be the first node
             free(temp);          // return the deleted node to memory
          }
        else                          // list has many node, search using trav, but keep track of prev node
          { while (trav != null && trav->data !=y)
                 {  prev=trav;         // prev keeps track of previous node
                    trav=trav->link;  // jump till last node or till node with value y found
                 }
           if (trav==null)
              printf("ERROR – no such node with value given");
           else
               {
                temp=trav                    // node to be deleted with value
                prev->link=trav->link       //disconnect the node
                free(temp);                 // return the deleted node to memory
               }
          }
 }  // function ends
```

Traverse the List :

Algorithm

1. If the list is null, print "EMPTY LIST"
2. If the list is not null, then
   - Traverse till the end of the list, printing the content of each node

Code

```
void traverse(snode* list)
{
    if (list==null)
```

```
        printf("EMPTY LIST);
    else
        { trav=list;
          printf("The contents of the List :")
          while (trav != null)     // traverse till end of list, printing each node value
                  { printf("node value %d ", trav->data);
                    trav=trav->link;  // jump to next node
                  }
         }
}
```

Doubly Linked List

Creating a Node :

Algorithm

1. declare a structure node with two fields – data and a pointer to the next node.
2. use malloc function to create a node.
3. Get the data from user and assign it to data
4. Map the link to the next node as required or assign as null.

Code

```
struct dnode
{ int data;
  struct node * flink, * rlink; }

snode * getnode()
{ snode * newnode;
  newnode = (struct snode *) malloc(sizeof(struct snode))
  newnode->flink=null;
  newnode->blink=null;
  return newnode;
}
```

Insertion in the Beginning:

Algorithm

1. Create a new node
2. Get the data value from the user and assign to the new node. Let flink and blink be null
3. If the list is null, head==null, then assign head to point to new node
   head = new node
4. If the list is not null, then
   - make the link of new node point to head.
   - Make the head point to new node

Code

```
void insert_beg(dnode* list)
{   int x;
    dnode* newnode;
    newnode=getnode();
    printf("Enter the data to be inserted");
    scanf("%d",x);
    newnode->data=x;
    if (list==null)
       list=newnode;
```

```
    else
        { newnode->flink=list;
          list=newnode;  }
}
```

Insertion at the end:

Algorithm

1. Create a new node
   - Get the data value from the user and assign to the new node. Let the link be null
2. If the list is null, head==null, then assign head to point to new node
   head = new node
3. If the list is not null, then
   - Traverse till the end of the list using trav, let trav point to the last node.
   - trav->flink should point to new node
   - newnode->blink should point to trav

Code

```
void insert_end(dnode* list)
{   int x;
    dnode* newnode, * trav;
    newnode=getnode();
    printf("Enter the data to be inserted");
    scanf("%d",x);
    newnode->data=x;
    if (list==null)
       list=newnode;
    else
        { trav=list;
          while (trav->flink != null)
                trav=trav->flink;  // jump till last node
          trav->flink=newnode;     // attach new node to last node
          newnode->blink=trav      //attach newnode backward link to point to trav
        }
}
```

Insertion after a node whose value is given:

Algorithm

1. If the list is null, head==null, then "ERROR – List is Empty"
2. If the list is not null, then
   - Get the node value after which insertion has to take place, say y.
   - Traverse the list with trav, till the node with value y is found or till trav becomes null.
   - If trav=null, then "ERROR – No node with value y – insertion not possible"
   - If trav!=null, then attach the newnode after trav
     - Create a new node, get the data value from the user and assign to the new node.
     - newnode->flink=trav->flink
     - newnode->blink=trav
     - trav->flink=newnode

Code

```
void insert_after(dnode* list)
{   int x,y;
    dnode* newnode, * trav;
```

```
    if (list==null)
       printf("ERROR – List Empty)
    else
        { printf("Enter value of node after which insertion to take place");
          Scanf("%d", &y);
          trav=list;
          while (trav != null && trav->data !=y)
                  trav=trav->flink;  // jump till last node or till node with value y found
          if (trav==null)        // no node with value y
             printf("no node with value y after which insertion to be done")
          else                          // node with value y found and pointed by trav
             {   newnode=getnode();
                 printf("Enter the data to be inserted");
                 scanf("%d",x);
                 newnode->flink=trav->flink;    //attach the new node after trav, the node with value y
                 newnode->blink=trav;
                 trav->flink=newnode;
             }
          }
} // function ends
```

Deletion in the Beginning:

Algorithm

1. If the list is null print "ERROR – Empty list, Deletion not done"
2. Let temp point to first node pointed by head.
   temp=head
3. Make head point to the next node
   Head=head->flink
4. Free temp

Code

```
void delete_beg(dnode* list)
{   dnode *temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done")
    else
       {  temp=list;
          list=list->flink;
          free(temp);     //return the deleted node to memory
       }
}
```

Deletion at the end:

Algorithm

1. If the list is empty print "ERROR – Empty list, Deletion not done"
2. If the list is not null, then
   - Traverse till the end of the list using trav
   - Make temp point to last node (as pointed by trav)
     temp=trav
   - Disconnect last nnode
     trav->blink->flink = null;
   - Free temp

Code

```
void delete_end(dnode* list)
{   dnode* temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done")
    else
        { trav=list;
          while (trav != null)    // move on till you reach last node
                trav=trav->flink;  // jump till last node
          temp=trav;     // last node pointed by temp
          trav->blink->flink=null;       //disconnect the last node
          free(temp);             // return the deleted node to memory
        }
}
```

Deletion of a node whose value is given:

Algorithm

1. If the list is null, head==null, then "ERROR – List is Empty"
2. if the list is not null, then
   - Get the value of the node to be deleted.
3. If that is the only node in the list, then make head null.
   If (head->data==y && head->flink==null)
   head=null
4. If there are more than one node, start traversing to the node with value y.
5. Traverse the list with trav, till the node with value y is found or till trav becomes null.
   - If trav=null, then "ERROR – No node with value y – deletion not possible"
   - If trav!=null, then
     - Make temp point to the node to be deleted
       temp=trav;
     - trav->blink->flink=trav->flink; //previous node connected to next node
       trav-flink->blink=trav->blink   // next node connected to previous node
     - free(temp);   // Return the deleted node to memory

Code

```
void delete-middle(dnode* list)
{   int y;
    dnode* temp;
    if (list==null)
       printf("ERROR – Empty list, Deletion not done");
    if (list!==null)
       {
        printf("enter the node value to be deleted");
        scanf("%d",&y)
        if (list->data==y)
          {  temp=trav;
             list=list->flink;              // happened to be the first node
             list->blink=null;
             free(temp);             // return the deleted node to memory
          }
        else                              // list has many node, search using trav, but keep track of prev node
           { while (trav!= null && trav->data !=y)
                   trav=trav->flink;  // jump till last node or till node with value y found
            if (trav==null)
```

```
            printf("ERROR – no such node with value given");
          else
            { temp=trav          // node to be deleted with value
              trav->blink-flink=trav->flink      //disconnect the node
              trav->flink->blink=trav->blink
              free(temp);          // return the deleted node to memory
            }
        }
  }  // function ends
```

Traverse the List :

Algorithm

3. If the list is null, print "EMPTY LIST"
4. If the list is not null, then
   - Traverse till the end of the list, printing the content of each node

Code

```
void traverse(dnode* list)
{
    if (list==null)
       printf("EMPTY LIST);
    else
        { trav=list;
          printf("The contents of the List :")
          while (trav != null)     // traverse till end of list, printing each node value
                  { printf("node value %d ", trav->data);
                    trav=trav->flink;  // jump to next node
                  }
         }
}
```