ASHOKA

INSTITUE OF TECHNOLOGY & MANAGEMENT

(DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING)

Lecture Notes: -

Discrete Structure & Theory of Logic

Faculty Name: -

Ms. Kavita Patel

Unit – 1

Set Theory, Functions and Natural Numbers

1.1 Set Theory: Introduction

A collection of unique things of the same type or class of objects is referred to as a set. A set's goals are referred to as its elements or members. Numbers, alphabets, names, and other objects are examples of objects.

Sets- A set is a collection of well-defined objects which are called the elements or members of the set.

We generally use capital letters to denote sets and lowercase letters for elements.

If any element which exists in the set is to be denoted as-

Suppose an element 'a' is from a set X, then it is represented as- $a \in X$

Which means the element 'a' belongs to the set X.

If the element is not form the group then we use 'not belongs to'.

Examples of sets are:

1. A set of rivers of India.
2. A set of vowels.

A set is denoted by the capital letters A, B, C, and so on, whereas the basics of the set are denoted by the small letters a, b, x, y, and so on.

If A is a set and an is one of its elements, we refer to it as an $a \in A$. "Element of" is the meaning of the sign $\in$ in this case.

Set Representation

There are two ways to express a set:

a) Roster or tabular form: We list all the components of the set within braces {} and separate them with commas in this type of representation.

For example, if A is a set of all odd numbers less than 10, then A= {1,3,5,7,9} can be written in the roster.

b) Set Builder form: In this representation, we list all of the qualities that all of the set's elements satisfy. We write as {x: x satisfies properties P}. and can be read as 'the set of all x such that each x has the attributes P.'

For example, If B= {2, 4, 8, 16, 32}, the set builder representation will be: $B=\{x: x=2^n$, where $n \in N$ and $1 \leq n \geq 5\}$.

Subsets-

Let every element of set X is also an element of a set Y, then X is said to be the subset of Y.

Symbolically it is written as-

$X \subseteq Y$

Which is read as- X is the subset of Y.

Note- If X = Y then $X \subseteq Y \text{ and } Y \subseteq X$

Proper sub-set-

A set X is called proper subset the set Y is-

1. X is subset of Y

2. Y is not subset of X

Note- every set is a subset to itself.

Equal sets-

If X and Y are two sets such that every element of X is an element of Y and every element of Y is an element of X, the set and set Y will be equal always.

We write it as X = Y and read as X and Y are identical.

Super set-

If X is a subset of Y, then Y is called a super set of X.

Null set-

A set which does not contain any element is known as null set.

We denote a null set by ∅

The null set is a subset of every set.

Singleton-

A set which has only one element is called singleton.

Example- A = {10} and B = {∅}

Theorem- if ∅ and ∅' are empty sets then ∅=∅'

Proof: Let ∅≠ ∅' then the following conditions must be true-

1. There is element x∈∅ such that x∉∅'

2. There is element x∈∅' such that x∉∅.

But both these conditions are false since ∅' and ∅' has any elements if follows that ∅=∅'.

Key takeaway

1. A collection of unique things of the same type or class of objects is referred to as a set.
2. A set's goals are referred to as its elements or members.
3. Numbers, alphabets, names, and other objects are examples of objects.

1.2 Combination of sets

A Venn diagram is a picture of a set in which the sets are represented by enclosed areas in the plane. The interior of a rectangle represents the universal set U, whereas disks within the rectangle represent the other sets. If A B, the disk representing A will be completely enclosed by the disk representing B, as seen in Figure (a). If A and B are disjoint, the disks representing A and B will be separated, as seen in Fig(b).

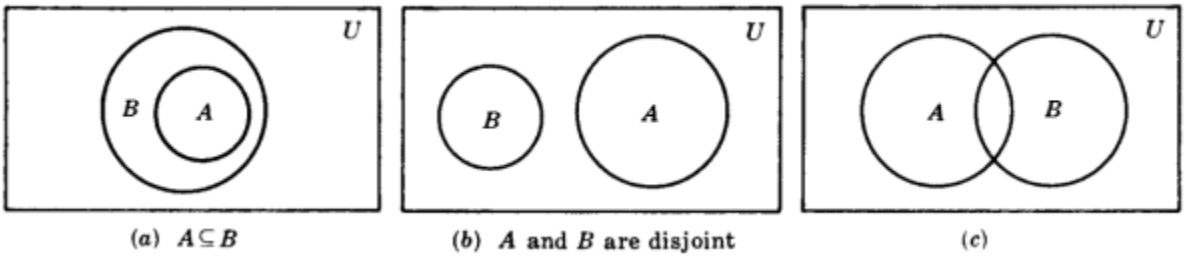


Fig 1: Venn diagram

However, if A and B are two arbitrary sets, some items may be in A but not in B, some in B but not in A, some in both A and B, and some in neither A nor B; so, in general, we express A and B as in Fig (c).

Union and Intersection

The set of all elements that belong to either A or B, denoted by A B, is the union of two sets A and B.

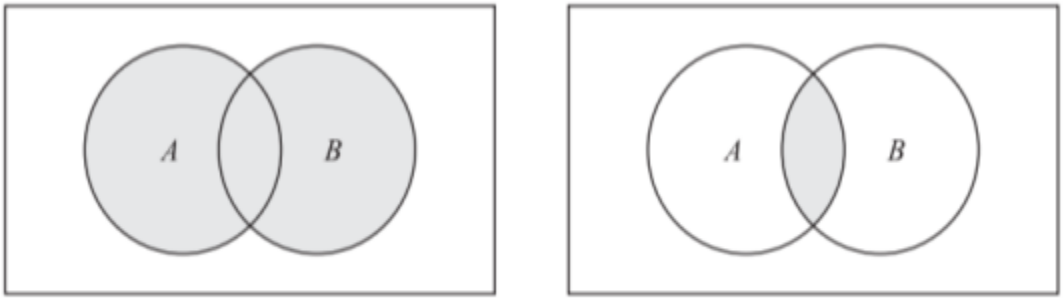
$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

The word "or" is used in this context to mean "and/or." A Venn diagram with A and B shading is shown in Figure (a).

The set of elements that belong to both A and B, indicated as A B, is the **intersection** of two sets A and B.

$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

A Venn diagram with A and B shading is shown in Figure (b).



(a) $A \cup B$ is shaded (b) $A \cap B$ is shaded

Fig 2: Union and intersection

Key takeaway

1. A Venn diagram is a picture of a set in which the sets are represented by enclosed areas in the plane.
2. The interior of a rectangle represents the universal set U, whereas disks within the rectangle represent the other sets.

1.3 Multisets

A multiset is an unsorted collection of elements in which each element's multiplicity can be one, multiples of one, or zero. The multiplicity of an element refers to how many times it appears in the multiset. To put it another way, we can say that an element can appear in a set any number of times.

Example

A = {l, l, m, m, n, n, n, n}

B = {a, a, a, a, a, c}

Operation on multiset

1. **Union of multisets**

The union of two multisets A and B is a multiset in which the multiplicity of an element equals the greatest multiplicity of an element in both A and B, and is represented by A ∪ B.

Example

Let A = {l, l, m, m, n, n, n, n}

B = {l, m, m, m, n},

A ∪ B = {l, l, m, m, m, n, n, n, n}

2. **Intersection of multisets**

A b is a multiset formed by the intersection of two multisets A and B, where the multiplicity of an element is equal to the minimum of the multiplicity of an element in both A ∩ B.

Example

Let A = {l, l, m, n, p, q, q, r}

B = {l, m, m, p, q, r, r, r, r}

A ∩ B = {l, m, p, q, r}.

3. Difference of multisets

The difference of two multisets A and B is a multiset whose multiplicity is equal to the multiplicity of the element in A minus the multiplicity of the element in B if the difference is positive, and is equal to 0 if the difference is negative or zero.

Example

Let A = {l, m, m, m, n, n, n, p, p, p}

B = {l, m, m, m, n, r, r, r}

A - B = {n, n, p, p, p}

4. Sum of multisets

The sum of two multisets A and B is a multiset in which an element's multiplicity equals the sum of its multiplicity in both A and B.

Example

Let A = {l, m, n, p, r}

B = {l, l, m, n, n, n, p, r, r}

A + B = {l, l, l, m, m, n, n, n, n, p, p, r, r, r}

5. Cardinality of sets

The cardinality of a multiset is the number of different elements in the set, ignoring the element's multiplicity.

Example: A = {l, l, m, m, n, n, n, p, p, p, p, q, q, q}

The multiset A has a cardinality of 5.

Key takeaway

1. A multiset is an unsorted collection of elements in which each element's multiplicity can be one, multiples of one, or zero.
2. The multiplicity of an element refers to how many times it appears in the multiset.

3. To put it another way, we can say that an element can appear in a set any number of times.

1.4 Ordered pairs

An Ordered Pair is made up of two parts, one of which is identified as the first and the other as the second.

The ordered pairs (a, b) and (b, a) are distinct. In the case of an ordered pair, an ordered triple can be represented as (a, b) c.

An ordered Quadrable is an ordered pair with the first element as an ordered triple (((a, b), c) d).

An ordered n-tuple is an ordered pair with an ordered (n - 1) tuple as the first component and the nth element as the second component.

{(n -1), n}

Example

Ordered set of 5 elements

$$\underbrace{\{((((a,b),c),d),e)\}}_{(n-1)\downarrow 5\text{th}}$$

1.5 Proofs of some general identities on sets

Example 1: Use set builder notation and logical equivalences to establish the first De Morgan law $A \cap B = A \cup B$.

Solution: We can prove this identity with the following steps

$A \cap B = \{x \mid x /\in A \cap B\}$ by definition of complement

$= \{x \mid \neg(x \in (A \cap B))\}$ by definition of does not belong symbol

$= \{x \mid \neg(x \in A \wedge x \in B)\}$ by definition of intersection

$= \{x \mid \neg(x \in A) \vee \neg(x \in B)\}$ by the first De Morgan law for logical equivalences

$= \{x \mid x /\in A \vee x /\in B\}$ by definition of does not belong symbol

$= \{x \mid x \in A \vee x \in B\}$ by definition of complement

$= \{x \mid x \in A \cup B\}$ by definition of union

$= A \cup B$ by meaning of set builder notation

This proof employs the second De Morgan law for logical equivalences, in addition to the definitions of complement, union, set membership, and set builder notation.

Proving a set identity involving more than two sets by demonstrating that one side is a subset of the other sometimes necessitates keeping track of distinct circumstances.

Example 2: Let A, B, and C be sets. Show that $A \cup (B \cap C) = (C \cup B) \cap A$.

Solution: We have

$A \cup (B \cap C) = A \cap (B \cap C)$ by the first De Morgan law

$= A \cap (B \cup C)$ by the second De Morgan law

$= (B \cup C) \cap A$ by the commutative law for intersections

$= (C \cup B) \cap A$ by the commutative law for unions

1.6 Relations: Definition, Operations on relations, Properties of relations

Definition and Properties

Let P and Q be two sets that aren't empty. A subset of P x Q from a set P to Q is defined as a binary relation R. If $(a, b) \in R$ and $R \subseteq P \times Q$, then a and b are connected by R, i.e., aRb. When the sets P and Q are equivalent, we say $R \subseteq P \times P$ is a relation on P, for example

(i) Let A = {a, b, c}

B = {r, s, t}

Then R = {(a, r), (b, r), (b, t), (c, s)}

Is a relation from A to B.

(ii) Let A = {1, 2, 3} and B = A

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

Is a relation (equal) on A.

Example - How many relations exist between A and A if a set has n elements?

Solution -

A x A has n^2 items if the set A has n elements. As a result, there exist 2^{n2} relationships between A and A.

Domain of relation - The set of elements in P that are related to some elements in Q, or the set of all initial entries of the ordered pairs in R, is the domain of relation R. DOM is the abbreviation for it (R).

Range of relation - The scope of the relationship R is the set of all second entries of the ordered pairs in R, alternatively it is the set of all elements in Q that are connected to some element in P. RAN is the abbreviation for it (R).

Example: Let A = {1, 2, 3, 4}

B = {a, b, c, d}

R = {(1, a), (1, b), (1, c), (2, b), (2, c), (2, d)}.

Solution:

DOM (R) = {1, 2}

RAN (R) = {a, b, c, d}

Operation on relation

A subset of the Cartesian product of A and B is a binary relation between two nonempty sets A and B (in that order) (in that order). Because relations are sets, they can be affected by standard set operations. For example, given two relations R1 and R2, we can compute their union, intersection, and difference.

Union of two relations

Given two relations R1 and R2, the union of these relations, represented by $R_1 \cup R_2$, will be defined as

$R_1 \cup R_2$ ab R ab R = $\{(a, b) / (a, b) \in R_1 \vee (a, b) \in R_2\}$

Intersection of two relations

Given two relations R1 and R2, we shall define $R_1 \cap R_2$ as the intersection of these two relations.

$R_1 \cap R_2 = \{(a, b) / (a, b) \in R_1 \wedge (a, b) \in R_2\}$

Difference of two relations

We will define the difference of relations R1 and R2 (in that order), represented by $R_1 - R_2$, as follows:

$R_1 - R_2 = \{(a, b) / (a, b) \in R_1 \wedge (a, b) \notin R_2\}$

Key takeaway

1. The set of elements in P that are related to some elements in Q, or the set of all initial entries of the ordered pairs in R, is the domain of relation R.
2. The scope of the relationship R is the set of all second entries of the ordered pairs in R.
3. A subset of the Cartesian product of A and B is a binary relation between two nonempty sets A and B (in that order).

1.7 Composite Relations

Let A, B, and C be sets, and let R be a relation from A to B and let S be a relation from B to C. That is, R is a subset of $A \times B$ and S is a subset of $B \times C$. Then R and S give rise to a relation from A to C indicated by $R \circ S$ and defined by:

a $(R \circ S)$ c if for some $b \in B$ we have aRb and bSc.

Is,

$R \circ S = \{(a, c) \mid$ there exists $b \in B$ for which $(a, b) \in R$ and $(b, c) \in S\}$

The relation $R \circ S$ is known the composition of R and S; it is sometimes denoted simply by RS.

Let R is a relation on a set A, that is, R is a relation from a set A to itself. Then $R \circ R$, the composition of R with itself, is always represented. Also, $R \circ R$ is sometimes denoted by R^2. Similarly, $R^3 = R^2 \circ R = R \circ R \circ R$, and so on. Thus R^n is defined for all positive n.

Example: Let X = {4, 5, 6}, Y = {a, b, c} and Z = {l, m, n}. Consider the relation R_1 from X to Y and R_2 from Y to Z.

$R_1 = \{(4, a), (4, b), (5, c), (6, a), (6, c)\}$

$R_2 = \{(a, l), (a, n), (b, l), (b, m), (c, l), (c, m), (c, n)\}$

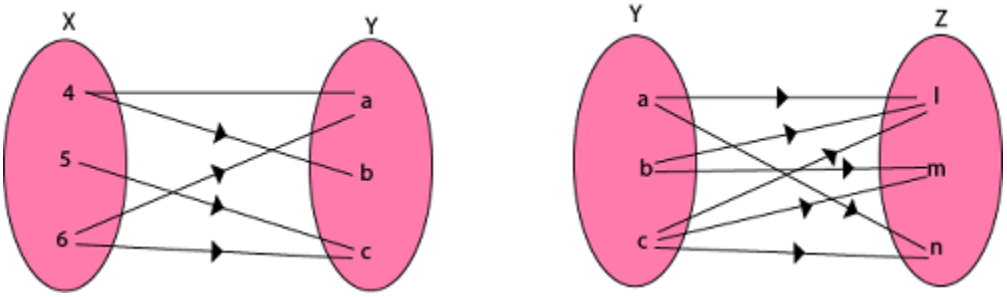


Fig 3: Example

Find the composition of relation (i) R_1 o R_2 (ii) R_1o R_1^{-1}

Solution:

(i) The composition relation R_1 o R_2 as shown in fig:

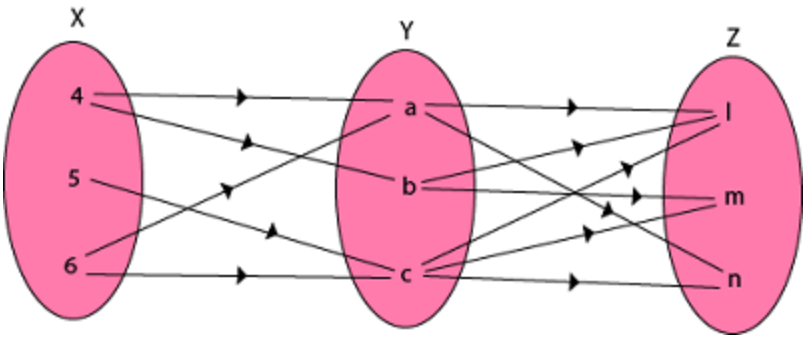


Fig 4: R_1 o R_2

R_1 o R_2 = {(4, l), (4, n), (4, m), (5, l), (5, m), (5, n), (6, l), (6, m), (6, n)}

(ii) The composition relation R_1o R_1^{-1} as shown in fig:

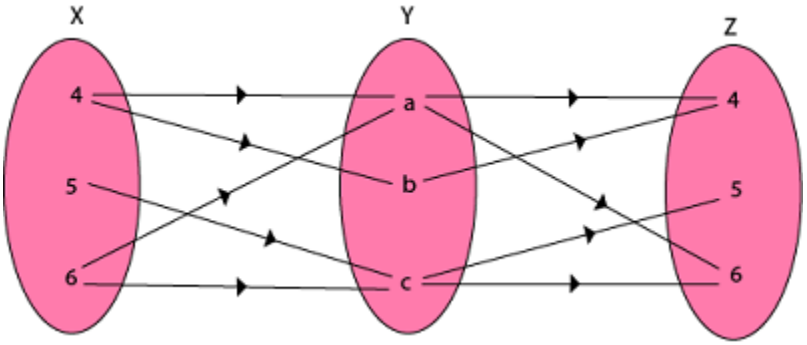


Fig 5: R_1 o R^{-11}

R_1o R_1^{-1} = {(4, 4), (5, 5), (5, 6), (6, 4), (6, 5), (4, 6), (6, 6)}

Key takeaway

1. Let A, B, and C be sets, and let R be a relation from A to B and let S be a relation from B to C.
2. That is, R is a subset of A × B and S is a subset of B × C.

1.8 Equality of relations

If a relation R on a set A satisfies the following three criteria, it is termed an equivalence relation:

1. Relation R is Reflexive, i.e., aRa ∀ a∈A.
2. Relation R is Symmetric, i.e., aRb ⟹ bRa
3. Relation R is transitive, i.e., aRb and bRc ⟹ aRc.

Example: Let A = {1, 2, 3, 4} and R = {(1, 1), (1, 3), (2, 2), (2, 4), (3, 1), (3, 3), (4, 2), (4, 4)}.

Demonstrate that R is a Relation of Equivalence.

Solution:

Reflexive: (1, 1), (2, 2), (3, 3) and (4, 4) ∈ R are all reflexive relationships.

Symmetric: Relation R is symmetric because whenever (a, b) ∈ R, (b, a) also belongs to R.

Example: (2, 4) ∈ R ⟹ (4, 2) ∈ R.

Transitive: R is transitive because anytime (a, b) and (b, c) are members of R, (a, c) is also a member of R.

Example: (3, 1) ∈ R and (1, 3) ∈ R ⟹ (3, 3) ∈ R

R is an Equivalence Relation since it is reflexive, symmetric, and transitive.

1.9 Recursive definition of relation

A relation R on a set A is reflexive if aRa for every a ∈ A, that is, if (a, a) ∈ R for every a ∈ A. Thus, R is not reflexive if there exists a ∈ A such that (a, a) ∈/ R.

Example: Consider the following five relations on the set A = {1, 2, 3, 4}:

R1 = {(1, 1), (1, 2), (2, 3), (1, 3), (4, 4)}

R2 = {(1, 1) (1, 2), (2, 1), (2, 2), (3, 3), (4, 4)}

R3 = {(1, 3), (2, 1)}

R4 = ∅, the empty relation

R5 = A × A, the universal relation.

Determine which of the relationships is reflexive.

Because A contains the four elements 1, 2, 3, and 4, any relation R on A that contains the four pairings (1, 1), (2, 2), (3, 3), and (4, 4) is reflexive (4, 4). R2 and the universal relation R5 = A × A are hence the sole reflexive functions. R1, R3, and R4 are not reflexive, because (2, 2), for example, does not belong to any of them.

1.10 Order of relations

If a relation R on a set A has the following three criteria, it is termed a partial order relation:

1. Relation R is Reflexive, i.e., aRa ∀ a∈A.
2. Relation R is Antisymmetric, i.e., aRb and bRa ⟹ a = b.
3. Relation R is transitive, i.e., aRb and bRc ⟹ aRc.

Example: Show that the relation (x, y) ∈ R is a partial order relation if x ≥ y are specified on the set of +ve integers.

Solution: Consider the four +ve integers in the set A = 1, 2, 3, 4. Determine a relationship for this collection, such as R = {(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3), (1, 1), (2, 2), (3, 3), (4, 4)}.

Reflexive - The relation is reflexive as for every a ∈ A. (a, a) ∈ R, i.e. (1, 1), (2, 2), (3, 3), (4, 4) ∈ R.

Antisymmetric - The relationship is antisymmetric since a = b whenever (a, b) and (b, a) ∈ R.

Transitive - Because we have (a, b) and (b, c) ∈ R whenever, we have (a, c) ∈ R.

Example - (4, 2) ∈ R and (2, 1) ∈ R, implies (4, 1) ∈ R.

Because the relationship is reflexive, anti symmetrical, and transitive, it is reflexive, anti symmetrical, and transitive. As a result, it's a partial order relationship.

1.11 Functions: Definition, Classification of functions, Operations on functions

It's a mapping in which each element in set A is linked to a specific element in set B. The domain of a function is the set of A, and the domain of B is the set of B.

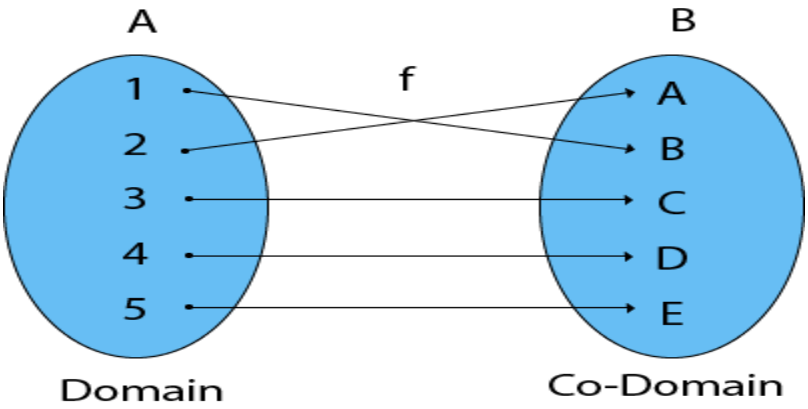


Fig 6: Domain and co-domain

Domain, Co-domain, and Range of function

Domain - Let f be a function that goes from point P to point Q. The domain of the function f is the set P.

Co-domain - Let f be a function that goes from point P to point Q. The set Q is referred to as the function f's Co-domain.

Range - The collection of pictures in a function's domain is its range. To put it another way, it is a subset of its co-domain. The letter f is used to represent it (domain).

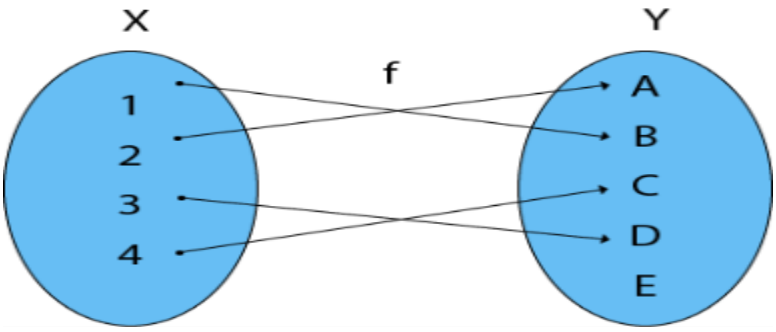
If f: P → Q, then f (P) = {f(x): x ∈ P} = {y: y ∈ Q | ∃ x ∈ P, such that f (x) = y}.

Example: Find the function's Domain, Co-Domain, and Range.

Let x = {1, 2, 3, 4}

y = {a, b, c, d, e}

f = {(1, b), (2, a), (3, d), (4, c)



Solution: Domain of function: {1, 2, 3, 4}

Range of function: {a, b, c, d}

Co-Domain of function: {a, b, c, d, e}

Classification of functions

1. **Injective (One-to-One) Functions:** One element of the Domain Set is associated to one element of the Co-Domain Set in this function.

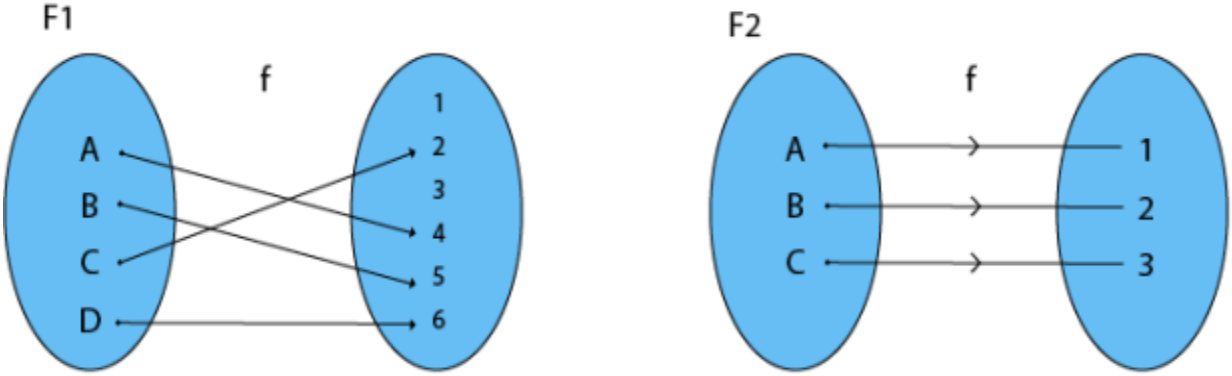


Fig 7: f1 and f2 show one to one function

2. **Surjective (Onto) Functions:** Every element of the Co-Domain Set has one pre-image in this function.

Example: Consider, A = {1, 2, 3, 4}, B = {a, b, c} and f = {(1, b), (2, a), (3, c), (4, c)}.

Because every element of B is the image of some A, it is a Surjective Function.

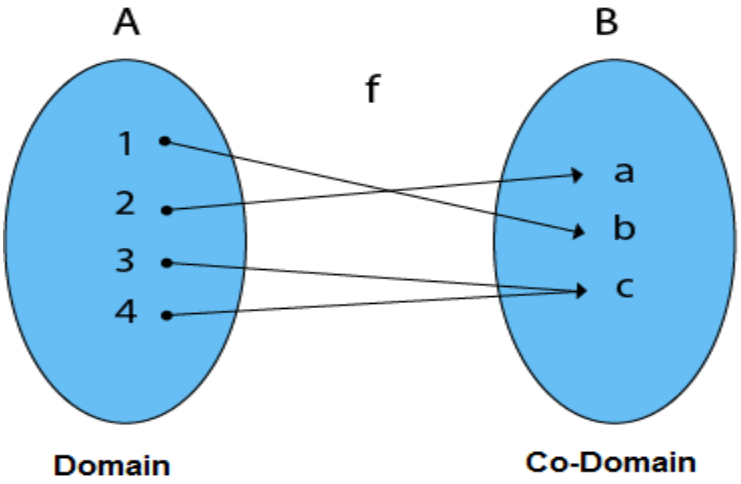


Fig 8: Surjective function

3. **Bijective (One-to-One Onto) Functions:** Bijective (One-to-One Onto) Function is a function that is both injective (one to one) and surjective (onto).

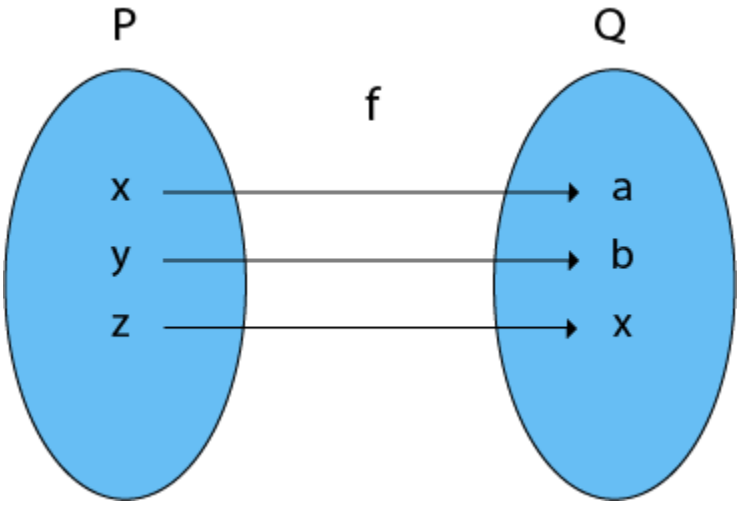


Fig 9: One to one function

Example: Consider P = {x, y, z}

Q = {a, b, c}

And f: P → Q such that

f = {(x, a), (y, b), (z, c)}

The f is both a one-to-one and an onto function. It is, thus, a bijective function.

4. **Into Functions:** There is no pre-image in domain X for a function that requires an element from co-domain Y.

Example: Consider, A = {a, b, c}

B = {1, 2, 3, 4} and f: A → B such that

f = {(a, 1), (b, 2), (c, 3)}

In the function f, the range i.e., {1, 2, 3} ≠ co-domain of Y i.e., {1, 2, 3, 4}

As a result, it is an into function.

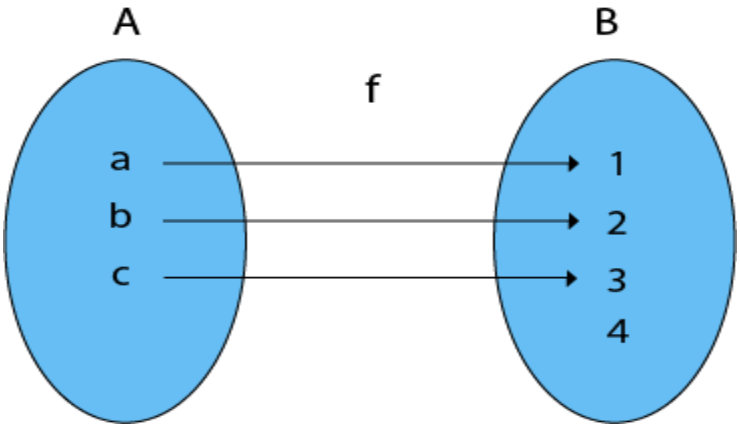


Fig 10: Into function

5. **One-One Into Functions:** Let f: X →Y be the case. If distinct elements of X have different unique images of Y, the function f is called one-one into function.

Example: Consider, X = {k, l, m}

Y = {1, 2, 3, 4} and f: X → Y such that

f = {(k, 1), (l, 3), (m, 4)}

The function f is a one-to-one conversion.

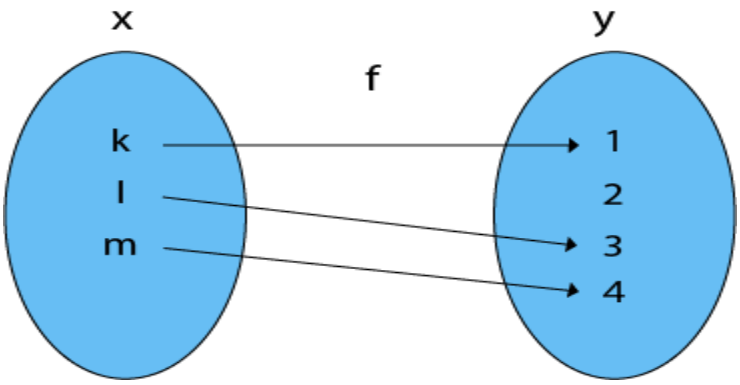


Fig 11: One one into function

6. **Many-One Functions:** Let f: X →Y be the case. If there are two or more separate items in X that have the same image in Y, the function f is said to be a many-one function.

Example: Consider X = {1, 2, 3, 4, 5}

Y = {x, y, z} and f: X → Y such that

f = {(1, x), (2, x), (3, x), (4, y), (5, z)}

The function f is a many to one

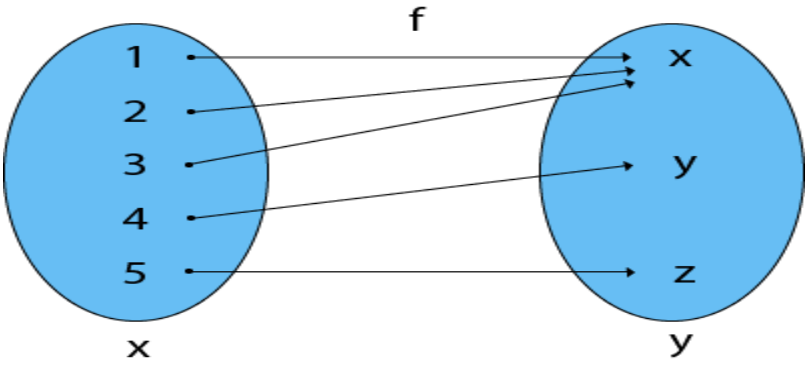


Fig 12: Many to one function

7. **Many-One Into Functions:** Let f: X→Y be the case. If and only if is both many one and into function, the function f is termed the many-one function.

Example: Consider X = {a, b, c}

Y = {1, 2} and f: X → Y such that

f = {(a, 1), (b, 1), (c, 1)}

Because f is a many-one and into function, it is also a many-one into function.

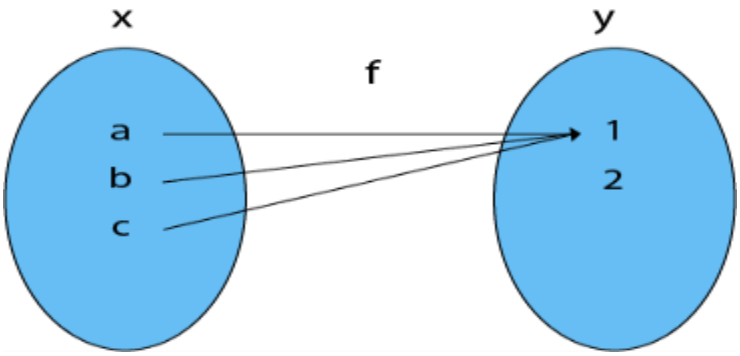


Fig 13: Many one into function

8. **Many-One Onto Functions:** Let f: X→ Y be the case. If and only if is both many one and onto, the function f is termed a many-one onto function.

Example: Consider X = {1, 2, 3, 4}

Y = {k, l} and f: X → Y such that

f = {(1, k), (2, k), (3, l), (4, l)}

The function f is a many-one (since the two elements in Y have the same image) and onto (as every element of Y is the image of some element X). As a result, it is a many-to-one function.

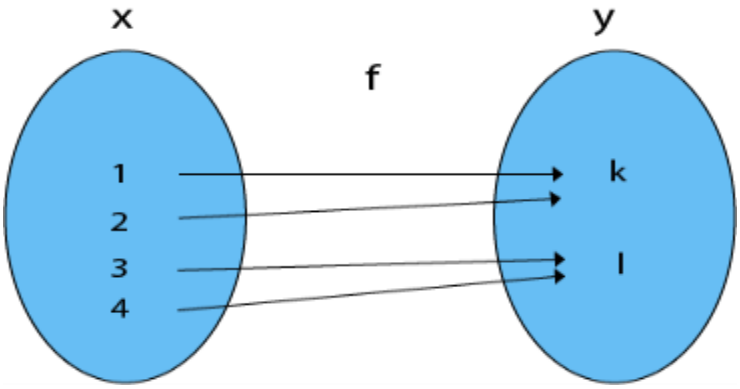


Fig 14: Many one onto function

Key takeaway

1. It's a mapping in which each element in set A is linked to a specific element in set B.
2. The domain of a function is the set of A, and the domain of B is the set of B.

1.12 Recursively defined functions

If the function definition refers to itself, it is considered to be recursively defined. The function definition must contain the following two properties in order to avoid being circular:

(1) The function must not refer to itself for specific parameters, referred to as base values.

(2) The function's argument must be closer to a base value each time the function refers to itself.

The term "well-defined" refers to a recursive function that has these two features.

The examples that follow should assist to understand these concepts.

Factorial Function

The product of positive numbers from 1 to n, inclusive, is known as "n factorial" and is generally written as n! That is to say,

$n! = n\,(n-1)\,(n-2)\cdots 3\cdot 2\cdot 1$

It's also handy to specify 0! = 1 to ensure that the function works for all nonnegative numbers. Thus:

$0! = 1,\ 1! = 1,\ 2! = 2\cdot 1 = 2,\ 3! = 3\cdot 2\cdot 1 = 6,\ 4! = 4\cdot 3\cdot 2\cdot 1 = 24\ 5! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120,\quad 6! = 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1 = 720$

And so forth. Take note of this:

$5! = 5 \cdot 4! = 5 \cdot 24 = 120$ and $6! = 6 \cdot 5! = 6 \cdot 120 = 720$

This holds for any positive integer n; in other words,

$n! = n \cdot (n - 1)!$

As a result, the factorial function can be written as follows.

Factorial function: (a) If $n = 0$, then $n! = 1$.

(b) If $n > 0$, then $n! = n \cdot (n - 1)!$

It's worth noting that the previous definition of n! is recursive, as it uses (n 1)! to refer to itself. However:

(1) When $n = 0$, the value of n! is explicitly stated (thus 0 is a base value).

(2) For each random n, the value of n! is defined in terms of a smaller n that is closer to the base value 0.

Key takeaway

1. If the function definition refers to itself, it is considered to be recursively defined.
2. The function definition must contain the following two properties in order to avoid being circular.

1.13 Growth of Functions

The highest order term determines the rate of expansion of a function: if you add a bunch of terms, the function expands about as quickly as the largest term (for large enough input values).

The big-O notation is commonly used to describe the expansion of functions.

Let f and g be functions from integers or real numbers to real numbers, respectively.

If there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$, we say f(x) is O(g(x)).

f(x) and g(x) are always positive when we look at the evolution of complexity functions. As a result, we can reduce the big-O requirement to $f(x) \leq C$.

When $x > k$, use g(x).

We simply need to identify one pair (C, k) to demonstrate that f(x) is O(g(x) (which is never unique).

The big-O notation is used to define an upper limit for the growth of a function f(x) for large x.

This border is defined by the g(x) function, which is typically much simpler than f. (x).

Because C does not grow with x, we accept the constant C in the criterion $f(x) \leq C.g(x)$ whenever $x > k$.

We only care about large x, thus it's fine if $f(x) > C.g(x)$ for $x \leq k$.

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

For $x > 1$ we have:

$x^2 + 2x + 1 \leq x^2 + 2x^2 + x2$

$x^2 + 2x + 1\ 4x^2$

Therefore, for $C = 4$ and $k = 1$:

$f(x) \leq Cx^2$ whenever $x > k$.

$f(x)$ is $O(x^2)$.

Key takeaway

The highest order term determines the rate of expansion of a function: if you add a bunch of terms, the function expands about as quickly as the largest term (for large enough input values).

1.14 Natural Numbers: Introduction

The nonnegative integers, N=0, 1, 2, 3..., are the natural numbers. Create definitions for the following integer relations using the concept of natural numbers: less than (), less than or equal to (), greater than (>), and greater than or equal to ().

Note that several authors define natural numbers as only positive integers; zero is not a natural number for them. This appears unnatural to me. The terms positive integers and nonnegative integers are unmistakable and well-understood among mathematicians. However, the term "natural number" is not completely standardized.

Set of natural numbers

A collection of elements is referred to as a set (numbers in this context). In mathematics, the set of natural numbers is written as 1,2,3, ... The set of natural numbers is symbolized by the symbol N. N = 1,2,3,4,5,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

Statement Form	N = Set of all numbers starting from 1.

Roaster Form	N = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ……}
Set Builder Form	N = {x : x is an integer starting from 1}

Smallest natural number

1 is the smallest natural number. We know that the smallest element in N is 1 and that we can talk about the next element in terms of 1 and N for any element in N. (which is 1 more than that element). Two is one greater than one, three is one greater than two, and so on.

Natural number from 1 to 100

The natural numbers from 1 to 100 are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99 and 100.

Key takeaway

1. The nonnegative integers, N=0, 1, 2, 3..., are the natural numbers.
2. Create definitions for the following integer relations using the concept of natural numbers: less than (), less than or equal to (), greater than (>), and greater than or equal to ().
3. In mathematics, the set of natural numbers is written as 1,2,3, ... The set of natural numbers is symbolized by the symbol N. N = 1,2,3,4,5,,,,,,,,,

1.15 Mathematical Induction

Mathematical induction is a mathematical technique is used to prove a statement, a theorem or a formula is true for every natural number.

The technique has two steps to prove any statement-

1. Base step (it proves that a statement is true for the initial value)

2. Inductive step (it proves that is the statement is true for n'th iteration then it is also true for $(n+1)^{th}$ iteration.

Example-1: $\mathbf{1 + 3 + 5 + \ldots + (2n - 1) = n^2}$ **for** $\mathbf{n = 1,2,\ldots\ldots}$

Sol. Here for n = 1, $1 = 1^2$

Now let us assume that the statement is true for n = k

Hence, we assume that $1 + 3 + 5 + \ldots + (2k - 1) = k^2$ is true.

Now we need to prove that $1 + 3 + 5 + \ldots + [2(k + 1) - 1] = (k + 1)^2$

$$1 + 3 + 5 + \ldots + [2(k + 1) - 1]$$
$$= 1 + 3 + 5 + \ldots + [2k + 2 - 1]$$
$$= 1 + 3 + 5 + \ldots + [2k + 1]$$
$$= 1 + 3 + 5 + \ldots + (2k - 1) + (2k + 1)$$
$$= k^2 + (2k + 1)$$
$$= (k + 1)^2$$

So that $1 + 3 + 5 + \ldots + [2(k + 1) - 1] = (k + 1)^2$ which satisfies the second step.

Hence-

$$1 + 3 + 5 + \ldots + (2n - 1) = n^2$$

Example-2: Prove 1+2+...+n = n(n+1)/2 using a proof by induction

Sol.

Let n=1: 1 = 1(1+1)/2 = 1(2)/2 = 1 is true,

Step-1: Assume n=k holds: 1+2+...+k = k(k+1)/2

Show n=k+1 holds: 1+2+...+k+(k+1) =(k+1) ((k+1) +1)/2

Substitute k with k+1 in the formula to get these lines. Notice that I write out what I want to prove.

Step-2: Now start with the left side of the equation

1+2+...+(k+1) =1+2+...+k+(k+1)

=k(k+1)/2 + (k+1)

=(k(k+1)+2(k+1))/2 by 2/2=1 and distridution of division over addition

=(k+2)(k+1)/2 by distribution of multiplication over addition

=(k+1)(k+2)/2 by commutativity of multiplication

Example-3: Prove the following by using the principle of mathematical induction for all n ∈ N-

$1.2 + 2.2^2 + 3.2^2 + \ldots + n.2^n = (n-1).2^{n+1} + 2$

Sol. Here, n = 1, $(1-1).2^{1+1} + 2 = 2$, which is true

Step-1: Assume n = k holds-
$1.2 + 2.2^2 + 3.2^2 + \ldots + k.2^k = (k-1).2^{k+1} + 2$

Now show n = k + 1 also holds-

Consider-

$$\{1.2 + 2.2^2 + 3.2^2 + \ldots + k.2^k\} + (k+1).2^{k+1}$$
$$= (k-1).2^{k+1} + 2 + (k+1).2^{k+1}$$
$$= 2^{k+1}\{(k-1) + (k+1)\} + 2$$
$$= 2^{k+1}.2k + 2$$
$$= k.2^{(k+1)+1} + 2$$
$$= \{(k+1) - 1\}2^{(k+1)+1} + 2$$

Which is also true for n = k + 1

Hence proved.

1.16 Variants of Induction

Strong induction

Another type of mathematical induction is strong induction. We can use the following procedures to prove that a propositional function, P(n), is true for all positive integers, n, using this induction technique:

Step 1(Base step) - It establishes the truth of the original statement P (1).

Step 2 (Inductive step) - It proves that the conditional statement

$[P(1) \wedge P(2) \wedge P(3) \wedge \cdots \wedge P(k)] \rightarrow P(k+1)$ is true for positive integers k.

1.17 Induction with Nonzero Base cases

Prove the following by Mathematical Induction:

1 + 3 + 5 +.... + 2n - 1 = n2.

Solution: let us assume that.

P (n) = 1 + 3 + 5 +..... + 2n - 1 = n2.

For n = 1, P (1) = 1 = 12 = 1

It is true for n = 1..... (i)

Induction Step: For n = r,

P (r) = 1 + 3 + 5 +..... +2r-1 = r2 is true..... (ii)

Adding 2r + 1 in both sides

P (r + 1) = 1 + 3 + 5 +..... +2r-1 + 2r +1

= r2 + (2r + 1) = r2 + 2r +1 = (r+1)2..... (iii)

As P(r) is true. Hence P (r+1) is also true.

From (i), (ii) and (iii) we conclude that.

1 + 3 + 5 +..... + 2n - 1 =n2 is true for n = 1, 2, 3, 4, 5 ....

Hence Proved.

1.18 Proof Methods, Proof by counter – example

A mathematical proof is a logical argument used to support a mathematical proposition. We evaluate two elements while validating a statement: the statement and the logical operators.

Proof by cases

A statement can be true or untrue, but not both at the same time. AND, OR, NOT, If then, and If and only if are logical operators. They exist when combined with quantifiers like for all. To ensure that the assertion is correct, we apply operators to it.

This method involves evaluating each scenario of the statement to determine its truthfulness.

Example: The integer x(x + 1) is even for every integer x.

Proof: If x is an even number, then x = 2k for some k. The statement now reads:

2k (2k + 1)

It is even because it is divisible by two.

If x is odd, the sentence becomes: x = 2k + 1 for some number k.

(2k+1) (2k+1+1) = (2k + 1) 2(k + 1)

We established that x(x+1) is even in both circumstances since it is divisible by 2.

Proof by induction

The Mathematical Induction Principle (PMI). P(n) represents a statement about the positive integer n. If the following statements are correct:

1. P (1),

2. (for all n there exists Z+) P(n) implies P (n + 1),

Then (for all n there exists Z+) P(n).

For each positive integer n, consider the following example.

1 + 2 +⋯+ n = n (n + 1)/ 2

Proof: If n = 1, this is the base case.

1 + ⋯ + n = 1

And

n (n + 1)/2 = 11

Inductive step: Assume that there is a Z+ for a given n.

1 + 2 +⋯+ n = n (n + 1)/ 2 ---- (i) (inductive hypothesis)

Our purpose is to demonstrate that:

1 + 2 +⋯+ n + (n + 1) = [n + 1] ([n + 1] + 1)/ 2

i.e., 1 + 2 +⋯+ n + (n + 1) = (n + 1) (n + 2) /2

When both sides of equation I are multiplied by n+1, we get

1 + 2 +⋯+ n + (n + 1)

= n (n + 1)/ 2 + (n + 1)

= n (n + 1) /2 + 2(n + 1) /2

= (n + 2) (n + 1) /2

1.19 Proof by contradiction

In logic and mathematics, proof by contradiction is a method of determining the truth of a statement by assuming it is false, then trying to show it is incorrect until the conclusion of that assumption is a contradiction.

Proof by contradiction: Steps

Let's break it down into steps to better understand how proof by contradiction works. When employing evidence by contradiction, we follow these steps:

- Assume that your statement is incorrect.
- Proceed as if you were working on a direct proof.
- You've come across an inconsistency.
- Declare that, because of the contradiction, the statement cannot be wrong, hence it must be true.

Example

Do you recall what I said earlier?

No integers y and z exist for which

$24y + 12z = 1$

You may spend days, weeks, or even years wandering around with precise numbers to demonstrate that any integer you try in the sentence works.

For example, try y = -5, and z = 7.

$24 \times -5 + 12 \times 7 = 1$

$-120 + 84 = -36$

Those integers didn't work, so how about repeating it again with a few hundred more integers for a few hours? Most likely not. However, we can try to disprove the statement by using evidence by contradiction:

No integers a and b exist for which

$24y + 12z = 1$

To show that this is wrong, we assume that we can find numbers y and z to solve the equation:

$24y + 12z = 1$ the original equation

By multiplying both sides by 12, the biggest common factor emerges.

$2y + z = 1/12$

The effect caused by dividing both sides by the greatest common factor of the two integers strikes us right away. The total number of integers is a fraction!

Key takeaway

1. In logic and mathematics, proof by contradiction is a method of determining the truth of a statement by assuming it is false, then trying to show it is incorrect until the conclusion of that assumption is a contradiction.

Unit - 2
Algebraic Structures

2.1 Definition

An algebraic structure is a non-empty set G that contains one or more binary operations. Let's pretend that * is a binary operation on G. The structure (G, *) is then algebraic. The algebraic structure is represented by (N,*), (1, +), and (1, -). (R, +,.) is an algebraic structure with two operations in this case.

Example:

1. Let R be the set of real numbers, then (R, +) is an algebraic structure.
2. If N denotes the set of natural numbers then (N, +) is an algebraic structure.

Binary operation

A binary operation * in a set A is a function from A × A to A.

If * is a binary operation in a Set A then than for the * image of the ordered pair (a, b) $\in$ A × A, we write a * b (or * (a, b)).

Note-

1. Addition (+) is a binary operation in the set of natural number N. Set of integers Z and set of real numbers R.

2. Multiplication (×) is a binary operation in N, Z, Q, R and C.

Property of algebraic structure

A property of an algebraic structure is one that any of its operations possesses. The following are some of the most important properties of an algebraic system:

1. Associative and commutative laws

If any of the members a, b, c in S meet the associative law, then an operation * on the set is said to be associative or to satisfy the associative law (a * b) * c = a * (b * c).

Note- The operations of addition and multiplication over the natural numbers are associative.

If a * b = b * a for any element a, b in S, the operation * is said to be commutative or meet the commutative law.

Note- Addition is commutative in the set of natural numbers.

2. Identity element and inverse

Consider the following operation * on the set S. If any element an in S - a * e = e * a = a, an element e in S is called an identity element.

Note- Zero is the identity element in the set integers with respect to the binary operation addition (i.e., +).

A left identity or a right identity for an element e is defined as e *a or a * e = a, where an is any element in S.

Let's pretend that an operation * on a set S has an identity element e. An element b is the inverse of an element in S, such that a * b = b * a = e.

3. Cancellation laws

A binary operation denoted by * in a set A, is said to satisfy

1. Left cancellation law if for all a, b, c $\in$ A,

a * b = a * c $\Rightarrow$ b = c

2. Right cancellation law if for all a, b, c $\in$ A

b * a = c * a $\Rightarrow$ b = c

For example:

In P (A), the power set of A, union is distributive over intersection and intersection is distributive of union of sets.

Theorem: If * is an associative binary operation in a set A, such every element is invertible, then * satisfies the left as well as the right cancellation laws, which means

$\mathbf{a * b = a * c \Rightarrow b = c}$

$\mathbf{b * a = c * a \Rightarrow b = c \ \ for\ every\ a,b,c \in A}$

Proof:

Suppose e be the identity element of A with respect to *

Every element in A is invertible $\Rightarrow$ a $\in$ A is invertible.

Let a' denotes the inverse of a in A then

$a * b = a * c$

$\Rightarrow a' * (a * b) = a' * (a * c)$

$\Rightarrow (a' * a) * b = (a' * a) * c)$ [* is associative]

$\Rightarrow e' * b = e * c$ [a'is the inverse of a]

$\Rightarrow b = c$

Similarly, it can be proved that

$b * a = c * a \Rightarrow b = c \quad for\ every\ a,b,c \in A$

Key takeaway

1. An algebraic structure is a non-empty set G that contains one or more binary operations.
2. A property of an algebraic structure is one that any of its operations possesses.

2.2 Groups

A group is an algebraic structure (G, *) in which the binary operation * on G satisfies the following conditions:

Condition-1: For all a, b, c, ∈G

a* (b * c) = (a * b) * c (associativity)

Condition-2: There exists an elements e ∈G such that for any a ∈G

a* e= e * a = a (existence of identity)

Condition-3: For every a ∈G, there exists an element denoted by a^{-1} in G such that

a* a^{-1} = a^{-1} * a = e

a^{-1} is called the inverse of a in G.

Example: (Z, +) is a group where Z denote the set of integers.

Example: (R, +) is a group where R denote the set of real numbers.

Abelian group-

Let (G, *) be a group. If * is commutative that is

a* b = b * a for all a, b ∈G then (G, *) is called an Abelian group.

Finite group-

A group G is said to be a finite group if the set G is a finite set.

Infinite group-

A group G, which is not finite is called an infinite group.

Order of a finite group-

The order of a finite group (G, *) is the number of distinct elements in G. The order of G is denoted by O (G) or by |G|.

Example: If G = {1, -1, i, -i} where i= $\sqrt{-1}$, then show that G is an abelian group with respect to multiplication as a binary operation.

Sol.

First, we will construct a composition table-

.	1	-1	i	-i
1	1	-1	i	-i

-1	**-1**	**1**	**-i**	**i**
i	**i**	**-i**	**-1**	**1**
-i	**-i**	**i**	**1**	**-1**

It is clear from the above table that algebraic structure (G,.) is closed and satisfies the following conditions.

Associativity- For any three elements a, b, c ∈G (a ·b) ·c = a·(b ·c)

Since

1·(−1 ·i) = 1 ·−i= −i

(1 ·−1) ·i= −1 ·i= −i

⇒1·(−1 ·i) = (1 ·−1) i

Similarly with any other three elements of G the properties hold.

∴Associative law holds in (G, ·)

Existence of identity: 1 is the identity element (G, ·) such that 1 ·a = a = a ·1 ∀a ∈G

Existence of inverse: 1 ·1 = 1 = 1 ·1 ⇒1 is inverse of 1

(−1)·(−1) = 1 = (−1) ·(−1) ⇒–1 is the inverse of (–1)

i·(−i) = 1 = −i·i⇒–iis the inverse of iin G.

−i·i= 1 = i·(−i) ⇒iis the inverse of –iin G.

Hence inverse of every element in G exists.

Thus, all the axioms of a group are satisfied.

Commutativity: a ·b = b ·a ∀a, b ∈G hold in G

1 ·1 = 1 = 1 ·1, −1 ·1 = −1 = 1 ·−1

i·1 = i= 1 ·i; i·−i= −i·i= 1 = 1 etc.

Commutative law is satisfied

Hence (G, ·) is an abelian group.

Example-

Prove that the set Z of all integers with binary operation * defined by a * b = a + b + 1 ∀a, b ∈G is an abelian group.

Sol: Sum of two integers is again an integer; therefore a +b ∈Z ∀a, b ∈Z

$\Rightarrow a + b + 1 \cdot \in Z \ \forall a, b \in Z$

$\Rightarrow Z$ is called with respect to *

Associative law for all a, b, a, b $\in G$ we have (a * b) * c = a * (b * c) as

(a* b) * c = (a + b + 1) * c

= a + b + 1 + c + 1

= a + b + c + 2

Also

a* (b * c) = a * (b + c + 1)

= a + b + c + 1 + 1

= a + b + c + 2

Hence (a * b) * c = a * (b * c) $\in$a, b $\in Z$.

Groupoid-

Groupoid is an algebraic structure consisting of non-empty set A and a binary operation *, which is such that A is closed under *.

For example- The set of real numbers is closed under addition, therefore (R, +) is a groupoid.

Theorem:

Statement 1- There is just one identification element in a Group G. (uniqueness of identity) Proof: - In G, let e and e' be two distinct identities, and let a $\in$ G

$\therefore$ ae = a $\longrightarrow$(i)

$\therefore$ ae' = a $\longrightarrow$(ii)

R.H.S of (i) and (ii) are equal $\Rightarrow$ae =ae'

As a result of the left cancellation law, we get e=e'.

For any G, there is only one identity element. As a result, the theorem is established.

Statement 2 - There is a unique element b in G for each element an in a group G, such that ab= ba=e (uniqueness if inverses)

Proof: Assume that b and c are the inverses of an a $\in$ G.

Then ab = e and ac = e

$\because$ c = ce {existence of identity element}

$\Longrightarrow$ c = c (ab) {$\because$ ab = e}

$\Longrightarrow$ c = (c a) b

$\Longrightarrow$ c = (ac) b {$\because$ ac = ca}

$\Longrightarrow$ c = eb

$\Longrightarrow$ c = b {$\because$ b = eb}

As a result, the inverse of a G is unique.

Key takeaway

1. A monoid having an inverse element is referred to as a group.
2. The order of a group G is equal to the number of members in G, and the order of an element in a group is equal to the least positive integer n such that an is the group G's identity element.

2.3 Subgroups and order

Let (G, *) be a group and H, be a non-empty subset of G. If (H, *) is itself is a group, then (H, *) is called sub-group of (G, *).

Example-Let a = {1, –1, i, –i} and H = {1, –1}

G and H are groups with respect to the binary operation, multiplication.

H is a subset of G, therefore (H, X) is a sub-group (G, X).

Theorem-

If (G, *) is a group and H ≤G, then (H, *) is a sub-group of (G, *) if and only if

(i) a, b ∈H ⇒a * b ∈H;

(ii) a∈H ⇒a^{-1} ∈H

Proof:

If (H, *) is a sub-group of (G, *), then both the conditions are obviously satisfied.

We, therefore prove now that if conditions (i) and (ii) are satisfied then (H, *) is a sub-group of (G, *).

To prove that (H, *) is a sub-group of (G, *) all that we are required to prove is : * is associative in H and identity e ∈H.

That * is associative in H follows from the fact that * is associative in G.

Also,

a∈H ⇒a^{-1} ∈H by (ii) and e ∈H and a^{-1} ∈H ⇒a * a^{-1} = e ∈H by (i)

Hence, H is a sub-group of G.

Cyclic subgroup

A cyclic subgroup K of a group G exists if an element x ∈ G exists such that every element of K can be expressed in the form x^n for some n ∈Z.

The element x is referred to as the K generator, and we write K= <x>.

Order

The number of components in the group G determines its order. |G| is the symbol for it. Only the identity element, (e *), is present in an order 1 group.

A group of order 2 consists of two members, one of which is the identity element and the other of which is some other element.

Definition (order of an element)

Let (G, *) be a group and a $\in$ G, then the least positive integers, n if it exists such that an = e is called the order of a $\in$ G.

The order of an element a $\in$ G is be denoted by O (a).

Note-

The order of every element of a finite group is finite.

The order of an element of a group is always equal to the order of its inverse.

The order of any integral power of any element of an group is less then or equal to the order of the element.

Example 1: Consider the order 2 group (e, x, *). The operation table is displayed.

*	E	x
e	E	x
x	X	e

Three items make up the order 3 group: one identity element and two other elements.

Key takeaway

1. H is a subgroup of G if a non-void subset H of a group G is itself a group under the operation of G.
2. The number of components in the group G determines its order. |G| is the symbol for it.
3. Only the identity element, (e *), is present in an order 1 group.

2.4 Cyclic Groups

A cyclic group is one that can be made up of only one element. Every member of a cyclic group is a power of a generator, which is a specific element. A generator 'g' can create a cyclic group in which every other element of the group can be represented as a power of the generator 'g'.

When G=, we say that G is cyclic and that x is a generator of G. That is, if there is an element x ∈ G such that every element of G can be written in the form x^n for any n ∈ Z, the group G is said to be cyclic.

Under ordinary multiplication, the group G=1, -1, i, -i is a finite cyclic group with I as generator, thus i^1=i, i^2=-1, i^3=-i and i^4=1

Although every cyclic group is an abelian group, not every abelian group is cyclic. The rational numbers under addition are abelian, not cyclic.

Example: Let Z denote the set of integers (Z, +) is an infinite cyclic group 1 and –1 are the generators of the group (Z, +).

Abelian group

Consider the algebraic system (G,*), in which * represents a binary operation on G. The system (G, *) is therefore considered to be an abelian group if it satisfies all of the group's properties plus the following property:

(1) The operation * is commutative i.e.,

a * b = b * a ∀ a, b ∈G

Example - Consider the algebraic system (G, *), in which G represents the set of all non-zero real numbers and * represents a binary operation defined by

$$a * b = \frac{ab}{4}$$

Demonstrate that the group (G, *) is an abelian one.

Solution:

Closure property - Because a * b $= \frac{ab}{2}$ Subgroup is a real number, the set G is closed by the operation *. As a result, it belongs to G.

Associative property - * is an associative operation. If we use a, b, c∈G, we get

$$(a * b) * c = \left(\frac{ab}{4}\right) * c = \frac{(ab)c}{16} = \frac{abc}{16}$$

Similarly,

$$a * (b * c) = a * \left(\frac{bc}{4}\right) = \frac{a(bc)}{16} = \frac{abc}{16}$$

Identity - Assume that e is a +ve real number to get the identity element. Then e * a = a, with a being G.

$\frac{Ea}{4} = a$ or $e = 4$

Similarly,

$a * e = a$

$\frac{Ae}{4} = a$ or $e = 4$

As a result, G's identification element is 4.

Inverse - Let's pretend that a ∈G. If the inverse of an is $a^{-1} \in Q$, then $a * a^{-1=4}$.

Therefore $\frac{aa^{-1}}{4} = 4$ or $a^{-1} = \frac{16}{a}$

Similarly, $a^{-1} * a = 4$

Therefore, $\frac{a^{-1}a}{4} = 4$ or $a^{-1} = \frac{16}{a}$

As a result, in G, the inverse of element an is $\frac{16}{a}$.

Commutative - * on G is a commutative operation.

Since $a * b = \frac{ab}{4} = b * a$

G, * is thus a closed, associative, identity element, inverse, and commutative algebraic system. As a result, the system (G, *) belongs to the abelian group.

Theorem: Every cyclic group is abelian

Proof:

Suppose (G, *) be a cyclic group

Generated by a

$x, y \in G$

$\Rightarrow x = a^m$ and $y = a^n$ for some integers

m and n

$x * y = a^m * a^n$

$= a^{m+n}$

$= a^{n+m}$

$= a^n * a^m = y * x$

Theorem- Every group of prime order is cyclic.

Proof:

Let O (G) = p where p is a prime number and

Suppose

$a \neq e \in G$

Consider the sub-group of G, generated by a

Let H = <a>

$\Rightarrow$ O (H) > 1

H is a sub-group of G

By Lagrange's theorem

O (H) / O (G) $\Rightarrow$ O (H) / p

$\Rightarrow$ O (H) = 1 or p

$\Rightarrow$ O (H) = p since O (H) $\neq 1$

O (H) = O (G)

But H is cyclic $\Rightarrow$ G is cyclic

Note- Every proper sub-group of an infinite cyclic group is isomorphic to the group itself.

Every proper sub-group of finite cyclic group is a finite cyclic group.

The only groups which do not possess proper sub-group are the prime order finite sub-groups.

Key takeaway

1. A cyclic group is one that can be made up of only one element.
2. Every member of a cyclic group is a power of a generator, which is a specific element.

2.5 Cosets

Assume that H is a subgroup of G. For each x G, a left coset of H in G is a subset of G whose elements may be represented as xH= {xh | h ∈ H}. A representation of the coset is the element x. Similarly, for any x∈ G, a right coset of H in G is a subset that can be represented as Hx= {hx | h∈ H}. As a result, the complexes xH and Hx are referred to as a left coset and a right coset, respectively.

A left coset is denoted by x + H= {x+h | h ∈ H} if the group operation is additive (+), while a right coset is denoted by H + x = {h+x | h ∈ H} if the group operation is subtractive (-).

If G is a finite group, H is a subgroup of G, and g is an element of G, then in group theory;

The left coset of H in G with regard to the element of G is gH = gh: h an element of H.

And

The right coset of H in G with regard to the element of G is Hg = hg: h an element of H.

Theorem: If (H, *) is a sub-group (G, *), then a * H = H if and only if a ∈H.

Proof: Let a * H = H

Since e ∈H then a = a * e ∈a * H

Hence a ∈H

Conversely

Let a ∈H then a * H ⊆H

(H, *) is a sub-group.

∴a∈H, h ∈H ⇒a^{-1} * h ∈H.

Now h ∈H

⇒h = a * (a^{-1} * h) ∈a * H

∴h∈H ⇒h ∈a * H

⇒H ⊆a * H

Hence a * H = H

Let us now consider the lemmas that aid in the proof of the Lagrange theorem.

Lemma 1: If G is a group with a subgroup H, then H and any coset of H have a one-to-one relationship.

Lemma 2: If G is a group with subgroup H, then the left coset relation, g1∼g2, is an equivalence relation if and only if g1 ∗ H = g2 ∗ H.

Let S be a set and be an equivalence relation on S. Lemma 3: Let S be a set and be an equivalence relation on S. If A and B are two equivalence classes, then A ∩ B =∅,

A = B.

Key takeaway

1. Assume that H is a subgroup of G.
2. For each x G, a left coset of H in G is a subset of G whose elements may be represented as xH= {xh | h ∈ H}.
3. A representation of the coset is the element x.

4. Similarly, for any $x \in G$, a right coset of H in G is a subset that can be represented as $Hx = \{hx \mid h \in H\}$.

2.6 Lagrange's theorem

One of the most important theorems in abstract algebra is the Lagrange theorem. It asserts that in group theory, the order of subgroup H of group G splits the order of G for any finite group G. The number of elements is represented by the group's order. Joseph-Louis Lagrange proved this theorem. Let us look at the statement and proof of the Lagrange theorem in Group theory, as well as the three lemmas that were utilized to prove this theorem with examples.

Lagrange Theorem Statement

The order of the subgroup H separates the order of the group G, according to the assertion. This can be written as:

$|G| = |H|$

The important terminologies and three lemmas that aid in the proof of the Lagrange theorem must be understood before proceeding with the proof.

Lagrange Theorem Proof

We may simply prove the Lagrange statement using the three lemmas presented before.

Proof of Lagrange Statement:

Let H be any subgroup of a finite group G of order m of order n. Take a look at G's cost breakdown in relation to H.

Let us now imagine that each coset of aH consists of n distinct items.

Let $H = \{h_1, h_2 \ldots, h_n\}$, then $ah_1, ah_2, \ldots, ah_n$ are the n distinct members of aH.

Suppose, $ah_i = ah_j \Rightarrow h_i = h_j$ be the cancellation law of G.

Because G is a finite group, there will be a finite number of discrete left cosets, say p. As a result, m=np since the total number of elements in all cosets equals np, which is equal to the total number of elements in G.

$p = m/n$

This proves that n, H's order, is a divisor of m, the finite group G's order. We can also observe that the index p is a divisor of the group's order.

Hence, proved, $|G| = |H|$

Key takeaway

1. One of the most important theorems in abstract algebra is the Lagrange theorem.
2. It asserts that in group theory, the order of subgroup H of group G splits the order of G for any finite group G.
3. The number of elements is represented by the group's order. Joseph-Louis Lagrange proved this theorem.

2.7 Normal Subgroups

Let G stand for a group. If for any $h \in H$ and $x \in G$, $x\ h\ x^{-1} \in H$, a subgroup H of G is said to be a normal subgroup of G.

If $x\ H\ x^{-1} = \{x\ h\ x^{-1} | \ h \in H\}$ then H is normal in G if and only if $xH\ x^{-1} \subseteq H$, $\forall\ x \in G$.

Note- (G, *) is normal in (G, *). It is called the improper normal sub-graph of G.

Statement - Every subgroup H of G is normal in G if G is an abelian group.

Proof:

Allow any $h \in H$, $x \in G$, and then

$x\ h\ x^{-1} = x\ (h\ x^{-1})$

$x\ h\ x^{-1} = (x\ x^{-1})\ h$

$x\ h\ x^{-1} = e\ h$

$x\ h\ x^{-1} = h \in H$

As a result, H is a normal subgroup of G.

2.8 Permutation and Symmetric groups

The set of all bijections from the set to itself with composition of functions as the group action is called a symmetric group on a set. The set of all permutations of the set's elements is known as the permutation group.

A "permutation group" is a group that acts (faithfully) on a set; this includes both symmetric groups (which are the groups of all permutations of the set) and subgroups of symmetric groups.

Although all groups can be realized as permutation groups (by acting on themselves), this type of action is rarely useful in studying the group; on the other hand, special types of actions (irreducible, faithful, transitive, doubly transitive, etc.) can provide a wealth of information about the group. Jordan, for example, established that A_7, S_6, S_5, and the Mathieu group M_{12} are the only finite sharply five transitive groups.

(A "sharply five transitive group" is a group G acting on a set X with five or more components, such that there exists one and only one $g \in G$ such that $g.\ a_i = b_i$ for every ten elements $a_1,...,a_5$, $b_1,...,b_5 \in X$, with $a_i \neq a_j$ for $i \neq j$ and $b_i \neq b_j$ for $i \neq j$.)

(In fact, Jordan showed that the only finite sharply k-transitive groups for $k \geq 4$ are S_k, S_{k+1}, A_{k+2}, M_{11}, and M_{12}.

A permutation group is composed of a group G and a faithful action: $G \times X \rightarrow X$ on a set X (faithful here implies that if $gx = x$ for all x, then $g = e$). Cayley's Theorem states that any group G can be thought of as a permutation group if X is the underlying set of G and multiplication is performed. However, because the set on which G acts is enormous, this results in an embedding of G into a very large symmetric group. If the set you're acting on is "small"-ish, you'll usually get additional information.

Cayley's Theorem was developed because, in the past, people only examined permutation groups: collections of functions that acted on sets (the sets of roots of a polynomial, the points on the plane via

symmetries, etc.). Cayley was attempting to abstract the concept of group; however, he pointed out that his more abstract definition included all of the things that people were already thinking about, and that it did not introduce any new ones in the sense that any abstract group could be considered a permutation group.

Key takeaway

1. The set of all bijections from the set to itself with composition of functions as the group action is called a symmetric group on a set.
2. The set of all permutations of the set's elements is known as the permutation group.
3. Cayley's Theorem was developed because, in the past, people only examined permutation groups: collections of functions that acted on sets (the sets of roots of a polynomial, the points on the plane via symmetries, etc.).

2.9 Group Homomorphisms

A homomorphism is a mapping f: G →G' that has the properties f (xy) =f(x) f(y), ∀ x, y ∈ G. Despite the fact that the binary operations of the groups G and G' are different, the mapping f preserves the group operation. The homomorphism condition is the one described above.

Kernel of Homomorphism: -The set {x∈ G | f(x) =e'} is the Kernel of a homomorphism f from a group G to a group G' with identity e'.

Ker f stands for the kernel of f.

If f: G→G' is a homomorphism of G intoG', then f's image set is the range of the map f, indicated by f (G). Thus

Im (f) = f (G) = {f(x)∈ G'| x ∈G}

G' is considered a homomorphic image of G if f (G) =G'

Example: Let G be (Z, +) i.e., the group of integers under addition and let f: G →G defined by Ø(x) = 3x ∀x ∈G. Prove that f is homomorphism, determine its Kernel.

Solution: We have Ø(x) = 3x ∀x ∈G

∀x, y ∈G ⇒x + y ∈G (∴G is a group under addition)

Now

f(x + y) = 3 (x + y)

= 3x + 3y

= f (x) + f (y)

Hence f is homomorphism.

Kernel of homomorphism consists of half of zero i.e., the integers whose double is zero.

Thus K = {0}

Isomorphism

Let's consider two algebraic systems, (G1, *) and (G2,0), where * and 0 are both binary operations. If there is an isomorphic mapping f: G1→G2, the systems (G1, *) and (G2,0) are said to be isomorphic.

When two algebraic systems are structurally comparable, one can be produced from the other by merely keeping the elements and operations the same.

Example, consider the two algebraic systems (A1, *) and (A2,⊡) illustrated in fig. Determine the isomorphism of the two algebraic systems.

x	a	b	c
a	a	b	c
b	b	c	a
c	c	a	b

⊡	1	w	w^2
1	1	w	w^2
w	w	w^2	1
w^2	w^2	1	w

Solution - The two algebraic systems (A1, *) and (A2,⊡), respectively, are isomorphic, and (A2,⊡) is an isomorphic image of A1.

f(a)=1

f (b)=w

f (c)= w^2

Automorphism

Let's consider two algebraic systems, (G1, *) and (G2,0), where * and 0 are binary operations on G1 and G2, respectively. If G1= G2, then an isomorphism from (G1, *) to (G2,0) is known as an automorphism.

Key takeaway

1. A homomorphism is a mapping f: G →G' that has the properties f (xy) =f(x) f(y), ∀ x, y ∈ G.
2. Despite the fact that the binary operations of the groups G and G' are different, the mapping f preserves the group operation.

2.10 Definition and elementary properties of Rings and Fields

A ring is an algebraic system (R, +,) that satisfies the following conditions: R is a set with two arbitrary binary operations + and.

1. (R, +) is an abelian group.
2. (R, ·) is a semigroup.
3. The addition operation is distributive over the multiplication operation, i.e.,

a (b+c) =ab +ac and (b+c) a = ba + ca for all a, b, c ∈ R.

Example - Under the addition and multiplication modulo 9 operations, the set Z_9 = 0, 1, 2, 3, 4, 5, 6, 7, 8 forms a ring.

Example: The set of integers Z, with respect to the operations + and × is a ring.

Types of Rings

1. Commutative Rings:

A commutative ring (R, +,) is defined as one that holds the commutative law under multiplication, i.e., a. b = b. a, for any a, b ∈ R.

Example, Consider a set E of all even numbers that is subjected to addition and multiplication operations. A commutative ring is formed by the set E.

2. Ring with Unity:

If a ring (R, +,) has a multiplicative identity, it is called a ring with unity.

Example, Consider a set M of all 2 x 2 matrices over integers that are subjected to matrix multiplication and addition. With unity $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, the set M forms a ring.

3. Ring with Zero Divisions:

If a.b=0, and a and b are any two non-zero R elements in the ring (R, +), then a and b are known as divisions of zero, and the ring (R, +) is known as ring with zero division.

4. Rings without Zero Division:

A ring without divisors of zero is an algebraic system (R, +) in which R is a set with two arbitrary binary operations + and we have $a.b \neq 0 \Longrightarrow a \neq 0$ and $b \neq 0$ a for any a, b ∈ R.

Order of ring

The number of elements in a finite ring R is called the order of ring R. We denote this by |R|

Let (R, +, .) be a ring with unity. An element a $\in$ R is said to be invertible, if there exists an element $a^{-1} \in$ R, called the inverse of a such that

a . a^{-1} = a^{-1} . a = 1

Theorem- If R is a ring then:

(i) a . 0 = 0 = 0 . a ∀ a $\in$ R

(ii) a (–b) = (–a) b = –(ab) ∀ a, b $\in$ R

(iii) (–a) (–b) = ab, for every a, b $\in$ R

Proof:

We know that

a + 0 = a ∀ a $\in$ R

$\Rightarrow$ a . (a + 0) = a · a

$\Rightarrow$ a . a + a . 0 = a · a + 0

$\Rightarrow$a . 0 = 0 (by left cancellation under addition)

Similarly, we can prove

0 . a = 0

0 . a = 0

(ii) $b \in R \Rightarrow -b \in R$ such that $b + (-b) = 0$

$\Rightarrow a \cdot (b + (-b)) = a \cdot 0$

$\Rightarrow a \cdot b + a \cdot (-b) = 0$

$\Rightarrow a \cdot (-b) = -(a \cdot b)$

Similarly, we can prove

$(-a) \cdot b = -(a \cdot b)$

(iii)

We have

$a \cdot (-b) + (-a) \cdot (-b)$

$= (a + (-a)) \cdot (-b)$

$= 0 \cdot (-b)$

$= 0$

$= a \cdot ((-b) + b)$

$= a \cdot (-b) + a \cdot b$

By left cancellation law

$(-a) \cdot (-b) = a \cdot b$

Corollary: Let $(R, +, .)$ be a ring, then

$a \cdot (b - c) = a \cdot b - a \cdot c$

$(b - c) \cdot a = b \cdot a - c \cdot a$ for all $a, b, c \in R$

Proof:

$a \cdot (b - c) = a \cdot (b + (-c))$

$= a \cdot b + a \cdot (-c)$

$= a \cdot b - a \cdot c$

Hence $a \cdot (b - c) = a \cdot b - a \cdot c$

Similarly, we can prove that

$(b - c) \cdot a = b \cdot a - c \cdot a$

Corollary: If $(R, +, .)$ is a ring with unity then for all $a \in R$

(i) $(-1) \cdot a = -a$

(ii) $(-1) \cdot (-1) = 1$

Proof:

(i) $(-1) \cdot a = -(1 \cdot a) = -a$

(ii) R is a ring with unity element, then $1 \cdot a = a \; \forall \; a \in R$

We have $(1)a + (-1) \cdot a = 1 \cdot a + (-1) \cdot a$

$= (1 + (-1)) \cdot a$

$= 0 \cdot a$

= 0

$\Rightarrow$ a + (–1) a = 0

$\Rightarrow$ (-1) . a = - a

If a = –1, then (-1) . (-1) = - (-1)

$\Rightarrow$ (-1) . (-1) = 1

Definition (Field) - A commutative division ring is called a field.

Let Q be the set of rational numbers and '+' and '.' be two binary operation, then (Q, +, .) is a ring.

Q is field too.

Note-

1. A finite integral domain is a field.
2. The characteristic of the ring of integers (Z, +, .) is zero
3. The characteristic of an integral domain is either a prime or zero.
4. The characteristic of a field is either a prime or zero.

Key takeaway

1. A ring is an algebraic system (R, +,) that satisfies the following conditions: R is a set with two arbitrary binary operations + and.
2. A commutative division ring is called a field.

Unit - 3
Lattices

3.1 Lattices: Definition

A lattice L can be defined in two ways. One method is to express L as a partially ordered set. For any pair of elements a, b L, a lattice L can be defined as a partially ordered set in which inf(a, b) and sup(a, b) exist. Another option is to axiomatically define a lattice L.

Let L be a non-empty set that can be closed using the binary operations meet and join, which are denoted by $\wedge$ and $\vee$. If the following axioms hold, and a, b, and c are elements in L, then L is termed a lattice:

1) Commutative Law:

(a) $a \wedge b = b \wedge a$ (b) $a \vee b = b \vee a$

2) Associative Law:

(a) $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ (b) $(a \vee b) \vee c = a \vee (b \vee c)$

3) Absorption Law:

(a) $a \wedge (a \vee b) = a$ (b) $a \vee (a \wedge b) = a$

Duality

Any statement in a lattice $(L,\wedge,\vee)$ has a dual, which is defined as a statement derived by interchanging $\wedge$ an $\vee$.

Example: The dual of $a \wedge (b \vee a) = a \vee a$ is $a \vee (b \wedge a) = a \wedge a$

Key takeaway

1. A lattice L can be defined in two ways. One method is to express L as a partially ordered set.
2. For any pair of elements a, b L, a lattice L can be defined as a partially ordered set in which inf(a, b) and sup(a, b) exist. Another option is to axiomatically define a lattice L.

3.2 Properties of lattices – Bounded, Complemented, Modular and Complete lattice

If a lattice L has a greatest element 1 and a least element 0, it is termed a bounded lattice.

Example

1. Because is the least element of P(S) and the set S is the greatest element of P, the power set P(S) of the set S under the operations of intersection and union is a bounded lattice $\emptyset$.

2. Because it has a least element 1 but no largest element, the set of +ve integer I+ in the normal order of ≤ is not a bounded lattice.

Properties of bounded lattice

If L is a bounded lattice, then we have the following identities for any element a ∈ L:

1. a ∨ 1 = 1
2. a ∧1= a
3. a ∨0=a
4. a ∧0=0

Theorem: Demonstrate that each finite lattice L = a_1, a_2, a_3,...an is bounded.

Proof: We have given the finite lattice:

L = {a_1,a_2,a_3....a_n}

Thus, the greatest element of Lattices L is $a_1 \vee a_2 \vee a_{3 \vee \ldots \vee an}$.

Also, the least element of lattice L is $a_1 \wedge a_2 \wedge a_3 \wedge \ldots \wedge a_n$.

Since every finite lattice has the greatest and least elements. As a result, L is constrained.

Complemented lattice

Let L be a bounded lattice with lower and upper bounds of o and I, respectively. If L is true, let a be an element. If a ∨ x = I and a ∧ x = 0, an element x in L is considered a complement of a.

If a lattice L is bounded and every element in L has a complement, it is said to be complemented.

Determine the complement of a and c in the following example:

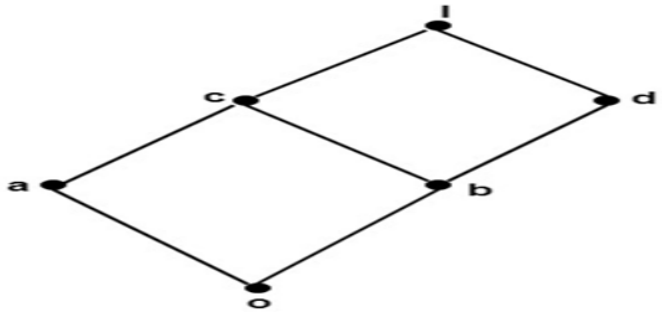


Fig: Example

Solution - The complement of the letter an is d. Because a ∨ d = 1 and a ∧ d = 0

There is no such thing as a complement of c. Since there is no element c with the properties c ∨ c'=1 and c ∧ c'=0,

Modular lattice

If a lattice satisfies the following property, it is called a modular lattice.

a^(b∨(a^d)) = (a^b)(a^d).

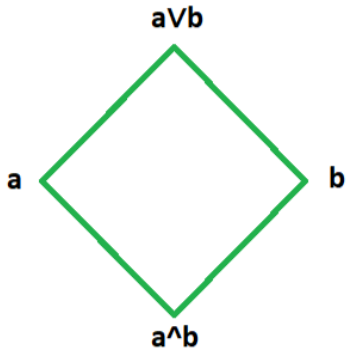


Fig: Example

Complete lattice

If all of a poset's subsets have both a join and a meet, it is called a complete lattice. Every full lattice, in particular, is a bounded lattice. Complete lattice homomorphisms are necessary to preserve arbitrary joins and meetings, whereas bounded lattice homomorphisms only retain finite joins and meets.

Every full semilattice poset is also a complete lattice. This finding is complicated by the fact that different definitions of homomorphism exist for this class of posets, depending on whether they are considered complete lattices, complete join-semilattices, complete meet-semilattices, or join-complete or meet-complete lattices.

It's worth noting that "partial lattice" isn't the polar opposite of "complete lattice"; rather, "partial lattice," "lattice," and "complete lattice" are all increasingly limited definitions.

Key takeaway

1. If a lattice L has a greatest element 1 and a least element 0, it is termed a bounded lattice.
2. Every full semilattice poset is also a complete lattice.

3.3 Boolean Algebra: Introduction

A Boolean Algebra is a supplemented distributive lattice. It is indicated as (B,∧,∨,',0,1), where B is a set that defines two binary operations ∧ (*) and ∨ (+) as well as a unary operation (complement). In this case, B has two unique elements: 0 and 1.

Each element of B has a unique complement since (B,∧,∨) is a complemented distributive lattice.

- It deals with binary numbers & variables.
- Therefore, also known as Binary Algebra or logical Algebra.
- A mathematician named George Boole had developed this algebra in 1854.
- The variables that are used in this algebra are known as Boolean variables.
- Considering the range of voltages as logic 'High' is represented with '1' and logic 'Low' is represented with '0'.

Postulates and Basic Laws of Boolean algebra

Here, the Boolean postulates and basic laws that are used are given underneath.

Boolean Postulates

- Considering the binary numbers 0 and 1, Boolean variable (x) and its complement (x').
- They know as **literal**.
- The possible **logical OR** operations are:

x + 0 = x

x + 1 = 1

x + x = x

x + x' = 1

- Similarly, the possible **logical AND** operations are:

x.1 = x

x.0 = 0

x.x = x

x.x' = 0

- These are the simple Boolean postulates and verification can be done by substituting the Boolean variable with '0' or '1'.

Key takeaway

1. A Boolean Algebra is a supplemented distributive lattice.
2. It is indicated as (B,∧,∨,',0,1), where B is a set that defines two binary operations ∧ (*) and ∨ (+) as well as a unary operation (complement). In this case, B has two unique elements: 0 and 1.

3.4 Axioms and Theorems of Boolean algebra

Basic Laws of Boolean algebra

- The three basic laws of Boolean Algebra are:
- Commutative law
- Associative law
- Distributive law

Commutative Law

- The logical operation carried between two Boolean variables gives the same result irrespective of the order of the two variables, then that operation is said to be **Commutative**. The logical OR & logical AND operations between x & y are shown below

x + y = y + x

x.y=y.x

- The symbol '+' and '.' indicates logical OR operation and logical AND operation.
- Commutative law holds logical OR & logical AND operations.

Associative Law

- If a logical OR operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable providing the same result, then that operation is said to be **Associative**. The logical OR & logical AND operations of x, y & z are:

x + (y + z) = (x + y) + z

x. (y.x) = (x.y).z

- Associative law holds well for logical OR & logical AND operations.

Distributive Law

- If a logical OR operation of any two Boolean variables is performed first and then AND operation is performed with the remaining variable, then that logical operation is said to be **Distributive**. The distribution of logical OR & logical AND operations between variables x, y & z are :

x. (y + z) = x.y + x.z

x + (y.x) = (x + y). (x + z)

- Distributive law holds well for logical OR and logical AND operations.
- These are the Basic laws of Boolean algebra and we can verify them by substituting the Boolean variables with '0' or '1'.

Numerical

- **Simplify** the Boolean function,

f = p'qr + pq'r + pqr' + pqr

Method 1 -

Given f = p'qr + pq'r + pqr' +pqr.

In first and second term r is common and in third and fourth terms pq is common.

So, taking out the common terms by using **Distributive law** we get,

⇒ f = (p'q + pq') r + pq (r' + r)

The terms present in first parenthesis can be simplified by using Ex-OR operation.

The terms present in second parenthesis is equal to '1' using **Boolean postulate** we get

$\Rightarrow f = (p \oplus q)\, r + Pq\ (1)$

The first term can't be simplified further.

But, the second term is equal to pq using **Boolean postulate**.

$\Rightarrow f = (p \oplus q)\, r + pq$

Therefore, the simplified Boolean function is **$f = (p \oplus q)\, r + pq$**

Method 2

Given $f = p'qr + pq'r + pqr' + pqr$.

Using the **Boolean postulate**, $x + x = x$.

Hence, we can write the last term pqr two more times.

$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$

Now using the **Distributive law** for 1^{st} and 4^{th} terms, 2^{nd} and 5^{th} terms, $3^{rd}$and 6^{th} terms we get.

$\Rightarrow f = qr\ (p' + p) + pr\ (q' + q) + pq\ (r' + r)$

Using **Boolean postulate**, $x + x' = 1$ and $x.1 = x$ for further simplification.

$\Rightarrow f = qr\ (1) + pr\ (1) + pq\ (1)$

$\Rightarrow f = qr + pr + pq$

$\Rightarrow f = pq + qr + pr$

Therefore, the simplified Boolean function is **$f = pq + qr + pr$**.

Hence we got two different Boolean functions after simplification of the given Boolean function. Functionally, these two functions are same. As per requirement, we can choose one of them.

Numerical

Find the **complement** of the Boolean function,

$f = p'q + pq'$.

Solution:

Using Demerger's theorem, $(x + y)' = x'.y'$ we get

$\Rightarrow f' = (p'q)'.(pq')'$

Then by second law, $(x.y)' = x' + y'$ we get

$\Rightarrow f' = \{(p')' + q'\}.\ \{p' + (q')'\}$

Then by using, $(x')'=x$ we get

$\Rightarrow f' = \{p + q'\}.\ \{p' + q\}$

$\Rightarrow f' = pp' + pq + p'q' + qq'$

Using $x.x'=0$ we get

$\Rightarrow f = 0 + pq + p'q' + 0$

$\Rightarrow f = pq + p'q'$

Therefore, the **complement** of Boolean function, p'q + pq' is **pq + p'q'**.

Key takeaway

1. The logical operation carried between two Boolean variables gives the same result irrespective of the order of the two variables, then that operation is said to be **Commutative**.
2. If a logical OR operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable providing the same result, then that operation is said to be **Associative**.

3.5 Algebraic manipulation of Boolean expressions

Standard forms of Boolean expressions

- Four product combinations is obtained by combining two variables x and y with logical AND operation. They are called as **min terms** or **standard product terms**. The min terms are given as x'y', x'y, xy' and xy.
- In the same way, four Boolean sum terms is obtained by combining two variables x and y with logical OR operation. They are called as **Max terms** or **standard sum terms**. The Max terms are given as x + y, x + y', x' + y and x' + y'.

The following table represents the min terms and MAX terms for 2 variables.

x	**y**	**Min terms**	**Max terms**
0	0	m_0=x'y'	M_0=x + y
0	1	m_1=x'y	M_1=x + y'
1	0	m_2=xy'	M_2=x' + y
1	1	m_3=xy	M_3=x' + y'

- If the binary variable is '0', then it is represented as complement of variable in min term and as the variable itself in Max term.
- Similarly, if it is '1', then it is represented as complement of variable in Max term and as the variable itself in min term.
- From the above table, we can easily notice that min terms and Max terms are complement of each other.
- If there are 'n' Boolean variables, then there will be 2^n min terms and 2^n Max terms.

Canonical SoP and PoS forms

- A truth table comprises of a set of inputs and output(s).
- If there are 'n' input variables, then there shall be 2^n possible combinations comprising of zeros and ones.
- So the value of every output variable depends on the combination of input variables.
- Hence, each output variable has '1' for some combination and '0' for other combination of input variables.

Canonical SoP form

- It means Canonical Sum of Products form.
- In this, each product term contains all literals.
- So that these product terms are nothing but the min terms.
- Hence is also known as **sum of min terms** form.
- Firstly, identification of the min terms is done and then the logical OR of those min terms is taken in order to get the Boolean expression (function) corresponding to that output variable.
- This Boolean function will be in sum of min terms form.
- Then following the same procedure for other output variables too.

Example

Consider the following **truth table**.

Inputs		**Output**	
P	**q**	**r**	**f**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- Here, the output (f) is '1' for only four combinations of inputs.
- The corresponding min terms are given as p'qr, pq'r, pqr', pqr.
- By doing logical OR, we get the Boolean function of output (f).
- Hence, the Boolean function of output is,

f = p'qr + pq'r + pqr' + pqr.

- This is the desired **canonical SoP form** of output, f.
- It can also be represented as:

f=m3+m5+m6+m7f=m3+m5+m6+m7

f=∑m (3, 5, 6, 7) f=∑m (3, 5, 6, 7)

- First, we represented the function as sum of respective min terms and then, the symbol for summation of those min terms is used.

Canonical PoS form

- It means Canonical Product of Sums form.
- Here In this form, each sum term contains all literals.
- These sum terms are the Max terms.
- Hence, canonical PoS form is also known as **product of Max terms** form.
- Identification of the Max terms for which the output variable is zero is done and then the logical AND of those Max terms is done in order to get the Boolean expression corresponding to that output variable.
- This Boolean function is in the form of product of Max terms.
- Following the same procedure for other output variables too.

Standard SoP and PoS forms

Standard SoP form

- It stands for **Standard Sum of Products** form.
- In this, each product term need not contain all literals.
- So, the product terms can or cannot be the min terms.
- Therefore, it is therefore the simplified form of canonical SoP form.

Standard SoP of output variable can be obtained by two steps.

- Getting the canonical SoP form of output variable
- Simplification the above Boolean function.

The same procedure is followed for other output variables too, if there is more than one output variable.

Numerical

Convert the Boolean function into Standard SoP form.

f = p'qr + pq'r + pqr' + pqr

Solution:

Step 1 – By using the **Boolean postulate**, x + x = x and also writing the last term pqr two more times we get

⇒ f = p'qr + pq'r + pqr' + pqr + pqr + pqr

Step 2 – By Using **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rdand 6th terms.

⇒ f = qr (p' + p) + pr (q' + q) + pq (r' + r)

Step 3 – Then Using **Boolean postulate**, x + x' = 1 we get

⇒ f = qr (1) + pr (1) + pq (1)

Step 4 – hence using **Boolean postulate**, x.1 = x we get

⇒ f = qr + pr + pq

⇒ f = pq + qr + pr

This is the required Boolean function.

Standard PoS form

- It stands for **Standard Product of Sum** form.
- Here, each sum term need not contain all literals.
- So, the sum terms can or cannot be the Max terms.
- Therefore, it is the desired simplified form of canonical PoS form.

Standard PoS form of output variable is obtained by two steps.

- Getting the canonical PoS form of output variable
- Simplification of the above Boolean function.

The same procedure is followed for other output variables too.

Numerical

Convert the Boolean function into Standard PoS form.

f = (p + q + r). (p + q + r'). (p + q' + r). (p' + q + r)

Solution:

Step 1 – By using the **Boolean postulate**, x.x = x and writing the first term p+q+r two more times we get

⇒ f = (p + q + r). (p + q + r). (p + q + r). (p + q + r'). (p +q' + r). (p' + q + r)

Step 2 – Now by using **Distributive law,** x + (y.x) = (x + y). (x + z) for 1st and 4th parenthesis, 2nd and 5th parenthesis, 3rd and 6th parenthesis.

⇒ f = (p + q + rr'). (p + r + qq'). (q + r + pp')

Step 3 – Applying **Boolean postulate**, x.x'=0 for simplifying of the terms present in each parenthesis.

⇒ f = (p + q + 0). (p + r + 0). (q + r + 0)

Step 4 – Using **Boolean postulate**, x + 0 = x we get

⇒ f = (p + q). (p + r). (q + r)

⇒ f = (p + q). (q + r). (p + r)

This is the simplified Boolean function.

Hence, both Standard SoP and Standard PoS forms are Dual to one another.

Key takeaway

1. Four product combinations is obtained by combining two variables x and y with logical AND operation. They are called as **min terms** or **standard product terms**.
2. A truth table comprises of a set of inputs and output(s).
3. If there are 'n' input variables, then there shall be 2^n possible combinations comprising of zeros and ones.

3.6 Simplification of Boolean Functions

One Boolean statement is minimized into an equivalent expression using Boolean identities in this method.

Problem 1:

Minimize the following Boolean expression using Boolean identities –

F(A,B,C)=(A+B)(A+C)

Solution:

Given, F(A,B,C)=(A+B)(A+C)

Or, F(A,B,C)=A.A+A.C+B.A+B.C

[Applying distributive Rule]

Or, F(A,B,C)=A+A.C+B.A+B.C

[Applying Idempotent Law]

Or, F(A,B,C)=A(1+C)+B.A+B.C

[Applying distributive Law]

Or, F(A,B,C)=A+B.A+B.C

[Applying dominance Law]

Or, F(A,B,C)=(A+1).A+B.C

[Applying distributive Law]

Or, F(A,B,C)=1.A+B.C

[Applying dominance Law]

Or, F(A,B,C)=A+B.C

[Applying dominance Law]

So, F(A,B,C)=A+BC is the minimized form.

Problem 2:

Reduce the following Boolean expression to its simplest form using Boolean identities:

F(A,B,C)=A′B+BC′+BC+AB′C′

Solution:

Given, F(A,B,C)=A′B+BC′+BC+AB′C′

Or, F(A,B,C)=A′B+(BC′+BC′)+BC+AB′C′

[By idempotent law, BC' = BC' + BC']

Or, F(A,B,C)=A′B+(BC′+BC)+(BC′+AB′C′)

Or, F(A,B,C)=A′B+B(C′+C)+C′(B+AB′)

[By distributive laws]

Or, F(A,B,C)=A′B+B.1+C′(B+A)

[(C' + C) = 1 and absorption law (B + AB')= (B + A)]

Or, F(A,B,C)=A′B+B+C′(B+A)

[B.1 = B]

Or, F(A,B,C)=B(A′+1)+C′(B+A)

Or, F(A,B,C)=B.1+C′(B+A)

[(A' + 1) = 1]

Or, F(A,B,C)=B+C′(B+A)

[As, B.1 = B]

Or, F(A,B,C)=B+BC′+AC′

Or, F(A,B,C)=B(1+C′)+AC′

Or, F(A,B,C)=B.1+AC′

[As, (1 + C') = 1]

Or, F(A,B,C)=B+AC′

[As, B.1 = B]

So, F(A,B,C)=B+AC′ is the minimized form.

3.7 Karnaugh maps

- Karnaugh map method or K-map method is the pictorial representation of the Boolean equations and Boolean manipulations are used to reduce the complexity in solving them. These can be considered as a special or extended version of the 'Truth table'.
- Karnaugh map can be explained as ''An array containing 2k cells in a grid like format, where k is the number of variables in the Boolean expression that is to be reduced or optimized''. As it is evaluated from the truth table method, each cell in the K-map will represent a single row of the truth table and a cell is represented by a square.

- The cells in the k-map are arranged in such a way that there are conjunctions, which differ in a single variable, are assigned in adjacent rows. The K-map method supports the elimination of potential race conditions and permits the rapid identification.
- By using Karnaugh map technique, we can reduce the Boolean expression containing any number of variables, such as 2-variable Boolean expression, 3-variable Boolean expression, 4-variable Boolean expression and even 7-variable Boolean expressions, which are complex to solve by using regular Boolean theorems and laws.

Minimization with Karnaugh Maps and advantages of K-map:

- K-maps are used to convert the truth table of a Boolean equation into minimized SOP form.
- Easy and simple basic rules for the simplification.
- The K-map method is faster and more efficient than other simplification techniques of Boolean algebra.
- All rows in the K-map are represented by using square shaped cells, in which each square in that will represent a minterms.
- It is easy to convert a truth table to k-map and k-map to Sum of Products form equation.

There are 2 forms in converting a Boolean equation into K-map:

1. Un-optimized form
2. Optimized form

- Un-optimized form: It involves in converting the number of 1's into equal number of product terms (min terms) in an SOP equation.
- Optimized form: It involves in reducing the number of min terms in the SOP equation.

Grouping of K-map variables

- There are some rules to follow while we are grouping the variables in K-maps.
- The square that contains '1' should be taken in simplifying, at least once.
- The square that contains '1' can be considered as many times as the grouping is possible with it.
- Group shouldn't include any zeros (0).
- A group should be the as large as possible.
- Groups can be horizontal or vertical. Grouping of variables in diagonal manner is not allowed.

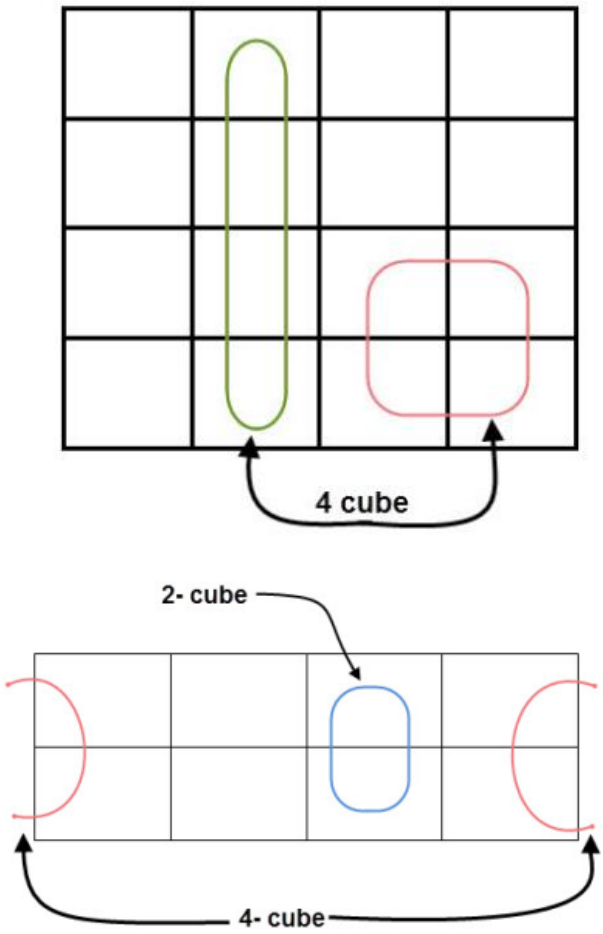
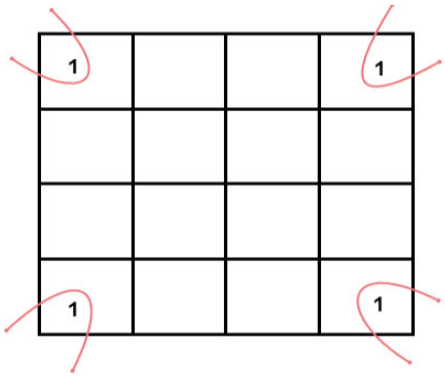
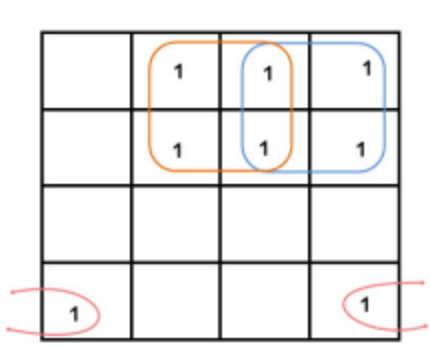
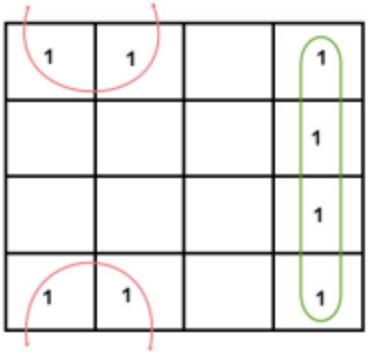


Fig: Grouping

- If the square containing '1' has no possibility to be placed in a group, then it should be added to the final expression.
- Groups can overlap.
- The number of squares in a group must be equal to powers of 2, such as 1, 2, 4, 8 etc.
- Groups can wrap around. As the K-map is considered as spherical or folded, the squares at the corners (which are at the end of the column or row) should be considered as they adjacent squares.
- The grouping of K-map variables can be done in many ways, so they obtained simplified equation need not to be unique always.
- The Boolean equation must be in must be in canonical form, in order to draw a K-map.

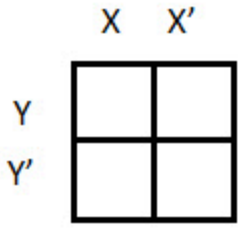






2 variable K-maps

There are 4 cells (22) in the 2-variable k-map. It will look like

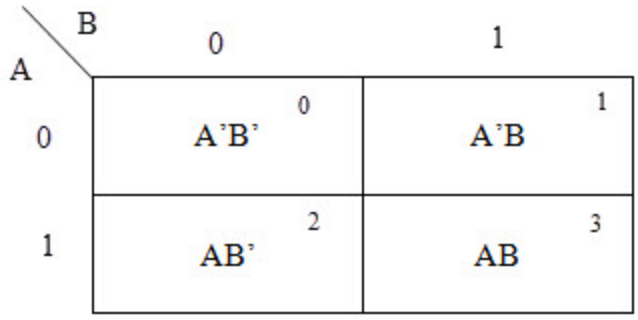


The possible min terms with 2 variables (A and B) are A.B, A.B', A'.B and A'.B'. The conjunctions of the variables (A, B) and (A', B) are represented in the cells of the top row and (A, B') and (A', B') in cells of the bottom row. The following table shows the positions of all the possible outputs of 2-variable Boolean function on a K-map.

A	B	Possible Outputs	Location on K-map
0	0	A'B'	0
0	1	A'B	1

1	0	AB'	2
1	1	AB	3

A general representation of a 2 variable K-map plot is shown below.



When we are simplifying a Boolean equation using Karnaugh map, we represent each cell of K-map containing the conjunction term with 1. After that, we group the adjacent cells with possible sizes as 2 or 4. In case of larger k-maps, we can group the variables in larger sizes like 8 or 16.

The groups of variables should be in rectangular shape that means the groups must be formed by combining adjacent cells either vertically or horizontally. Diagonal shaped or L-shaped groups are not allowed. The following example demonstrates a K-map simplification of a 2-variable Boolean equation.

Example

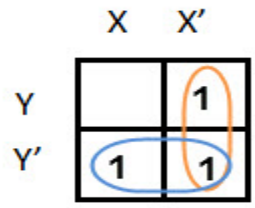
Simplify the given 2-variable Boolean equation by using K-map.

F = X Y' + X' Y + X'Y'

First, let's construct the truth table for the given equation,

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

We put 1 at the output terms given in equation.



In this K-map, we can create 2 groups by following the rules for grouping, one is by combining (X', Y) and (X', Y') terms and the other is by combining (X, Y') and (X', Y') terms. Here the lower right cell is used in both groups.

After grouping the variables, the next step is determining the minimized expression.

By reducing each group, we obtain a conjunction of the minimized expression such as by taking out the common terms from two groups, i.e. X' and Y'. So the reduced equation will be X' +Y'.

3 variable K-maps

For a 3-variable Boolean function, there is a possibility of 8 output min terms. The general representation of all the min terms using 3-variables is shown below.

A	B	C	Possible Outputs	Location on K-map
0	0	0	A'B'C'	0
0	0	1	A'B'C	1
0	1	0	A'BC'	2
0	1	1	A'BC	3
1	0	0	AB'C'	4
1	0	1	AB'C	5
1	1	0	ABC'	6
1	1	1	ABC	7

A typical plot of a 3-variable K-map is shown below. It can be observed that the positions of columns 10 and 11 are interchanged so that there is only change in one variable across adjacent cells. This modification will allow in minimizing the logic.

A \ BC	00	01	11	10
0	A'B'C' 0	A'B'C 1	A'BC 3	A'BC' 2
1	AB'C' 4	AB'C 5	ABC 7	ABC' 6

Up to 8 cells can be grouped in case of a 3-variable K-map with other possibilities being 1, 2 and 4.

Example

Simplify the given 3-variable Boolean equation by using k-map.

F = X' Y Z + X' Y' Z + X Y Z' + X' Y' Z' + X Y Z + X Y' Z'

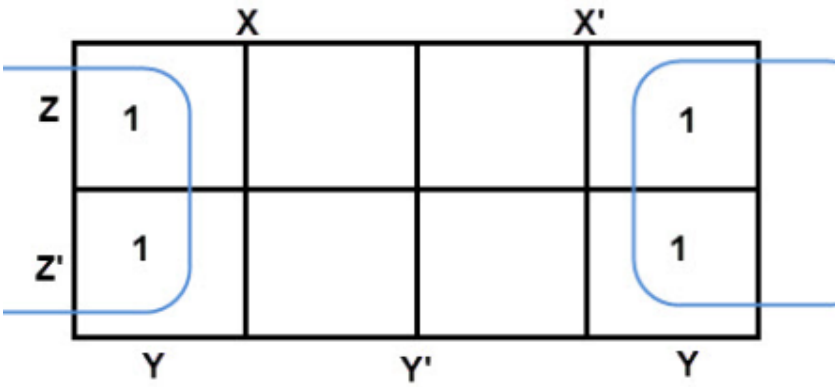
First, let's construct the truth table for the given equation,

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

We put 1 at the output terms given in equation.

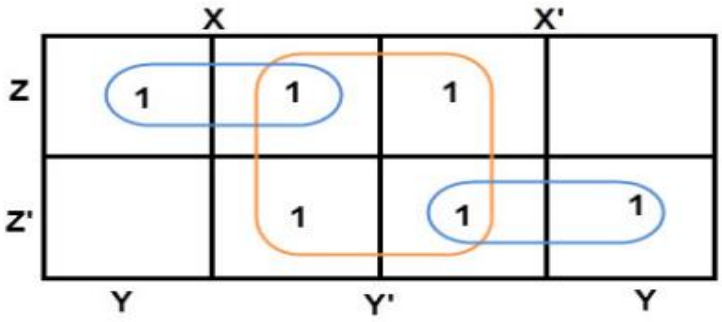
There are 8 cells (23) in the 3-variable k-map. It will look like (see below image).

The largest group size will be 8 but we can also form the groups of size 4 and size 2, by possibility. In the 3 variable Karnaugh map, we consider the left most column of the k-map as the adjacent column of rightmost column. So the size 4 group is formed as shown below.



And in both the terms, we have 'Y' in common. So the group of size 4 is reduced as the conjunction Y.

To consume every cell which has 1 in it, we group the rest of cells to form size 2 group, as shown below.



The 2 size group has no common variables, so they are written with their variables and its conjugates. So the reduced equation will be X Z' + Y' + X' Z. In this equation, no further minimization is possible.

4 variable K-maps

There are 16 possible min terms in case of a 4-variable Boolean function. The general representation of minterms using 4 variables is shown below.

A	B	C	D	Possible Outputs	Location on K-map
0	0	0	0	A'B'C'D'	0
0	0	0	1	A'B'C'D	1
0	0	1	0	A'B'CD'	2
0	0	1	1	A'B'CD	3
0	1	0	0	A'BCD	4
0	1	0	1	A'BC'D	5
0	1	1	0	A'BCD'	6
0	1	1	1	A'BCD	7
0	0	0	0	A'B'C'D'	8
1	0	0	1	AB'C'D	9
1	0	1	0	AB'CD'	10
1	0	1	1	AB'CD	11
1	1	0	0	ABC'D'	12
1	1	0	1	ABC'D	13
1	1	1	0	ABCD'	14
1	1	1	1	ABCD	15

A typical 4-variable K-map plot is shown below. It can be observed that both the columns and rows of 10 and 11 are interchanged.

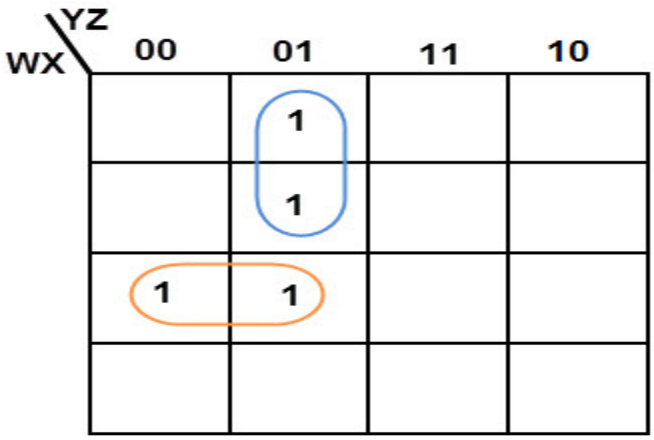
AB \ CD	00	01	11	10
00	0 A' B' C' D'	1 A' B' C' D	3 A' B' C D	2 A' B' C D'
01	4 A' B C' D'	5 A' B C' D	7 A' B C D	6 A' B C D'
11	12 A B C' D'	13 A B C' D	15 A B C D	14 A B C D'
10	8 A B' C' D'	9 A B' C' D	11 A B' C D	10 A B' C D'

The possible numbers of cells that can be grouped together are 1, 2, 4, 8 and 16.

Example

Simplify the given 4-variable Boolean equation by using k-map. F (W, X, Y, Z) = (1, 5, 12, 13)

Sol: F (W, X, Y, Z) = (1, 5, 12, 13)



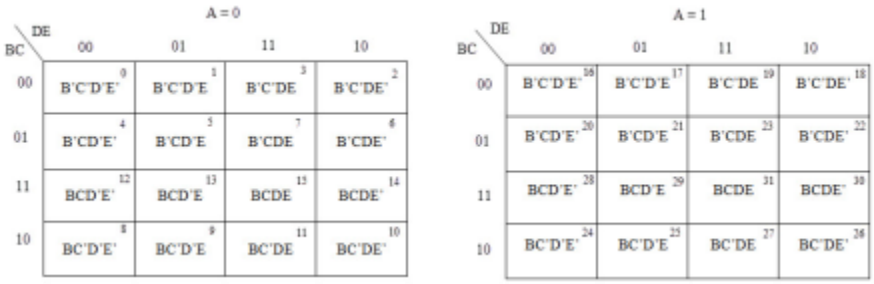
By preparing k-map, we can minimize the given Boolean equation as

F = W Y' Z + W 'Y' Z

5 variable K-maps

A 5-variable Boolean function can have a maximum of 32 minterms. All the possible minterms are represented below

In 5-variable K-map, we have 32 cells as shown below. It is represented by F (A, B, C, D, and E). It is

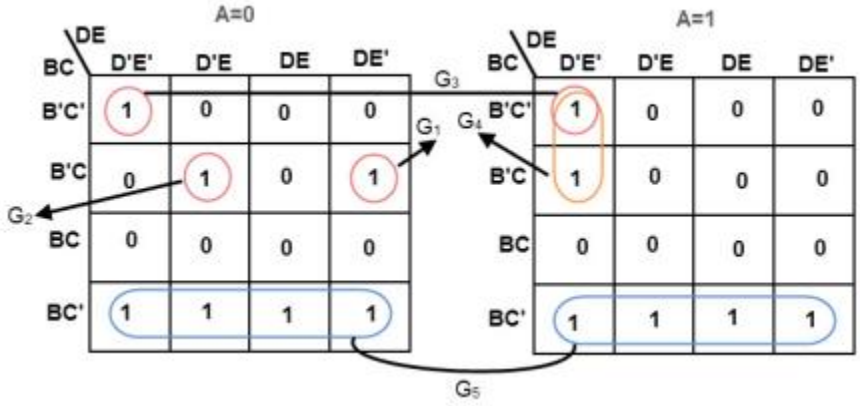


divided into two grids of 16 cells with one variable (A) being 0 in one grid and 1 in other grid.

Example

Simplify the given 5-variable Boolean equation by using k-map.

f (A, B, C, D, E) = ∑ m (0, 5, 6, 8, 9, 10, 11, 16, 20, 42, 25, 26, 27)



K-map with "Don't care" conditions

The "Don't care" conditions are used to replace the empty cell to form a possible grouping of variables. They can be used as either 0 or 1, based on the adjacent variables in the group. The cells that contain "don't care" conditions are represented by an asterisk (*) symbol among the normal 0's and 1's.

In grouping of variables, we can also ignore the "don't cares". "Don't care" conditions are very useful in grouping the variables of large size.

Minimizing an Expression with Don't Cares

We can minimize the Boolean expression by finding the relative functions of the 'don't care' conditions, by assigning them 0 or 1. If n is the number of don't cares in a Boolean equation, the number of functions obtained will be 2^n.

Implementation of BCD to Gray Code Converter using K-map

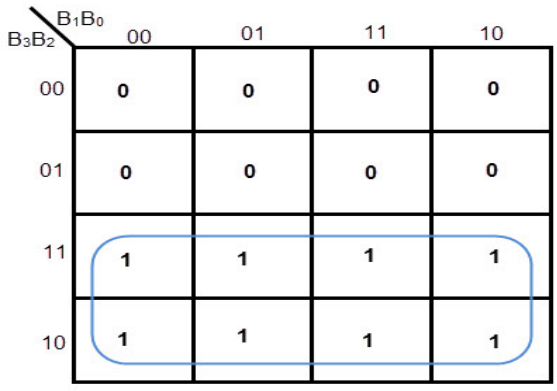
A Gray code is a number series in which two successive numbers differ by one bit. This code acquired its name by the scientist "Frank Gray". He owned the patent for using the Gray code in shaft registers, in the year of 1953.

We can convert the binary coded decimal (BCD) code to Gray code by using k-map simplification.

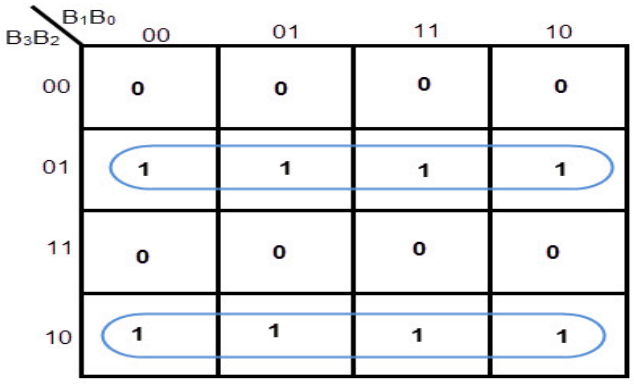
Table for BCD code and Gray code

BCD code				**Gray code**			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

K-MAP FOR G3

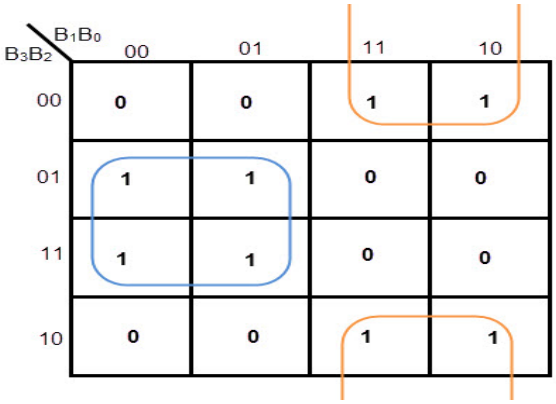


K-MAP FOR G2



Equation for G2= B3' B2 + B3 B2'= B3 XOR B2

K-MAP FOR G1



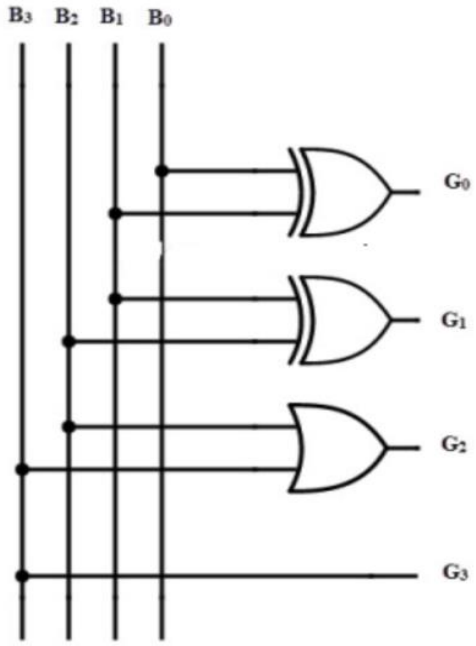
Equation for G1= B1' B2 + B1 B2'= B1 XOR B2

K-MAP FOR G0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

Equation for G0= B1' B0 + B1 B0'= B1 XOR B0

The the implementation of BCD to Gray conversion using logic gates is shown below. Two EX-OR gates and an OR gate are used.



Key takeaway

1. Karnaugh map method or K-map method is the pictorial representation of the Boolean equations and Boolean manipulations are used to reduce the complexity in solving them.
2. These can be considered as a special or extended version of the 'Truth table'.
3. There are some rules to follow while we are grouping the variables in K-maps.
4. There are 4 cells (22) in the 2-variable k-map.
5. For a 3-variable Boolean function, there is a possibility of 8 output min terms.

3.8 Logic gates

The basic gates are namely AND gate, OR gate & NOT gate.

AND gate

It is a digital circuit that consists of two or more inputs and a single output which is the **logical AND** of all those inputs. It is represented with the symbol '.'.

The following is the **truth table** of 2-input AND gate.

A	**B**	**Y = A.B**
0	0	0
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output of two input AND gate.

If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'.

The figure below shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.

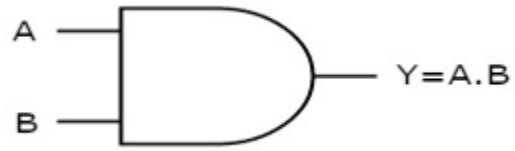
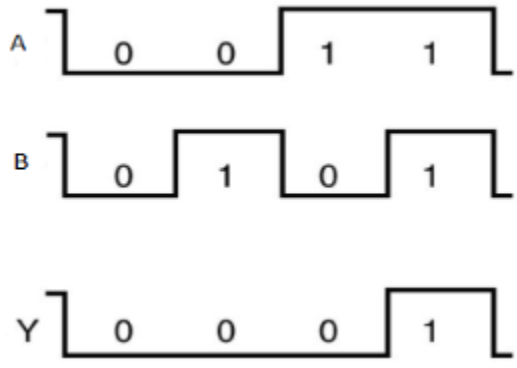


Fig: AND gate

Timing Diagram:



OR gate

It is a digital circuit which has two or more inputs and a single output which is the logical OR of all those inputs. It is represented with the symbol '+'.

The **truth table** of 2-input OR gate is:

A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Here A, B are the inputs and Y is the output of two input OR gate.

When both inputs are '0', then only the output, Y is '0'. For remaining combinations of inputs, the output, Y is '1'.

The figure below shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.

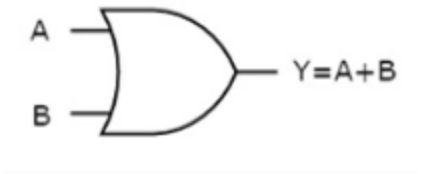
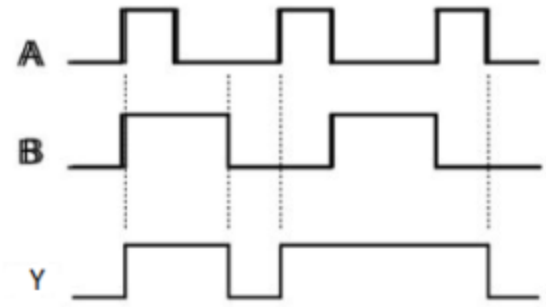


Fig: OR gate

Timing Diagram:



NOT gate

It is a digital circuit that has one input and one output. Here the output is the **logical inversion** of input. Hence, it is also called as an inverter.

The **truth table** of NOT gate is:

A	**Y = A'**
0	1
1	0

Here A and Y are the corresponding input and output of NOT gate. When A is '0', then, Y is '1'. Similarly, when, A is '1', then, Y is '0'.

The figure below shows the symbol of NOT gate, which has one input, A and one output, Y.

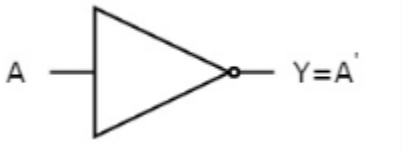
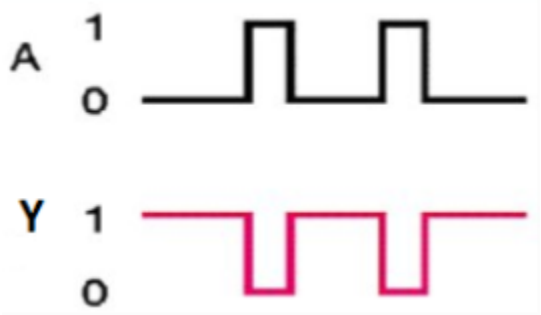


Fig: NOT gate

Timing Diagram:



NAND gate

It is a digital circuit which has two or more inputs and single output and it is the **inversion of logical AND** gate.

The **truth table** of 2-input NAND gate is:

A	**B**	**Y = (A.B)'**
0	0	1
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input NAND gate. When both inputs are '1', then the output, Y is '0'. If at least one of the inputs is zero, then the output, Y is '1'. This is just the inverse of AND operation.

The image shows the **symbol** of NAND gate:

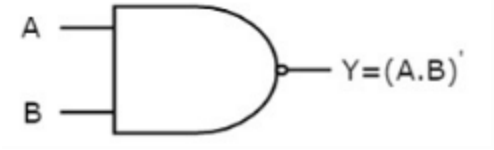
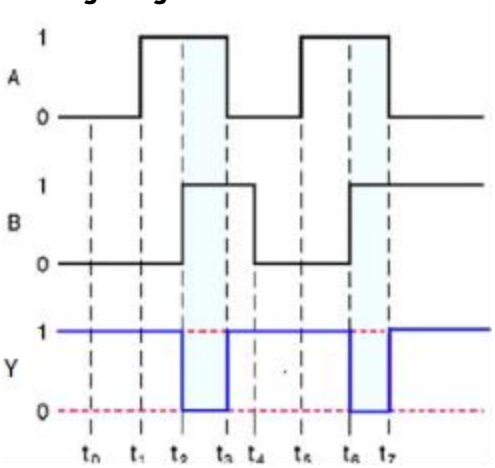


Fig: NAND gate

NAND gate works same as AND gate followed by an inverter.

Timing Diagram:



NOR gate

It is a digital circuit that has two or more inputs and a single output which is the **inversion of logical OR** of all inputs.

The **truth table** of 2-input NOR gate is:

A	**B**	**Y = (A+B)'**
0	0	1
0	1	0
1	0	0
1	1	0

Here A and B are the two inputs and Y is the output. If both inputs are '0', then the output is '1'. If any one of the inputs is '1', then the output is '0'. This is exactly opposite to two input OR gate operation.

The **symbol** of NOR gate is:

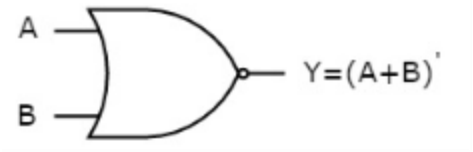
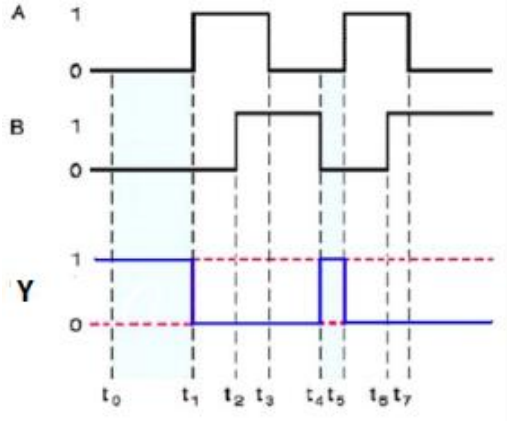


Fig: NOR gate

NOR gate works exactly same as that of OR gate followed by an inverter.

Timing Diagram:



Ex-OR gate

It stands for **Exclusive-OR** gate. Its function varies when the inputs have even number of ones.

The **truth table** of 2-input Ex-OR gate is:

A	**B**	**Y = A⊕B**
0	0	0
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input Ex-OR gate. The output (Y) is zero instead of one when both the inputs are one.

Therefore, the output of Ex-OR gate is '1', when only one of the two inputs is '1'. And it is zero, when both inputs are same.

The symbol of Ex-OR gate is as follows:

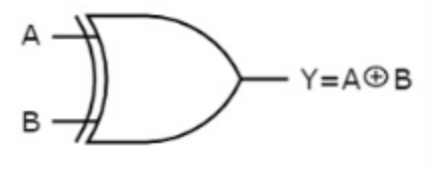
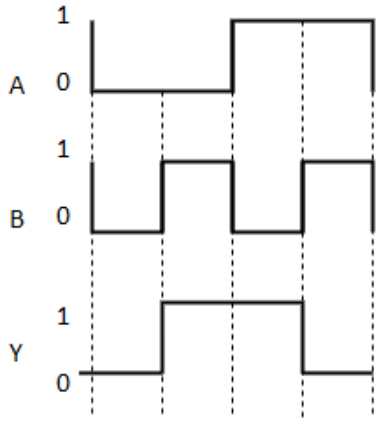


Fig: XOR gate

It is similar to that of OR gate with an exception for few combination(s) of inputs. Hence, the output is also known as an **odd function**.

Timing Diagram:



Ex-NOR gate

It stands for **Exclusive-NOR** gate. Its function neither is same as that of NOR gate except when the inputs having even number of ones.

The **truth table** of 2-input Ex-NOR gate is:

A	**B**	**Y = A⊙B**
0	0	1
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output. It is same as Ex-NOR gate with the only modification in the fourth row. The output is 1 instead of 0, when both the inputs are one.

Hence the output of Ex-NOR gate is '1', when both inputs are same and 0, when both the inputs are different.

The **symbol** of Ex-NOR gate is:

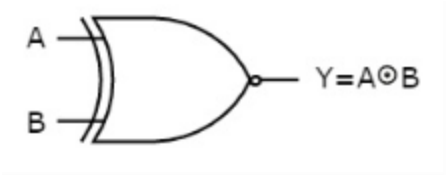
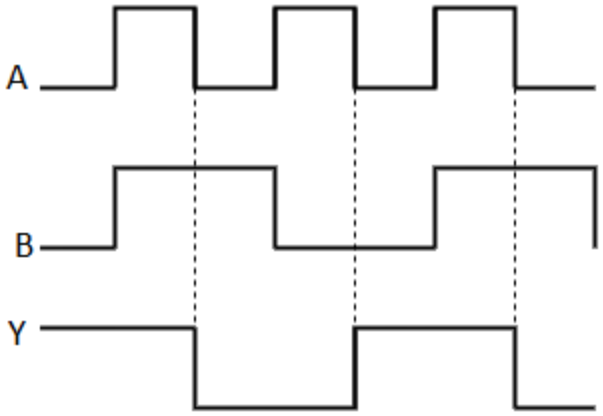


Fig: XNOR gate

It neither is similar to NOR gate except for few combination of inputs. Here the output is '1', when even number of 1 is present at the inputs. Hence is also called as an **even function**.

Timing Diagram:



Key takeaway

1. It is a digital circuit that consists of two or more inputs and a single output which is the **logical AND** of all those inputs.
2. It is a digital circuit which has two or more inputs and a single output which is the logical OR of all those inputs.
3. It is a digital circuit that has one input and one output. Here the output is the **logical inversion** of input.

3.9 Digital circuits and Boolean algebra

The truth tables for the OR, AND, and NOT gates are identical to the truth tables for the propositions $p \vee q$ (disjunction, "p or q"), $p \wedge q$ (conjunction, "p and q"), and $\neg p$ (negation, "not p"), respectively. The only change is that instead of T and F, 1 and 0 are utilized. As a result, logic circuits obey the same principles as propositions, forming a Boolean algebra. This is a formal statement of the result.

Theorem: Logic circuits form a Boolean Algebra

As a result, any Boolean algebra words, such as complements, literals, fundamental products, minterms, sum-of-products, and complete sum-of-products, can be employed with our logic circuits.

AND - OR circuits

An AND-OR circuit is a logic circuit that correlates to a Boolean sum-of-products statement.

(1) Some of the inputs or their complements are fed into each AND gate in a circuit L with multiple inputs.

(2) All of the AND gates' outputs are fed into a single OR gate.

(3) The circuit L's output is the output of the OR gate.

This type of logic circuit is seen in the diagram below.

An example AND-OR circuit with three inputs, A, B, C, and output Y is shown in Figure 15-12. In the inputs A, B, and C, we can easily express Y as a Boolean expression as follows. First, we need to figure out what each AND gate's output is.

(a) The first AND gate inputs are A, B, and C, hence the output is A . B . C.

(b) The second AND gate's inputs are A, B', and C, hence the output is A .B' .C.

(c) The third AND gate's inputs are A' and B; so, the output is A'. B.

The output of the OR gate, which is the circuit's output Y, is the total of the AND gates' outputs. Thus:

$Y = A \cdot B \cdot C + A \cdot B' \cdot C + A' \cdot B$

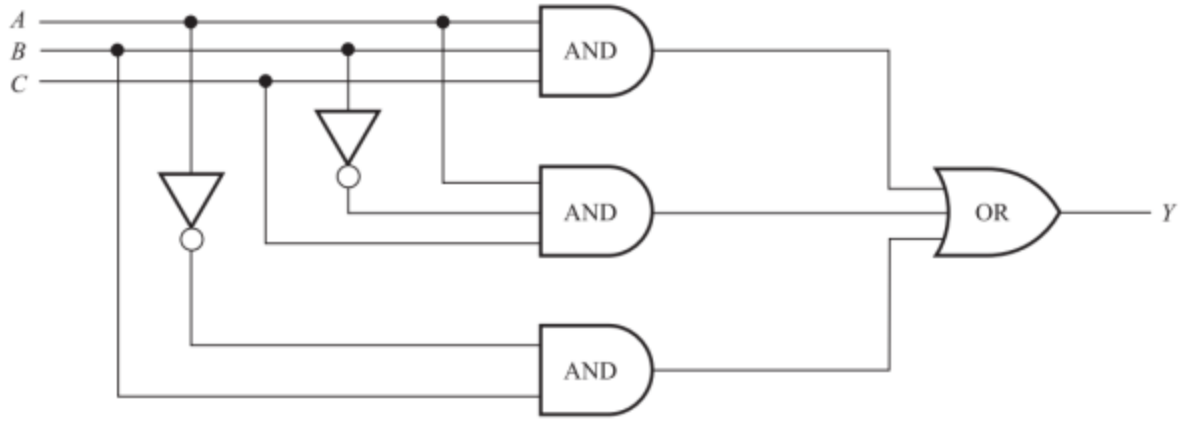


Fig: AND-OR circuit

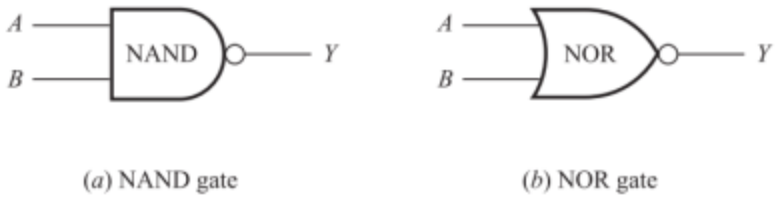
NAND and NOR Gates

There are two more gates that are equal to combinations of the fundamental gates listed above.

(a) A NAND gate is the same as an AND gate followed by a NOT gate, as shown in Fig.(a).

(b) A NOR gate is the same as an OR gate followed by a NOT gate, as shown in Fig.(b).

Figure shows the truth tables for these gates (with two inputs A and B) (c). The NAND and NOR gates, like the corresponding AND and OR gates, can have two or more inputs. Furthermore, a NAND gate's output is 0 if and only if all of its inputs are 1, while a NOR gate's output is 1 if and only if all of its inputs are 0.



(*a*) NAND gate (*b*) NOR gate

Fig: NAND and NOR Gates

A	B	NAND	NOR
1	1	0	0
1	0	1	0
0	1	1	0
0	0	1	1

The only difference between the AND and NAND gates and the OR and NOR gates is that the NAND and NOR gates both have a circle following them. A tiny circle is also used in some manuscripts to denote a complement before a gate. The Boolean expressions for two logic circuits in Fig., for example, are as follows:

(a) Y = (A' B)', (b) Y = (A' + B' + C)'

(*a*) (*b*)

Fig: Two logic circuit

Key takeaway

1. Logic circuits obey the same principles as propositions, forming a Boolean algebra. This is a formal statement of the result.
2. An AND-OR circuit is a logic circuit that correlates to a Boolean sum-of-products statement.
3. There are two more gates that are equal to combinations of the fundamental gates.

Unit – 4

Propositional and Predicate Logic

4.1 Propositional Logic: Proposition, well formed formula, Truth tables, Tautology, Satisfiability, Contradiction, Algebra of proposition, Theory of Inference

Mathematical logic principles define how to reason about mathematical statements. Aristotle, a Greek philosopher, is credited with inventing logical thinking. Many disciplines of mathematics and, as a result, computer science, have their theoretical foundations in logical reasoning. It has numerous applications in computer science, including the design of computing devices, artificial intelligence, and the construction of data structures for programming languages, among others.

Propositional logic is concerned with assertions that can be assigned the truth values "true" and "false." The goal is to evaluate these assertions individually and as a group.

Proposition

A proposition is a collection of declarative statements with either a truth value of "true" or "false." Propositional variables and connectives make up a propositional. The propositional variables are denoted by capital letters (A, B, etc). The connectives are used to link the propositional variables together.

Below are some instances of propositions.

- "Man is Mortal", it returns truth value "TRUE"
- "12 + 9 = 3 – 2", it returns truth value "FALSE"

The following does not constitute a proposition.

- "A is less than 2," says the narrator. It's because we can't tell if a statement is true or incorrect unless we give it a precise value of A.

Well formed formula

Examples of well-formed formulae:

- $(\neg a)\ (\neg(\neg a))$
- $(a \wedge (b \wedge c))$
- $(a \rightarrow (b \rightarrow c))$

We omit parentheses whenever we may restore them through operator precedence:

Binds stronger

$\leftarrow$

$\neg \wedge \vee \rightarrow \boxed{\leftrightarrow}$

Truth tables

When the statement $(P \rightarrow Q) \vee (Q \rightarrow R)$ is true, we must decide. Using the connective definitions, we can see that for this to be true, either $P \rightarrow Q$ or $Q \rightarrow R$ must be true (or both). If P is false or Q is true (in the first case), and Q is false or R is true, then those are true (in the second case). So—yeah, things get a little tangled. Fortunately, we can create a chart to keep track of all the options.

Let's talk about truth tables. The notion is that for each of the sentential variables, we list a possible combination of Ts and Fs (for true and false), and then mark whether the statement in question is true or false in that case. This is done for every possible T and F combination.

This is done for every possible T and F combination. Then we can see whether the assertion is true or untrue in each situation. We will first fill in values for each element of the statement in order to divide up our effort into smaller, more manageable chunks for complicated statements.

Because the truth value of a statement is entirely decided by the truth values of its constituent parts and how they are related, all you need to know are the truth tables for each of the logical connectives. They are as follows:

P	Q	P∧Q
T	T	T
T	F	F
F	T	F
F	F	F

P	Q	P∨Q
T	T	T
T	F	T
F	T	T
F	F	F

P	Q	$P \longrightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

P	Q	$P \leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

The truth table for negation looks like this:

P	$\neg P$
T	F
F	T

Negation - It has the exact opposite meaning as the original sentence. If p is a statement, then ~p is the negation of p, which means 'it is not the case that p.' If p is true, then ~p is false, and vice versa.

Example - If p is the statement that Paris is in France, then ~ p is the statement that Paris is not in France.

p	~ p
T	F
F	T

Conjunction - It refers to the combining of two assertions. If p and q are two statements, then "p and q" is a compound statement represented by p q and referred to as "p and q's conjunction." Only when both p and q are true is the combination of p and q true. Otherwise, it's untrue.

p	q	p ∧ q
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction - It refers to the combining of two assertions. If p and q are two statements, then "p or q" is a compound statement indicated by p ∨ q and known as the p and q disjunction. When at least one of the two statements is true, the disjunction of p and q is true, and it is false only when both p and q are false.

p	q	p ∨ q
T	T	T
T	F	T
F	T	T
F	F	F

Implication / if-then (⟶) - The proposition "if p, then q" is an implication p⟶q. If p is true and q is false, it is false. The remainder of the scenarios are correct.

p	q	p ⟶ q
T	T	T
T	F	F
F	T	T
F	F	F

If and Only If (↔) - p ↔ q is a bi-conditional logical connective that holds true when p and q are the same, i.e., when both are false or true.

p	q	p ↔ q
T	T	T
T	F	F
F	T	F
F	F	T

Tautology

In logic, a tautology is a proposition that is so phrased that it cannot be refuted without causing inconsistency. As a result, the statement "All humans are mammals" is taken to mean that anything is either a person or a mammal. However, that universal "truth" is derived solely

from the actual use of the terms "human" and "mammal," and is thus entirely a question of definition.

The concept of tautology in the propositional calculus was first created by American philosopher Charles Sanders Peirce, the founder of the school of pragmatism and a notable logician, in the early twentieth century. However, the term was coined by the Austrian-born British philosopher Ludwig Wittgenstein, who claimed in his Logisch-philosophische Abhandlung (1921; Tractatus Logico-Philosophicus, 1922) that all necessary propositions are tautologies and that, as a result, all necessary propositions say the same thing—namely, nothing at all.

A tautology is a formula that holds true regardless of the value of its propositional variables.

Example – Prove [(A→B)∧A]→B is a tautology

The following is the truth table:

A	B	A → B	(A →B) ∧ A	[(A→B) ∧ A] → B
True	True	True	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	True

As the final column contains all T's, so it is a tautology.

Satisfiability

A satisfying sentence is one in which the variables have a truth value assignment that makes the sentence true (truth value = t).

• Algorithm? • Try out all of the different assignments to discover which one works best.

The phrase is true because all of the truth values assigned to the variables are true.

• What is an algorithm? • Try out all of the possible assignments and make sure they all work.

Are there any algorithms that are superior than these?

Satisfiability problem

Many problems can be expressed as a list of constraints. Answer is assignment to variables that satisfy all the constraints.

Examples:

Scheduling people to work in shifts at a hospital

– Some people don't work at night

– No one can work more than x hours a week

– Some pairs of people can't be on the same shift

– Is there assignment of people to shifts that satisfy all constraints?

Finding bugs in programs

– Write logical specification of, e.g. Air traffic controller

– Write assertion "two airplanes on same runway at same time"

– Can these be satisfied simultaneously?

Contradiction

A formula that is always untrue for every value of its propositional variables is called a contradiction.

Example – Prove $(A \vee B) \wedge [(\neg A) \wedge (\neg B)]$ is a contradiction

The following is the truth table:

A	B	A ∨ B	¬ A	¬ B	(¬A) ∧ (¬B)	(A∧B) ∧ [(¬A) ∧ (¬B)]
True	True	True	False	False	False	False
True	False	True	False	True	False	False
False	True	True	True	False	False	False
False	False	False	True	True	True	False

Algebra of proposition

A "sum-of-products" or disjunctive form is equivalent to every propositional formula. A disjunctive form is essentially an OR of AND-terms, where each AND-term is an AND of variables or variables' negations.

Conjunctive forms

If a compound statement is generated by operating AND among variables (negation of variables included) connected with ORs, it is said to be in conjunctive normal form. It is a compound statement formed by Intersection among variables connected with Unions in terms of set operations.

Example

- (A∨B)∧(A∨C)∧(B∨C∨D)
- (P∪Q)∩(Q∪R)

Disjunctive forms

If a compound statement is generated by operating OR among variables coupled with ANDs (negation of variables included), it is in disjunctive normal form. It's a compound statement derived from the union of variables linked to Intersections.

Example

- $(A \wedge B) \vee (A \wedge C) \vee (B \wedge C \wedge D)$
- $(P \cap Q) \cup (Q \cap R)$

Theory of Inference

Modus ponens is the basic inference rule. It says that if both $P \rightarrow Q$ and P hold, Q can be deduced, and it's written as

P

$P \rightarrow Q$

Q

The premises are the lines above the dotted line, and the conclusion formed from the premises is the line below it.

If both "if it rains, the game is not played" and "it rains," for example, we can deduce that the game is not played.

In addition to modus ponens, identities and implications can be used to argue.

When the left(right) hand side of a statement's identity is substituted by the right(left) hand side of the identity, the resulting proposition is logically equal to the original proposition. As a result, the new proposition is derived from the old. For example in the proposition $P \wedge (Q \rightarrow R)$, $(Q \rightarrow R)$ can be replaced with $(\neg Q \vee R)$ to conclude P $(Q \rightarrow R)$, since $(Q \rightarrow R) \Leftrightarrow (\neg Q \vee R)$.

If the right(left) hand side of an implication of a proposition is substituted by the left(right) hand side of the implication, the resulting proposition is logically implied by the original proposition. As a result, the new proposition is derived from the old.

As illustrated below, the tautologies described as "implications" can also be considered inference rules.

Rules of Inference	**Tautological form**	**Name**
P ---- $P \lor Q$	$P \Rightarrow (P \lor Q)$	Addition
P $P \to Q$ ------- Q	$(P \land Q) \Rightarrow Q$	Simplification
P $P \to Q$ -------- Q	$[P \land (P \to Q)] \Rightarrow Q$	Modus ponens
$\neg Q$ $P \to Q$ ------- $\neg P$	$[\neg Q \land (P \to Q)] \Rightarrow \neg P$	Modus tollens
$P \lor Q$ $\neg P$	$[(P \lor Q) \land \neg P] \Rightarrow Q$	Disjunctive syllogism
$P \to Q$ $Q \to R$ --------- $P \to R$	$[(P \to Q) \land (Q \to R)] \Rightarrow [P \to R]$	Hypothetical syllogism
P Q ------- $P \land Q$		Conjunction

Example

Consider the following proposition:

1. Is it Tuesday or Wednesday today?

2. But it can't be Wednesday because the doctor's office is open today, and Wednesdays are typically closed.

3. As a result, today is Tuesday.

As seen below, this chain of reasoning (inferencing) can be expressed as a succession of modus ponens applications to the appropriate propositions.

The modus ponens is an inference rule that deduces Q from P -> Q and P -> Q and P -> Q and P -> Q and P -> Q and P -> Q and P -> Q and P -> Q and P -> Q

T: It's the second day of the week.

W: It's Wednesday today.

D: Today the doctor's office is open.

C: On Wednesdays, the doctor's office is always closed.

Key takeaway

1. Propositional logic is concerned with assertions that can be assigned the truth values "true" and "false."
2. The goal is to evaluate these assertions individually and as a group.
3. Propositional variables and connectives make up a propositional. The propositional variables are denoted by capital letters (A, B, etc).
4. A tautology is a formula that holds true regardless of the value of its propositional variables.
5. A formula that is always untrue for every value of its propositional variables is called a contradiction.

6. A "sum-of-products" or disjunctive form is equivalent to every propositional formula. A disjunctive form is essentially an OR of AND-terms, where each AND-term is an AND of variables or variables' negations.

4.2 Predicate Logic: First order predicate, well formed formula of predicate, quantifiers, Inference theory of predicate logic

Predicate Logic deals with predicates, which are propositions containing variables.

First order predicate

A predicate is a set of one or more variables that are defined on a certain domain. A variable-based predicate can be turned into a proposition by either assigning a value to the variable or quantifying it.

Some examples of predicates are as follows:

- Let E(x, y) denote "x = y"
- Let X(a, b, c) denote "a + b + c = 0"
- Let M(x, y) denote "x is married to y"

Well formed formula of predicate

The Well Formed Formula (wff) is a predicate that holds one or more of the following:

- Wffs are used for all propositional constants and variables.
- If x is a variable and Y is a wff, ∀xY and ∃xY are also wff
- Truth value and false values are wffs
- Each atomic formula is a wff
- All connectives connecting wffs are wffs

Quantifiers

Quantifiers quantify the variable of predicates. In predicate logic, there are two types of quantifiers: Universal Quantifier and Existential Quantifier.

Universal Quantifier

The assertions within its scope are true for every value of the particular variable, according to the universal quantifier. It's represented by the symbol ∀.

∀xP(x) is read as for every value of x, P(x) is true.

Example - "Man is mortal" can be translated into the propositional form xP(x), where P(x) implies that x is mortal and that the universe of discourse is made up of all men.

Existential Quantifier

The assertions within its scope are true for some values of the specified variable, according to the existential quantifier. It's represented by the symbol ∃.

∃xP(x) is read as for some values of x, P(x) is true.

Example - "Some people are dishonest," for example, can be translated into the propositional form xP(x), with P(x) denoting x is dishonest and some people denoting the universe of discourse.

Nested Quantifiers

The term "nested quantifier" refers to a quantifier that appears within the scope of another quantifier.

Example - ∀ a∃bP(x,y) where P(a,b) denotes a+b=0

∀ a∀b∀cP(a,b,c) where P(a,b) denotes a+(b+c)=(a+b)+c

Inference theory of predicate logic

A series of claims is referred to as an argument. The conclusion is the last assertion, and the premises are the ones before it (or hypothesis). Before the conclusion, the symbol "∴" (read thus) is placed. The conclusion of a valid argument flows from the truth values of the premises.

The templates or instructions for creating valid arguments from the statements we already know are provided by Rules of Inference.

Simple arguments can be used as the foundation for more complex valid arguments. Certain simple arguments that have been proven to be true are extremely essential in terms of their application. These are known as Rules of Inference arguments.

The most widely used Inference Rules are listed below —

Rules of Inference	**Tautological form**	**Name**
$\begin{array}{l} p \\ p \to q \\ \hline \therefore q \end{array}$	$p \Rightarrow (p \vee q)$	Modus Ponens
$\begin{array}{l} \neg q \\ p \to q \\ \hline \therefore \neg p \end{array}$	$(\neg q \wedge (p \to q)) \to \neg p$	Modus Tollens
$\begin{array}{l} p \to q \\ q \to r \\ \hline \therefore p \to r \end{array}$	$((p \to q) \wedge (q \to r)) \to (p \to r)$	Hypothetical syllogism
$\begin{array}{l} \neg p \\ p \vee q \\ \hline \therefore q \end{array}$	$(\neg p \wedge (p \vee q)) \to q$	Disjunctive syllogism
$\begin{array}{l} p \\ \hline \therefore (p \vee q) \end{array}$	$p \to (p \vee q)$	Addition
$\begin{array}{l} (p \wedge q) \to r \\ \hline \therefore p \to (q \to r) \end{array}$	$((p \wedge q) \to r) \to (p \to (q \to r))$	Exportation
$\begin{array}{l} p \vee q \quad \neg p \vee r \\ \hline \therefore q \vee r \end{array}$	$(p \vee q) \wedge (\neg p \vee r) \to q \vee r$	Resolution

Similarly, we have Rules of Inference for quantified statements –

Rule of Inference	**Name**
$\frac{\forall x P(x)}{\therefore P(c)}$	Universal instantiation
$\frac{P(c) \text{ for an arbitrary } c}{\therefore \forall x P(x)}$	Universal generalization
$\frac{\exists x P(x)}{\therefore P(c) \text{ for some } c}$	Existential instantiation
$\frac{\therefore P(c) \text{ for some } c}{\therefore \exists x P(x)}$	Existential generalization

Key takeaway

1. Predicate Logic deals with predicates, which are propositions containing variables.
2. A predicate is a set of one or more variables that are defined on a certain domain.
3. A variable-based predicate can be turned into a proposition by either assigning a value to the variable or quantifying it.
4. Quantifiers quantify the variable of predicates. In predicate logic, there are two types of quantifiers: Universal Quantifier and Existential Quantifier.
5. A series of claims is referred to as an argument. The conclusion is the last assertion, and the premises are the ones before it (or hypothesis).

Unit - 5

Trees, Graphs and Combinatorics

5.1 Trees: Definition

An acyclic graph is one that does not have a cycle. A tree is a graph with no cycles, or an acyclic graph.

A tree, also known as a general tree, is a non-empty finite set of items called vertices or nodes that have the property of having a minimum degree of 1 and a maximum degree of n. It can be divided into n+1 disjoint subsets, with the first subset being the tree's root and the other n subsets containing the n subtree's members.

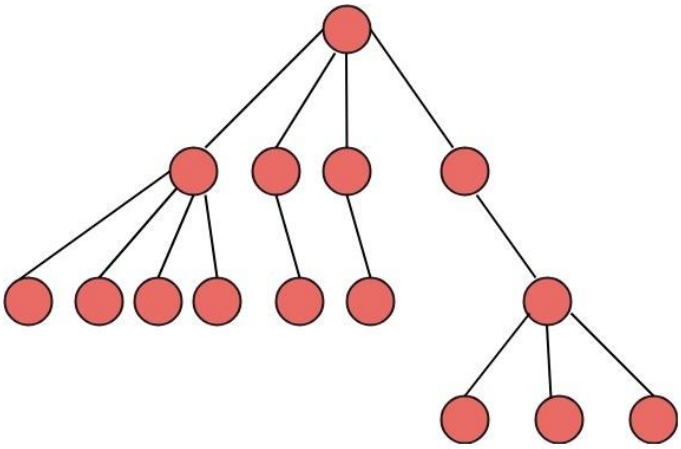


Fig: General tree

Key takeaway

An acyclic graph is one that does not have a cycle. A tree is a graph with no cycles, or an acyclic graph.

5.2 Binary tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.

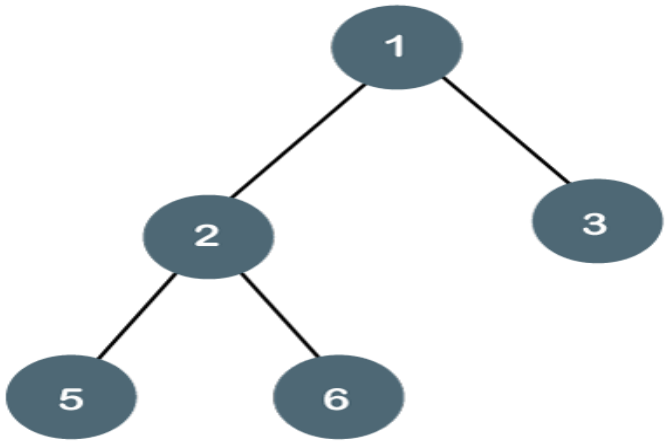


Fig - Example

The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:

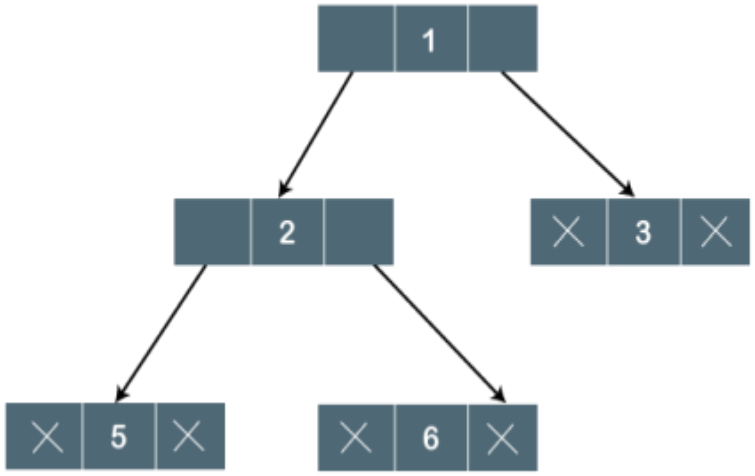


Fig - Logical representation

In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.

- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.

- The minimum number of nodes possible at height h is equal to **h+1**.

- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$Log_2(n+1) = log2(2^{h+1})$

$Log_2(n+1) = h+1$

$h = log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

h= n-1

Types of Binary Tree

There are four types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

1. **Full/ proper/ strict Binary tree**

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.

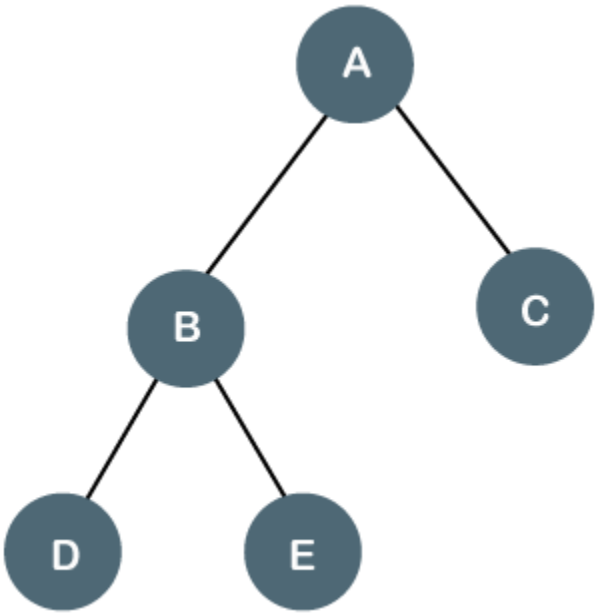


Fig - Example

In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., $2^{h+1} -1$.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

n= 2*h - 1

n+1 = 2*h

h = n+1/2

2. Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.

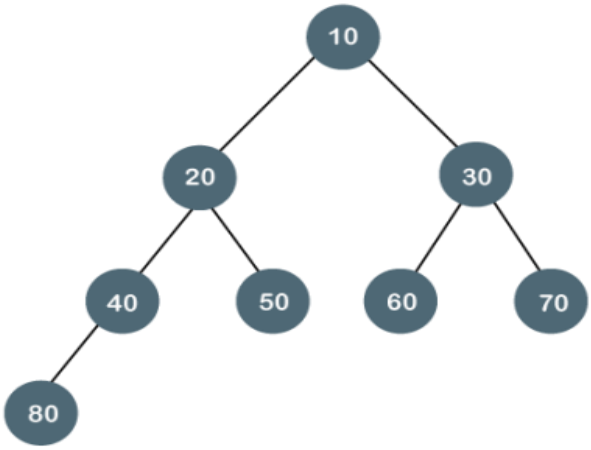


Fig - Complete binary tree

The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

3. Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.

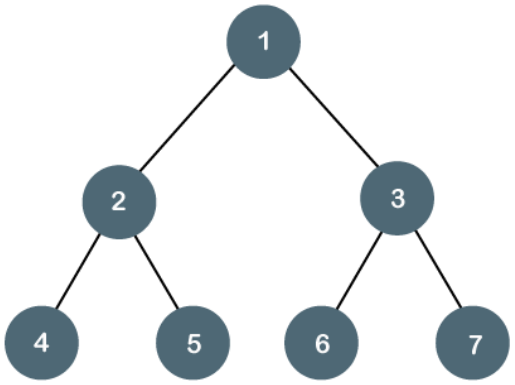


Fig - Perfect Binary Tree

Let's look at a simple example of a perfect binary tree.
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

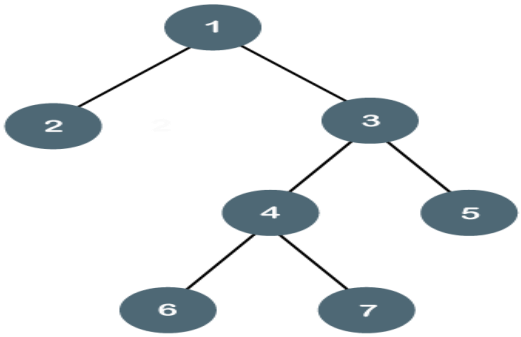


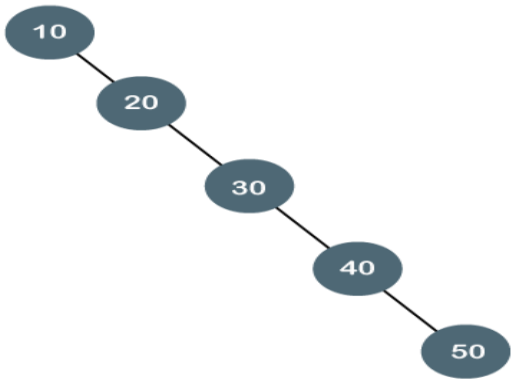
Fig – Example

Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

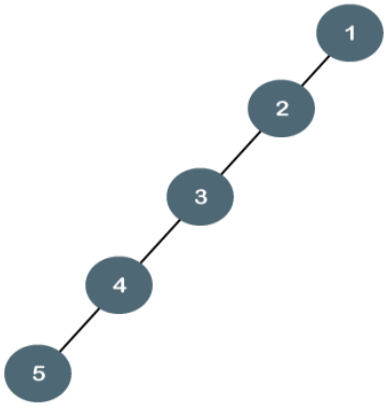
4. Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one child.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

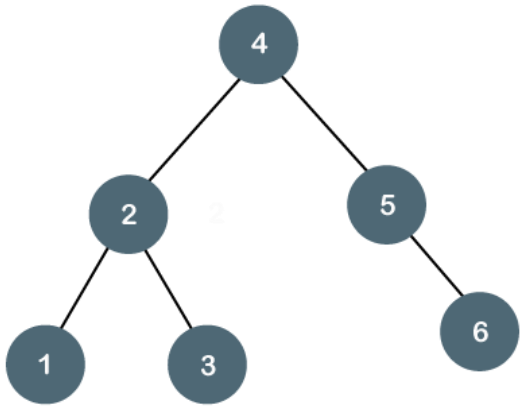


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

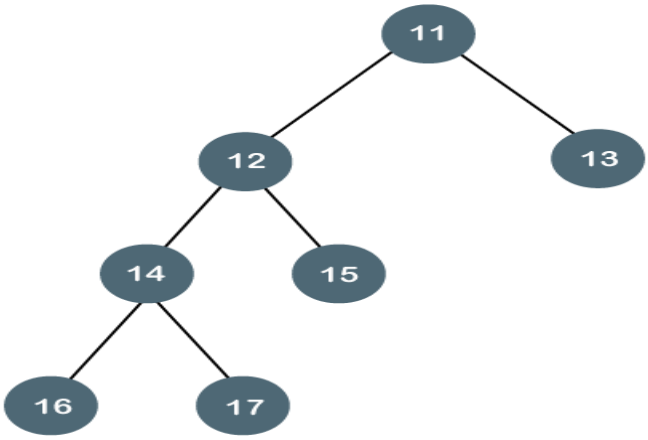
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by at most 1. For example, **AVL** and **Red-Black trees** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node.

We will create the node of user-defined as shown below:

```
1. Struct node
2. {
3. Int data,
4. Struct node *left, *right;
5. }
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary Tree program in C

```
1.  #include<stdio.h>
2.  Struct node
3.  {
4.  Int data;
5.  Struct node *left, *right;
6.  }
7.  Void main()
8.  {
9.  Struct node *root;
10. Root = create();
11. }
12. Struct node *create()
13. {
14. Struct node *temp;
```

```
15. Int data;
16. Temp = (struct node *)malloc(sizeof(struct node));
17. Printf("Press 0 to exit");
18. Printf("\nPress 1 for new node");
19. Printf("Enter your choice : ");
20. Scanf("%d", &choice);
21. If(choice==0)
22. {
23. Return 0;
24. }
25. Else
26. {
27. Printf("Enter the data:");
28. Scanf("%d", &data);
29. Temp->data = data;
30. Printf("Enter the left child of %d", data);
31. Temp->left = create();
32. Printf("Enter the right child of %d", data);
33. Temp->right = create();
34. Return temp;
35. }
36. }
```

The above code is calling the create() function recursively and creating new node on each recursive call. When all the nodes are created, then it forms a binary tree structure.

Key takeaway

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

1. The full binary tree is also known as a strict binary tree.
2. The complete binary tree is a tree in which all the nodes are completely filled except the last level.
3. A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.
4. The degenerate binary tree is a tree in which all the internal nodes have only one children.

5. 3 Binary tree traversals

The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- In-order traversal
- Pre-order traversal
- Post-order traversal

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right subtree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

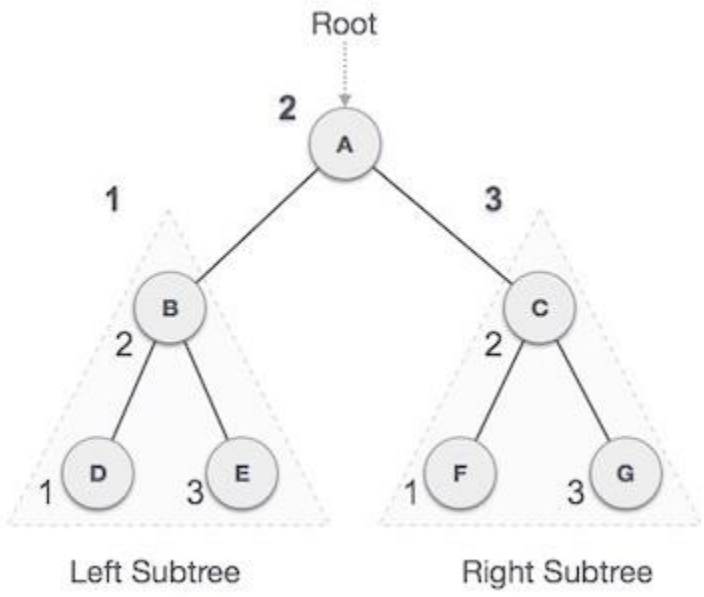


Fig - Inorder

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

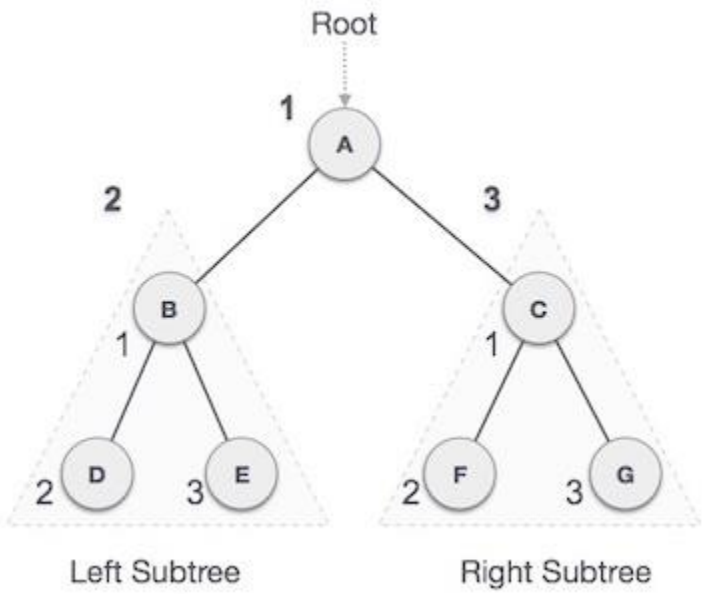


Fig – Pre order

We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.

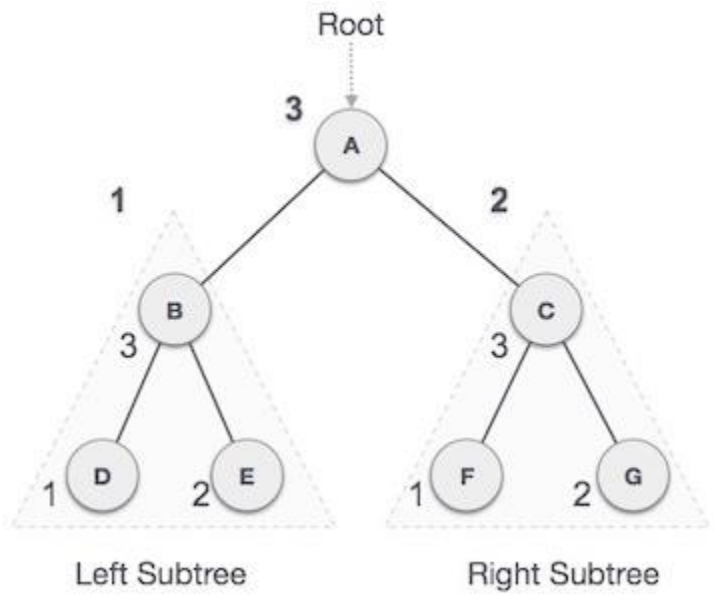


Fig – Post order

We start from **A**, and following post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

Tree Traversal in C

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

We shall now look at the implementation of tree traversal in C programming language here using the following binary tree –

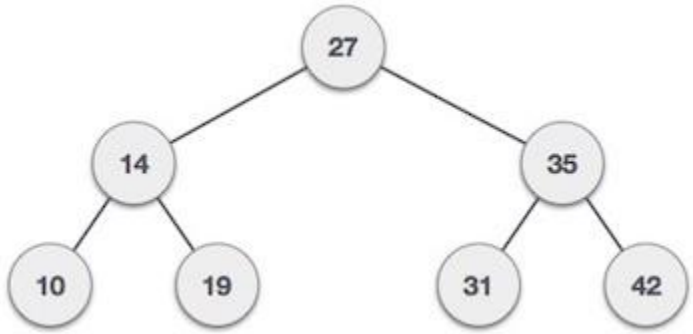


Fig – Example

Implementation in C

```
#include <stdio.h>
#include <stdlib.h>
Struct node {
Int data;
Struct node *leftChild;
Struct node *rightChild;
};
Struct node *root = NULL;
Void insert(int data) {
Struct node *tempNode = (struct node*) malloc(sizeof(struct node));
Struct node *current;
Struct node *parent;
TempNode->data = data;
TempNode->leftChild = NULL;
TempNode->rightChild = NULL;
//if tree is empty
If(root == NULL) {
Root = tempNode;
} else {
Current = root;
Parent = NULL;
While(1) {
Parent = current;
//go to left of the tree
If(data < parent->data) {
Current = current->leftChild;
//insert to the left
If(current == NULL) {
Parent->leftChild = tempNode;
Return;
}
}  //go to right of the tree
Else {
Current = current->rightChild;
//insert to the right
```

```
If(current == NULL) {
Parent->rightChild = tempNode;
Return;
}
}
}
}
}
Struct node* search(int data) {
Struct node *current = root;
Printf("Visiting elements: ");
While(current->data != data) {
If(current != NULL)
Printf("%d ",current->data);
//go to left tree
If(current->data > data) {
Current = current->leftChild;
}
//else go to right tree
Else {
Current = current->rightChild;
}
//not found
If(current == NULL) {
Return NULL;
}
}
Return current;
}

Void pre_order_traversal(struct node* root) {
If(root != NULL) {
Printf("%d ",root->data);
Pre_order_traversal(root->leftChild);
Pre_order_traversal(root->rightChild);
}
}

Void inorder_traversal(struct node* root) {
If(root != NULL) {
Inorder_traversal(root->leftChild);
Printf("%d ",root->data);
Inorder_traversal(root->rightChild);
}
}

Void post_order_traversal(struct node* root) {
If(root != NULL) {
Post_order_traversal(root->leftChild);
Post_order_traversal(root->rightChild);
Printf("%d ", root->data);
}
}

Int main() {
Int i;
Int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
For(i = 0; i < 7; i++)
Insert(array[i]);
i = 31;
Struct node * temp = search(i);
```

```
If(temp != NULL) {
Printf("[%d] Element found.", temp->data);
Printf("\n");
}else {
Printf("[ x ] Element not found (%d).\n", i);
}
i = 15;
Temp = search(i);
If(temp != NULL) {
Printf("[%d] Element found.", temp->data);
Printf("\n");
}else {
Printf("[ x ] Element not found (%d).\n", i);
}
Printf("\nPreorder traversal: ");
Pre_order_traversal(root);
Printf("\nInorder traversal: ");
Inorder_traversal(root);
Printf("\nPost order traversal: ");
Post_order_traversal(root);
Return 0;
}
```

If we compile and run the above program, it will produce the following result –

Output

Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [x] Element not found (15).
Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Key takeaway

1. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node.
2. In order traversal method, the left subtree is visited first, then the root and later the right subtree. We should always remember that every node may represent a subtree itself.
3. Pre-order traversal method, the root node is visited first, then the left subtree and finally the right subtree.
4. Post-order traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.

5.4 Binary search tree

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both

an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

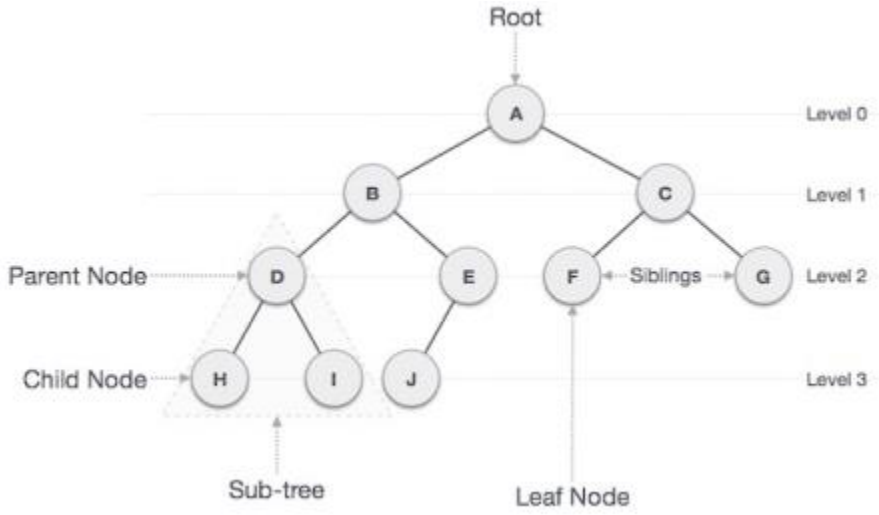


Fig - Binary Tree

Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

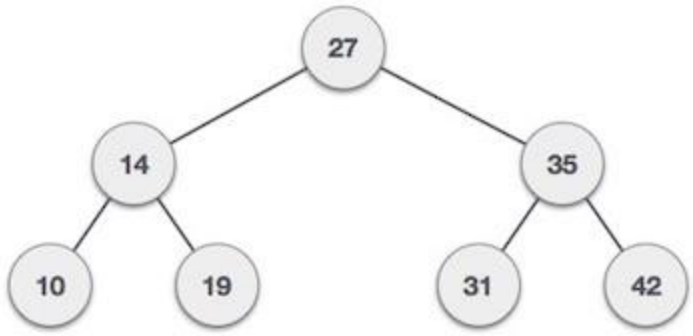


Fig - Binary Search tree

We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

Struct node {

Int data;

Struct node *leftChild;

Struct node *rightChild;

};

In a tree, all nodes share a common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

If root is NULL

```
Then create root node
Return
If root exists then
Compare the data with node.data
While until insertion position is located
If data is greater than node.data
Goto right subtree
Else
Goto left subtree
End while
Insert data
End If
```

Implementation

The implementation of insert function should look like this –

```
Void insert(int data) {
Struct node *tempNode = (struct node*) malloc(sizeof(struct node));
Struct node *current;
Struct node *parent;
TempNode->data = data;
TempNode->leftChild = NULL;
TempNode->rightChild = NULL;
//if tree is empty, create root node
If(root == NULL) {
Root = tempNode;
} else {
Current = root;
Parent  = NULL;
While(1) {
Parent = current;
//go to left of the tree
If(data < parent->data) {
Current = current->leftChild;
//insert to the left
If(current == NULL) {
Parent->leftChild = tempNode;
Return;
}
}
//go to right of the tree
Else {
Current = current->rightChild;
//insert to the right
If(current == NULL) {
Parent->rightChild = tempNode;
Return;
}
}
}
}
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
Return root
```

Else
While data not found
If data is greater than node.data
Goto right subtree
Else
Goto left subtree
If data found
Return node
End while
Return data not found
End if

Implementation

The implementation of this algorithm should look like this.

```
Struct node* search(int data) {
Struct node *current = root;
Printf("Visiting elements: ");
While(current->data != data) {
If(current != NULL)
Printf("%d ",current->data);
//go to left tree
If(current->data > data) {
Current = current->leftChild;
}
//else go to right tree
Else {
Current = current->rightChild;
}
//not found
If(current == NULL) {
Return NULL;
}
Return current;
}
}
```

Key takeaway

1. The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

5.5 Graphs: Definition and terminology

Graph

A graph can be defined as a group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

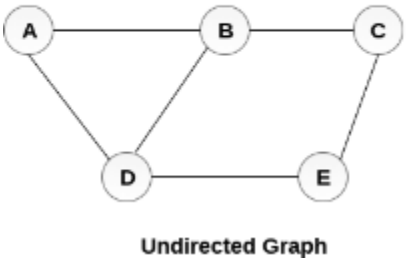


Fig – Undirected graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.

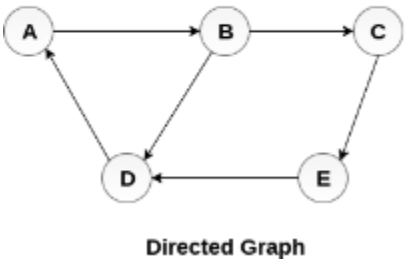


Fig – Directed graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Key takeaway

1. A graph can be defined as group of vertices and edges that are used to connect these vertices.
2. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

5.6 Representation of graphs

Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

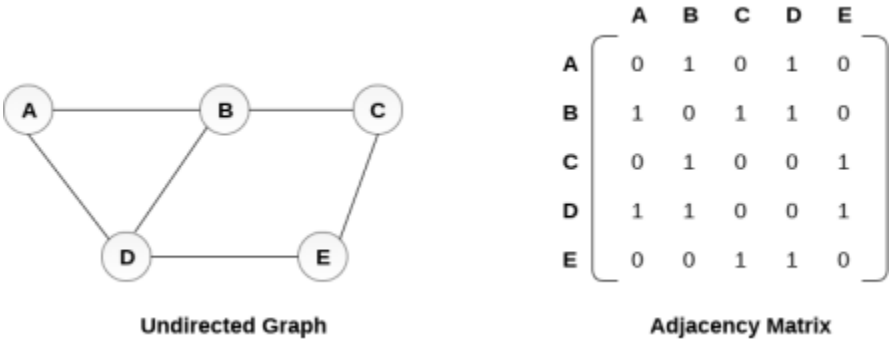
There are two ways to store Graph into the computer's memory.

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension **n x n**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



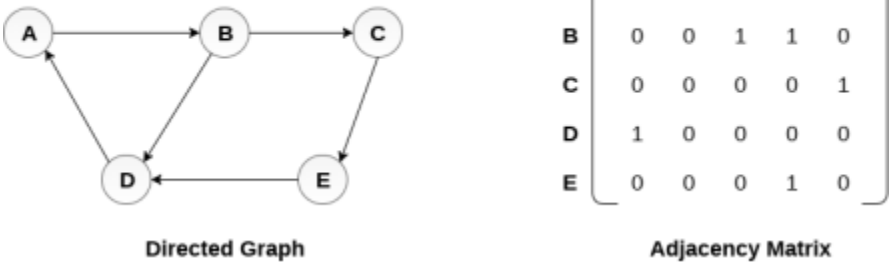
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Fig – Undirected graph and its adjacency matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

A directed graph and its adjacency matrix representation is shown in the following figure.



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

Fig - Directed graph and its adjacency matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.

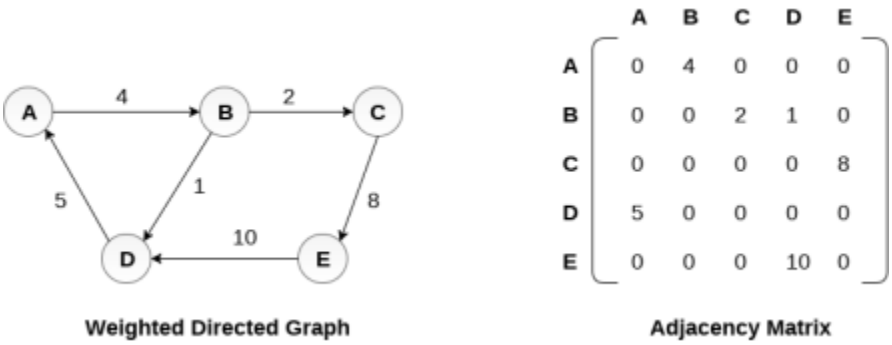


Fig - Weighted directed graph

Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.

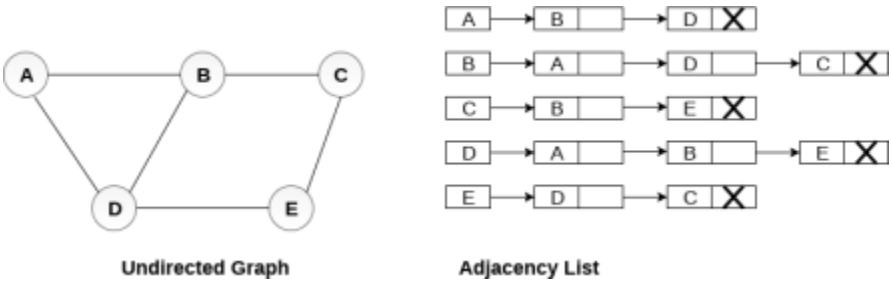


Fig - Undirected graph in adjacency list

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

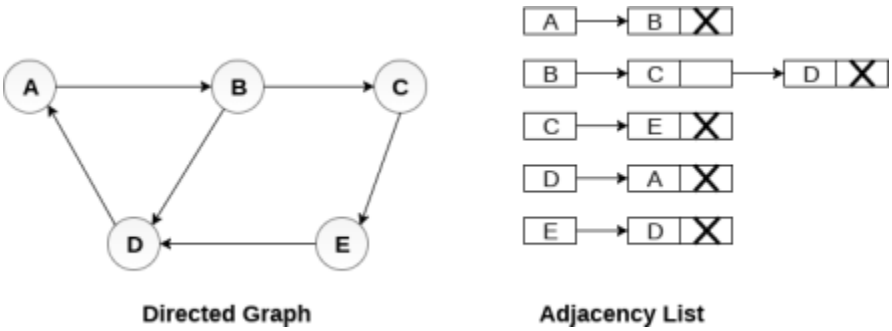


Fig - Directed graph in adjacency list

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.

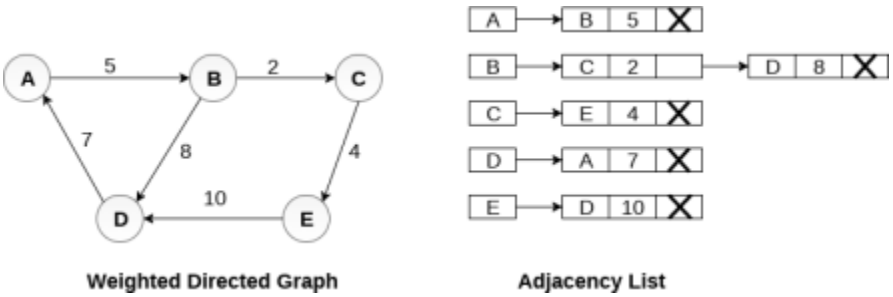


Fig - Weighted directed graph

Key takeaway

1. By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.
2. There are two ways to store Graph into the computer's memory.

5.7 Multigraphs

Multigraph is a graph in which numerous edges between the same set of vertices are allowed. To put it another way, it's a graph with at least one loop and numerous edges.

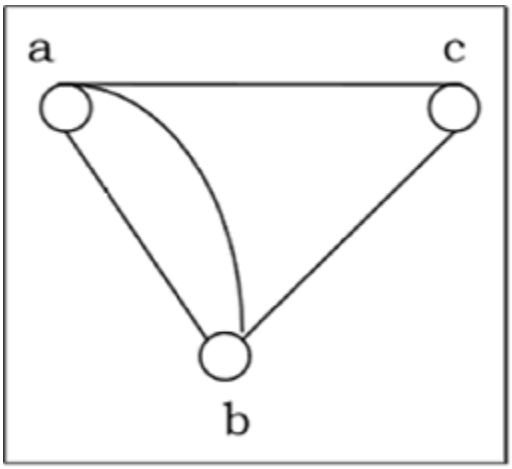


Fig: Multigraph

5.8 Bipartite graphs

A bipartite graph G=(V, E) has vertices V that can be partitioned into two subsets V1 and V2, with each edge of G connecting a vertex of V1 to a vertex of V2. Kmn is the symbol for it, where m and n are the vertices in V1 and V2, respectively.

Example - Draw the bipartite graphs K2, 4 and K3,4. Assume that there are any number of edges.

Solution - Draw the appropriate number of vertices on two parallel columns or rows, then connect the vertices in one column or row to the vertices in the other column or row. The bipartite graphs K2,4 and K3,4 are depicted in the figures.

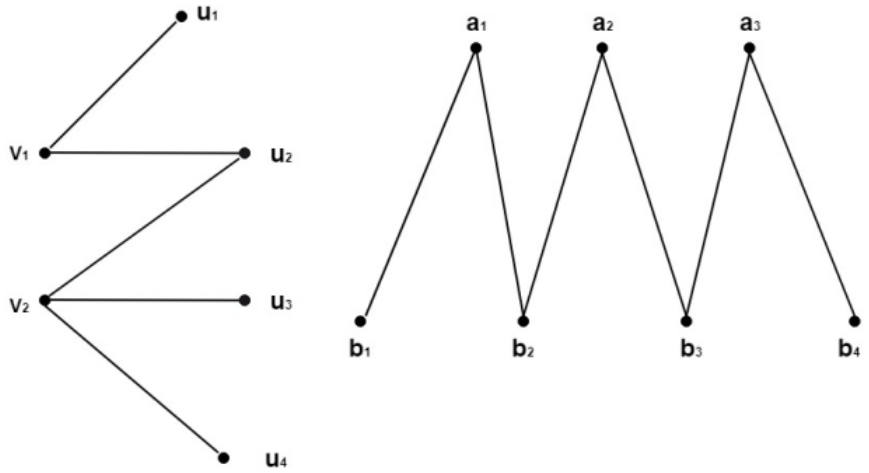


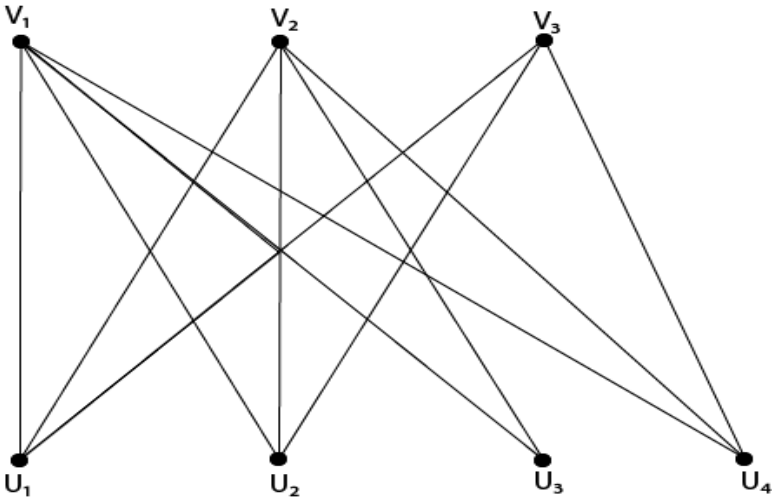
Fig: Bipartite graphs

Complete bipartite graph

If the vertices V of a graph G = (V, E) can be partitioned into two subsets V1 and V2, each vertex of V1 is connected to each vertex of V2, the graph is called a full bipartite graph. Because each of the m vertices is connected to each of the n vertices, a complete bipartite graph has m.n edges.

Example - Draw the entire bipartite graphs K3,4 and K1,5 as an example.

Solution - Draw the appropriate number of vertices in two parallel columns or rows, then connect the vertices in the first column or row to all of the vertices in the second column or row. Figure shows the graphs K3,4 and K1,5.



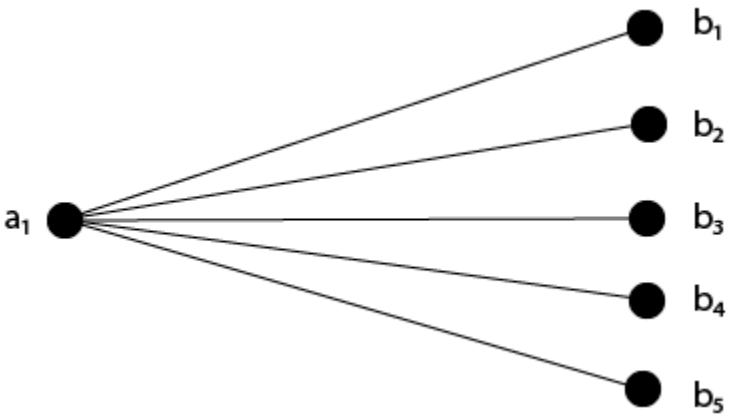


Fig: Graphs K3,4 and K1,5

5.9 Planar graphs

"A diagram is supposed to be planar if it very well may be drawn on a plane with no edges crossing. Such a drawing is known as a planar portrayal of the graph."

A chart might be planar regardless of whether it is drawn with intersections, since it very well might be conceivable to attract it an alternate route without intersections.

For instance, consider the total chart K_{4} and its two potential planar portrayals –

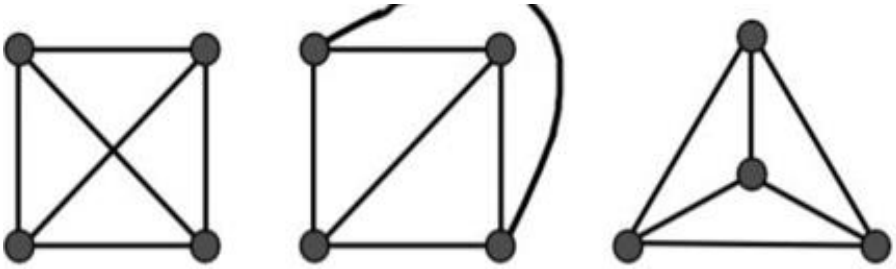


Fig: Planner graph

Regions in Planar Graphs –

The planar portrayal of a diagram parts the plane into areas. These locales are limited by the edges with the exception of one district that is unbounded. For instance, think about the accompanying chart
"

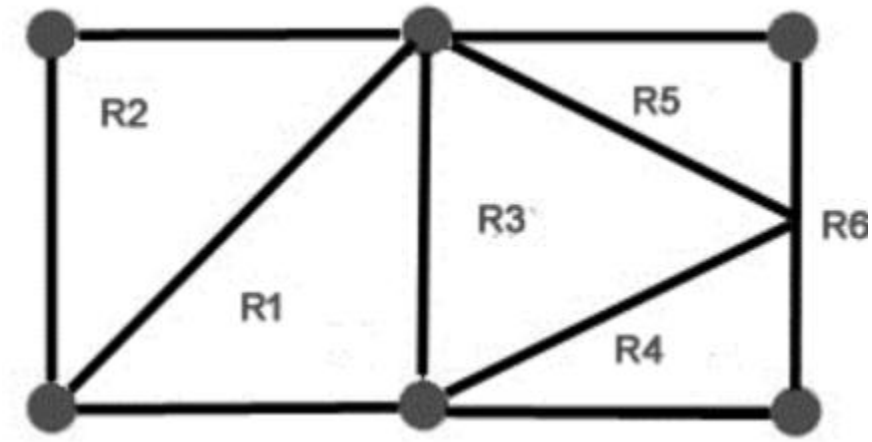


Fig: Regions planar graph

There are a sum of 6 regions with 5 limited regions and 1 unbounded area R{6}.

Key takeaway

1. "A diagram is supposed to be planar if it very well may be drawn on a plane with no edges crossing.
2. Such a drawing is known as a planar portrayal of the graph."

5.10 Isomorphism and Homeomorphism of graphs

Graph Isomorphism

Graph isomorphism is an equivalence relationship on graphs which, as such, splits the class into equivalence groups on all graphs. An isomorphism class of graphs is called a group of graphs isomorphic to one another. The two graphs shown below are isomorphic, despite their different looking drawings.

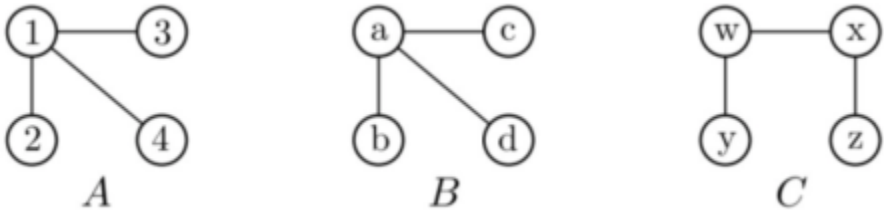


Fig: Isomorphism

Two isomorphic plots A and B and one non-isomorphic plot C; Each of them has four vertices with three corners.

Homeomorphism of graphs

If two graphs G and G* can be produced by this approach from the same graph or isomorphic graphs, they are said to be homeomorphic. The graphs (a) and (b) are not isomorphic, but they are homeomorphic since they may be created by adding appropriate vertices to the graph (c).

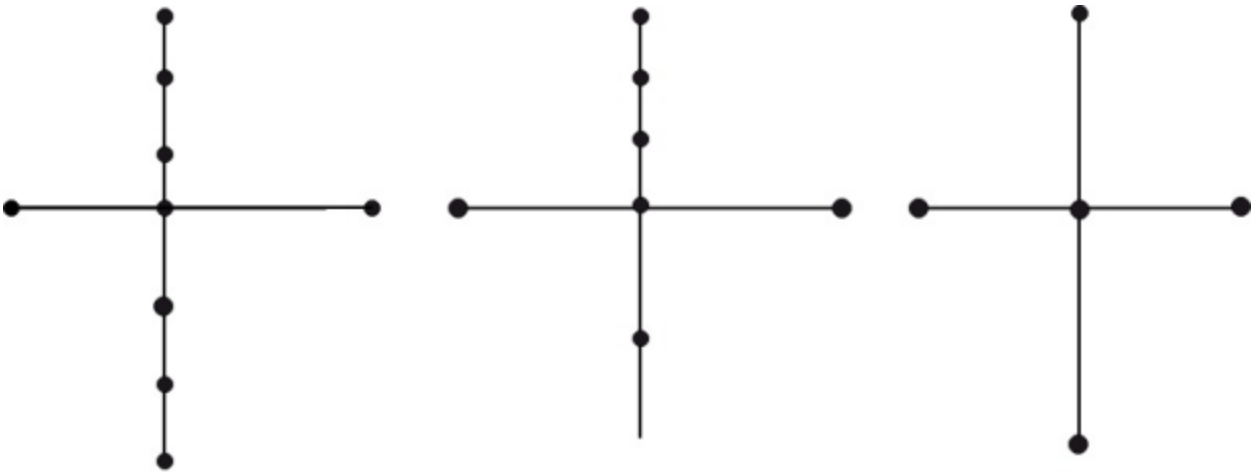


Fig:a Fig:b Fig:c

Fig: Homeomorphism graph

Properties

- If a homomorphism is a bijective mapping, it is an isomorphism.
- Homomorphism always retains a graph's edges and connectedness.
- Homomorphisms' compositions are also homomorphisms.
- It's an NP complete problem to figure out if there's any homomorphic graph of another graph.

Key takeaway

1. Graph isomorphism is an equivalence relationship on graphs which, as such, splits the class into equivalence groups on all graphs.
2. If two graphs G and G* can be produced by this approach from the same graph or isomorphic graphs, they are said to be homeomorphic.

5.11 Euler and Hamiltonian paths

A Euler Path through a graph is a path whose edge list contains each edge of the graph exactly once.

Euler Circuit: A Euler Circuit is a path through a graph, in which the initial vertex appears a second time as the terminal vertex.

Euler Graph: A Euler Graph is a graph that possesses a Euler Circuit. A Euler Circuit uses every edge exactly once, but vertices may be repeated.

Example - The graph shown in fig is a Euler graph. Determine Euler Circuit for this graph.

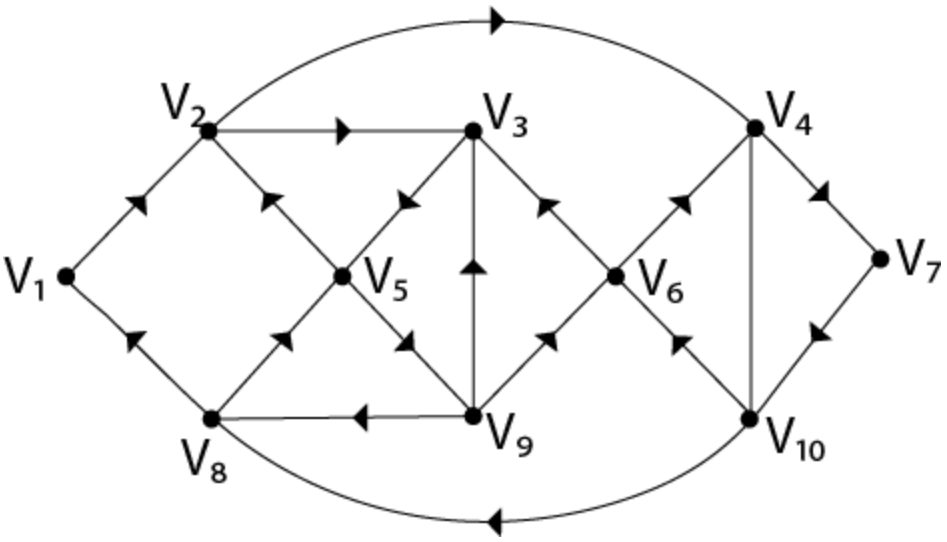


Fig: Euler graph

Solution - The Euler Circuit for this graph is

V_1, V_2, V_3, V_5, V_2, V_4, V_7, V_{10}, V_6, V_3, V_9, V_6, V_4, V_{10}, V8, V_5, V_9, V8, V_1

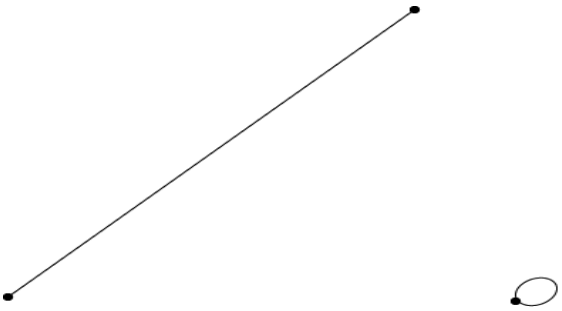
For a connected network with no vertices of odd degrees, we can create a Euler Circuit.

State and Prove Euler's Theorem:

Consider any linked planar network with R regions, V vertices, and E edges, G= (V, E). V+R-E=2 in this case.

Proof: To prove this theorem, use induction on the number of edges.

Basis of Induction: Assume that each edge has the value e=1.



Then there are two examples, both of which have graphs in fig:

We have V=2 and R=1 in Fig. As a result, 2+1-1=2.

V=1 and R=2 is shown in Fig. As a result, 1+2-1=2.

As a result, the induction's foundation is established.

Induction step: Assume the formula is valid for connected planar graphs with K edges.

Consider the graph G, which has K+1 edges.

To begin, we assume that G is devoid of circuits. Take a vertex v and create a path that starts at v. We have a new vertex whenever we locate an edge in G since it is a circuit-free language. Finally, we'll arrive at a vertex v with degree 1. As a result, we are unable to proceed as depicted in fig.
Remove the corresponding edge incident on v and the vertex v. As a result, we have a graph G* with K edges, as shown in fig.

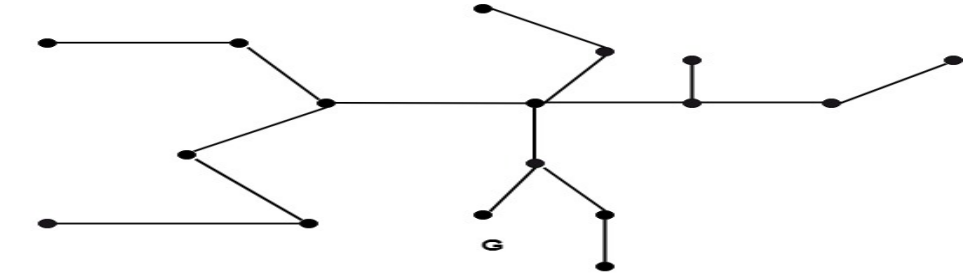


Fig:G

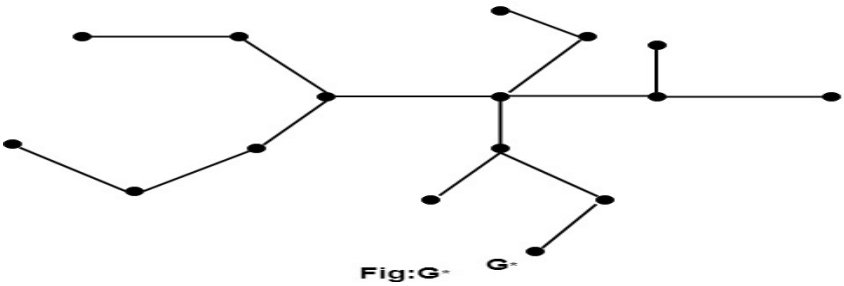
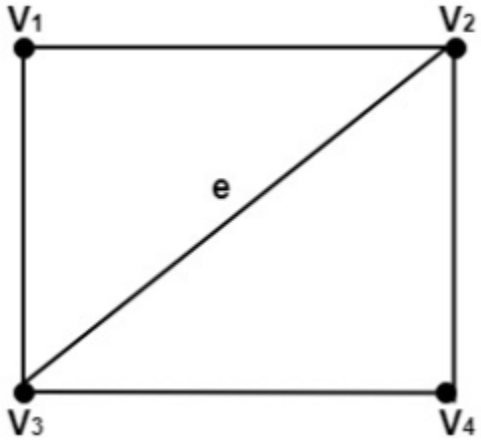


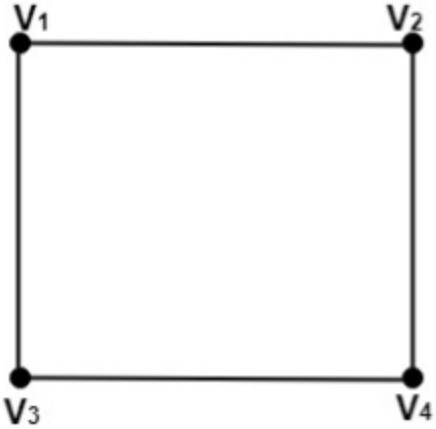
Fig:G*

As a result, Euler's formula holds for G* by inductive assumption.

Now, G has one extra edge and one more vertex than G*, with the same number of regions. As a result, the formula also applies to G.

Second, we suppose that G contains a circuit and that e is an edge in the circuit depicted in figure:





Now, because e is a portion of a two-region boundary. As a result, we merely delete the edge, leaving us with a graph G* with K edges.

As a result, Euler's formula holds for G* by inductive assumption.

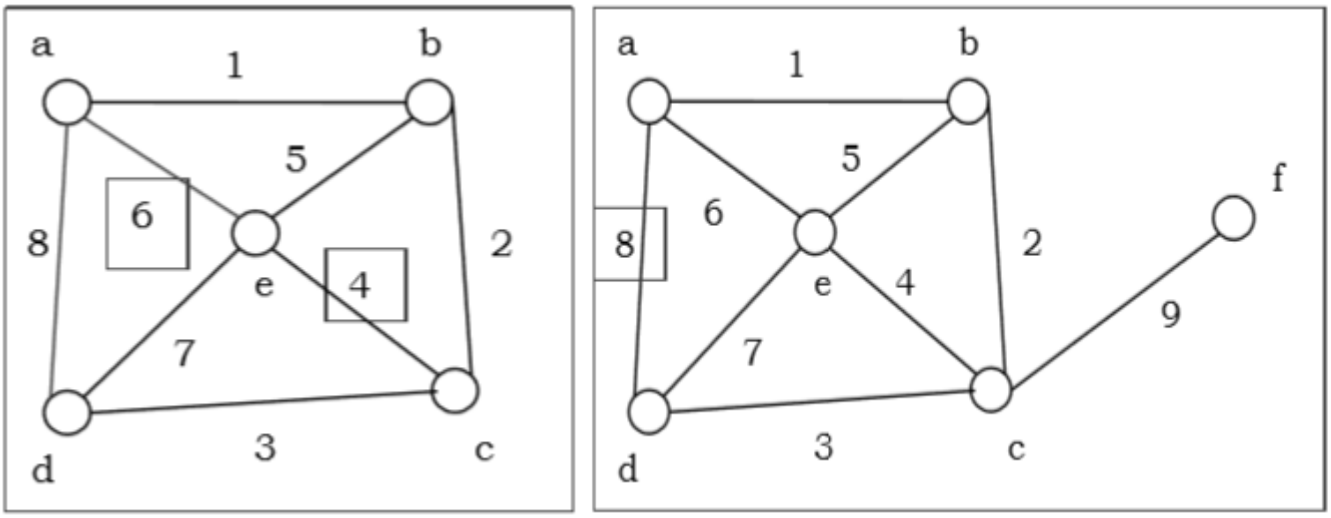
G now has one more edge than G*, as well as one more area with the same number of vertices as G*. As a result, the formula also holds for G, proving the thesis by verifying the inductive steps.

Hamiltonian graph

A graph that is connected If a cycle that includes every vertex of G is called a Hamiltonian cycle, then G is called a Hamiltonian graph. In graph G, a Hamiltonian walk is one that passes through each vertex precisely once.

If G is a simple graph with n vertices, where n≥3 is the number of vertices, then if each vertex v has a deg(v) ≥ n/2 value, then the graph G is a Hamiltonian graph. This is referred to as Dirac's Theorem.

G is a Hamiltonian graph if it is a simple graph with n vertices, where n≥2 is deg(x)+deg(y) ≥ n for each pair of non-adjacent vertices x and y. Ore's theorem is the name for this.



Key takeaway

1. A Euler Path through a graph is a path whose edge list contains each edge of the graph exactly once.
2. A graph that is connected If a cycle that includes every vertex of G is called a Hamiltonian cycle, then G is called a Hamiltonian graph. In graph G, a Hamiltonian walk is one that passes through each vertex precisely once.

5.12 Graph coloring

Diagram shading is the methodology of task of tones to every vertex of a chart G with the end goal that no adjoining vertices get same tone. The goal is to limit the quantity of shadings while shading a chart. The most modest number of tones needed to shading a diagram G is called its chromatic number of that chart. Diagram shading issue is a NP Complete issue.

Technique to Colour a Graph

The means needed to shading a diagram G with n number of vertices are as per the following –

Stage 1 – Arrange the vertices of the chart in some request.

Stage 2 – Choose the main vertex and shading it with the principal tone.

Stage 3 – Choose the following vertex and shading it with the most reduced numbered shading that has not been hued on any vertices neighboring it. In the event that all the contiguous vertices are hued with this tone, dole out another tone to it. Rehash this progression until all the vertices are shaded.

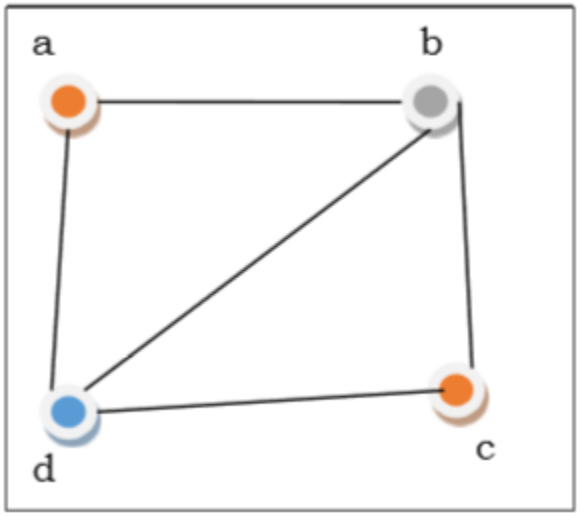


Fig: Graph colouring

In the above figure, from the start vertex an is shaded red. As the contiguous vertices of vertex an are again neighbouring, vertex b and vertex d are shaded with various tones, green and blue individually. At that point vertex c is hued however red as no adjoining vertex of c may be hued red. Consequently, we could colour the chart by 3 tones. Consequently, the chromatic number of the chart is 3.

5.13 Recurrence Relation & Generating function

Recurrence Relation

The process for recursively finding the terms of a sequence is called the recurrence relation.

Definition:

A recurrence relationship is an equation that represents a series recursively where a function of the previous terms is the next term (Expressing F_n as some combination of F_i with $i<n$).

Example – Fibonacci series – $Fn=Fn-1+Fn-2$ Tower of Hanoi – $Fn=2Fn-1+1$

Generating function:

Generating functions are sequences in which each sequence term is expressed in a formal power series as a coefficient of a variable x.

Mathematically, the generating function would be – for an infinite number, say $a_0,a_1,a_2,\ldots,a_k,\ldots$

$$G_x = a_0 + a_1x + a_2x^2 + \ldots + a_kx^k + \ldots = \sum_{k=0}^{\infty} a_kx^k$$

Some Domain Fields

For the following reasons, generating functions may be used –

- For solving a number of problems with counting. For instance, the number of ways to modify a note of Rs. 100 with denominations of Rs.1, Rs.2, Rs.5, Rs.10, Rs.20 and Rs.50 notes

- To solve relationships with recurrence

- Any of the combinatorial identities to prove

- For the finding of asymptotic formulas for sequence words

5.14 Recursive definition of functions

In logic and mathematics, a recursive function is a type of function or expression that predicts some concept or property of one or more variables and is defined by a procedure that generates values or instances of the function by applying a given relation or routine operation to known values of the function repeatedly.

Thoralf Albert Skolem, a pioneer in metalogic in the twentieth century, created the theory of recursive functions as a way of avoiding the so-called paradoxes of the infinite that arise in particular cases when "all" is applied to functions that span infinite classes, it accomplishes this by defining a function's range without referring to limitless classes of entities.

Taking a recognizable term like "human"—or the function "x is human"—recursion can be intuitively conveyed. "Adam and Eve are human; and any offspring of theirs is human; and any offspring of offspring... Of their offspring is human," one can say instead of defining this notion or function by its features and dispositions.

The method known as mathematical induction is strongly related to this recursiveness in a function or notion, and it is primarily important in logic and mathematics. "x is a logical system L formula" or "x is a natural number," for example, are typically defined recursively.

There are two pieces to a recursive definition:

1. Definition of the smallest argument (usually f (0) or f (1)).
2. Definition of f (n), given f (n - 1), f (n - 2), etc.

An example of a recursively defined function is as follows:

$$f(0) = 5$$
$$f(n) = f(n-1) + 2$$

This function's values can be calculated as follows:

f (0) =5

f (1) =f (0) + 2 = 5 + 2 = 7

f (2) =f (1) + 2 = 7 + 2 = 9

f (3) =f (2) + 2 = 9 + 2 = 11

The explicitly defined function f (n) = 2n + 5 is equivalent to this recursively defined function. The recursive function, on the other hand, is only defined for nonnegative numbers.

Key takeaway

1. In logic and mathematics, a recursive function is a type of function or expression that predicts some concept or property of one or more variables and is defined by a procedure that generates values or instances of the function by applying a given relation or routine operation to known values of the function repeatedly.

5.15 Recursive algorithms

If an algorithm solves an issue by reducing it to a smaller version of the same problem, it is called recursive.

Recursion is a type of concept and method that is crucial in both theory and practice in computer science. It is possible for recursive algorithms to be inefficient or efficient. Self-reference is a characteristic of a recursive definition or algorithm. A function is typically defined in terms of an earlier version of itself when using recursion. There must be a termination condition because this self-reference can't continue on forever. The method checks the termination condition first, and if it doesn't apply, it moves on to the self-reference.

Example: The factorial function, which is generally defined by n! = n.(n–1).(n–2).(n–3)...3.2.1 . For positive numbers and 0! = 1 for negative integers, is a classic example of a recursive definition. The following is a recursive definition of n factorial:

0! = 1

N! = N.(N!1)!; N > 0

5! = 5.4! is the formula for evaluating 5! In order to evaluate 4!, we must return to the definition, which yields 4! = 4.3! As a result, 5! = 5.4.3! Likewise, 3! = 3.2! As a result, 5! = 5.4.3.2!, 2! = 2.1! and 1! = 1.0! are the two possibilities. The first portion of the definition, however, yields 0! = 1. 5! = 5.4.3.2.1 = 120.

Now, while the preceding example may appear awkward, recursive definitions are typically easier to develop and debug in a computer environment than non-recursive definitions. The compiler is in charge of the grunt labor and record keeping. A definition similar to the two-line definition given above will be included in the program.

Key takeaway

1. If an algorithm solves an issue by reducing it to a smaller version of the same problem, it is called recursive.
2. Recursion is a type of concept and method that is crucial in both theory and practice in computer science. It is possible for recursive algorithms to be inefficient or efficient.

5.16 Method of solving recurrences

Take a look at the recurrence relationship.

$a_n = 5a_{n-1} - 6a_{n-2}$

1. If the initial conditions are $a_0=1$, $a_1=2$, what sequence do you get? For this sequence, give a closed formula.
2. If the initial conditions are $a_0=1$, $a_1=3$, what sequence do you get? Give a formula that is closed.
3. What if $a_0=2$, $a_1=5$? Find a closed formula.

We've shown that recursive definitions are frequently easier to locate than closed formulas. There are a few strategies for turning recursive definitions to closed formulas, which is fortunate for us. Solving a recurrence relation is the term for this. Keep in mind that the recurrence relation is a recursive definition that does not include the initial conditions. The Fibonacci sequence, for example, has a recurrence relation of $F_n = F_{n-1} + F_{n-2}$.

(This, along with the initial conditions $F_0 = 0$, $F_1 = 1$, completes the sequence's recursive definition.)

Example

Find the beginning conditions and a recurrence relation for 1,5,17,53, 161, 485……

Solution

We'll attempt to solve these recurring relationships. We want to discover a function of n (a closed formula) that satisfies the recurrence relation and the initial condition, which is quite similar to solving differential equations. Finding a solution, as with differential equations, might be difficult, but verifying that the solution is accurate is simple.

5.17 Combinatorics: Introduction, Counting Techniques

Sum Rule Principle: Assume that one event E can happen in m ways and another event F can happen in n ways, and that both events cannot happen at the same time. Then E or F can happen in m + n different ways.

In general, if there are n events and no two of them occur at the same moment, the event can have n_1+n_2…….n different forms.

Example - If 8 male processors and 5 female processors teach DMS, the student has a total of 8+5=13 professor options.

Product Rule Principle: Assume there is an event E that can happen in m different ways, and a second event F that can happen in n different ways. Then E and F can be combined in a variety of ways.

In general, if n events occur simultaneously, they can all occur in the order represented by n_1 x n_2 x n_3……..n ways.

Example - If there are four boys and ten girls in the class, and a boy and a girl must be picked for the class monitor, the pupils have a total of four x ten = forty options.

Mathematical Functions

Factorial Function:

Factorial n is the product of the first n natural numbers. It's represented by the letter n!, which stands for "n Factorial."

It's also possible to write Factorial n as

n! = n (n-1) (n-2) (n-3)……1.

= 1 and 0! = 1.

5.18 Pigeonhole Principle

If n+1 or more pigeons occupy n pigeonholes, then at least one pigeonhole is occupied by more than one pigeon. If n pigeonholes are occupied by kn+1 or more pigeons, where k is a positive integer, then at least one pigeonhole is inhabited by k+1 or more pigeons, according to the generalized pigeonhole principle.

Example - Find the least number of pupils in a class to ensure that three of them share the same month of birth.

Solution - Pigeonholes are defined as n = 12 months.

And k + 1 = 3

K = 2

Inclusion-Exclusion Principle:

Let A1,A2.....Ar be the subset of Universal set U. Then the number m of the element which does not appear in any subset A1,A2.....Ar of U.

Pigeonhole Principle

Example: Let U be the set of positive integers not exceeding 1000. Then |U|= 1000 Find |S| where S is the set of such integers which is not divisible by 3, 5 or 7?

Solution: Let A be the subset of integer which is divisible by 3

Let B be the subset of integer which is divisible by 5

Let C be the subset of integer which is divisible by 7

Then S = Ac ∩ Bc∩ Cc since each element of S is not divisible by 3, 5, or 7.

By Integer division,

|A|= 1000/3 = 333

|B|= 1000/5 = 200

|C| = 1000/7 = 142

|A∩B|=1000/15=66

|B∩C|=1000/21=47

|C∩A|=1000/35=28

|A∩B∩C|=1000/105=9

Thus, by Inclusion-Exclusion Principle

|S|=1000-(333+200+142) +(66+47+28)-9

|S|=1000-675+141-9=457

Key takeaway

1. If n+1 or more pigeons occupy n pigeonholes, then at least one pigeonhole is occupied by more than one pigeon.
2. If n pigeonholes are occupied by kn+1 or more pigeons, where k is a positive integer, then at least one pigeonhole is inhabited by k+1 or more pigeons, according to the generalized pigeonhole principle.