DATA STRUCTURES USING C

Assignment - Set I PAPER - II

1. Write notes on various operators, Conditional and Looping Statements available in C With Suitable Syntax and example.

operators in c are symbols or tokens that perform operations on operands. operands can be variable, constants, expressions, or function calls. operators allow you to manipulate data and perform various tasks like arthmetic operations, comparisons, Logical operations, bitwise operations, and more.

C language supports wide range of operators categorized into different types based on their functionality and usage. These operators play a crucial role in writing efficent and expressive C programs.

Arithmetic operators: These are used to perform mathematical operations.

+ Addition
- Subtraction
* Multiplication
/ Division
% modulus Division

Realational operators : These are used to compare values.

==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Logical operators These are used to perform Logical operations.

&&	Logical AND
||	Logical OR
!	Logical NOT

Assignment operators These are used to assign value to variables.

=	Simple assignment
+=	Add and assign
-=	Subtract and assign
*=	multiply and assign
/=	Divide and assign
%=	Modulus and assign

Increment and Decrement operators

++ Increment by 1
-- Decrement by 1

Bitwise operators: These operate on bits and perform bit-level operations.

& Bitwise AND
| Bitwise OR
^ Bitwise XOR
~ Bitwise NOT
<< Left Shift
>> Right Shift

Conditional operator (Ternary operator) it's a shorthand for if-else statements.

```
Condition ? expression1 : expression2
```

Comma operator it evaluates multiple expressions and return the value of the last expression.

```
expr1, expr2
```

Size of operator it returns the size of a variable or data type.

```
sizeof(type)
```

Conditional Statement (Syntax and example)

Syntax condition? statement1: statement2;

ex:- a>b? printf("A is Big\n"): printf("B is Big\n");

Looping statement (Syntax and example)

- While Loop

Syntax:

```
While(condition)
{
 Statement 1;
 Statement 2;
 ----
 Increment/decrement;
}
```

ex:- print all integers from 1 to 40

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i=1;
 While(i<=40)
 {
   printf("%d\n",i);
    i++
}}
```

- do While Loop

Syntax :
```
do
{
 Statement body;
}
 While(Condition);
```

example:- To read in a number from the keyboad untill a value in the range 1 to 10 is entered.

```
int i;
do
{
 Scanf("%d\n", &i);
  -flushall();
 }
 While(i<1 &&i>10);
```

- for Loop

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

example :- print out all numbers from 1 to 100

```
#include<stdio.h>
void main()
{
 int x;
for (x=1; x<=100; x++)
   Printf("%d \n", x);
}
```

(2.) Discuss briefly about Functions, call-by-value, call-by-reference and Recursion.

In C arguments are passed to functions using the call-by-value (CBV) scheme. This means that compiler copies the value of the arguments passed by the calling function into the formal parameter list of the called functions. Thus if we change the values of the formal parameters within the called function we will have no effect on the calling arguments. The formal parameter of a function are thus local variable of the function are created upon entry and destroyed on exit.

for example program to add two numbers (CBV)

```
#include <stdio.h>
int add(int, int);
void main()
{
 int x, y;
 puts("Enter two integers");
```

```
Scanf("%d %d", &x, &y);
Printf("%d + %d = %d \n", x, y, add(x,y));
}
int add(int a, int b)
  {
   int result;
   result = a+b;
   return result;
  }
```

The add() function here has three local variable, two formal parameters and the variable result. There is no connection between the calling arguments, x and y, and the formal parameters a and b other than that the formal parameter are initialized with the value in the calling arguments when the function is invoked. The situation is depicted below to emphasize the independence of the various variable

for example:- program that attempts to swap the value of two number (Call-by-Reference)

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

```
Void swap(int* ,int*);

Void main()
{
 Int a,b;

 Printf("Enter two numbers");
 Scanf("%d%d", &a, &b);
 Printf("a = %d ; b = %d \n",a,b);
 Swap(&a, &b);
 Print ("a= %d ; b= %d \n", a,b);
}
 Void swap(int *x, int *y)
 {
  Int t;
  t = *x;
  *x = *y;
  *y = t;
}
```

Since C uses call by value to pass parameters What we have actually done in this program is to swap the value of the formal parameters but we have no changed the value in main(). Also since we can only return one value the return statement we must find some other means to alter the value in the calling function.

Recursion

A recursive function is a function that calls itself either directly or indirectly through another function. Recursive function calling is often the simplest method to eventually simplified into a series of more basic operations of the same type as the original complex operation.

This especially true of certain types of mathematical functions. for example to evaluate the factorial of a number, n

$n! = n * n-1 * n-2 * \ldots * 3 * 2 * 1.$

```
int Fact(int n)
 {
 if(n<=1)   //terminating condition
    return 1;
   else
   return (n* Fact (n-1));
 }
```

we can simplify this operation into

$n! = n * (n-1)!$

where the original problem has been reduced in complexity slightly. we continue this process

Untill we get the problem down to a task that may be solved directly. In this case as far as evaluating the factorial of 1 which is simply 1.

So a recusive function to evaluate the factorial of a number will simply keep calling itself until the argument is 1. All of the previous (n-1) recursive calls will still be active waiting until the simplest problem is solved before the more complesc intermediate steps can be built back up giving the final solution.

3

Write notes on stacks, Queues and programs to implement all operations on stacks and Queues.

Stack is a linear data structure (an ordered list) in which all insertions and deletions are performed through a single end called "top". Stack is also called as a LIFO (Last in First out) list or a FILO (First in Last out) list.

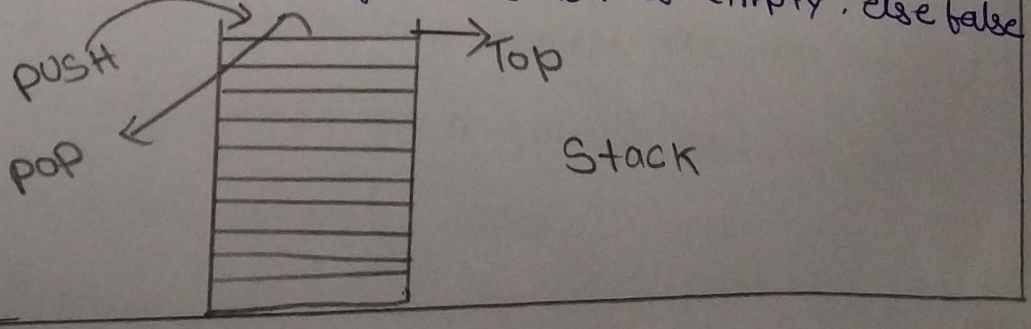
The following are the basic operations performed in the stack

Push Adds an item in the stack. If the stack is full, then it is said to be an overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.

Peek or Top Returns the top element of the stack.

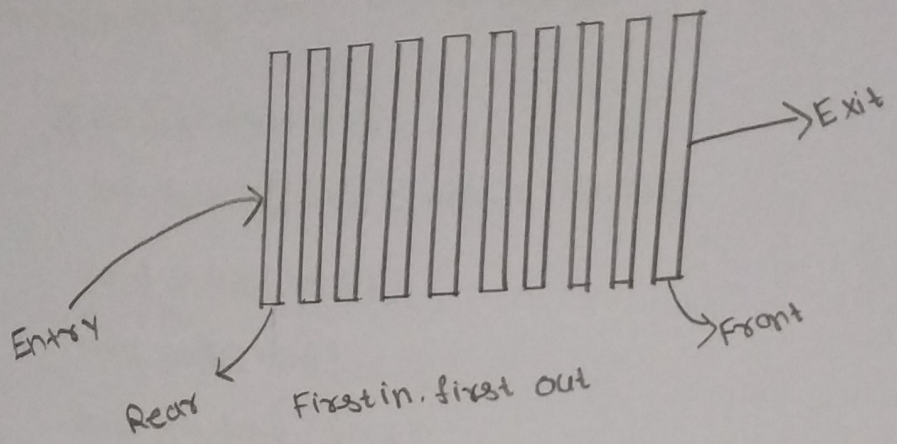
IsEmpty Returns true if the stack is empty, else false



There are many real life examples of stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed i-e the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So it can be simply seen to follow the LIFO/FILO order.

Queue Data Structure Like Stack

Queue is also a linear structure (an ordered list), in which all insertions are performed from the "rear" end and all deletions are performed from the "front" end. A Queue is also called as a First In First out (FIFO) list. A good example of queue is any queue of consumers for a resource where the consumes that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; In a queue, we remove the item the least recently added.



Operations on queue

The following are the basic operation performed on queue

Inset/Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an overflow condition

Delete/Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. if the queue is empty, then it is said to be an underflow condition.

Front: Get the front item from queue.

Rear Get the last item from queue.

Implementation

There are two ways to implement a Stack

- using Array
- using linked List

Implementing Stack using Arrays

```
#include<stdio.h>
 int stack[100], choice, n, top, x, i;
 void push(void);
 void pop(void);
 void display (void);
 int main()
{
 top = -1;
 Printf("\n Enter the size of STACK [max =100]: ");
 Scanf("%d", &n);
 Printf ("\n\t STACK OPERATIONS USING ARRAY ");
 Printf("\n\t----------");
 Printf("\n\t 1. PUSH\n\t 2. POP\n\t 3. DISPLAY\n\t
                                    4. EXIT");
do
{
 Printf("\nEnter the choice:");
 Scanf("%d, &choice);
 switch(Choice)
 {
   case 1: push();
           break;
```

```
    case 2: pop();
            break;
    case 3: display();
            break;
    case 4: printf("\n\t EXIT POINT");
            break;
    default: printf("\n\t plese enter a valid choice");
    }
}
While(choice!=4);
    return 0;
}
void push()
{
  if (top>=n-1)
  {
   printf("\n\tstAck is FULL, Push operation not
                        possible\n");
  }
  else
  {
    printf("Enter a value to be pushed:");
    scanf("%d", &x);
    stack[++top]=x;
```

```
}
}
void pop()
{
 if (top<=-1)
 {
 printf("\n\t Stack is empty\npop operation not
                                possible \n");
 }
 else
 {
 printf("\n\t The popped elements is %d", stack[top--]);
 }
}
void display()
{
 if (top>=0)
 {
 printf("\n The elements in STACK \n");
 for(i=top; i>0; i--)
     printf("\n%d", stack[i]);
   printf("\n Press Next choice
 }
 else
 {
   printf("\n The STACK is empty");
 }
 }
```

OUTPUT

```
Enter the size of STACK [MAX = 100]: 10

    STACK OPERATIONS USING ARRAY
    ----------------------------

1. PUSH
2. POP
3. DISPLAY
4. EXIT

Enter the choice: 1
Enter a value to be pushed: 2

Enter the choice: 1
Enter a value to be pushed: 24

Enter the choice: 1
Enter the value to be pushed: 98
Enter the choice: 3
The elements in STACK
98
24
12

Press Next choice
Enter the choice: 2

        The popped elements is 98

Enter the choice : 3
The elements in STACK
24
12

Press Next choice
Enter the choice: 4
```

Implementing stack using Linked List

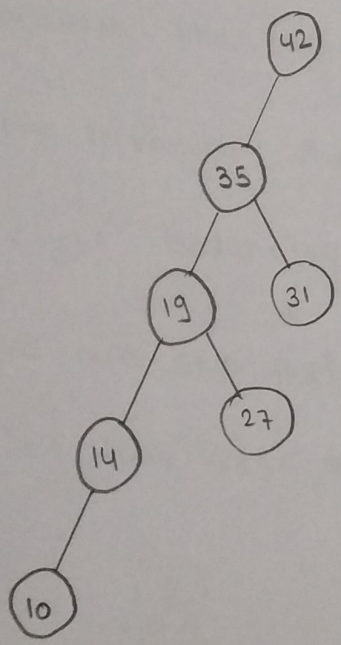
```
# include <stdio.h>
# include <stdlib.h>

// Define a structure for each element of th stack

Struct StackNode * next;
};

// Function to create a new node Struct StackNode * newNode
                                        (int data)
{
Struct StackNode * StackNode = (Struct StackNode *)
  malloc(Size of (Struct StackNode));
  StackNode->data = data;
  StackNode->next = NULL;
  return StackNode;
}
int isEmpty( Struct StackNode *root)
{
 return !root;
}

void push( Struct StackNode = newNode(data);
  StackNode->next = *root;
  *root = StackNode;
  Printf ("%d pushed to stack\n", data);
}
int pop (Struct StackNode** root)
```

```
{
if (isEmpty(*root))
  {
  printf("stack underflow\n");
  return -1;
}
struct stackNode * temp = * root
 * root = (*root)->next;
 int popped = temp->data;
 free(temp);
 return popped;
}
 void display (struct stackNode*root)
{
 if (isEmpty (root)){
  printf("stack is empty\n");
  return;
}
 struct stackNode * temp=root;
 printf(" stack elements are:\n");
 while (temp != NULL){
  printf("%d\n ", temp->data);
  temp = temp->next
 }
}
int main()
{
```

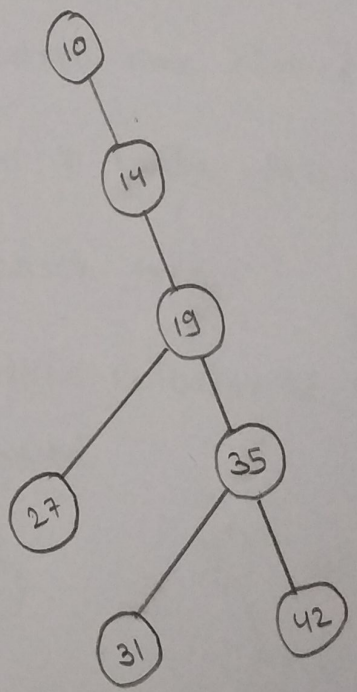
```
Struct StackNode * root = NULL;

  Push (&root, 10);
  Push (&root, 20);
  Push (&root, 30);

  display (root);

Printf ("%d popped from stack\n", pop (&root));

Printf ("%d popped from stack\n", pop (&root));

  display (root);
  return 0;
}
```

4. Briefly discuss about AVL Tree and the insertion operation on AVL Tree with suitable Rotations and examples.

AVL Tree if the input to binary search tree comes in a sorted (ascending or descending) manner; it will then look like this



if input appears non-increasing manner



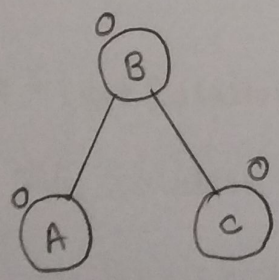
if input appears in non-decreasing

It is observed that BST's worst-case performance is closest to Linear Search algorithms, that is O(n) In real-time data, we cannot predict data pattern and their frequencies So, a need arises to balance out the existing BST.

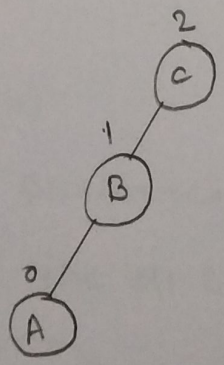
AVL trees are binary search trees in which the difference between the hight of the left and right subtree is either -1, 0, or +1. This difference is called the Balance Factor.

AVL trees are also called a self-balancing binary search tree. These trees help to maintain the logarithmic search time. Named after their inventor Adelson, velski & Landis. AVL tree are height balancing binary search tree.

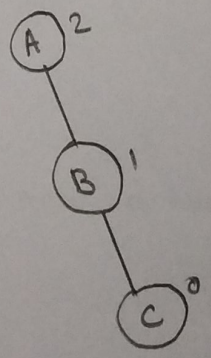
Here we see that the first tree is balanced and the next two tree are not balanced.



Balanced



Not Balanced



Not Balanced

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the

right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again

AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height (left-subtree) - height (right-subtree)

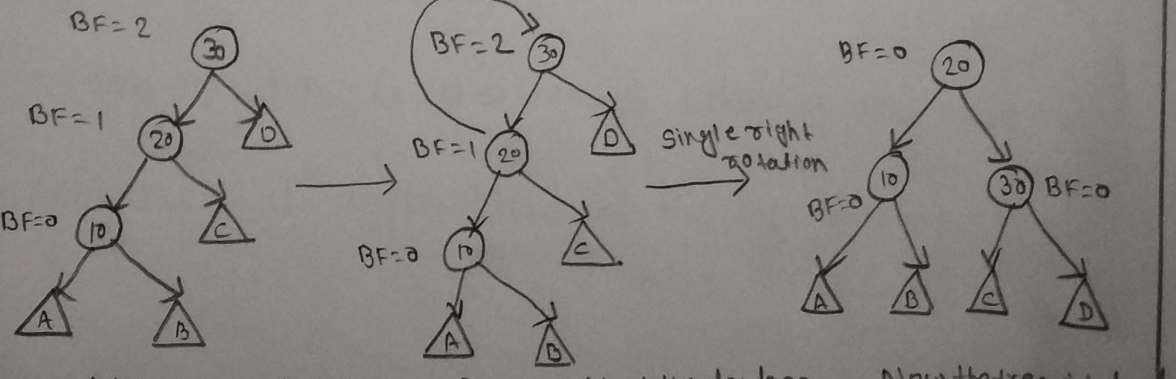
AVL Roatations

To make the AVL Tree balance itself, when inserting or deleting a node from the tree, rotations are performed.

We performed the following LL rotation, RR rotation. LR rotation, and RL rotation

Left - Left Rotation (LL)

This rotation is performed when a new is inserted at the left child of the left subtree.

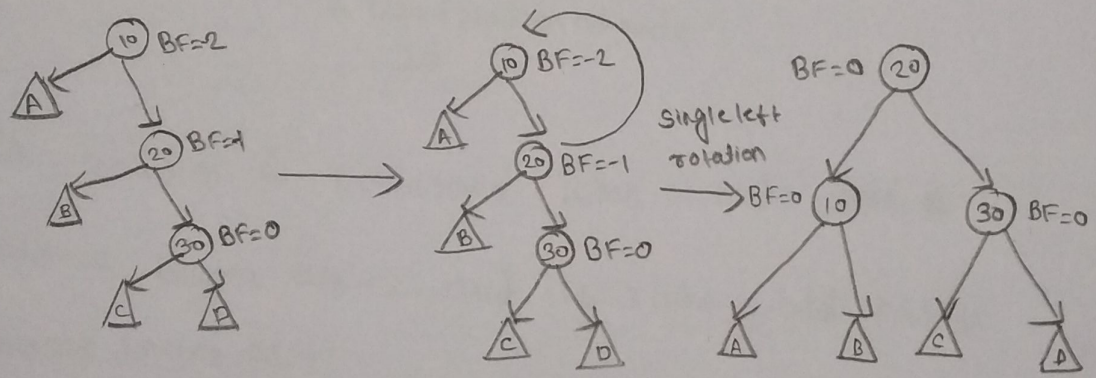


Tree is imbalance as BF(30)=+2 Perform right rotation at node 20 Now the tree is balanced

A single right rotation is performed. This type of rotatin is identified when a node has a balanced factor as +2. and its left-child has a balance factor as +1

Right - Right Rotation (RR)

This rotation is performed when a new node is ingerted at the right child of the right subtree.



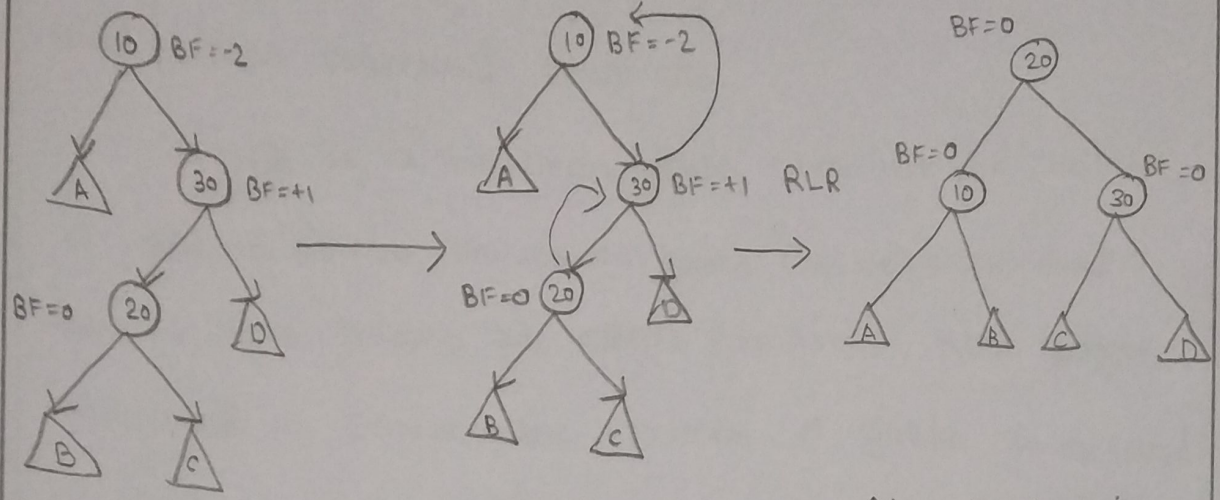
Tree is imbalance as BF(10)=-2

Perform a left rotation at node 20

Now the tree is balanced

Right -'Left Rotation (RL)

This rotation is performed when a new node is inserted at right child of the left subtree.



Tree is imbalance as BF(10) = -2

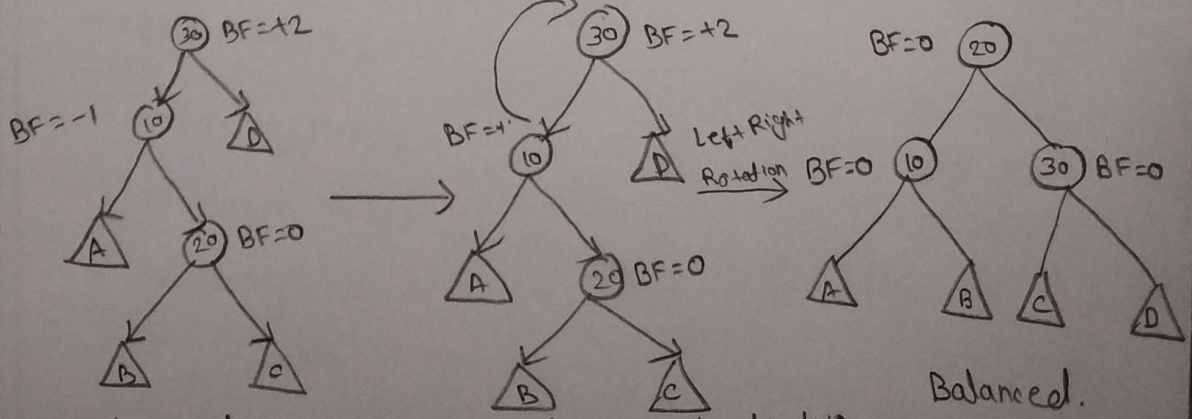
perform a right rotation at node 20 & a left rotation at node 30

Now the tree is balanced.

This rotation is performed when a node has a balance factor as -2, and its right child has a balance factor as +1

Left - Right Rotation (LR)

This rotation is performed when a new node is inserted at the left child of the right subtree.



Tree is imbalance as BF(30) = -2 node 20 & a right at node 10

Balanced.

5

Discuss in detail about Graph, Graph Representations and Graph Traversal Technique.

Graph is a non-Linear data structure. It contains a set of points knows as nodes (or vertices) and a set of links knows as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follow

Graph is a collection of vertics and arcs in which vertices are connected with arcs.

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph G is represented as G = (V, E) where V is set of vartices and E is set of edges.

Graph Representations

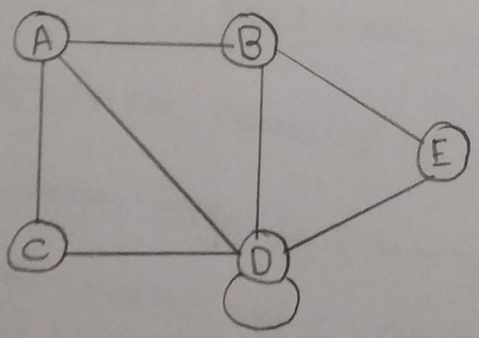
Graph data structure is represented using following representation

1 Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

Adjacency Matrix

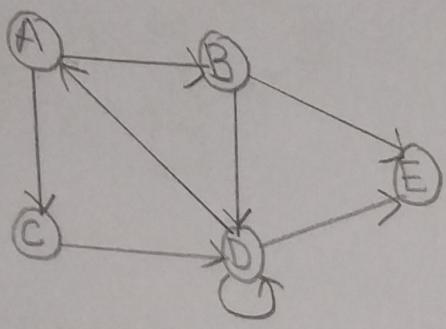
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That mean a graph with 4 vertices is represented using a matrix of size 4X4 In this matrix, both rows and colums represent vertices. This matrix is filled with either 1 or 0. Here 1 represents that there is an edge from row vertex to colomn vertex and 0 represents that there is no edge from row vertex to colomn vertex.

for example, consider the following undirected 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

Directed graph representation



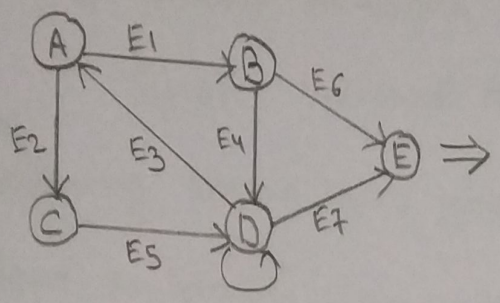
$$\Rightarrow$$

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

Incidence Matrix

In this representation, the graph is represented using matrix of size total number of vertices by total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1 Here, 0 represents that the row edge is not connected to column vertex. 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex

For example, consider the following directed graph

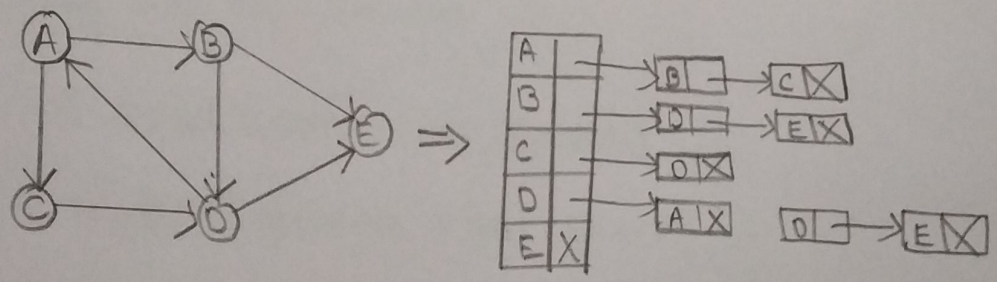


$$\Rightarrow \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{pmatrix} E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{pmatrix}$$

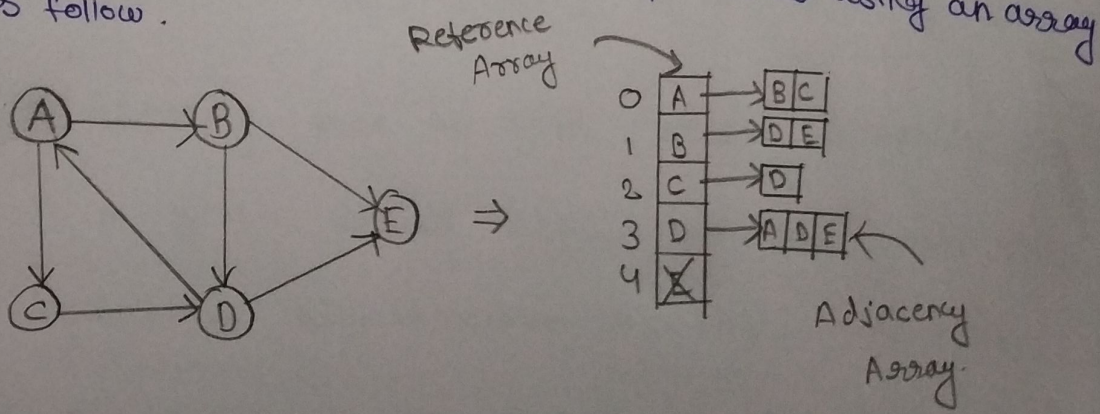
Adjacency List

In this representation, every vertex of a graph contains List of its adjacent vertices.

For example. Consider the following directed graph representation implemented using linked list.



This representation can also be implemented using an array as follow.



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. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal technique and they are as follows

1) DFS (Depth First Search)
2) BFS (Breadth First Search)

DFS (Depth First Search)

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops we use stack data structure with maximum size of total number of vertices in the graph to implement DFS

We use the following Steps to implement DFS traversal

Step 1- Define a stack of size total number of vertices in the graph.

Step 2- Select any vertex as starting point for traversal. visit that vertex and push it on the stack

Step 3- visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on the stack.

Step 4. Repeat Step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5. When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

Step 6. Repeat Steps 3, 3, 4 and 5 until stack becomes Empty

Step 7: When stack becomes Empty then produce final spanning tree by removing unused edges from the graph.

BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result Spanning Tree is a graph without loops. We use Queue data Structure with maximum size of total number of vertices in the graph to implement BFS traversal

We use the following Steps to implement BFS traversal

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2. Select any vertex as starting point for traversal Visit that vertex and insert it into the Queue.

Step 3. Visit all the non-visited adjacent vertices of the vertex. Which is at front of the Queue and insert them into the Queue.

Step 4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5. Repeat step 3 and 4 until queue becomes empty

Step 6. When queue becomes empty, then produce final spanning tree by removing unused edge from the graph.