Data Structers in "C"

Assignment - II

Q1) Discuss about pointers, String Management / Manipulation functions and structures with Syntax egs.

Pointers are one of the most important mechanism in 'C' They are the means by which we implement call by "reference functions calls". They are closely related to arrays and strings in C, they are fundamental to utilize C' dynamic theory Memory allocation features, and they can lead to faster and more efficient code when we use correctly.

"A pointer is a variable that is used to store a memory address".

Pointer variables * and &

Pointers are like all over other variables in C must be declared as such prior to use.

* Asterisk :- indicates that it is a pointer

& and :- is a unary operator that returns the address of its operand. which must be variable.

Syntax:-

```
int *m;
int count=125, i;
m= &count;
```

Note:- /* m is a pointer to int, count, i are integers /

Q1) Write a program to pass the structure to a function using pointers

```
#include<stdio.h>

struct car
{
char brand[20];
int year;
};
void updateyear(struct Car *c)
{
c->year = 2025;
}
int main()
{
struct Car mycar = {"Toyota", 2020};
updateyear(&mycar);
printf("Brand: %s\n", mycar.brand);
printf("Year: %d\n", mycar.year);
return 0;
}
```

Output:

Brand : Toyota
Year : 2025

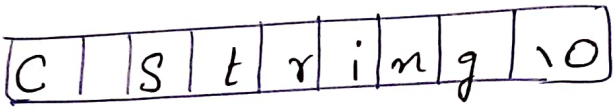
Q1) Strings & String operations in 'C'

In 'C' programming, a string is a sequence of characters terminated with a null character \0. for example.

Char c[] = "cstring";

When the compiler encounters a sequence of characters enclosed in the double quotations marks, it appends a null character \0 at the end by default.

Memory diagram.



Example:- Scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name:");
    scanf("%s", name);
    printf("your name is %s", name);
    return 0;
}
```

Output:

Enter your name: A Satya

Your name is A Satya

Note:- Scanf() already allocated the address of "name" variable, that is why we did not use "&".

Common String manipulation functions in "C".

1. strlen() - Returns the length of the string
2. strcpy() - copies one string to another
3. strncpy() - copies 'n' characters from one string to another.
4. strcat() - Concatenate two strings. Appends two strings
5. strcmp() :- Compares two strings
6. strncmp() :- Compares n strings of first two characters.
7. strrev() - Compares two strings & Reverse a string.

Q 1) Write a program to Concatenate two strings.

```
# include<stdio.h>

Void concat (char * S1, char * S2)
  {
   int i=0, j=0;
  while (S1[i] != '\0')
     i++;
  while (S2[j] != '\0')
      {
      S1[i++] = S2[j++];
      }
      S1[i]= '\0';
}
   int   main ()
        {
      Char S1[50] = "Hello";
      char S2[ ] : " Satya";
      Concat (S1, S2);
      printf("%s\n", S1);
      Return 0;
    }
```

output

Hello Satya

Q1) Write a program to find the length of string.

```
# include <stdio.h>
int main()
  {
  char str[100];
  int length = 0, i = 0;
// input string
printf(" Enter a string: ");
gets(str);
// count character until null terminator
while( str[i] != '\0' )
    {
      length = length + 1; (or) length ++;
      i++;
// out put result statement.
printf(" Length of the string is : %d\n", length);
return 0;
}
```

Output: Enter a string : Hello World

Length of the string is : 11

Q1) Write a program to count the no of words, characters, alphabets, vowels, consonants and digit in a line of text

```
# include <stdio.h>
# include <conio.h>
 main()
 {
  int  noa = 0,  nob=0,  noc = 0,  nov=0,  now= 0,  noch=0, i;
 char  ch.s[100];
 clrscr();
 printf(" enter 2 lines of text ");
  gets(s);
   l= strlen(s);
  for(i=0; i<l; i++)
     {
      switch (s[i])
         {
          Case  'a';
          case  'e',
          Case   'i';
          Case   'o',
          case    'u',
          Case   'A',
          Case    'E',
          Case  'I';
          Case    'O';
          Case    'U';
          nov++;
          break;
          }
           if(isalpha( s[i]))
             noa++;
```

Q2) Explain about looping statements of 'c'. and operator precedence.

Looping statements (or) Iterative statements in "C"

Looping is a way by which we can execute set of statements, more than one time. In 'C' there are mainly three types of loops are used.

- → while loop
- → do while loop
- → for loop

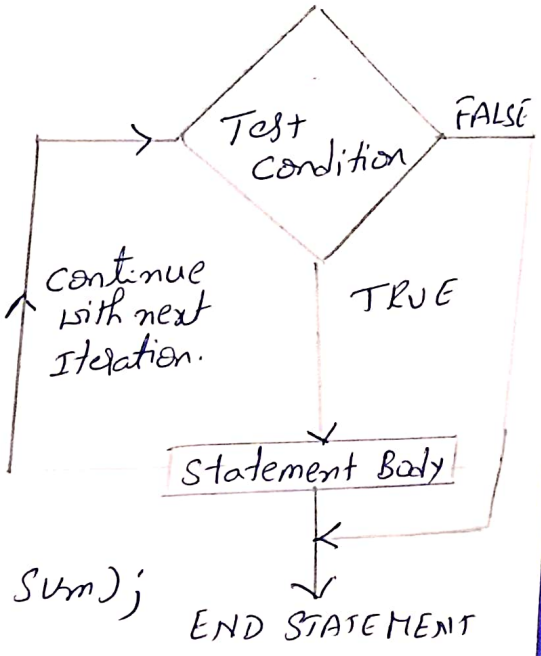
while loop :- The while statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax:-

```
while <condition>
{
statement 1;
statement 2;
- - -
increment / decrement;
}
```

Q2) Write a program to find the sum of integers 100 to 1 in descending order using While loop.

```
# include <stdio.h>
  int main ()
    {
  int n = 100;
  int sum = n * (n+1)/2;
  printf (" Sum of integers
   from 100 to 1 is : %d \n", sum);
    return 0;
    }
```

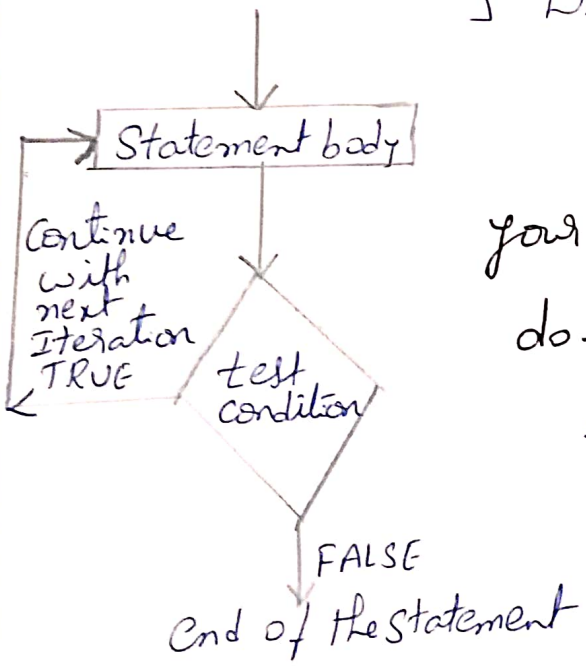


output :- Sum of the integers from 100 to 1 : 5050

Q2) **Do while loop:-** The do while loop is a post-tested loop in C. Which means the loop body is executed atleast once, and the condition is tested. If the condition is true, the loop executes again. It continues until the condition becomes false.

Syntax:-

```
do
{
 statement body;
} while <condition>;
```



Write a program to print your name '5' times using do... while loop.

```
#include <stdio.h>
void main()
 int count = 1;
  do{
  printf("Satya");
  count++;
 }
while(count<=5);
  return 0;
  }
```

Output:

Satya
Satya
Satya
Satya
Satya

Q2) **For loop** :- The for statement is most often used in situations where the programmer knows in advance how many times a particular set of statements are to be repeated. The for statement is sometimes termed a counted loop.

Syntax:- for([initialisation]; [condition]; [increment])
[Statement body];

Write a program to print all numbers 1 to 100 and find their sum.

```
# include <stdio.h>
    Void main ( )
    {int   x, sum=0;
     for(x=1;  x<=100; x++)
       {
        printf (" %d \n", x);
        sum+ = x;
       }
        printf( "\n\n sum is %d \n", sum);
          }
```

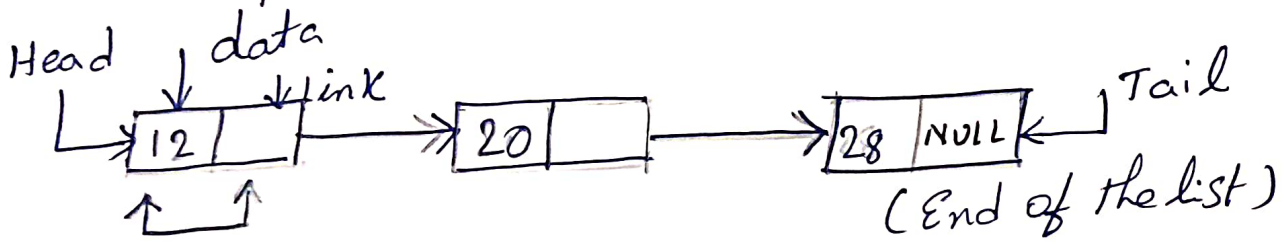
output:-

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 ------- 100

Sum of numbers from 1 to 100 is : 5050

Q3) What are linked lists ? Write various operations on a linked list with their implementation code.

Linked lists :- Linked list can be defined as a collection of objects called "nodes" that are randomly stored in the memory.
A node contains two fields i.e data stored at that particular address & the pointer which is connected to the address of the next node in the memory. The last node of the list contains pointer to the null.



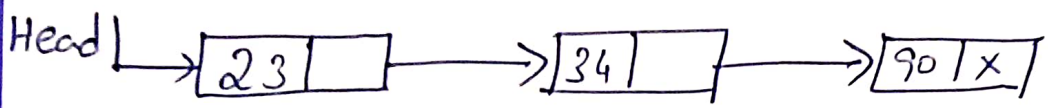
Uses of Linked list :-

→ The list is not required to be contiguously present in the memory. The node can reside any where in the memory.
→ List size is limited to the memory size and doesn't need to be declared in advance.
→ Empty node cannot be present in the linked list.
→ We can store values of primitive types (or) objects in the single linked list.

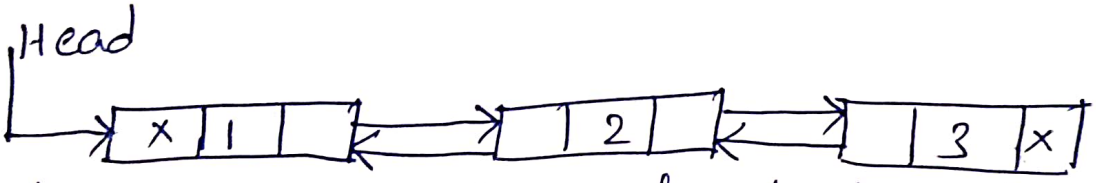
Types of linked lists :-

① **Singly linked list:-** It can be defined as the collection of ordered set of element. A node in a singly linked list consists of two parts `data part` and `next / link` part. Singly linked list can be traversed in only one direction.

Ex:- Marks obtained by a student in three subjects can be stored in a linked list.

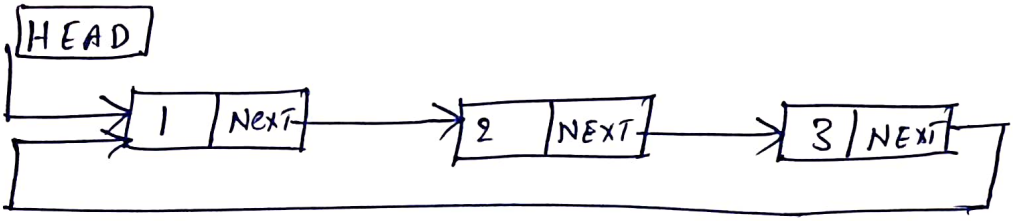


② **Doubly Linked list:-** Doubly linked list is a complex type of linked list in which a node contains pointer to the previous as well as the next node in the sequence. Therefore doubly linked list consists of three parts node data, pointer to the next node, pointer to the previous node.



Note:- prev part of the first node & next part of the last node will always contains the null value.

③ Circular linked list : In a Circular linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list, as well as circular doubly linked list. It has no begining & no ending. there is not null value present in the next part of any of the nodes.



Basic operations on Linked List

→ Creation of Linked list
- Creating nodes dynamically using malloc();
- Linking nodes by setting the "next" pointer

→ Insertion operations.
- At the begning (head insertion)
- At the end (tail insertion)
- At a specific position (index-based)
- After a given node
- Before a given node (require traversal)

→ Deletion operations
- From the begining
- From the end
- From a specific position
- By value.

→ Traversal / Display
- Traverse the entire list & display node data

Syntax void display list (Node * head);

→ Search operation :- Searches specific nodes in a list
- Find a node with a specific value

Syntax Node * search (Node * search, int key);

→ Count nodes :- Counts no of nodes in a list.
- counts & returns the no of nodes

Syntax :- int countNodes (Node * head);

→ Reverse the Linked list :- It reverse the list.

Syntax:- void reverselist (Node * * head);

→ Sort the linked list :- using sorting algorithms arranges list in a specific order like bubble sort, merge sort, ect.

Syntax:- void Sortlist (Node * * head);

→ Free the entire list
- Delocate the memory used by all nodes.

Syntax :- void freelist (Node * * head);

Q3) Write a program to create a singly linked list and display its elements.

```
#include <stdio.h>
#include <stdlib.h>
// define the node structure
struct Node {
    int data;
    struct Node * next;
};
// function to create a new node
struct Node * CreateNode (int value) {
struct Node * newNode = (struct Node *)
    malloc (size of (struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed \n");
        exit(1);
    }
        newNode -> data = value;
        newNode -> next = Null;
        return newNode;
    }
    // function to display the linked list
void display list (struct Node * head) {
        struct Node * temp = head;
```

```
Printf ("Linked list Elements : ");
    while (temp != NULL) {
        printf ("%d -> ", temp -> data);
        temp = temp -> next;
    }
    printf ("NULL \n");
}
// Main function
int main() {
    Struct Node *head = NULL;
    Struct Node * temp = NULL;
    int n, value;
    printf ("Enter the no of nodes :");
    Scanf ("%d", & value);
    Struct Node * new Node = Create Node(value);
    if (head == NULL) {
        head = new Node;
        temp = head;
    } else {
        temp -> next = new Node;
        temp = new Node;
    }
}
    display list (head);
    return 0;
}
```

Output

```
Enter the number of nodes : 4
Enter value for node 1: 10
Enter value for node 2: 20
Enter value for node 3: 30
Enter value for node 4: 40
Linked list elements : 10 -> 20 -> 30 -> 40 -> NULL
```

Q3) Write a program to insert a node at a specific position in a doubly linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Structure for a doubly linked list node
struct Node {
    int data;
    struct Node * prev;
    struct Node * next;
};
// function to create a new node
struct Node * createNode(int data) {
    struct Node *newNode = (struct Node *)
    malloc (sizeof (struct Node));
        newNode -> data = data;
        newNode -> prev = NULL;
        newNode -> next = NULL;
        return newNode;
    }
// function to display the doubly linked list
    void displayList(struct Node * head) {
        struct ("Doubly linked List : ");
        while (temp != NULL) {
            printf ("%d <-> ", temp -> data);
            temp = temp -> next;
        }
```

```
    printf ("NULL \n");
}
// function to insert at a specific position
Struct Node * insertAtPosition( struct Node *head,
    int data, int position) {
    struct Node * newNode = CreateNode (data);
    if (position == 1)
      {
      newNode -> next = newNode;
      head = newNode;
      return head;
    }
    struct Node * temp = head;
    int i;
    for (i = 1; i < position - 1 && temp != NULL;
        i++)
      {
        temp = temp -> next;
    }
    if(temp == NULL)
      {
        printf(" position out of range. Inserting
        at end. \n");
        return head;
        }
```

```
newNode -> next = temp -> next;
newNode -> prev = temp;
  if (temp -> next != NULL)
     temp -> next -> prev = newNode;
temp -> next = newNode;
 return head;
}
 // Main function
int main () {
    Struct Node * head = NULL;
    int n, data, position;
 printf (" Enter the no of initial nodes : ");
 Scanf ("%d", &n);
  for (int i = 1 ; i <= n; i++ ) {
    printf (" Enter data for node %d : ", i);
 head = insertAtposition( head, data, i);
                                  // inserting at the
                                     end
}
   printf ("\n original ");
   displayList(head);
   printf("\n Enter data to insert: ");
  Scanf ("%d" , & data);
 printf ( " Enter a position to insert at: ");
 Scanf( "%d", & position);
```

```
printf("\n after insertion");
display List (head);
return 0;
}
```

Output :-

Enter the no of initial nodes : 3

Enter data for node 1 : 10

Enter data for node 2 : 20

Enter data for node 3 : 30

originally Doubly Linked List : 10 <-> 20 <-> 30 <-> NULL

Enter data to insert : 15

Enter position to insert at : 2

After insertion doubly Linked List :

10 <-> 15 <-> 20 <-> 30 <-> NULL.

Q4) Write notes on storage classes, preprocessor directives and structure of a c program.

Storage classes in 'C' :- Storage classes in "C" determine the scope, lifetime, and memory location of variables. They specify how memory is allocated and deallocated for variables. The four main storage classes are :

auto :-
The default storage class for local variables. They are created when a block is entered and destroyed upon ~~existing~~ exiting the block.

extern :- Used to declare global variables of functions that are defined in another file, making them accessible across multiple source files.

static :- For local variables, static preserves their values between function calls. For global variables, it limits their visibility to the current source file.

Register :- Suggests that the variable should be stored in a CPU register for faster access, though modern compilers often optimize this automatically

Preprocessor Directives :-

Preprocessor directives are instructions processed by the 'C' preprocessor before compilation. They begin with a # symbol. Common directives include:

#include : inserts the content of a specified header file into the current source file.

#define : creates macros, which are symbolic constants or function-like substitutions.

#ifdef, #ifndef, #else, #endif :

Used for conditional compilation, allowing parts of the code to be included or executed based on defined macros.

~~#progr~~ #pragma : provides compiler-specific instructions.

Structure of a 'C' program

A typical 'C' program follows a structured format:

Preprocessor (or) Directives : includes header files and defines macros (eg #include <stdio.h>)

Global declarations : Declaration of global

Variables and function prototypes that are accessible throughout the program.

main() function : The entry point of the program. Execution begins here.

→ Local Declarations : Variables declared with main() (or) other functions.

→ Statements : Instructions that perform operations.

User-defined functions : optional functions defined by the programmer, called from main() (or) other functions.

Q4) Write a program to display classes, preprocessors directives and structures.

```
#include <stdio.h>  // preprocessor directive
int globalVar=10;  // Global declaration
void myfunction();  // function prototype
int main()
{
int localVar= 5;  // Local declaration
printf("Global variable : %d\n", globalVar);
                                  // Statement
printf("Local variable : %d\n", localVar);
                                  // Statement
myfunction();  // function call
return 0;
}
void myfunction()  // user-defined function.
{
printf("Inside myfunction.\n");
}
```

Output

```
Global variable : 10
Local variable : 5
Inside myfunction.
```

Q4) Write a program to "#define" using a macro for constants.

```
#include <stdio.h>
// macro definition
#define PI=3.14159
   int main()
      {
   float radius, area;
printf("Enter the radius of the circle :");
scanf("%f", &radius);
area = PI * radius * radius;
printf("Area of the circle = %.2f \n", area);
return 0;
}
```

Output :

```
Enter the radius of the circle :5
Area of the circle = 78.54
```

Q4) Write a program to demonstrating "static" variable behavior in "c"

```
#include <stdio.h>
Void counter function() {
  Static int count = 0;  // Static variable,
                            initialised only once
  count++;
  printf(" Function called %d times \n", count);
}
int main()    // calling function multiple
                times.
  counter function();
  counter function();
  counter function();
  return 0;
}
```

output

Function called 1 times
Function called 2 times
Function called 3 times

Q5)

A) Write notes about Graphs, Graph representations and Graph traversals.

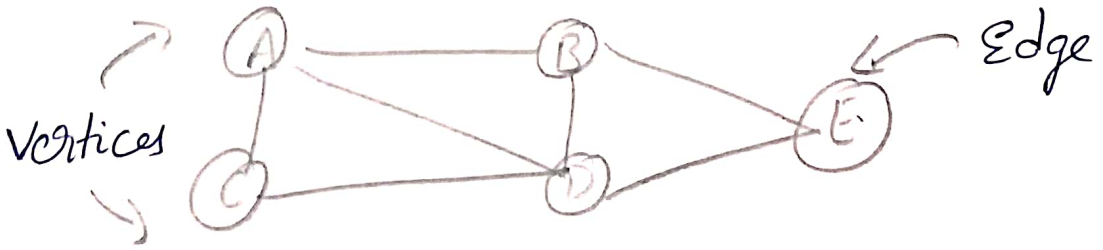
A graph is a non-linear data structure. It contains a set of points known as nodes (or) vertices and a set of links known as edges (or) Arcs. Here edges are used to connect the vertices. "Graph is a collection of nodes & edges in which nodes are connected with edges".

Ex:- The following is a graph with 5 vertices & 6 edges.

This graph G can be defined as $G = (V, E)$

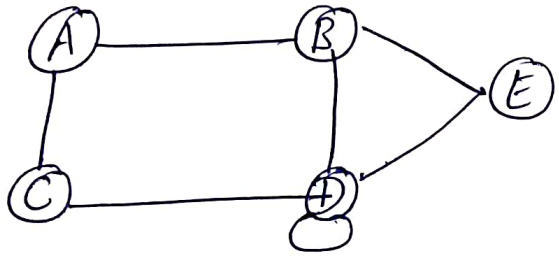
where $V = \{A, B, C, D, E\}$ and

$E = \{(A,B)(A,C)(A,D), (B,D), (C,D), (B,E)(E,D)\}$



Graph Representations Graph data structure is represented using following ways.

① Adjecency Matrix:- In this representation, the graph is represented using a matrix of size total no of vertices by the a total no of vertices. For example, consider the following undirected graph representation....



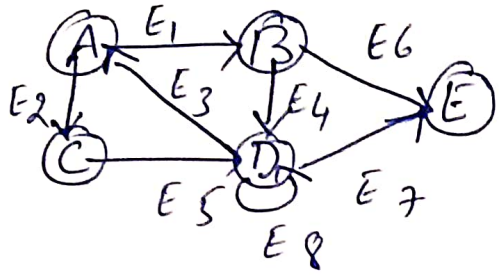
Directed graph representation

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

② Incidence Matrix :- In this representation, the graph is represented using a matrix of size total no of vertices by a total no of edges. That means 4 vertices and 6 edges is represented using a matrix size of 4x6. This matrix is filled with 0 or 1 or -1. Here "0" represent row edge is not connected to column vertex, 1 represents that row edge is connected as the outgoing edge to the column vertex and -1.

represents that the row edge is connected at the incoming edge to column vertex.

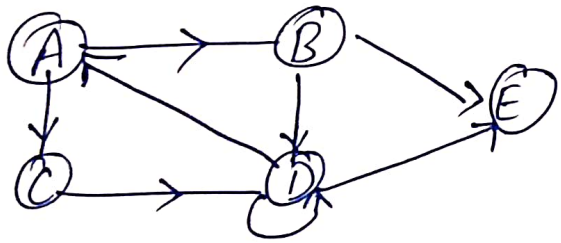
for example.



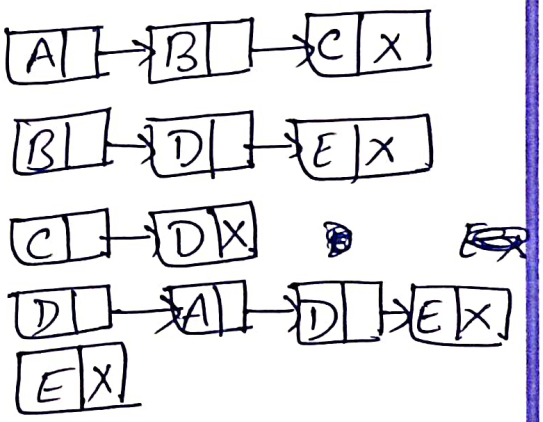
	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8
A	1	+1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	-1	-1	0	

(3) Adjecency List In this graph, every vertex of a graph contains list of its adjecent vertices.

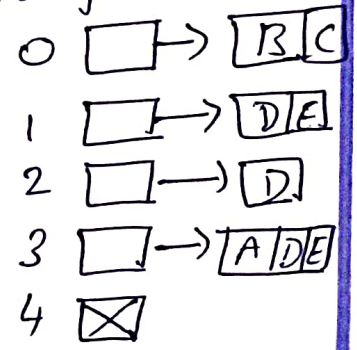
for example- following directed graph represents implemented using linked list.



⇒



This representation can also be implemented using an array. Reference array



Graph Traversal Techniques :-

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process.

There are two graph traversal techniques.

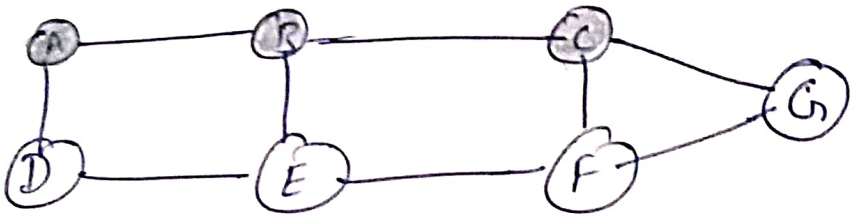
① DFS (Depth First Search) :-

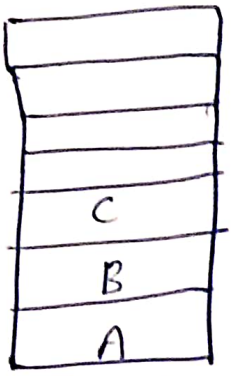
DFS traversal of a graph produces a spanning trees as result. Spanning Tree is a graph without loops. We use stack data structure with maximum size of total no of vertices in the graph to implement DFS traversal.

Steps to be followed for DFS travel

1. Define a stack of size with total no of vertices
2. Select any one point as "starting" traversal
3. visit any one of the "adjacent" vertices at the top of the stack
4. Repeat step 3 until there is no new vertex
5. when there is no vertex use back tracking & pop one.
6. Repeat step 3, 4, 5 until becomes empty.
7. when Stack becomes empty, then produce final spanning tree by removing unused edges from graph.

Sample diagram of stacks





Stack

(2) BFS (Breadth first search)

BFS traversal of a graph produces a spanning tree as a final result. Spanning tree is a graph with out loops. We use queue data structure with maximum size of total no of vertices in this graph.

steps involved in BFS

(1) Define queue of size total no of vertices in the graph

(2) select any starting point of the vertex

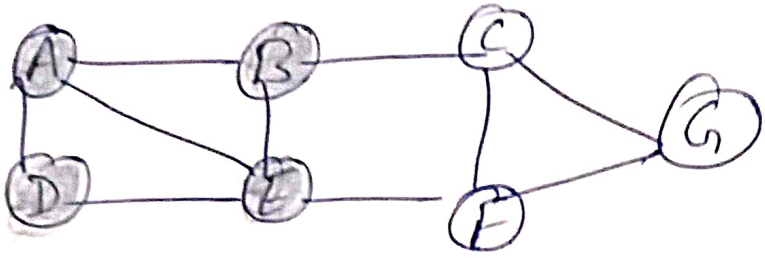
(3) visit all non-visited adjacent vertices. and insert them into queue.

(4) when no new vertex is visited delete the queue infront of them.

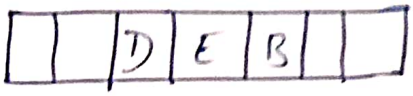
(5) Repeat steps 3 & 4 until queue becomes empty

⑥ When queue becomes empty produce final spanning tree by removing unwanted edges.

Sample queue diagram



Queue



Q5) Write a program to implement Stack using linked list.

```
#include <stdio.h>
#include <stdlib.h>

// node structure for linked list
struct Node {
    int data;
    struct Node *next;
};
// top of the stack
struct Node *top = NULL;
// function to push an element on to the stack
void push(int value)
{
struct Node *newNode = (struct Node *)
    malloc(sizeof(struct Node));
if (!newNode) {
    printf("Heap overflow\n");
    return;
}
```

```
}
newNode -> data = value;
newNode -> next = top;
top = newNode;
printf("pushed %d onto the stack\n", value);
}
// Function to pop an element from the Stack
void pop() {
    if (top == NULL) {
        printf("Stack underflow\n");
        return;
    }
    struct Node *temp = top;
    printf("popped %d from the stack\n", temp->data);
    top = top -> next;
    free(temp);
// function to display stack elements
void display() {
    if (top == NULL) {
        printf("stack is empty\n");
        return;
    }
```

```
struct Node *temp= top;
  printf ("stack elements:");
  while (temp!= NULL) {
    printf ("%d", temp->data);
     temp= temp->next;
  }
    printf ("\n");
  }
// Main menu
  int main () {
     int choice, value;
   while (1) {
   printf ("\n-- Stack Menu---\n");
   printf (" 1 Push \n 2. POP \n 3. Display \n
                4. Exit \n");
printf ("Enter your choice:");
 scanf ("%d", &choice);
switch (choice) {
 Case 1:
    printf("Enter value to push: ");
    scanf("%d", &value);
    push(value);
    break;
```

```
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting---\n");
            exit(0);
        default:
            printf("Invalid choice!...Try
                                again");
    }
}
    return 0;
}
```

Output

```
---Stack Menu---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 10
Pushed 10 into the stack.
```

--- Stack Menu ----

Enter your choice: 1

Enter value to push: 20

pushed 20 onto the stack

---- Stack Menu ----

Enter your choice: 3

Stack elements : 20, 10

--- Stack Menu ---

Enter your choice : 2

popped ~~from~~ 20 from Stack.

--- Stack Menu ----

Enter your choice : 3

Stack elements : 10

output

```
Enter number of elements : 5
Enter elements :
3 8 2 7 5
Sorted array :
2 3 5 7 8
```