NAME: - KAMLESH KUMAR SAHU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, DIPLOMA

SEMSETER: 3RD **SEM**

SUBJENCT NAME: - **DATA STRUCTURE**

SUBJECT CODE: - **2022373(022)**

CHHATTISGARH INSTITUTE OF TECHNOLOGY, JASHPUR

SESSION 2025-26

UNIT 01
INTRODUCTION TO DATA STRUCTURE & ARRAY in 'C'

DATA (डाटा): -

डाटा "Facts, Concepts, या Instructions" का एक Formalized Manner में एक Representation होता है जो की Human या Electronic Machine के द्वारा Communication, Interpretation, या Processing के लिए Suitable होता है। Data को हम Characters के मदद से Represent कर सकते हैं जैसे की Alphabets (A-Z, a-z), Digits (0-9) या कोई Special Characters (+,-,/,*,<,>,=) इत्यादि। ये डाटा Character, Text, Numbers, Pictures, Sound, या फिर Video भी कुछ भी हो सकता है। Data को जब हम Process और Interpret करते हैं तब जाकर उनका सही मतलब सामने आता है और जो की हमारे लिए बहुत उपयोगी होते हैं। इन्ही Processed Data को Information भी कहा जाता है।

DATA TYPES (डाटा टाइप्स): -

एक प्रोग्रामिंग लैंग्वेज में जिस तरह का डाटा वेरिएबल में रखा जा सकता, उसे ही डाटा टाइप्स कहा जाता है। डाटा टाइप्स कई तरह के होते है जैसे Integer, Character, String, Real आदि जो की किसी भी प्रोग्रामिंग लैंग्वेज में एक वेरिएबल में रखे जाते है। इन सभी टाइप्स को डाटा टाइप्स के नाम से जाना जाता है।

प्रोग्रामिंग के दौरान जब हम कोई Variable Declare करते है, तब कम्पाइलर को यह बताना होगा की वेरिएबल में किस प्रकार का डाटा स्टोर करना चाहते है, इससे कम्पाइलर उतनी ही मेमोरी उस वेरिएबल को कंप्यूटर मेमोरी से प्रदान करता है।

DATA VARIABLE (डाटा वेरिएबल): -

Computer Memory में Data को Store करने के लिए Variables का प्रयोग किया जाता है। यदि आप किसी Data के साथ Operations Perform करना चाहते है तो इसके लिए सबसे पहले आपको उसे Computer की Memory में Store करना पड़ता है। एक Variable, Memory में किसी Location का नाम होता है। यह नाम (या Variable) उस Memory Location को Computer की संपूर्ण Memory में Uniquely Identify करने के लिए प्रयोग किया जाता है और इसी नाम के द्वारा उस Memory Location में Data Store करते है और उस Data को पुनः प्राप्त करते है।

Variables की Values Changeable होती है। आप एक Value को हटाकर दूसरी Value डाल सकते है। ऐसा आप Compile Time पर भी कर सकते है और Dynamically (Program Execution के दौरान) भी कर सकते है।

उदाहरण: - **int Age = 25;**

इस Statement के द्वारा एक Integer Variable Create किया गया है, जिसका नाम Age है और इस Variable को 25 Value Assign की गई है। जब Compiler सबसे पहले int को Execute करता है तो वह Computer की Memory में से 2 Bytes की Memory Allot करता है। इसके बाद जब Compiler Age को Execute करता है तो वह उस 2 Bytes की Memory को Age नाम दे देता है। इसके बाद जब Compiler = 25 को Execute करता है तो वह 25 को इस Memory Location पर Store कर देता है।

CONSTANT (कॉन्स्टेंट): -

Constants वो Variables होते है जिनकी Value Program Execution के दौरान किसी प्रकार भी Change नहीं होती है। जब भी कोई Constant Declare करते है तो Program के Execution के दौरान उसकी Value Fixed रहती है। यदि इसकी Value Change करने की कोशिश की जाती है तो Program में Error आ जाती है।

C Language में Constants दो Types के होते है।

a. Constant Literals.
b. Constant Variables.

a. Constant Literal (कॉन्स्टेंट लिट्रलस): - Constant Literals ऐसी Values होती है जिन्हें आप Program में Directly Use करते है। उदाहरण y = x+2; दिए गए Statement में 2 एक Constant Literal है। इसे Program के Execution के दौरान Change नहीं किया जा सकता है। यदि एक Literal Constant का प्रयोग Program में कई जगह पर किया है, और इस Constant को Change करने की आवश्यकता होती है तो उसे Manually ढूंढ कर Program में हर जगह Change करना होगा। इसलिए Literal Constants का प्रयोग कम से कम करना चाहिए।

b. Constant Variables (कॉन्स्टेंट वेरिएबल्स): - Constant Variables को Variables की तरह Declare करते है। Constant Variables को प्रयोग करने का फायदा ये है बाद में Constant को Change करना पड़े तो इसे Program में सिर्फ Constant Variable की Value Change करते है और वह Program में हर जगह Change हो जाती है।

DATA TYPES (डाटा टाइप्स): -

एक प्रोग्रामिंग लैंग्वेज में जिस तरह का डाटा वेरिएबल में रखा जा सकता, उसे ही डाटा टाइप्स कहा जाता है। डाटा टाइप्स कई तरह के होते है जैसे Integer, Character, String, Teal आदि जो की किसी भी प्रोग्रामिंग लैंग्वेज में एक वेरिएबल में रखे जाते है। इन सभी टाइप्स को डाटा टाइप्स के नाम से जाना जाता है।

साधारण अर्थो में प्रोग्रामिंग के दौरान जब हम कोई Variable Declare करते है, तब आपको कम्पाइलर को यह बताना होगा की आप वेरिएबल में किस प्रकार का डाटा स्टोर करना चाहते है, इससे कम्पाइलर उतनी ही मेमोरी उस वेरिएबल को कंप्यूटर मेमोरी से प्रदान करता है।

a. Character Type (char) (कैरेक्टर).
b. Interger Type (int) (इन्टिजर).
c. Floatting Point Type (float) (फ्लोटिंग).
d. Double Type (डबल).

a. Character Type (char) (कैरेक्टर): - Character Type को एक Character स्टोर करने के लिए यूज़ किया जाता है। इनको 2 कैटेगरीज में डिवाइड किया गया है।

Data type	Size (Bytes)	Range
char	1	-128 से 127
unsigned char	1	0 से 255

b. Integer Type (int) (इन्टिजर): - Integer Type किसी भी पूर्ण नंबर (बिना दशमलव वाली संख्या) को स्टोर करने के लिए यूज़ किये जाते है। Integer टाइप्स 5 प्रकार के होते है। हालांकि ये सभी पूर्ण नंबर्स को स्टोर करते है। लेकिन मेमोरी साइज और रेंज के बेस पर इन्हें बाँटा गया है।

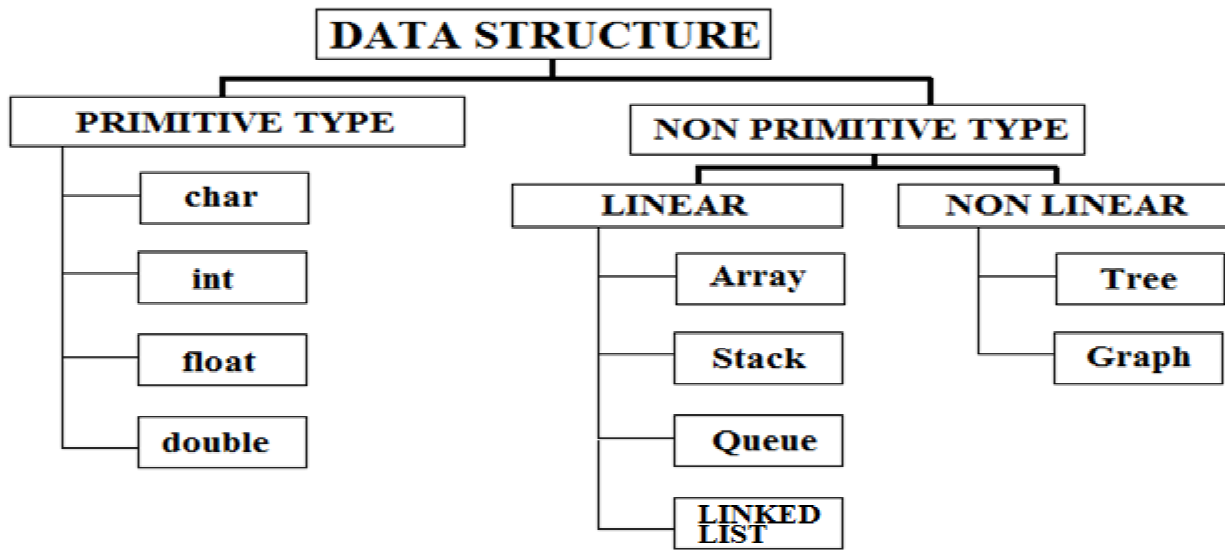
Data type	Size (Bytes)	Range
int	2	-32768 से 32767
short int	1	-128 से 127
long int	4	-2,147,483,648 से 2,147,483,647
signed int	2	-32768 से 32767 (नेगेटिव वैल्यूज के लिए)
unsigned int	2	0 से 65535 (पॉजिटिव वैल्यूज के लिए)

c. Floating Point Type and Double (float & double) (फ्लोटिंग और डबल): - Floating Point डेटा टाइप दशमलव वाली संख्याओं को स्टोर करने के लिए डिफाइन किया गया है। Floating Point डेटा टाइप 2 तरह के होते है। इनको साइज और रेंज के बेस पर बाँटा गया है। float टाइप में आप दशमलव के बाद 7 डिजिट्स तक स्टोर कर सकते है। जबकि डबल टाइप में दशमलव के बाद 17 डिजिट्स तक स्टोर किये जा सकते है।

Data type	Size (Bytes)	Range
float	4	3.4E-38 से 3.4E+38
double	8	1.7E-308 से 1.7E+308

DATA STRUCTURE (डाटा स्ट्रक्चर): -

एक ही तरह से Data को Store और Organize करके आप अलग अलग Problems को Solve नहीं कर सकते है। अलग अलग तरह की Problems को Solve करने के लिए आपको अलग अलग तरह के Data Structures Create करने की आवश्यकता होती है।



हर तरह के Data Structure का Data Organization Mechanism और Operational Behaviour अलग अलग होता है जो उसे दूसरे Data Structures से अलग बनाता है। ऐसी ही Characteristics के आधार पर Data Structures को अलग अलग श्रेणियों में विभाजित किया गया है।

1. **Primitive Data Structures (प्रिमिटिव डाटा स्ट्रक्चर): -**

एक Data Type किसी Specific Type के Data को Store करने के लिए Structure Provide करता है। इसलिए Programming Languages द्वारा Provided Primitive Data Types को भी Data Structures ही माना जाता है। ये Data Structures Primitive Data Structures कहलाते है।

i. char : - Character (शब्द) Store करने के लिए।
ii. int : – Integer Data Store करने के लिए।
iii. float : – Floating Point Data Store करने के लिए।
iv. double : – यह Data float की तरह ही होता है। इसे दशमलव के बाद 7 से अधिक Values Store करने के लिए प्रयोग किया जाता है।
v. boolean : – True और False Values Store करने के लिए।

2. **Non-Primitive Data Structures (नॉन प्रिमिटिव डाटा स्ट्रक्चर): -**

Primitive Data Types के Combination से Non-Primitive या User Defined Data Types Create किये जाते है इसलिए User Defined Data Types को Non-Primitive Data Structures भी कहा जाता है। Non-Primitive Data Structures या अलग अलग Type के Primitive Data Structures के Combination से मिलकर बने होते है। उदाहरण के लिए एक Integer Numbers का Array Non-Primitive Data Structure होता है। Non-Primitive Data Structures को (Linear और Non-Linear) दो Categories में Divide किया गया है।

i. **Linear Data Structures (लीनियर डाटा स्ट्रक्चर): -** Linear Data Structures ऐसे Data Structures होते है जो Elements को Linear Sequence में Store करते है। उदाहरण के लिए एक Array के Elements Continuous Locations पर एक बाद एक Store होते है। लीनियर डाटा स्ट्रक्चर के प्रकार: -

a. Arrays (ऐरे): - एक Array सबसे Simple Non Primitive Linear Data Structure होता है। Array में Elements Contiguous Memory Locations में Store किये जाते है। एक Array Same (Data) Type के Variables का Collection होता है जिसे एक Common नाम के द्वारा Present किया जाता है।

b. Stacks (स्टैक): - Stack एक Linear Data Structure है जिसमें Elements एक ही तरफ से (Top) Add और Remove किये जाते है। Stack में Elements उसी प्रकार Organize किये जाते है जिस प्रकार किसी Restaurant में Plates को (एक के ऊपर एक) Organize किया जाता है। Stack में सबसे Last में Insert किया गया Element सबसे पहले Access होता है और सबसे पहले Insert किया गया Element सबसे Last में Access होता है।

c. Queues (क्यु): - एक Queue ऐसा Data Structure होता है जिसमे Elements को एक तरफ से (पीछे की तरफ से) Insert किया जाता है और दूसरी तरफ से (आगे की तरफ से) Remove किया जाता है। जिस प्रकार आप किसी Line में खड़े होते समय

सबसे आखिर में खड़े होते है और Service पाने के बाद सबसे आगे से जाते है। उसी प्रकार एक Queue Data Structure भी "First In First Out Order" में काम करता है।

d. Singly Linked Lists (सिंगली लिंक्ड लिस्ट): - एक Linked List Data Structure Elements का Linear Collection होता है। Linked List Data Structure में एक Element दूसरे Element को Point करता है। हर Element के साथ एक Next Pointer या Link Node जुडी हुई होती है जो List के अगले Element को Memory में Point करती है। Linked List के द्वारा Arrays की Drawbacks को Overcome कर पाते है और एक ऐसा Data Structure उपयोग कर पाते है जो Memory का सही Utilization करता है और जिसमे Operations आसानी से Perform किये जा सकते है।

ii. **Non-Linear Data Structures (नॉन लीनियर डाटा स्ट्रक्चर): -** Non-Linear Data Structures ऐसे Data Structures होते है जिनमें Elements Linear Sequence में नहीं Store होते है। उदाहरण के लिए एक Tree Data Structure में Elements को Linear Sequence में नहीं Store किया जाता है इसलिए Tree एक Non-Linear Data Structure होता है। नॉन लीनियर डाटा स्ट्रक्चर के प्रकार: -

a. Trees (ट्री): - Tree Data Structures का प्रयोग ऐसे Data को Represent करने के लिए किया जाता है जिसमें किसी Entity और उसके Attributes में Hierarchical Relationship होती है। Tree Data Structure में Data और उसकी Entities Parent Nodes और Child Nodes के रूप में Represent की जाती है। एक Linked List में एक Node किसी दूसरी एक ही Node को Point करती है लेकिन एक Tree Data Structure में एक Node कई Nodes को Point कर सकती है। Tree Data Structure में Child Nodes की भी Child Nodes हो सकती है।

b. Graphs (ग्राफ): - Graphs Non-Linear Data Structures होते है जिनका प्रयोग कई प्रकार से किया जाता है। Graphs का प्रयोग Electrical Circuits के Analysis के लिए, Shortest Routes ढूँढने के लिए, Project Planning के लिए, Highway, Landlines और Railway Lines आदि को Represent करने के लिए भी Graphs का प्रयोग किया जाता है।

Difference Between Linear & Non Linear Data Structure (लीनियर एंड नॉन लीनियर डाटा स्ट्रक्चर में अंतर)

SN.	LINEAR DATA STRUCTURE	NON LINEAR DATA STRUCTURE
01	ऐसा डाटा स्ट्रक्चर जहाँ डाटा सिक्वेन्सीली एक के बाद एक अरेंज रहते है।	ऐसा डाटा स्ट्रक्चर जहाँ डाटा सिक्वेन्सीली अरेंज नहीं रहते है तथा वे एक दूसरे से अलग अलग लोकेशन से जुड़े होते है।
02	इसमें डाटा में ट्रैवर्स एक ही रन में आसानी से किया जा सकता है।	इसमें डाटा ट्रैवर्स एक ही रन में नहीं किया जा सकता।
03	इसे आसानी से डेवेलप किया जा सकता है।	इसे डेवेलप करने में बहुत ही कठिनाई होती है।
04	इसमें मेमोरी यूटिलाइजेशन बहुत अच्छे से नहीं होता।	इसमें मेमोरी यूटिलाइजेशन बहुत अच्छे से होता है।
05	इसे एक लेवल में इम्प्लीमेंट किया जा सकता है।	इसे मल्टी लेवल में इम्प्लीमेंट किया जाता है।
06	इसका उदहारण: - Array, Stack, Queue etc.	इसका उदहारण: - Tree and Graph

ARRAY (ऐरे): -

Array एक Non - Primitive तथा Linear डेटा स्ट्रक्चर है जो की एक समान (Similar) डेटा Items का समूह होता है, अर्थात यह सिर्फ एक ही प्रकार के डेटा (या तो यह सिर्फ सभी Integer डेटा को स्टोर करेगा या फिर सभी Floating point को या ऐसी प्रकार किसी अन्य डाटा टाइप्स के डाटा को) ही स्टोर करता है। Array डेटा स्ट्रक्चर का प्रयोग डेटा ऑब्जेक्ट्स के समूह को संग्रहित करने के लिये किया जाता है। Arrays एक Static डेटा स्ट्रक्चर है अर्थात् हम केवल Compile Time में ही मेमोरी को Allocate कर सकते है और इसे Run-Time में बदल नही सकते है।

मान लीजिये किसी Company में 200 Employees है और इनके नाम को Store किया जाना है। यदि 200 Employee के लिए 200 Variables Create करेंगे और उनमे Store करेंगे तो ये एक बहुत ही Complex Approach होगी। Arrays ऐसी Facility Provide करती है कि आप सिर्फ एक Variable Create करे और उस Cariable में 200 Employees के नाम (या जो भी Information आप Store करना चाहते है) Store सकते है।

Characterstics of An Array (ऐरे की विशेषताएं): -

a. ऐरे एक ही प्रकार के डाटा वैल्यू को स्टोर करता है।

b. ऐरे एक लगातार मेमोरी लोकेशन में डाटा वैल्यू को स्टोर करता है।

c. ऐरे का नाम को प्रदर्शित करता है और वही ऐरे का प्रथम डाटा वैल्यू को स्टोर करता है।
d. ऐरे का इंडेक्स '0' से प्राम्भ होता है और 'N – 1' में अंत होता है।
e. ऐरे की साइज को प्राम्भ में ही Declear कर दिया जाता है। इसकी साइज को Run Time में नहीं बदला जा सकता ।

Advantage (लाभ): -

- Array को आसानी से Implement किया जा सकता है।
- एक ही प्रकार के विभिन्न डेटा Items को केवल एक नाम के द्वारा प्रदर्शित किया जा सकता है।
- Array एक ही समय में अनेक डेटा Items को स्टोर कर सकता है।

Disadvantages (हानि): -

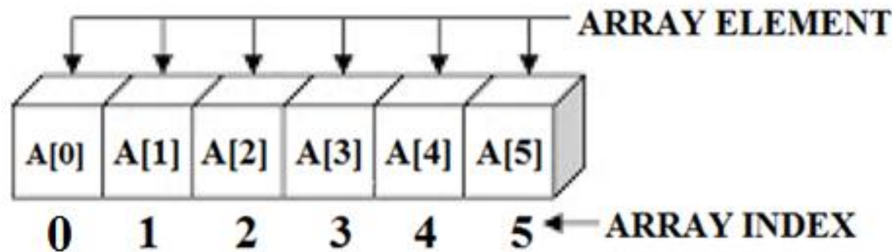
- Array के द्वारा मैमोरी का Waste होता है।
- Array एक Static डेटा स्ट्रक्चर है जिसके कारण इसका Size पहले से ही Define होता है।
- हमें Array में एक Element को Delete तथा Insert करने के लिए पूरे Array को Traverse करना पड़ता है।

Array के सारे Variables मेमोरी में Contiguous Locations में अर्थात् एक के बाद एक स्टोर होते है। Subscript की संख्या के आधार पर Array निम्नलिखित तीन प्रकार के होते है: -

a. One Dimensional Array.
b. Two Dimensional Array.
c. Multi Dimensional Array.

a. One Dimensional Array {1-D Array} (वन डायमेंशनल ऐरे): -

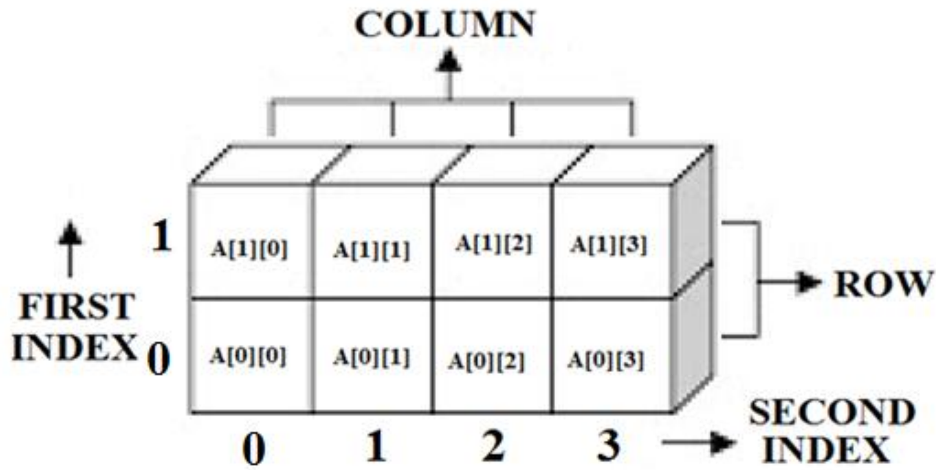
ऐसा Array जिसमें केवल एक Index या Subscript का प्रयोग किया जाता है One Dimensional Array {1-D Array} कहलाता है। इसमें Row या Column में डेटा को स्टोर किया जाता है। यह सबसे सरल Array होता है जिसका प्रयोग मुख्यतः String Manipulation एवं Linear रूप में डेटा को स्टोर करने के लिए किया जाता है। इस Array मे कोई Value Store करने या Access करने के लिए इसकी Index का Use किया जाता है। 1-D Array मे Design और Run Time दोनों मे Values को Insert किया जा सकता है। Design Time मे Value Store करने के लिए Array Index या Curly Brackets {} का Use किया जाता है।



Example: **int A[6];** इस Declaration मे A[6] मे 0 से 5 Index तक या 6 Values Store की जा सकती हैं।

b. Two Dimenstional Array {2-D Array}(टू डायमेंशनल ऐरे): -

ऐसा Array जिसमें दो Subscript का प्रयोग किया जाता है Two Dimensional Array {2-D Array} कहलाता है। इसमें Row एवं Column दोनों में डेटा को स्टोर किया जाता है। इसका प्रयोग मुख्यतः Spreadsheet Software बनाने एवं Matrix Manipulation में किया जाता है। इसे Array of Arrays भी कहा जाता है। इसका Struture, Matrix की तरह होता है। इसे Matrix Array भी कहते है।



इसका Syntax है: - *{data type} {array name} [row size] [column size];*

- data type: ये Define करता है की Array मे किस तरह का Data Store होगा।
- array name : ये Array का नाम होता है।
- row size: ये Number of Row को Define करता है।
- column size: ये Number of Column को Define करता है।

उदहारण के लिए: - **int A[2][4];** इस Statement से एक 2*4 2-D Array बनता है जिसका नाम 'A' और Data Type 'int' है। इसमें 2 Row और 4 Column है।

2-D Array, 1-D Array की Array होती है। सभी 1-D Array का Address / Index 0 से Start होता है और 2-D Array मे, Index Two डिजिट का होता है एक Row के Index की Value होती है। दूसरा Column का Index होता है। 2-D Array को दो प्रकार से प्रदर्शित किया जाता है: -

i. Row Major Ordr (रौ मेजर आर्डर).
ii. Column Major Order (कॉलम मेजर आर्डर)

i. **Row Major Ordr (रौ मेजर आर्डर): -** इस मेथड में 2-D Array का पहला इंडेक्स Row को और दूसरा इंडेक्स Column को प्रदर्शित करता है। इसमें डाटा का स्टोर भी Row by Row किया जाता है।

उदाहरण: - मान लीजिये Data {1, 2, 3, 4, 5, 6, 7, 8, 9} एक 3 x 3 Matrix का है तथा इसे 2-D Array के Row Major Oder में Store किया जाना है। तब सूचनाएं पहले Row में {1, 2, 3}, दूसरे Row में {4, 5, 6} तथा तीसरे Row में {7, 8, 9} होगा तथा आउटपुट निमानुसार होगा।

3 x 3 MATRIX

1	2	3
4	5	6
7	8	9

⇨

OUTPUT

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Values are Store in This Order: -

A[0][0] = 1; A[0][1] = 2; A[0][2] = 3;
A[1][0] = 4; A[1][1] = 5; A[1][2] = 6;
A[2][0] = 7; A[2][1] = 8; A[2][2] = 9;

यदि 2-D Array की प्राम्भिक Address 2000 है तथा है Data Type 'int' है जिसका साइज 1 Byte का होता है तब सम्बंधित डाटा का Address निम्नानुसार होगा: -

A[0][0] = 2000; A[0][1] = 2001; A[0][2] = 2002;
A[1][0] = 2003; A[1][1] = 2004; A[1][2] = 2005;
A[2][0] = 2006; A[2][1] = 2007; A[2][2] = 2008;

Row Major Oder 2-D Array में किसी Data का Location / Address ज्ञात करने का Formula निम्नानुसार है: -

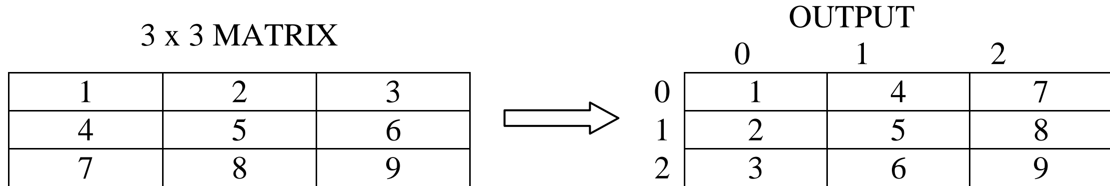
A[J][K] में (Loc A[m][n]) = Base + w [K (m-1) + (n-1)];

जहाँ पर: - A[m][n] = जिस Data Value का Address ज्ञात करना है।
Base = Row Major Order 2-D Array का प्राम्भिक Address।

w = Data Type का Size।
K = 2-D Array में Column की कुल संख्या।
m = Data Value के Row की संख्या।
n = Data Value के Coumn की संख्या।

ii. **Column Major Ordr (कॉलम मेजर आर्डर): -** इस मेथड में 2-D Array का पहला इंडेक्स Column को और दूसरा इंडेक्स Row को प्रदर्शित करता है। इसमें डाटा को स्टोर Column से Row में किया जाता है।

उदाहरण: - मान लीजिये Data {1, 2, 3, 4, 5, 6, 7, 8, 9} एक 3 x 3 Matrix का है तथा इसे 2-D Array के Column Major Oder में Store किया जाना है। तब सूचनाएं पहले Row में {1, 4, 7}, दूसरे Row में {2, 5, 8} तथा तीसरे Row में {3, 6, 9} होगा तथा आउटपुट निमानुसार होगा।



3 x 3 MATRIX

1	2	3
4	5	6
7	8	9

OUTPUT

	0	1	2
0	1	4	7
1	2	5	8
2	3	6	9

Values are Store in This Order: -

A[0][0] = 1; A[0][1] = 4; A[0][2] = 7;
A[1][0] = 2; A[1][1] = 5; A[1][2] = 8;
A[2][0] = 3; A[2][1] = 6; A[2][2] = 9;

यदि 2-D Array की प्राम्भिक Address 2000 है तथा है Data Type 'int' है जिसका साइज 1 Byte का होता है तब सम्बंधित डाटा का Address निम्नानुसार होगा: -

A[0][0] = 2000; A[0][1] = 2001; A[0][2] = 2002;
A[1][0] = 2003; A[1][1] = 2004; A[1][2] = 2005;
A[2][0] = 2006; A[2][1] = 2007; A[2][2] = 2008;

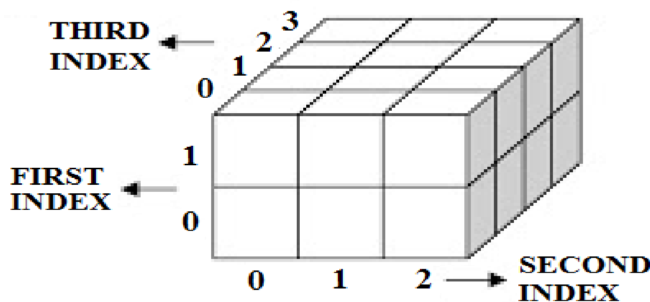
Column Major Oder 2-D Array में किसी Data का Location/Address ज्ञात करने का Formula निम्नानुसार है:

A[J][K] में (Loc A[m][n]) = Base + w [J (n-1) + (m-1)];

जहाँ पर: - A[m][n] = जिस Data Value का Address ज्ञात करना है।
Base = Row Major Order 2-D Array का प्राम्भिक Address।
w = Data Type का Size।
J = 2-D Array में Row की कुल संख्या।
m = Data Value के Row की संख्या।
n = Data Value के Coumn की संख्या।

c. Multi Dimensional Array (मल्टी डायमेंशनल ऐरे): -

ऐसा Array जिसमें दो से अधिक Subscript का प्रयोग किया जाता है Multi Dimensional Array कहलाता है। इसमें Subscript की संख्या 3, 4, 5 या 'n'हो सकता है। यह सबसे कठिन Array होता है। इसका प्रयोग मुख्यतः 3D Structures को दर्शाने के लिए किया जाता है। इसे Array of Arrays of Arrays भी कहा जाता है।



Insertion in Array (ऐरे में जोड़ना): -

Program to Insert and Delete in an Array: किसी Array में यदि जगह उपलब्ध हो तो नई इकाई को Array के अन्त में जोडना काफी आसान होता है। लेकिन जब हमें Array के किसी विशेष Index Number पर Value को Insert करना होता है, तो इसके लिये Array के जिस Element के बाद नई इकाई जोडनी है, उससे बाद के सारे Elements को एक-एक स्थान आगे प्रतिस्थापित किया जाता है। फिर नए Value को Array में जोडा जाता है।

यदि हम Array में Value Insert करने से पहले जिस स्थान पर Value Insert करना है, उससे आगे के सभी मानों को प्रतिस्थापित नहीं करते हैं तो हमारा नया Value पुराने Value पर Over Write हो जाता है।

माना एक Array A[N] है जिसमें N Items हैं। इस Array के Index Number K पर एक Element ITEM को Insert करना है जबकि हम ये मान कर चलते हैं कि इस Array में अभी इतना स्थान है कि हम इसमें नया Item Insert कर सकें। चूंकि हमें Index Number K पर नया ITEM Insert करना है इसलिए हमें Index Number K को खाली करना होगा ताकि नया Data इसमें Store हो सके। क्योकि नया Data Store करने के लिए हम Index Number K पर जगह बना रहे हैं इसलिए हमें Index Number K से Array के अन्तिम Data Items तक के सभी Data Items को एक स्थान Right में Move करना होगा। इस प्रक्रिया को हम निम्न चित्र द्वारा समझ सकते हैं-

A[8]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	10	25	32	45	95	75		
	0	1	2	3	4	5	6	7

माना K का मान 4 है तो हमें Index Number 4 - 1 = 3 को खाली करना होगा। ऐसा करने पर Index Number 3 के बाद के सभी Data Items को एक स्थान Right में Move करना होगा। ऐसा करने पर ये Array निम्नानुसार दिखाई देगा-

A[8]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	10	25	32	100	45	95	75	
	0	1	2	3	4	5	6	7

Deletion in Array (ऐरे में हटाना): -

किसी Array के अन्तिम Element को Delete करना काफी आसान होता है, लेकिन जब किसी Array के किसी अन्य Element को Delete किया जाता है, तो Array के उस Element से आगे के सभी Elements को एक स्थान पीछे प्रतिस्थापित करना पडता है। यदि ऐसा ना किया जाए तो जिस स्थान के मान को Delete किया गया है उस स्थान पर Garbage मान Store हो जाता है।

माना कि एक Array A[N] है जिसमें N Items हैं। इस Array के Index Number K पर स्थित Element को Delete करना है। चूंकि हमें Index Number K पर स्थित Item को Delete कर रहे हैं इसलिए हमें Index Number K के बाद के सभी Data Items को एक स्थान पीछे की तरफ Move करना होगा। इस प्रक्रिया को हम निम्न चित्र द्वारा समझ सकते हैं-

A[8]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	10	25	32	100	45	95	75	
	0	1	2	3	4	5	6	7

यदि हम K का मान 5 मानें तो Index Number 5 - 1 = 4 के Data Item को Delete करना है। जब हम Index Number 4 के Data Item को Delete करना चाहते हैं तो हमें बस इतना ही करना है कि Index Number 4 के Data Item पर Index Number 5 के Data Item को Place कर दें। यानी Index Number 4 के बाद के सभी Data Items को एक स्थान आगे सरका दें। ऐसा करने पर ये Array निम्नानुर दिखाई देगा-

A[8]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	10	25	32	100	95	75		
	0	1	2	3	4	5	6	7

Array Searching (ऐरे सर्चिंग): - सर्चिंग का अर्थ है "ढूंढना" या "खोजना", ऐरे डेटा स्ट्रक्चर में 'Searching' वह प्रक्रिया है जिसमें किसी Element को लिस्ट में खोजा जाता है जो कि एक या एक से अधिक Condition को संतुष्ट करता हो ।

Types of Searching: - ऐरे डेटा स्ट्रक्चर में Searching के लिए हम दो तकनीकों का प्रयोग करते हैं: -

a. Linear Search (लीनियर सर्च)
b. Binary search (बाइनरी सर्च)

a. Linear Search (लीनियर सर्च): -

इसको Sequential Search भी कहते है। इस Searching तकनीक में दिए गये डेटा Element को तब तक एक एक करके लिस्ट के प्रत्येक Element के साथ Compare किया जाता है जब तक कि Element मिल नहीं जाता। इसमें सबसे पहले दिए गये Element को लिस्ट के प्रथम Element के साथ Compare किया जाता है यदि दोनों Element एक समान है तो वह Index Value रिटर्न करता है नहीं तो -1 रिटर्न करता है। फिर इसके बाद दिए गये Element को लिस्ट के दुसरे Element के साथ Compare किया जाता है, यदि दोनों Element समान है तो वह Index Value रिटर्न करता है नहीं तो -1 रिटर्न करता है। इसी प्रकार पूरी लिस्ट को Compare किया जाता है जब तक कि Element मिल नहीं जाता है, अगर पूरी लिस्ट Compare करने के बाद भी Element नहीं मिलता है तो सर्च Unsuccessful हो जाएगा। यह सबसे सरल Searching तकनीक है परन्तु इसमें समय बहुत लगता है।

A[8]

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
10	25	32	100	95	75	29	63
0	1	2	3	4	5	6	7

Search Element = 75

Step 1: - दिए गये Element (75) को लिस्ट के प्रथम Element (10) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान नहीं है तो Retrun -1 कर हम दुसरे Element में जायेंगे।

Step 2: - दिए गये Element (75) को लिस्ट के दूसरे Element (25) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान नहीं है तो Retrun -1 कर हम तीसरे Element में जायेंगे।

. . . . . .

. . . . . .

Step 6: - दिए गये Element (75) को लिस्ट के छटवें Element (25) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान है तो Retrun Index value 5 कर Successful Search प्रिंट करेंगे।

b. Binary search (बाइनरी सर्च): -

जब कोई बड़ा डाटा स्ट्रक्चर होता है तो Linear Search में बहुत अधिक समय लग जाता है, इसलिए Linear Search की कमी को दूर करने के लिए Binary Search को Develop किया गया। Binary Search बहुत ही तेज Searching अल्गोरिथम है जिसकी Time Complexity O (log n) है, यह Divide & Conquer सिद्धांत पर आधारित है।

Binary Search केवल उसी लिस्ट में की जा सकती है जो कि Sorted (क्रमानुसार) हों, इसका प्रयोग ऐसी लिस्ट में नहीं कर सकते जो कि Sorted Order में नहीं है। इस सर्चिंग तकनीक में दिए गये Element की लिस्ट के Middle Element के साथ तुलना की जाती है। यदि दोनों एकसमान है तो वह Index Value रिटर्न करता है। यदि एक समान नहीं है तो हम Check करते है कि दिया गया Element, Middle Element से बड़ा है या छोटा।

यदि वह छोटा है तो हम लिस्ट के छोटे भाग में यही प्रक्रिया दोहराएंगे।

और यदि वह बड़ा है तो हम लिस्ट के बड़े भाग में यही प्रक्रिया दोहराएंगे और यह तब तक करेंगे जब तक कि Element मिल नहीं जाता।

A[8]

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
10	25	32	45	57	64	71	85	91
0	1	2	3	4	5	6	7	8

Search Element = 32

Step 1: - दिए गये Element (32) को लिस्ट के Middle Element A [4] अर्थात (57) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान नहीं है तो Retrun -1 कर हम यह चेक करेंगे की Search Element, Middle Element से छोटा है या बड़ा, यहाँ पर Search Element, Middle Element से छोटा है तब हम Middile Element के पहले वाले हिस्से में Search Element को Searchकरेंगे।

Step 2: - पुनः Element (32) को लिस्ट के Middle Element A [1] अर्थात (25) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान नहीं है तो Retrun -1 कर हम यह चेक करेंगे की Search Element, Middle Element से छोटा है या बड़ा, यहाँ पर Search Element, Middle Element से बड़ा है तब हम Middile Element के दूसरे वाले हिस्से में Search Element को Searchकरेंगे।

Step 3: - पुनः Element (32) को लिस्ट के Middle Element A [2] अर्थात (32) के साथ Compare (तुलना) किया जाता है। दोनों एकसमान है तो Retrun Index value 2 कर Successful Search प्रिंट करेंगे।

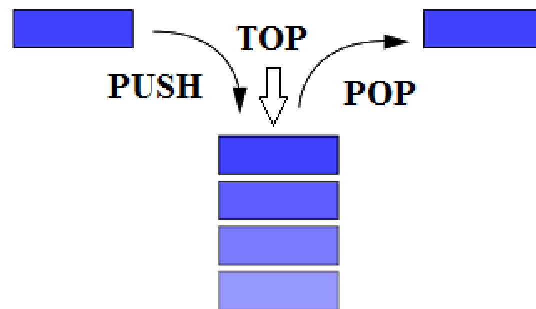
UNIT 02
STACKS AND QUEUE

STACK (स्टैक): -

Stack एक विशेष प्रकार का Linear डेटा स्ट्रक्चर होता है जो कि LIFO (Last In First Out) के सिद्धान्त पर कार्य करता है अर्थात वह Item जो की सबसे अंत में Add किया जाता है उसे सबसे पहले Remove कर दिया जाता है तथा जो Item सबसे पहले Add किया जाता है उसे सबसे अंत में Remove किया जाता है। इसको FILO (First In Last Out) भी कहा जा सकता है।

स्टैक Data ऑपरेशन परफॉर्म (डाटा एलिमेंट Add एंड Remove) करने के लिए एक Particular Sequential आर्डर फॉलो करता है। जहाँ पर नए आइटम का Addition एवं Removal हमेशा Same End से होता है। इस End Point को Commonly "Top" के नाम से एंड दूसरे Opposite End को "Base" नाम से जाना जाता है। स्टैक का Base आइटम सबसे ज्यादा टाइम तक रहने वाला एलिमेंट होता है क्योकि यह सबसे पहले Add कर दिया जाता है एवं इसको सबसे बाद मे Remove किया जाता है। स्टैक मे सबसे कम समय तक रहने वाला एलिमेंट Top एलिमेंट होता है जो की सबसे Recently ऐड किया हो।

Example: – जब भी आप Stack of Trays or Plates देखते है एवं आपको एक प्लेट लेनी होती है तो आप सबसे ऊपर वाली (टॉप) प्लेट सबसे पहले उठाते है एवं उसके बाद नेक्स्ट प्लेट्स। सबसे नीचे रखी गयी प्लेट सबसे बाद मे उठायी जाती है जिसे बेस प्लेट कहा जायेगा।



Stacks Features: –

- यह एक डायनामिक डाटा स्ट्रक्चर है।
- इसका कोई एक Fixed साइज नहीं होता।
- यह एक Fixed Amount of Memory खर्च नहीं करता।
- स्टैक का साइज हर Push() एंड Pop() ऑपरेशन के बाद चेंज होता है।

Basic Operation on Stack (स्टैक के बेसिक ऑपरेशन): -

1. **PUSH: –** जब Stack में कोई Item Insert किया जाता है तो वह Operation Push Operation कहलाता है।

Step 1 – सबसे पहले Stack के स्टेटस को चेक किया जाता है कि ये भरा हुआ तो नहीं है।

Step 2 – अगर स्टैक फुल होता है तो यह एरर मैसेज दे कर Exit कर जाता है।

Step 3 – अगर स्टैक Full नहीं है तो टॉप में डाटा एलिमेंट को ऐड करके Top वैल्यू को एक Increment कर देता है।

Step 4 – Push ऑपरेशन कम्पलीट होता है।

2. **POP : –** जब Stack से कोई Item Delete किया जाता है तो वह Operation Pop Operation कहलाता है।

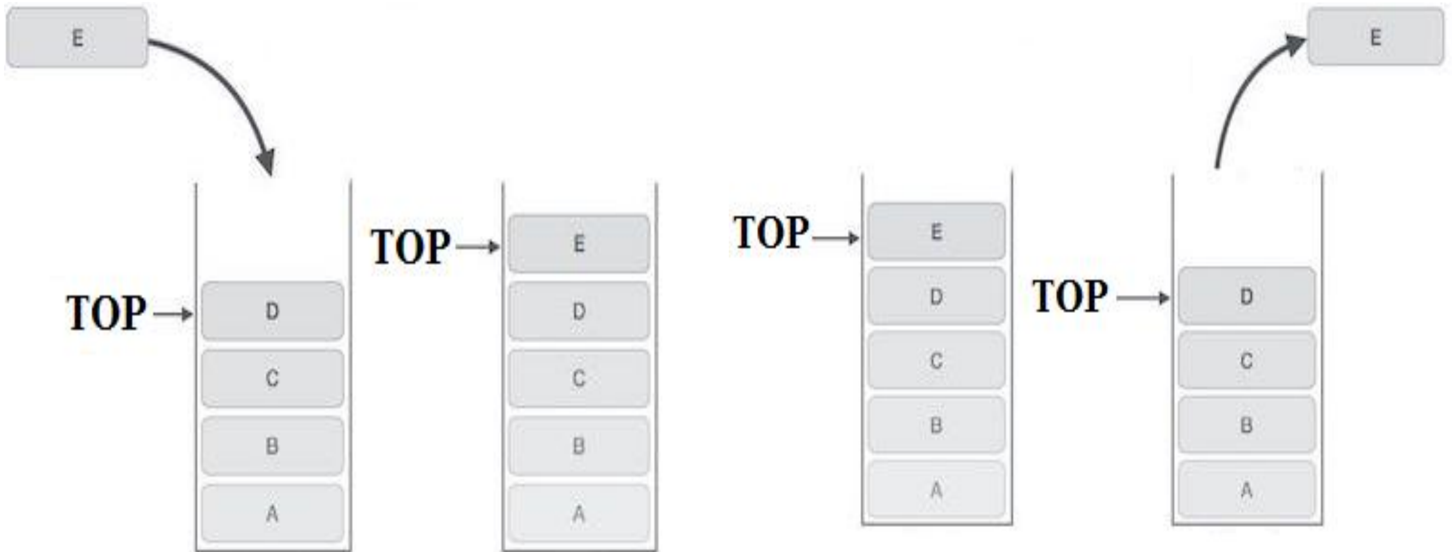
Step 1 – सबसे पहले Stack के स्टेटस को चेक करता है कि स्टैक खाली तो नहीं है।

Step 2 – अगर स्टैक खाली है तो एरर मैसेज दे कर Exit करता है।

Step 3 – अगर स्टैक खाली नहीं है तो Top डाटा एलिमेंट को Access करता है।

Step 4 – Top वैल्यू को एक कम कर देता है और टॉप एलिमेंट को Remove कर देता है।

Step 5 – Pop ऑपरेशन कम्पलीट होता है।



TOP VALUE INCREASE <u>“4 TO 5”</u>

<u>PUSH OPERATION ITEM “E”</u>

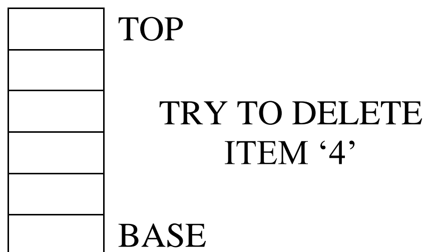
TOP VALUE DECREASE <u>“5 TO 4”</u>

<u>POP OPERATION ITEM “E”</u>

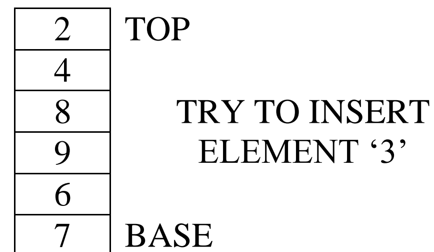
3. **PEEP: –** जब Stack में किसी Particular Location से Data प्राप्त किया जाता है तो वह Operation Peep Operation कहलाता है।
4. **Update: –** जब Stack के किसी Element की Value Change की जाती है तो वह Operation Update Operation कहलाता है।

<u>Underflow: -</u> यदि Stack Empty हो (Top = = -1) और उसमें से किसी Element को Delete करने का प्रयास किया जाए तो ऐसी Situation Underflow कहलाती है।

<u>Overflow: -</u> यदि Stack Full हो (Top = N – 1) और उसमें नया Element Add करने का प्रयास किया जाये तो ऐसी Situation Overflow कहलाती है।



EMPTY STACK TOP = = -1



FULL STACK TOP = “N-1”

Applications of Stack in Data Structure (स्टैक डाटा स्ट्रक्चर के अनुप्रयोग): -

1. Expression Evaluation: – स्टैक का प्रयोग Prefix, Postfix, तथा Infix Expressions को Evaluate करने के लिए किया जाता है।
2. Expression Conversion: - एक Expression को Prefix, Postfix या Infix Notation में प्रस्तुत किया जाता है। Stack का प्रयोग एक Expression के Form को दूसरे Form में Convert करने के लिए किया जाता है।
3. Backtracking: – Backtracking का अर्थ है वापस पीछे जाना। माना कि हमने Maze Problem (भूल भुलैया) को Solve करने के लिए एक Path (रास्ते) को ढूंडना है। हमने एक Path को Choose किया और हम उस Path में आगे बढ़ गये। बाद में हमको पता चला कि यह Path तो गलत है। अब हमें दूसरे नए Path में जाने के लिए जहाँ से हमने शुरू किया था वापस वहां जाना पड़ेगा और यह Stack की मदद से कर सकते है।
4. Parenthesis Checking: – Stack का प्रयोग यह Check करने के लिए किया जाता है कि Parenthesis सही ढंग से Open तथा Close हुए है या नहीं।
5. String Reversal: – Stack का प्रयोग String को Reverse करने के लिए किया जाता है। हम String के Characters को एक के बाद एक Stack में Push करते है तथा उसके बाद Characters को Stack से Pop करते है।
6. Memory Management में इसका Use होता है। CPU Scheduling और Disk Scheduling में।
7. Graph Traversal Algorithms में।

EXPRESSION NOTATION & EVALUATION USING STACK (स्टैक के अनुप्रयोग से समीकरण): -

ऐसे ट्रीज, जिनमें किसी समीकरण को प्रस्तुत किया जाता है, Expression Tree कहलाते है। इन ट्रीज में Operands और Operators दो प्रकार की सूचनायें होती हैं। Operands (ओपरेंड्स) वे Elements जिन पर कार्य किया जाता है। Operators (ऑपरेटर्स) वे Elements जिनके द्वारा Operands पर कार्य किया जाता है। अच्छी तरह समझने के लिए निम्न उदाहरण देखें: - (p - a) + (b * c ^ f)

इसमें -, +, *, तथा ^ Operators हैं और p, a, b, c तथा n Operands है। इस प्रकार के ट्रीज में प्रयुक्त किये जाने वाले Operators की Priorty (स्थान) निश्चित कर दिया गया है: - "BoE-DM-AS"

1st Priority (सबसे ऊँचा स्थान) : ^ (घात)

2nd Priority : / (भाग), * (गुणा),

3rd Priority : + (जोड़), – (घटाना)

Operator and Operand Stack (ऑपरेटर एंड ऑपरेंड स्टैक): -

जब किसी Expression को एक Form से दूसरे Form में Convert किया जाता है। तब दो प्रकार के Element के लिए 02 अलग अलग स्टैक प्रयोग किया जाता है।

01. Operand Stack: - इसमें ऑपेरेंड को स्टोर किया जाता है। जैसे a, b, n, z, etc.
02. Operator Stack: - इसमें ऑपरेटर को स्टोर किया जाता है। जैसे +, -, *, /, ^ etc.

Stack का Use कई Expression को Solve करने के लिए किया जाता है। ये सब प्रोसेस कंप्यूटर Memory मे होती है। इसलिए इन प्रोसेस मे, लगने वाला Time बहुत उपयोगी होता होता है। Expression Evaluation पढने का मतलब है किसी लम्बे Expression को छोटे Expression मे ट्रांसफॉर्म करना है। Expression Tree को निम्न तीन प्रकार से प्रदर्शित करते है: -

a. Infix (इन्फ़िक्स): - इस Expression Notation में Operator, Operands के बीच में उपस्थित रहते है।

b. Postfix (पोस्टफिक्स): - इस Expression Notation में Operator, Operands के बाद में उपस्थित रहते है।

c. Prefix (प्रीफिक्स): - इस Expression Notation में Operator, Operands के पहले उपस्थित रहते है।

Note: - उपरोक्त वर्णित Expression Notation Tree में **"Operator Stack"** का उपयोग किया जाता है।

Example (उदहारण): -

S.N.	PREFIX	INFIX	POSTFIX
01	+ a b	a + b	a b +
02	* + a b c	(a + b) * c	a b + c *
03	* a + b c	a * (b + c)	a b c + *
04	+ / a b / c d	a / b + c / d	a b / c d / +
05	* + a b + c d	(a + b) * (c + d)	a b + c d + *
06	- * + a b c d	((a + b) * c) - d	a b + c * d -

INFIX to POSTFIX TRANSFORMATION: -

इस प्रोसेस मे Stack का Use किया जाता है। इसमे Stack, Operator को Hold करता है। इसका Algorithm है: -

i. Expression को Left To Right Side से स्कैन करेंगे।
ii. अगर Expression मे Operand है तब इसे Output String मे ट्रांसफॉर्म करेंगे।
iii. अगर Current Input Token '(' है तब इसे Stack मे Push करेंगे।
iv. अगर Current Input कोई Operator है और Operator जिसका Precedence Equal और Higher है फिर Operator को Stack मे Push करेंगे।
v. अगर Current Input कोई Operator है और Operator जिसका Precedence, Stack में उपस्थित Operator से Lower है तब Stack के Current Operator को Stack से Pop कर Output String में जोड़ देंगे एवं Currrent Input Operator को Stack में Push कर देंगे।
vi. अगर Current Input Token ')' है तब सभी Operator को Stack से निकल देंगे और इसे आउटपुट String मे Transform करेंगे जब तक '(' नहीं आ जाता है।
vii. जब Input String का End हो जाता है। सभी Operator को Stack से निकल देगे और इसे आउटपुट String मे Transform करेंगे।
viii. प्राप्त Result Postfix Form में होगा।

- यह Algorithm Error को Handel नहीं करता है इसलिए Bracket ( ) को केयरफुल Handel करना होता है।

Example (उदहारण) : - (A * B – (C – D)) / (E + F)

Input	Stack	Output String
(		
	(	

Step 1

Input	Stack	Output String
A		A
	(	

Step 2

Input	Stack	Output String
*		A
	*	
	(	

Step 3

Input	Stack	Output String
B		AB
	*	
	(	

Step 4

Input	Stack	Output String
–		AB*
	–	
	(	

Step 5

Input	Stack	Output String
(		AB*
	(	
	–	
	(	

Step 6

Input	Stack	Output String
C		AB*C
	(	
	–	
	(	

Step 7

Input	Stack	Output String
–	–	AB*C
	(	
	–	
	(	

Step 8

Input	Stack	Output String
D	–	AB*CD
	(	
	–	
	(	

Step 9

Input	Stack	Output String
)		AB*CD–
	(	
	–	
	(	

Step 10

Input	Stack	Output String
)		AB*CD– –

Step 11

Input	Stack	Output String
/		AB*CD– –
	/	

Step 12

Input	Stack	Output String
(		AB*CD– –
	(	
	/	

Step 13

Input	Stack	Output String
E		AB*CD– –E
	(	
	/	

Step 14

Input	Stack	Output String
+		AB*CD– –E
	+	
	(	
	/	

Step 15

Input	Stack	Output String
F		AB*CD– –EF
	+	
	(	
	/	

Step 16

Input	Stack	Output String
)		AB*CD– –EF+
	/	

Step 17

Input	Stack	Output String
		AB*CD– –EF+/
	/	

Step 18

INFIX to PREFIX TRANSFORMATION: -

इस प्रोसेस मे Stack का Use किया जाता है। इसमे Stack, Operator को Hold करता है। इसका Algorithm है: -

i. दिए गए Expression को Reverse Order में Convert करेंगे। फिर Left To Right Side से स्कैन करेंगे।
ii. अगर Expression में Operand है तब इसे Output String में ट्रांसफॉर्म करेंगे।
iii. अगर Current Input Token ')' है तब इसे Stack मे Push करेंगे।
iv. अगर Current Input कोई Operator है और Operator जिसका Precedence Equal और Higher है फिर Operator को Stack मे Push करेंगे।
v. अगर Current Input कोई Operator है और Operator जिसका Precedence, Stack में उपस्थित Operator से Lower है तब Stack के Current Operator को Stack से Pop कर Output String में जोड़ देंगे एवं Currrent Input Operator को Stack में Push कर देंगे।
vi. अगर Current Input Token '(' है तब सभी Operator को Stack से निकल देंगे और इसे Output String में Transform करेंगे जब तक ')' नहीं आ जाता है।
vii. जब Input String का End हो जाता है। सभी Operator को Stack से निकल देगे और इसे Output String में Transform करेंगे।
viii. अंत में प्राप्त Result को पुनः Reverse Order में Convert करेंगे और प्राप्त Result Prefix Form में होगा।

- यह Algorithm Error को Handel नहीं करता है इसलिए Bracket ( ) को केयरफुल Handel करना होता है।

Example (उदहारण) : - (A * B – (C – D)) / (E + F) Reverse Expression: -) F + E (/)) D – C (– B * A (

Input	Stack	Output String
)	)	

Step 1

Input	Stack	Output String
F	)	F

Step 2

Input	Stack	Output String
+	+)	F

Step 3

Input	Stack	Output String
E	+)	FE

Step 4

Input	Stack	Output String
(	+)	FE+

Step 5

Input	Stack	Output String
/	/	FE+

Step 6

Input	Stack	Output String
)	) /	FE+

Step 7

Input	Stack	Output String
)	)) /	FE+

Step 8

Input	Stack	Output String
D	)) /	FE+D

Step 9

Input	Stack	Output String
–	–	FE+D
	)	
	)	
	/	

Step 10

Input	Stack	Output String
C	–	FE+DC
	)	
	)	
	/	

Step 11

Input	Stack	Output String
(		FE+DC–
	)	
	/	

Step 12

Input	Stack	Output String
–		FE+DC–
	–	
	)	
	/	

Step 13

Input	Stack	Output String
B		FE+DC–B
	–	
	)	
	/	

Step 14

Input	Stack	Output String
*	*	FE+DC–B
	–	
	)	
	/	

Step 15

Input	Stack	Output String
A	*	FE+DC–BA
	–	
	)	
	/	

Step 16

Input	Stack	Output String
(	*	FE+DC–BA*–/
	–	
	)	
	/	

Step 17

1. Again Reverse Expression (FE+DC–BA*–/) : - " **/ – * A B – C D + E F"**

POSTFIX EXPRESSION EVALUATION

इस प्रोसेस मे Stack का Use किया जाता है। इसमे Stack, Operand को Hold करता है। इसका Algorithm है: -

i. किसी Expression को Left से Right की तरह Scan करेंगे।
ii. अगर Expression में Operand है तब इसे Stack मे Push करेंगे।
iii. अगर Current Input Token '(' है तब इसे Stack मे Push करेंगे।
iv. अगर Current Input कोई Operator है तब इसे Operand पर अप्लाई करेंगे और Operand पर Operator को लगा कर आये आउटपुट को Operand की जगह Replace कर देंगे।
v. यह Scan जब तक होगी जब तक की एक Value नहीं आ जाती है।

इस Algorithm की Time Complxity O (n) होती है क्योकि इसमें Expression को केवल एक बार स्कैन किया जाता है ।

Example (उदहारण): - Infix Expression (5 + 3) * (8 – 2) PostFix Expression: - **5 3 + 8 2 – ***

Input	Stack	Output String
5		
	5	

Step 1

Input	Stack	Output String
3		
	3	
	5	

Step 2

Input	Stack	Output String
+		5 + 3 = 8
	8	

Step 3

Input	Stack	Output String
8		
	8	
	8	

Step 4

Input	Stack	Output String
2		
	2	
	8	
	8	

Step 5

Input	Stack	Output String
–		8 – 2 = 6
	6	
	8	

Step 6

Input	Stack	Output String
*		8 * 6 = 48
	48	

Step 7

Final Answer for Expression (5 + 3) * (8 – 2) is **48**

RECURSION: -

- जिस Function में वही Function को Call किया जाता है, उसे Recursion कहते है।
- Recursion एक ऐसा Process है, जो Loop के तरह काम करता है।
- Recursion को एक Satisfied Condition लगती है, जिससे Recursive Function काम करना बंद कर दे।
- Recursive Function तबतक Call होता रहता है, जबतक उसका Satisfaction नहीं होता।
- अगर Recursive Function का Satisfaction नहीं हो तो 'Infinite Looping' की संभावना होती है।

FACTROIAL USING RECURSION Example: - FACT OF VALUE 5 (5 * 4 * 3 * 2 * 1) = **120**

Input	Stack	Output
Fact (5)		5 * Fact (4)
	5	

Step 1

Input	Stack	Output
Fact (4)		5 * 4 * Fact (3)
	4	
	5	

Step 2

Input	Stack	Output
Fact (3)		5 * 4 * 3 Fact (2)
	3	
	4	
	5	

Step 3

Input	Stack	Output
Fact (2)		5 * 4 * 3 * 2 * Fact (1)
	2	
	3	
	4	
	5	

Step 4

Input	Stack	Output
Fact (1)	1	5*4*3*2*1
	2	
	3	
	4	
	5	

Step 5

Input	Stack	Output
1st retrunt		(2 * 1) = 2
	2	
	3	
	4	
	5	

Step 6

Input	Stack	Output
2nd Return		3 * 2 = 6
	6	
	4	
	5	

Input	Stack	Output
3rd Retrun		4 * 6 = 24
	24	
	5	

Input	Stack	Output
4th Return		5 * 24 = 120
	120	

QUEUE (क्यू): -

Queue एक Non – Primitive तथा Linear डेटा स्ट्रक्चर है यह FIFO (First In First Out) के सिद्धान्त पर कार्य करता है अर्थात वह Item जो की सबसे पहले Add किया जाता है वही Item सबसे पहले Remove किया जायेगा और वह Item जो की सबसे अंत में Add किया जाता है उसे अंत में ही Remove किया जायेगा।

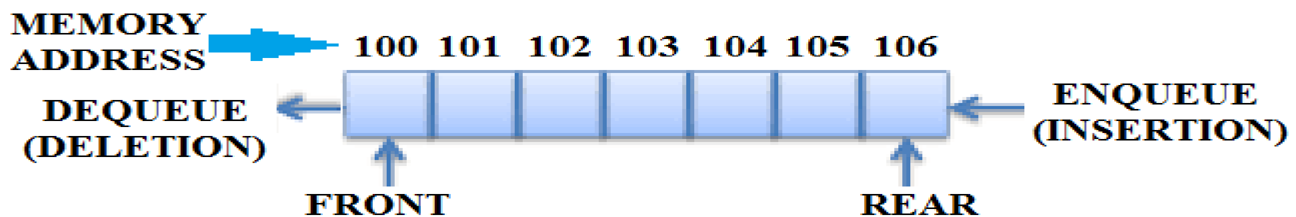
Queue को हम अपनी वास्तविक दुनिया में अक्सर ही प्रयोग करते हुए देखते है, इसका उदाहरण है: - "एक व्यक्ति जो रेलवे में टिकट रिजर्वेशन की लाइन में सबसे पहले लगा होता है और सबसे पहले टिकट लेकर चले जाता है, वह व्यक्ति जो Last में लगा हुआ रहता है वह अंत में ही बाहर जायेगा।"

Queue में दो End होते है एक Front End होता है तथा दूसरा Rear End होता है। Rear End में Item को Add किया जाता है तथा Front End से Item को Remove किया जाता है। क्यू से संबंधित कुछ शब्द हैं जैसे Enqueue (एनक्यू), Dequeue (डीक्यू), Front (फ्रंट), Rear (रियर): -

1. Enqueue (एनक्यू): - जब क्यू में डाटा को भेजा जाता है तो इसे एनक्यू कहते हैं। एनक्यू ऑपरेशन को Insertion (इंसर्शन) भी इस्तेमाल किया जाता है।
2. Dequeue (डीक्यू): - जब क्यू में से डाटा को निकाला जाता है तो इसे डीक्यू कहते हैं। डीक्यू को Deletion भी कहते हैं।
3. Front (फ्रंट): - क्यू में सबसे पहली पोजीशन में जो डाटा होता है उसे फ्रंट कहते हैं।
4. Rear (रियर): - सबसे आखिरी पोजीशन में जो डाटा होता है उसके लिए रियर शब्द इस्तेमाल किया जाता है।

Queue की निम्नलिखित शर्तें होती है: -

1. FRONT < 0 or (Front = Rear = – 1) है तो, Queue खली (रिक्त) है।
2. REAR = Size of Queue है तो, Queue पूरा भरा हुआ होता है।
3. FRONT < REAR है तो, Queue में कम से कम एक Item तो होता ही है।
4. [(REAR – FRONT) + 1] Queue में कुल Item की संख्या को प्रदर्शित करता है।



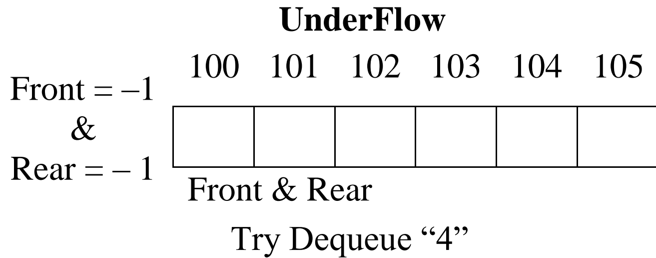
डायग्राम में देख सकते हैं कि मैमोरी में क्यू एक लीनियर ऐरे की तरह स्टोर होता है। जब भी डाटा निकालेंगे तो जो डाटा पहले आया था वही पहले बाहर निकलेगा जैसे की 100 एड्रेस पर सबसे पहले डाटा आया था इसलिए 100 एड्रेस में मौजूद डाटा सबसे पहले बाहर निकलेगा। इसके बाद डाटा निकालेंगे तो 101 एड्रेस से निकलेगा इसी क्रम से डाटा बाहर निकलता रहेगा।

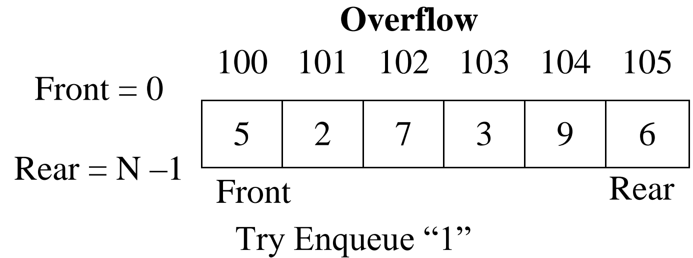
यदि डाटा को क्यू में डालना हो तो सबसे आखिरी पोजीशन के बाद नया डाटा आएगा इस क्यू की सबसे आखिरी पोजीशन एड्रेस 106 पर है नया डाटा डालने पर इसके आगे के एड्रेस यानी 107 में नया डाटा आएगा। इसके आगे यदि और डाटा डालेंगे तो मेमोरी में 108 एड्रेस पर एक नयी जगह डाटा के लिए बनाई जायेगी इस नयी जगह पर नया डाटा स्टोर होगा इस तरह नया डाटा डालने पर यही क्रम चलता रहेगा।

Underflow: - यदि Queue Empty हो (Front = Rear = – 1 or Front < 0) और उसमें से किसी Element को Delete करने का प्रयास किया जाए तो ऐसी Situation Underflow कहलाती है।

Overflow: - यदि Queue Full हो (Front > 0 and Rear = N – 1) और उसमें नया Element Add करने का प्रयास किया जाये तो ऐसी Situation Overflow कहलाती है।

Example (उदाहरण): -





Basic Operation on Queue (क्यू के बेसिक ऑपरेशन): -

1. **ENQUEUE (INSERTION): –** जब Queue में कोई Item Insert किया जाता है तो वह Operation Enqueue (Insertion) Operation कहलाता है।

Step 1 – सबसे पहले Queue के स्टेटस को चेक किया जाता है कि ये भरा हुआ तो नहीं है।

Step 2 – अगर Queue फुल होता है तो यह एरर मैसेज दे कर Exit कर जाता है।

Step 3 – अगर Queue फुल नहीं है तो Rear में डाटा एलिमेंट को ऐड करके Rear वैल्यू को एक बढ़ा दिया जाता है।

Step 4 – Enqueue ऑपरेशन कम्पलीट होता है।

2. **DEQUEUE (DELETION): –** जब Queue से कोई Item Delete किया जाता है तो वह Operation Dequeue (Deletion) Operation कहलाता है।

Step 1 – सबसे पहले Queue के स्टेटस को चेक करता है कि Queue खाली तो नहीं है।

Step 2 – अगर Queue खाली है तो एरर मैसेज दे कर Exit करता है।

Step 3 – अगर Queue खाली नहीं है तो Front वैल्यू को एक बढ़ा दिया जाता है और Front एलिमेंट को Remove कर दिया जाता है।

Step 4 – Dequeue ऑपरेशन कम्पलीट होता है।

3. SEARCHING: -

Step 1 – सबसे पहले Queue के स्टेटस को चेक करता है कि Queue खाली तो नहीं है।

Step 2 – अगर Queue खाली है तो एरर मैसेज दे कर Exit करता है।

Step 3 – अगर Queue खाली नहीं है तो Search एलिमेंट को Front एलिमेंट से Match किया जाता है। Front वैल्यू को एक – एक कर बढ़ा दिया जाता है।

Step 4 – जब Search एलिमेंट Front एलिमेंट से Match करता है तो "Search Successful" Message के साथ Queue से बाहर आ जाता है। Search ऑपरेशन कम्पलीट होता है।

<u>Type of Queue (क्यू के प्रकार): -</u>

Queue Data Structure 3 प्रकार के होते है: -

1. Priority Queue (प्रायोरिटी क्यू)।
2. Circular Queue. (सर्कुलर क्यू)।
3. Doubly Ended Queue (DEQueue) (डबली इंडेड क्यू)।

1. **<u>Priority Queue (प्रायोरिटी क्यू): -</u>** Priority Queue, Element का इस प्रकार का संग्रह हैं जिसमे प्रत्येक Element को एक Priority Provide की जाती हैं तथा Priority के हिसाब से Element को Insert, Remove और Process किया जाता हैं। यह निम्नलिखित Rules पर आधारित हैं: -
   - High priority प्राप्त करने वाले Elements को Low Priority प्राप्त करने वाले Element से पहले Process किया जाता हैं।
   - समान Priority के Element को उसी क्रम में Process किया जायेगा जिस क्रम में उन्हें Queue में जोड़ा गया हैं।

Priority Queue के दो प्रमुख तरीके है: -

- एक One Way List का Use करके।
- Multiple Queue का Use करके।

Priority Queue के दो प्रकार है: -

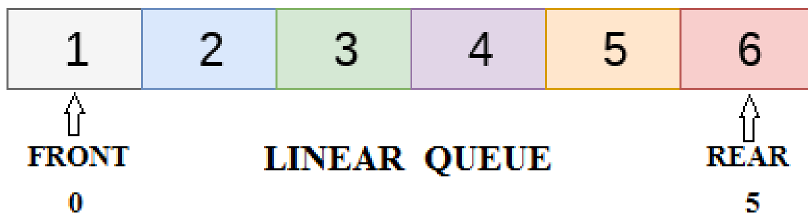
a. Ascending Priority Queue (असेंडिंग प्रायोरिटी क्यू)।
b. Descending Priority Queue (डिसेंडिंग प्रायोरिटी क्यू)।

a. **Ascending Priority Queue (असेंडिंग प्रायोरिटी क्यू): -** Ascending Priority Queue में Element किसी भी क्रम में जोड़े जा सकते हैं लेकिन सबसे छोटा Element ही सबसे पहले हटाया जाता हैं।

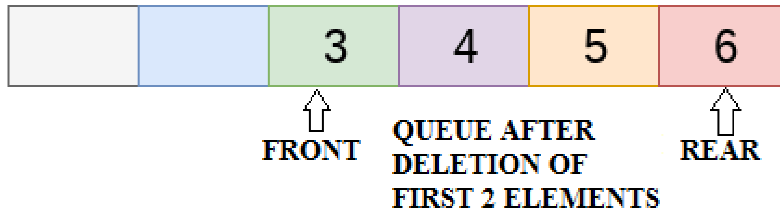
b. **Descending Priority Queue** (डिसेंडिंग प्रायोरिटी क्यू): - Descending Priority Queue, Ascending Priority Queue की तरह ही होता हैं फ़र्क़ सिर्फ इतना होता हैं की Descending Priority Queue में सबसे पहले बड़ा Element को Remove किया जाता हैं।

Priority Queue का Use Multi User System में किया जाता हैं, जिसमे अनेको Users के कई Program जब एक ही समय में Center Processor में Execute होने आते हैं तो प्रत्येक प्रोग्राम को कुछ Priority क्रम दे दिया जाता हैं तथा जिस Program की Priority ज्यादा होगी वह सबसे पहले Execute होगा।

2. **<u>Circular Queue (सर्कुलर क्यू): -</u>** Circular Queue को हम Ring - Buffer भी कहते है। Circular Queue में जो अंतिम नोड होता है वह सबसे पहले नोड से जुड़ा हुआ रहता है। जिससे की Circle का निर्माण होता है। यह FIFO के सिद्धान्त पर कार्य करता है। Circular Queue में Item को Rear End से Add किया जाता है तथा Item को Front End से Remove किया जाता है। नीचे डायग्राम में दिखाए गए Queue पर विचार करें।



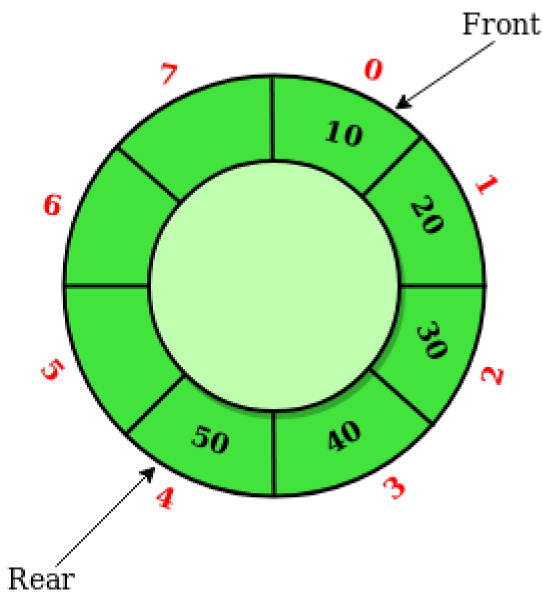
इस डायग्राम में Queue पूरी तरह से भरी हुई है और Rear Condition Full होने के कारण कोई और Element नहीं डाला जा सकता है। Rear = = Max – 1



यदि हम Queue के Front से 2 Elements को हटाते हैं, तो हम अभी भी किसी भी Element को नहीं डाल सकते हैं क्योंकि Condition Rear = = Max – 1 है।

यह Linear Queue के साथ मुख्य समस्या है, हालांकि हमारे पास Queue में Space उपलब्ध है, लेकिन हम Queue में कोई और Element नहीं डाल सकते हैं। यह केवल Memory Wastage है और हमें इस समस्या को दूर करने की आवश्यकता है। इस समस्या के समाधान में से एक Circular Queue है। Circular Queue में, पहला Index आखिरी Index के ठीक बाद आता है।

Circular Queue का Implementation एक Linear Queue के समान है। Insertion और Deletion के मामले में Logic एक Linear Queue में इससे भिन्न होता है। Circular Queue तब पूरा भरा हुआ माना जायेगा जब Front = – 1 और Rear = Max – 1 की शर्त को पूरा करें। एक Circular Queue में एक Element Insert करने के तीन Scenario हैं: -



- If (rear + 1) % Maxsize = Front, Queue पूरा भरा हुआ है। इस स्थिति में, “Overflow” होगा और इसलिए Queue में Insertion नहीं किया जा सकता है।
- If Rear ! = Max – 1, तो Rear को Mod (Maxsize) में बढ़ाया जाएगा और नया Value, Queue के पीछे Insert किया जायेगा।
- If Front ! = 0 and Rear = Max – 1, तो इसका मतलब है कि Queue पूरा नहीं भरा है इसलिए, Rear के Value को 0 पर सेट कर वहां नया Element Insert किया जा सकता है।

3. **Doubly Ended Queue (DEQueue) (डबली इंडेड क्यू): -** यह एक Liner Queue हैं जिसमे Insertion और Deletion दोनों सिरों से Possible हैं परन्तु उसे बीच से नहीं किया जा सकता हैं। DEQueue के दो प्रकार होते है जो निम्न है: -
   a. Input Restricted DEQueue (इनपुट रिस्ट्रिक्टेड डबली इंडेड क्यू).
   b. Output Restricted DEQueue (आउटपुट रिस्ट्रिक्टेड डबली इंडेड क्यू).

a. **Input Restricted DEQueue (इनपुट रिस्ट्रिक्टेड डबली इंडेड क्यू): -**

इस प्रकार के DEQueue में Items को दोनों Ends से Delete किया जा सकता है परन्तु केवल एक ही End से Insert कर सकते है।

b. **Output Restricted DEQueue (आउटपुट रिस्ट्रिक्टेड डबली इंडेड क्यू): -**

इस प्रकार के DEQueue में Items को दोनों तरफ से ही Insert किया जा सकता है परन्तु केवल एक ही End से Delete कर सकते है।

Difference between STACK And QUEUE (स्टैक और क्यू में अंतर): -

क्र.	STACK (स्टैक)	QUEUE (क्यू)
01	इसमें Element केवल एक ही End (TOP) से Insertion और Deletion होता है।	इसमें Element, Insertion Rear से और Deletion Front से होता है।
02	इसमें केवल एक ही Pointer का उपयोग TOP के लिए होता है।	इसमें दो Pinter का प्रयोग Front और Rear के लिए होता है।
03	इसमें Last Insert Element को पहले Remove किया जाता है।	इसमें First Insert Element को पहले Remove किया जाता है।
04	स्टैक Last In First Out (LIFO) को Follow करता है।	क्यू First In First Out (FIFO) को Follow करता है।
05	स्टैक के Operation “PUSH” और “POP” है।	क्यू के Operation “Enqueue” और “Dequeue” है।
06	स्टैक का स्वरुप Vertical Collection जैसा होता है।	क्यू का स्वरुप Horizontal Collection जैसा होता है।
07	स्टैक का उपयोग Recursion, Reverse, Expression Evaluation आदि के लिए किया जाता है।	क्यू का उपयोग CPU Scheduling, Memory Management आदि के लिए किया जाता है।
08	स्टैक का उदाहरण " टेबल के ऊपर एक के ऊपर एक रखे किताबो का समूह" है।	क्यू का उदाहरण " ट्रैन टिकट लेने के लिए खड़े लोगो की लाइन" है।

UNIT 04
LINKED LIST

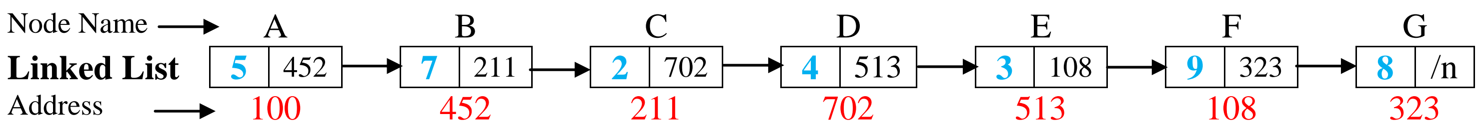
LINKED LIST (लिंक्ड लिस्ट): -

Linked List एक Non - Primitive, Linear डेटा स्ट्रक्चर है। लिंक्ड लिस्ट, यह एक ऐसी लिस्ट होती जिसमे की डाटा एलिमेंट स्टोर होते है एवं सभी डाटा एलिमेंट्स Links की हेल्प से आपस मे Connected होते है। लिंक्ड लिस्ट मे जहाँ डाटा एलिमेंट्स स्टोर होते है उसको Noded (नोड्स) कहते है एवं इन नोड्स मे डाटा Element को एक Sequence मे रखा जाता है।

Linked List, Nodes के समूह से मिलकर बना होता हैं, प्रत्येक Node के दो भाग होते हैं पहला भाग "Data Or Information" का होता है और दूसरा "Pointer Or Link" होता है। Linked List का Pointer भाग अगले Node के Address को Hold किये रहता है।

लिंक्ड लिस्ट मे हर एक नोड को Element (एलिमेंट) भी कहा जाता है और Reference Field जो की नेक्स्ट नोड को पॉइंट करता है उसको Next Pointer (नेक्स्ट पॉइंटर) or Next Link (नेक्स्ट लिंक) भी कहते है। ऐरे के बाद यह दूसरी सबसे ज्यादा काम मे आने वाला Data Structure है।

Linked List Dynamic Nature का होता है अर्थात यह एक ऐसा Data Structure होता है जिसकी Length को Run - Time में बढ़ाया या घटाया जा सकता है। Linked List का प्रयोग Tree तथा Graph को बनाने के लिए किया जाता है।



Uses of Linked List (लिंक्ड लिस्ट का उपयोग): -

1. इसे हम Stacks और Queue में आसानी से Implement कर सकते है।
2. Graphs में हम इसे Implement कर सकते है। Linked List में हम Adjacent Nodes को स्टोर कर सकते है।
3. Dynamic Memory Allocation के लिए इसका प्रयोग किया जाता है।
4. Directory के नामों को Maintain करने के लिए इसका Use किया जाता है।
5. Long Integers में लिंक्ड लिस्ट के द्वारा Arithmetic Operations को Perform किया जा सकता है।
6. Linked List के Node में Constant को स्टोर करके Polynomials को Manipulate किया जा सकता है।
7. Sparse Matrices को प्रस्तुत करने के लिए इसका Use किया जाता है।
8. इसका इस्तेमाल हम Image Viewer में Images को Next और Previous करने के लिए कर सकते है।
9. Web Browser में हम पिछले और अगले Url को Back और Next Button के द्वारा देख सकते है।
10. लिंक्ड लिस्ट की सहायता से हम Music Player में Songs को आगे और पीछे कर सकते है।
11. Hash Tables को Implement करने के लिए भी इनका Use किया जा सकता है।
12. Word और Photoshop में Undo करने के लिए भी लिंक्ड लिस्ट का Use होता है।

Advantage of Linked List (लिंक्ड लिस्ट के लाभ): -

1. Linked List एक Dynamic Data Structure है।
2. Linked List को Run Time में हम घटा भी सकते है और बढ़ा भी सकते हैं। अर्थात Memory को Run Time में ही Allocate और Deallocate कर सकते है।
3. इसमें हम आसानी से Insertion और Deletion कार्यों को कर सकते है। अर्थात आसानी से हम Node को Insert तथा Delete कर सकते है।
4. लिंक्ड लिस्ट में Memory को अच्छी तरह Utilize किया जाता है, क्योंकि हमें इसमें पहले से मैमोरी Allocate नहीं करनी पड़ती है।
5. इसका Access Time बहुत ही Fast होता है और बिना Memory Overhead के एक नियत समय में Access कर सकते है।
6. Linked List का प्रयोग करके हम Linear Data Structures जैसे: - Stack, Queue को आसानी से Implement कर सकते है।

Disadvantage of Linked List (इसकी कुछ हानियाँ): -

1. लिंक्ड लिस्ट में Array की तुलना में Elements को स्टोर करने के लिए अधिक Memory की जरूरत पड़ती है। क्योंकि लिंक्ड लिस्ट की प्रत्येक Node एक Pointer को Contain करती है जिसके कारण इसे अधिक Memory की आवश्यकता होती है।
2. लिंक्ड लिस्ट में Nodes को Traverse करना बहुत कठिन होता है। इसमें हम किसी एक Element को Randomaly एक्सेस नही कर सकते है। (जैसा कि हम Array में Index के द्वारा करते है।) उदाहरण के लिए:- अगर हम किसी 'n' Position में स्थित Node को Traverse करना चाहें तो हमें 'n' से पहले आने वाले सभी Nodes को Traverse करना पड़ेगा। जिससे हमारा बहुत सारा समय नष्ट हो जायेगा।
3. Linked List में Reverse Traversing करना बहुत ही Difficult होता है। Doubly Linked में हम आसानी से कर सकते है परन्तु उसमें Pointer के लिए ज्यादा Memory की जरूरत होती है।

Pointer (पॉइंटर): -

Pointer वह Variable है जो कि Linked List के Address को contain किये रहता है। या पॉइंटर वह Variable है जो कि दूसरें Variable के Address को Contain किये रहता है। Address किसी भी साधारण वेरिएबल में सुरक्षित नहीं हो सकते, इनको सुरक्षित करने के लिए केवल Pointers ही प्रयोग में लाये जाते हैं। Pointer पर किये जा सकने वाले वैध कार्य निम्नलिखित हैं:-

1. एक Cast ऑपरेटर को प्रयुक्त करके समान डेटा प्रकार के पॉइंटर्स को Assign करना।
2. एक Pointer को किसी Integer के साथ जोड़ना अथवा घटाना।
3. किन्ही दो Pointers की तुलना करना जोकि एक ही Linked List को Point करते हों।
4. किसी भी पॉइंटर को NULL Assign करना अथवा NULL से तुलना करना।

NULL Pointer (नल पॉइंटर): -

किसी Linked List के अंत वाले नोड में जो Pointer का उपयोग किया जाता है वह NULL Pointer कहलाता है जिसका उपयोग Linked List को Stop या बंद करने के लिए करते है।

Types of Linked List (लिंक्ड लिस्ट के प्रकार): -

साधारणत: Link List के तीन अलग-अलग प्रकार है: -

(1) Singly Link List (सिंगली लिंक लिस्ट).

(2) Doubly Link List (डब्ली लिंक लिस्ट).

(3) Circular Link List (सर्क्युलर लिंक लिस्ट).

(1) Singly Linked List (सिंगली लिंक लिस्ट) (SLL): -

Singly Link List ऐसी Link List होती है जिसमे सारे Node एक दूसरे के साथ क्रम मे जुड़े होते है इसलिए इसको Linear Link List या One Way Chain भी कहते है। Singly Linked List को केवल एक ही दिशा में Traverse किया जा सकता है। दूसरे शब्दों में, हम कह सकते हैं कि प्रत्येक Node में केवल Next Pointer होता है, इसलिए हम List में एक बार Pointer के आगे जाने के बाद पीछे वाले Node को Access नहीं किया जा सकता है। Singly Linked List के प्रत्येक Node में दो Fields होते है। पहला वह Field होता है जहां डेटा स्टोर रहता है। दूसरा Pointer या लिंक होता है।

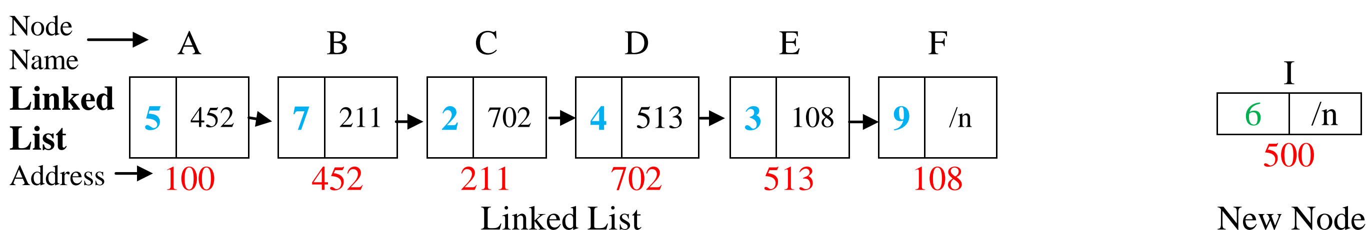
Singly Linked List Operations (सिंगली लिंक्ड लिस्ट के ऑपरेशन): -

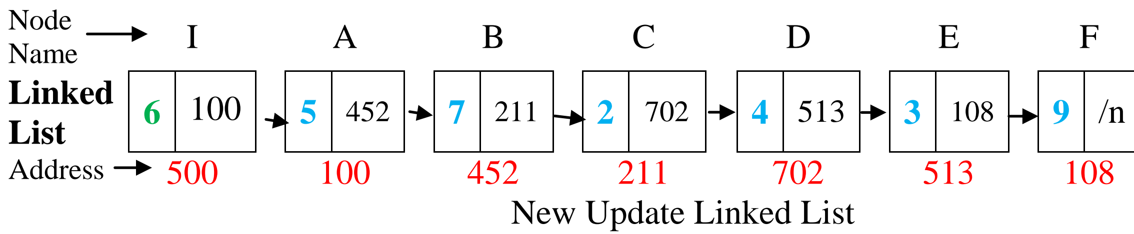
a. Insertion – List में एक Element (Node) को Add करना।
b. Deletion – List में से Element को Delete करना।
c. Traverse – List में Elements को Traverse करना।
d. Search – दी हुई Key के द्वारा Element को Search करना।

a. Insertion Operation: - एक Singly Linked List में Insertion विभिन्न Positions पर की जा सकती है। Insert किया जा रहे नए Node को स्थिति के आधार पर निम्नलिखित Categories में वर्गीकृत किया गया है।

i. Insertion at Beginning (शुरुवात में जोड़ना): - इसमें List के शुरुआत में किसी भी Node को Insert करना शामिल है। हमें List के Head के रूप में नया Node बनाने के लिए बस कुछ लिंक Adjustments की आवश्यकता है।

- सबसे पहले इसमें नए Node के लिए Memory को Allocate किया जाता है।
- Data को स्टोर किया जाता है।
- New Node जो है वह Linked List का नया Head बन जाता है, क्योंकि इसे सबसे पहले Insert किया है।

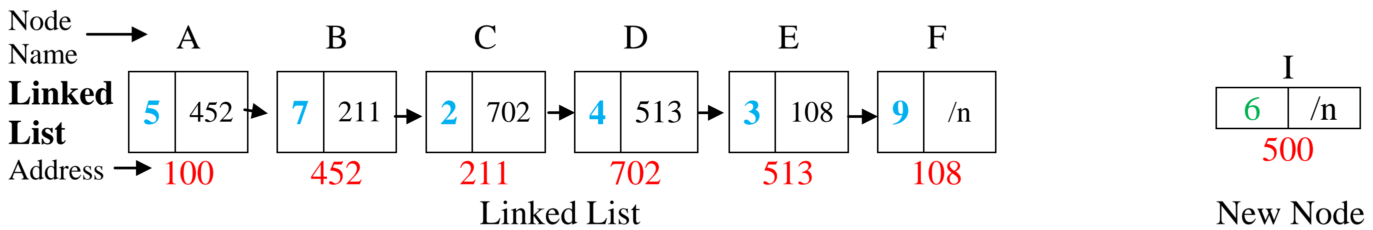




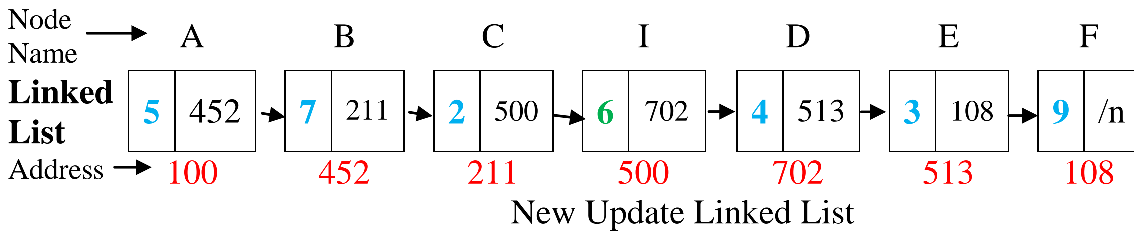
New Update Linked List

ii. Insert at Middle Anywhere (बीच में जोड़ना): - इसमें Linked List के Specified नोड के बाद नए नोड को जोड़ते है। हमें नोड तक पहुंचने के लिए वांछित संख्या के Nodes को Travers करने के बाद नया नोड डाला जाएगा।

- पहले इसमें Memory को Allocate किया जाता है और New Node के लिए Data को स्टोर किया जाता है।
- New Node के लिए Position को Search किया जाता है।
- नये Node को Include करने के लिए Next Pointers को बदल दिया जाता है।



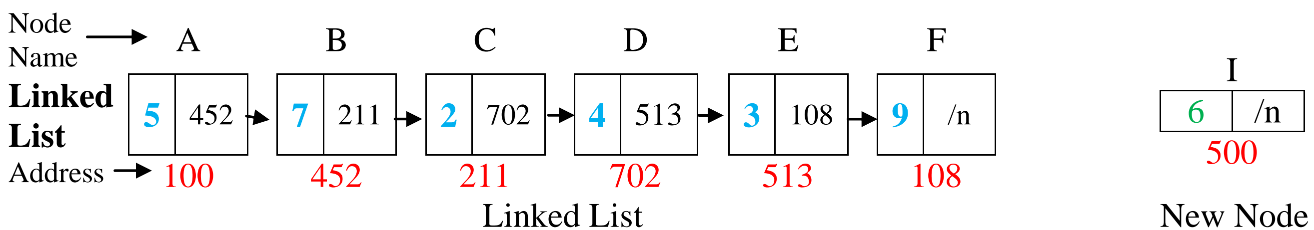
Linked List New Node



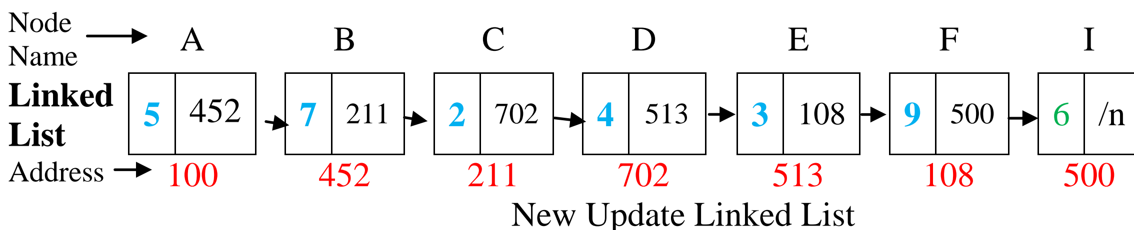
New Update Linked List

iii. Insertion at End (अंत में जोड़ना): - इसमें Linked List के आखिरी में नोड को जोड़ते है। नए नोड को सूची में एकमात्र नोड के रूप में या इसे Last Node के रूप में डाला जा सकता है।

- पहले इसमें Memory को Allocate किया जाता है और New Node के लिए Data को स्टोर किया जाता है।
- Last Node को Search किया जाता है।
- Last Node के Next Pointer को नए Node के Address के साथ बदल दिया जाता है।



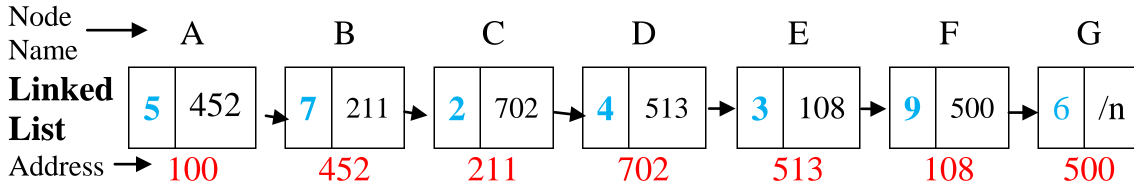
Linked List New Node



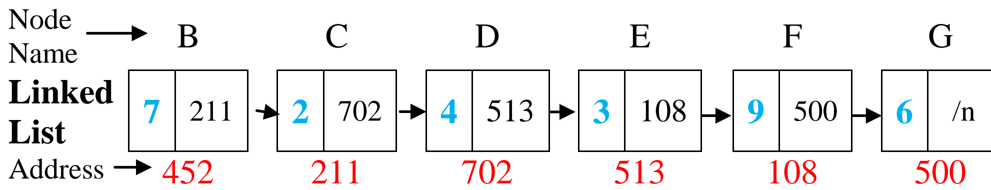
New Update Linked List

b. <u>Deletion Operation: -</u> एक Singly Linked List से नोड का Deletion विभिन्न Positions पर किया जा सकता है। Delete किये जा रहे नोड की Position के आधार पर, इस Operation को निम्न श्रेणियों में वर्गीकृत किया गया है: -

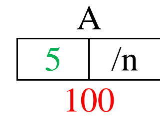
i. Deletion at Beginning (प्राम्भ से हटाना): - इसमें List की शुरुआत से एक नोड को हटाया जाता है। यह इन सब में सबसे सरल Operation है। इसके लिए सिर्फ Node Pointers में कुछ Adjustments की जरूरत है। जब शुरू के Node को Delete करते है तो उसके बाद Nodes को दुबारा Link नही करना पड़ता। लेकिन हमें इसमें Head को Second Node की तरफ Point करना पड़ता है। head = head -> next;



Linked List



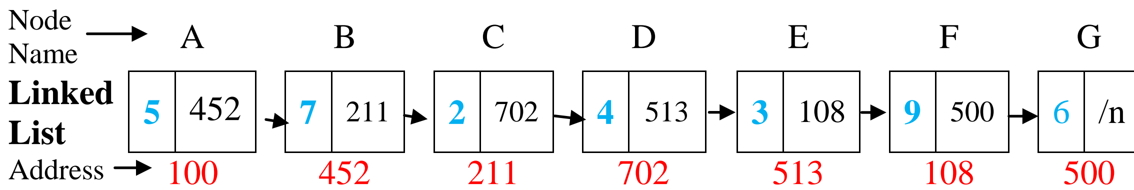
New Updated Linked List



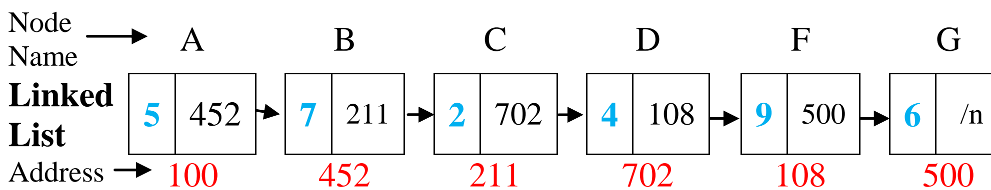
Delete Node

ii. Deletion After Specified Place (बीच में से हटाना): - इसमें List में Specified नोड के बाद नोड को हटाया जाता है। हमें नोड तक पहुंचने के लिए वांछित संख्याओं को छोड़ना होगा जिसके बाद नोड हटाया जाता है। इसके लिए List में Traversing की आवश्यकता होती है।

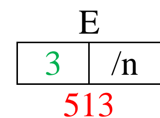
- सबसे पहले इसमें उस Node को Search किया जाता है जिसे हमने Delete करना है।
- Node को List में से निकालने के लिए Next Pointers को बदलना पड़ता है।



Linked List



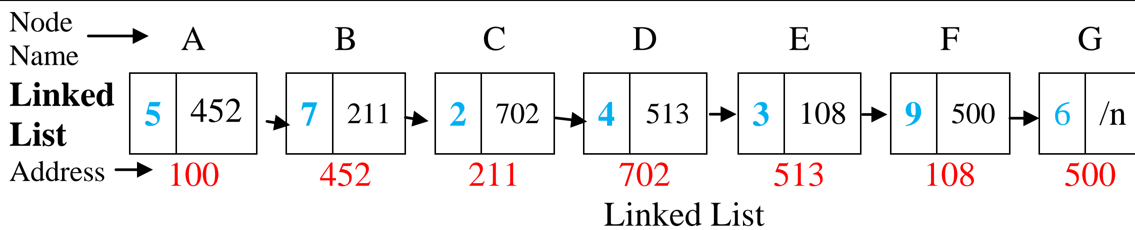
New Updated Linked List



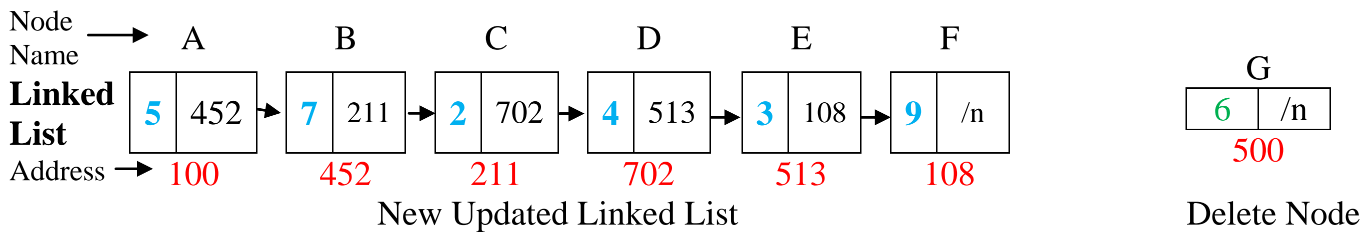
Delete Node

iii. Deletion at End (अंतिम से हटाना): - इसमें List के अंतिम Node को हटाया जाता है। List या तो खाली हो सकती है या भरी हो सकती है। अलग-अलग परिदृश्यों (Scenarios) के लिए अलग-अलग तर्क (Logic) लागू किए जाते हैं।

- इसमें Second Last (पीछे से दूसरे) Node को Search किया जाता है।
- इसके Next Pointer को Null में Change कर दिया जाता है।



Linked List



New Updated Linked List

Delete Node

c. Traversing: - Traversing में, हम सूची के प्रत्येक नोड को कम से कम एक बार उस पर जाते है। ताकि उस पर कुछ विशिष्ट (Specific) ऑपरेशन किये जा सके, उदाहरण के लिए, List में मौजूद प्रत्येक नोड के Data भाग को Print करना। Traverse Operation में, Recursive Operation का प्रयोग Linked List को Reverse Order (उल्टे क्रम) में Traverse करने के लिए किया जाता है।

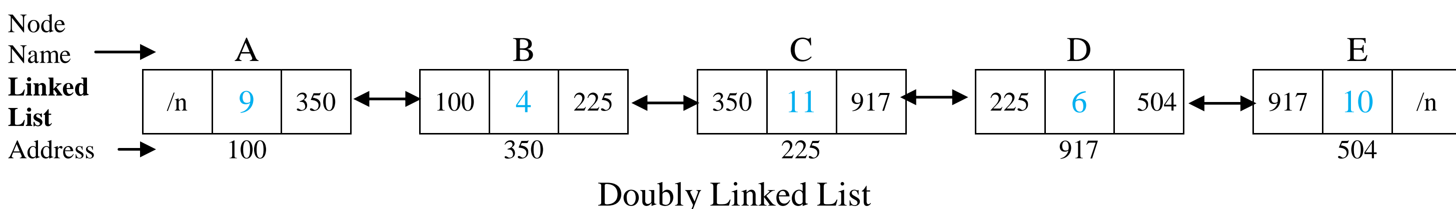
d. Searching: - Linked List में Node को Search करने के लिए Sequential Search का प्रयोग सबसे ज्यादा किया जाता है। Searching में, हम दिए गए Element (Key) के साथ सूची के प्रत्येक Element से Match करते हैं। यदि Element किसी भी Location पर पाया जाता है, तो उस Element की Location आती है अन्यथा Null वापस आता है।

(2) Doubly Link List (डब्ली लिंक लिस्ट) (DLL): -

एक Doubly Linked List जिसमें 1 से 3 तक की संख्या वाले तीन Nodes हैं। बीच वाला भाग डाटा को प्रदर्शित करता है। हर नोड में दो बिंदु होते हैं, एक अगले नोड की ओर इशारा करते हैं और दूसरे पिछले नोड की ओर इशारा करते हैं। पिछले नोड के अगले Pointer और पहले नोड (हेड) के पिछले Pointer शून्य (NULL Poiter /n) हैं।

एक Singly Linked List में, हम केवल एक ही दिशा में आगे बढ़ सकते हैं, क्योंकि प्रत्येक नोड में अगले नोड का Address होता है और इसके पिछले Nodes का कोई रिकॉर्ड नहीं होता है। हालाँकि, Doubly Linked List, Singly Linked List की इस सीमा को पार कर जाती है। इस Fact के कारण कि, List के प्रत्येक नोड में इसके पिछले नोड का Address होता है, हम प्रत्येक Node के पिछले भाग के अंदर संग्रहीत पिछले Address का उपयोग करके पिछले नोड के बारे में सभी Details पा सकते हैं।

Doubly Linked List हर Node के लिए अधिक स्थान की खपत करती है और इसलिए , Insertion और Deletion जैसे अधिक Expansive Basic Operations का कारण बनती है। हालाँकि, List के Elements में आसानी से हेरफेर कर सकते हैं क्योंकि List दोनों दिशाओं (आगे और पीछे) में Pointers बनाए रखती है।



Doubly Linked List

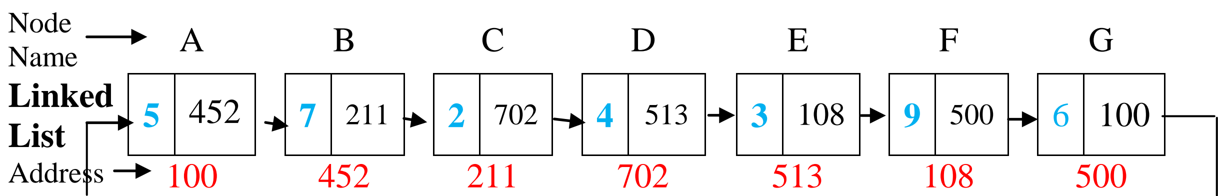
Doubly Linked List के बारे में शेष सभी Operations निम्न तालिका में वर्णित हैं।

SN	Operation	Description
1	Insertion at Beginning	: शुरुआत में Linked List में नोड जोड़ना।
2	Insertion at End	: नोड को Linked List में अंत में जोड़ना।
3	Insertion After Specified Node	: Specific Node के बाद नोड को Linked List में जोड़ना।
4	Deletion at Beginning	: List की शुरुआत से नोड को हटाना।
5	Deletion at End	: List के end से नोड को हटाना।
6	Deletion After Specified Node	: दिए गए डेटा वाले Node के ठीक बाद वाले Node को हटाना है।
7	Searching	: खोजे जाने वाले आइटम के साथ प्रत्येक Node Data की तुलना करना और List में आइटम के स्थान को वापस करना अगर आइटम वापस मिला तो अथवा Null.
8	Traversing	: Searching, Sorting, Display, आदि जैसे कुछ विशिष्ट ऑपरेशन करने के लिए कम से कम एक बार List के प्रत्येक नोड पर जाना।

(3) <u>Circular Linked Lists (सर्क्युलर लिंक लिस्ट): -</u>

यह ऐसी Link List होती है जिसकी कोई शुरुआत व अन्त नहीं होता है एक Single Link List के पहले Node से अंतिम Node को जोड़ दिया जाता है अतः List के Last Node मे List के First Node का Address रखा जाता है। Simple Link List मे Last Node का Link Part हमेशा NULL होता है लेकिन Circular Linked List मे List के Last Node के Link Part मे हमेशा Head का Address रहता है अतः Circular Link List का आखरी Node हमेशा List के First Node को Point करता है। किसी भी नोड के Link Part में कोई Null value मौजूद नहीं है।

Circular Linked List ज्यादातर ऑपरेटिंग सिस्टम में कार्य रखरखाव में उपयोग किया जाता है। ऐसे कई उदाहरण हैं जहां कंप्यूटर साइंस में Circular Linked List का इस्तेमाल किया जा रहा है, जिसमें Browser Surfing भी शामिल है, जहां User द्वारा अतीत में देखे गए पेजों का रिकॉर्ड Circular Linked List के रूप में बना रहता है और पिछले बटन पर क्लिक करने पर फिर से Access किया जा सकता है।



<u>Difference Between Array and Linked List (ऐरे और लिंक्ड लिस्ट में अंतर): -</u>

S. NO.	ARRAY	LINKED LIST
01.	Array समान प्रकार के Data Type का एक Ordered Collection होता है।	Linked List समान प्रकार के Elements का एक Ordered Collection होता है जो कि एक दूसरे से Pointers के द्वारा Connect रहते है।
02.	यह Random Access को Support करता है जिसका	यह Sequential Access को सपोर्ट करता है, जिसका

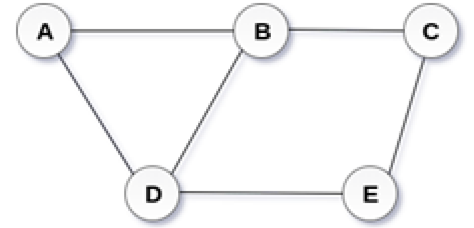
	मतलब यह है कि हम इसे Direct इसके Index का प्रयोग करके Access कर सकते है जैसे- 1st Element के लिए arr[0], 8वें Element के लिए arr[7] आदि।	मतलब है कि Linked List में किसी Elements / Nodes को एक्सेस करने के लिए हमें पूरी List को Sequentially Traverse करना पड़ेगा।
03.	इसमें Elements को Contiguous Memory Location में स्टोर किया जाता है।	इसमें नये Elements को Memory में कहीं भी स्टोर किया जा सकता है।
04.	इसमें Elements को बहुत तेजी से Access कर सकते है तथा इसकी नियत Time Complexity O(1) है।	इसकी Time Complexity O(n) है।
05.	इसमें, Insertion तथा Deletion ऑपरेशन में अधिक Time लगता है क्योंकि मैमोरी लोकेशन Continuous तथा Fix होते है।	इसमें Insertion तथा Deletion ऑपरेशन Fast होते है।
06.	इसमें Memory को Compile Time में Allocate किया जाता है। इसे Static Memory Allocation भी कहते है।	इसमें Memory को Run Time में Allocate किया जाता है। इसे Dynamic Memory Allocation भी कहते है।
07.	यह Single Dimensional, Two Dimensional या Multi-Dimensional हो सकते है।	यह Linear (Singly), Doubly या Circular हो सकते है।
08.	Array का Size इसके Declaration के समय Specify किया जाता है।	Linked List का साइज़ Run-Time में बढ़ता है जैसे जैसे इसमें नए नोड Add किये जाते है।

UNIT 05
GRAPH AND TREE

GRAPH (ग्राफ): -

ग्राफ एक Non Primitive, Non Linear डेटा स्ट्रक्चर होता है। इसमें एक Vertax (Node) का समूह होता है। एक Vertex दूसरे Vertex के साथ जुड़े रहते है और इन दो Vertex के मध्य Connection को हम Edge कहते है। Edge दो Nodes के मध्य एक कम्युनिकेशन लिंक की तरह कार्य करता है। एक Graph G को एक Order Set G (V, E) के रूप में परिभाषित किया जा सकता है जहां V(G) Vertices के सेट का प्रतिनिधित्व करता है और E(G) Edges के सेट का प्रतिनिधित्व करता है जो इन Vertices को जोड़ने के लिए उपयोग किया जाता है। एक Graph को Cyclic Tree के रूप में देखा जा सकता है, जहां Vertices (Nodes) Parent Child Connection रखने के बजाय उनके बीच किसी भी अन्य जटिल (complex) Connection को बनाए रखते हैं। एक Graph, Directed या Undirected हो सकता है।

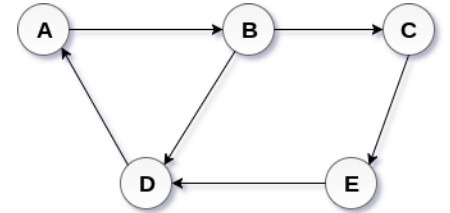
Graph G(V,E) जिसकी 5 Vertices है (A, B, C, D, E) और 6 Edges है ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A) को निम्न Figure में दिखाया गया है।



Undirected Graph

Undirected Graph (अनडाइरेक्टेड ग्राफ): - एक Undirected Graph (अनडाइरेक्टेड ग्राफ) में Edges को उनके साथ Directions से नहीं जोड़ा जाता है। एक Undirected Graph निम्न चित्र में दिखाया गया है क्योंकि इसके किनारों (Edges) को किसी भी दिशा से Attach नहीं किया गया है। यदि एक Edge, Vertex A और B के बीच मौजूद है, तो Vertiex B से A और साथ ही A से B तक Traverse कर सकते हैं।

Directed Graph (डायरेक्टेड ग्राफ): - एक Directed Graph (डायरेक्टेड ग्राफ) में Edges एक क्रम (Order) में जोड़ी बनाते है। Edges किसी Vertex A से दूसरे Vertex B पर एक विशिष्ट Path का प्रतिनिधित्व करते हैं। नोड A को प्रारंभिक (Initial) नोड कहा जाता है जबकि Node B को Terminal नोड कहा जाता है। निम्न चित्र में एक Directed Graph दिखाया गया है।



Directed Graph

Terminology of Graph: -

- Path: - Graph G (U, V) प्रारंभिक नोड U से किसी Terminal नोड V तक पहुंचने के लिए नोड्स के अनु-क्रम (Order) के रूप में एक पथ को परिभाषित किया जा सकता है।
- Closed path: - एक Path को closed Path के रूप में कहा जाएगा यदि Initial नोड Terminal नोड के समान है। $V_0 = V_N$ होने पर एक रास्ता बंद हो जाएगा।
- Simple Path: - यदि Graph के सभी नोड्स Exception $V_0 = V_N$ के साथ अलग हैं, तो ऐसे पथ P को Closed Simple Path कहा जाता है।
- Cycle: - एक Cycle को उस पथ के रूप में परिभाषित किया जा सकता है जिसमें पहले और अंतिम Vertices को छोड़कर कोई दोहराया Edges या Vertices नहीं दोहराया जाता हैं।
- Connected Graph: - Connected ग्राफ़ वह है जिसमें V में प्रत्येक दो Vertices (u, v) के बीच कोई Path मौजूद है। Connected ग्राफ़ में कोई Isolated नोड नहीं होता हैं।

- Complete Graph: - एक Complete Graph वह है जिसमें प्रत्येक नोड अन्य सभी नोड्स के साथ जुड़ा हुआ है। एक Complete Graph में n (n-1) / 2 Edges होते हैं जहां Graph में नोड्स की संख्या "n" होती है।
- Weighted Graph: - Weighted Graph में, प्रत्येक Edge को कुछ डेटा जैसे Length या Weight के साथ Assign किया जाता है। Edge E का भार W(E) के रूप में दिया जा सकता है।
- Digraph: - Digraph एक Directed Graph है जिसमें Graph का प्रत्येक Edge किसी न किसी Direction से जुड़ा होता है और Traversing केवल निर्दिष्ट दिशा में ही किया जा सकता है।
- Loop: - एक Edge जो समान End Points के साथ जुड़ा हुआ है उसे Loop कहा जा सकता है।
- Adjacent Nodes: यदि दो Nodes u और v एक किनारे E के माध्यम से जुड़े हुए हैं, तो नोड्स u और v को पड़ोसी (Neighbours) या Adjacent नोड कहा जाता है।
- Degree of Node: - नोड की Degree, Edges की संख्या है जो उस नोड के साथ जुड़े हुए है। Degree 0 वाले नोड को Isolated नोड कहा जाता है।

Representation of Graph in Memory

GraphRepresentation से तात्पर्य उस तकनीक से है जिसका उपयोग कंप्यूटर की मेमोरी में कुछ Graph को स्टोर करने के लिए किया जाना है। Graph को Memory मे Representation करने के लिए निम्नलिखित तरीके है: -

(i) Adjacency Matrix (अडजैकेन्सी मैट्रिक्स)

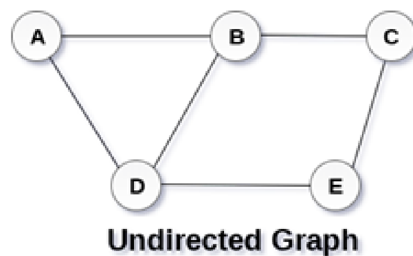
(ii) Indecency Matrix (इन्डेन्सी मैट्रिक्स)

(iii) Adjacency List Representation (अडजैकेन्सी लिस्ट रिप्रजेंटेशन)

i. Adjacency Matrix (अडजैकेन्सी मैट्रिक्स): -

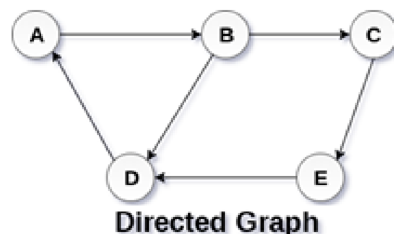
इस प्रकार के Representation मे एक Vertex का दूसरी Vertex से संबंध (Relation) को एक Matrix के द्वारा प्रस्तुत करते है। Adjacency Matrix में, Rows और Columns को Graph Vertices द्वारा दर्शाया जाता है। एक Graph जिसमें "n" Vertices हैं, की Dimension $n_x n$ होगी। यहां उन Vertices के नीचे 1 लिखा है जो सामने लिखी Vertex की Adjacent है । शेष के नीचे 0 लिखा जाता है। निम्नांकित चित्र मे दर्शाये गए Graph का Adjacency Matrix Representation चित्र के नीचे दर्शाया गया है।

एक Undirected Graph G के Adjacency Matrix में एक Entry Mij, 1 होगी अगर वहाँ Vi और Vj के बीच एक Edge मौजूद होगा।



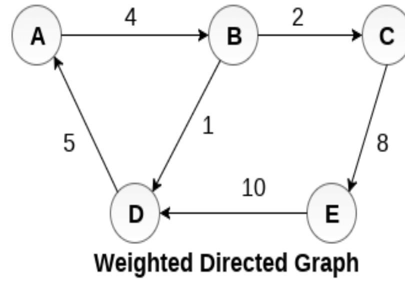
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Directed Graph में, एक Entry Iij केवल 1 होगी जब Vi से Vj तक Directed एक Edge होगा।



	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

Weighted Directed Graph का प्रतिनिधित्व अलग है। 1 से Entry भरने के बजाय, Adjacency Matrix के Non-Zero Entries को संबंधित Edges के Wight से दर्शाया जाता है।



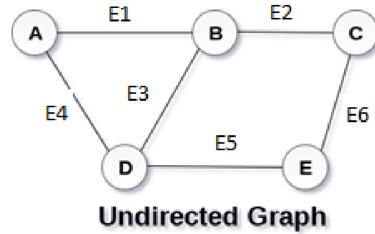
Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	0	5	0	0	0
E	0	0	0	10	0

ii. **<u>Indecency Matrix (इन्डेन्सी मैट्रिक्स): -</u>**

इस प्रकार के प्रस्तुतीकरण (Submission) मे Vertex का विभिन्न Edges से Relation को एक Matrix के द्वारा प्रस्तुत करते है। निम्न चित्र मे दर्शाये गए Graph का Indecency Matrix Representation को नीचे दर्शाया गया है: -

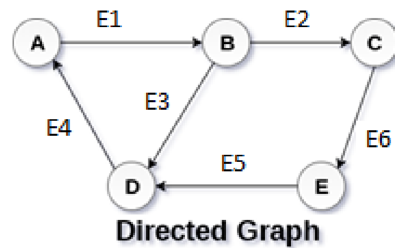
यहां हमने उन Vertices के आगे 1 लिखा है जो ऊपर दी गई Edge के Indecent है अर्थात जुड़ रहे है।



Undirected Graph

	E1	E2	E3	E4	E5	E6
A	1	0	0	1	0	0
B	1	1	1	0	0	0
C	0	1	0	0	0	1
D	0	0	1	1	1	0
E	0	0	0	0	1	1

यहां हमने उन Vertices के आगे 1 लिखा है जो ऊपर दी गई Edge के Indecent है अर्थात जुड़ रहे है।

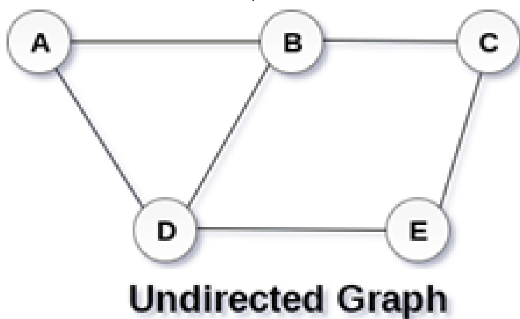


Directed Graph

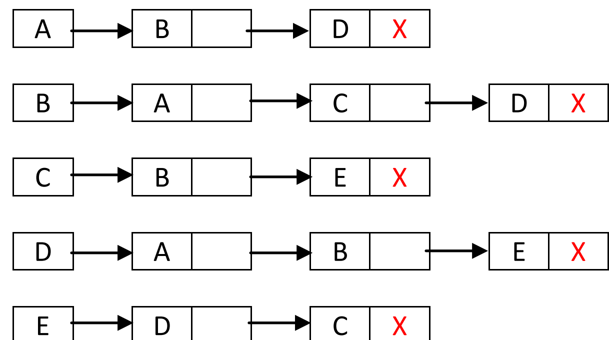
	E1	E2	E3	E4	E5	E6
A	0	0	0	1	0	0
B	1	0	0	0	0	0
C	0	1	0	0	0	0
D	0	0	1	0	1	0
E	0	0	0	0	0	1

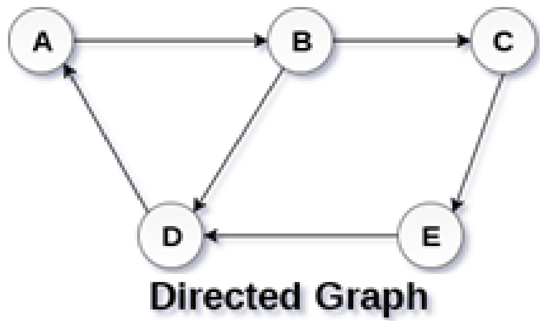
iii. **<u>Adjacency List Representation (अडजैकेन्सी लिस्ट रिप्रजेंटेशन): -</u>**

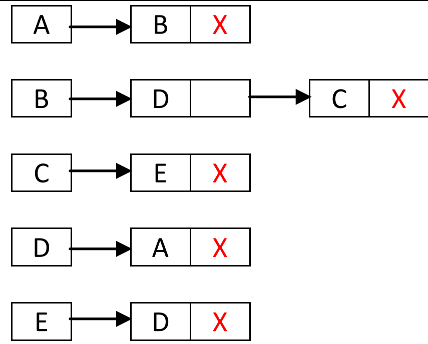
Linked Representation में, एक Adjacency List का उपयोग Graph को कंप्यूटर की मेमोरी में संग्रहीत करने के लिए किया जाता है। ग्राफ़ में मौजूद प्रत्येक नोड के लिए एक Adjacency List को बनाए रखा जाता है, जो नोड Value और संबंधित नोड के बगल के नोड के लिए एक Pointer को संग्रहीत करता है। Adjacency Lists की Length का योग एक Undirected Graph में मौजूद Edges की संख्या के दोगुने के बराबर है।

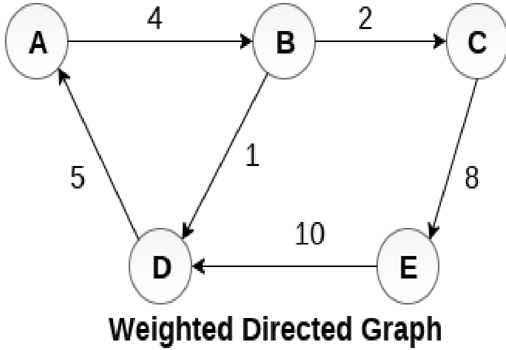


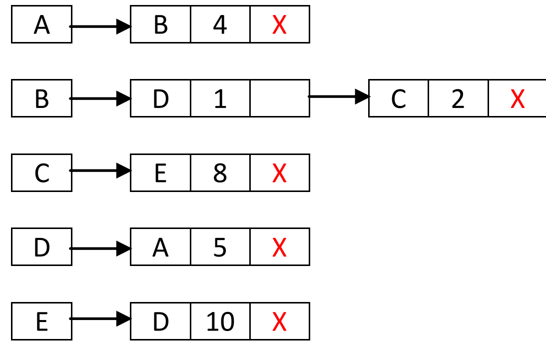
Undirected Graph











TREE (ट्री): -

- Tree एक Non-Linear डेटा स्ट्रक्चर होता है। Tree के प्रत्येक Data Item को हम Node कहते है। "ट्री (Tree), Nodes का एक समूह होता है जिनमें सामान्यतय: Hierarchical Relationship होती है।"
- Tree में Parent-Child Relationship होती है। Tree में जो सबसे ऊपर वाला Node होता है उसे हम Root Node कहते है। एक Node का अधिकतम एक ही Parent हो सकता है। लेकिन केवल Root Node का कोई Parent नही होता है।
- एक Tree में प्रत्येक Node का शून्य या ज्यादा Child Nodes हो सकते है। ऐसे Nodes जिनके एक भी Child Nodes नही होते है उन्हें Leaf Node या Terminal Node कहते है। वैसे तो Tree हमेशा ऊपर की ओर बढ़ता है लेकिन Data Structure का Tree हमेशा नीचे की ओर बढ़ता है।

TREE TERMINOLOGY (ट्री टर्मिनोलॉजी): -

नीचे आपको Tree का चित्र दिया गया है, जिसमें इसके कुछ महत्वपूर्ण Terms दिए गये हैं: -

Root: - एक Tree में Root Node सबसे ऊपर का Node होता है, दूसरे शब्दों में कहें तो, Root Node का कोई भी Parent Node नहीं होता है। ऊपर दिए गये चित्र में A एक Root Node है।

Parent Node: - यदि कोई Node किसी Sub Node को Contain करता है तो इस Node को Sub Node का Parent कहा जाता है। दूसरे शब्दों में कहें तो, "एक Node का ठीक पिछला वाला Node, Parent Node होता है।" चित्र में, B जो है वह D, E, F का Parent है।

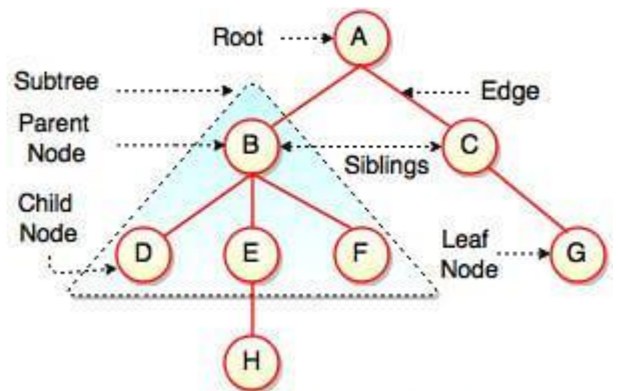


Fig. Structure of Tree

Child Node: - यदि एक नोड किसी नोड का वंशज (Descendant) है, तो इस नोड को Child Node के रूप में जाना जाता है। ऊपर दिए गये चित्र में, D, E, F जो हैं वे B के Child हैं।

Leaf Node: - वह नोड जिसका कोई भी Child नहीं होता है उसे Leaf Node कहते हैं। चित्र में, D, H, F, G Leaf Nodes हैं। Tree में, एक Leaf Node सबसे नीचे वाला नोड होता है। एक Tree में कितने भी Leaf Nodes हो सकते है। Leaf Nodes को External Nodes भी कहते हैं।

Edge: - एक Edge दो Nodes के मध्य का Connection होता है। यह दो Nodes के बीच की Line होती है। चित्र में, A और B के बीच की Line एक Edge है।

Sibling: - वे Nodes जिनका एक ही Parent होता है उन्हें Siblings कहते हैं। चित्र में, B और C Siblings हैं।

Internal Node: - वह नोड जिसका कम से कम एक Child Node होता है उसे Internal Node कहते हैं।

Path: - निरंतर (Consecutive) किनारों के Sequence को Path कहा जाता है। उपरोक्त चित्र में दिखाए गए Tree में, Node H का मार्ग A → B → E → H है।

Ancestor Node: - नोड का Ancestor किसी भी पूर्ववर्ती (Predecessor) नोड पर Root से उस नोड तक का मार्ग है। Root नोड का कोई पूर्वज (Ancestors) नहीं है। उपरोक्त चित्र में, नोड F के पूर्ववर्ती, B और A हैं।

Degree: - एक नोड की डिग्री बच्चों की संख्या के बराबर है जो एक नोड के पास है। उपरोक्त चित्र में दिखाए नोड B की Degree “3’है। एक Leaf Node की Degree हमेशा “0” होती है जबकि एक Complete Binary Tree में, प्रत्येक नोड की Degree “2” के बराबर होती है।

Level Number: - Tree के प्रत्येक नोड को एक Level Number इस तरह से दी गई है कि प्रत्येक नोड अपने माता-पिता की तुलना में एक Level ऊपर पर मौजूद है। Tree का Root नोड हमेशा Level “0” पर मौजूद होता है।

Properties of Tree (ट्री के गुणधर्म): - Tree के निम्न गुण है: -

- कोई भी Node Tree की Root हो सकती है। Tree मे जुड़ी हर Node का एक गुण होता है।
- Tree की प्रत्येक node को किसी दूसरी Node से जोड़ने के लिए केवल एक ही Path होता है।
- एक Tree जिसमे किसी Root को Identify नहीं किया गया है। वह Free Tree कहलाता है।
- जिस Tree मे “n” No.of Node है उसमे “n-1” Edges होंगे।
- वे Node जिनका कोई Child Node नहीं होता है Leaves या Terminal Node कहलाता है।
- वे Node जिनके पास कम से कम एक Child होता है Non Terminal Node कहलाते है।
- Non-Terminals Node को Internal Node तथा Terminal Node को External Node कहते है।
- एक Level पर उपस्थित सभी Node के लिए Depth और Height का मान वही होता है जो उसके Level का मान होता है।

Application of Tree (ट्री के अनुप्रयोग): - इसका प्रयोग बहुत सारें कार्यों के लिए किया जाता है ।

- Tree एक Non-Linear डेटा स्ट्रक्चर है इसलिए इसका प्रयोग Data को Non-Linear तरीके से Store करने के लिए किया जाता है।
- इसका प्रयोग Data को प्रभावी रूप से Organize (व्यवस्थित) करने के लिए किया जाता है।
- बाइनरी सर्च ट्री का प्रयोग तेजी से Data को Search, Insert, Delete करने के लिए किया जाता है।

- Heap एक ट्री है जिसका प्रयोग Priority Queues को Implement करने के लिए किया जाता है।
- B-Tree का प्रयोग Database में Indexing को Implement करने के लिए किया जाता है।
- इसका प्रयोग Artificial Intelligence में किया जाता है।
- Games को बनाने में इसका प्रयोग किया जाता है।
- Syntax Tree का प्रयोग Compilers में किया जाता है।

TYPES OF TREE (ट्री के प्रकार): -

1. Gerneral Tree.
2. Binary Tree: -
    a. Full Binary Tree.
    b. Complete Binary Tree.
    c. Scewed Binary Tree.
    d. Extended Binary Tree.
3. Binary Search Tree.
4. AVL Tree.
5. B-Tree.
6. Heap Tree.

1. Gerneral Tree (जनरल ट्री): -

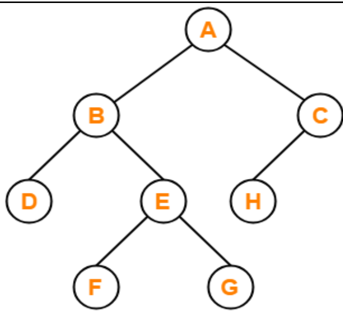
General Tree, Nodes को एक पदानुक्रमित (Hierarchical) क्रम में संग्रहीत करता है जिसमें Top Level Node हमेशा Root Node के रूप में "0" Level पर मौजूद होता है। Root नोड को छोड़कर सभी नोड्स अलग अलग Level की संख्या पर मौजूद होते हैं। समान Level पर मौजूद Nodes को भाई-बहन (Sibling) कहा जाता है जबकि विभिन्न Levels पर मौजूद नोड्स उनके बीच Parent - Child के संबंध को प्रदर्शित करते हैं। एक नोड में किसी भी संख्या में Sub Tree हो सकते हैं।

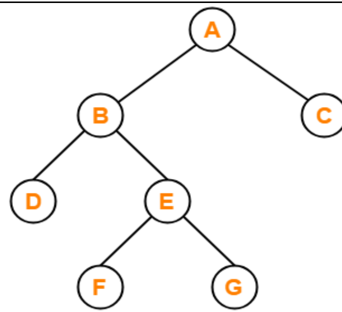
2. Binary Tree (बाइनरी ट्री): -

बाइनरी Tree एक ऐसा Tree हैं जिसमे अधिकतम दो Child हो सकते हैं, Binary Tree Empty भी हो सकता हैं या उस Tree में 1 भी Node हो सकता हैं। बाइनरी Tree में सबसे पहले Node Root Node कहलाता हैं, Root Node के दो और Node होते हैं जो Generally Left Sub Tree और Right Sub Tree होती हैं, Left Sub Tree और Right Sub Tree भी Empty हो सकती हैं, जब Tree में एक भी Node नही होता है उसे हम Empty Tree कहते है। बाइनरी ट्री चार प्रकार चार के होते है।

a. Full Binary Tree: - Full Binary Tree में सभी Nodes की या तो 0 या 2 Child Nodes होती है। आसान शब्दों में कहते तो जिस Binary Tree की सभी Node के या तो Left और Right दोनों Child हो या कोई Child ना हो तो ऐसा Binary Tree Full Binary Tree कहलाता है। Full Binary Tree को Strictly Binary Tree भी कहा जाता है। Full Binary Tree में कोई भी Node खाली नहीं होता हैं, अर्थात सिंगल Node नहीं हो सकता हैं यदि Left Side वाला Node रहेगा तो Right Side Node भी होगा, अगर सिर्फ एक Side Node होगा तो वह Full Binary Tree नहीं होगा।

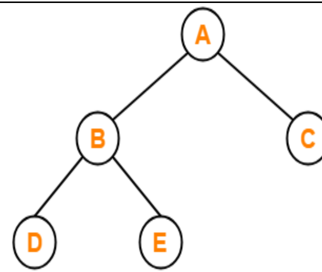
b. Complete Binary Tree: - Complete Binary Tree में सभी Nodes की 2 Child Nodes हो और जिसकी सभी Leaf Nodes Same Level पर हो Complete Binary Tree कहलाता है। Complete Binary Tree में Left Side Tree अगर 2 चाइल्ड हैं तो Right Side के भी 2 Child होना जरुरी हैं, अर्थात Leaf Nodes Same Level में होने चाहिए, अगर Same Level नहीं होगा तो वह Complete Binary Tree नहीं होगा।

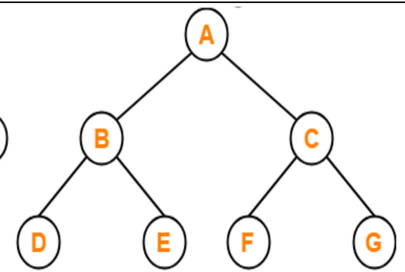




X ✓

Full Binary Tree



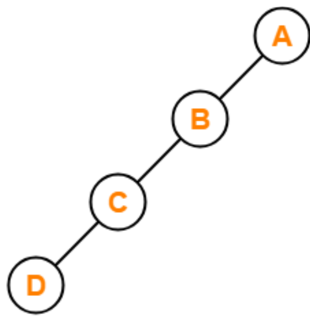


X ✓

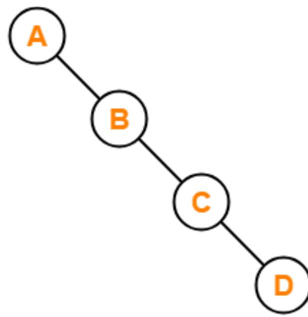
Complete Binary Tree

c. Skewed Binary Tree: - Skewed Binary Tree में सभी Nodes के या तो Left या Right Child Nodes होती है। Skewed Binary Tree में Left Side Child के केवल Left Side Node होते हैं और Right Side Child के केवल Right Side Node होते हैं।

d. Extended Binary Tree: - एक Binary Tree Extended Binary Tree तब कहलाता है जब उसकी सभी Nodes के या तो 0 या 2 Child Nodes हो। Extended Binary Tree में Internal Nodes को Circle द्वारा और External Nodes को Rectangle द्वारा दर्शाया जाता है। जिन Nodes की 2 child Nodes होती है वे Internal Nodes कहलाती है और जिन Nodes की 0 Child Nodes होती है वे External Nodes कहलाती है।

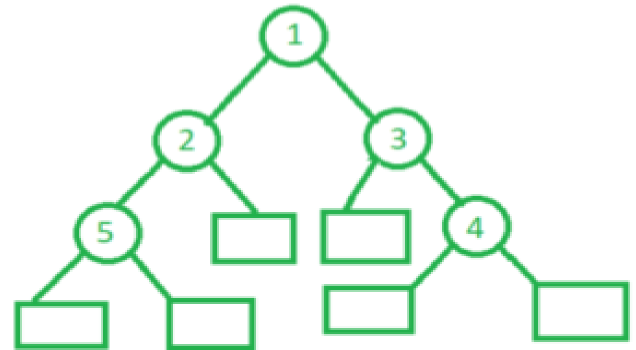


Left Skewed Binary Tree



Right Skewed Binary Tree

Skewed Binary Tree



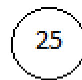
Extended Binary Tree

3. **BINARY SEARCH TREE BST (बाइनरी सर्च ट्री): -**

Binary Search Tree एक विशेष प्रकार का Binary Tree है जो Class के रूप में परिभाषित किया जा सकता है। जिसमें Nodes को एक विशिष्ट क्रम में व्यवस्थित किया जाता है। Left Sub Tree की सारी Information Root से छोटी होती है और Right Sub Tree की सारी Information Root से बड़ी होती है। Right Sub Tree में सभी नोड्स की Value, Root की Value से अधिक या बराबर है। Left और Right Subtree दोनों ही Sub Binary Search Tree होते है। Insertion of BST Example: -

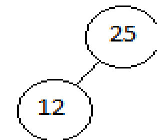
25 12 18 11 27 34 26 09

Insertion in BST: -

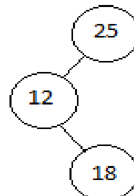


Insert 12 As Root

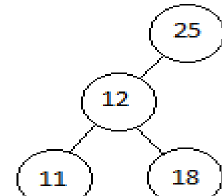
Step 01



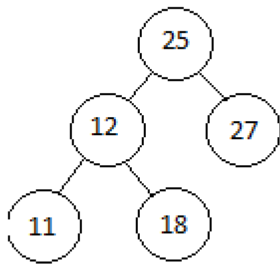
12 < 25 So
Insert 12 in Left of 25
Step 02



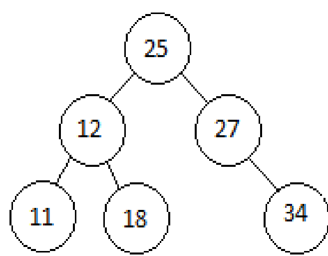
18 < 25 and 18 > 12 So
Insert 18 in Right of 12
Step 03



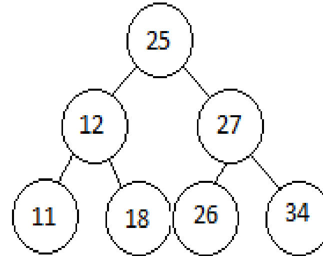
11 < 25 and 11 < 12 So
Insert 11 in Left of 12
Step 04



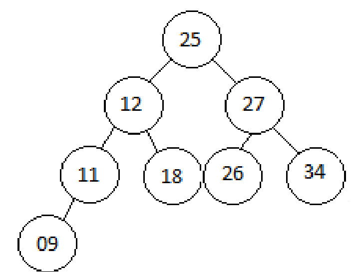
27 > 25 So
Insert 27 in Right of 25
Step 05



34 > 25 and 34 > 27 So
Insert 34 in Right of 27
Step 06

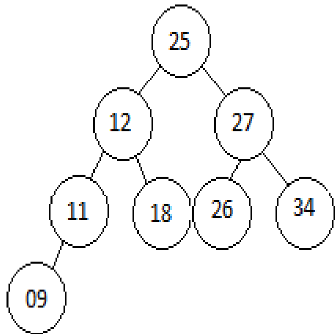


36 > 25 and 26 < 27 So
Insert 26 in Left of 27
Step 07

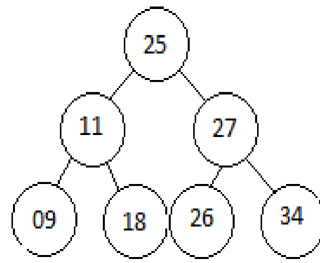


09 < 25, 09 < 12 and 09 < 11
So Insert 09 in Left of 11
Step 08

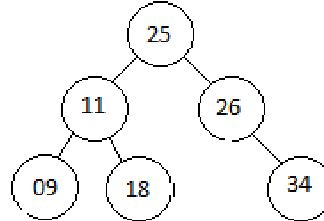
Deletion In BST: -



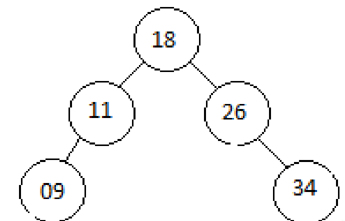
Step 01



Delete 12
Step 02



Delete 27
Step 03



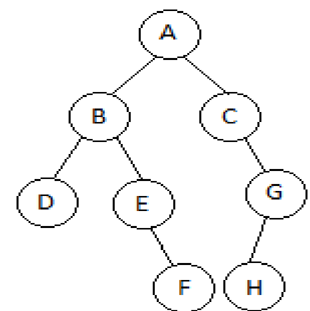
Delete 25
Step 04

Traversal in Binary Tree: -

Tree Traversal का अर्थ हैं हर एक Node को Visit करना। एक Binary Tree को Traverse करते समय उसकी हर Node को सिर्फ एक बार Access किया जाता है और उसके साथ कुछ Operation Perform किया जाता है। एक Binary Tree को 3 प्रकार से Traverse किया जा सकता है।

a. In-order Traversal (इन आर्डर ट्रेवेर्सल).
b. Pre-order Traversal (प्री आर्डर ट्रेवेर्सल).
c. Post-order Traversal (पोस्ट आर्डर ट्रेवेर्सल)

a. In Order Traversal: - In- Order Traversal में सबसे पहले Left Sub-Tree को एक्सेस किया जाता हैं और उसके बाद Root Node को Access किया जाता हैं Last में Right Sub Tree Access / Visit किया जाता हैं । शार्ट में इसे Left – Root – Right Order कहते है। In Order Traversal में Root हमेशा बीच में होता है।

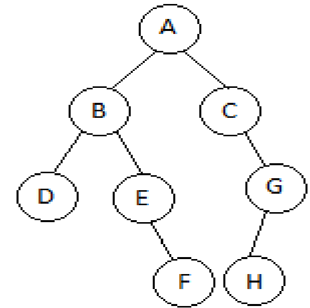


Output:- D → B → E → F → A → C → G → H

- सबसे पहले हम Left Subtree को Access करेंगे तो Left Side में D हैं और फिर D के Left में कोई भी Child नहीं हैं।
- अब हम Root Node को Access करेंगे तो Root Node B हैं उसके बाद हम Right Subtree अर्थात E को एक्सेस करेंगे, और फिर E के Left में कोई भी Child नहीं हैं, तो अब हम Right Child F को Access करेंगे। यहाँ B हमारा Root Node की तरह वर्क किया हैं।
- अब हम A को Root Mode मानते हैं तो हमारा Left Side का Subtree Access हो गया अब हम Root Node A को Access करेंगे उसके बाद Right Side के Subtree को Access करेंगे।

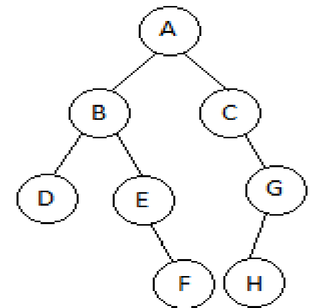
- Right Side में भी एक Root Node C हैं उस Root Node के Left में कोई भी Child नहीं हैं और उसके बाद Right Subtree Access करेंगे।
- अब हम Root Node G के Left Subtree H को Access करेंगे उसके बाद Root Node G को Access करेंगे ।

b. Pre Order Traversal: - Pre Order Traversal में सबसे पहले Root Node को Visit किया जाता हैं उसके बाद Left Subtree को Access किया जाता हैं उसके बाद Right Subtree को Access किया जाता हैं। शार्ट में इसे Root – Left – Right Order कहते है। Pre Order Traversal में Root हमेशा Left में होता है।



Output:- A → B → D → E → F → C → G → H

c. Post Order Traversal: - Post Order Traversal में सबसे पहले Left Subtree को Visit किया जाता हैं उसके बाद Right Subtree को Access किया जाता हैं उसके बाद Root Node को Access किया जाता हैं। शार्ट में इसे Left – Right – Root Order कहते है। PosT Order Traversal में Root हमेशा Right में होता है।



Output:- D → F → E → B → H → G → C → A