Date
Page

Ques→ What is Data Structure? What are its classifications?

Ans→ Data Structure is a storage format that is used to store and organise and data. It is a way of arranging data on a computer so that it can be accessed and updated easily. In other words, data structures refers to the way of assembling or organising data so that it can be used efficiently. It is a particular way of storing data in computer's memory so that it can be used easily. The choice of a good data structure makes it possible to perform a variety of critical operations effectively. An efficient data structure uses minimum memory space and execution time to process the structure as possible. The choice of choosing the right data structure may depend upon the size of program or time complexity of the program. The data structure only deals with the organization of data in main memory (RAM) and not in secondary memory. Hence, data structure is one of the factors that is responsible for the ~~execution~~ time taken by the program to load in Main memory.

CLASSIFICATIONS ☞

Data structures are classified into four categories

1) Linear and Non-Linear D.S.

In a linear d.s. the elements are arranged in a sequence so that processing of elements can be possible in linear manner. In non-linear d.s. the elements are not arranged in a sequence manner.

Date
Page

2) Homogeneous and Non-Homogeneous D.S.
In Homogeneous D.S., all elements are of same datatype & in heterogenous D.S., all elements are of different datatype

3) Static and Dynamic Data Structure
The D.S. whose size is fixed is known as static D.S. and whose size is not fixed is known as Non-static data structure.

4) Primitive & Non-Primitive D.S.
Primitive D.S. is the D.S. that can not be further subdivided.
Non-Primitive D.S is the D.S. that can be further sub-divided into various parts.

Ques Define array and its types. Also discuss the advantages and disadvantages of Arrays.

Ans→ Arrays ⇒ An array is a collection of homogeneous data elements of a specific datatype that have been given a single name, which are stored in contigious memory location. All the elements of array can be accessed by using its index number enclosed in square brackets after the array name.

Ex int a[5];

Or

a[5] = 10;

Date
Page

Types of Arrays:-

1) While declaring an array, if only one square bracket is used then the array is one-dimensional.
2) While declaring an array, if two square brackets are used then the array is two-dimensional.
3) While declaring an array, if three or more square brackets are used then the array is multidimensional.

Example :⇒

```
# include <stdio.h>
void main()
{
    int a[10];          // 1-D
    int b[10][10];      // 2-D
    int a1[10][10][10]; // Multi-D
}
```

The index of array always starts from 0.
The array elements can be accessed by incrementing the index value by 1. We can access the array randomly and as well as sequentially.
By incrementing the value of index by 1 we can access the next block of the array. We can perform various operations on arrays like

1) Searching
2) Sorting
3) Traversing
4) Merging
5) Insertion
6) Deletion

Date
Page

5)

We can also initialize the array while we are declaring it

Ex:- int a[5] = { 1,2,3,4,5 }

By default, an array contains garbage value. But if we intialize less elements then the size of array during declaration, rest of the blocks will contain the value 0.

Advantages of Array :⇒

1) Array is the simplest kind of D.S.
2) Array is the most compact data structure i.e. it takes minimum possible space in memory.
3) Computation of vectors & matrices become easy.
4) Elements of array have direct access to them.
5) Array elements can be accessed instantly.
6) Arrays help programmers to refer to many values with a single name.

Disadvantages of Array :⇒

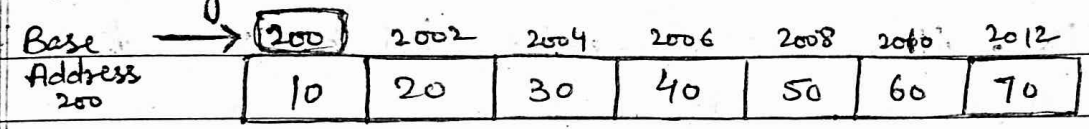
1) An array use reference mechanism to work with memory which can cause instable behaviour of Operating system.
2) The size of array should be known in advance i.e. before compile time.
3) Insertion and Deletion in case of array is very costly since a lot of data movement has to be done.
4) Sometimes, it's not easy to work with many ins

arrays.

5) Array holds the similar kind of information. So it is impossible to store unrelated information into an array.

Memory Representation of 1-D Array

Array elements are stored in continuous memory locations. Computer does not need to keep track of the address of every element of linear array, but only of the first element. The address of the First element of linear array is called Base Address.



Memory Representation of 2-D Array

In 2-D arrays, elements are arranged in rows and columns, But computer memory is linear. So, elements of 2-D array are mapped into linear memory. 2-D arrays can be stored in memory in two ways:

1) Column - major Order
2) Row - major Order

1) Column - major Order

In column - major order, the elements are stored column wise i.e. first column is stored, then second column & then third column and so on.

Date
Page

If we have 3*2 matrix, then elements will be stored as :-

A_{11}
A_{21}
A_{31}
A_{12}
A_{22}
A_{32}

2) Row - Major Order

In row-major order, the elements are stored row-wise i.e. first row is stored, then second row, then third row and so on. If we have 3*2 matrix, then the elements will be stored as :-

A_{11}
A_{12}
A_{21}
A_{22}
A_{31}
A_{32}

Date
Page

STACK

Stack is a linear d.S. that implements LIFO scheme. A stack is a simple d.S. that allows adding and removing elements only at one end. That end is called Top of the Stack i.e. items can be inserted or removed only at the top of the stack. The last item added to the stack will be first item to removed from stack.

Imagine stacking a set of books on top of each other - you can place a new book on the top of stack and you can remove the book that is currently on the top of stack. Size of stack depends upon number of elements currently present in the stack.

In D.S., we can represent stack as follows :-

Data Structure (Variable) used :

TOP → It always address top most element of stack.

Operations

PUSH → Pushing something on the stack means "placing it on top". Push operation is used to insert any element at the TOP of stack.

During a push operation, first we check availability of space in stack. If stack has reached its limit, it is overflow condition. In such case, we can not insert the element in stack. After push operation, TOP of stack is incremented by 1.

Date
Page

Push in Array Implementation of Stack.

In array representation, stack is represented using a 1-D array say 'stack' with size 'n'. Here 'n' is the limit of Stack. Following steps are followed to Push an element onto stack.

1) Firstly, we check for overflow. If Top = n, then we cannot insert a new element.
2) Then we check whether the stack is empty. If yes, i.e top = NULL, then we initialize top to 1 otherwise we simply increment top by 1 i.e Top = Top + 1
3) Then, we insert new element at the Top.

POP → Pop means "retrieve data" from the stack. Pop operation of the stack is used to remove any element from top of stack. After pop operation, top element is retrieved from stack and next element becomes top of Stack.

Pop in Array implementation of stack.

Following are the steps to pop an element from Stack.

1) Firstly, we check for underflow. If the stack is empty i.e if TOP = NULL then it is underflow
2) If underflow condition is false then we move to next step. In this step, we retrieve the Top element of stack.
3) Then we set the top pointer. If the popped element was the last element of stack i.e top was equal to

then top is set to NULL otherwise we simply decrement top by 1. i.e. Top = Top - 1

Queue

A queue is a particular kind of collection in which the entries in the collection are kept in order. A queue is a data structure where insertions are made at one end and deletions are made at another end. We add elements at the end also called as rear of queue and removes element from the front.
For ex!- Any new entry is made at the back of the queue and persons at the first are served first. Queue is an example of linear Data Structure.
In short, we summarize Queue as follows:

1) A list structure with two access points called the front and the rear.
2) All insertions occurs at the rear and deletions occur at the front.
3) Queue is of varying length.
4) Queue mostly stores homogeneous components.
5) Queue has FIFO characteristics.

Applications Of Queue

Queue has applications in day to day life as well as in computers.

1) In Real life

a) Queue in front of ticket window
b) Queue for admission
c) Queue for fee deposit
d) Queue at a bank counter
e) Queue waiting for bus at local bus stop.

2) In Computers

a) Multiprogramming - Queues are used to implement multiprogramming which is very important area of operating system. Multiprogramming requires queues for every field from processor management to device management.

b) Simulating Real life Queue - We can use queues for simulating real life simulations like banking.

c) Message Queue - Message queue is a good way to communicate from one application to another.

d) Playlist - Add songs to the end, play from the front of list.

Queue Operations

① Enqueue

Enqueue means insertion of new elements in the queue. In array representation of queue, we have an array 'queue' with maximum 'n' elements and two variables 'front' and 'rear' which points to the first and last element of queue. Initially, both front and rear are set to NULL. When we want to insert first element, we set both front and rear to 1. After that for each insertion, we check for overflow condition. It can be checked as

```
if ( Rear = n) then
    Print "Overflow"
    Return
[end if]
```

If no overflow, it means that space is available and we increment Rear pointer as

Rear = Rear + 1

After that, we insert new item in the Queue.

(2) Dequeue

Dequeue operation is used to delete an element from the queue. An element can be deleted only from the front of queue. Before Dequeue, we will check for underflow condition. It means that we will check for the availability of element. Underflow can be checked using the statement

```
If ( FRONT = NULL And REAR = NULL) then
    print " Underflow"
    Return
[end if]
```

If element is available i.e. front and rear are not NULL then the front element is deleted from the queue. After the deletion, we will set the values of front and rear. If front and rear both are equal then after deletion both front and rear will set to NULL; otherwise we will increment value of front by 1. In this case, there will be no change in rear

```
if ( front = rear) then
    a) set front = NULL
    b) set rear = NULL
else
        front = front +1
[end if]
```

Date
Page

Difference

Stack	Queue
1) The stack is based on LIFO principle.	The Queue is based on FIFO principle.
2) Insertion operation is called Push operation.	Insertion operation is called Enqueue operation.
3) Deletion operation is called Pop operation.	Deletion operation is called Dequeue operation.
4) Push and Pop operation takes place from only one end of stack	Enqueue and Dequeue operation takes place from different end of queue.
5) Only one pointer is used for performing operation	Two pointers are used to perform operations.
6) Empty condition is checked using TOP == -1	Empty condition is checked using FRONT==-1 && REAR==-1
7) Full condition is checked using TOP== Max-1	Full condition is checked using (FRONT==0 && REAR== Max-1) || (Front== REAR+1)
8) There are no variants available for stack	There are three types of variants :- circular, double-ended and priority Queue.

Date
Page

TREES

TREE TERMINOLOGY

1. **Node**: It is a key component of data structure and it stores the data and address of left child and right child.
2. **Edge**: Connection between two nodes is known as edges.
3. **Parent**: The immediate predecessor of a node is called its parent. All the nodes except root node exactly have one parent in level.
4. **Child**: The immediate successor of node is called child.
5. **Sibling**: Two or more nodes having same parents are known as siblings
6. **Root**: Top most node of Tree is called root of tree. Root Node has no parent.
7. **Leaf**: All the nodes having no children at any level are called leaf nodes.
8. **Path**: Path is the interconnection of nodes and edges to move from one node to another node.
9. **Level**: It is always an integer value which measures the distance from one node to another node in tree. Root is always at level 0 and child of root nodes are at level 1 and grandchild of root nodes are at level 2, and so on.
10. **Degree**: Maximum number of children that are possible for a node is known as degree of a node
11. **Successor**: Left and right subtrees of tree are called

as successors or children of node.

12. Ancestors and Decendant: The parents and grandparents of nodes are called the ancestors and children of a node and children of these children are called descendants.

13. Height / Depth of tree: The depth of a tree is maximum level of any node. It can be also described as the lengths of the longest path from the root node.

14. Complete Binary Tree: A tree is said to be complete if it has maximum number of nodes at each level except the last level. And if all the nodes at last level appear as far left as possible. A complete tree of depth d has maximum $2^{d+1} - 1$ nodes.

Properties of Binary Tree

1. Maximum number of nodes in binary tree at any level will be evaluated by 2^n. where "n" is the level at any given location.
2. Minimum number of nodes in binary Tree with height H is H+1.
3. If N is the number of nodes in binary tree then the number of edges E in a non-empty binary tree will be N-1.
4. Number of leaf nodes in a binary tree will be calculated with the formula as N+1 where N is the number of nodes with exactly two children.
5. For a complete Binary Tree T with N nodes, the

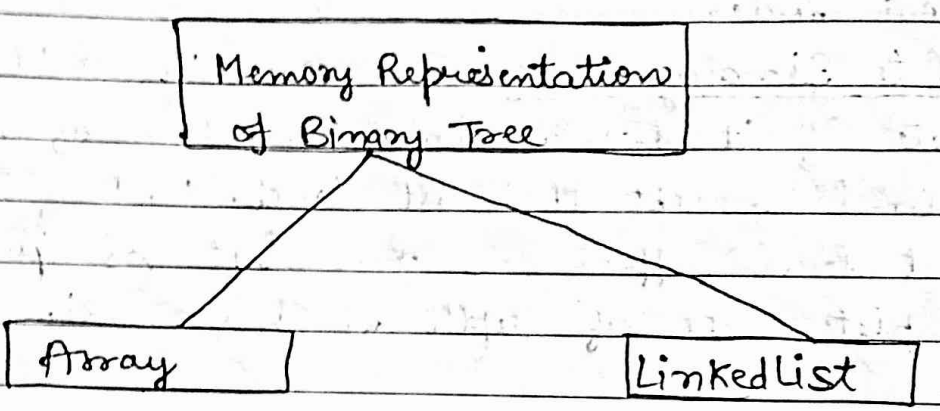
Date
Page

height is $\lfloor \log_2 (n+1) \rfloor$.

6. The number of leaf nodes in a complete Binary Tree containing N nodes is $(N+1)/2$, where N is the number of nodes in a tree.

7. A binary tree of height H has at most $2^{h+1} - 1$ elements.

Representation of Binary Tree in Memory



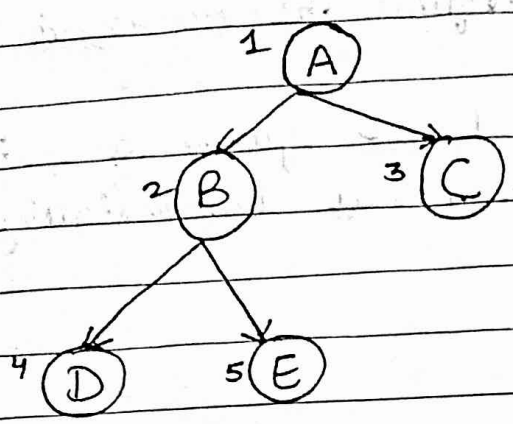
i) Array Representation of Binary Tree

The general data structure, array representation of binary Tree follows the normal convention. If a binary tree is complete binary tree, it can be represented using one array capable of holding n elements where index position -1 in array and left and right child will store at their corresponding position calculated using 2K and 2K+1 where K is the index position of parent node.

a) The 0th position is ignored

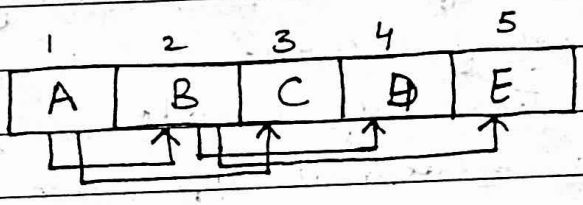
b) i) Left child : 2*K
   ii) Right child : 2*K+1

c) To find the parent of a given element we will use $\lfloor K/2 \rfloor$. Here, K is the index of particular element.

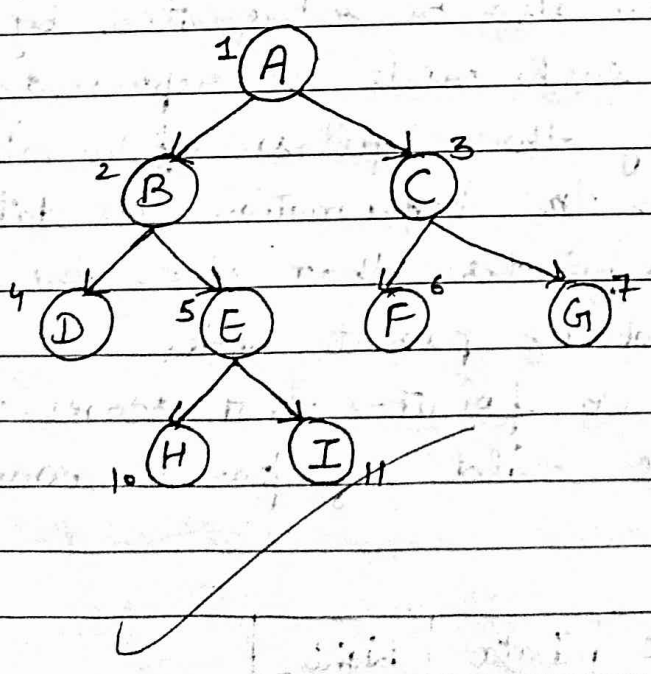


1	A
2	B
3	C
4	D
5	E

Array Tree



We can see that an array representation of complete binary Tree does not lead to the waste of any storage. But if you want to represent an incomplete binary tree using array representation, then it leads to the wastage of storage. Ex:-



Array Tree	
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	
9	
10	H
11	I

Date
Page

Advantages :-

1. This method does not require the overhead of maintaining pointers.
2. It is very simple method as given a child node, its parent can be determined immediately.

Disadvantages :-

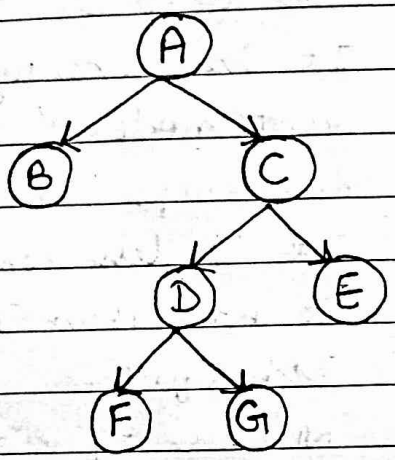
1. The sequential representation is usually inefficient to use. In case of incomplete Binary tree, a lot of wastage of memory occurs.
2. Insertion or deletion of a node requires considerable data movement up and down the array, using an excessive amount of processing time.

ii) Linked list Representation of Binary Tree

A binary tree can also be represented by linked list. In linked list, each node is represented by a structure having three fields of members.

a) Data : It stores the information or data.
b) Left : It is a pointer that stores the address of left child of parent node.
c) Right : It is a pointer that stores the address of right child of parent node.

Some programming languages like C, CPP support pointers, so the above mentioned concept can be implemented very easily. But the language which does not support pointer like FORTRAN 70, BASIC and then it is difficult to implement this concept. But the same concept can be implemented using three parallel linear array INFO, LEFT, RIGHT as shown below:-



Binary Tree

	LEFT		INFO		RIGHT
1	2	1	A	1	3
2	-1	2	B	2	-1
3	4	3	C	3	5
4	6	4	D	4	7
5	-1	5	E	5	-1
6	-1	6	F	6	-1
7	-1	7	G	7	-1

Advantages :-

1. It is most efficient way to represent binary tree in memory.
2. Memory wastage is less as compared to sequential representation.

Disadvantages :-

1. Wasted memory space in -1 as shown above.
2. Given a node, it is difficult to determine its parent node.
3. It's implementation algorithm is more difficult.

Date
Page

Difference between Graph and Tree

	Graph	Tree
1.	More than one path is allowed.	Only one path between two vertices.
2.	It does not have a root node.	It has exactly one root node.
3.	Graph can have loops.	No loops are permitted.
4.	It is more complex.	It is less complex.
5.	It has two Traversal Techniques.	It has Three Traversal Techniques.
6.	Here number of edges are not defined.	Number of edges are N-1 where N is number of nodes
7.	It can be directed or indirected.	Trees are always directed.
8.	For finding shortest path in networking graph is used.	For game trees, decision trees, the tree is iesed.
9.	Represents Data in hierarchial manner.	Represents data similar to a network.

Date
Page

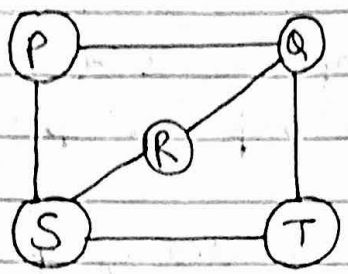
Graphs

Types of Graphs :-

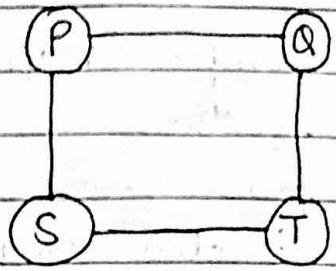
1. Simple Graph - A graph is said to be simple graph if there are no ~~longer~~ loop or parallel edges present.
2. Directed graph - In a directed graph every edge of the graph is an ordered pair of vertices connected by the edge.
3. Undirected graph - In this, every edge is an unordered pair of vertices connected by the edge.
4. Complete graph - A graph is said to be complete if there is edge between each vertices.
5. Weighted graph - A graph is said to be a weighted graph if its edges are assigned some value.
6. Isomorphic graph - Two graphs are said to be isomorphic if
   i) They have the same number of vertex.
   ii) They have same number of edges.
   iii) They have an equal number of vertex with a given degree.
7. Tree Graph - A connected graph which does not create cycle is said to be tree graph. In simple words, a connected graph is a tree if it does not create cycle or loop. All trees are graph but all graphs are not tree.
8. Sub Graph - A graph is said to be a sub graph if and only if it contains the vertex set V and Edges set E from another graph G. In other

Date
Page

words we can say that a graph $G_1' = G_1'(V', E')$ is called a sub graph of $G_1 = G_1(V, E)$ if the vertices and edges of G_1' are contained in the vertices and edges of G_1.



a) Graph G_1



b) Sub graph G_1' of G_1

Terminology of Graphs

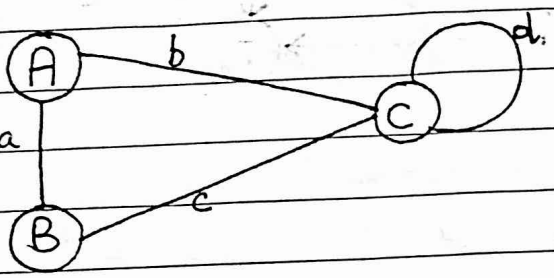
A graph G_1 may be defined as a finite set V of vertices and a set E of edges -

- A graph is a non-linear data structure with cycles.
- A graph $G_1 = (V, E)$ is finite non-empty set of objects.
- V called set of vertices or nodes.
- E called set of all edges or arcs.

1. Adjacent ~~Edge~~ Vertex - Two vertex are called adjacent if there is an edge between them. The degree of a vertex in an undirected graph is the number of edges associated with it.
2. Paths - A path in a graph is a sequence of vertices $V_1, V_2, V_3, \ldots, V_n$ such that $\ldots$ are in E for $i = 1, 2, \ldots (n-1)$. The length of such a path is the number of edges on the path, which is equal

to (n-1). A simple path is a path such that all vertices are distinct, except that the first and last could be the same.

3. Loops – An edge joining only one vertex is called as loop or if it has identical end points.



4. Simple Path –
   - A path is said to be simple path if all the vertices on the path except the first line and last are distinct.
   - If first and last vertices are same, the path will be a cycle.

5. Degree – In the undirected graph, number of edges can be connected to the vertex to form a graph. Connected number of edges to the vertex is the degree of the vertex. If G is a graph with m edges, then

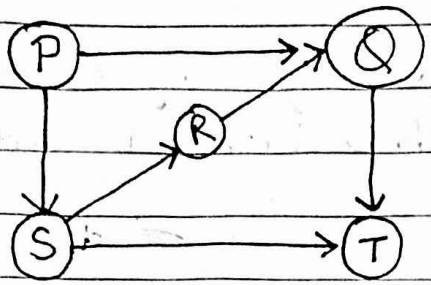
$$\sum \deg(v) = 2m$$

An edge (u, v) is counted twice in the summation above; once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges.

In Degree – This term is used in directed graph, where direction is also defined with edge. Total

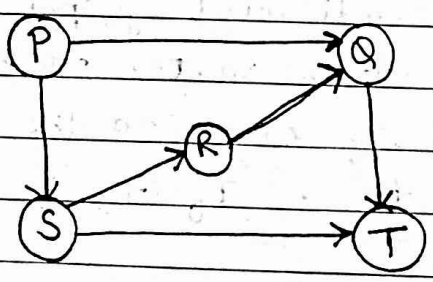
Date
Page

number of incoming edges to the vertex is known as 'In degree' of the vertex.



P → 0
Q → 2
R → 1
S → 1
T → 2

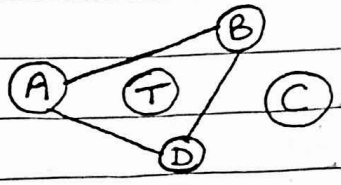
Out-Degree - This term is also used for the directed graph. Total number of outgoing edges from the vertex is known as 'Out Degree' of the vertex.



P → 2
Q → 1
R → 1
S → 2
T → 0

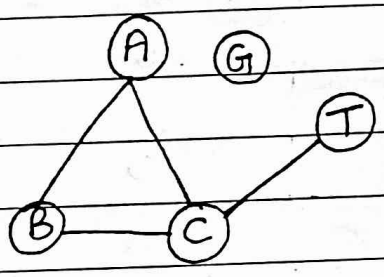
6. Isolated - Vertex - A vertex having no incident edge is called an isolated vertex. Or, we can

say that isolated vertices are vertices with zero degree.



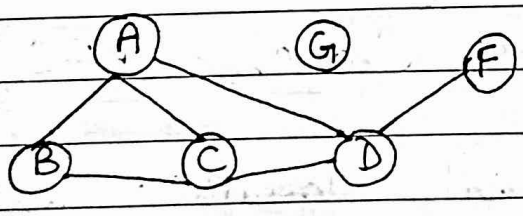
Here, isolated vertices are T and C

7. Pendent Vertex – A vertex of degree one is called pendent or end vertex.



Here Pendent vertex is T

8. Incident edge – The edge (Vi, Vj) is said to be incident if there is any edge (Vi, Vj).

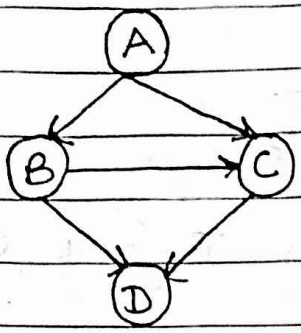


The edges incident on vertex A are (A,B), (A,C) and (A,D).

9. Source and Sink
- A vertex V is said to be source if it has a positive out degree but 0 in-degree.
- A vertex V is said to be sink if it has 0 out-

Date
Page

degree. eg.



Vertex	Indegree	Out-Degree
A	0	2
B	1	2
C	2	1
D	2	0

Here Vertex A is source and Vertex D is sink.

Searching Techniques

(1) Breadth First Search (BFS)

The BFS traversal algorithm visits each node in a graph as follows :-

i) Firstly we examine the starting node V, then we examine all the neighbours of V, then we examine all the neighbours of the neighbours of V and so on.

ii) We need to keep track of the neighbours of a node and we need to guarantee that no node is processed more than once.

(iii) This can be done by using a queue to hold nodes that are waiting to be processed.

iv) A queue is similar to a line of people, that operates on a FIFO Principle.

v) During the execution of algorithm each node n of Graph G will be in one of the three states called as the status of n as follows:-

a) Status = 1 (ready state): the initial state of node n

b) Status = 2 (waiting state): the node n is in the queue waiting to be processed.

c) Status = 3 (processed state): the node has been processed.

Algorithm:-

1) Initialize all the nodes to ready state. (Status = 1).
2) Put the starting node in queue and change its state to waiting state (Status = 2).
3) Repeat step 4 and 5 until queue is empty.
4) Remove the front node n of queue. Process n and change its status to processed state (Status = 3).
5) Add all the neighbours of n that are in ready state (Status = 1) to the rear of queue and change their status to waiting state (Status = 2).
6) Exit.

(2) Depth First Search (DFS)

The DFS traversal algorithm visits each node in a graph as follows:-

Date
Page

i) First we examine the starting node V.
ii) Then we examine each node along a paths that begins at V.
iii) After coming to the end of the path, we backtrack on this path until we can continue along another path and so on.
iv) Eventually this method will visit every node in a connected graph.
v) The actual traverse is done using stack. First the starting node is pushed onto the stack. Then the following process repeats:
   1) Pop a node off the stack.
   2) Traverse or print this current node.
   3) Push all the nodes adjacent to the current node onto the stack.
vi) This process terminates when stack is empty.

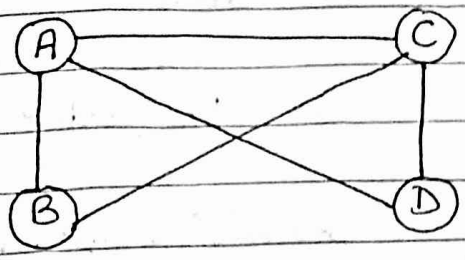
Algorithm —

1) Initialize all the nodes to the ready state (Status = 1).
2) Push the starting node A onto stack and change its status to waiting state (Status = 2).
3) Repeat step 4 and 5 until stack is empty.
4) Pop the top node N of stack. Process N and change its status to the processed state (Status = 3).
5) Push onto stack all the neighbours of N that are still in the ready state (Status = 1) and change their status to the waiting state (Status = 2).
6) EXIT.

26/10/22

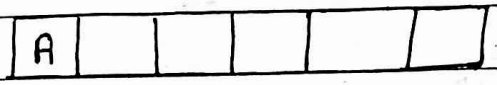
Date
Page

Here is an example of BFS



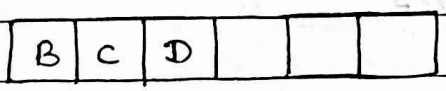
Suppose, we consider node (A) as a root node. ∴ Traversing would be started from node ➡ (A).

Now we insert the node (A) in queue as shown.



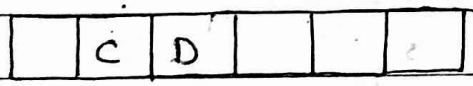
Processed = NULL

* Now we process the node (A) and insert the adjacent nodes of node (A) in queue. After processing, node (A) will be marked as processed.



Processed = A

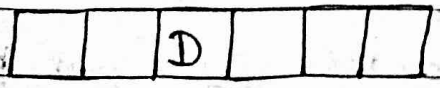
* Now process the node (B) and remove it from queue and insert adjacents of node (B) in queue.



Processed = A, B

Note that we have not to insert the processed nodes again in the queue.

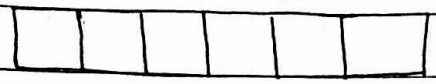
* Now process the node (C) and remove it from queue and add Adjacents of node (C) in queue.



Processed = A, B, C

Date
Page

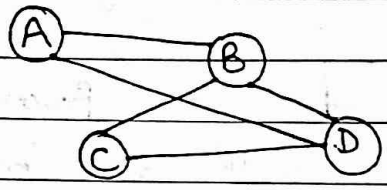
* Now process the node (D) and remove it from queue and add its adjacents in the Queue.



Processed = A, B, C, D

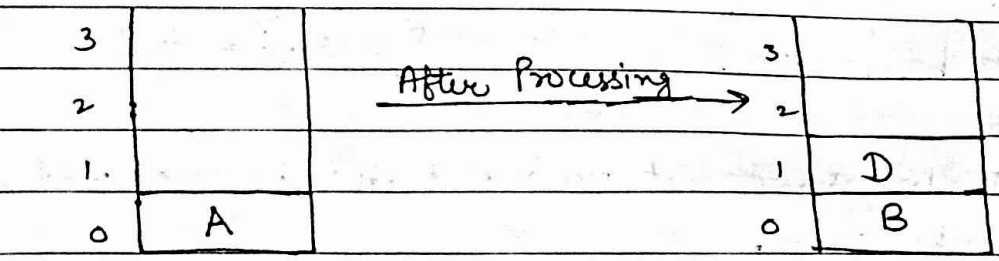
We see that our queue becomes empty. So the final answer of traversal is (A), (B), (C), (D).

Here is an example of DFS



Suppose we consider node (A) as a root node. For DFS Traversal, we use stack.

* Push the node (A) onto the stack. Process the node (A) and pop it from stack and ~~ad~~ push the adjacents of a onto stack.



Processed = A

* Process the top element of stack i.e. (D) and after processing pop it from stack and ~~add~~ push adjacents

of (D) onto the stack.

3	
2	
1	C
0	B

Processed = A, D

* Now, Process the top element of stack i.e (C) and after processing pop it from stack and push adjacents of (C) in stack.

3	
2	
1	
0	B

Processed = A, D, C

* Now, process the top element of stack i.e (B) and after processing pop it from stack and push its adjacents onto stack.

3	
2	
1	
0	

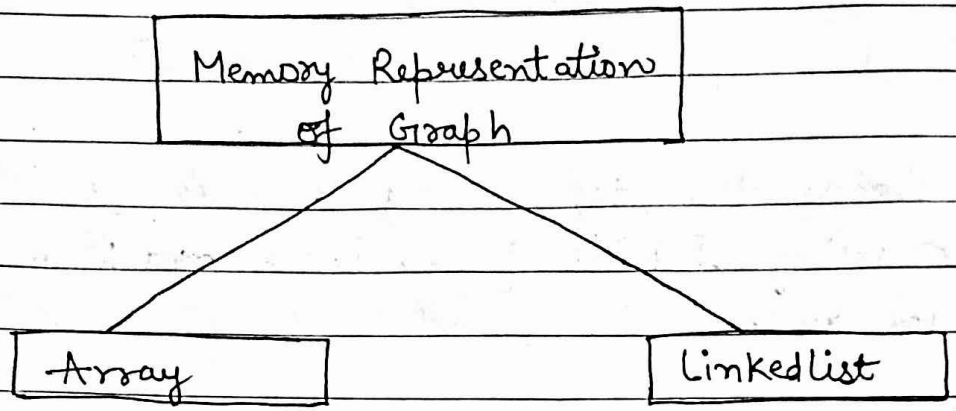
Processed = A, D, C, B

We see that our stack becomes empty. So the final result of DFS Traversal is (A), (D) (C), (B)

Memory Representation Of Graphs

Date
Page

Graphs can be represented in memory by two ways
1) Using Array
2) Using Linkedlist



(i) Representation of Graph in memory using Array :-

Adjacency Matrix :- Suppose G is a simple directed graph with m nodes and suppose the nodes of G have been ordered and are called $V_1, V_2, \ldots V_m$. Then the adjacency Matrix $A = (a_{ij})$ of graph G is the $m \times m$ matrix defined as follows :-

$$a_{ij} = \begin{cases} 1 & \text{if } V_i \text{ is adjacent to } V_j \\ 0 & \text{otherwise} \end{cases}$$

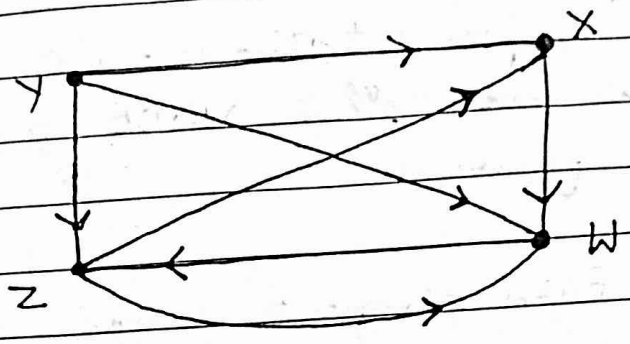
Such matrix A, which contains entries of only 0 and 1 is called bit matrix.

(1) Suppose G is an undirected graph. Then the adjacency matrix A of G will be a symmetric matrix i.e one in which $a_{ij} = a_{ji}$ for every i and j. This follows from the fact that each undirected edge $[u, v]$ corresponds to the two directed edges

(u, v) and (v, u).

(2) The above matrix representation of a graph may be extended to multigraphs. If G is a multigraph then the adjacency Matrix of G is the m x m matrix A = (aij) defined by setting aij equal to the number of edges from Vi to Vj.

Example:-



Suppose the nodes are stored in memory in a linear array DATA as follows:

DATA: X, Y, Z, W

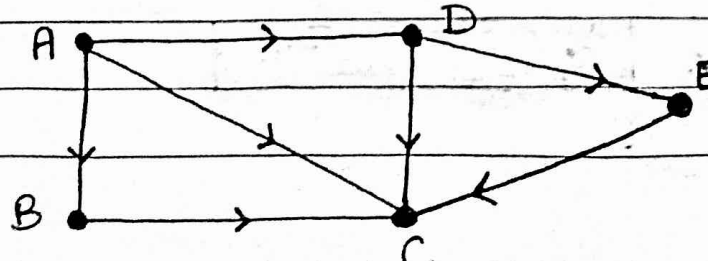
Then we assume that the ordering of the nodes in G is as follows: $V_1 = X$, $V_2 = Y$, $V_3 = Z$, $V_4 = W$.

The adjacency Matrix A of G is as follows:

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

Note that the number of 1's in A is equal to the number of edges in G.

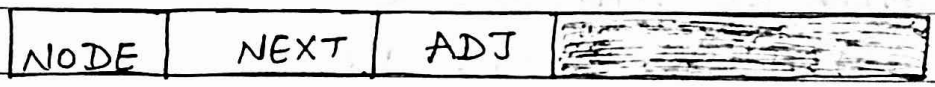
(ii) Representation of Graph using linkedlist :-



Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

The linked representation will contain two lists (or files) a node list NODE and an edge list EDGE, as follows.

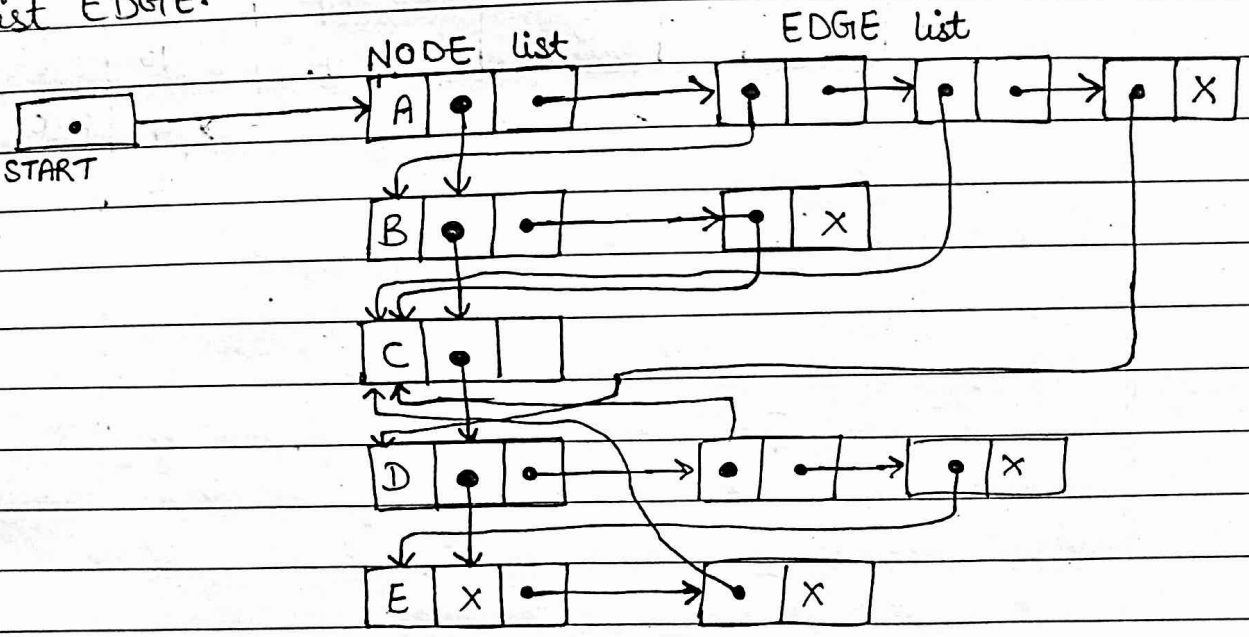
a) Node List → Each element in the list NODE will correspond to a node in G, and it will be a record of form:



Here NODE will be the name or key value of node. NEXT will be a pointer to next node in the list NODE and ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE. The shaded area indicates that there may be other information in record such as indegree or outdegree or status of node.

b) Edge List → Each element in the list EDGE will correspond to an edge of G will be a record of the form:

The field DEST will point to the location in the list NODE of the destination or terminal node of edge. The field LINK will link together the edges with the same initial node, ie nodes in the same adjacency list. The shaded area indicates that there may be other information in record corresponding to the edge, such as a field EDGE containing the labelled data of the edge when G is a labelled graph. We also need a pointer variable AVAILE for the list of available space in the list EDGE.



Schematic diagram of linked representation of G in memory.

AVAILE

Date _____ Page _____ 2

START

4

5

AVAILN

	NODE	NEXT	ADJ
1		3	
2	C	9	0
3		8	
4	A	7	3
5		1	
6	E	0	11
7	B	2	6
8		10	
9	D	6	1
10		0	

	DEST	LINK
1	2 (C)	7
2		5
3	7 (B)	10
4	9 (D)	0
5		8
6	2 (C)	0
7	6 (E)	0
8		9
9		12
10	2 (C)	4
11	2 (C)	0
12		0

i.

Date
Page

Algorithm Complexity

It is very rare to have a single algorithm for solving a problem. How efficient a particular algorithm is for solving a problem if of no concern if the amount of data to be processed is very large. We know that when a program implementing an algorithm is executed, it uses resources of the computing system such as CPU, memory, etc. The study of algorithm is necessary to determine the amount of time and storage space an algorithm may require for execution.
The complexity of an algorithm is a function $f(n)$ that gives the running time and memory space required by an algorithm for processing the input data of size n.

Space Complexity

The analysis of algorithm based on how much memory an algorithm needs to solve a particular problem is called the space complexity of an algorithm. It is expressed as a function $f(n)$ that describes the amount of memory an algorithm takes for processing n input data elements. The space required by the algorithm when implemented not only includes instruction space but also includes data space. The study of space complexity of an algorithm is of major concern when programs implementing algorithms are made to run on a multi-user system when storage space is shared among users. Although space complexity is important but inexpensive memory has reduced the significance of space complexity. Thus our main area of concern is the

time complexity that focuses on amount of time elapsed when processing data.

Time Complexity

The analysis of algorithm based on time of computation is called time complexity of an algorithm. The main objective of time complexity is to compare the performance of different algorithms in solving the same problem. As we are more interested in time complexity so we have to decide how to measure it. One possible approach is to implement the algorithm used for solving a particular problem using a programming language and run them. But this method has a number of pitfails.

1. Time is wasted in implementing each of the algorithm as finally only one of these will be used.
2. The machine on which the program run might influence its running time as time complexity in such a situation depends on a number of factors as follows:-
   - RAM capacity of computer.
   - Operating system used in computer.
   - Quality of code generated by the compiler.

A better method is to employ mathematical model that analyze algorithm independently of specific implementations, computers or data. For Ex!- In searching algos, the key operations are the number of comparisons made and in sorting algorithms the key operations are swapping and comparisons. In sorting algorithms, time complexity will be small if the number of elements are to be processed are small compared to when large number of elements are to be processed.

Date
Page

in former case the total number of comparisons made will be smaller than the later case. Thus, the time complexity of an algorithm is a function $f(n)$ that describes the efficiency of an algorithm in terms of size n of input data. As time complexity is dependent on the key operations rather than all operations so we usually express complexity function $f(n)$ as proportional to input size (n) rather than exact function. Now let us consider a few segments of algorithm performing some key operations where we have to measure the time complexity.

1. A segment of algorithm that involves simple loop.

```
for ( i=1 ; i< 501 ; i++)
{     statement(s);
}
```

In above segment, we need to know that how many times the body of loop is executed. It is 500 times. As Time complexity is directly proportional to the number of iterations so for loop factor n, it can be generally expressed as

$$f(n) \propto n$$

2. Now, let us consider another loop

```
for (i=1 ; i< 501 ; i=i+2)
{     statements;
}
```

In above case, the number of iteration is half the loop factor as step size is 2. So the time complexity of this loop is proportional to half the loop factor.

$$f(n) \propto n/2$$

Date
Page

3. Now let us consider an example in which loop counter is multiplied or divide each iteration.

for (i=1; i<=500; i=i*2)	for (i=500; i>=1; i=i/2)
{ Statement (s);	{ statements;
}	}

In above two segments of algorithm, we can not say that the body of loop iterates 500 times in each case as the number of times we iterate is governed by loop counter variable I which does not change in a linear fashion. In first case, it changes by a multiple of 2 and in the second case it changes by dividing it by 2.

For multiply	For Divide
$2^{Iteration} <= 500$	$500/2^{Iteration} >= 1$

Expressing it in term of logarithms, we can say that there are $\log_2 500$ i.e 9 iterations. So, for loop factor n, time complexity is proportional to $\log_2 n$.

i.e $f(n) \propto \log_2 n$

4. Now let us consider an example of nested loop where we need to determine how many times each loop iterates. In nested loop, the total no. of iterations is the product of number of iterations in the inner loop and the outer loop.

a)
```
for (i=1 ; i<=20; i++).
{ for (j=1; j<=20; j++)
    {   Statements;
    }
```

Date
Page

```
}
```

If both loop iterates n times then the time complexity which is proportional to no. of iterations i.e.

$$f(n) \propto n^2$$

b)
```
for( i=1 ; i<=20 ; i++)
{
    for( j=1 ; j<=20 ; j= j*2)
    {
        Statement(s);
    }
}
```

Here the outer loop iterates 20 times and inner loop iterates $\log_2 20$ times as loop factor J changes by multiple of 2

$$\therefore f(n) \propto n \log n$$

c)
```
for( i=1 ; i<=20 ; i++)
{
    for(j=1 ; j<=i ; j++)
    {
        Statement(s);
    }
}
```

As in previous case, the outer loop iterates 20 times but in this case the number of times inner loop iterates is dependent on the value of outer loop counter variable I.

if I=1, the body of loop iterates only one.
if I=2, " " " " " twice.
:
and so on.

∴ The number of times the body of inner loop iterates is $1+2+3+\ldots\ldots+20 = 210$ which is total no. of iterations performed by the nested loop.

Date
Page

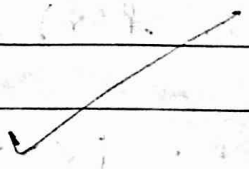
Since, the number of times the ~~body of inner loop~~ outer loop iterates is 20 so the number of times inner loop iterates is 210/20 = 21/2 which is same as the number of iterations (20) plus 1 divided by 2 i.e (20+1)/2.

As we know that in a nested loops,

Total number of iterations = Number of iterations of Outer loop
X Number of iterations of inner loop.

In general, if loop factor is 'n' then time Complexity which is proportional to the number of iteration is

$f(n) \propto n \cdot (n+1)/2$.



Difference between

Array	Linklist
1. An array is a grouping of data elements of equivalent datatype.	A linkedlist is a group of entities called a node. It has two segments [Data + Address].
2. The memory is assigned at compile time.	The memory is assigned at run time.
3. It stores data in a contiguous memory location.	It stores data randomly in the memory.
4. The elements are not dependent on each other.	The elements are dependent on each other.
5. It is faster to access the element in an array.	In this, process of accessing the elements takes time.
6. Memory utilization is ineffective.	Memory utilization is effective.
7. In case of array, memory size is fixed and is not possible to change it during run time.	In linklist, the placement of element is allocated during the run time.

Date
Page

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For Ex:- In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But when the input array is in reverse condition, the algorithm takes the maximum time to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations :-

1) Big O notation
2) Omega notation
3) Theta notation

1. **Big O notation** ⇒ Big O notation represents upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.

$$O(g(n)) = \{ f(n) : \text{there exists positive constant } C \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

graph

example

The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exist a positive constant C such that it lies between 0 and $cg(n)$,

for sufficiently large n.

2. Omega (Ω) notation $\Rightarrow$ Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as function $f(n)$ belongs to the set $\Omega(g(n))$ if there exist a positive constant c such that it lies above $cg(n)$, for sufficiently large n.

3. Theta (Θ) notation $\Rightarrow$ Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

$\Theta(g(n)) = \{ f(n)$: there exist +ve constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist +ve constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

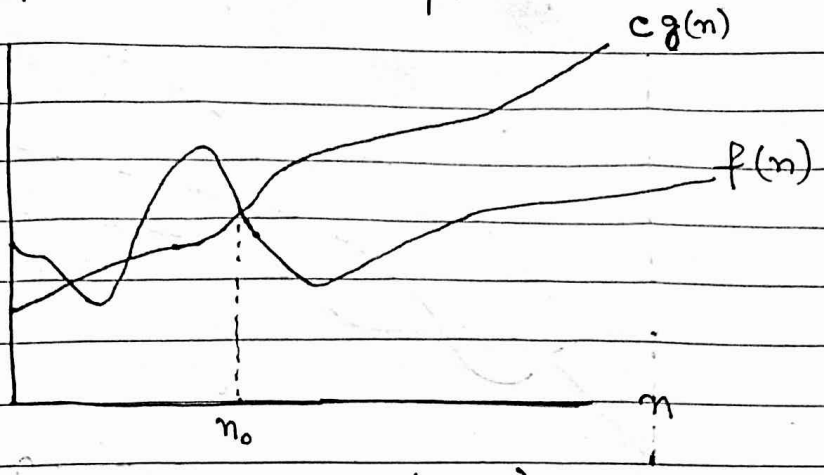
Date _____
Page _____

	Best	Average	Worst
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log_2(n))$	$O(\log_2(n))$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
Merge Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

31/10/22

Date
Page

* Big O Graph with example



$f(n) = O(g(n))$

Ex:- $f(n) = n^2 + n + 1$

$= O(n^2)$

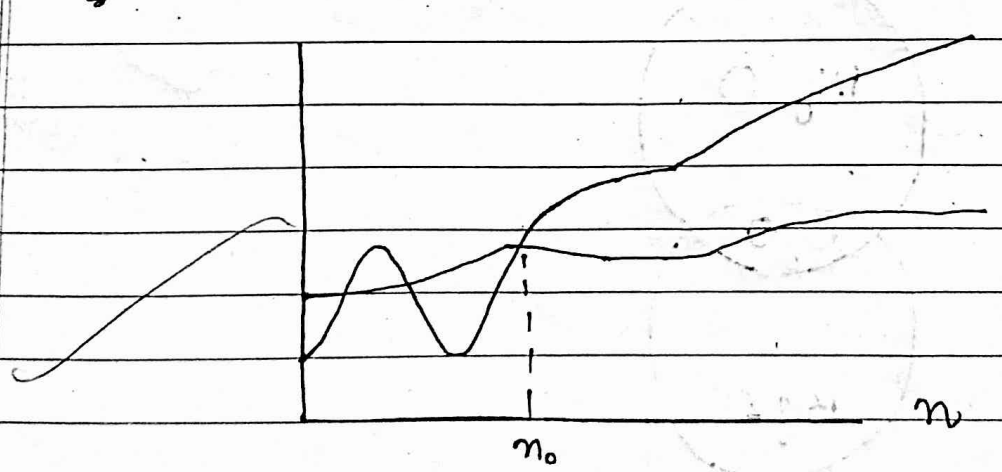
Let $1 \leq n \leq n^2$

$f(n) \leq n^2 + 2n^2 + n^2$

$f(n) \leq 4n^2$ which is of form $c \cdot g(x)$

$\therefore O(n^2)$

* Ω Graph with example



$f(n) = \Omega(g(n))$

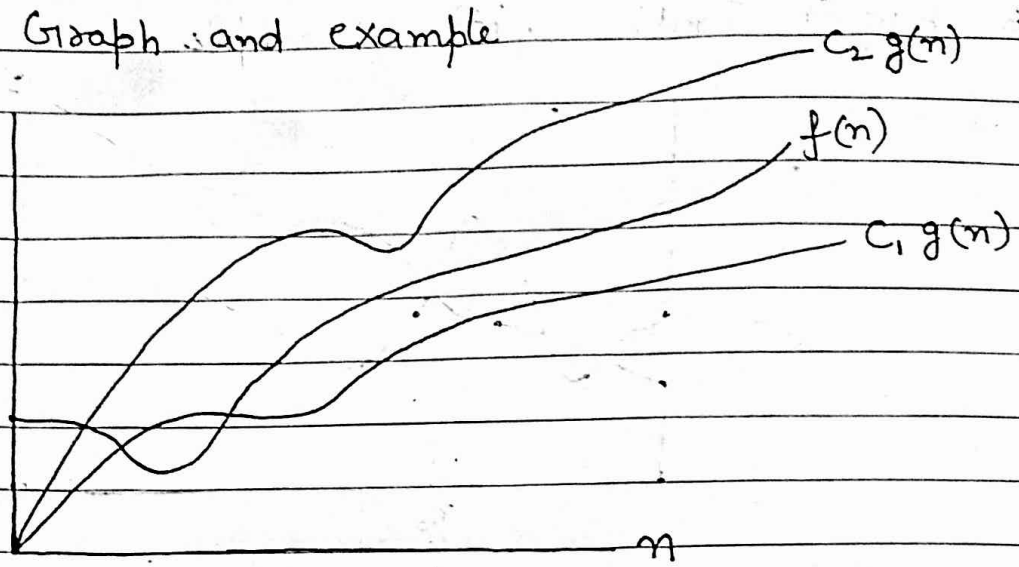
Ex $f(n) = n^2 + 2n + 1$

$f(n) \geq 1 \cdot n^2$

$\therefore \Omega(n^2)$

Date
Page

* Theta (Θ) Graph and example

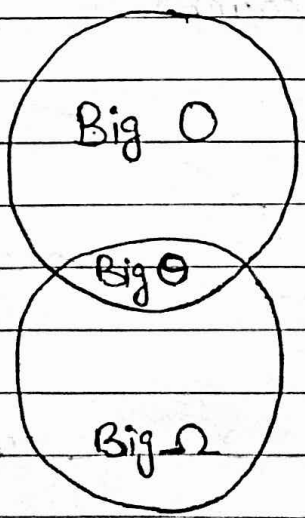


$$f(n) = \Theta(g(n))$$

$$1 \cdot n^2 \leq f(n) \leq 4 \cdot n^2$$

Lower Bound Ω

Upper Bound Big O



02/11/22