D.N.R.COLLEGE(AUTONOMOUS)::BHIMAVARAM

M.Sc COMPUTER SCEINCE DEPARTMENT

E-CONTENT

I - M.Sc(CS)

DISCRETE MATHEMATICAL STRUCTURES

Presented by
A.NAGA RAJU

DISCRETE MATHEMATICAL STRUCTURES (MSCS 101)

Theory	**: 4 Periods**	**Mid Marks**	**: 25**
Lab Hrs	**: 0 Periods**	**Ext. Marks**	**: 75**
Exam	**: 3 Hrs.**	**Credits**	**: 4**

Unit I
Sets, relations and functions: Operations on sets, relations and functions, binary relations, partial ordering relations, equivalence relations, principles of mathematical induction. Permutations and combinations; recurrence relation and generating functions.

Unit II
Algebraic structures and morphisms: Algebraic structures with one binary operation - semigroups, monoids and groups, congruence relation and quotient structures. Free and cyclic monoids and groups, permutation groups, substructures, normal subgroups.
Algebraic structures with two binary operations, Lattices, Principle of Duality, Distributive and Complemented Lattices, Boolean Lattices and Boolean Algebras, Uniqueness of Finite Boolean Algebras, Boolean Functions and Boolean Expressions, Propositional Calculus.

Unit III
Mathematical logic: Syntax, semantics of Propositional and predicate calculus, valid, satisfiable and unsatisfiable formulas, encoding and examining the validity of some logical arguments.
Proof techniques: forward proof, proof by contradiction, contrapositive proofs, proof of necessity and sufficiency.

Unit IV
Graph Theory: Graphs and digraphs, Eulerian cycle and Hamiltonian cycle, adjacency and incidence matrices, vertex colouring, planarity.
Trees: Introduction of trees, Applications of trees, Tree traversal, Spanning trees, minimum spanning trees

Text Book

1. J. P. Tremblay and R. P. Manohar, Discrete Mathematical Structures with Applications to Computer Science, Tata McGraw-Hill, 2001.

Reference Books:

1. Kenneth H. Rosen, Discrete Mathematics and its Applications, Tata McGraw-Hill.
2. C. L. Liu, Elements of Discrete Mathematics, 2nd Edition, Tata McGraw-Hill, 2000.

UNIT- I
Sets, Relations and Functions

Sets

Definition:

A set is an unordered collection of different elements. A set can be written explicitly by listing its elements using set bracket. If the order of the elements is changed or any element of a set is repeated, it does not make any changes in the set.

Some Example of Sets

A set of all positive integers

A set of all the planets in the solar system

A set of all the states in India

A set of all the lowercase letters of the alphabet

Representation of a Set

Sets can be represented in two ways −

Roster or Tabular Form

Set Builder Notation

Roster or Tabular Form

The set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas.

Example 1 − Set of vowels in English alphabet, A={ a, e ,I ,o ,u}

Example 2 − Set of odd numbers less than 10, B={1,3,5,7,9}

Set Builder Notation

The set is defined by specifying a property that elements of the set have in common. The set is described as A={x:p(x)}

Example 1 − The set {a, e ,I ,o ,u}

is written as −

A={x:x is a vowel in English alphabet}

Example 2 − The set {1,3,5,7,9}

is written as −

$B=\{x:1\leq x<10 \text{ and } (x\%2)\neq 0\}$

If an element x is a member of any set S, it is denoted by $x \in S$

and if an element y is not a member of set S, it is denoted by $y \notin S$.

Example − If $S=\{1,1.2,1.7,2\}, 1\in S$

but $1.5\notin S$

Some Important Sets

N − the set of all natural numbers = {1,2,3,4, ...}

Z − the set of all integers = {.....,−3,−2,−1,0,1,2,3,.....}

Z+ − the set of all positive integers

Q − the set of all rational numbers

R − the set of all real numbers

W − the set of all whole numbers

Cardinality of a Set

Cardinality of a set S, denoted by |S|

, is the number of elements of the set. The number is also referred as the cardinal number. If a set has an infinite number of elements, its cardinality is ∞.

Example − $|\{1,4,3,5\}|=4, |\{1,2,3,4,5,\dots\}|=\infty$

If there are two sets X and Y, $|X|=|Y|$ denotes two sets X and Y having same cardinality. It occurs when the number of elements in X is exactly equal to the number of elements in Y. In this case, there exists a objective function 'f' from X to Y. $|X|\le|Y|$ denotes that set X's cardinality is less than or equal to set Y's cardinality. It occurs when number of elements in X is less than or equal to that of Y. Here, there exists an injective function 'f' from X to Y. $|X|<|Y|$ denotes that set X's cardinality is less than set Y's cardinality. It occurs when number of elements in X is less than that of Y. Here, the function 'f' from X to Y is injective function but not objective.

If $|X|\le|Y|$ and $|X|\ge|Y|$ then $|X|=|Y|$.

The sets X and Y are commonly referred as equivalent sets.

Types of Sets

Sets can be classified into many types. Some of which are finite, infinite, subset, universal, proper, singleton set, etc.

Finite Set

A set which contains a definite number of elements is called a finite set.

Example − $S=\{x \mid x \in N \text{ and } 70>x>50\}$

Infinite Set

A set which contains infinite number of elements is called an infinite set.

Example − $S=\{x \mid x \in N \text{ and } x>10\}$

Subset

A set X is a subset of set Y (Written as $X\subseteq Y$) if every element of X is an element of set Y.

Example 1 − Let, $X=\{1,2,3,4,5,6\}$ and $Y=\{1,2\}$.

Here set Y is a subset of set X as all the elements of set Y is in set X. Hence, we can write $Y\subseteq X$.

Example 2 − Let, $X=\{1,2,3\}$ and $Y=\{1,2,3\}$.

Here set Y is a subset (Not a proper subset) of set X as all the elements of set Y is in set X. Hence, we can write $Y\subseteq X$

.

Proper Subset

The term “proper subset” can be defined as “subset of but not equal to”. A Set X is a proper subset of set Y (Written as $X \subset Y$) if every element of X is an element of set Y and $|X|<|Y|$.

Example − Let, $X=\{1,2,3,4,5,6\}$ and $Y=\{1,2\}$.

Here set $Y \subset X$ since all elements in Y are contained in X too and X has at least one element is more than set Y.

Universal Set

It is a collection of all elements in a particular context or application. All the sets in that context or application are essentially subsets of this universal set. Universal sets are represented as U.

Example − We may define U as the set of all animals on earth. In this case, set of all mammals is a subset of U , set of all fishes is a subset of U , set of all insects is a subset of U , and so on.

Empty Set or Null Set

An empty set contains no elements. It is denoted by ∅ . As the number of elements in an empty set is finite, empty set is a finite set. The cardinality of empty set or null set is zero.

Example − $S=\{x \mid x \in N \text{ and } 7<x<8\}=\emptyset$

Singleton Set or Unit Set

Singleton set or unit set contains only one element. A singleton set is denoted by {s}.

Example − $S=\{x \mid x \in N, 7<x<9\}= \{8\}$

Equal Set

If two sets contain the same elements they are said to be equal.

Example − If $A=\{1,2,6\}$ and $B=\{6,1,2\}$, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.

Equivalent Set

If the cardinalities of two sets are same, they are called equivalent sets.

Example − If $A=\{1,2,6\}$ and $B=\{16,17,22\}$, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. $|A|=|B|=3$

Overlapping Set

Two sets that have at least one common element are called overlapping sets.

In case of overlapping sets −

$n(A\cup B)=n(A)+n(B)-n(A\cap B)$

$n(A\cup B)=n(A-B)+n(B-A)+n(A\cap B)$

$n(A)=n(A-B)+n(A\cap B)$

$n(B)=n(B-A)+n(A\cap B)$

Example − Let, $A=\{1,2,6\}$ and $B=\{6,12,42\}$.

There is a common element ‘6’, hence these sets are overlapping sets.

Disjoint Set

Two sets A and B are called disjoint sets if they do not have even one element in common. Therefore, disjoint sets have the following properties −

$n(A \cap B)=\emptyset$

$n(A \cup B)=n(A)+n(B)$

Example − Let, $A=\{1,2,6\}$ and $B=\{7,9,14\}$, there is not a single common element, hence these sets are overlapping sets.

Venn Diagrams

Venn diagram, invented in 1880 by John Venn, is a schematic diagram that shows all possible logical relations between different mathematical sets.

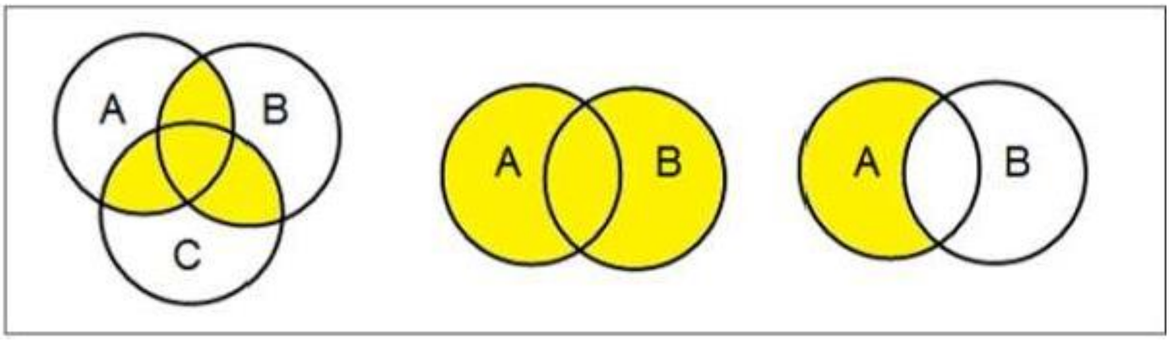
Examples

Venn Diagram

ADVERTISEMENT

Set Operations

Set Operations include Set Union, Set Intersection, Set Difference, Complement of Set, and Cartesian Product.



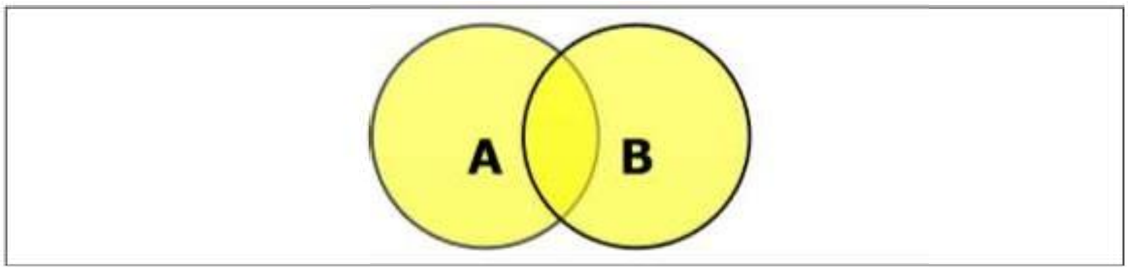
Set Union

Set Intersection

The intersection of sets A and B (denoted by $A \cap B$) is the set of elements which are in both A and B. Hence, $A \cap B=\{x| x \in A \text{ AND } x \in B\}$.

Example − If $A=\{11,12,13\}$ and $B=\{13,14,15\}$, then $A \cap B=\{13\}$

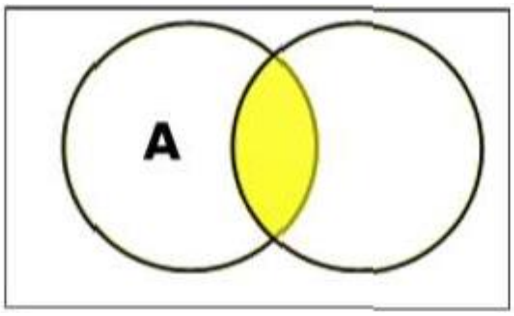
.



Set Intersection

The intersection of sets A and B (denoted by A∩B) is the set of elements which are in both A and B. Hence, A∩B={x| x ∈A AND x ∈B}.

Example − If A={11,12,13} and B={13,14,15} , then A∩B={13}.



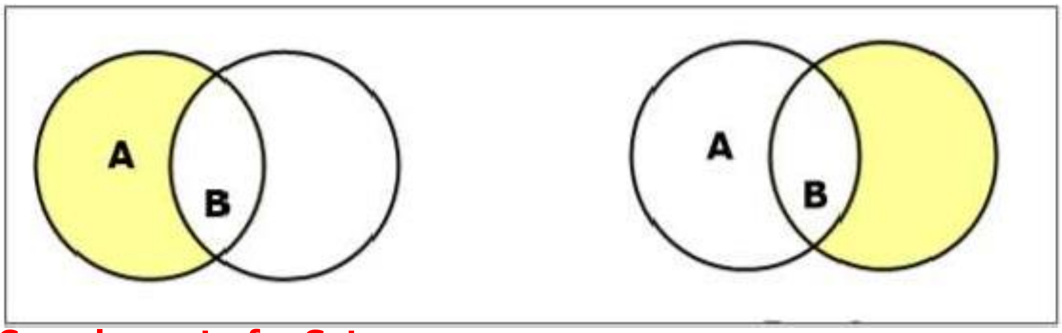
Set Difference/ Relative Complement

The set difference of sets A and B (denoted by A–B) is the set of elements which are only in A but not in B. Hence, A−B={x| x ∈A AND x ∉B}

Example − If A={10,11,12,13} and B={13,14,15} , then (A−B)={10,11,12} and (B−A)={14,15}.

Here, we can see (A−B)≠(B−A)

Set Difference



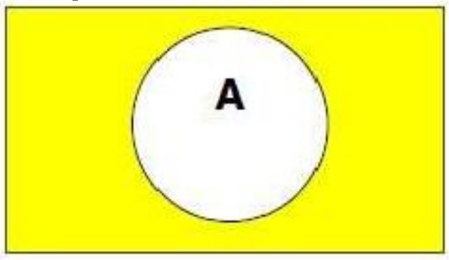
Complement of a Set

The complement of a set A (denoted by A′) is the set of elements which are not in set A. Hence, A′={x|x∉A}.

More specifically, A′=(U−A) where U is a universal set which contains all objects.

Example − If A={x| x belongs to set of odd integers} then A′={y| y does not belong to set of odd integers}

Complement Set



Cartesian Product / Cross Product

The Cartesian product of n number of sets A1,A2,...An denoted as A1×A2⋯×An can be defined as all possible ordered pairs (x1,x2,...x n) where x1∈A1,x2∈A2,...x n ∈An

Example − If we take two sets A={a ,b} and B={1,2},

The Cartesian product of A and B is written as − A×B={(a,1),(a,2),(b,1),(b,2)}

The Cartesian product of B and A is written as − B×A={(1,a),(1,b),(2,a),(2,b)}

Power Set

Power set of a set S is the set of all subsets of S including the empty set. The cardinality of a power set of a set S of cardinality n is 2n. Power set is denoted as P(S).

Example :-For a set S={a, b, c, d} let us calculate the subsets − Subsets with 0 elements − {∅} (the empty set) Subsets with 1 element − {a},{b},{c},{d}

Subsets with 2 elements − {a ,b},{a ,c},{a ,d},{b, c},{b, d},{c ,d}

Subsets with 3 elements − {a, b ,c},{a, b, d},{a, c, d},{b, c, d}

Subsets with 4 elements − {a, b ,c, d}

Hence,

P(S)={{∅},{a},{b},{c},{d},{a,b},{a,c},{a,d},{b,c},{b,d},{c,d},{a,b,c},{a,b,d},{a,c,d},{b,c,d},{a,b,c, d}}

|P(S)|=24=16

Note − The power set of an empty set is also an empty set. |P({∅})|=20=1

Partitioning of a Set

Partition of a set, say S, is a collection of n disjoint subsets, say P1,P2,…P n that satisfies the following three conditions -Pi does not contain the empty set.

[Pi≠{∅} for all 0<i≤ n]

The union of the subsets must equal the entire original set.[P1∪P2∪⋯∪P n=S]

The intersection of any two distinct sets is empty.

[Pa ∩ Pb={∅}, for a ≠b where n≥a,b≥0]

Example

Let S={a, b ,c, d, e ,f, g, h}

One probable partitioning is {a},{b, c, d},{e, f, g, h}

Another probable partitioning is {a ,b},{c, d},{e, ,g ,h}

Bell Numbers

Bell numbers give the count of the number of ways to partition a set. They are denoted by B n where n is the cardinality of the set.

Example –

Let S={1,2,3} , n=|S|=3 The alternate partitions are −

1. ∅,{1,2,3}
2. {1},{2,3}
3. {1,2},{3}
4. {1,3},{2}
5. {1},{2},{3}

Hence B3=5

RELATIONS

Introduction

The elements of a set may be related to one another. For example, in the set of natural **numbers there is the ‘less than’ relation between the elements.** The elements of one set may also be related to the elements another set.

Binary Relation

A binary relation between two sets A and B is a rule R which decides, for any elements, whether a is in relation R to b. If so, we write a R b. If a is not in relation R to b, then we shall write a /R b.

We can also consider a R b as the ordered pair (a, b) in which case we can define a binaryrelation from A to B as a subset of A X B. This subset is denoted by the relation R.

In general, any set of ordered pairs defines a binary relation.

For example, the relation of father to his child is F = {(a, b) / a is the father of b} In this relationF, the first member is the name of the father and the second is the name of the child.

The definition of relation permits any set of ordered pairs to define a relation.

For example, the set S given by
S = {(1, 2), (3, a), (b, a) ,(b, Joe)}

Definition

The **domain** D of a binary relation S is the set of all first elements of the ordered pairs in therelation.(i.e) D(S)= {a / $ b for which **(a, b) € S}**
The **range** R of a binary relation S is the set of all second elements of theordered **pairs in the relation.(i.e) R(S) = {b / $ a for which (a, b) €**
S}

For example
For the relation S = {(1, 2), (3, a), (b, a) ,(b,
Joe)}D(S) = {1, 3, b, b} and
R(S) = {2, a, a, Joe}
Let X and Y be any two sets. A subset of the Cartesian product X * Y defines a relation, say C.For any such relation C, we have D(C) Í X and R(C) Í Y, and the relation C is said to from X

to Y. If Y = X, then C is said to be a relation form X to X. In such case, c is called a relation in X. Thus any relation in X is a subset of X * X . The set X * X is called a universal relation in X, while the empty set which is also a subset of X * X is called a void relation in X.

For example

Let L denote the relation "less than or equal to" and D denote the relation "divides" where x D y means " x divides y". Both L and D are defined on the

set {1, 2, 3, 4}

L = {(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4),(,4)4)}

D = {(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4)}

L Ç D = {(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4,4)} = D

Properties of Binary Relations:

Definition: A binary relation R in a set X is **reflexive if, for every x Є X, x Rx, that is (x, x) Є** R, or R is reflexive in X **ó (x) (x Є X ® x R x).**

For example

The relation £ is reflexive in the set of real numbers.

The set inclusion is reflexive in the family of all subsets of a universal set.

The relation equality of set is also reflexive.

The relation is parallel in the set lines in a plane.

The relation of similarity in the set of triangles in a plane is reflexive.

Definition: A relation R in a set X is **symmetric** if for every x and y in X, whenever **x R y, theny R x.(i.e) R is symmetric in X ó (x) (y) (x Є X ^ y Є X ^ x R y ®** y R x}

Example:

The relation equality of set is symmetric.

The relation of similarity in the set of triangles in a plane is symmetric.

The relation of being a sister is not symmetric in the set of all people.

However, in the set females it is symmetric.

Definition: A relation R in a set X is whenever x R y and y R z , then x R z. (i.e)

transitive if, for every x, y, and z are in X, R istransitive in X ó **(x) (y) (z) (x Є X^ y Є X^ z Є**

The relations **<, £, >, t** and = are transitive in the set of real numbers

The relations Í, Ì, Ê, É and equality are also transitive in the family of sets.
The relation of similarity in the set of triangles in a plane is transitive.

Definition: A relation R in a set X is **irreflexive if, for every x Є X ,**

(x, x)ÏX.For example

The relation < is irreflexive in the set of all real numbers.
The relation proper inclusion is irreflexive in the set of all nonempty subsets of auniversal set.
Let X = {1, 2, 3} and S = {(1, 1), (1, 2), (3, 2), (2, 3), (3, 3)} is neither irreflexive norreflexive.

Definition:A relation R in a set x is **anti symmetric** if , for every x and y in X, whenever x R y and y R x, then x = y.
Symbolically,(x) (y) (x Є X ˄ y Є X ˄ x R y ˄ y R x ® x = y)

for example

The relations £ , ³ and = are anti symmetric

The relation Í is anti symmetric in set of subsets.

The relation **"divides" is anti symmetric in set of real numbers.**

Consider the relation "is a son of" on the male children in a family.Evidently the relation is not symmetric, transitive and reflexive.

The relation " is a divisor of " is reflexive and transitive but not symmetric on the setof natural numbers.

Consider the set H of all human beings. Let r be a relation " is married to " R is symmetric.

Let I be the set of integers. R on I is defined as a R b if a – b is an even number.R is an reflexive, symmetric and transitive.

Equivalence Relation:
Definition: A relation R in a set A is called an **equivalence** relation if
a R a for every i.e. R is reflexive

a R b => b R a for every a, b Є A i.e. R is symmetric

a R b and b R c => a R c for every **a, b, c Є A, i.e. R is transitive.**

For example
The relation equality of numbers on set of real numbers.
The relation being parallel on a set of lines in a plane.

Problem1: Let us consider the set T of triangles in a plane. Let us define a relation R in T as R= {(a,b)/ (a, b Є T and a is similar to b}
We have to show that relation R is an equivalence relation

Solution :

A triangle a is similar to itself. a R If the triangle a is similar to the triangle b, then triangle b is similar to the triangle a thena R b => b R a If a is similar to b and b is similar to c, then a is similar to c (i.e) a R b and b R c => a Rc. Hence R is an equivalence relation.

Problem 2: Let x = {1, 2, 3, … 7} and R = {(x, y) / x – y is divisible by 3} Show that R is anequivalence relation.

Solution: For any a Є X, a- a is divisible by 3, Hence a R a, R is reflexive

For any a, b Є X, if a – b is divisible by 3, then b – a is also divisible by3, R is symmetric.

For any a, b, c Є, if a R b and b R c, then a – b is divisible by 3 and b–c is divisible by 3. So that (a – b) + (b – c) is also divisible by3, hence a – c is also divisible by 3. Thus R is transitive.

Hence R is equivalence.

Problem3 Let Z be the set of all integers. Let m be a fixed integer. Two integers a and b are said to be congruent modulo m if and only if m divides a-b, in which case we write a º b (mod m). This relation is called the relation of congruence modulo m and we can show that isan equivalence relation.

Solution :

a - a=0 and m divides a – a (i.e) **a R a, (a, a) Є R, R is** reflexive .

a R b = m divides a-b

m
divides b -
a b º a
(mod m) b
R a
that is R is symmetric.

a R b and b R c => a ºb (mod m) and bº c (mod m)
○m divides a – b and m divides b-c
○a – b = km and b – **c = lm for some k ,l Є z**
○(a – b) + (b – c) = km + lm
○a – c = (k +l) m
○aº c (mod m)
○a R c
○R is transitive

Hence the congruence relation is an equivalence relation.

Equivalence Classes:

Let R be an equivalence relation on a set A. For any **a ЄA, the equivalence class** generated by a **is the set of all elements b Є A such a R b and is denoted [a].** It is also called the R – **equivalence class and denoted by a Є A.** i.e., **[a] = {b Є A / b R a}**

Let Z be the set of integer and R be the relation **called "congruence modulo3"** defined by R = {(x, y)/ xÎ Z Ù yÎZ Ù (x-y) is divisible by 3}

Then the equivalence classes are

[0] = {… -6, -3, 0, 3, 6, **…}**

[1] = {…, -5, -2, 1, 4, 7, …}

[2] = {…, -4, **-1, 2, 5, 8, …}**

Composition of binary relations:

Definition: Let R be a relation from X to Y and S be a relation from Y to Z. Then the relation R o S is given by R o S = {(x, z) / xÎX Ù z Î Z Ù y Î Y such that (x, y) Î R Ù (y, z) Î S)} iscalled the composite relation of R and S.

The operation of obtaining R o S is called the **composition of relations**.

Example: Let R = {(1, 2), (3, 4), (2, 2)} and
S = {(4, 2), (2, 5), (3, 1),(1,3)}
Then R o S = {(1, 5), (3, 2), (2, 5)} and S o R = {(4, 2), (3, 2), (1, 4)}
It is to be noted that R o S ≠ S o R.
Also Ro(S o T) = (R o S) o T = R o S o T

Note: We write R o R as R2; R o R o R as R3 and so on.

Definition

Let R be a relation from X to Y, a relation Ř from Y to X is called the **converse** of R, **where the ordered pairs of Ř are obtained by interchanging the numbers in each of the ordered** pairs of R. This means for x Î X and y Î **Y, that x R y ó y Ř x.**

Then the relation Ř is given by R = {(x, y) / (y, x) Î R} is called the converse

of R Example:

Let R = {(1, 2),(3, 4),(2, 2)}

Then Ř = {(2, 1),(4, 3),(2, 2)}

Note: If R is an equivalence rel**ation, then Ř is also an equivalence relation.**

Definition: Let X be any finite set and R be a relation in X. The relation
R+ = R U R2 U R3…in X. is called the *transitive closure* of R in X

Example: Let R = {(a, b), (b, c),
(c, a)}.

Now R2 = R o R = {(a, c), (b, a),
(c, b)}

R3 = R2 o R = {(a, a), (b, b), (c, c)}
R4 = R3 o R = {(a, b), (b, c), (c, a)} = R
R5= R3o R2 = R2 and so on.

Thus, R+ = R U R2 U R3 U R4 U...
= R U R2 U R3.
={(a, b),(b, c),(c, a),(a, c),(b, a),(c ,b),(a, b),(b, b),(c, c)}

We see that R+ is a transitive relation containing R. In fact, it is the smallest transitive relation containing R.

Partial Ordering Relations:

Definition
A binary relation R in a set P is called ***partial order relation*** or *partial ordering* in Piff R is reflexive, anti symmetric, and transitive.

A partial order relation is denoted by the symbol £., If £ is a partial ordering on P,then the ordered pair (P, £) is called a ***partially ordered set*** or a ***poset***.

Let R be the set of real numbers. The relation "less than or equal to " or

- ○ , is a partial ordering on R.
- Let X be a set and r(X) be its power set. The relation subset, Í on X is partial ordering.
- **Let Sn be the set of divisors of n. The relation D means "divides" on Sn ,**is partialordering on Sn .

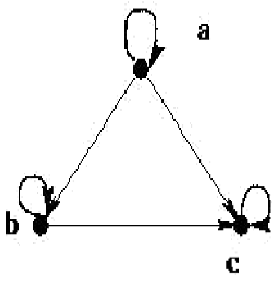
In a partially ordered set (P, £) , an element y Î P is said to cover an element x Î P if x <y and if there does not exist any element z Î P such that x £ z and z £ y;
that is, y covers x Û (x < y Ù (x £ z £ y Þ x = z Ú z = y))
A partial order relation £ on a set P can be represented by means of a diagram known as

a Hasse diagram or partial order set diagram of (P, £). In such a diagram, each element is represented by a small circle or a dot. The circle for x Î P is drawn below the circle for y Î P if x < y, and a line is drawn between x and y if y covers x.

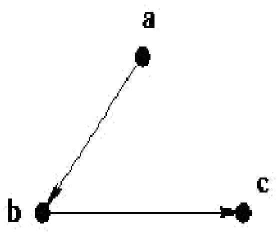
If x < y but y does not cover x, then x and y are not connected directly by a single line.However,they are connected through one or more elements of P.

Hasse Diagram:

A Hasse diagram is a digraph for a poset which does not have loops and arcs implied by thetransitivity.
Example 10: For the relation {< a, a >, < a, b >, < a, c >, < b, b >, < b, c >, < c, c >} on set {a,b,c}, the Hasse diagram has the arcs {< a, b >, < b, c >} as shown below.

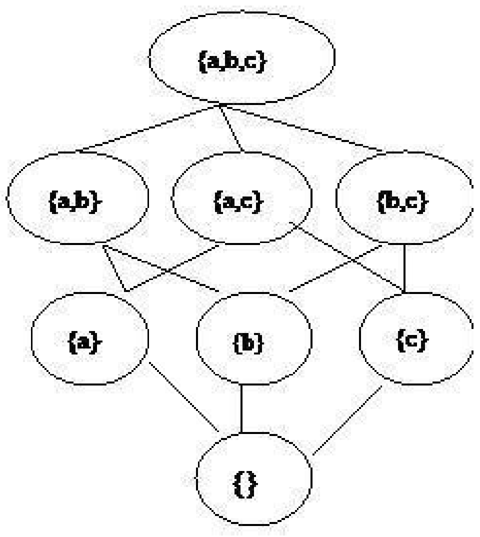


Digraph for Partial Order



Hasse Diagram

Ex: Let A be a given finite set and r(A) its power set. Let Í be the subset relation on theelements of r(A). Draw Hasse diagram of (r(A), Í) for A = {a, b, c}



Functions

Introduction

A function is a special type of relation. It may be considered as a relation in which each element of the domain belongs to only one ordered pair in the relation. Thus a function from A **to B is a subset of A X B having the property that for each a ЄA, there is one and only oneb Є B** such that (a, b) Î G.

Definition

Let A and B be any two sets. A relation f from A to B is called a function **if for every a Є Athere is a unique b Є B such that (a, b) Є f .**

Note that the definition of function requires that a relation must satisfy two additionalconditions in order to qualify as a function.

The first condition is that every a Є A must be related to some b Є B, (i.e) the domain off must be A and not merely subset of A. The second requirement of uniqueness can be expressed **as (a, b) Є f ^ (b, c) Є f => b = c**

Intuitively, a function from a set A to a set B is a rule which assigns to every element of A, a uniqueelement of B. **If a ЄA, then the unique element of B assigned to a under f is denoted by f** (a).The usual notation for a function f from A to B is f: A® B defi**ned by a ® f (a) where a Є A,** f(a) is called the image of **a** under f and **a** is called pre image of f(a).

Let X = Y = **R** and f(x) = x2 + 2. Df = R and Rf Í R.

Let X be the set of all statements in logic and let Y = {True,

False}. A mapping f: X®Y is a function.

A program written in high level language is mapped into a machine language by a compiler. Similarly, the output from a compiler is a function of its input.

Let X = Y = **R** and f(x) = x2 is a function from X ® Y,and g(x2) = x is not a functionfrom X ® Y.

A mapping f: A ® B is called **one-to-one** (injective or 1 –1) if distinct elements of Aare mapped into distinct elements of B. (i.e) f is one-to-one if
a1 = a2 => f (a1) = f(a2) or equivalently f(a1) ¹ f(a2) => a1 ¹ a2
For example, f: N ® N given by f(x) = x is 1-1 where N is the set of a natural numbers.
A mapping f: A® B is called **onto (surjective) if for every b Є B there is an a Є A such** that f (a) = B. i.e. if every element of B has a pre-image in A. Otherwise it is called **into**.

For example, f: Z®Z given by f(x) = x + 1 is an onto mapping.A mapping is both 1-1 and onto is called bijective
.

For example f: R®R given by f(x) = X + 1 is bijective.

Definition: A mapping f: R® b is called a **constant mapping** if, for all aÎA, f (a) =b, a fixed element.

For example f: Z®Z given by f(x) = 0, for all x ÎZ is a constant mapping.

Definition

A mapping f: A®A is called the **identity mapping** of A if f (a) = a, for all aÎA. Usually it is denoted by IA or simply I.

Composition of functions:

If f: A®B and g: B®C are two functions, then the composition of functions f and g, denotedby g o f, is the function is given by g o f : A®C and is given by

g o f = {(a, c) / a € A ʌ c € C ʌ $bÎ B ': f(a)= b ʌ g(b) =c}

and (g of)(a) = ((f(a))

Example 1: Consider the sets A = {1, 2, 3},B={a, b} and C = {x, y}. Let f: A® B be defined by f (1) = a ; f(2) = b and f(3)=b andLet g: B® C be defined by g(a) = x and g(b) = y

(i.e) f = {(1, a), (2, b), (3, b)} and g = {(a, x), (b,y)}. Then g o f: A®C is defined by

(g of) (1) = g (f(1)) = g(a) = x

(g o f) (2) = g (f(2)) = g(b) = y

(g o f) (3) = g (f(3)) = g(b) = y

i.e., g o f = {(1, x), (2, y),(3, y)}

If f: A® A and g: A®A, where A= {1, 2, 3}, are given by

f = {(1, 2), (2, 3), (3, 1)} and g = {(1, 3), (2, 2), (3, 1)} Then

g of = {(1, 2), (2, 1), (3, 3)}, fog= {(1, 1), (2, 3), (3, 2)}

f of = {(1, 3), (2, 1), (3, 2)} and gog= {(1, 1), (2, 2), (3, 3)}

Example 2: Let f(x) = x+2, g(x) = x – 2 and h(x) = 3x for x Î R, where R is the set of realnumbers.

Then f o f = {(x, x+4)/xÎ R} fo g = {(x, x)/ x Î X} g

o f = {(x, x)/ xÎ X}

g o g = {(x, x-4)/x Î X}

h o g = {(x,3x-6)/ x Î X} ho f = {(x, 3x+6)/ x Î X}

Inverse functions:

Let f: A® B be a one-to-one and onto mapping. Then, its inverse, denoted by f -1 is given by f -1 = {(b, a) / (a, b) Î f} Clearly f-1: B® A is one-to-one and onto.

Also we observe that f o f -1 = IB and f -1o f = IA.
If f -1 exists then f is called invertible.
For example:Let f: R ®R be defined by f(x) = x + 2Then f -1: R® R is defined by f -1(x) = x - 2

Theorem: Let f: X ®Y and g: Y ® Z be two one to one and onto functions. Then gof is also oneto one and onto function.

Proof

Let f:X ® Y g : Y ® Z be two one to one and onto functions. Let x1, x2 Î X

g o f (x1) = g o f(x2),

g (f(x1)) = g(f(x2)),

g(x1) = g(x2) since [f is 1-1]

x1 = x2 since [g is 1-1}so that gof is 1-1.

By the definition of composition, gof : X ® Z is a function.
We have to prove that every element of z Î Z an image element for some x Î X under gof.
Since g is onto $ y ÎY ': g(y) = z and f is onto from X to Y,
$ x ÎX ': f(x) = y.
Now, gof (x) = g (f (x))
= g(y) [since f(x) = y]= z [since g(y) = z]
which shows that gof is onto.

Theorem (g o f) -1 = f -1 o g -1

(i.e) the inverse of a composite function can be expressed in terms of the composition of the inverses in the reverse order.

Proof.

f: A ® B is one to one and onto.g: B ® C is one to one and onto.
gof: A ® C is also one to one and onto. Þ(gof) -1: C ® A is one to one and onto.

Let a Î A, then there exists an element b Î b such that f (a) = b Þ a = f-1(b). Now b Î B Þ there exists an element c Î C such that g (b) = c Þ b = g - 1(c). Then (gof)(a) = g[f(a)] = g(b) = c Þ a = (gof) -1(c). **(1)**
(f -1 o g-1) (c) = f -1(g -1 (c)) = f -1(b) = a Þ a = (f -1 o g **-1)(c)**

….(2)

Combining (1) and (2), we have (gof)

-1 = f -1 o g -1

Theorem: If f: A ® B is an invertible mapping ,
thenf o f -1 = I B and f-1 o f = IA
Proof: f is invertible, then f -1 is defined by f(a) = b ó f-1(b)
=a where a Î A and bÎ B .
Now we have to prove that f of -1 = IB
. Let bÎ B and f -1(b) = a, a Î Athen fof-
1(b) = f(f-1(b))
= f(a) = b
therefore f o f -1 (b) = b " b Î B => f o f -1 = IBNow f -1
o f(a) = f -1 (f(a)) = f -1 (b) = a therefore f -1 o f(a) = a "
a Î A => f -1 o f = IA. Hence the theorem.

Recursive Functions:

The term "recursive function" is often used informally to describe any function that is defined with recursion. There are several formal counterparts to this informal definition, many of which only differ in trivial respects.

Kleene (1952) defines a "partial recursive function" of nonnegative integers to be any function f that is defined by a noncontradictory system of equations whose left and right sides are composed from
(1) function symbols (for example, f, g, h, etc.), (2) variables for nonnegative integers (for example, x, y, z, etc.), (3) the constant 0, and (4) the successor function $S(x) = x + 1$.
defines $f(x, y)$to be the function $x\,y$that computes the product of $x$and y.

Note that the equations might not uniquely determine the value of $f$for every possible input, and in that sense the definition is "partial." If the system of equations determines the value of f for every input, then the definition is said to be "total." When the term "recursive function" is used alone, it is usually implicit that "total recursive function" is intended. Note that some authors use the term "general recursive function to mean partial recursive function, although others use it to mean "total recursive function."

The set of functions that can be defined recursively in this manner is known to be equivalent tothe set of functions computed by Turing machines and by the lambda calculus.

Lattice and its Properties:

Introduction:

A lattice is partially ordered set (L, £) in which every pair of elements a, b ÎL has a greatest lower bound and a least upper bound.
The glb of a subset, {a, b} Í L will be denoted by a * b and the lub by a Å b.
.
Usually, for any pair a, b Î L, GLB {a, b} = a * b, is called the **meet** or **product** and LUB{a,

b } = a Å b , is called the **join** or **sum** of a and b

Example1 Consider a non-empty set S and let P(S) be its power set. The relation Í“contained in” is a partial ordering on P(S). For any two subsets A, BÎ P(S)
GLB {A, B} and LUB {A, B} are evidently A Ç B and A È B respectively.

Example2 Let I+ be the set of positive integers, and D denote the relation of **“division”** inI+ such that for any a, b Î I+ , a D b iff a divides b. Then (I+, D) is a lattice in which the join of a and b is given by the least common multiple(LCM) of a and b, that is, a Å b = LCM of a and b, and the meet of a and b, that is , a * b is the greatest common divisor (GCD) of a and b.

A lattice can be conveniently represented by a diagram.

For example, let Sn be the set of all divisors of n, where n is a positive integer. Let D denote the *relation “division” such that for any a, b Î Sn, a D b iff a divides b.*
Then (Sn, D) is a lattice with a * b = gcd(a, b) and a Å b = lcm(a, b).
Take n=6. Then S6 = {1, 2, 3, 6}. It can be represented by a diagram in Fig(1).Take n=8. Then S8 = {1, 2, 4, 8}

Two lattices can have the same diagram. For example if S = {1, 2, 3} then (p(s), Í) and (S6,D)
under gof. Since g is onto $ y ÎY ': g(y) = z and f is onto from X to Y,
$ x ÎX ': f(x) = y.
Now, gof (x) = g (f (x))
= g(y) [since f(x) = y]
have the same diagram viz. fig(1), but the nodes are differently labeled .
We observe that for any partial ordering relation £ on a set S the converse relation ³ is also partial ordering relation on S. If (S, £) is a lattice With meet a * b and join a Å b , then (S, ³) is the also a lattice with meet a Å b and join a * b i.e., the GLB and LUB get interchanged . Thus we have the principle of duality of lattice as follows.

Any statement about lattices involving the operations ^ and V and the relations £ and ³ remains true if ^, V, ³ and £ are replaced by V, ^, £ and ³ respectively.
The operation ^ and V are called duals of each other as are the relations £ and ³.. Also, the lattice (L, £) and (L, ³) are called the duals of each other.

Properties of lattices:

Let (L, £) be a lattice with the binary operations * and Å then for any a, b, c Î L,

For any a Î L, a £ a, a £ LUB {a, b} =

> a £ a * (a Å b). On the other hand,GLB {a, a Å b} £ a i.e., (a Å b) Å a,
hence a * (a Å b) = a

Theorem 1

Let (L, £) be a lattice with the binary operations * and Å denote the operations of meet and joinrespectively For any a, b Î L,
a £ b ó a * b = a ó a Å b = b

Proof

Suppose that a £ b. we know that a £ a, a £ GLB {a, b}, i.e., a £ a * b.But from the definition of a * b, we get a * b £ a.
Hence a £ b => a * b = a ………………………… (1)
Now we assume that a * b = a; but is possible only if a £ b,
that is a * b = a => a £ b ………………………… (2)
From (1) and (2), we get a £ b ó a * b = a.
Suppose a * b = a.
then b Å (a * b) = b Å a = **a Å b** ……………………. (3)
but b Å (a * b) = b (by iv)…………………….. (4)
Hence a Å b = b, from (3) => (4)
Suppose aÅ b = b, i.e., LUB {a, b} = b, this is possible only if a£ b, thus(3) => (1)
(1) => (2) => (3) => (1). Hence these are equivalent.

Let us assume a * b = a.
Now (a * b) Å b = a Å b
We know that by absorption law , (a * b) Å b = b
so that a Å b = b, therefore a * b = a Þ a Å b = b (5)
similarly, we can prove a Å b = b Þ a * b = a (6)
From (5) and (6), we get
a * b = a Û a Å b = bHence the theorem.

Theorem2 For any a, b, c Î L, where (L, £) is a lattice. b £ c => { a * b £ a * c and
{ a Å b £ a Å c = a Å c which => (a Å b) £ (a Å c)

note:These properties are known as **isotonicity.**

Proof *Suppose b £ c. we have proved that b £ a ó b * c = b …………………… (1)*

Now consider
(a * b) * (a * c) = (a * a) * (b * c) (by Idempotent)
= a * (b * c)
= a * b (by (1))
Thus (a * b) * (a * c) = a * b which => (a * b) £ (a * c)
Similarly (a Å b) Å (a Å c) = (a Å a) Å (b Å c)
= a Å (b Å c)

Principles of

Mathematical Induction:

1. **Base Case**:
   - **Statement**: The first part of mathematical induction involves proving that the statement P(n)P(n)P(n) holds for the smallest value of nnn in the set of natural numbers (often starting from n=0n = 0n=0 or n=1n = 1n=1).
   - **Example**: If you want to prove a statement P(n)P(n)P(n) for all natural numbers n≥1n \geq 1n≥1, you first prove P(1)P(1)P(1).
2. **Inductive Step**:

Statement: The second part is the inductive step, which establishes that if the statement P(k)P(k)P(k) holds for some arbitrary but fixed natural number kkk, then P(k+1)P(k+1)P(k+1) must also hold.

Example: Assuming P(k)P(k)P(k) is true, you then prove P(k+1)P(k+1)P(k+1).

Steps in Mathematical Induction:

1. **Base Case Verification**:

Verify P(n)P(n)P(n) for the smallest value of nnn. Typically, this is n=0n = 0n=0 or n=1n = 1n=1, depending on the problem statement.

2. **Inductive Hypothesis**:

Assume P(k)P(k)P(k) is true for an arbitrary but fixed kkk. This assumption is crucial for the inductive step.

3. **Inductive Step**:

Prove that assuming P(k)P(k)P(k) leads to P(k+1)P(k+1)P(k+1) being true. This step often involves manipulating P(k)P(k)P(k) to derive P(k+1)P(k+1)P(k+1) using the assumption that P(k)P(k)P(k) is true.

Conclusion:

By combining the base case and the inductive step, you conclude that P(n)P(n)P(n) is true for all nnn in the set of natural numbers for which the statement is intended.

Example Application:

Example: Prove by induction that 1+2+3+…+n=n(n+1)21 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}1+2+3+…+n=2n(n+1) for all n∈Nn \in \mathbb{N}n∈N.

- **Base Case**: For n=1n = 1n=1, 1=1·(1+1)2=11 = \frac{1 \cdot (1+1)}{2} = 11=21·(1+1)=1, which is true.
- **Inductive Step**: Assume 1+2+…+k=k(k+1)21 + 2 + \ldots + k = \frac{k(k+1)}{2}1+2+…+k=2k(k+1) for some k≥1k \geq 1k≥1. Now prove

1+2+…+k+
k+1)=(k+1)(
k+2)21 + 2
+ \ldots + k
+ (k+1) =
\frac{(k+1)(
k+2)}{2}1+
2+…+k+(k
1)=2(k+1)(
+2):

1+2+…+k+
k+1)=k(k+1
)2+(k+1)1 +
2 + \ldots +
k + (k+1) =
\frac{k(k+1
}{2} +
(k+1)1+2+
…+k+(k+1
=2k(k+1)
+(k+1)

=k(k+1)+2(
k+1)2=
\frac{k(k+1
+
2(k+1)}{2}
=2k(k+1)+2
(k+1)

=(k+1)(k+2
2=
\frac{(k+1)(
k+2)}{2}=2
(k+1)(k+2)

- **Con
clusion**: By
mathematic
al induction
1+2+…+n=
n(n+1)21 +
2 + \ldots +
n =
\frac{n(n+1
}{2}1+2+…
+n=2n(n+1
holds for all
n∈Nn \in

\mathbb{N}n∈N.

Permutations & Combinations:

Permutations and combinations are fundamental concepts in combinatorics, a branch of mathematics that deals with counting, arranging, and selecting objects. Here’s a breakdown of each concept:

Permutations:

Definition: A permutation is an arrangement of objects in a specific order.

Formula: The number of permutations of \(n \) distinct objects taken \(r \) at a time is given by:

\[P(n, r) = \frac{n!}{(n-r)!} \]

where \(n! \) (read as "n factorial") denotes the factorial of \(n \), which is the product of all positive integers up to \(n \).

Explanation:

- \(P(n, r) \) calculates the number of ways to arrange \(r \) objects chosen from \(n \) distinct objects, considering the order of arrangement.
- \(n! = n \times (n-1) \times \ldots \times 2 \times 1 \) ensures that each object is used exactly once.
- \((n-r)! \) adjusts for the fact that we're arranging only \(r \) objects out of \(n \).

Example: How many ways can 3 students (A, B, C) be arranged in a line?

\[P(3, 3) = \frac{3!}{(3-3)!} = \frac{6}{0!} = 6 \]

So, there are 6 permutations: ABC, ACB, BAC, BCA, CAB, CBA.

Combinations:

Definition: A combination is a selection of objects without regard to the order of selection.

Formula: The number of combinations of \(n \) distinct objects taken \(r \) at a time is given by:\[C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!} \]

Explanation:

- \(C(n, r) \) calculates the number of ways to choose \(r \) objects from \(n \) distinct objects without considering the order.

- \(\binom{n}{r} \) is read as "n choose r".

- \(r! \) (r factorial) corrects for the fact that different arrangements of the same \(r \) objects are counted only once.

Example: How many ways can a committee of 2 students be selected from 3 students (A, B, C)?

\[C(3, 2) = \binom{3}{2} = \frac{3!}{2!(3-2)!} = \frac{6}{2 \times 1} = 3 \]

So, there ar
3
combinatio
s: AB, AC,
BC.

Key
Differences

-
Order:
Permutation
s consider
the order of
arrangemen
, while
combinatio
s do not.

-
**Formula*
*: The
formulas fo
permutation
s and
combinatio
s differ
mainly in
the
denominato
, where
permutation
s include \(
(n-r)! \) to
account for
order,
whereas
combinatio
s divide by
\(r! \) to
correct for
overcountin
g
arrangemen
s.

- **Use
Cases**:
Permutation
s are used
when the

order matters (e.g., arranging people in a line), while combinations are used when the order does not matter (e.g., selecting members for a committee).

Example:

Understanding permutations and combinations is essential for solving problems involving arranging objects, selecting teams, distributing items, and various other counting scenarios in mathematics and beyond.

Recurrence Relation

Generating Functions:

In mathematics, a **generating function** is a formal power series in one indeterminate, whose coefficients encode information about a sequence of numbers a_n that is indexed by the natural numbers. Generating functions were first introduced by Abraham de Moivre in 1730, in order to solve the general linear recurrence problem. One can generalize to formal power series in more than one indeterminate, to encode information about arrays of numbers indexed by several natural numbers.

Generating functions are not functions in the formal sense of a mapping from a domain to a codomain; the name is merely traditional, and they are sometimes more correctly called **generating series**.

Ordinary generating function

The *ordinary generating function* of a sequence a_n is

$$G(a_n;x)=\sum_{n=0}^{\infty}a_nx^n.$$

When the term *generating function* is used without qualification, it is usually taken to mean an ordinary generating function.

If a_n is the probability mass function of a discrete random variable, then its ordinary generating function is called a probability-generating function.

The ordinary generating function can be generalized to arrays with multiple indices. For example, the ordinary generating function of a two-dimensional array $a_{m,n}$ (where n and m are natural numbers) is

$$G(a_{m,n};x,y)=\sum_{m,n=0}^{\infty}a_{m,n}x^my^n.$$

$$G(n^2;x) = \sum_{n=0}^{\infty} n^2 x^n = \frac{x(x+1)}{(1-x)^3}$$ (2)

(3)

(4) for th partition function where $(q$ a q- Pochham r symbo and

Exponential generating function

The *exponential generating function* of a sequence a_n is

$$\mathrm{EG}(a_n;x) = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}.$$

Example:

$$\mathrm{EG}(n^2;x) = \sum_{n=0}^{\infty} \frac{n^2 x^n}{n!} = x(x+1)e^x$$

Function of Sequences:

$$= \sum_{n=0}^{\infty} P(n)\, x^n$$

$$= 1 + x + 2x^2 + 3x^3 + \dots$$

Generating functions giving the first few powers of the nonnegative integers are given inthe following table.

n^p	$f(x)$	series
1	$\frac{x}{1-x}$	$x + x^2 + x^3 + \dots$
n	$\frac{x}{(1-x)^2}$	$x + 2x^2 + 3x^3 + 4x^4 + \dots$
n^2	$\frac{x(x+1)}{(1-x)^3}$	$x + 4x^2 + 9x^3 + 16x^4 + \dots$
n^3	$\frac{x(x^2+4x+1)}{(1-x)^4}$	$x + 8x^2 + 27x^3 + \dots$
n^4	$\frac{x(x+1)(x^2+10x+1)}{(1-x)^5}$	$x + 16x^2 + 81x^3 + \dots$

here are many beautiful generating functions for special functions in number theory. Afew particularly nice examples are

$$f(x) = \frac{1}{(x)_\infty}$$

$$f(x) = \frac{x}{1-x-x^2} \tag{5}$$

$$= \sum_{n=0}^{\infty} F_n x^n \tag{6}$$

$$= x + x^2 + 2x^3 + 3x^4 + \dots \tag{7}$$

for the Fibonacci numbers F_n.

Generating functions are very useful in combinatorial enumeration problems. For example, the subset sum problem, which asks the number of ways $c_{m,s}$ to select out of given m integers M such that their sum equals s, can be solved using generating functions.

Calculating Coefficient of generating function:

By using the following polynomial expansions, we can calculate the coefficient of a generatingfunction.

Polynomial Expansions:

1) $\frac{1-x^{m+1}}{1-x} = 1 + x + x^2 + \dots + x^m$

2) $\frac{1}{1-x} = 1 + x + x^2 + \dots$

3) $(1+x)^n = 1 + C(n,1)x + C(n,2)x^2 + \dots + C(n,r)x^r + \dots + C(n,n)x^n$

4) $(1-x^m)^n = 1 - C(n,1)x^m + C(n,2)x^{2m} + \dots + (-1)^k C(n,k)x^{km} + \dots + (-1)^n C(n,n)x^{nm}$

5) $\frac{1}{(1-x)^n} = 1 + C(1+n-1,1)x + C(2+n-1,2)x^2 + \dots + C(r+n-1,r)x^r + \dots$

6) If h(x)=f(x)g(x), where f(x) $= a_0 + a_1x + a_2x^2 + \ldots +$ and g(x) $= b_0 + b_1x + b_2x^2 + \ldots +$, then

h(x) $= a_0b_0 + (a_1b_0 + a_0b_1)x + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \ldots + (a_rb_0 + a_{r-1}b_1 + a_{r-2}b_2 + \ldots + a_0b_r)x^r + \ldots +$

Recurrence relations:

Introduction :A recurrence relation is a formula that relates for any integer n ≥ 1, the n-th term of a sequence A = {ar}∞r=0 to one or more of the terms a0,a1,….,an-1. Example. If Sn denotes the sum of the first n positive integers, then

9. Sn = n + Sn-1. Similarly if d is a real number, then the *n*th term of an arithmeticprogression with common difference d satisfies the relation

10. an = an -1 + d. Likewise if pn denotes the nth term of a geometric progression withcommon ratio r, then

pn = r pn – 1. We list other examples as:an –
3an-1 + 2an-2 = 0.
an – 3 an-1+ 2 an-2 = n2 + 1.
an – (n - 1) an-1 - (n - 1) an-2 = 0. an – 9
an-1+ 26 an-2 – 24 an-3 = 5n.an – 3(an-
1)2 + 2 an-2 = n.
an = a0 an-1+ a1 an-2+ … + an-1a0.a2n
+ (an-1)2 = -1.

Definition. Suppose n and k are nonnegative integers. A recurrence relation of the form c0(n)an + c1(n)an-1 + …. + ck(n)an-k = f(n) for n ≥ k, where c0(n), c1(n),…., ck(n), and f(n) are functions of n is said to be a **linear recurrence relation**. If c0(n) and ck(n) are not identically zero, then it is said to be a **linear recurrence relation *degree* k**. If c0(n), c1(n),…., ck(n) are constants, then the recurrence relation is known as a **linear relation with constant coefficients**. If f(n) is identically zero, then the recurrence relation is said to be **homogeneous**; otherwise, it is **inhomogeneous.**

Thus, all the examples above are linear recurrence relations except (8), (9), and (10);the relation (8), for instance, is not linear because of the squared term.
The relations in (3), (4) , (5), and (7) are linear with constant coefficients.

Relations (1), (2), and (3) have degree 1; (4), (5), and (6) have degree 2; (7) has degree
3. Relations (3) , (4), and (6) are homogeneous.

There are no general techniques that will enable one to solve all recurrence relations. There are, nevertheless, techniques that will enable us to solve linear recurrence relations with constant coefficients.

SOLVING RECURRENCE RELATIONS BY SUSTITUTION AND GENERATING FUNCTIONS

We shall consider four methods of solving recurrence relations in this and the nexttwo sections:

5. Substitution (also called iteration),
6. Generating functions,
7. Characteristics roots, and
8. Undetermined coefficients.

In the substitution method the recurrence relation for an is used repeatedly to solve for ageneral expression for an in terms of n. We desire that this expression involve no other terms of the sequence except those given by boundary conditions.

The mechanics of this method are best described in terms of examples. We used this method in Example5.3.4. Let us also illustrate the method in the following examples.

Example

Solve the recurrence relation an = a n-1 + f(n) for n ³1 by substitutiona1=

a0 + f(1)

a2 = a1 + f(2) = a0 + f(1) + f(2))

a3 = a2 + f(3)= a0 + f(1) + f(2) + f(3)

an = a0 + f(1) + f(2) +….+f(n) n

= a0 + ∑ f(k)

K = 1

Thus, an is just the sum of the f(k) „s plus a0.

More generally, if c is a constant then we can solve an = c a n-1 + f(n) for n ³1 in the same way:

a1 = c a0 + f(1)

a2 = c a1 + f(2) = c (c a0 + f(1)) + f(2)
= c2 a0 + c f(1) + f(2)

a3= c a2 + f(3) = c(c 2 a0 + c f(1) + f(2)) +
f(3) =c3 a0 + c2 f(1) + c f(2) + f(3)

an = c a n-1 + f(n) = c(c n-1 a0 + c n-2 f(1) +. . . + c n-2 + f(n-1)) +
f(n) =c n a0 + c n-1 f(1) + c n-2 f(2) +. . .+ c f(n-1) + f(n)
Or
an = c n a0 + ∑c n-k f(k)

Solution of Linear Inhomogeneous Recurrence Relations:

The equation + 1 an−1+c 2an −2=( ) an, where n 1and 2 are constant, and ( ) is not identically 0, is called a second-order linear **inhomogeneous** recurrence relation (or difference equation) with constant coefficients. The homogeneous case, which we"ve looked at already, occurs when

( )≡0. The inhomogeneous case occurs more frequently. The homogeneous case is so important largely because it gives us the key to solving the inhomogeneous equation. If you"ve studied linear differential equations with constant coefficients, you"ll see the parallel. We will call the difference obtained by setting the right-hand side equal to 0, the "associated homogeneous equation." We know how to solve this. Say that V is a solution. Now suppose that ( ) is n any particular solution of the inhomogeneous equation. (That is, it solves the equation, but does not necessarily match the initial data.) Then $U= V+$( ) is n a solution to the inhomogeneous equation, which you can see simply by substituting U into the equation. On the other hand, every solution U of the inhomogeneous equation is of the form $U=V+$( ) n where V is a solution of the homogeneous equation, and ( ) is a $g n$ particular solution of the inhomogeneous equation. The proof of this is straightforward. If we have two solutions to the inhomogeneous equation, say $U1$ and U 2, then their difference $U1- U2= V$ is a solution to the homogeneous equation, which you can check by substitution. But then $U1=V+U2$, and we can set $U2=$( ), n since by assumption, U 2 is a particular solution. This leads to the following theorem: **the general solution to the inhomogeneous equation is the general solution to the associated homogeneous equation, plus any particular solution to the inhomogeneous equation.** This gives the following procedure for solving the inhomogeneous equation:

4. Solve the associated homogeneous equation by the method we‟ve learned. This will involve variable (or undetermined) coefficients.

5. Guess a particular solution to the inhomogeneous equation. It is because of the guess that I‟vecalled this a procedure, not an algorithm. For simple right-hand sides , we can say how to compute a particular solution, and in these cases, the procedure merits the name "algorithm."

6. The general solution to the inhomogeneous equation is the sum of the answers from the two steps above.

7. Use the initial data to solve for the undetermined coefficients from step 1.

To solve the equation $- 6$ an1 + 8an −2 = 3an Let‟s suppose that we are also given the initial data 0 = 3, 1 = 3a The associated homogeneous equation is − 6 −1 + 8 −2 = 0, a so the an an characteristic equation is 2 − 6r+ 8 = r 0, which has roots 1 = 2 r and 2 = 4. r Thus, the general solution to the associated homogeneous equation is 12 + 24 n. When the right-hand side is a polynomial, as in this case, there will always be a particular solution that is a polynomial. Usually, a polynomial of the same degree will work, so we‟ll guess in this case that there is a constant C that solves the homogeneous equation. If that is so, then = an −1 an −2 = C, and substituting into the equation gives C − C + C = 3, and we find that C = 1. Now, the general solution to the inhomogeneous equations is 12 + 24 + 1. n Reassuringly, this is the answer given in the back of the book. Our initial data lead to the equations 1 + 2 + 1 = 3 and 2 1 + 4 2 + 1 = 3, c whose solution is 1 = 3, 2 = −1. Finally, the solution to the inhomogeneous equation, with the initial condition given, is = 3 · 2 − 4 + 1. an Sometimes, a polynomial of the same degree as the right-hand side doesn‟t work. This happens when the characteristic equation has 1 as a root. If our equation had been − 6 −1 + 5 −2 = 3, an when we guessed that the particular solution was a constant C, we‟d have arrived at the equation C − C + C = 3, or 0 = 3. The way to deal with this is to increase the degree of the polynomial. Instead of assuming that the solution is constant, we‟ll assume that it‟s linear. In fact, we‟ll guess that it is of the form

$=nC$. Then we have $nC-6$ -1 $C+5$ -2 $C=3$, which simplifies to $C-10C=3$ so that C $=-34$. Thus, $g_n = -3\,n\,4$. This won‟t be enough if 1 is a root of multiplicity 2, that is, if $-1\ 2\ r$ is a factor of the characteristic polynomial. Then there is a particular solution of the form g_n $=Cn2$. For second-order equations, you never have to go past this. If the right-hand side is a polynomial of degree greater than 0, then the process works juts the same, except that you start with a polynomial of the same degree, increase the degree by 1, if necessary, and then once more, if need be. For example, if the right-hand side were $f_n=2\ \ -1$, we would start by guessing a particular solution $C1n$ 2. If it turned out that 1 was a characteristic root, we would amend our guess to $C1\,g_n\,2$ 3C If 1 is a double root, this will fail also, but g_n $=C1$ 2 $n\,3$ $C4\,n$ will work in thiscase.

Another case where there is a simple way of guessing a particular solution is when the right- hand side is an exponential, say $Cn.f\,n$ In that case, we guess that a particular solution is just a constant multiple of f, say ( )= . Again, we gave trouble when 1 is a characteristic root. We then guess that $g_n = kCn$, which will fail only if 1 is a double root. In that case we must use $kn2\,Cn$, which is as far as we ever have to go in the second-order case. These same ideas extend to higher-order recurrence relations, but we usually solve them numerically, rather than exactly. A third-order linear difference equation with constant coefficients leads to a cubic characteristic polynomial. There is a formula for the roots of a cubic, but it‟s very complicated.

For fourth-degree polynomials, there‟s also a formula, but it‟s even worse. For fifth and higher degrees, no such formula exists. Even for the third-order case, the exact solution of a simple-looking inhomogeneous linear recurrence relation with constant coefficients can take pages to write down. The coefficients will be complicated expressions involving square roots and cube roots. For most, if not all, purposes, a simpler answer with numerical coefficients is better, even though they must in the nature of things, be approximate.

The procedure I‟ve suggested may strike you as silly. After all, we‟ve already solved the characteristic equation, so we know whether 1 is a characteristic root, and what it‟s multiplicity is. Why not start with a polynomial of the correct degree? This is all well and good, while you‟re taking the course, and remember the procedure in detail. However, if you have to use this procedure some years from now, you probably won‟t remember all the details. Then the method I‟ve suggested will be valuable. Alternatively, you can start with a general polynomial of the maximum possible degree This leads to a lot of extra work if you‟re solving by hand, but it‟s the approach I prefer for computer solution.

UNIT-II
Algebraic structures

Algebraic systems:

An algebraic system, loosely speaking, is a set, together with some operations on the set. Before formally defining what an algebraic system is, let us recall that a n -ary operation (or operator) on a set A is a function whose domain is A_n and whose range is a subset of A . Here, n is a non-negative integer. When $n=0$, the operation is usually called a nullary operation, or a constant, since one element of A is singled out to be the (sole) value of this operation. A finitary operation on A is just an n -ary operation for some non-negative integer n.

Definition. An *algebraic system* is an ordered pair $(A\ O)$, where A is a set, called the underlying set of the algebraic system, and O is a set, called the operator set, of finitary operations on A .

We usually write $\boldsymbol{A}$, instead of $(A\ O)$, for brevity.

A prototypical example of an algebraic system is a group, which consists of the underlying set G , and a set O consisting of three operators: a constant e called the multiplicative identity, a unary operator called the multiplicative inverse, and a binary operator called the multiplication.

For a more comprehensive listing of examples, please see this entry.

Remarks.

An algebraic system is also called algebra for short. Some authors require that Abe non-empty.

Note that A is automatically non-empty if contains constants. A *finite algebra* is an algebra whose underlying set is finite.

By definition, all operators in an algebraic system are finitary. If we allow O to contain infinitary operations, we have an *infinitary algebraic system*. Other generalizations are possible. For example, if the operations are allowed to be multivalued, the algebra is said to be a *multialgebra*. If the operations are not everywhere defined, we get a *partial algebra*. Finally, if more than one underlying set is involved, then the algebra is said to be *many-sorted*.

The study of algebraic systems is called the theory of universal algebra. The first important thing in studying algebraic system is to compare systems that are of the same ``type''. Two algebras are said to have the same *type* if there is a one-to-one correspondence between their operator sets such that an n -ary operator in one algebra is mapped to an n-ary operator in the other algebra.

Examples:

Some recurring universes: **N**=natural numbers; **Z**=integers; **Q**=rational numbers; **R**=real numbers; **C**=complex numbers.

N is a pointed unary system, and under addition and multiplication, is both the standard interpretation of Peano arithmetic and a commutative semiring.

Boolean algebras are at once semigroups, lattices, and rings. They would even be abelian groups if the identity and inverse elements were identical instead of complements.

Group-like structures

- Nonzero **N** under addition (+) is a magma.
- **N** under addition is a magma with an identity.
- *Z under subtraction (−) is a quasigroup.*
- Nonzero **Q** under division (÷) is a quasi group.
- Every group is a loop, because $a * x = b$ if and only if $x = a^{-1} * b$
  - and only if $y = b * a^{-1}$.

2x2 matrices(of non-zero determinant) with matrix multiplication form a group.

Z under addition (+) is an abelian group.

Nonzero **Q** under multiplication (×) is an abelian group.

Every cyclic group *G* is abelian, because if *x*, *y* are in *G*, then $xy = a^m a^n = a^{m+n} = a^{n+m} = a^n a^m = yx$. In paartaicular, **Z** is an abelian group under addition, as is the integers modulo *n* **Z**/*n***Z**.

A monoid is a category with a single object, in which case the composition of morphisms and the identity morphism interpret monoid multiplication and identityelement, respectively.

The Boolean algebra **2** is a boundary algebra.

General Properties:

Property of Closure

If we take two *real numbers* and multiply them together, we get another real number. (The real numbers are all the rational numbers and all the irrational numbers.) Because this is always true, we say that the real numbers are "closed under the operation of multiplication": there is no way to escape the set. When you combine any two elements of the set, the result is also included in the set.

Real numbers are also closed under addition and subtraction. They are not closed under the square root operation, because the square root of -1 is not a real number.

Inverse

The inverse of something is that thing turned inside out or upside down. The inverse of an operation undoes the operation: division undoes multiplication.

A number's *additive inverse* is another number that you can add to the original number to get the additive identity. For example, the additive inverse of 67 is -67, because 67 + -67 = 0, the additive identity.

Similarly, if the product of two numbers is the *multiplicative identity*, the numbers are *multiplicative inverses*. Since 6 * 1/6 = 1 (the multiplicative identity), the multiplicative inverse of 6 is 1/6.

Zero does not have a multiplicative inverse, since no matter what you multiply it by, the answer is always 0, not 1.

Equality

The equals sign in an equation is like a scale: both sides, left and right, must be the same in order for the scale to stay in balance and the equation to be true.

The *addition property of equality* says that if a = b, then a + c = b + c: if you add the same number to (or subtract the same number from) both sides of an equation, the equation continues to be true.

The *multiplication property of equality* says that if a = b, then a * c = b * c: if you multiply (or divide) by the same number on both sides of an equation, the equation continues to be true.

The *reflexive property of equality* just says that a = a: anything is congruent to itself: the equals sign is like a mirror, and the image it "reflects" is the same as the original.

The *symmetric property of equality* says that if a = b, then b = a.

The *transitive property of equality* says that if a = b and b = c, then a = c.

Semi groups and monoids:

In the previous section, we have seen several algebraic system with binary operations. Here we consider an algebraic system consisting of a set and an associative binary operation on the set and then the algebraic system which possess an associative property with an identity element. These algebraic systems are called semigroups and monoids.

Semi group

Let S be a nonempty set and let * be a binary operation on S. The algebraic system (S, *) is called a semi-group if * is associative
if a * (b*c) = (a * b) * c for all a, b, c **Î** S.

Example The N of natural numbers is a semi-group under the operation of usual addition of numbers.

Monoids

Let M be a nonempty set with a binary operation * defined on it. Then (M, *) is called amonoid if * is associative (i.e) a * (b * c) = (a * b) * c for all a, b, c Î M andthere exists an element e in M such that a * e = e * a = a for all a Î M e is called the identity element in (M,*).
It is easy to prove that the identity element is unique. From the definition it follows that (M,*) isa semi group with identity.

Example1 Let S be a nonempty set and r(S) be its power set. The algebras (r(S),U) and (r(S), Ç) are monoids with the identities f and S respectively.

Example2 Let N be the set of natural numbers, then (N,+), (N, X) are monoids with the identities 0 and 1 respectively.

Groups Sub Groups:

Recalling that an algebraic system (S, *) is a semigroup if the binary operation * is associative. If there exists an identity element e Î S, then (S,*) is monoid. A further condition is imposed on the elements of the monoid, i.e., the existence of an inverse for each element of S then the algebraic system is called a group.

Definition

Let G be a nonempty set, with a binary operation * defined on it. Then the algebraic system (G,*) is called a **group** if

* is associative i.e. a * (b * c) = (a * b) * c for all a, b, c,Î G there exists an element e in G such that a * e = e * a = a for all a Î G for each a Î G there is an element denoted by a-1 in G such thata * a-1 = a-1 * a = e, a-1 is called the inverse of a. From the definition it follows that (G,*) is a monoid in which each element has an inverse w.r.t.* in G. A group (G,*) in which * is commutative is called an abelian group or a commutative group. If * is not commutative then (G,*) is called a non-abelian group or non-commutative group.

The order of a group (G,*) is the number of elements of G, when G is finite and is denotedby o(G) or |G| Examples 1. (Z5, +5) is an abelian group of order 5.

G = {1, -1, i, -i} is an abelian group with the binary operation x is definedas 1 x 1 = 1, -1 x -1 = 1, i x i = -1 , -i x -i **= 1, …**

Homomorphism of Semigroups and Monoids

Semigroup Homomorphism

Let (S, *) and (T, D) be any two semigroups. A mapping g: S ® T such that anytwo elements a, b Î S , g(a * b) = g(a) D g(b) is called a semigroup homomorphism.

Monoid Homomorphism

Let (M, *,eM) and (T, D,eT) be any two monoids. A mapping g: M® T such that anytwo elements a, b Î M ,
g(a * b) = g(a) D g(b) andg(eM) =
eT
is called a monoid homomorphism.

Theorem 1 Let (s, *) , (T, D) and (V, Å) be semigroups. A mapping g: S ® T and h: T ® V be semigroup homomorphisms. Then (hog): S ® V is a semigroup homomorphism from (S,*) to(V,Å).

Proof. Let a, b Î S. Then
(h o g)(a * b) = h(g(a* b))
= h(g(a) D g(b))
= h(g(a)) Å h(g(b))
= (h o g)(a) Å (h o g)(b)

Theorem 2 Let (s,*) be a given semi group. There exists a homomorphism g: S ® SS, where (SS, o) is a semi group of function from S to S under the operation of composition.
Proof For any element a Î S, let g(a) = fa where f aÎ SS and f a is defined by
f a(b) = a * b for any a, bÎ S
g(a * b) = f a*b Now f a*b(c) = (a * b) * c = a*(b * c)
where = f a(f b(c)) = (f a o f b) (c).

Therefore, g(a * b) = f a*b = f a o f b = g(a) o g(b), this shows that g: S ® SS is a homomorphism.

Theorem 3 For any commutative monoid (M, *),the set of idempotent elements of M forms a submonoid. Proof.

Let S be the set of idempotent elements of M.

Since the identity element e Î M is idempotent, e Î S.Let a, b Î S, so that a* a = a and b * b = b

Now (a * b) * (a * b) = (a * b) * (b * a)

[(M, *) is a commutative monoid]

= a * (b * b) * a

= a * b * a

= a * a * b

= a * b Hence a * b Î S and (S, *) is a submonoid.

Isomorphism:

In abstract algebra, an **isomorphism** is a bijective map *f* such that both *f* and its inverse f^{-1} are homomorphisms, i.e., *structure-preserving* mappings. In the more general setting of category theory, an **isomorphism** is a morphism *f*: *X* → *Y* in a category for which there exists an "inverse"f^{-1}: *Y* → *X*, with the property that both $f^{-1}f = \mathrm{id}_X$ and $f f^{-1} = \mathrm{id}_Y$.

Informally, an isomorphism is a kind of mapping between objects, which shows a relationship between two properties or operations. If there exists an isomorphism between two structures, we call the two structures **isomorphic**. In a certain sense, isomorphic structures are **structurally identical**, if you choose to ignore finer-grained differences that may arise from how they are defined.

Purpose:

Isomorphisms are studied in mathematics in order to extend insights from one phenomenon to others: if two objects are isomorphic, then any property which is preserved by an isomorphism and which is true of one of the objects, is also true of the other. If an isomorphism can be found from a relatively unknown part of mathematics into some well studied division of mathematics, where many theorems are already proved, and many methods are already available to find answers, then the function can be used to map whole problems out of unfamiliar territory over to "solid ground" where the problem is easier to understand and work with

. **Homomorphism of semigroups and monoids**

Semigroup homomorphism.

Let (S, *) and (T, D) be any two semigroups. A mapping g: S ® T such that any two elements a, b Î S , g(a * b) = g(a) D g(b) is called a semigroup homomorphism.

Monoid homomorphism

Let (M, *,eM) and (T, D,eT) be any two monoids. A mapping g: M® T such that any two elements a, b Î M , g(a * b) = g(a) D g(b) and g(eM) = eT

is called a monoid homomorphism.

Theorem 1 Let (s, *) , (T, D) and (V, Å) be semigroups. A mapping g: S ® T and h: T ® V be semigroup homomorphisms. Then (hog): S ® V is a semigroup homomorphism from (S,*) to(V,Å).

Proof. Let a, b Î S. Then

(h o g)(a * b) = h(g(a* b))

= h(g(a) D g(b))

= h(g(a)) Å h(g(b))

= (h o g)(a) Å (h o g)(b)

Theorem 2 Let (s,*) be a given semigroup. There exists a homomorphism g: S ® SS, where (SS, o) is a semigroup of function from S to S under the operation of composition.
Proof For any element a Î S, let g(a) = fa where f aÎ SS and f a is defined by
f a(b) = a *
b for any a, bÎ S
g(a * b) =
f a*b
Now f a*b(c) =
(a * b) * c = a*(b * c)
where =
f a(f b(c)) = (f a o f b) (c).

Therefore, g(a * b) = f a*b = f a o f b = g(a) o g(b), this shows that g: S ® SS is a homomorphism.
Congruence relations and quotient structures are fundamental concepts in mathematics, particularly in algebra, number theory, and abstract algebra. Here's an overview of each:

Congruence Relation:

In modular arithmetic and more generally in algebraic structures, a **congruence relation** on a set is a relation that preserves a certain property (typically arithmetic operations or algebraic structures) under the relation.

1. **Definition**: A congruence relation on a set SSS is an equivalence relation that is compatible with some algebraic operation defined on SSS.
2. **Example**: In modular arithmetic, a≡b(modn)a \equiv b \pmod{n}a≡b(modn) means aaa and bbb have the same remainder when divided by nnn. This relation is compatible with addition and multiplication modulo nnn.
3. **Properties**: A congruence relation has the following properties:

Reflexivity: a≡aa \equiv aa≡a.
Symmetry: If a≡ba \equiv ba≡b, then b≡ab \equiv ab≡a.
Transitivity: If a≡ba \equiv ba≡b and b≡cb \equiv cb≡c, then a≡ca \equiv ca≡c.
Applications: Congruence relations are used in various areas such as number theory (modular arithmetic), abstract algebra (group theory, ring theory), and computer science (in defining equivalence classes).

Quotient Structures:

A **quotient structure** is formed by partitioning a set with respect to an equivalence relation (like a congruence relation) and then defining operations or structures on the resulting equivalence classes.

1. **Quotient Set**: Given a set SSS and an equivalence relation ∼\sim∼ on SSS, the quotient set S/∼S / \simS/∼ consists of equivalence classes of SSS under ∼\sim∼.
2. **Operations on Quotient Structures**: Typically, operations defined on the quotient set are derived from operations on SSS. For instance, if SSS is a group and ∼\sim∼ is a congruence relation that respects the group operation, then S/∼S / \simS/∼ can inherit a group structure.
3. **Example**: In modular arithmetic, Z/nZ\mathbb{Z} / n\mathbb{Z}Z/nZ represents the set of equivalence classes modulo nnn. Addition and multiplication can be defined on these classes (e.g., [a]+[b]=[a+b]mod n[a] + [b] = [a + b] \mod n[a]+[b]=[a+b]modn).
4. **Properties**: Quotient structures preserve the essential properties of the original structure (like a group, ring, or module) while abstracting away details within equivalence classes.

Free monoids and groups, as well as cyclic monoids and groups, are important concepts in algebra and group theory. Let's explore each of these concepts in detail:

Free Monoids and Groups:

1. **Free Monoid**:

Definition: A **free monoid** on a set SSS is the monoid formed by all finite sequences (or strings) that can be constructed using elements from SSS, with the operation being concatenation of sequences and the identity element being the empty sequence.

Example: If S={a,b}S = \{a, b\}S={a,b}, then the free monoid M(S)M(S)M(S) consists of all strings that can be formed using aaa and bbb, such as ϵ\epsilonϵ (empty string), aaa, bbb, aaaaaa, ababab, bababa, etc. Concatenation of strings is the monoid operation.

2. **Free Group**:

Definition: A **free group** on a set SSS is a group formed by all possible reduced words (sequences of elements and their inverses) that can be constructed from SSS, with the group operation being concatenation and reduction (removing pairs of an element and its inverse).

○ **Example**: If S={a,b}S = \{a, b\}S={a,b}, then the free group F(S)F(S)F(S) consists of all reduced words that can be formed using aaa, bbb, a−1a^{-1}a−1, and b−1b^{-1}b−1, such as ϵ\epsilonϵ (empty word), aaa, bbb, ababab, ba−1ba^{-1}ba−1, etc. The group operation is concatenation followed by reduction.

Cyclic Monoids and Groups:

1. **Cyclic Monoid**:

Definition: A **cyclic monoid** is a monoid where every element can be expressed as a power of a single generator. In other words, there exists an element ggg such that every element of the monoid can be written as gng^ngn for some integer n≥0n \geq 0n≥0.

Example: The monoid of natural numbers under addition (N\mathbb{N}N with 000) is a cyclic monoid generated by 111, since every natural number nnn can be expressed as 1n1^n1n.

2. **Cyclic Group**:

Definition: A **cyclic group** is a group that is generated by a single element ggg. In other words, every element of the group can be expressed as gng^ngn for some integer nnn.

Example: The group of integers under addition Z\mathbb{Z}Z is a cyclic group generated by 111 or −1-1−1. The group of nnn-th roots of unity under multiplication is a cyclic group generated by e2πi/ne^{2\pi i / n}e2πi/n.

Key Differences:

Free vs. Cyclic: Free monoids and groups are characterized by the absence of relations between generators, allowing for arbitrary sequences or words. Cyclic monoids and groups, on the other hand, are characterized by having a single generator that can produce all other elements through repeated application of the operation (concatenation or multiplication).

Structure: Free structures are typically more flexible and have more elements (sequences or words) compared to cyclic structures, which are constrained by the generator and its powers.

Applications: Free structures are used in formal languages, automata theory, and combinatorics, while cyclic structures are fundamental in number theory, group theory, and the study of symmetry.

Understanding these concepts helps in grasping fundamental aspects of algebraic structures and their applications across various branches of mathematics.

Permutation groups are a fundamental concept in group theory that arise from studying the symmetries of a set. Here's a detailed explanation of permutation groups:

Definition and Basics:

1. **Permutation of a Set**:

A **permutation** of a set SSS is a bijective (one-to-one and onto) mapping σ:S⟶S\sigma: S \to Sσ:S→S. In simpler terms, it is a rearrangement of the elements of SSS.

2. **Permutation Group**:

A **permutation group** GGG on a set SSS is a subgroup of the symmetric group SSS_SSS, where SSS_SSS consists of all permutations of SSS. This means GGG is a set of permutations of SSS that forms a group under function composition.

3. **Group Action**:

A permutation group GGG acts on SSS if each element g∈Gg \in Gg∈G induces a permutation of SSS (i.e., ggg maps elements of SSS to other elements of SSS in a bijective manner).

Examples and Properties:

1. **Symmetric Group**:

The most basic example of a permutation group is the **symmetric group** SnS_nSn, which consists of all permutations of {1,2,…,n}\{1, 2, \ldots, n\}{1,2,…,n}. The order of SnS_nSn is n!n!n! (n factorial).

2. **Transpositions**:

A **transposition** is a permutation that swaps two elements and leaves all other elements fixed. For example, in S3S_3S3, (12)(12)(12) is a transposition that swaps 1 and 2.

3. **Cycle Notation**:

Permutations can be represented in **cycle notation**, which expresses how elements are moved cyclically. For example, (123)(123)(123) in S3S_3S3 means 1 goes to 2, 2 goes to 3, and 3 goes back to 1.

4. **Permutation Group Structure**:

Permutation groups can have different structures depending on the set SSS and the permutations involved. They can be cyclic, abelian, or non-abelian depending on the elements and their compositions.

Applications and Importance:

1. **Group Theory**:
Permutation groups are central to the study of group theory, providing insights into group structures, subgroups, and group actions.
2. **Combinatorics**:
They are essential in combinatorics for counting and analyzing arrangements, selections, and permutations of objects.
3. **Cryptography**:
Permutation groups are used in cryptographic algorithms for scrambling data and ensuring security through permutation-based encryption techniques.
4. **Symmetry**:
They capture the symmetries of mathematical objects, such as geometric shapes and arrangements, aiding in their classification and study.

In mathematics, specifically in algebra and related disciplines, the concept of substructures refers to smaller structures that are contained within larger structures. Here's a detailed explanation of substructures in different contexts:

Substructures in Algebra:

1. **Subgroups**:
In group theory, a **subgroup** of a group GGG is a subset of GGG that forms a group under the same group operation as GGG. It must include the identity element of GGG and be closed under inverses and the group operation.
Example: In Z\mathbb{Z}Z (integers under addition), the subset 2Z={2n∣n∈Z}2\mathbb{Z} = \{ 2n \mid n \in \mathbb{Z} \}2Z={2n∣n∈Z} forms a subgroup, as it is closed under addition, contains 0, and has inverses.
2. **Subrings**:
In ring theory, a **subring** of a ring RRR is a subset of RRR that forms a ring under the same addition and multiplication operations as RRR. It must contain the multiplicative identity of RRR and be closed under addition, multiplication, and additive inverses.
Example: Z\mathbb{Z}Z is a subring of Q\mathbb{Q}Q (rational numbers under addition and multiplication).
3. **Subfields**:
In field theory, a **subfield** of a field FFF is a subset of FFF that forms a field under the same addition, multiplication, and multiplicative inverse operations as FFF. It must include the multiplicative identity of FFF and be closed under these operations.
Example: Q\mathbb{Q}Q (rational numbers) is a subfield of R\mathbb{R}R (real numbers).

Substructures in Other Areas:

1. **Subspaces**:

In linear algebra, a **subspace** of a vector space VVV over a field FFF is a subset of VVV that is itself a vector space over FFF, under the same vector addition and scalar multiplication operations.

Example: The set of all polynomials of degree at most nnn forms a subspace of the vector space of all polynomials.

2. **Subgraphs**:

In graph theory, a **subgraph** of a graph GGG is a graph formed by selecting a subset of vertices and a subset of edges from GGG, such that the selected edges connect only the selected vertices.

Example: Removing some vertices and corresponding edges from a complete graph KnK_nKn results in a subgraph.

Properties and Relationships:

Inclusion: A substructure AAA is said to be contained within a structure BBB (denoted A⊆BA \subseteq BA⊆B) if AAA consists of elements or objects that also belong to BBB, and AAA inherits the operations and properties defined on BBB.

Closure: Substructures are closed under the operations of the larger structure. For instance, a subgroup remains closed under group operations within the larger group.

Generators: Substructures can often be generated by a subset of the elements of the larger structure that satisfy certain properties. For example, a subgroup can be generated by a subset of the group's elements.

1. Normal subgroups are a specific type of subgroup in group theory that play a significant role in understanding the structure and properties of groups. Here's a detailed explanation of normal subgroups:
Key Properties:

Normal subgroups are closed under conjugation: gNg−1⊆NgNg^{-1} \subseteq NgNg−1⊆N for all g∈Gg \in Gg∈G.

They form a kernel for group homomorphisms: If φ:G→H\varphi: G \to Hφ:G→H is a group homomorphism, ker (φ)={g∈G∣φ(g)=eH}\ker(\varphi) = \{ g \in G \mid \varphi(g) = e_H \}ker(φ)={g∈G∣φ(g)=eH} is a normal subgroup of GGG.

The quotient group G/NG/NG/N formed by the cosets of NNN in GGG is naturally a group under the operation (gN)(hN)=(gh)N(gN)(hN) = (gh)N(gN)(hN)=(gh)N.

Examples:

1. **Trivial Subgroup**:

Every group GGG has two trivial normal subgroups: {e}\{e\}{e} (the trivial subgroup consisting only of the identity element) and GGG itself.

2. **Non-trivial Examples**:

In Z\mathbb{Z}Z, the subgroup 2Z={2n∣n∈Z}2\mathbb{Z} = \{ 2n \mid n \in \mathbb{Z} \}2Z={2n∣n∈Z} is a normal subgroup because n+2Z=2Z+nn + 2\mathbb{Z} = 2\mathbb{Z} + nn+2Z=2Z+n for all n∈Zn \in \mathbb{Z}n∈Z.

In S3S_3S3 (the symmetric group on 3 elements), the subgroup {(),(12)(3),(13)(2),(23)(1),(123),(132)}\{(), (12)(3), (13)(2), (23)(1), (123), (132)\}{(),(12)(3),(13)(2),(23)(1),(123),(132)} is a normal subgroup.

Importance and Applications:

1. **Factorization Theorem (Isomorphism Theorems)**:

Normal subgroups are crucial in the study of factorization theorems in group theory, such as the First and Second Isomorphism Theorems, which relate the structure of a group to the structure of its normal subgroups and quotient groups.

2. **Group Actions and Representations**:

Normal subgroups are used to study group actions and representations, providing a way to understand symmetry properties and structure-preserving transformations in various mathematical contexts.

3. **Applications in Algebraic Structures**:

Normal subgroups help classify groups into different types (e.g., simple groups, solvable groups), leading to deeper insights into algebraic structures and their relationships.

Algebraic structures with two operations refer to mathematical structures where two binary operations are defined, typically satisfying certain axioms. Here are some fundamental algebraic structures with two operations:

1. Ring:

- **Definition**: A **ring** (R,+,·)(R, +, \cdot)(R,+,·) consists of a set RRR equipped with two binary operations: Addition +++: R×R→RR \times R \to RR×R→R, which is associative, commutative, has an identity element (0), and each element has an inverse.

Multiplication ·\cdot·: R×R→RR \times R \to RR×R→R, which is associative and distributive over addition.

Examples: **Integers Z\mathbb{Z}Z**: Z\mathbb{Z}Z forms a ring under usual addition and multiplication.

Polynomials R[x]\mathbb{R}[x]R[x]: The set of polynomials with real coefficients forms a ring under polynomial addition and multiplication.

2. Field:

Definition: A **field** (F,+,·)(F, +, \cdot)(F,+,·) is a ring where every non-zero element has a multiplicative inverse. Addition +++ and multiplication ·\cdot· satisfy all the ring properties.

Multiplicative inverses exist for all non-zero elements.

Examples:

Rational numbers Q\mathbb{Q}Q: Q\mathbb{Q}Q forms a field under usual addition and multiplication.

Real numbers R\mathbb{R}R: R\mathbb{R}R is a field under usual addition and multiplication.

3. Algebra:

Definition: An **algebra** over a field FFF (or ring) is a vector space over FFF equipped with an additional bilinear (or linear) multiplication operation.

The vector space structure uses addition and scalar multiplication from the field FFF.

The multiplication operation is bilinear or linear over FFF.

Examples:

Vector spaces: A vector space VVV over a field FFF is an algebra with scalar multiplication and vector addition.

Matrix algebra: The set of n×nn \times nn×n matrices over a field FFF, with matrix addition and matrix multiplication, forms an algebra.

4. Lattice:

Definition: A **lattice** (L,∨,∧)(L, \vee, \wedge)(L,∨,∧) is a partially ordered set where any two elements have a least upper bound (join ∨\vee∨) and a greatest lower bound (meet ∧\wedge∧).

The operations ∨\vee∨ (join) and ∧\wedge∧ (meet) are associative, commutative, idempotent, and satisfy absorption laws.

Examples:

Boolean algebra: The set of all subsets of a given set XXX, with union (join) and intersection (meet), forms a lattice. **Intervals in real numbers**: The set of all intervals in R\mathbb{R}R, with union and intersection, forms a lattice.

Ordered Ring or Ordered Field:

Definition: An **ordered ring** or an **ordered field** is a ring or a field (R,+,·)(R, +, \cdot)(R,+,·) equipped with a total order ≤\leq≤ that is compatible with addition and multiplication.

RRR is an ordered set under ≤\leq≤. ≤\leq≤ satisfies compatibility with addition and multiplication: a≤b⇒a+c≤b+ca \leq b \Rightarrow a + c \leq b + ca≤b⇒a+c≤b+c and a≥0,b≥0⇒ab≥0a \geq 0, b \geq 0 \Rightarrow ab \geq 0a≥0,b≥0⇒ab≥0.

Examples:

Real numbers R\mathbb{R}R: R\mathbb{R}R is an ordered field with the usual order relation and operations.

Polynomial rings over ordered fields: For example, R[x]\mathbb{R}[x]R[x] is an ordered ring when R\mathbb{R}R is ordered.

Lattice and its Properties:

Introduction:
A lattice is partially ordered set (L, £) in which every pair of elements a, b ÎL has a
greatest lower bound and a least upper bound.
The glb of a subset, {a, b} Í L will be denoted by a * b and the lub by a Å b.
Usually, for any pair a, b Î L, GLB {a, b} = a * b, is called the **meet** or **product** and LUB{a, b} = a Å b, is called the **join** or **sum** of a and b.
Example1 Consider a non-**empty set S and let P(S) be its power set. The relation Í** “contained in” is a partial **ordering on P(S). For any two subsets A, BÎ P(S)**
GLB {A, B} and LUB {A, B} are evidently A Ç B and A È B respectively.
Example2 Let I+ be the set of positive integers, and D denote the relation of “division” in I+ such that for any a, b Î I+ , a D b iff a divides b. Then (I+, D) is a lattice in which
the join of a and b is given by the least common multiple(LCM) of a and b, that is,
a Å b = LCM of a and b, and the meet of a and b, that is , a * b is the greatest common divisor (GCD) of a and b.
A lattice can be conveniently represented by a diagram.
For example, let Sn be the set of all divisors of n, where n is a positive integer. Let D denote the relation “division” such that for any a, b Î Sn, a D b iff a di**vides b.**
Then (Sn, D) is a lattice with a * b = gcd(a, b) and a Å b = lcm(a, b).
Take n=6. Then S6 = {1, 2, 3, 6}. It can be represented by a diagram in Fig(1). Take n=8. Then S8 = {1, 2, 4, 8}
Two lattices can have the same diagram. For example if S = {1, 2, 3} then (p(s), Í) and (S6,D)
Have the same diagram viz. fig(1), but the nodes are differently labeled .We observe that for any partial ordering relation £ on a set S the converse relation is also partial ordering relation on S. If (S,£) is a lattice
With meet a *b and join a Å b , then (S, ³) is the also a lattice with meet a Å b and join a * b i.e., the GLB and LUB get interchanged . Thus we have the principle of duality of lattice as follows.
Any statement about lattices involving the operations ^ and V and the relations £ and ³ remains true if ^, V, ³ and £ are replaced by V, ^, £ and ³ respectively.
The operation ^ and V are called duals of each other as are the relations £ and ³.. Also, the lattice (L, £) and (L, ³) are called the duals of each other.
Properties of lattices:
Let (L, £) be a lattice with the binary operations * and Å then for any a, b, c Î L,
a * a = a
a Å a = a
(Idempotent)
a * b = b * a
a Å b = b Å a
(Commutative)
(a * b) * c = a * (b * c) , (a Å) Å c = a Å (b Å c)
○
(Associative)
a * (a Å b) = a
a Å (a * b) = a
(absorption)
For any a ÎL, a £ a, a £ LUB {a, b} => a £ a * (a Å b). On the other hand, GLB {a, a Å b} £ a i.e., (a Å b) Å a, hence a * (a Å b) = a
Theorem 1
Let (L, £) be a lattice with the binary operations * and Å denote the operations of meet and join respectively

For any a, b Î L,
a £ b ó a * b = a ó a Å b = b
Proof
Suppose that a £ b. we know that a £ a, a £ GLB {a, b}, i.e., a £ a * b. But from the definition of a * b, we get a * b £ a.
Hence a £ b => a * b = a ....(1)
Now we assume that a * b = a; but is possible only if a £ b,
that is a * b = a => a £ b ·····························
(2)
From (1) and (2), we get a £ b ó a * b = a.
Suppose a * b = a.
then b Å (a * b) = b Å a = a Å b ························.
(3)
but b Å (a * b) = b (by iv)·······················. .
(4)
Hence a Å b = b, from (3) => (4)
Suppose aÅ b = b, i.e., LUB {a, b} =
b, this is possible only if a£ b, thus(3) => (1)
(1) => (2) => (3) => (1). Hence these are equivalent.
Let us assume a * b = a.
Now (a * b) Å b = a Å b
We know that by absorption law , (a * b) Å b = b
so that a Å b = b, therefore a * b = a Þ a Å b = b
(5)
similarly, we can prove a Å b = b Þ
a * b = a
(6)
From (5) and (6), we get
a * b = a Û a Å b = b Hence the theorem.
Theorem2 For any a, b, c Î L, where (L, £) is a lattice. b
£ c => { a * b £ a * c and
{ a Å b £ a Å c
Proof Suppose b £ c. we have proved that b £ a ó b * c = b..............**(1)**
Now consider
(a * b) * (a * c) = (a * a) * (b * c) (by Idempotent)
= a * (b * c)
= a * b (by (1))
Thus (a * b) * (a * c) = a * b which => (a * b) £ (a * c) Similarly (a Å b) Å (a Å c) = (a Å a) Å (b Å c)
= a Å (b Å c)
= a Å c which => (a Å b) £ (a Å c)

note:These properties are known as **isotonicity.**

Principles of Duality

The principles of duality in mathematics refer to a concept where certain mathematical structures or statements can be transformed into equivalent forms by interchanging certain elements or operations. This concept appears in various branches of mathematics, offering deeper insights and connections between seemingly different concepts. Here's an exploration of the principles of duality across different mathematical areas:

1. Boolean Algebra and Lattice Theory:

Principle: In Boolean algebra and lattice theory, duality is expressed through operations and relationships between meets (infimums) and joins (supremums).

Example: In Boolean algebra, De Morgan's laws illustrate duality: ¬(a∨b)=(¬a)∧(¬b)\neg (a \vee b) = (\neg a) \wedge (\neg b)¬(a∨b)=(¬a)∧(¬b) and ¬(a∧b)=(¬a)∨(¬b)\neg (a \wedge b) = (\neg a) \vee (\neg b)¬(a∧b)=(¬a)∨(¬b). These laws show how operations and relationships between meets and joins can be interchanged.

2. Linear Algebra:

Principle: Duality in linear algebra relates vector spaces and their dual spaces, showing how linear functionals and linear transformations can be paired.

Example: The concept of dual spaces V∗V^*V∗ for a vector space VVV allows us to study linear functionals and their relationships with elements of VVV. The Riesz representation theorem in Hilbert spaces is a notable example of this duality.

3. Graph Theory:

Principle: Duality in graph theory relates properties of a graph to those of its complement or dual graph.

Example: The concept of planar duality in graph theory relates a graph to its dual graph in the context of planar graphs. Properties such as faces, vertices, and edges are transformed under this duality relationship.

4. Topology:

Principle: In topology, duality often relates spaces and their complements or certain transformations.

Example: In point-set topology, the duality principle is illustrated by concepts like open sets and closed sets, where the complement of an open set is a closed set and vice versa.

5. Order Theory:

Principle: Duality in order theory relates order relations and properties of elements within a partially ordered set.

Example: The principle of duality in order theory can be seen through concepts such as least upper bounds and greatest lower bounds, where the properties and operations between these elements can be interchanged under certain conditions.

Benefits and Applications:

Insight and Symmetry: Duality often reveals underlying symmetries and relationships between different mathematical structures or statements.

Problem Solving: It can simplify problem-solving by transforming a problem into an equivalent form that may be easier to analyze.

Unified Framework: Duality provides a unified framework for understanding seemingly disparate concepts across different branches of mathematics.

DISTRIBUTIVE AND COMPLIMENTED LATTICES:

In lattice theory, distributive and complemented lattices are important classes of lattices that exhibit specific algebraic properties. Let's explore each of these concepts in detail:

Distributive Lattices:

1. **Definition**:

A **distributive lattice** is a lattice (L,∨,∧)(L, \vee, \wedge)(L,∨,∧) where the operations of meet ∧\wedge∧ and join ∨\vee∨ distribute over each other.

Mathematically, distributivity is defined as a∧(b∨c)=(a∧b)∨(a∧c)a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)a∧(b∨c)=(a∧b)∨(a∧c) and a∨(b∧c)=(a∨b)∧(a∨c)a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)a∨(b∧c)=(a∨b)∧(a∨c) for all elements a,b,c∈La, b, c \in La,b,c∈L.

2. **Example**:

Boolean Algebra: The lattice of subsets of a set XXX, ordered by inclusion, forms a distributive lattice. Here, meet corresponds to intersection of sets A∩BA \cap BA∩B, and join corresponds to union of sets A∪BA \cup BA∪B.

3. **Properties**:

Distributive lattices satisfy the distributive laws, which generalize the distributive laws of set theory.

They have unique complements for each element, leading to a well-defined lattice structure.

Complemented Lattices:

1. **Definition**:

A **complemented lattice** is a lattice (L,∨,∧)(L, \vee, \wedge)(L,∨,∧) where every element a∈La \in La∈L has a complement a′a'a′ such that a∨a′=1a \vee a' = 1a∨a′=1 (the lattice's maximum element) and a∧a′=0a \wedge a' = 0a∧a′=0 (the lattice's minimum element).

The complement a′a'a′ is unique for each element aaa, satisfying these conditions.

2. **Example**:

Boolean Algebra: The lattice of subsets of a set XXX, ordered by inclusion, is also a complemented lattice. The complement of a subset A⊆XA \subseteq XA⊆X is its complement Ac=X\AA^c = X \setminus AAc=X\A.

3. **Properties**:

Complemented lattices generalize Boolean algebras by relaxing the requirement of distributivity while retaining the existence of complements.

They allow for the formulation of logical operations similar to those in Boolean algebra, such as negation, conjunction, and disjunction.

Relationship between Distributive and Complemented Lattices:

Boolean Algebras: A **Boolean algebra** is a distributive lattice that is also complemented. It combines the distributive properties of sets with the existence of complements for each element.

Examples: The power set of any set, ordered by inclusion, is a Boolean algebra. This includes all subsets of a given set, with operations defined by set union and intersection.

Applications:

Logic and Set Theory: Boolean algebras are fundamental to formal logic, set theory, and computer science (especially in Boolean logic and circuit design).

Order Theory: Lattices provide a framework for studying order relations and hierarchy in various mathematical and applied contexts.

Boolean Lattices and Boolean Algebra:

Boolean lattices and Boolean algebra are closely related concepts in mathematics, specifically within the framework of lattice theory and algebraic structures. Let's explore each of these concepts in detail:

Boolean Lattices:

1. **Definition**:

A **Boolean lattice** is a lattice (L,∨,∧)(L, \vee, \wedge)(L,∨,∧) where each element has a complement (or negation) and satisfies the following properties:

Complement Law: For every element a∈La \in La∈L, there exists an element a′∈La' \in La′∈L such that a∨a′=1a \vee a' = 1a∨a′=1 (the lattice's maximum element) and a∧a′=0a \wedge a' = 0a∧a′=0 (the lattice's minimum element).

Idempotent Law: a∨a=aa \vee a = aa∨a=a and a∧a=aa \wedge a = aa∧a=a for all a∈La \in La∈L.

Commutative, Associative, and Distributive Laws: The operations ∨\vee∨ (join) and ∧\wedge∧ (meet) are commutative, associative, and distributive over each other.

2. **Examples**:

Power Set: The lattice of all subsets of a set XXX, ordered by inclusion, is a Boolean lattice. Here, the join operation ∨\vee∨ corresponds to set union and the meet operation ∧\wedge∧ corresponds to set intersection.

Boolean Algebra: Every Boolean algebra is a Boolean lattice, but not all Boolean lattices are Boolean algebras (unless they satisfy additional closure properties).

3. **Properties**:

Boolean lattices generalize the properties of sets (union and intersection) into a structured algebraic framework.

They provide a formalism for reasoning about logical operations, truth values, and subsets in a systematic manner.

Boolean Algebra:

1. **Definition**:

A **Boolean algebra** (B,∨,∧,¬,0,1)(B, \vee, \wedge, \neg, 0, 1)(B,∨,∧,¬,0,1) is a special type of Boolean lattice where:

(B,∨,∧)(B, \vee, \wedge)(B,∨,∧) is a Boolean lattice.

¬\neg¬ (negation) is an unary operation satisfying ¬¬a=a\neg \neg a = a¬¬a=a and De Morgan's laws: ¬(a∨b)=¬a∧¬b\neg (a \vee b) = \neg a \wedge \neg b¬(a∨b)=¬a∧¬b and ¬(a∧b)=¬a∨¬b\neg (a \wedge b) = \neg a \vee \neg b¬(a∧b)=¬a∨¬b. 000 (zero) and 111 (one) are designated elements of BBB such that a∨0=aa \vee 0 = aa∨0=a, a∧1=aa \wedge 1 = aa∧1=a, a∨1=1a \vee 1 = 1a∨1=1, and a∧0=0a \wedge 0 = 0a∧0=0 for all a∈Ba \in Ba∈B.

2. **Examples**:

Binary Boolean Algebra: The set {0,1}\{0, 1\}{0,1} with operations defined by standard logical operations (AND, OR, NOT) forms a Boolean algebra.

Algebra of Propositions: The set of all propositions with operations defined by logical connectives (AND, OR, NOT) forms a Boolean algebra.

3. **Properties**:

Boolean algebras provide a formalism for manipulating truth values, logical operations, and propositions.

They are fundamental in formal logic, set theory, digital electronics (as they model digital circuits), and computer science.

Relationship Between Boolean Lattices and Boolean Algebra:

Every Boolean algebra is a Boolean lattice because it satisfies the lattice properties (associativity, commutativity, distributivity) and the additional properties of negation, zero, and one.

Boolean lattices provide a broader class of structures than Boolean algebras alone, encompassing any lattice where complementation is defined.

Applications:

Logic: Boolean algebras are essential in formal logic for analyzing truth values, logical propositions, and deductions.

Computer Science: They are used extensively in digital electronics, circuit design, Boolean logic operations (e.g., in programming and algorithms), and database queries (Boolean search operations).

Mathematics: They provide a foundational structure for studying order theory, lattice theory, and algebraic structures in general.

UNIQUENESS OF FINITE BOOLEAN ALGEBRA:

In the context of finite Boolean algebras, there are several aspects of uniqueness that can be discussed. Let's explore the uniqueness of finite Boolean algebras in terms of their cardinality, structure, and properties:

Cardinality:

1. **Cardinality of Finite Boolean Algebras**:

The cardinality of a finite Boolean algebra BBB, denoted as ∣B∣|B|∣B∣, must be a power of 2 (i.e., ∣B∣=2n|B| = 2^n∣B∣=2n for some non-negative integer nnn).

This follows from the fact that every finite Boolean algebra is isomorphic to the power set of its minimal generating set, which inherently has 2n2^n2n elements where nnn is the number of elements in the generating set.

2. **Uniqueness of Cardinality**:

For a given cardinality 2n2^n2n, there exists a unique (up to isomorphism) finite Boolean algebra.

This uniqueness is derived from the structure theorem for finite Boolean algebras, which states that every finite Boolean algebra of size 2n2^n2n is isomorphic to P(X)\mathcal{P}(X)P(X), where XXX is a set of size nnn.

Structure and Isomorphism:

1. **Isomorphism Theorems**:

Two finite Boolean algebras of the same cardinality 2n2^n2n are isomorphic to each other.
This isomorphism is based on the fact that any two finite Boolean algebras of the same cardinality have the same structure and properties, including the number of elements and their relationships under Boolean operations (AND, OR, NOT).

2. **Structure Theorem**:

Every finite Boolean algebra BBB is isomorphic to P(X)\mathcal{P}(X)P(X), where XXX is a set of atoms (minimal non-zero elements) of BBB.

This theorem ensures that the structure of a finite Boolean algebra is uniquely determined by its underlying set of atoms and the Boolean operations defined on them.

Conclusion:

The uniqueness of finite Boolean algebras primarily revolves around their cardinality and isomorphism properties. For a given finite Boolean algebra of size 2n2^n2n, there exists a unique structure and set of operations that define it, up to isomorphism. This uniqueness is fundamental in the study of Boolean algebras, providing a clear and systematic way to analyze and categorize these algebraic structures based on their finite size and properties.

BOOLEAN FUNCTIONS AND BOOLEAN EXPRESSIONS:

Boolean functions and Boolean expressions are fundamental concepts in computer science and mathematics, particularly in the context of logic and digital circuit design. Let's explore each of these concepts in detail:

Boolean Functions:

1. **Definition**:

A **Boolean function** f:{0,1}n→{0,1}f: \{0, 1\}^n \to \{0, 1\}f:{0,1}n→{0,1} is a function that operates on nnn Boolean variables (inputs), each taking values from {0,1}\{0, 1\}{0,1} (typically interpreted as false and true, respectively), and produces a single Boolean output.

The output f(x1,x2,…,xn)f(x_1, x_2, \ldots, x_n)f(x1,x2,…,xn) is determined by applying logical operations (AND, OR, NOT, etc.) to the input variables x1,x2,…,xnx_1, x_2, \ldots, x_nx1,x2,…,xn.

2. **Representation**:

Boolean functions can be represented in various forms, including:

Truth Table: A complete enumeration of all possible input combinations and their corresponding outputs.

Boolean Algebraic Expression: An algebraic expression constructed using Boolean operations (AND, OR, NOT, XOR, etc.) that describes how the outputs are derived from the inputs.

Boolean Circuit: A graphical representation composed of logic gates (AND, OR, NOT gates) that implement the Boolean function.

3. **Examples**:

AND function: f(x,y)=x⋅yf(x, y) = x \cdot yf(x,y)=x⋅y
OR function: f(x,y)=x+yf(x, y) = x + yf(x,y)=x+y
NOT function: f(x)=¬xf(x) = \neg xf(x)=¬x
XOR function: f(x,y)=x⊕yf(x, y) = x \oplus yf(x,y)=x⊕y (exclusive OR)

4. **Properties**:

Completeness: A Boolean function is defined for every possible combination of input values.

Determinism: Given the same input values, a Boolean function always produces the same output.

Operations: Boolean functions can be combined using Boolean algebra to create more complex functions.

Boolean Expressions:

1. **Definition**:

A **Boolean expression** is a symbolic representation of a Boolean function using variables, constants (0 and 1), and logical operations (AND, OR, NOT, XOR, etc.).

It represents how inputs are combined and manipulated to produce the output of the Boolean function.

2. **Forms of Boolean Expressions**:

Canonical Form: A Boolean expression in which each term represents a product (AND) of literals (variables or their negations) that must be combined using OR operations.

Simplified Form: A Boolean expression that is reduced to its simplest form using Boolean algebraic rules (e.g., De Morgan's laws, distribution laws) or Karnaugh maps (for optimization).

3. **Examples**:
- **Canonical Form**: f(x,y,z)=x¯yz+xy¯z+xyz¯+xyzf(x, y, z) = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyzf(x,y,z)=x¯yz+xy¯z+xyz¯+xyz
- **Simplified Form**: f(x,y,z)=x+yzf(x, y, z) = x + yzf(x,y,z)=x+yz

4. **Applications**:
- **Digital Circuit Design**: Boolean expressions are used to design and analyze digital circuits composed of logic gates.
- **Logic and Computer Science**: Boolean expressions are fundamental in formal logic, programming (in conditions and control flow), and algorithms (especially in decision-making processes).

Relationship Between Boolean Functions and Boolean Expressions:

Representation: Boolean functions can be represented by Boolean expressions, which provide a symbolic and concise way to describe the function's behavior.

Evaluation: Boolean expressions describe how inputs are combined to produce outputs, while Boolean functions define the actual mapping from inputs to outputs.

Interchangeability: Boolean functions and Boolean expressions are interchangeable in the sense that each Boolean function has a corresponding Boolean expression, and vice versa, capturing the same logical behavior. In summary, Boolean functions and Boolean expressions are essential concepts in computer science and mathematics, providing the foundation for digital logic, circuit design, and formal reasoning about logical operations and conditions. They play a critical role in various applications where logical operations and decision-making are involved.

Propotional Calculus:

Propositional calculus, also known as propositional logic or sentential logic, is a formal system in logic that deals with propositions (statements that can be either true or false) and their logical relationships. It forms the basis for reasoning about compound propositions using logical connectives. Let's delve into the key aspects of propositional calculus:

Basics of Propositional Calculus:

1. **Propositions**:

A **proposition** is a declarative statement that is either true or false. Examples include "It is raining" (which could be true or false depending on the weather) and "2 + 2 = 5" (which is false).

2. **Logical Connectives**:

Negation (¬): Denoted as ¬p\neg p¬p, it reverses the truth value of a proposition ppp.

Conjunction (AND, ∧): Denoted as p∧qp \land qp∧q, it is true if both propositions ppp and qqq are true.

Disjunction (OR, ∨): Denoted as p∨qp \lor qp∨q, it is true if at least one of the propositions ppp or qqq is true.

Implication (→): Denoted as p→qp \rightarrow qp→q, it is false only when ppp is true and qqq is false; otherwise, it is true.

Biconditional (↔): Denoted as p↔qp \leftrightarrow qp↔q, it is true when both propositions ppp and qqq have the same truth value (both true or both false).

3. **Logical Equivalences**:

De Morgan's Laws: ¬(p∧q)≡¬p∨¬q\neg(p \land q) \equiv \neg p \lor \neg q¬(p∧q)≡¬p∨¬q and ¬(p∨q)≡¬p∧¬q\neg(p \lor q) \equiv \neg p \land \neg q¬(p∨q)≡¬p∧¬q.

Double Negation: ¬(¬p)≡p\neg(\neg p) \equiv p¬(¬p)≡p.

Implication Equivalence: p→q≡¬p∨qp \rightarrow q \equiv \neg p \lor qp→q≡¬p∨q.

Idempotent Laws: p∧p≡pp \land p \equiv pp∧p≡p and p∨p≡pp \lor p \equiv pp∨p≡p, etc.

4. **Truth Tables**:

A **truth table** is a systematic way to evaluate the truth values of compound propositions for all possible truth values of their component propositions.

It shows how the truth values of compound propositions depend on the truth values of their components and the logical connectives used.

Applications and Propotional Calculus

Propositional calculus, also known as propositional logic or sentential logic, is a formal system in logic that deals with propositions (statements that can be either true or false) and their logical relationships. It forms the basis for reasoning about compound propositions using logical connectives. Let's delve into the key aspects of propositional calculus:

Limitations:

Propositional calculus is limited in expressing complex relationships involving variables and quantifiers (such as "for all" and "there exists"), which are addressed in predicate logic.

It does not deal with the internal structure or semantics of propositions beyond their truth values.

In conclusion, propositional calculus provides a robust framework for understanding and manipulating logical statements using simple connectives and truth values. It serves as a building block for more complex logical systems and finds widespread applications in various fields of mathematics, computer science, and formal reasoning

UNIT-3

Mathematical Logic

Statements and Notations:

A proposition or statement is a declarative sentence that is either true or false (but not both). For instance, the following are propositions: “Paris is in France” (true), “London is in Denmark” (false), “2 < 4” (true), “4 = 7 (false)”. However the following are not propositions: “what is your name?” (this is a question), “do your homework” (this is a command), “this sentence is false” (neither true nor false), “x is an even number” (it depends on what x represents),

“Socrates” (it is not even a sentence). The truth or falsehood of a proposition is called its truth value.

Connectives:

Connectives are used for making compound propositions. The main ones are the following (p and q represent given propositions):

Name	Represented	Meaning
Negation	$\neg p$	“not p”
Conjunction	$p \wedge$ q	“p and q”
Disjunction	$p \vee$ q	“p or q (or both)”
Exclusive Or	$p \oplus q$	“either p or q, but not both”
Implication	$p \rightarrow q$	“if p then q”
Biconditional	$p \leftrightarrow q$	“p if and only if q”

Truth Tables:

Logical identity

Logical identity is an operation on one logical value, typically the value of a proposition that produces a value of *true* if its operand is true and a value of *false* if its operand is false.

The truth table for the logical identity operator is as follows:

Logical Identity	
p	***P***
T	T
F	F

Logical Negation

Logical negation is an operation on one logical value, typically the value of a proposition that produces a value of *true* if its operand is false and a value of *false* if its operand is true.

The truth table for **NOT p** (also written as **¬p** or **~p**) is as follows:

Logical Negation	
p	**¬*p***
T	F
F	T

Binary Operations

Truth table for all binary logical operators

Here is a truth table giving definitions of all 16 of the possible truth functions of 2 binary variables (P,Q are thus boolean variables):

P	***Q***		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
T	**T**		F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T
T	**F**		F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
F	**T**		F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T
F	**F**		F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T

where T = true and F = false.

Key:

0, false, Contradiction1, NOR,

Logical NOR

2, Converse nonimplication

3, **¬p**, Negation

4, Material nonimplication

5, **¬q**, Negation

6, XOR, Exclusive disjunction7,

NAND, Logical NAND

8, AND, Logical conjunction

9, XNOR, If and only if, Logical biconditional

10, **q**, Projection function

11, if/then, Logical implication

12, **p**, Projection function
13, then/if, Converse implication14, OR,

Logical disjunction

15, true, Tautology

Logical operators can also be visualized using Venn diagrams.

Logical Conjunction

Logical conjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if both of its operands are true.
The truth table for p AND q (also written as p A q, p & q, or p q) is as follows:

Logical Conjunction		
p	***q***	*p* A *q*
T	T	T
T	F	F
F	T	F
F	F	F

In ordinary language terms, if both *p* and *q* are true, then the conjunction *p* A *q* is true. For all other assignments of logical values to *p* and to *q* the conjunction *p* A *q* is false.
It can also be said that if *p*, then *p* A *q* is *q*, otherwise *p* A *q* is *p*.

Logical Disjunction

Logical disjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if at least one of its operands is true.
The truth table for p OR q (also written as p ∨ q, p || q, or p + q) is as follows:

Logical Disjunction		
p	***q***	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Logical Implication

Logical implication and the material conditional are both associated with an operation on two logical values, typically the values of two propositions, that produces a value of *false* just in the singular case the first operand is true and the second operand is false.The truth table associated with the material conditional **if p then q** (symbolized as **p → q**) and the logical implication **p implies q** (symbolized as **p ⇒ q**) is as follows:

Logical Implication		
p	***q***	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Logical Equality

Logical equality (also known as biconditional) is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if both operands are false or both operands are true. The truth table for **p XNOR q** (also written as **p ↔ q** ,**p = q**, or **p ≡ q**) is as follows:

Logical Equality		
p	***q***	***p ≡ q***
T	T	T
T	F	F
F	T	F
F	F	T

Exclusive Disjunction

Exclusive disjunction is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if one but not both of its operands is true.The truth table for **p XOR q** (also written as **p ⊕ q**, or **p ≠ q**) is as follows:

Exclusive Disjunction		
p	***q***	***p ⊕ q***
T	T	F
T	F	T
F	T	T
F	F	F

Logical NAND

The logical NAND is an operation on two logical values, typically the values of two propositions, that produces a value of *false* if both of its operands are true. In other words, it produces a value of *true* if at least one of its operands is false.The truth table for **p NAND q** (alsowritten as **p ↑ q** or **p | q**) is as follows:

Logical NAND		
p	***q***	***p* ↑ *q***
T	T	F
T	F	T
F	T	T
F	F	T

It is frequently useful to express a logical operation as a compound operation, that is, as an operation that is built up or composed from other operations. Many such compositions are possible, depending on the operations that are taken as basic or "primitive" and the operations that are taken as composite or "derivative".In the case of logical NAND, it is clearly expressible as a compound of NOT and AND.The negation of a conjunction: $\neg(p \land q)$, and the disjunction of negations: $(\neg p) \lor (\neg q)$ can be tabulated as follows:

p	***q***	***p* A *q***	**¬(*p* A *q*)**	**¬*p***	**¬*q***	**(¬*p*) ∨ (¬*q*)**
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Logical NOR

The logical NOR is an operation on two logical values, typically the values of two propositions, that produces a value of *true* if both of its operands are false. In other words, it produces a value of *false* if at least one of its operands is true. ↓ is also known as the Peirce arrow after its inventor, Charles Sanders Peirce, and is a Sole sufficient operator.

The truth table for **p NOR q** (also written as **p ↓ q** or **p ⊥ q**) is as follows:

Logical NOR		
p	***q***	***p* ↓ *q***
T	T	F
T	F	F
F	T	F
F	F	T

The negation of a disjunction ¬(*p* ∨ *q*), and the conjunction of negations (¬*p*) A (¬*q*) can betabulated as follows:

p	***q***	***p*** ∨ ***q***	¬(***p*** ∨ ***q***)	¬***p***	¬***q***	(¬***p***) A (¬***q***)
T	T	T	F	F	F	F
T	F	T	F	F	T	F
F	T	T	F	T	F	F
F	F	F	T	T	T	T

Inspection of the tabular derivations for NAND and NOR, under each assignment of logical values to the functional arguments *p* and *q*, produces the identical patterns of functional values for ¬(*p* A *q*) as for (¬*p*) ∨ (¬*q*), and for ¬(*p* ∨ *q*) as for (¬*p*) A (¬*q*). Thus the first and second expressions in each pair are logically equivalent, and may be substituted for each other in all contexts that pertain solely to their logical values.

This equivalence is one of De Morgan's laws.

The truth value of a compound proposition depends only on the value of its components.

Writing F for "false" and T for "true", we can summarize the meaning of the connectives in the following way:

p	q	¬p	p Aq	p∨ q	p⊕ q	p → q	p ↔ q
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

Note that ∨ represents a non-exclusive or, i.e., p ∨ q is true when any ofp, q is true and also when both are true. On the other hand ⊕ represents an exclusive or, i.e., p ⊕ q is true only when exactly one of p and q is true.

Well Formed Formulas(Wff):

Not all strings can represent propositions of the predicate logic. Those which produce a proposition when their symbols are interpreted must follow the rules given below, and they are called wffs(well-formed formulas) of the first order predicate logic.

Rules for constructing Wffs

A predicate name followed by a list of variables such as P(x, y), where P ispredicate name, andx and y are variables, is called an atomic formula.

A well formed formula of predicate calculus is obtained by using the following rules.

1. An atomic formula is a wff.
2. If A is a wff, then 7A is also a wff.
3. If A and B are wffs, then (A V B), (A ^ B), (A → B) and (A D B).
4. If A is a wff and x is a any variable, then (x)A and ($x)A are wffs.
5. Only those formulas obtained by using (1) to (4) are wffs.

Since we will be concerned with only wffs, we shall use the term formulas for wff. We shall follow the same conventions regarding the use of parentheses as was done in the case ofstatement formulas.

Wffs are constructed using the following rules:

True and *False* are wffs.

Each propositional constant (i.e. specific proposition), and each propositionalvariable (i.e. a variable representing propositions) are wffs.

Each atomic formula (i.e. a specific predicate with variables) is a wff.

If *A*, *B*, and *C* are wffs, then so are*A*, (*A* ∧*B*), (*A* ∨*B), (AB)*, and *(AB)*.If *x* is→a variable (re⇔presenting objects of the universe of discourse), and *A* is a wff, thenso are ∀*x A* and ∃*x A* .

For example, "The capital of Virginia is Richmond." is a specific proposition. Hence it is a wff by Rule 2. Let B be a predicate name representing "being blue" and let x be a variable. Then B(x) is an atomic formula meaning "x is blue". Thus it is a wff by Rule 3. above. By applying Rule 5. to B(x), ∀xB(x) is a wff and so is ∃xB(x). Then by applying Rule 4. to them ∀x B(x) ∧∃x B(x) is seen to be a wff. Similarly if R is a predicate name representing "being round". Then R(x) is an atomic formula. Hence it is a wff. By applying Rule 4 to B(x) and R(x), a wff B(x) ∧R(x) is obtained.

In this manner, larger and more complex wffs can be constructed following the rules given above.

Note, however, that strings that can not be constructed by using those rules are not wffs. For example, ∀xB(x)R(x), and B(∃x) are NOT wffs, NOR are B(R(x)), and B(∃x R(x)) . More examples: To express the fact that Tom is taller than John, we can use the atomic formula ***taller*****(Tom, John)**, which is a wff. This wff can also be part of some compound statement such as ***taller*****(Tom, John)** ∧¬***taller*****(John, Tom)**, which is also a wff. ***If x is a variable representing people in the world, then taller(x,Tom), ∀x taller(x,*****Tom), ∃*****x taller(x*****,Tom), ∃*****x ∀y taller(x,y)*** are all wffs among others. However, ∃ *x*,John) and *taller*(Tom ∧Mary, Jim), for example, are **NOT** wffs.

Tautology, Contradiction, Contingency:

A proposition is said to be a tautology if its truth value is T for any assignment of truth values to its components. Example: The proposition p ∨ ¬p is a tautology.

A proposition is said to be a contradiction if its truth value is F for any assignment of truth values to its components. Example: The proposition p A ¬p is a contradiction.

A proposition that is neither a tautology nor a contradiction is called a contingency.

p	p ∨ ¬p	p A ¬p
T	T	F
T	T	F
F	T	F
F	T	F

Equivalence Implication:

We say that the statements *r* and *s* are logically equivalent if their truth tables are identical. For example the truth table of $\neg p \lor q$

p	q	$\neg p \vee q$
T	T	T
T	F	F
F	T	T
F	F	T

shows that $\neg p \vee q$ is equivalent to $p \to q$. It is easily shown that the statements *r* and *s* areequivalent if and only if $r \leftrightarrow s$ is a tautology.

Normal Forms:

Let A(P1, P2, P3, …, Pn) be a statement formula where P1, P2, P3, …, Pn are the atomic variables. If A has truth value T for all possible assignments of the truth values t the variables P1, P2, P3, …, Pn , then A is said to be a tautology. If A has truth value F, then A is said to be identically false or a contradiction.

Disjunctive Normal Forms

A product of the variables and their negations in a formula is called an elementary product. A sum of the variables and their negations is called an elementary sum. That is, a sum of elementary products is called a disjunctive normal form of the given formula.

Example:

$$A \quad (1)$$

$$(A \wedge B) \vee (!A \wedge C) \quad (2)$$

$$(A \wedge B \wedge !A) \vee (C \wedge !B) \vee (A \wedge !C) \quad (3)$$

$$A \wedge B \quad (4)$$

$$A \vee (B \wedge C), \quad (5)$$

Conjunctive Normal Forms

A formula which is equivalent to a given formula and which consists of a product of elementary sums is called a conjunctive normal form of a given formula.

Example:

$$A \quad (1)$$

$$(A \vee B) \wedge (!A \vee C) \quad (2)$$

$$A \vee B \quad (3)$$

$$A \wedge (B \vee C), \quad (4)$$

Predicates

Predicative Logic:

A predicate or propositional function is a statement containing variables. For instance “x + 2 = 7”, “X is American”, “x < y”, “p is a prime number” are predicates. The truth value of the predicate depends on the value assigned to its variables. For instance if we replace x with 1 in thepredicate “x + 2 = 7” we obtain “1 + 2 = 7”, which is false, but if we replace it with 5 we get “5 + 2 = 7”, which is true. We represent a predicate by a letter followed by the variables enclosed between parenthesis: P (x), Q(x, y), etc. An example for P (x) is a value of x for which P (x) is true. A counterexample is a value of x for which P (x) is false. So, 5 is an example for “x + 2 = 7”, while 1 is a counterexample. Each variable in a predicate is assumed to belong to a universe

(or domain) of discourse, for instance in the predicate “n is an odd integer” ’n’ represents an integer, so the universe of discourse of n is the set of all integers. In “X is American” we may assume that X is a human being, so in this case the universe of discourse is the set of all human beings.

Free & Bound Variables:

Let's now turn to a rather important topic: the distinction between free variable s and **bound variable**s.

Have a look at the following formula:

$$\neg(\textsc{therapist}(x) \vee \forall x(\textsc{moron}(x) \wedge \forall y\textsc{person}(y)))$$

The first occurrence of x is *free*, whereas the second and third occurrences of x are *bound*,namely by the first occurrence of the quantifier . The first and ∀second occurrences of the variable y are also bound, namely by the second occurrence of the quantifier . ∀

Informally, the concept of a *bound variable* can be explained as follows: Recall that quantifications are generally of the form:

$\forall x\varphi$

or $\exists x\varphi$

where x may be any variable. Generally, all occurences of this variable within the quantification are bound. But we have to distinguish two cases. Look at the following formula to see why:

$$\exists x(\mathrm{MAN}(x) \land (\forall x\,\mathrm{WALKS}(x)) \land \mathrm{HAPPY}(x))$$

1. x may occur within another, embedded, quantification $\forall x\psi$ or $\exists x\psi$, such as the x in $\mathrm{WALKS}(x)$ in our example. Then we say that it is bound by the quantifier of this embedded quantification (and so on, if there's another embedded quantification over x within ψ).
2. Otherwise, we say that it is bound by the top-level quantifier (like all other occurences of x in our example).

Here's a full formal simultaneous definition of *free* and *bound*:

1.Any occurrence of any variable is free in any atomic formula. 2.No occurrence of any variable is bound in any atomic formula.

3. If an occurrence of any variable is free in φ or in ψ, then that same occurrence is free in $\neg\varphi$, $(\varphi \to \psi)$, $(\varphi \lor \psi)$, and $(\varphi \land \psi)$.
4. If an occurrence of any variable is bound in φ or in ψ, then that same occurrence is bound in $\neg\varphi$, $(\varphi \to \psi)$, $(\varphi \lor \psi)$, $(\varphi \land \psi)$. Moreover, that same occurrence is bound in $\forall y\varphi$ and $\exists y\varphi$ as well, for any choice of variable y.
5. In any formula of the form $\forall y\varphi$ or $\exists y\varphi$ (where y can be any variable at all in this case) the occurrence of y that immediately follows the initial quantifier symbol is bound.
6. If an occurrence of a variable x is free in φ, then that same occurrence is free in $\forall y\varphi$ and $\exists y\varphi$, for any variable y distinct from x. On the other hand, all occurrences of x that are free in φ, are bound in $\forall x\varphi$ and in $\exists x\varphi$.

If a formula contains no occurrences of free variables we call it a ***sentence***.

Rules of Inference:

The two rules of inference are called rules P and T.

Rule P: A premise may be introduced at any point in the derivation.

Rule T: A formula S may be introduced in a derivation if s is tautologically implied by any one or more of the preceding formulas in the derivation.

Before proceeding the actual process of derivation, some important list of implications and equivalences are given in the following tables.

Implications

I1	P∧Q =>P	} Simplification
I2	PQ∧ =>Q	
I3	P=>PVQ	} Addition
I4	Q =>PVQ	
I5	7P => P→ Q	
I6	Q => P→ Q	
I7	7(P→Q) =>P	
I8	7(P → Q) => 7Q	
I9	P, Q => P ∧ Q	
I10	7P, PVQ => Q	(disjunctive syllogism)
I11	P, P→ Q => Q	(modus ponens)
I12	7Q, P → Q => 7P	(modus tollens)
I13	P → Q, Q → R => P → R	(hypothetical syllogism)
I14	P V Q, P → Q, Q → R => R	(dilemma)

Equivalences

E1	77P <=>P	
E2	P ∧ Q <=> Q ∧ P	} Commutative laws
E3	P V Q <=> Q V P	
E4	(P ∧ Q) ∧ R <=> P ∧ (Q ∧ R)	} Associative laws
E5	(P V Q) V R <=> PV (Q V R)	
E6	P ∧ (Q V R) <=> (P ∧ Q) V (P ∧ R)	} Distributive laws
E7	P V (Q ∧ R) <=> (P V Q) ∧ (PVR)	
E8	7(P ∧ Q) <=> 7P V7Q	
E9	7(P V Q) <=>7P ∧ 7Q	} De Morgan's laws

E10P V P <=> P E11 P ∧ P <=> P

E12	R V (P ^ 7P) <=>R
E13	R ^ (P V 7P) <=>R
E14	R V (P V 7P) <=>T
E15	R ^ (P ^ 7P) <=>F
E16	P → Q <=> 7P V Q
E17	7 (P→ Q) <=> P ^ 7Q
E18	P → Q <=> 7Q → 7P
E19	P → (Q → R) <=> (P ^ Q) → R
E20	7(PD Q) <=> P D 7Q
E21	PDQ <=> (P → Q) ^ (Q → P)
E22	(PDQ) <=> (P ^ Q) V (7 P ^ 7Q)

Example 1.Show that R is logically derived from P → Q, Q → R, and P

Solution.	{1}	(1) P → Q	Rule P
	{2}	(2) P	Rule P
	{1, 2}	(3) Q	Rule (1), (2) and I11
	{4}	(4) Q → R	Rule P
	{1, 2, 4}	(5) R	Rule (3), (4) and I11.

Example 2.Show that S V R tautologically implied by (P V Q) ^ (P → R) ^ (Q → S).

Solution .	{1}	(1) P V Q	Rule P	
	{1}	(2) 7P → Q		T, (1), E1 and E16
	{3}	(3) Q → S	P	
	{1, 3}	(4) 7P → S		T, (2), (3), and I13
	{1, 3}	(5) 7S → P		T, (4), E13 and E1
	{6}	(6) P → R	P	
	{1, 3, 6}	(7) 7S → R		T, (5), (6), and I13
	{1, 3, 6)	(8) S V R		T, (7), E16 and E1

Example 3. Show that 7Q, P→ Q => 7P

Solution .	{1}	(1) P → Q	Rule P
	{1}	(2) 7P → 7Q	T, and E 18

{3}	(3) 7Q	P
{1, 3}	(4) 7P	T, (2), (3), and I11 .

Example 4 .Prove that R ˄ (P V Q) is a valid conclusion from the premises PVQ ,
Q → R, P → M and 7M.

Solution . {1}	(1) P → M	P
{2}	(2) 7M	P
{1, 2}	(3) 7P	T, (1), (2), and I12
{4}	(4) P V Q	P
{1, 2 , 4}	(5) Q	T, (3), (4), and I10.
{6}	(6) Q → R	P
{1, 2, 4, 6}	(7) R	T, (5), (6) and I11
{1, 2, 4, 6}	(8) R ˄ (PVQ)	T, (4), (7), and I9.

There is a third inference rule, known as rule CP or rule of *conditional proof*.

Rule CP: If we can derives s from R and a set of premises , then we can derive R → S from the set of premises alone.

Note. 1. Rule CP follows from the equivalence E10 which
states that(P ˄ R) → S óP → (R → S).

2. Let P denote the conjunction of the set of premises and let R be any formula Theabove equivalence states that if R is included as an additional premise and S is derived from P ˄ R then R → S can be derived from the premises P alone.

3. Rule CP is also called the *deduction theorem* and is generally used if theconclusion is of the form R → S. In such cases, R is taken as an additional premise and S is derived from the given premises and R.

Example 5 .Show that R → S can be derived from the premises
P → (Q → S), 7R V P , and Q.

Solution.	{1}	(1) 7R V P	P
	{2}	(2) R	P, assumed premise
	{1, 2}	(3) P	T, (1), (2), and I10
	{4}	(4) P → (Q → S)	P
	{1, 2, 4}	(5) Q → S	T, (3), (4), and I11
	{6}	(6) Q	P
	{1, 2, 4, 6}	(7) S	T, (5), (6), and I11
	{1, 4, 6}	(8) R → S	CP.

Example 6.Show that P → S can be derived from the premises, 7P V Q, 7Q V R, and R → S .

Solution.

{1}	(1) 7P V Q	P
{2}	(2) P	P, assumed premise
{1, 2}	(3) Q	T, (1), (2) and I11
{4}	(4) 7Q V R	P
{1, 2, 4}	(5) R	T, (3), (4) and I11
{6}	(6) R → S	P
{1, 2, 4, 6}	(7) S	T, (5), (6) and I11
{2, 7}	(8) P → S	CP

Example 7. ” If there was a ball game , then traveling was difficult. If they arrived on time, then traveling was not difficult. They arrived on time. Therefore, there was no ball game”. Show that these statements constitute a valid argument.

Solution. Let P: There was a ball game
Q: Traveling was difficult.R: They arrived on time.

Given premises are: P → Q, R → 7Q and R conclusion is: 7P

{1}	(1) P → Q	P
{2}	(2) R → 7Q	P
{3}	(3) R	P
{2, 3}	(4) 7Q	T, (2), (3), and I11
{1, 2, 3}	(5) 7P	T, (2), (4) and I12

Consistency of premises:

Consistency

A set of formulas H1, H2, …, Hm is said to be consistent if their conjunction has the truthvalue T for some assignment of the truth values to be atomic appearing in H1, H2, …, Hm.

Inconsistency

If for every assignment of the truth values to the atomic variables, at least one of the formulasH1, H2, … Hm is false, so that their conjunction is identically false, then the formulas
H1, H2, …, Hm are called inconsistent.

A set of formulas H1, H2, …, Hm is inconsistent, if their conjunction implies a contradiction, that is H1∧ H2∧ … ∧ Hm => R ∧ 7R

Where R is any formula. Note that R ∧ 7R is a contradiction and it is necessary and sufficient that H1, H2, …,Hm are inconsistent the formula.

Indirect method of proof

In order to show that a conclusion C follows logically from the premises H1, H2,…, Hm, we assume that C is false and consider 7C as an additional premise. If the new set of premises is inconsistent, so that they imply a contradiction, then the assumption that 7C is true does not hold simultaneously with H1∧ H2∧ ..… ∧ Hm being true. Therefore, C is true whenever H1∧ H2∧ ..… ∧ Hm is true. Thus, C follows logically from the premises H1, H2 ….., Hm.

Example 8 Show that 7(P ∧ Q) follows from 7P∧ 7Q.

Solution.

We introduce 77 (P∧ Q) as an additional premise and show that this additional premise leads to a contradiction.

{1}	(1) 77(P∧ Q)	P assumed premise
{1}	(2) P∧ Q	T, (1) and E1
{1}	(3) P	T, (2) and I1
{1}	{4) 7P∧7Q	P
{4}	(5) 7P	T, (4) and I1
{1, 4}	(6) P∧ 7P	T, (3), (5) and I9

Here (6) P∧ 7P is a contradiction. Thus {1, 4} viz. 77(P∧ Q) and 7P∧ 7Q leadsto a contradiction P ∧ 7P.

Example 9Show that the following premises are inconsistent.

1. If Jack misses many classes through illness, then he fails high school.
2. If Jack fails high school, then he is uneducated.
3. If Jack reads a lot of books, then he is not uneducated.
4. Jack misses many classes through illness and reads a lot of books.

Solution.

P: Jack misses many classes.Q:
Jack fails high school.
R: Jack reads a lot of books.S:
Jack is uneducated.

The premises are P→ Q, Q → S, R→ 7S and P∧ R

{1}	(1) P→Q	P
{2}	(2) Q→ S	P
{1, 2}	(3) P → S	T, (1), (2) and I13
{4}	(4) R→ 7S	P
{4}	(5) S → 7R	T, (4), and E18
{1, 2, 4}	(6) P→7R	T, (3), (5) and I13
{1, 2, 4}	(7) 7PV7R	T, (6) and E16
{1, 2, 4}	(8) 7(P∧R)	T, (7) and E8

{9}	(9)P∧ R	P
{1, 2, 4, 9)	(10) (P∧ R) ∧ 7(P∧ R)	T, (8), (9) and I9

The rules above can be summed up in the following table. The "Tautology" column shows how to interpret the notation of a given rule.

Rule of inference	**Tautology**	**Name**
$\begin{array}{r} p \\ \hline \therefore p \vee q \end{array}$	$p \rightarrow (p \vee q)$	Addition
$\begin{array}{r} p \wedge q \\ \hline \therefore p \end{array}$	$(p \wedge q) \rightarrow p$	Simplification
$\begin{array}{r} p \\ q \\ \hline \therefore p \wedge q \end{array}$	$((p) \wedge (q)) \rightarrow (p \wedge q)$	Conjunction
$\begin{array}{r} p \\ p \rightarrow q \\ \hline \therefore q \end{array}$	$((p \wedge (p \rightarrow q)) \rightarrow q$	Modus ponens
$\begin{array}{r} \neg q \\ p \rightarrow q \\ \hline \therefore \neg p \end{array}$	$((\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$	Modus tollens
$\begin{array}{r} p \rightarrow q \\ q \rightarrow r \\ \hline \therefore p \rightarrow r \end{array}$	$((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$	Hypothetical syllogism
$\begin{array}{r} p \vee q \\ \neg p \\ \hline \therefore q \end{array}$	$((p \vee q) \wedge \neg p) \rightarrow q$	Disjunctive syllogism
$\begin{array}{r} p \vee q \\ \neg p \vee r \\ \hline \therefore q \vee r \end{array}$	$((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$	Resolution

Example 1

Let us consider the following assumptions: "If it rains today, then we will not go on a canoe today. If we do not go on a canoe trip today, then we will go on a canoe trip tomorrow. Therefore (Mathematical symbol for "therefore" is $\therefore$), if it rains today, we will go on a canoe trip tomorrow. To make use of the rules of inference in the above table we let p be the proposition "If it rains today", q be " We will not go on a canoe today" and let r be "We will go on a canoe trip tomorrow". Then this argument is of the form:

$$\begin{array}{r} p \rightarrow q \\ q \rightarrow r \\ \hline \therefore p \rightarrow r \end{array}$$

Example 2

Let us consider a more complex set of assumptions: "It is not sunny today and it is colder than yesterday". "We will go swimming only if it is sunny", "If we do not go swimming, then we will have a barbecue", and "If we will have a barbecue, then we will be home by sunset" lead to the conclusion "We will be home before sunset." Proof by rules of inference: Let p be the proposition "It is sunny this today", q the proposition "It is colder than yesterday", r the proposition "We will go swimming", s the proposition "We will have a barbecue", and t the proposition "We will be home by sunset". Then the hypotheses become $\neg p \wedge q, r \rightarrow p, \neg r \rightarrow s$ and $s \rightarrow t$. Using our intuition we conjecture that the conclusion might be t. Using the Rules of Inference table we can proof the conjecture easily:

Step	**Reason**
1. $\neg p \wedge q$	Hypothesis
2. $\neg p$	Simplification using Step 1

3. $r \rightarrow p$	Hypothesis
4. $\neg r$	Modus tollens using Step 2 and 3
5. $\neg r \rightarrow s$	Hypothesis
6. s	Modus ponens using Step 4 and 5
7. $s \rightarrow t$	Hypothesis
8. t	Modus ponens using Step 6 and 7

Proof of contradiction:

The "Proof by Contradiction" is also known as reductio ad absurdum, which is probably Latin for "reduce it to something absurd".

Here's the idea:

1. Assume that a given proposition is untrue.
2. Based on that assumption reach two conclusions that contradict each other.

This is based on a classical formal logic construction known as Modus Tollens: If P implies Q and Q is false, then P is false. In this case, Q is a proposition of the form (R and not R) which is always false. P is the negation of the fact that we are trying to prove and if the negation is not true then the original proposition must have been true. If computers are not "not stupid" then they are stupid. (I hear that "stupid computer!" phrase a lot around here.)

Example:

Lets prove that there is no largest prime number (this is the idea of Euclid's original proof).
Prime numbers are integers with no exact integer divisors except 1 and themselves.

To prove: "There is no largest prime number" by contradiction.

Assume: There is a largest prime number, call it p.

Consider the number N that is one larger than the product of all of the primes smallerthan or equal to p. N=1*2*3*5*7*11...*p + 1. Is it prime?

N is at least as big as p+1 and so is larger than p and so, by Step 2, cannot be prime.

On the other hand, N has no prime factors between 1 and p because they would all leave a remainder of 1. It has no prime factors larger than p because Step 2 says that there are no primes larger than p. So N has no prime factors and therefore must itself be prime(see note below).

We have reached a contradiction (N is not prime by Step 4, and N is prime by Step 5) and therefore our original assumption that there is a largest prime must be false.

Note: The conclusion in Step 5 makes implicit use of one other important theorem: The Fundamental Theorem of Arithmetic: Every integer can be uniquely represented as the product of primes. So if N had a composite (i.e. non-prime) factor, that factor would itself have prime factors which would also be factors of N.

Automatic Theorem Proving:

Automatic Theorem Proving (ATP) deals with the development of computer programs that show that some statement (the *conjecture*) is a *logical consequence* of a set of statements (the *axioms* and *hypotheses*). ATP systems are used in a wide variety of domains. For examples, a mathematician might prove the conjecture that groups of order two are commutative, from the axioms of group theory; a management consultant might formulate axioms that describe how organizations grow and interact, and from those axioms prove that organizational death rates decrease with age; a hardware developer might validate the design of a circuit by proving a conjecture that describes a circuit's performance, given axioms that describe the circuit itself; ora frustrated teenager might formulate the jumbled faces of a Rubik's cube as a conjecture and prove, from axioms that describe legal changes to the cube's configuration, that the cube can be rearranged to the solution state. All of these are tasks that can be performed by an ATP system, given an appropriate formulation of the problem as axioms, hypotheses, and a conjecture.

The **language** in which the conjecture, hypotheses, and axioms (generically known as *formulae*) are written is a logic, often classical 1st order logic, but possibly a non-classical logic and possibly a higher order logic. These languages allow a precise formal statement of the necessary information, which can then be manipulated by an ATP system. This formality is the underlying strength of ATP: there is no ambiguity in the statement of the problem, as is often the case when using a natural language such as English. Users have to describe the problem at hand precisely and accurately, and this process in itself can lead to a clearer understanding of the problem domain. This in turn allows the user to formulate their problem appropriately for submission to an ATP system.

The **proofs** produced by ATP systems describe how and why the conjecture follows from the axioms and hypotheses, in a manner that can be understood and agreed upon by everyone, even other computer programs. The proof output may not only be a convincing argument that the conjecture is a logical consequence of the axioms and hypotheses, it often also describes a process that may be implemented to solve some problem. For example, in the Rubik's cube example mentioned above, the proof would describe the sequence of moves that need to be madein order to solve the puzzle.

ATP systems are enormously powerful computer programs, capable of solving immensely difficult problems. Because of this extreme capability, their application and operation sometimes needs to be guided by an expert in the domain of application, in order to solve problems in a reasonable amount of time. Thus ATP systems, despite the name, are often used by domain experts in an interactive way. The interaction may be at a very detailed level, where the user guides the inferences made by the system, or at a much higher level where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture. There is often a synergetic relationship between ATP system users and the systems themselves:

The system needs a precise description of the problem written in some logical form,

the user is forced to think carefully about the problem in order to produce an appropriate formulation and hence acquires a deeper understanding of the problem,

the system attempts to solve the problem,

if successful the proof is a useful output,

if unsuccessful the user can provide guidance, or try to prove some intermediate result, orexamine the formulae to ensure that the problem is correctly described,

and so the process iterates.

ATP is thus a **technology** very suited to situations where a clear thinking domain expert can interact with a powerful tool, to solve interesting and deep problems. Potential ATP users need not be concerned that they need to write an ATP system themselves; there are many ATP systems readily available for use.

Contrapositive Proofs:

A proof by contrapositive is another fundamental technique in mathematics and logic closely related to proof by contradiction. It is used to prove a statement \(P \rightarrow Q \) by instead proving its contrapositive statement \(\neg Q \rightarrow \neg P \). Here's how it works:

Steps to Prove by Contrapositive:

Identify the Statement: Start with the statement \(P \rightarrow Q \),
which asserts that if \(P \) is true, then \(Q \) must also be true.

Formulate the Contrapositive: The contrapositive of \(P \rightarrow Q \) is \(\neg Q \rightarrow \neg P \).
This means if \(Q \) is false, then \(P \) must also be false.

Prove the Contrapositive: Instead of proving \(P \rightarrow Q \) directly, prove \(\neg Q \rightarrow \neg P \) using logical deduction and possibly other established truths (axioms, definitions, previously proven theorems).

. **Conclude the Original Statement**: Since \(\neg Q \rightarrow \neg P \) is logically equivalent to \(P \rightarrow Q \), establishing the truth of the contrapositive proves the original statement.

Example:
Let's prove the statement: "If \(n \) is an integer and \(n^2 \) is even, then \(n \) is even" using the contrapositive.

Statement to prove: \(P \rightarrow Q \): If \(n \) is an integer and \(n^2 \) is even, then \(n \) is even.
*Contrapositive**: \(\neg Q \rightarrow \neg P \): If \(n \) is an integer and \(n \) is not even, then \(n^2 \) is not even.

Proof by Contrapositive:

1. Assume \(n \) is an integer and \(n \) is not even (i.e., \(n \) is odd).

2. \(n \) can be expressed as \(n = 2k + 1 \) for some integer \(k \).

3. Square \(n \): \(n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 \).

4. \(n^2 \) can be rewritten as \(n^2 = 2(2k^2 + 2k) + 1 \), which is odd.

5. Therefore, \(n^2 \) is odd, not even.

Thus, we have proven that if \(n \) is not even, then \(n^2 \) is not even.Hence, by contrapositive, if \(n^2 \) is even, then \(n \) must be even.
Proofs by contrapositive are often used when directly proving \(P \rightarrow Q \) might be challenging, but proving \(\neg Q \rightarrow \neg P \) is more straightforward or insightful. It leverages the logical equivalence between a statement and its contrapositive to establish the truth of the original implication.

Proofs of necessity and sufficiency:

Proofs of necessity and sufficiency are used to establish the conditions under which a statement \(P \) implies another statement \(Q \). There are two main types of proofs in this context:

1. **Proof of Sufficiency**: This type of proof shows that if \(P \) is true, then \(Q \) must also be true. It establishes that \(P \) is sufficient for \(Q \). Symbolically, this is written as \(P \Rightarrow Q \).

2. **Proof of Necessity**: This type of proof shows that if \(Q \) is true, then \(P \) must also be true. It establishes that \(P \) is necessary for \(Q \). Symbolically, this is written as \(Q \Rightarrow P \).

Example:

Let's illustrate both types of proofs with an example:

Statement to prove: For real numbers \(x \) and \(y \), \(x = y^2 \) if and only if \(x \geq 0 \).

Proof of Sufficiency (\(x = y^2 \Rightarrow x \geq 0 \)):

- Assume \(x = y^2 \).
- Since \(y^2 \geq 0 \) for all real numbers \(y \), it follows that \(x = y^2 \geq 0 \).
- Therefore, \(x \geq 0 \) whenever \(x = y^2 \).
- This proves that \(x = y^2 \) is sufficient to ensure \(x \geq 0 \).

Proof of Necessity (\(x \geq 0 \Rightarrow x = y^2 \)):

- Assume \(x \geq 0 \).
- If \(x = 0 \), then \(x = 0 = 0^2 \), so \(x = y^2 \) where \(y = 0 \).
- If \(x > 0 \), then \(\sqrt{x} \) and \(-\sqrt{x} \) are both square roots of \(x \). Hence, \(x = (\sqrt{x})^2 \) or \(x = (-\sqrt{x})^2 \).
- Therefore, \(x \geq 0 \) implies \(x = y^2 \).

These proofs establish the bidirectional implication (if and only if) between \(x = y^2 \) and \(x \geq 0 \), demonstrating both sufficiency and necessity.

Key Points:

- **Sufficiency**: \(P \) is sufficient for \(Q \) means that \(P \Rightarrow Q \). If \(P \) is true, then \(Q \) must be true.

- **Necessity**: \(P \) is necessary for \(Q \) means that \(Q \Rightarrow P \). If \(Q \) is true, then \(P \) must be true.

- **Bidirectional Implication**: To prove \(P \iff Q \) (if and only if), you need to prove both \(P \Rightarrow Q \) and \(Q \Rightarrow P \).

These proofs are fundamental in establishing the logical relationships between statements and are widely used across mathematics, logic, and other fields where precise reasoning is required.

UNIT-IV

Graph Theory

There are two different equential representations of a graph. They are Adjacency Matrix representation

•PathMatrixrepresentation

Adjacency Matrix Representation

Suppose G is a simple directed graph with m nodes, and suppose the nodes of G havebeen ordered and are called v1, v2, . . . , vm. Then the adjacencymatrix A= (aij) of the graph G is the m x m matrix defined as follows:

1ifviisadjacenttoVj,thatis,ifthereisanedge(Vi,Vj) aij =0 otherwise

Suppose Gisanundirectedgraph.Thentheadjacencymatrix AofGwillbea symmetric matrix, i.e., one in which aij = aji; for every i and j.

Drawbacks

Itmaybedifficult to insert and deletenodes inG.Ifthenumberofedgesis0(m)or0(mlog2m),thenthematrix Awillbesparse, hence a great deal of space will be wasted.

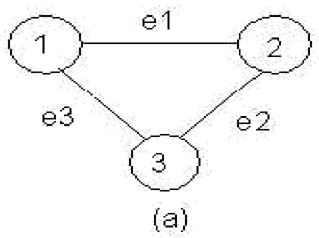
Path Matrix Representation

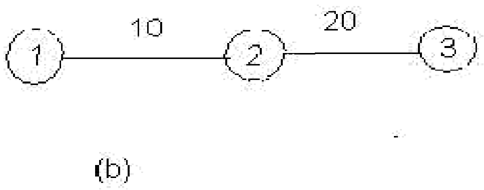
LetGbeasimpledirectedgraphwithmnodes, v1,v2,...,vm.Thepathmatrix ofGis the m-square matrix P = (pij) defined as follows:

1ifthereisapathfromVitoVj Pij =0 otherwise

GraphsandMultigraphs

Agraph G consistsoftwo things:





weighted or Labeled Graph

1.Aset Vofelements callednodes (orpoints or vertices)

2.Aset Eof edgessuch thateachedge einE isidentified witha unique

(unordered) pair [u, v of nodes in V, denoted by e = [u, v] SometimesweindicatethepartsofagraphbywritingG=(V,E).

Suppose e = [u, v]. Then the nodes u and v arecalled the endpoints of e, and u and v aresaid to be adjacent nodes or neighbors. The degree of a node u, written deg(u), is the number of edgescontainingu. Ifdeg(u)=0–that is,if u does not belong to any edge–the n u is called an isolated node.

Path and Cycle

Apath P of length n from n odeu to a no dev is defined a a sequence of n +1 nodes.

P=(v0, v1, v2, . . . , vn) such that u =v 0; vi-1 is adjacent to vi for i = 1,2, . . ., n and vn =v.

Types of Path

1. Simple Path
2. Cycle Path

(i) Simple Path

Simple path is a path in which first and last vertex are different (V0≠Vn)

(ii) CyclePath

Cycle path is a path in which first and last vertex are same (V0=Vn) .It is also called as Closed path.

Connected Graph

A graph G is said to be connected if there is a path between any two of its nodes.

Complete Graph

Agraph G is said to be complete if every node u in G is adjacent to every other no de v in G.

Tree

A connected graph T without any cycles is called a tree graph or free tree or,simply, a tree.

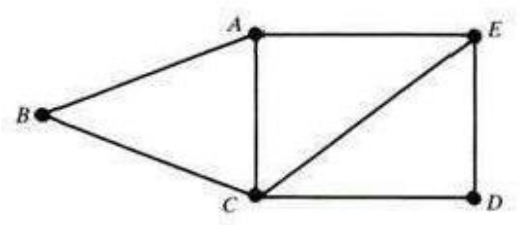
Labeledor Weighted Graph

If the weight is assigned to each edge of the graph the n it is called as Weighted or Labeled graph.

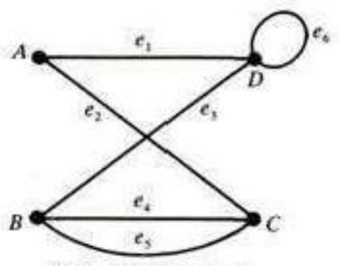
The definition of a graph may be generalized by permitting the following:

MultipleEdges: Distinct edges e and e' are called multiple edges if they connect the same end points, that is, if e = [u, v] and e' = [u, v].

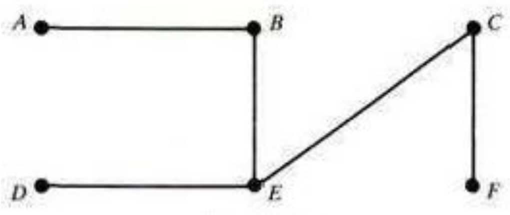
Loops* :** An edge e is called a loop if it has identical end points , that is, if e=[u, u]. □ ***FiniteGraph : A multigraph M is said to be finite if it has finite number of node s and a finite



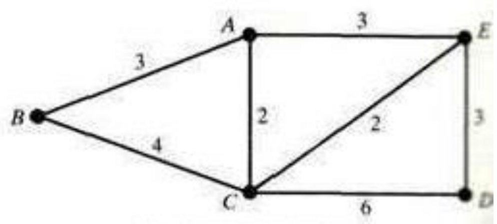
(*a*) Graph.



(*b*) Multigraph.



(*c*) Tree.



(*d*) Weighted graph.

number of edges.

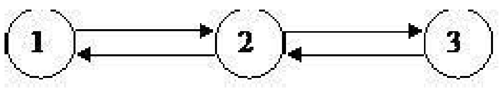
Directed Graphs

A directed graph G, also called a digraph or graph is the same as a multigraph except that each edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u, v) of nodes in G.

Outdegree and Indegree

Indegree:The indegree of a vertex is the number of edges for which v is head

Example

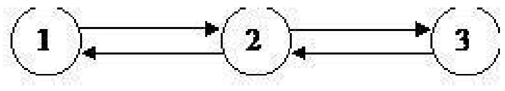


Indegree of 1=1

Indegree of 2=2

Outdegree : The outdegree of an ode or vertex is the number of edges for which v is tail.

Example



Outdegree of 1 =1

Outdegree of 2 =2

Simple Directed Graph

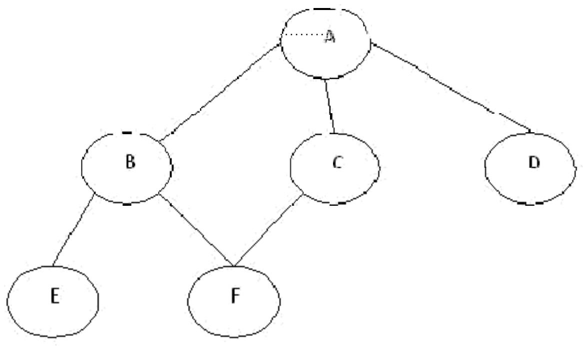
A directed graph G is said to be simple if G has no parallel edges.A simple graph G may have loops, but it cannot have more than one loop at a given node.

Graph Traversal

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Depth FirstSearch (DFS)

The aim of DFS algorithm is to traverse the graph in such a way the atittries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

Step1:Push the root node in the Stack.

Step2:Loop until stack is empty.

Step3:Peek the node of the stack.

Step4:If the node has unvisited childnodes ,get the unvisited childnode , mark it as traversed and push it on stack.

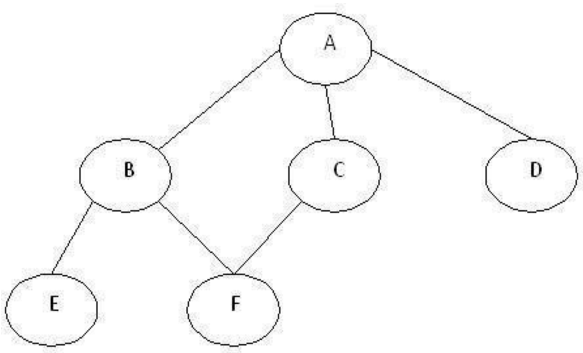
Step5:If the node does not have any unvisited childnodes , pop the node from the stack.

Based upon the above steps, the following Java codes hows the implementation of the DFS algorithm:

```
Public void dfs()

{//DFSusesStackdatastructure

 Stack s=new Stack();

 s. push(this. Root Node);

root Node .visited=true;

printNode(rootNode);
while(!s.isEmpty()){Noden=(Node)s.peek();Nodechild=getUnvisitedChildNode(n); if(child!=null)

{child .visited=true; print Node(child); s. push(child);

} else{



}

}
s.pop();

//Clear visited property of nodes clear Nodes();

}
```

Breadth First Search(BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "ABC DEF". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step1**:Push the rootnode in the Queue.
2. **Step2**:Loop until the queue is empty.
3. **Step3**: Remove the node from the Queue.
4. **Step4**:If there moved node has unvisited childnodes, mark the m as visited and insert the unvisited children in the queue.

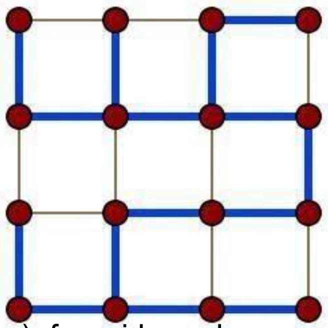
Based upon the above steps,the following Java code shows the implementation of the BFS algorithm:

```
Public void bfs() {//BFS uses Queue datastructure
Queue q=new Linked List();

q .add(this.  root  Node);

print Node(this. rootNode);

rootNode. visited=true;

while(!q. isEmpty()) {Noden=(Node)q.remove();

 Node child=null;

while((child=getUnvisitedChildNode(n))!=null){child.visited=true;

print Node(child);

 q. add(child);}}                        //Clearvisitedpropertyof nodes clearNodes();}
```

Spanning Trees: In the mathematical field of graph theory, a **spanning tree** *T* of a connected, undirected graph *G* is a tree composed of all the vertices and some (or perhaps all) of the edges of *G*. Informally, a spanning tree of *G* is a selection of edges of *G* that form a tree *spanning* every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed. On the other hand, every bridge of *G* must belong to *T*.

A spanning tree of a connected graph *G* can also be defined as a maximal set of edges of *G* that contains no cycle, or as a minimal set of edges that connect all vertices.

Example:



A spanning tree (blue heavy edges)of a grid graph.

Spanning Forests

A **spanning forest** is a type of sub graph that generalises the concept of a spanning tree. However, there are two definitions in common use. One is that a spanning forest is a sub graph that consists of a spanning tree in each connected component of a graph. (Equivalently, it is a maximal cycle-free subgraph.) This definition is common in computer science and optimisation. It is also the definition used when discussing minimum spanning forests, the generalization to disconnected graphs of minimum spanning trees. Another definition, common in graph theory, is that a spanning forest is any sub graph that is both a forest (contains no cycles) and spanning

Counting Spanning Trees

The number *t(G)* of spanning trees of a connected graph is an important in variant. In some cases, it is easy to calculate *t(G)* directly. It is also widely used in data structures in different computer

languages. For example, if *G* is itself a tree, then *t(G)=1*, while if *G* is the cycle graph *C n* with *n* vertices, then *t(G)=n*. For any graph *G*, the number *t(G)* can be calculated using Kirchhoff's matrix-tree theorem (follow the link for an explicit example using the theorem).

Cayley's formula is a formula for the number of spanning trees in the complete graph$_n K$ with *n*

n−2vertices.The formula states that $t(K_n)$ =*nn*−2 . Another way of stating Cayley's formula is th there are exactly *n* labeled trees with *n* vertices.Cayley's formula can be prove d using Kirchhoff's matrix-tree theorem or via the Prüfer code.*q* −1*p* −1If *G* is the complete bipartite graph *Kp*,*q*, then *t*(*G*) = *p* *q*,while if *G* is the*n*-dimensional hyper cube graph *Qn*, then

$$t(G) = 2^{2^n-n-1}\prod_{k=2}^{n} k^{\binom{n}{k}}.$$

These formulae are also consequences of the matrix-tree theorem.

If *G* is a multigraph and *e* is an edge of *G*, then the number *t(G)* of spanning trees of *G* satisfies the *deletion-contraction recurrence (G)=t(G-e)+t(G/e)*,where *G-e* is the multigraph obtained by deleting *e* and *G/e* is the contraction of *G* bye, where multiple edges a rising from

This contraction are not deleted.

Uniform Spanning Trees

A spanning tree chosen randomly from among all the spanning trees with equal probability is called a uniform spanningtree (UST). This model has been extensively researched in probability and mathematical physics.

Algorithms

The classic spanning tree algorithm, depth-first search (DFS), is due to Robert Tarjan. Another important algorithm is based on breadth-first search (BFS).

Planar Graphs: In graph theory, a **planar graph** is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

A planar graph already drawn in the plane without edge intersections is called a **plane graph** or **planar embedding of the graph**. A plane graph can be defined as a planar graph with a mapping from every node to a point in 2D space, and from every edge to a plane curve, such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points. Plane graphs can be encoded by combinatorial maps.

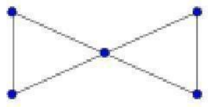
It is easily seen that a graph that can be drawn on the plane can be drawn on the sphere as well, and vice versa.

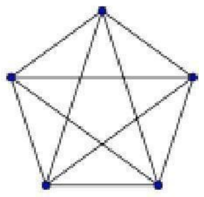
The equivalence class of topologically equivalent drawings on the sphere is called a **planar map**. Although a plane graph has an **external** or **unbounded** face, none of the faces of a planar map have a particular status.

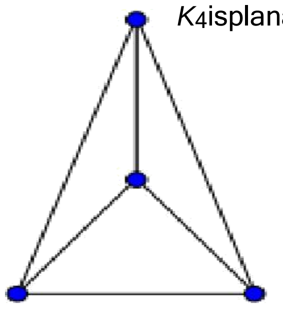
Applications

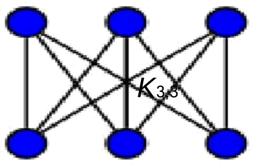
- Telecommunications-e .g. spanning trees
- Vehicle routing-e .g. planning routes on roads without under passes • VLSI – e.g. laying out circuits on computer chip.
- ThepuzzlegamePlanarityrequirestheplayerto"untangle"aplanargraphsothatnone of its edges intersect.

Example graphs Planar and Nonplanar*K*5









The complete graph

Graph Theory and Applications:

Graphs are among the most ubiquitous models of both natural and human-made structures. They can be used to model many types of relations and process dynamics in physical, biological and social systems. Many problems of practical interest can be represented by graphs.

In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc .One practical example : The link structure of a website could be represented by a directed graph. The vertices are the web pages available at the website and a directed edge from page *A* to page *B* exists if and only if *A* contains a link to *B*. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems (e.g. confluence) are studied. Complementary to graph transformation systems focussing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

Graph-theoretic methods, in various forms, have proven particularly useful in linguistics, since natural language often lends itself well to discrete structure. Traditionally, syntax and compositional semantics follow tree-based structures, whose expressive power lies in the Principle of Compositionality, modeled in a hierarchical graph. Within lexical semantics, especially as applied to computers, modeling word meaning is easier when a given word is understood in terms of related words; semantic networks are therefore important in computational linguistics.

Still other methods in phonology (e.g. Optimality Theory, which uses lattice graphs) and morphology (e.g. finite-state morphology, using finite-state transducers) are common in the analysis of language as a graph. Indeed, the usefulness of this area of mathematics to linguistics has borne organizations such as Text Graphs, as well as various 'Net' projects, such as Word Net, Verb Net, and others.

Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. For example, Franzblau's shortest-path (SP) rings. In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems.

Graph theory is also widely used in sociology as a way, for example, to measure actors' prestige or to explore diffusion mechanisms, notably through the use of social network analysis software. Likewise, graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or habitats) and the edges represent migration

paths,ormovementbetweentheregions.Thisinformationisimportantwhenlookingatbreeding patterns or tracking the spread of disease, parasites or how changes to the movement can affect other species.

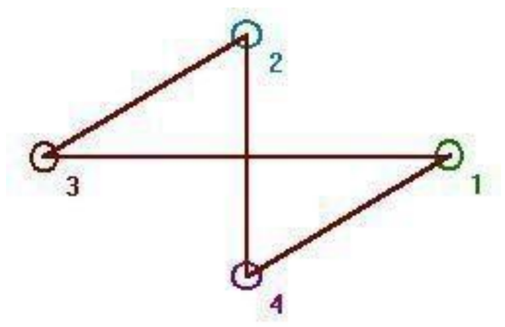
In mathematics, graphs are useful in geometry and certain parts of topology, e.g. Knot Theory. Algebraic graph theory has close links with group theory.

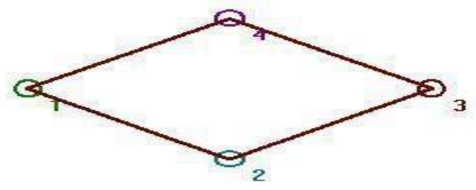
A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pair wise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road.

Basic Concepts Isomorphism:

Let G1and G1betwo graphs and let f be a function from the vertex set ofG1to the vertex set of G2. Suppose that f is one-to-one and onto & f(v) is adjacent to f(w) in G2 if and only if v is adjacent to w in G1.

Then we say that the function f is an isomorphism and that the two graphs G1 and G2 are isomorphic. So two graphs G1 and G2 are isomorphic if there is a one-to-one correspondence between vertices of G1 and those of G2 with the property that if two vertices of G1 are adjacent then so are their images in G2. If two graphs are isomorphic then as far as we are concerned they are the same graph though the location of the vertices may be different. To show you how the program can be used to explore isomorphism draw the graph in figure 4 with the program (first get the null graph on four vertices and then use the right mouse to add edges).





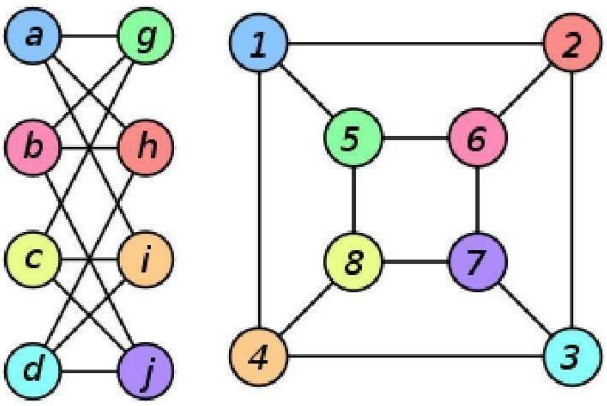
Save this graph as Graph1 (you need to click Graph then Save).Now get the circuit graph with 4 vertices. It looks like figure 5, and we shall call it C(4).

Example:

The two graphs shown below are isomorphic, despite their different looking drawings.

Graph G	**Graph H**	**An isomorphism** **Between G and H**
		$f(a)$=1 $f(b)$=6 $f(c)$=8

			$f(d)=3$
			$f(g)=5$
			$f(h)=2$
			$f(i)=4$
			$f(j)=7$



Subgraphs:

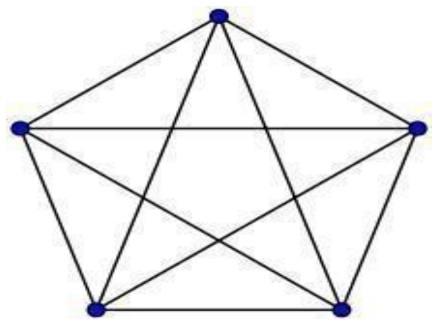
A **subgraph** of a graph *G* is a graph whose vertex set is a subset of that of *G*, and whose adjacency relation is a subset of that of *G* restricted to this subset. In the other direction, a **supergraph** of a graph *G* is a graph of which *G* is a subgraph. We say a graph *G* **contains** another graph *H* if some subgraph of *G* is *H* or is isomorphic to *H*.

Asubgraph *H* is a **spanning subgraph**, or **factor**, of a graph *G* if it has the same vertex set as *G*. We say *H* spans *G*.

A subgraph *H* of a graph *G* is said to be **induced** if, for any pair of vertices *x* and *y* of H, *x y* is an edge of *H* if and only if *x y* is an edge of G. In other words ,*H* is an induced subgraph of *G* if it has all the edges that appear in *G* over the same vertex set .If the vertex set of *H* is the subset *S* of *V(G)*, then *H* can be written as *G*[*S*] and is said to be **induced by *S***.

A graph that does *not* contain *H* as an induce d subgraph is said to be ***H*-free**.

A **universal graph** in a class ***K*** of graphs is a simple graph in which every element in ***K*** can be embedded as a subgraph.



K5, a complete graph.If as u b graph looks like this , the vertices in that subgraph form aclique of size 5.

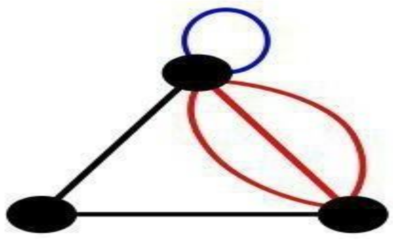
Multigraphs:

In mathematics, a **multigraph** or **pseudograph** is a graph which is permitted to have multiple edges, (also called "parallel edges"), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge. Formally, a multigraph *G* is an ordered pair *G*:=(*V*, *E*) with

V as e to•f *vertices* or *nodes*,

- *E* a multiset of unordered pairs of vertices, called *edges* or *lines*.

Multigraphs might be used to model the possible flight connections offered by an airline. In this case the multigraph would be a directed graph with pairs of directed parallel edges connecting cities to show that it is possible to fly both *to* and *from* these locations.



A multigraph with multiple edges (red) and a loop (blue). Not all authors allow multigraphs to have loops.

Euler Circuits:

In graph theory, an **Eulerian trail** is a trail in a graph which visits every edge exactly once. Similarly,an **Eulerian circuit** is an Eulerian trail which starts and endson the same vertex.They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this:

Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once?

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published in 1873 by Carl Hierholzer.

The term **Eulerian graph** has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

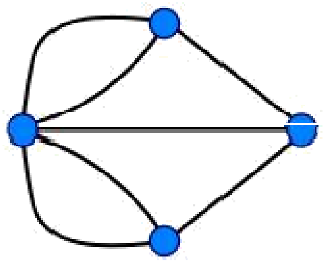
For the existence of Eulerian trails it is necessary that no more than two vertices have an odd degree; this means the Königsberg graph is *not* Eulerian. If there are no vertices of odd degree, all Eulerian trails are circuits. If there are exactly two vertices of odd degree, all Eulerian trails start at one of them and end at the other. Sometimes a graph that has an Eulerian trail but not an Eulerian circuit is called **semi-Eulerian**.

An **Eulerian trail**, **Eulerian trail** or **Euler walk** in an undirected graph is a path that uses each edge exactly once. If such a path exists, the graph is called **traversable** or **semi-eulerian**.

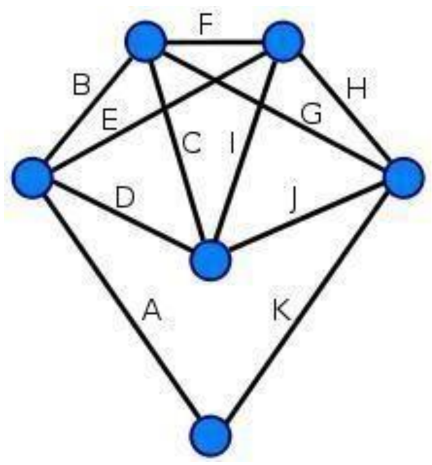
An **Eulerian cycle**, **Eulerian circuit** or **Euler tour** in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called **unicursal**. While such graphs are Eulerian graphs, not every Euler an graph possesses an Eulerian cycle.

For directed graphs path has to be replaced with directed path and cycle with directed cycle.

The definition and properties of Eulerian trails, cycles and graphs are valid for multigraphs as well.



This graph is not Eulerian , therefore ,a solution does not exist.



Every vertex of this graph has an even degree, therefore this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Hamiltoniangraphs:

In the mathematical field of graph theory, a **Hamiltonian path** (or **traceable path**) is a path in an undirected graph which visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a cyclein an undirected graph which visits each vertex exactly once and also returns to the starting vertex. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem which is NP-complete.

Hamiltonian paths and cycles are named after William Rowan Hamilton who invented the Icosian game, now a ls o known as *Hamilton's puzzle* , which involves finding a Hamiltonian cycle in the edge graph of the do decahedron. Hamilton solved this problem using the Icosian Calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton). This solution does not generalize to arbitrary graphs.

A *Hamiltonian path* or *trace able path* is a path that visits each vertex exactly once .A graph that contains a Hamiltonian path is called a **traceable graph** .A graph is **Hamilton-connected** if for every pair of vertices there is a Hamiltonian path between the two vertices.

A *Hamiltonian cycle*, *Hamiltonian circuit* ,*vertex tour* or *graph cycle* is a cycle that visits each vertex exactly once (except the vertex which is both the start and end, and so is visited twice). A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**.

Similar notions may be defined for *directed graphs* , where each edge(arc) of a path or cycle can only be traced in a single direction (i.e., the vertices are connected with arrows and the edges traced "tail-to-head").A **Hamiltoni and e composition** is an edged e composition of a graphin to Hamiltonian circuits.

Examples• a complete graph with more than two vertices is Hamiltonian • every cycle graph is Hamiltonian• every tour name nths and odd number of Hamiltonian paths •every platonic solid, considered as a graph, is Hamiltonian

Chromatic Numbers:

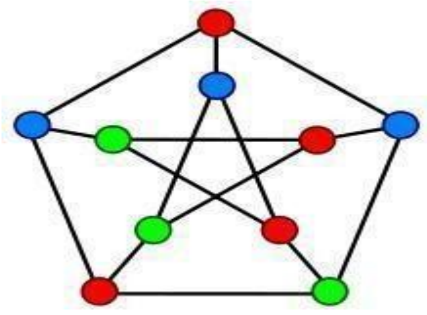
In graph theory, **graph coloring** is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a **vertex coloring**. Similarly, an **edge coloring** assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an edge coloring of a graph is just a vertex coloring of its line graph, and a face coloring of a planar graph is just a vertex coloring of its planar dual. However, non-vertex coloring problems are often stated and studied *as is*. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring.

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. By planar duality it became coloring the vertices, and in this form it generalizes to all graphs. In mathematical and computer representations it is typical to use the first few positive or nonnegative integers as the "colors". In general one can use any finite set as the "color set". The nature of the coloring problem depends on the number of colors but not on what they are.

Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a

color is assigned, or even on the color itself. It has even reached popularity with the general public in the form of the popular number puzzle Sudoku. Graph coloring is still a very active field of research.



A proper vertex coloring of the Petersen graph with 3 colors , the minimum number possible.

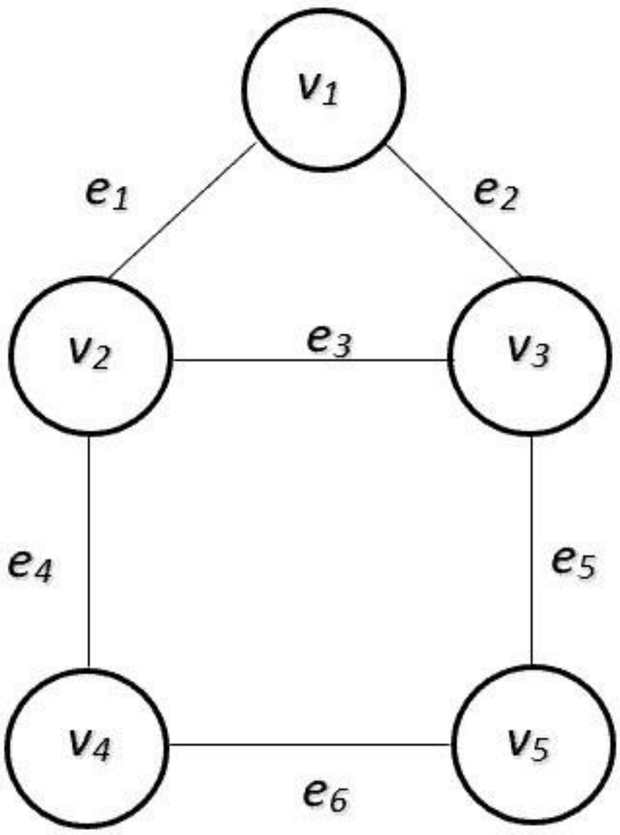
Adjacency and Incidence Matrix

1. Introduction

In this tutorial, we'll discuss graph adjacency and incidence. Also, we'll show how to use them to represent a graph.

2. Graph

In computer science, a graph is a data structure that consists of a set of vertices and edges . An edge is a pair of vertices , where . For example, the following picture shows a graph with vertices and edges:



3. Adjacency

If two vertices in a graph are connected by an edge, we say the vertices are adjacent. In our graph example, vertex has two adjacent vertices, and . Base on this property, we can use an adjacency matrix or adjacency list to represent a graph.

Adjacency Matrix

Suppose we have a graph with vertices, we can use a square matrix to represent the adjacency relationships among these vertices. For example, the adjacency matrix of the example graph is:

	0	1	1	0	0
	1	0	1	1	0
	1	1	0	0	1
	0	1	0	0	1
	0	0	1	1	0

In this matrix, the number means the corresponding two vertices are adjacent. Otherwise, the entry value is .Since we have entries, the space complexity of the adjacency matrix is .

To build the adjacency matrix, we can go through all edges and set 1 to the corresponding vertex-vertex entry. Therefore, the time complexity to build this matrix is , where is the number of graph edges.

Adjacency List

We can also use an adjacency list to represent a graph. For example, the adjacency list of the example graph is:

In this table, each row contains a list of vertices that is adjacent to the current vertex . Each pair represent an edge in the graph. Therefore, the space complexity of the adjacency list is . Similarly, we need time to build the adjacency list.

Adjacency Matrix v.s. Adjacency List

For a dense graph, where the number of edges is in the order of , the adjacency matrix and adjacency list have the same time and space complexity. However, if the graph is sparse, we need less space to represent the graph. Therefore, an adjacency list is more space-efficient than an adjacency matrix when we work on sparse graphs.

However, there are some graph operations where the adjacency matrix is more efficient to use. For example, when we want to check if there exists an edge in the graph, we can just look up the adjacency matrix in constant time to get the result. If we use the adjacency list, it will take us time to check.

Another example is to remove an edge from the graph. In the adjacency matrix, we can just set to the corresponding entries in constant time. However, we need time to remove the vertex from the adjacency list.

4. Incidence

In a graph , two edges are incident if they share a common vertex. For example, edge and edge are incident as they share the same vertex .

Also, we can define the incidence over a vertex. **A vertex is an incident to an edge if the vertex is one of the two vertices the edge connects.** Therefore, an incidence is a pair where is a vertex and is an edge incident to .

Base on this property, we can use an incidence matrix to represent a graph. For example, the incidence matrix of the example graph is:

	1	1	0	0	0	0
	1	0	1	1	0	0
	0	1	1	0	1	0
	0	0	0	1	0	1
	0	0	0	0	1	1

In this matrix, rows represent vertices and columns represent edges. Therefore, the space complexity of the incidence matrix is . To build the incidence matrix, we can go through all edges and set 1 to the corresponding vertex-edge entry. Therefore, the time complexity to build this matrix is .

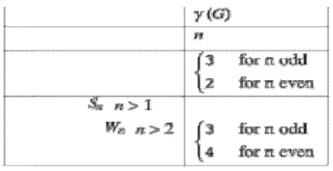
The incidence matrix and adjacency matrix of a graph have a relationship of , where is the identity matrix.

The incidence matrix has more space complexity than the other graph representations. We normally use it in theoretic graph areas. e.g., incidence coloring of a graph.

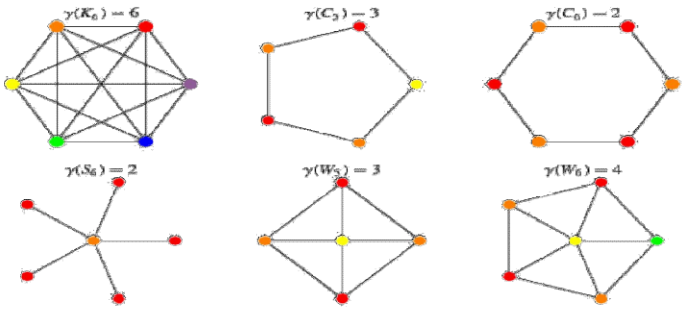
5. Conclusion

This article provided the definitions of adjacency and incidence in graph theory. Also, we showed different ways to represent a graph using adjacency and incidence.

Vertexcoloring



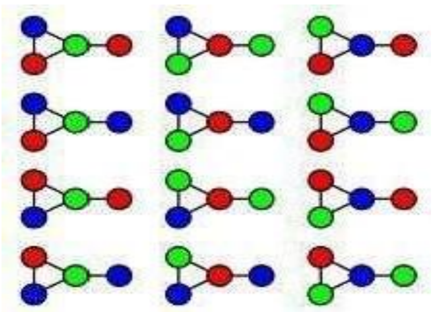
	$\gamma(G)$
	n
	$\begin{cases} 3 & \text{for n odd} \\ 2 & \text{for n even} \end{cases}$
$S_n \quad n > 1$ $W_n \quad n > 2$	$\begin{cases} 3 & \text{for n odd} \\ 4 & \text{for n even} \end{cases}$



When used without any qualification, a **coloring** of a graph is almost always a *proper vertex coloring*, namely a labelling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color .Since a vertex with a loop could never be properly colored, it is understood that graphs in this context are loop less.

The terminology of using *colors* for vertex labels goes back to map coloring. Labels like *red* and *blue* are only used when the nunber of colors is small, and normally it is understood that the labels are drawn from the integers {1,2,3,...}.

A coloring using at most *k* colors is called a (proper) ***k*-coloring**. The smallest number of colors needed to color a graph *G* is called its **chromatic number**, χ(*G*). A graph that can be assigned a (proper) *k*-coloring is ***k*-colorable**, and it is ***k*-chromatic** if its chromatic number is exactly *k*. A subset of vertices assigned to the same color is called a *color class*, every such class forms an independent set. Thus, a *k*-coloring is the same as a partition of the vertex set into *k* independent



sets, and the terms *k-partite* and *k-colorable* have the same meaning.

This graph can be 3-colored in12 different ways.

TREES

Introduction:

A graph which has no cycle is called an acyclic graph. A tree is an acyclic graph or graph having no cycles.

Definition : A tree or general trees is defined as a non-empty finite set of elements called vertices or nodes having the property that each node can have minimum degree 1 and maximum degree n. It can be partitioned into n+1 disjoint subsets such that the first subset contains the root of the tree and remaining n subsets includes the elements of the n subtree.

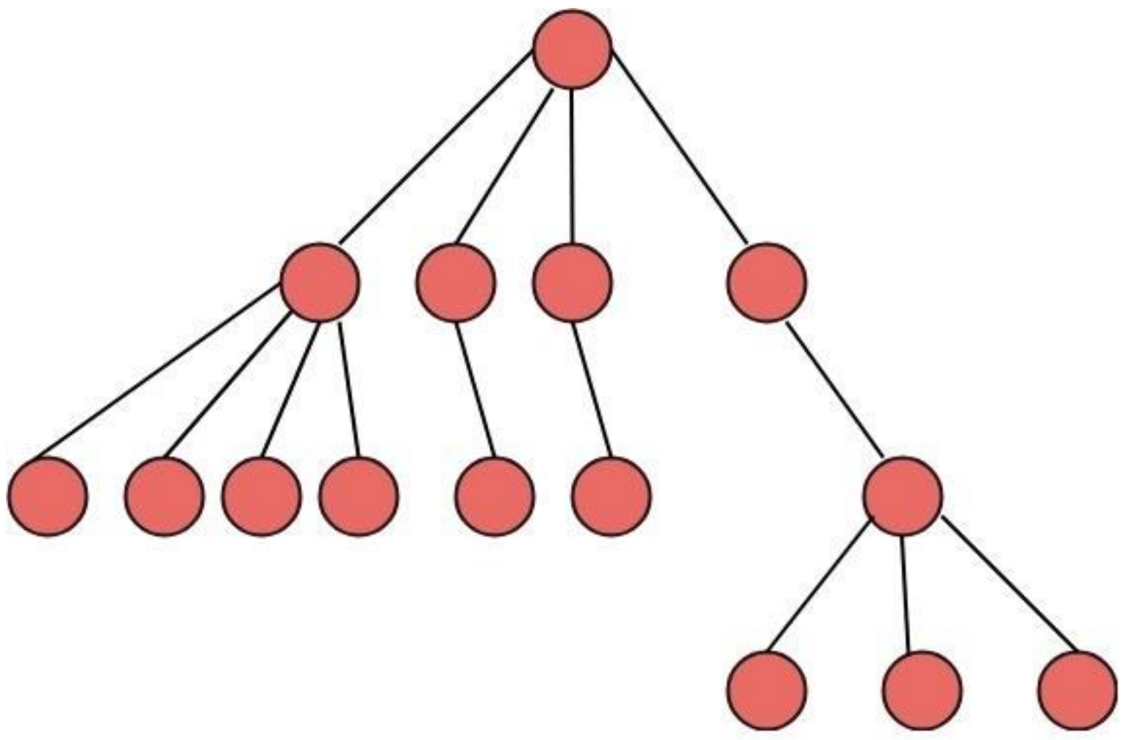
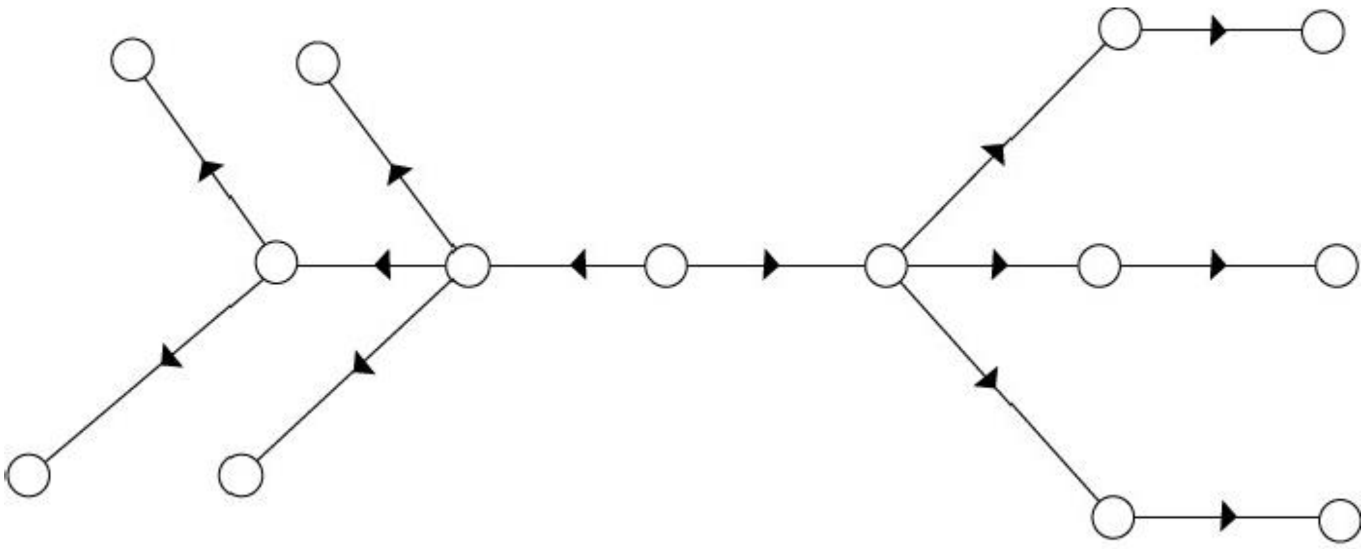


Fig:General Trees

Directed Trees:

A directed tree is an acyclic directed graph. It has one node with indegree 1, while all other nodes have indegree 1 as shown in fig:



Directed Trees

The node which has outdegree 0 is called an external node or a terminal node or a leaf. The nodes which have outdegree greater than or equal to one are called internal node.

Binary Trees:

If the outdegree of every node is less than or equal to 2, in a directed tree than the tree is called a binary tree. A tree consisting of the nodes (empty tree) is also a binary tree. A binary tree is shown in fig:

Basic Terminology:

Root: A binary tree has a unique node called the root of the tree.

Left Child: The node to the left of the root is called its left child.

Right Child: The node to the right of the root is called its right child.

Parent: A node having a left child or right child or both are called the parent of the nodes.

Siblings: Two nodes having the same parent are called siblings.

Leaf: A node with no children is called a leaf. The number of leaves in a binary tree can vary from one (minimum) to half the number of vertices (maximum) in a tree.

Descendant: A node is called descendant of another node if it is the child of the node or child of some other descendant of that node. All the nodes in the tree are descendants of the root.

Left Subtree: The subtree whose root is the left child of some node is called the left subtree of that node.

Right Subtree: The subtree whose root is the right child of some node is called the right subtree of that node.

Level of a Node: The level of a node is its distance from the root. The level of root is defined as zero. The level of all other nodes is one more than its parent node. The maximum number of nodes at any level N is 2^N.

Depth or Height of a tree: The depth or height of a tree is defined as the maximum number of nodes in a branch of a tree. This is more than the maximum level of the tree, i.e., the depth of root is one. The maximum number of nodes in a binary tree of depth d is 2^d-1, where d ≥1.

External Nodes: The nodes which have no children are called external nodes or terminal nodes.

Internal Nodes: The nodes which have one or more than one children are called internal nodes or non-terminal nodes.

Binary Expression Trees:

An algebraic expression can be conveniently expressed by its expression tree. An expression having binary operators can be decomposed into

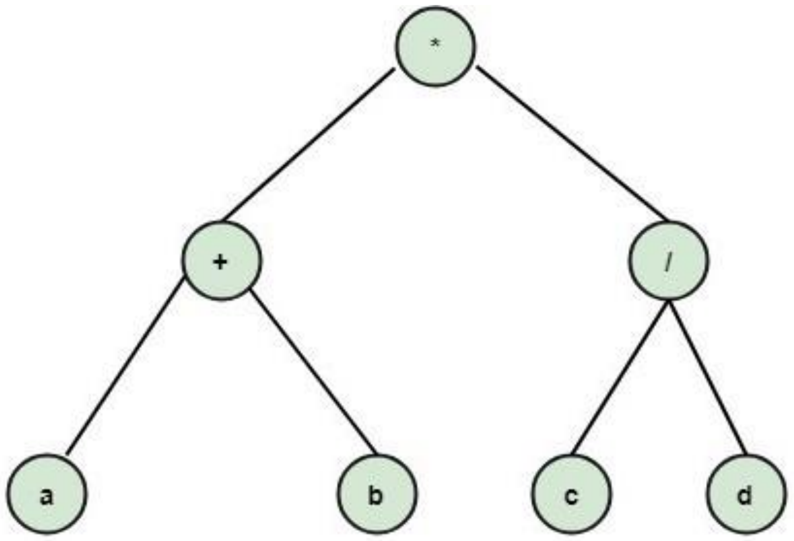
<left operand or expression> (operator) <right operand or expression>

Depending upon precedence of evaluation.

The expression tree is a binary tree whose root contains the operator and whose left subtree contains the left expression, and right subtree contains the right expression.

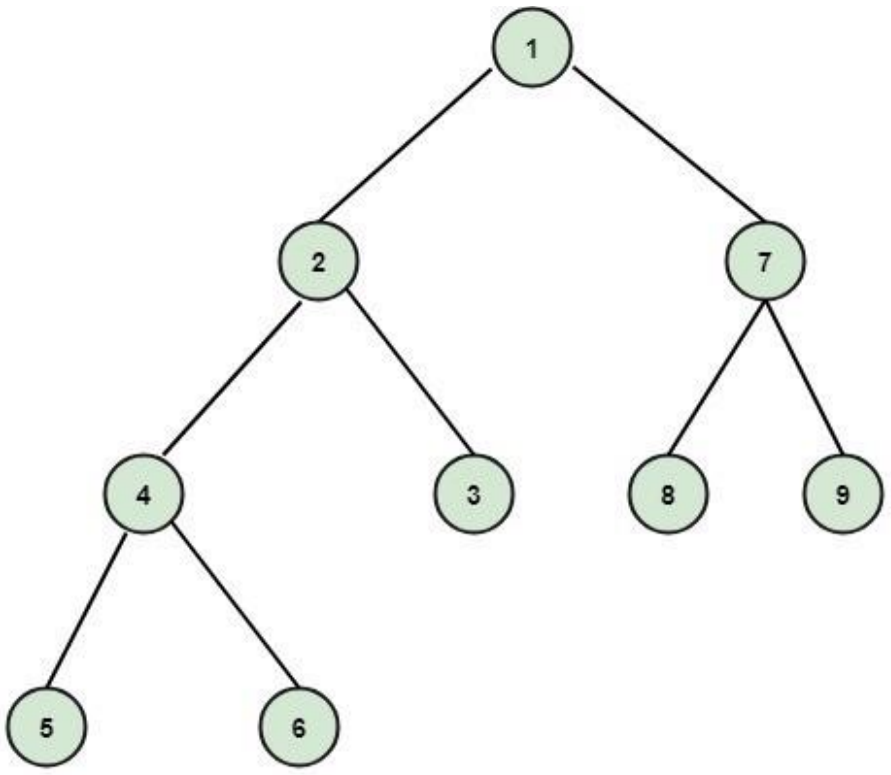
Example: Construct the binary expression tree for the expression (a+b)*(d/c)

Solution: The binary expression tree for the expression (a+b)*(d/c) is shown in fig:

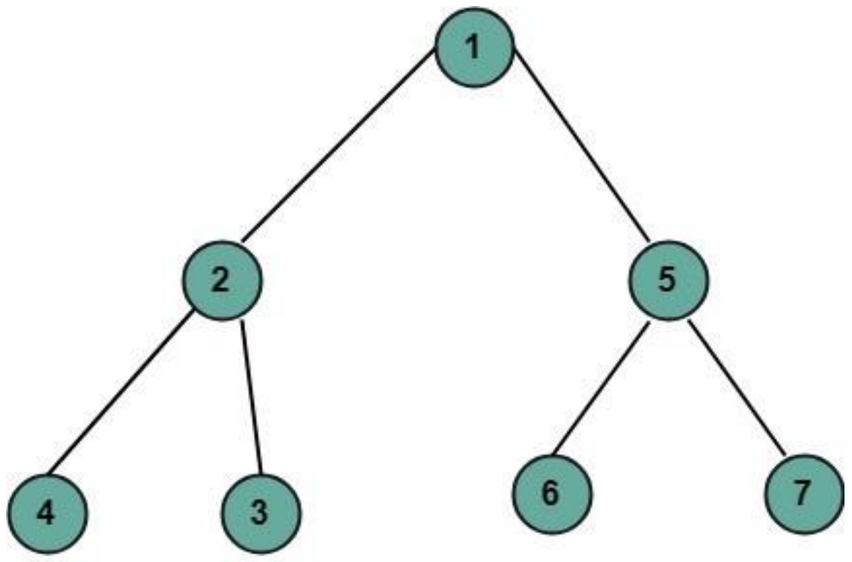


Complete Binary Tree: Complete binary tree is a binary tree if it is all levels, except possibly the last, have the maximum number of possible nodes as for left as possible. The depth of the complete binary tree having n nodes is $\log_2 n+1$.

Example: The tree shown in fig is a complete binary tree.



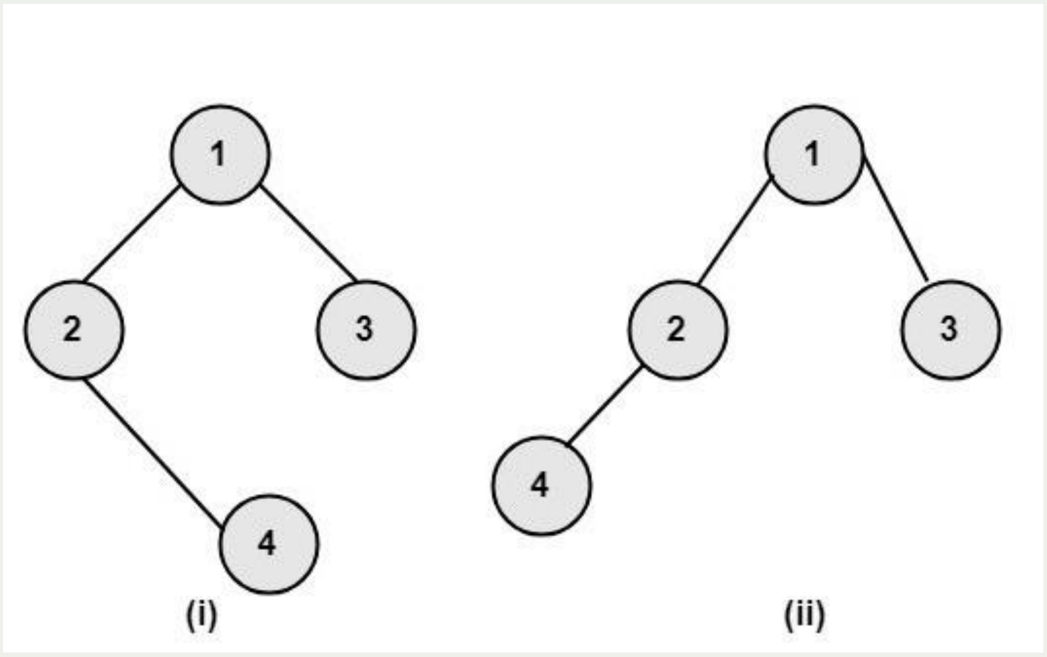
Full Binary Tree: Full binary tree is a binary tree in which all the leaves are on the same level and every non-leaf node has two children.

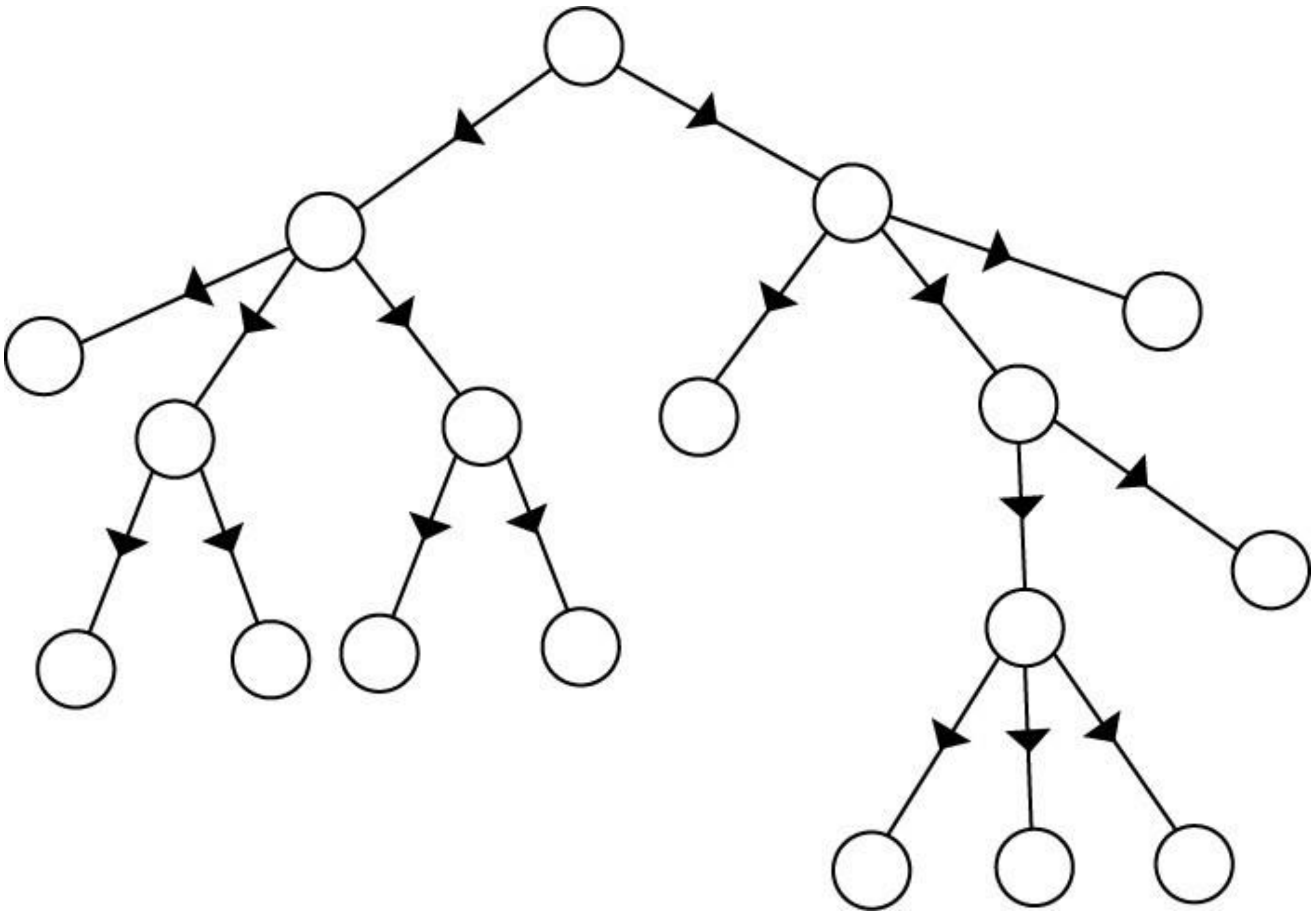


Full Binary Tree

Differentiate between General Tree and Binary Tree

General Tree	Binary Tree
1. There is no such tree having zero nodes or an empty general tree.	1. There may be an empty binary tree.
2. If some node has a child, then there is no such distinction.	2. If some node has a child, then it is distinguished as a left child or a right child.
3. The trees shown in fig are the same, when we consider them as general trees.	3. The trees shown in fig are distinct, when we consider them as binary trees, because in (4) is the right child of 2 while in (ii) 4 is a left child of 2.

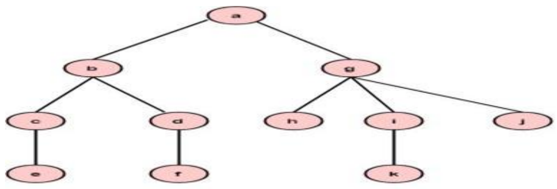


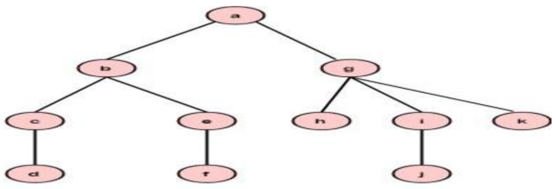


Ordered Trees:

If in a tree at each level, an ordering is defined, then such a tree is called an ordered tree.

Example: The trees shown in the figures represent the same tree but have different orders.





Properties of Trees:

1. There is only one path between each pair of vertices of a tree.
2. If a graph G there is one and only one path between each pair of vertices G is a tree.

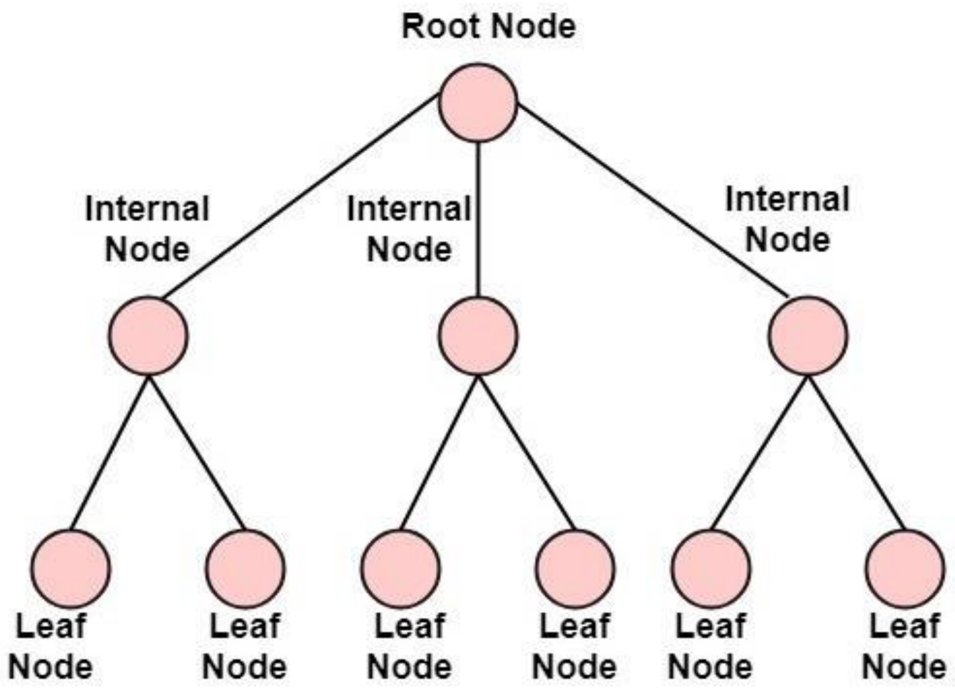
3. A tree T with n vertices has n-1 edges.

4. A graph is a tree if and only if it a minimal connected.

Rooted Trees:

If a directed tree has exactly one node or vertex called root whose incoming degrees is 0 and all other vertices have incoming degree one, then the tree is called rooted tree.

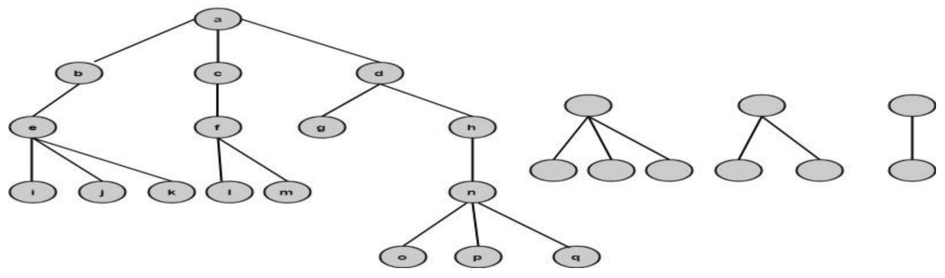
Note: 1. A tree with no nodes is a rooted tree (the empty tree)
2. A single node with no children is a rooted tree.



Path length of a Vertex:

The path length of a vertex in a rooted tree is defined to be the number of edges in the path from the root to the vertex.

Example: Find the path lengths of the nodes b, f, l, q as shown in fig:



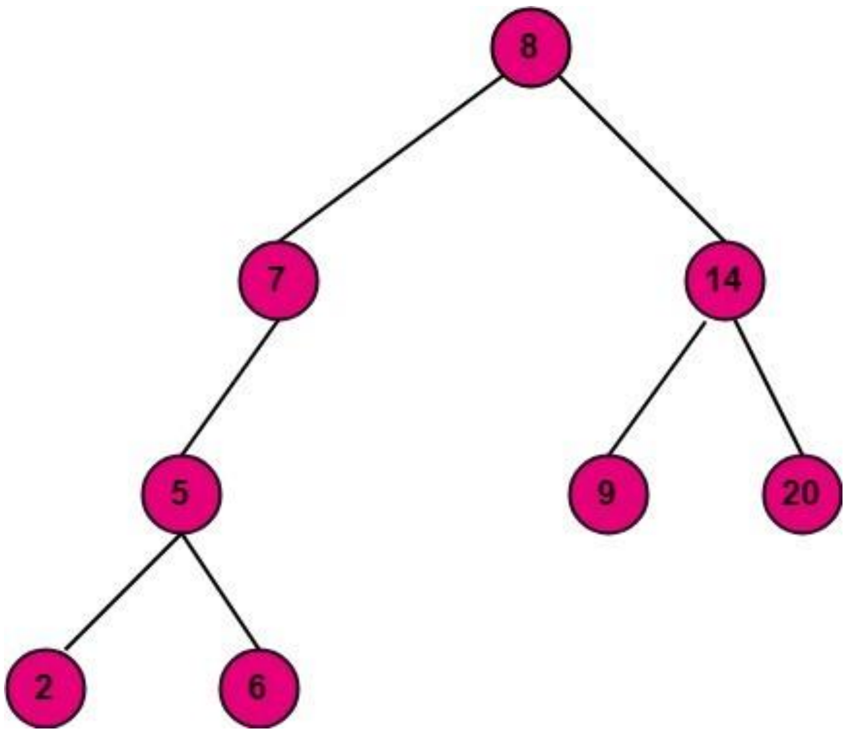
Solution: The path length of node b is one.
The path length of node f is two.
The path length of node l is three
The path length of the node q is four.

Binary Search Trees

Binary search trees have the property that the node to the left contains a smaller value than the node pointing to it and the node to the right contains a larger value than the node pointing to it.

It is not necessary that a node in a 'Binary Search Tree' point to the nodes whose value immediately precede and follow it.

Example: The tree shown in fig is a binary search tree.



Inserting into a Binary Search Tree: Consider a binary tree T. Suppose we have given an ITEM of information to insert in T. The ITEM is inserted as a leaf in the tree. The following steps explain a procedure to insert an ITEM in the binary search tree T.

1. Compare the ITEM with the root node.
2. If ITEM>ROOT NODE, proceed to the right child, and it becomes a root node for the right subtree.
3. If ITEM<ROOT NODE, proceed to the left child.
4. Repeat the above steps until we meet a node which has no left and right subtree.
5. Now if the ITEM is greater than the node, then the ITEM is inserted as the right child, and if the ITEM is less than the node, then the ITEM is inserted as the left child.

Example: Show the binary search tree after inserting 3, 1,4,6,9,2,5,7 into an initially empty binary search tree.

Solution: The insertion of the above nodes in the empty binary search tree is shown in fig:

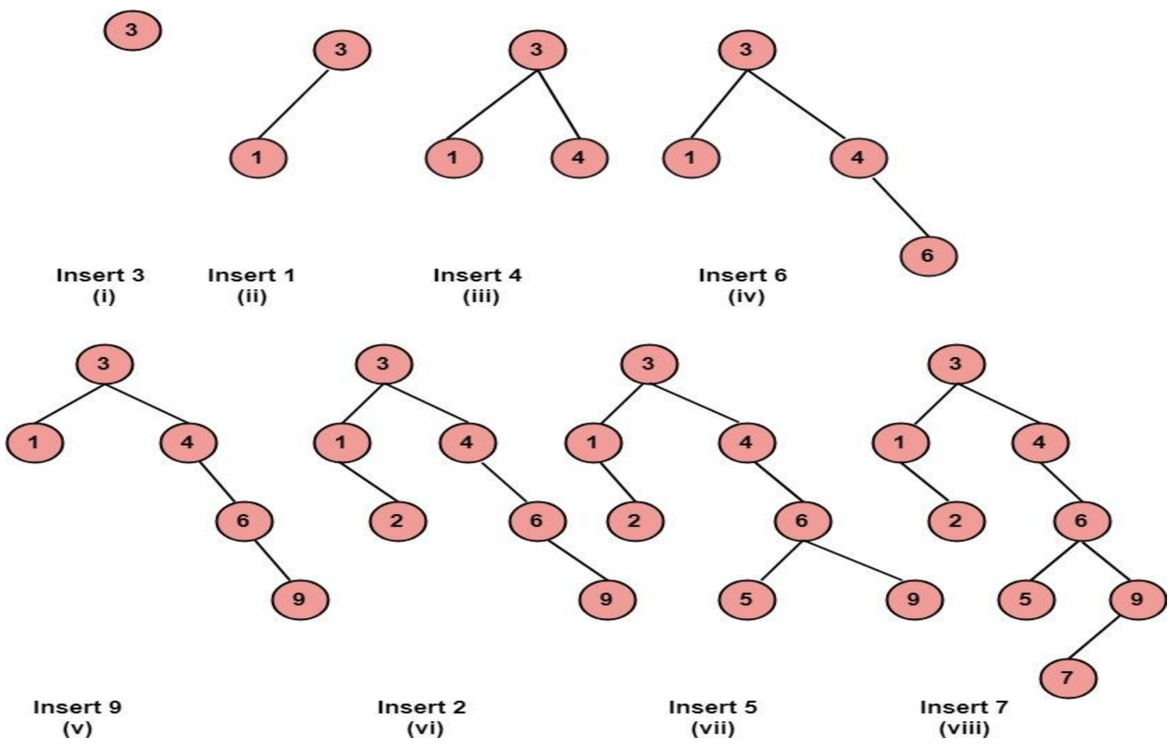
Deletion in a Binary Search Tree: Consider a binary tree T. Suppose we want to delete a given ITEM from binary search tree. To delete an ITEM from a binary search tree we have three cases, depending upon the number of children of the deleted node.

1. **Deleted Node has no children:** Deleting a node which has no children is very simple, as replace the node with null.
2. **Deleted Node has Only one child:** Replace the value of a deleted node with the only child.
3. **Deletion node has only two children:** In this case, replace the deleted node with the node that is closest in the value to the deleted node. To find the nearest value, we move once to the left and then to the right as far as possible. This node is called the immediate predecessor. Now replace the value of the deleted node with the immediate predecessor and then delete the replaced node by using case1 or case2.

Example: Show that the binary tree shown in fig (viii) after deleting the root node.

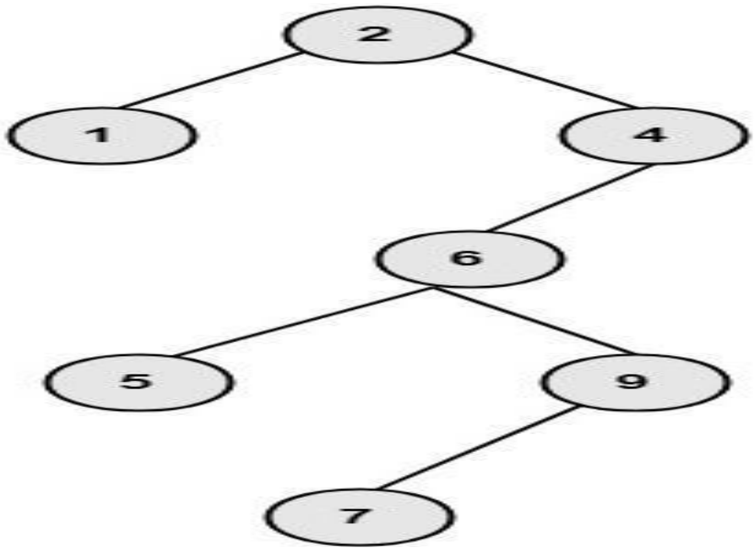
Solution: To delete the root node, first replace the root node with the closest elements of the root. For this, first, move one step left and then to the right as far as possible to the node. Then delete the replaced node.

The tree after deletion shown in fig:



Spanning Tree

A subgraph T of a connected graph G is called spanning tree of G if T is a tree and T include all vertices of G.



Minimum Spanning Tree:

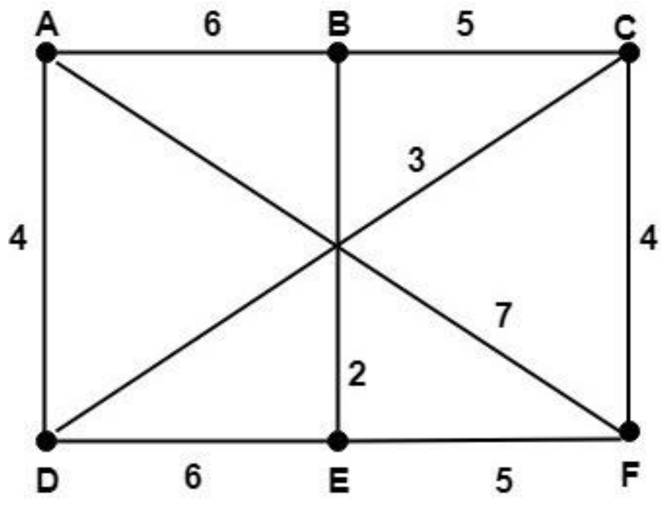
Suppose G is a connected weight graph i.e., each edge of G is assigned a non-negative number called the weight of edge, then any spanning tree T of G is assigned a total weight obtained by adding the weight of the edge in T.

A minimum spanning tree of G is a tree whose total weight is as small as possible.

Kruskal's Algorithm to find a minimum spanning tree: This algorithm finds the minimum spanning tree T of the given connected weighted graph G.

1. Input the given connected weighted graph G with n vertices whose minimum spanning tree T, we want to find.
2. Order all the edges of the graph G according to increasing weights.
3. Initialize T with all vertices but do include an edge.
4. Add each of the graphs G in T which does not form a cycle until n-1 edges are added.

Example1: Determine the minimum spanning tree of the weighted graph shown in fig:

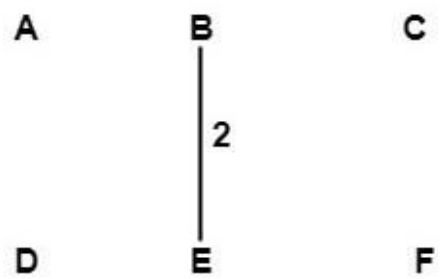


Solution: Using kruskal's algorithm arrange all the edges of the weighted graph in increasing order and initialize spanning tree T with all the six vertices of G. Now start adding the edges of G in

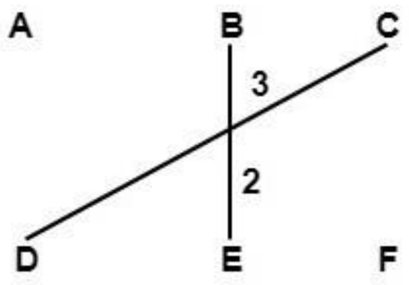
T which do not form a cycle and having minimum weights until five edges are not added as there are six vertices.

Edges	Weights	Added or Not
(B,E)	2	added
(C,D)	3	added
(A,D)	4	added
(C,F)	4	added
(B,C)	5	added
(E,F)	5	Not added
(A,B)	6	Not added
(D,E)	6	Not added

Step1:



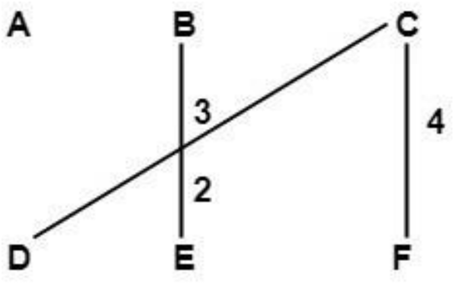
(add B, E)



(add D, C)

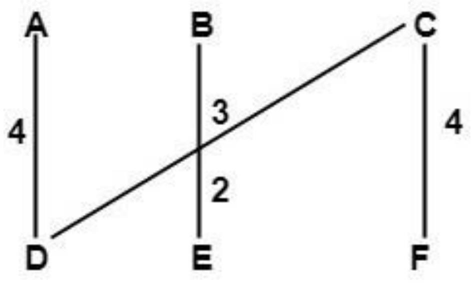
Step2:

Step3:

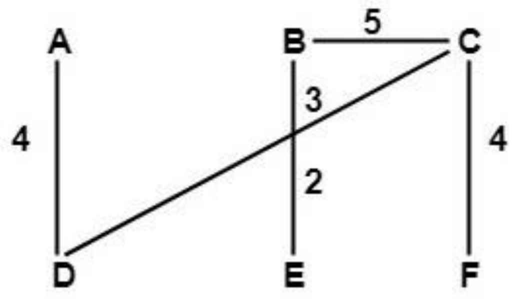


(add C, F)

Step4:



(add A, D)

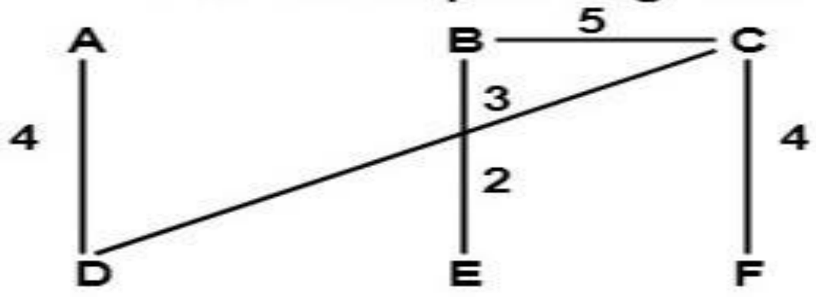


Step5: (add B, C)

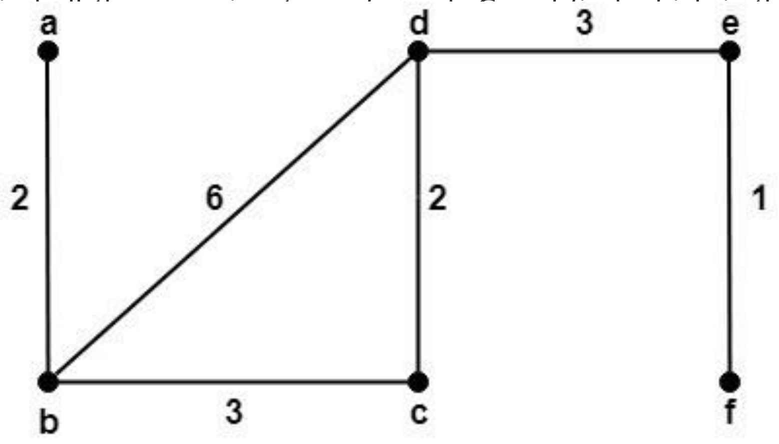
Step6: Edge (A, B), (D, E) and (E, F) are discarded because they will form the cycle in a graph.

So, the minimum spanning tree form in step 5 is output, and the total cost is 18.

Minimum Spanning Tree

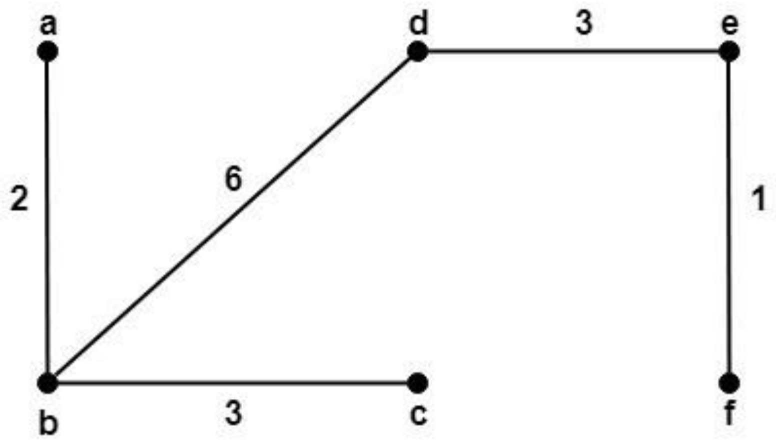


Example2: F [illegible] e minimal spanning tree of G

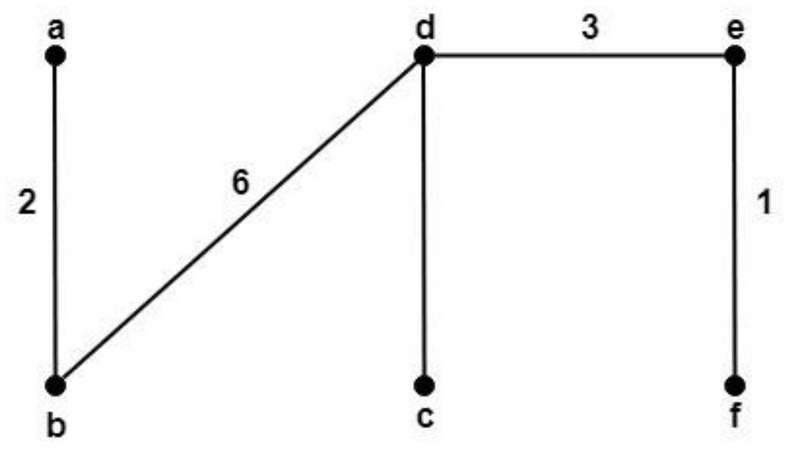


shown in fig:

Solution: There are total three spanning trees of the graph G which are shown in fig:



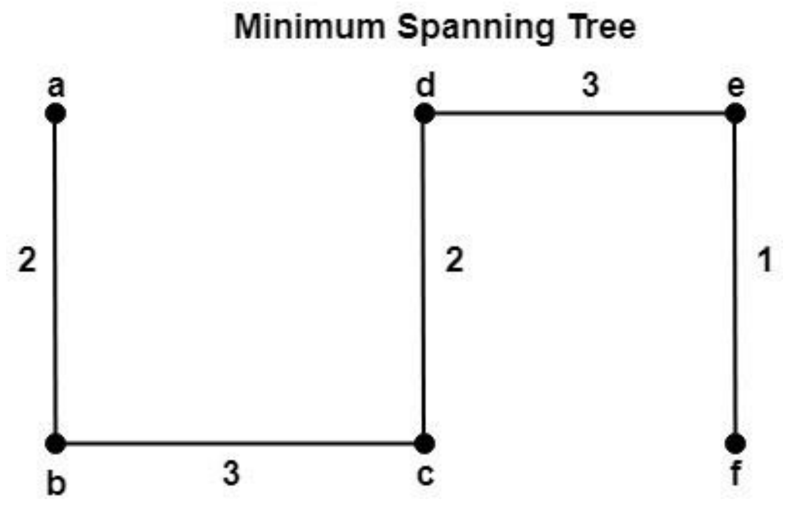
(ii)



(iii)

To find the minimum spanning tree, use the KRUSKAL'S ALGORITHM. The minimal spanning tree is shown in fig:

Edges	Weights	Added or Not
(E, F)	1	Added
(A, B)	2	Added
(C, D)	2	Added
(B,C)	3	Added
(D,E)	3	Added
(B,D)	3	Not Added



The first one is the minimum spanning having the minimum weight = 11.