UNIT – I

Data Structure

किसी भी **Program** में **Data** को विभिन्न प्रकार से **Arrange** किया जा सकता हैं। किसी भी एक **Representation** का **Logical** या **Arithmetical Representation** ही **Data Structure** कलाता हैं। किसी **Data Representation** को **Choose** करना दो **concepts** पर निर्भर करता हैं।

(1) Data इतना **Arranged** होना चाहिए कि वह **Actual Relation** को प्रदर्शित करता हैं।

(2) वह इतना **Easy** होना चाहिए कि आवश्यकता पड़ने पर उसे आसानी से समझा जा सकता हैं।

(3) Data Structure को मुख्यतः दो **Parts** में **Divide** किया गया हैं।

(i) Linear Data Structure: - यह एक ऐसा **data structure** हैं जिसमें **Elements Sequence** में **Store** रहते हैं तथा हर **Element** एक **Unique Pre-decessor** और **Success** होता हैं। **Linear Data Structure** के उदाहरण निम्न हैं। **Arrays, Linked List, Stack** तथा **Queue**

(ii) Non Linear Data Structure: - एक ऐसा **Data Structure** हैं जिसमें **Elements Sequence** में नहीं रहते हैं। और इसके लिए किसी **Unique Pre-decessor** या **Successor** की आवश्यकता नहीं होती हैं। **Example: - Trees, Graphs etc.**

विभिन्न **Data Structures** निम्न हैं।

(1) Arrays: - Array Ltd. No. of Elements तथा **Same Data Type** की एक **List** को **Represent** करते हैं। **Array** को **Individual elements** को **Index** के द्वारा **Access** किया जा सकता हैं। **Array 3** प्रकार के होते हैं।

(i) One Diemensional Array: - इस प्रकार के **Array** में एक **Index** को **Access** करने की आवश्यकता होती हैं।

(ii) Two Diemensional Array: - एक **Individual element** को **Access** करने के लिए दो **Indexes** की आवश्यकता होती हैं। **(Matrix)**

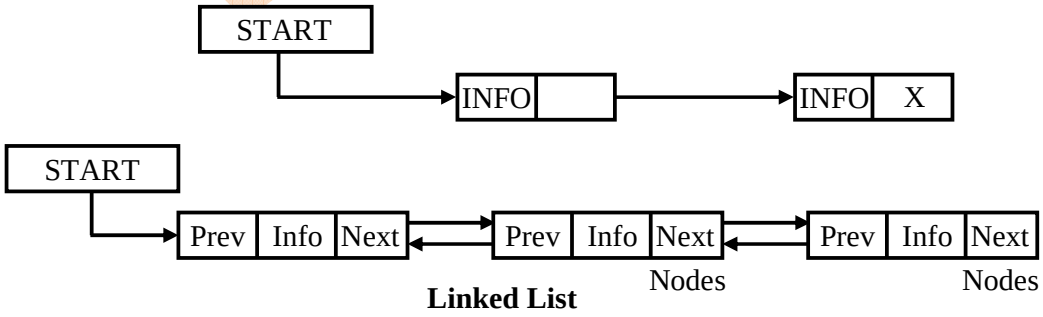
(iii) Multi Diemensional Array: - इसके लिए दो या दो से अधिक **Indexes** को **Store** किया जाता हैं।

(2) Linked List: - Link List Data Elements का एक **Linear Collection** हैं। जिसे **Node** कहते हैं। **Linear Order** को **Pointers** के द्वारा **Maintain** किया जाता हैं। हर **Node** को दो **Parts** में **Divide** करते हैं। एक **Link List Linear Link List** या **Doubly Link List** हो सकती हैं।

(i) Linear Link List में हर **Node** को दो **Parts** में **divide** किया जाता हैं

- **First Part** में **Information** को रखा जाता हैं।
- **Second Part** को **Link Part** या **Next Pointer Field** कहा जाता हैं जिसमें **List** के **next Node** का **Address** रहता हैं।

List की **First Element** को एक **Pointer Point** करता हैं जिसे **Head** कहते हैं। **Linear Link List** को एक **Direction** में **Traverse** किया जाता हैं।



Linked List

(ii) Doubly Link List: - Doubly Link List में हर **Node** को **3 Parts** में **Divide** किया गया हैं।

- **First Part** में **Information** को रखा जाता हैं।
- **Second Part** को **Previous Pointer field** कहा जाता हैं जिसमें **List** के पिछले **(Pre-decessor) Element** का **Address** रहता हैं।
- **Third Part** को **Next Pointer Field** कहा जाता हैं जिसमें **List** के **next element** का **address** रहता हैं।

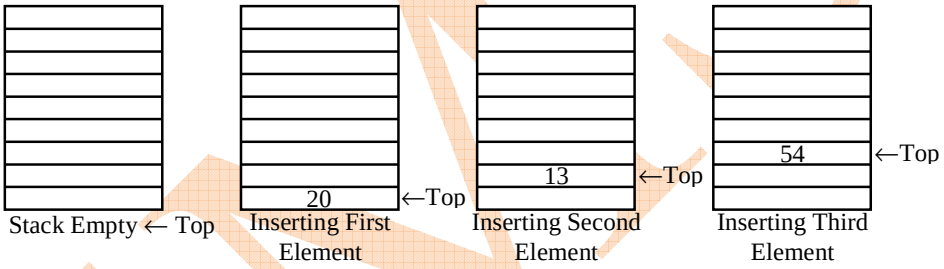
दो **Pointer Variable** को **Perform** किया जाता हैं जिसे **Head** और **Tail** कहते हैं जो **Doubly Link List** के **First** और **Last Element** के **Address** को **Contain** करते हैं। **First Element** के **Previous Pointer** में **NULL Store** किया जाता हैं जिससे यह पता लगता हैं कि इसके पीछे कोई **Node** नहीं हैं। **Last Element** के **Next Pointer** में **NULL** को **Store** किया जाता हैं। जिसे यह पता लगता हैं कि इसके आगे कोई **Node** नहीं हैं। **Doubly Link List** को दोनो **Direction** में **Traverse** किया जा सकता हैं।

Stacks: - Stack को **Last In First Out (LIFO) System** भी कहा जाता हैं। यह एक **Linear Link List** हैं जिसमें **Insertion** और **Deletion** एक ही **End** से किया जाता हैं। जिसे **Top** कहते हैं। **Stack** को बनाने और मिटाने के लिए केवल दो ही **Function** है

(1) Push: - Stack में किसी भी सूचना को जोड़ने के लिए

(2) Pop: - Stack से किसी भी **Item** को मिटाने के लिए।

Stack एक प्रकार की **List** हैं जिसे दो प्रकार से प्रदर्शित किया जा सकता हैं **(1) Array** के द्वारा और **(2) Link List** के द्वारा



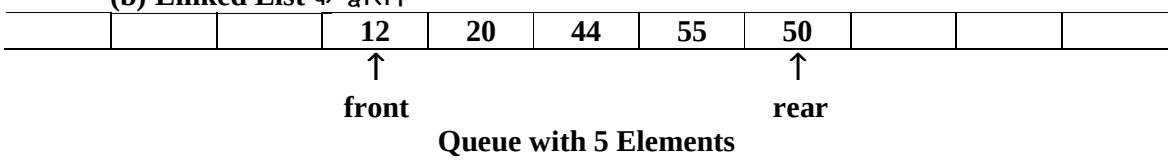
Queues: - Queue को **First In Frist Out (FIFO) System** कहा जाता हैं। **queues** एक विशेष प्रकार की **Data Structure** हैं जिसमें **Stack** की तरह ही **Information** को **Add** करने और **Delete** करने की प्रक्रिया नहीं होती बल्कि इसमें एक स्थान से **Information** को **Add** किया जाता हैं और दूसरे स्थान से **Information** को **Delete** किया जाता हैं। जिस जगह पर **Information** को **Queue** में जोड़ा जाता हैं वह **Position Rear (Queue** का अन्त) कहलाती हैं। जिस स्थान से **queue** में से **Information** को **Delete** किया जाता हैं। वो **Front** कहलाता हैं। **Front** और **Rear** से जुड़ी कुछ **Information** नहीं हैं।

(i) **Front** और **Rear** केवल दो ही **Condition** में बराबर होते हैं।
- **(a)** जब **Queue** खाली हो यानी **Front = 0** और **Rear = -1**
- **(b)** जब **Queue** में केवल एक ही **element** हो अतः **front = 0** और **Rear = 0**

(ii) **Queue** खाली होगी जब **Front** तथा **Rear** दोनों ही **–1** को प्रदर्शित करें।

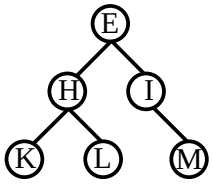
(iii) **Queue** पूरी भरी हुई होगी जब
- **(a) Queue** का **Front 0** तथा **Rear Max –1** को प्रदर्शित करेगा
- **(b)** जब **Queue** का **Front = Rear + 1**

(iv) **Queue** को भी दो प्रकार से बनाया जा सकता हैं।
- **(a) Array** के द्वारा।
- **(b) Linked List** के द्वारा।



Queue with 5 Elements

Trees: - Tree एक **Important Concept** है **Tree** में **Stored Information** को **Node** कहते हैं। **Tree** के सभी **Nodes Edges** के द्वारा जुड़े होते हैं। **Tree** एक ऐसा **Data Structure** हैं जिसे विभिन्न **Hierarchical Relations** हैं के द्वारा प्रदर्शित किया जाता हैं। **Tree** का निर्माण उसके **Root** से किया जाता हैं। **tree** के प्रत्येक **Node** के उपर **(Root** को छोड़कर) एक **Node** होती हैं। जिसे **Parent Node** कहा जाता हैं। **Node** के सीधे नीचे वाली **Node** को **child node** कहा जाता हैं। एक **tree vertices** और **Edges** का **Collection** हैं जो निश्चित आवश्यकताओं को **Satisfy** करता हैं।



Vertex: - Vertex एक **Object** या **Node** है जो नाम या अन्य **Relative Information** को **Store** करता हैं।
Edge: - जब दो **Vertices** के बीच **Connection Establish** किया जाता हैं तो उसे **H** कहते हैं।

Graph: - कभी – कभी **Data Elements** के मध्य **relationship** को प्रदर्शित किया जाता हैं जो प्रकृति से **Hierarchical** हो यह आवश्यक नहीं अतः ऐसा **Data Strcutre** जो एक विशेष **Relationship** को प्रदर्शित करता हैं। उसे **Graph Data Structure** कहा जाता हैं। **Graph** के अंतर्गत **set of notes** तथा **set of edges** को रखा जाता हैं। **Graph** को **2** भागों में बांटा गया हैं।
(1) Direct Graph **(2) In Direct Graph**

DATA STRUCTURE OPERATION
किसी भी **Data Structure** पर **different operation** को **process** किया जा सकता हैं। एक **special data structure** जो दी हुई **condition** के लिए चुना जाता हैं। वह **operation perform** करने की संख्या पर निर्भर करता हैं। **Normally** किसी भी **data structure** को **5 operations** को **perform** किया जाता हैं।

1. **Traversing : -** जब एक बार **data structure** तैयार हो जाता हैं तो कभी–कभी प्रत्येक **element** को **Access** करने की आवश्यकता होती हैं। इस प्रकार के **Concept** को **Traversing** कहा जाता हैं।
2. **Searching: -** जब कभी दी गई **key value** के साथ **Record** की या **Node Value** को **Locate** करना हो तो इसके लिए भी **Link list** पर **Searching Operation Perform** किया जाता हैं।
3. **Insertion: -** जब किसी भी नये **Node** को **List** में **Add** करना हो तो उसे **List** में **Item Insert** करना कहा जाता हैं।
4. **Deletion: -** जब **Structure** में से किसी **Node** को **delete** करना हो तो इसे **List Deletion Operation** कहा जाता हैं।
5. **Updation: -** जब किसी **Structure** में किसी **Existing Record** की **Value** को **Change** करना हो तो उसके लिए **List** में **Updation Operation** को **Perform** किया जाता हैं।

कभी–कभी एक से अधिक **Operation** को उपयोग में लेने की आवश्यकता होती हैं।

1. **Sorting: -** इस तकनीक के द्वारा **Records** को **Arrange** किया जाता हैं। ये **Records Logically Arrange** होते हैं। **Sorting Ascending** (आरोही) या **descending** (अवरोही) **Order** में होती हैं।
2. **Merging: -** जब अलग–अलग **Records** को **2** फाइलों से किसी एक **file** में लाना हो तो इसके लिए **List Merging Operation** को **Perform** किया जाता हैं।

ADT (Abstract Data Type) or (Abstract Data Structure)
किसी भी **Data Type** की **Logical Properties** को **Specify** करने के लिए **Abstract Data Type** को उपयोग में लिया जाता हैं। **Data Type** बहुत सी **Values** तथा **Operations** का **Collection** हैं। जिससे **Hardware** तथा **Software Data Structure** द्वारा **Implement** किया जाता हैं। **Abstract Data**

Type किसी भी **Data Type** के **Mathematical Concept** को **define** करते हैं। जब **Abstract Data Type** के **Mathematical Concept** को **Define** किया जाता हैं तो **Time** व **space** पर ध्यान नहीं दिया जाता हैं। **ADT** को **Specify** करने के लिए के बहुत से **Methods** हैं। जिसके द्वारा इनका **Implementation** किया जाता हैं। मुख्यतः **ADT** को **2 Parts** में बांटा जाता हैं।

(1) Value Definition
(2) Operator Definition

Value Definition collection of Values को **ADT** के लिए **Define** करती हैं। जिसके **2 Parts** होते हैं।

(1) Definition
(2) Condition

किसी भी **Operator** को **Define** करने के लिए **3 Parts** की आवश्यकता होती हैं।

(1) Header
(2) Optional Pre-condition
(3) Post Conditions

Abstract Typedef के द्वारा **Value Definition** को **Define** किया जाता हैं तथा **Condition Keyword** के द्वारा किसी भी **Condition** को **Specify** करते हुए हमेशा **Value Definition** के बाद **Operator Definition** को **Define** किया जाता हैं।

ADT For Strings

ADT Specification को **Strings** पर भी **Perform** किया जाता हैं। **4 Basic Operation** को **Strings Support** करते हैं।

(1) Length: - यह **Function Current Strings** की **Length** को **Return** करता हैं।
(2) Concat: - यह **Function 2 Inputted Strings** को जोड़ कर एक **Single String Return** करता हैं।
(3) Substr: - यह **Function** किसी भी **String** का **Substring Return** करता हैं।
(4) Pos: - यह **Function** किसी एक **String** की **First Position** को **Return** करता हैं।

Arrays को भी **Abstract Data Type** की तरह **Represent** किया जाता हैं। **Arrays** को **Normally 3 Parts** में **Define** किया जा सकता हैं।

(1) One dimensional Array (1-D Array)
(2) Two dimensional Array (2-D Array)
(3) Multi dimensional Array

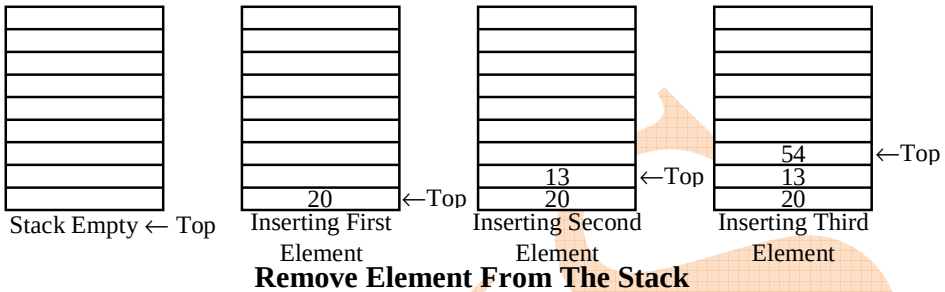
Analysis Of Algorithm

कोई भी **Program Data Structure** तथा **Algorithm** दोनो का **Combination** होता हैं। **Algorithm** किसी भी **specific Problem** को **Solve** करने के लिए विभिन्न **Steps** का **combination** होता हैं। **Algorithm** को किसी भी **Problem** को **Read** करने व उसकी **Solution** को **define** करने के लिए **Use** में लिया जाता हैं। यह **Operations** जिन्हें **Perform** किया जायेगा साथ ही **Example** के साथ उस **Problem** के **Solution** को **contain** करता हैं। **Algorithm** के द्वारा **Time** और **Space Complexity** को **Check** किया जाता हैं। **Algorithm** के **Complexity function** होती हैं जो **Run time** या **Input size** को **consider** करती हैं।

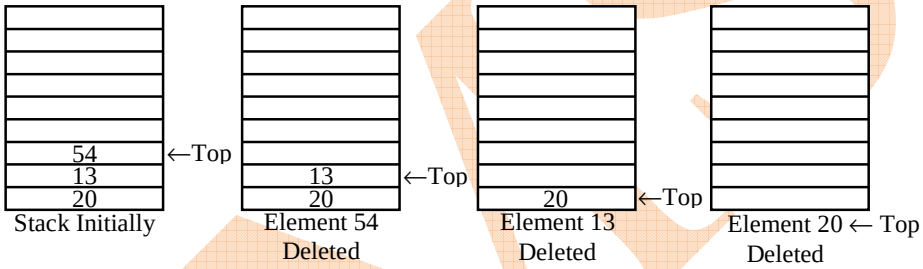
Stack

Stack एक **Non primitive data Structure** हैं यह एक **Sequence List** में **Data** को **Add** करता हैं और वहीं से **Data Item** को हटाया जाता हैं। **Stack** में **Single Entry** और **Single Exit Point** होता हैं। **Stack Lifo concepts (Last in First out)** पर कार्य करती हैं।

जब किसी **Data Item** को **Stack** में **Insert** करना हो तो इसके लिए जिस **Operation** को **Perform** किया जाता हैं उसे **Push Operation** कहते हैं। **Item** को **Remove** करने के लिए **Pop Operation** को **Perform** किया जाता हैं। इसके अन्तर्गत एक **Pointer Stack** के **Last** में डाले गये **Item** को **Point** करता हैं। जिससे **Top** कहा जाता हैं। जब **Stack** में कोई नया **Item Add** किया जाता हैं तो उसका **Top** बढ़ता हैं और जब किसी **Item** को **remove** किया जाता हैं तो **Top** घटता हैं। Insert Element In The Stack



Remove Element From The Stack



Stack Operation

Stack पर मुख्यतः **2** प्रकार के **Operations** को **Perform** किया जाता हैं।

(1) **Push Operation: -Top** पर कोइ भी नया **Element** डालना या जोड़ना **Push Operation** कहलाता हैं। जब भी **Push Operation** होता हैं, **Top** में एक संख्या डाली जाती हैं और **Top** एक **Level** बढ़ता हैं। ऐसा करते करते एक **Level** पर आकर वह पूरा भर जाता हैं तो वह स्थिति **STACK-FULL** स्थिति कहलाती हैं।

(2) **Pop Operation: - Top** पर से **Element** को **Delete** करने की क्रिया को **Pop Operation** कहते हैं। हर **Pop Operation** के बाद **Stack** एक **Level** कम हो जाता हैं। सारे **Element** खत्म होने के बाद **Stack Under Flow** स्थिति में पहुँच जाता हैं।

Stack Implementation With Array

सबसे पहले **Stack Array** को निम्न प्रकार **Define** किया जायेगा।

(1) **Define the Maximum Size of the Stack Variable**

(1) एक **Int Stack Variable** को **Array define** कर **Maximum Size Declare** करना।

```
#define MAXSIZE 10
int stack [MAXSIZE];
```

(2) एक **int Top** को **–1** से **Initialized** किया जायेगा।

(2) **Define the Integer Variable Top and Initialized it with -1**

Algorithm for Push Operation

Let Stack[Max size] is an array for implementing stack.

1. **[Enter for stack overflow ?]**
if Top = MAXSIZE –1, then : print : overflow and exit.

2. Set Top = Top+1[Increase Top by 1]
3. Set STACK[TOP] : = ITEM [Insert new element]
4. Exit.

Function for Push Operation

```
void push( )
{       int item;
        if (top= =MAXSIZE-1)
        {       printf("Stack is full");
                getch( );
                exit(0);
        }
        else
        {
                prinf("Enter the element to be inserted");
                scanf("%d", & item);
                top=top+1;
                stack[top] = item;
        }
}
```

Algorithm for Deleting an Element from the Stack

1. [Check for the Stack under flow]
   if Top<0, then
   printf("Stack under flow") and exit
   else [Remove the Top element]
   Set item=Stack [Top]
2. Decrement the stack top
   set Top = Top-1
3. Return the deleted item from the stack.
4. Exit.

Function for Pop Operation

```
int pop( )
{
        if(top = = -1)
        {       printf ("The stack is Empty")
                getch( );
                exit(0);
        }
        else
        {       item = stack[top];
                top = top-1;
        }
        return(item);
}
```

Stack Implementation with Link List

Algorith For Push Function

Step1- struct node n = (struct node) malloc (size of (struct node))

(malloc Function के द्वारा एक struct node प्रकार के Pointer n को Memory दिलाएं।)

Step2- set n → info = data (अर्थात् n की info में data को सुरक्षित करें।)
Step3- n → next = top (n के next क्षेत्र में top का address (पता) भेजें।)
Step4- set top = n (top को n पर भेजें।)
Step5- Exit

Function to Insert an Item in the Stack

```
void push (stack **top, int value)
{
        stack *ptr;
        ptr = (stack*) malloc (sizeof(stack));
        if (ptr = = NULL)
        {
        printf("\nUnable to allocate to memory for new node...");
        printf("\nPress any key to exit ...");
        getch( );
        return;
        }
ptr->info = value;
ptr->next = *top;
*top = ptr;
}
```

Algorithm for Pop

Step1- set int data (एक data नामक Variable को intialize कराएं।)
Step2- set struct *t = top (Stack के टॉप पर struct node प्रकार के Varialbe t को भेजें।)
Step3- check whether top = NULL (जांचें कि क्या top, NULL पर है।)
if yes then (यदि हाँ तो
Process step 4 to 5 4 से 5 तक के step को करें
else अन्यथा
process step 6 to 9 6 से 9 तक के step को करें।)
Step4- print ("underflow") (underflow को प्रिन्ट कराएं।)
Step5- Exit (0) (Main() Function से बाहर आ जाएं।)
Step6- set data = t → info (t के info क्षेत्र में सुरक्षित सूचना को डाटा में सुरक्षित कराएं।)
Step7- set top = top → next (top को उसके अगली Node पर भेजें।)
Step8- free (t) (t में सुरक्षित Memory को मुक्त कराएं।
Step9- return(data) (data में सुरक्षित सूचना को main( ) Function में भेजें।
Step10- Exit

Function to Delete an Item From the Stack

```
int pop (stack **top)
{
        int temp;
        stack *ptr;
        temp = (*top)->info;
        ptr = *top;
        *top = (*top)->next;
```

```
        free(ptr);
        return temp;
}
```

MULTIPLE STACKS

यदि **Stack** को **Array** के द्वारा **Implement** किया जाता हैं तो **Stack** में **Sufficient Space** को **Allocate** किया जाना चाहिए यदि **Array** के द्वारा **allocate Space** कम होगी तो **Overflow Condition Occur** हो जायेगी इसे दूर करने के लिए **Array** में और **Memory** को **allocate** करना होगा। यदि **Number of Overflow** को कम करना हैं तो इसके लिए **Large Space Of Array** को **Allocate** करना होगा जिससे **Memory** के **waste** होने की सम्भावना भी बढ़ जाती हैं।
इस प्रकार की **Condition** के लिए एक से ज्यादा **Stack** को एक **Array** में बनाया जा सकता हैं।

Representing Two Stack: -
एक ही **Array** में **2 Stack** को **Implement** किया जा सकता हैं। इन दोनो **Stack** की **Size 0** से **N-1** तक होगी जब दोनो **Stacks** को **Combine** किया जायेगा तो इनकी **Size N (Array Size)** से ज्यादा
नहीं होगी।

0	1	2	⇢	⇠	n-3	n-2	n-1

Stack-A **Stack - B**
Stack-A Left से **right** की तरफ बढ़ती है तथा **Stack-B Right** से **Left** की तरफ बढ़ती हैं।

Representing More Than Two Stack: -
एक ही **Array** में **2** से ज्यादा **Stack** को भी **Implement** किया जा सकता हैं। हर **Stack** को **Array** में **Equal Space** दिया जाता हैं और **Array Indexes** को **Define** कर दिया जाता हैं।
b[1] t[1] b[2] t[2] b[3] t[3] b[4] t[4] b[5] t[5] b[6] t[6]

→	→	→	→	→	→

Stack 1 **Stack 2** **Stack 3** **Stack 4** **Stack 5** **Stack 6**

APPLICATION OF STACKS

Stack को **Main Data Structure** की तरह **Use** में लिया जाता हैं। इसकी कुछ **Application** निम्न हैं।
(1) **Stack** को **Function** के बीच **Parameters Pass** करने के लिए **Use** में लिया जाता हैं।
(2) **High level Programming Language** जैसे **Pascal, C** आदि **Stack** को **Recursion** के लिए भी **Use** में लेते हैं।

Stack को बहुत सी **Problems** को **solve** करने के लिए **Use** में लिया जाता हैं। मुख्यतः **3 Notation** में **Stack** को प्रदर्शित किया जाता हैं।
(1) **Prefix Notation**
(2) **Infix Notation**
(3) **Post Fix Notation**

Infix Notation : - Normally किसी भी **Expression** को या **Instruction** को बनाने के लिए **2Values** की आवश्यकता होती हैं।
(1) **Operand**
(2) **Operator**
अतः **Operator** व **Operand** से मिलकर एक **Expression** या **Instruction** बनती हैं। जैसे
A+B यहां पर **A** और **B Operand** हैं तथा **+** एक **Operator** हैं। अतः **Infix Notation** में **Operands** के बीच **Operators** होते हैं।

Precedence of Operator: -

Highest:	**Exponentiation (↑)**
Next highest:	**Multiplication (*) and division (/)**
Lowest:	**Addition (+) and subtraction (-)**

Prefix Notation: - Prefix Notation वह **Notation** हैं जहां पर **Operator Operand** से पहले आता हैं। जैसे **+AB**

Post Fix Notation: - यह वह **Notation** हैं जहां **Operator Operand** के बाद आते हैं जैसे **AB+ Post Fix Notation** को **Suffix Notation** या **Reverse Police Notation** ही कहा जाता हैं।
Conversion of Notation: - यदि किसी **Expression** को **Infix Notation** में लिखा गया है और इन्हें **Solve** करना हैं तो इसके लिए **Bodmass** के **Rule** को **Use** में लिया जाता हैं। जैसे
A+B*C यह **A=4, B=3** और **C = 7**
4+3 x 7 = 7 x 7 = 49
उपरोक्त **Expression** का **Solution** गलत हैं क्योंकि **Bodmass** के हिसाब से पहले * होना चाहिए बाद में **Edition** होना चाहिए

Conversion Into Infix To Postfix

A+B * C (Infix Notation)
A+ (B * C)
A + (BC *) → (B * C को **Postfix Notation** में बदला गया)
अब **BC*** एक **Operand** हैं और **A** एक **Operand** हैं। अतः अब इसे **Postfix Notation** में **Convert** किया जायेगा।
ABC*+ → यह **A+B*C** का **Postfix Notation** हैं।

(1) Example of Postfix Notation: - A + [(B+C)+(D+E)*F]/G के लिये **Postfix Form** प्रदान कीजिये।
Solution: -
सबसे पहले **Bracket** को **solve** किया जायेगा
A+[(BC+)+(DE+)*F]/G
Bracket के अन्दर **Multiplication** की **Precedence High** हैं अतः पहले **Multiplication** को **Solve** किया जायेगा।
A+[(Bc+)+(DE+F*]/G
A+[BC+DE+F*+]/G
Plus व **Division** के बीच **Division** की **Precedence High** हैं अतः
A+BC+DE+F*+G/
ABC+DE+F*+G/+

(2) A+B-C के लिये **Postfix Form** प्रदान कीजिये।
Solution: -
(A+B)-Cx
(AB+)-C
AB+C-

(3) A*B+C/D
यह **Multiplication** की **Precedence High** हैं। अतः सबसे पहले **Multiplication** को **Solve** किया जायेगा।

Solution: -

(A*B)+C/D
(AB*) + C/D
Divide की **Precedence High** हैं।
(AB*) + (C/D)
(AB*) + (CD/)
AB*CD/+ (Post Fix Notation)

(4) (A+B)/(C-D)

वैसे तो **Division** की **Precedence High** हैं परन्तु उससे भी **High Bracket** की **Precedence** होती हैं अतः सबसे पहले **Bracket** को **Solve** किया जायेगा।

(AB+)/(CD-)
AB+CD-/

(5) (A+B)*C/D+E^F/G

यहां सबसे **High Precedence (^) Operator** की हैं। अतः सबसे पहले **Bracket** को और उसके बाद **(^)** को **solve** किया जायेगा

(AB+)*C/D+E^F/G
(AB+)*C/D+EF^/G
AB+C*/D+EF^/G
AB+C*D/+EF^G/
AB+C*D/EF^G/+

Conversion Into Infix To Prefix Notation

Prefix Expression में पहले **Operators** आते हैं तथा बाद में **Operands** को लिखा जाता हैं। **Prefix** को निम्न प्रकार **Solve** करते हैं।

1. A*B+C

Multiplication की **Precedence High** हैं अतः पहले **Multiplication** को **Solve** किया जायेगा।

(A*B)+C
(*AB)+C
+*ABC

2. A/B^C+D

^Operator की **Precedence High** हैं अतः सबसे पहले **^** को **Solve** किया जायेगा।

A/(B^C) + D
A/(^BC) + D
/A^BC + D
+/A^BCD

3. (A-B/C)*(D*E-F)

Left to Right Associativity के **According** सबसे पहले **First Bracket** को **Solve** किया जायेगा। इसके अन्दर भी **Operator Precedence** के **According Solution** किया जायेगा।

(A-(B/C))*((D*E)-F)
(A-/BC)*((D*E)-F)
-A/BC*(*DE-F)
-A/BC*(-*DEF)
***-A/BC-*DEF**

Introduction to Queue

Queue Logical First in First Out (FIFO) तरह की **List** हैं इसे **Normally Use** में लिया जाता हैं। **Queue** को दो तरह से **Implement** किया जा सकता हैं।

(1) Static Implementation
(2) Dynamic Implementation

जिस **Queue** को **Array** के द्वारा **Implement** किया जाता हैं। वह **Static** होती हैं और जिस **Queue** के लिए **Pointers** को **Use** में लेते हैं। उसे **Dynamic Queue** कहा जाता हैं। किसी भी **queue** पर दो तरह के **Operation** को **Perform** किया जाता हैं।

(1) Queue में **Value Insert** करना।
(2) किसी **Element** को **Queue** से **Delete** करना।

किसी भी **Element** को जब **Queue** में **Add** करना हो तो उसके लिए **Queue** के **Last** में **Add** किया जाता हैं। **Queue** के **Last** को **Rear** कहते हैं तथा जहां से **Data Element** को **Remove** किया जाता हैं। वह **Front** कहलाता हैं।

Insert Element In The Queue: -

1. **Initialize front = 0 and rear = -1**
2. **if Rear>= MAXSIZE**
   write Queue Overflow and return
   else;
   Set Rear = rear +1
3. **Queue [Rear] = item ;**
4. **if Front = -1** **[Set the Front Pointer]**
5. **Return.**

Function for Insert an Item in the Queue: -

```
void insert (int item)
{
        if(rear>=maxsize)
        {       printf("Queue overflow");
                break;
        }
else
        {
                Rear = Rear+1;
                Queue [Rear] = item;
        }
}
```

Delete Element From the Queue: -

1. **If front <0** **[Check UnderFlow Condition]**
   Write Queue is Empty and return
2. **else ; item = Queue [front]** **[Remove Element from Front]**
3. **Find new Value of front**
   if (front = = Rear) **[Checking for Empty Qeueu]**
   Set Front = 0 ; rear = -1; **[Re-initialize the Pointer]**
   else
   Front = Front+1;

Function for Delete an Item from the Queue: -

```
void delete( )
{
```

```
        if (Front<0)
        {       printf("Queue is empty");
        break;
        }
else
        {
        item=Queue[Front];
        Front=Front+1;
        printf("item deleted=%d",item);
        }
}
```

Queue Implementation with Link List

Algorithm for Insertion

1. Struct_Queue NEW PTR, *temp [Declare variable type struct_queue]
2. temp=start [Initialize temp]
3. NEW = new node [Allocate memory for new element]
4. NEW→no = value [Insert value into the data filed of element];
5. NEWPTR→NEXT=NULL
6. If (Rear = = NULL) [Queue is empty, and element to be entered is first element]

   Set Front = NEW PTR
   Set Rear = NEW PTR
   else:
   while(temp→next = NULL)
   tem=temp→next
7. Temp = temp→NEWPTR
8. End.

Algorithm for Deletion

1. if(front = = NULL)
   write Queue is empty and exit
   else
   temp = start
   value = temp→no
   Start = Start→next
   free(temp) [End else]
   return(value) [End if]
2. Exit.

Circular Queue

Linear Queue की तरह ही **Circular Queue** में भी **Insertion, Initialization Underflow Condition** की **Testing** आदि को **Perform** किया जाता हैं। अन्य **Operations** कुछ **Different** होते हैं।

Testing of Overflow Condition in Circular Queue: - किसी भी नये **Element** को जब **Circular Queue** में **Insert** करना हो तो उससे पहले **Circular Queue** में **Space Allocate** करना पड़ता हैं और उससे पहले **Circular Queue Full** हैं या नहीं उसे **Check** किया जायेगा। यदि **Queue Full** नहीं हैं तो **Queue** के **Rear** में **Insert Operation Perform** किया जायेगा।

front = 0 and rear = capacity –1

front = rear + 1

अगर इनमें से ये दोनो **Conditions Satisfy** होती हैं तो **Circular Queue Full** होगी।

Function For Checking The Full Condition Of The Queue: -

```
boolean isFull(queue *pq)
{
        if(((pq->front = = 0) && (pq->rear = = CAPACITY-1))
                        ||(pa->front = = pq->rear+1))
                return true;
        else
                return false;
}
```

Circular queue में यदि **Data Item Insert** करना हो और **Circular Queue Full** नहीं हैं तो निम्न **Operations** को **Perform** किया जायेगा।

Insert Item in a Circular Queue

इसके लिए **3 Conditions** को ध्यान में रखा जाता हैं। यदि **Circular Queue full** नहीं हैं तो कौनसी **Condition Occur** होगी वह निम्न हैं : –

(1) यदि **Queue Empty** हैं तो **Front** और **Rear** को **–1** या **0** से **Set** कर दिया जाता हैं।
(2) यदि **Queue Empty** नहीं हैं तो **Rear** की **Value Queue** के **Last Element** को **Pointer** करेंगे और **Rear Increment** कर दिया जायेगा।
(3) यदि **Queue Full** नहीं हैं तो **Rear Variable** को **Capacity –1** के = होगा तथा **Rear Variable** को **0** से **Initialize** कर दिया जायेगा।

```
void enqueue (queue *pa, int value)          {
        /*      adjust rear variable
        if (pq->front = = -1)
                pq->front = pq->rear = 0;
        else if (pq->rear = = CAPACITY-1)
                pq->rear = 0;
        else
                pq->rear++;
        /*      store element at new rear     */
        pq->elements [pq->rear] = value;
}
```

```
void enqueue(queue *pq, int value){
        /*      adjust rear variable  */
        if (pq->front = = -1)
                pq->front = pq->rear = 0;
        else
                pq->rear = (pq->rear + 1) % CAPACITY;
        /*      store element at new rear     */
        pq->elements[pq->rear] = value;
```

DEQUEUE

Dequeue एक प्रकार की **Queue** हैं जिसमें **Elements** को किसी भी **End** से **Remove** और **Add** किया जा सकता हैं परन्तु **Middle** से **Insert** नहीं किया जा सकता। **Dequeue** को **Double Ended Queue** कहा जाता हैं। **Dequeue** दो प्रकार की हैं।

(1) **Input Restricted Dequeue: -** इसमें **Insertion** तो **Rear** से होता हैं परन्तु **Deletion Front** और **Rear** दोनो से किया जा सकता हैं।
(2) **Output Restricted Dequeue: -** इसमें **Insertion Front** और **Rear** दोनो से होता हैं परन्तु **Deletion** सिर्फ **Front** से किया जाता हैं।

PRIORITY QUEUE

Priority Queue वह **Queue** जिसमें हर **element** की **Priority** को **Assign** किया जाता हैं और किस **Order** में **Element** को **Delete** किया जायेगा या **Process** किया जायेगा इसके लिए निम्न **Rules** को **Follow** किया जाता हैं।

(1) **Highest Priority Element** को सबसे पहले **Process** किया जायेगा।
(2) दो या दो से अधिक **Elements** जिनकी **Priority Same** हैं उन्हें **Queue** में वे किस आधार पर **Add** किये गये हैं के द्वारा **Process** किया जायेगा।

Priority Decide करने के लिए कुछ अन्य तरीके भी **Define** किये जा सकते हैं

(1) बड़ी **Jobs** पर छोटी **Jobs** की **Priority High** होती हैं अतः सबसे पहले **Small Jobs** को **Execute** किया जायेगा।
(2) **Amount** के आधार पर ही किसी भी **Job** की **Priority** को **Set** किया जा सकता हैं।
(3) यदि कोई **Job High Priority** के साथ **Queue** में **Enter** होती हैं तो सबसे पहले उसे ही **Execute** किया जायेगा। **Memory** में **Priority queue** को **3** प्रकार से प्रदर्शित किया जा सकता हैं।
    (i) **Linear Link List**
    (ii) **Using Multiple Queue**
    (iii) **Using heap**

Link List Representation: - इस प्रकार की **Queue** को जब **Link List** की तरह प्रदर्शित किया जाता हैं तो **List** के प्रमुख **3 Fields** (हर **Node** के) होते हैं।

(1) **Information Field** जिसमें **Data Elements** को **Store** किया जाता हैं।
(2) **Priority Field** जिसमें **Element** के **Priority Number** को **Store** किया जाता हैं।
(3) **Next Pointer Link** जिसमें **Queue** के **Next Element** का **Address** होता हैं।

Multiple Queue Representation

Multiple Queue या **Priority Queue** को **Ascending Priority Queue** का **Descending Priority Queue** में भी **divide** किया जाता हैं।

(1) **Ascending Priority Queue : -** इस प्रकार की **queue** में सबसे छोटे **Item** की **Priority** सबसे अधिक होती हैं। अतः सबसे छोटे **Item** को सबसे पहले **Delete** किया जायेगा।
(3) **Descending Priority Queue: -** इस प्रकार की **queue** में सबसे बड़े **Item** की **Priority** सबसे अधिक होती हैं। अतः सबसे बड़े **Item** को सबसे पहले **Delete** किया जायेगा।

Heap Representation of Priority Queue

Heap एक **Complete Binary Tree** हैं जिसकी कुछ **Additional Property** होती हैं। जैसे या तो **Root Element Small** होगा यह **Largest** होगा। यदि **Root Element** सबसे छोटा हो उसे **Minimum Heap** कहा जाता हैं। यदि **Root Element Largest** हो तो उसे **Maximum Heap** कहा जायेगा।

Application of Queue

Queue की कुछ **Application** निम्न हैं :–

(1) **Queue** के द्वारा बहुत सी **Problem** को आसानी से **Solve** कर दिया जाता हैं जैसे **BFS (Breadth First Search)** या **DFS (Depth First Search)**
(2) जब **Jobs** को **Network Printer** पर **Submit** किया जाता हैं तो वे **Arrival** के **Order** में **Arrange** होता हैं।

(3) यह एक प्रकार का **Computer Network** हैं जहां पर **Disk** एक **Computer** से **Attach** होती हैं जिसे **File Server** कहा जाता हैं। अन्य **Computers** उन **Files** को **FCFS (First Come First Server)** के आधार पर **access** करते हैं।

(4) Queue Application को **Real Life** में भी **Use** में लिया जाता हैं जैसे **Cinema Hall, Railway Reservation, Bus Stand** आदि।

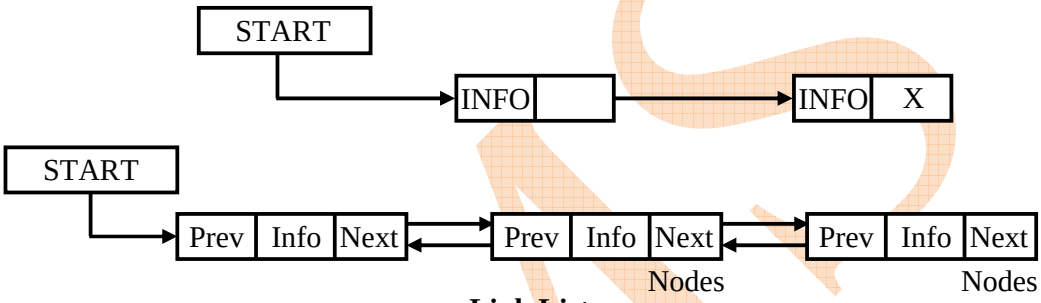
UNIT – II

Link List

यदि किसी **Programs** के **Execution** के पहले **Memory** को **Divide** किया जाता है तो वह **Static** हो जाती है और उसे **Change** नहीं किया जा सकता एक विशेष **Data Structure** जिसे **Link List** कहते हैं। वह **Flexible Storage System** प्रदान करता हैं। जिसको **Array** के प्रयोग की आवश्यकता नहीं होती हैं। **Stack** और **Queue** को **computer memory** में प्रदर्शित करने के लिए हम **Array** का प्रयोग करते हैं। जिससे **Memory** को पहले से **Declare** करना पड़ता है। इससे **Memory** का **Wastage** होता हैं। **Memory** एक महत्वपूर्ण **Resource** हैं अतः इसे कम से कम **Use** में लेकर अधिक से अधिक **Programs** को **Run** करना चाहिए।

Link List एक विशेष प्रकार के **Data Element** की **List** होती हैं जो एक – दूसरे से जुड़ी होती हैं। **Logical Ordering** हर **Element** को अगले **Element** से **Point** करते हुए **Represent** करते हैं। जिसके दो भाग होते हैं।

(1) Information Part
(2) Pointer Part



Link List

Singe Link List में एक **Node** में दो **Parts** होते हैं।

(1) Information Part
(2) Link Part

जबकि **Doubly Link List** में दो **Pointer Part** तथा एक **Information Part** होता हैं।

Operation on Link List

Link List में निम्न **Basic Operation** को **Perform** किया जाता हैं।

(1) Creation : - यह **Operation Link List** बनाने के लिए **Use** में लेते हैं। जब भी आवश्यकता हो किसी भी **Node** को **List** में जोड़ दिया जाता हैं।

(2) Insertion : - यह **Operation Link List** में एक नयी **Node** को विशेष स्थिति में विशेष स्थान पर जोड़ने के लिए काम में लिया जाता हैं। एक नयी **Node** को निम्न स्थानों पर **Insert** किया जा सकता हैं।

- **(i)** **List** के प्रारम्भ में
- **(ii)** **List** के अन्त में
- **(iii)** किसी **List** के मध्य में
- **(iv)** अगर **List** पूरी खाली हैं तब नया **Node Insert** किया जाता हैं।

(3) Deletion: - यह **Operation Link List** से किसी **Node** को **Delete** करने के लिए प्रयोग में लिया जाता हैं। **Node** को जिन तरीकों से **Delete** किया जाता हैं वह निम्न हैं।

- **(i)** **List** के प्रारम्भ में
- **(ii)** **List** के अन्त में
- **(iii)** और **List** के मध्य में

(4) Traversing : - यह प्रक्रिया किसी **Link List** को एक **Point** से दूसरे **Point** तक **Values** को **Print** करवाने के काम आती हैं। अगर पहले **Node** से आखरी **Node** की ओर **Traversing** करते हैं तो यह **Forward Traversing** कहलाती हैं।

(5) Con Catenation: - यह प्रक्रिया दूसरे **List** से **Node** को जोड़ने के काम आती हैं यदि दूसरी **List** पर **Node** होता हैं तो **Concatenation Node** में **M+N Node** होगा

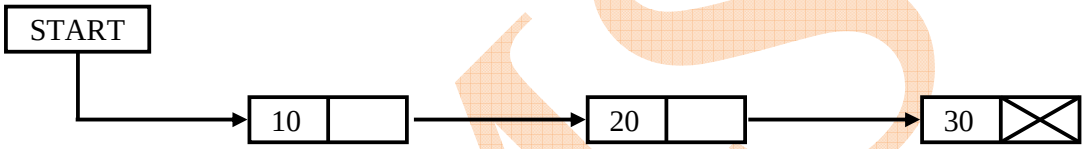
(6) Display: - यह **Operation** प्रत्येक **Node** की सूचना को **Point** करने के प्रयोग में आता हैं। अतः **Information Part** को **Print** करना ही **List Display** कहलाता हैं।

TYPES OF LINK LIST

साधारणतयाः **List** को **4 Part** में बांटा जाता हैं।

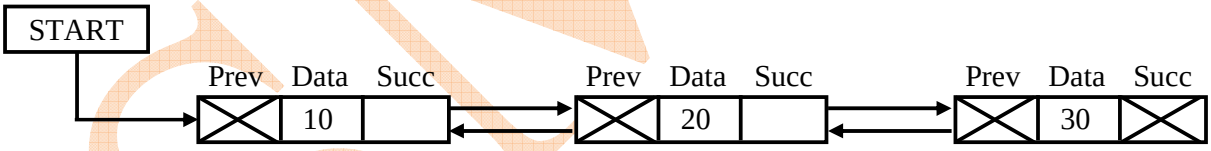
(1) Single Link List
(2) Doubly Link List
(3) Circular Link List
(4) Circular Double Link List

(1) Single Link List : - Single Link List ऐसी **Link List** होती हैं जिसमें सारे **Node** एक दूसरे के साथ क्रम में जुड़े होते हैं इसलिए इसको **Linear Link List** भी कहते हैं। इसका प्रारम्भ अन्त होता हैं। इसमें एक समस्या यह हैं कि **List** के आगे वाले **Node** तो **Access** किया जा सकता हैं परन्तु एक बार **Pointer** के आगे जाने के बाद पीछे वाले **Node** को **Access** नहीं किया जा सकता हैं।



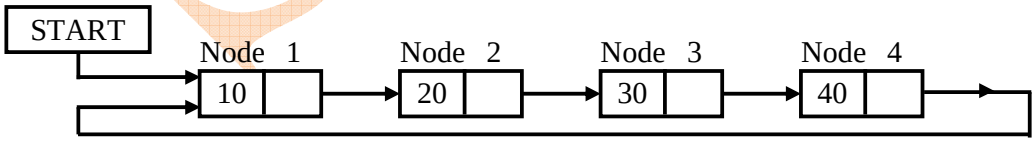
Single Link List

(2) Doubly Link List: - Doubly Link List वह **Link List** होती हैं जिसमें सारे **Node** एक दूसरे के साथ **Multiple Link** के द्वारा जुड़े होते हैं। इसके अन्तर्गत **List** के आगे वाले **Node** व **List** के पीछे वाले **Node** दोनो को **Access** किया जा सकता हैं। इसीलिए **Doubly Link List** में पहले **Node** का **Left** अगले **Node** के **Right** का **Address** रखता हैं।



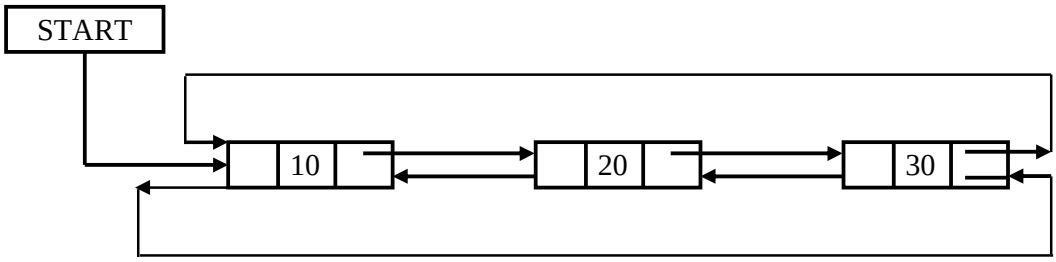
Doubly Link List

(3) Circular Link List: - Circular Link List ऐसी **Link List** होती हैं जिसकी कोई शुरूआत व अन्त नहीं होता हैं। एक **Single Link List** के पहले **Node** से अन्तिम **Node** को जोड़ दिया जाता हैं। अतः **List** के **Last Node** में **List** के **First Node** का **Address** रखा जाता हैं।



Circular Link List

(4) Circular Doubly Link List: - यह एक ऐसी **List** होती हैं जिसमें दोनो **Pointer (1) Successor Pointer (2) Pre-Decessor Pointer Circle Shape** में होते हैं।



Circular Doubly Link List

HEADER NODE

कभी–कभी एक **Extra Node** को **List** के **Front** में रखना आवश्यक हो जाता हैं। इस **Node** में किसी **Item** को **Represent** नहीं किया जाता यह **Node Header Node** या **List Header** कहलाता हैं। इस **Node** का **Information Part Unused** होता हैं। इस **Node** के **Information Part** में **List** की **Global Information** को रखा जाता हैं जैसे **Header Node** के अन्तर्गत के **List** में कितने **Nodes** हैं की **Information** को **Store** किया जाता हैं। **(Not Including Header Node)** इस प्रकार के **Data Structure** में किसी भी **Data Item** को **add** और **Delete** करने के लिए **Header Node** को भी **Adjust** करना पड़ता हैं। **Number of Nodes** कितने हैं। इसे **Directly Header Node** से **Access** कर लिया जाता है। इसके लिए **List Traversing** की आवश्यकता नहीं पड़ती हैं।

Header Node का एक अन्य उदाहरण निम्न हैं। एक **Particular Machine** का बनाने के लिए निम्न **Parts** को **Use** में लिया जाता हैं। जैसे **B841, K321, A087, J492** आदि। इस **Assembly** को **List** द्वारा प्रदर्शित किया जाता हैं। **Empty List** को **NULL Pointer** के द्वारा प्रदर्शित करते हैं।

Operation of Single Link List

Single List में **Structure** को **Use** में लिया जाता हैं। क्योंकि एक **Node** दो अलग **Data Types** की **Value** को **Store** करता हैं।

(1) Information Part
(2) Link Part

अतः **Link List** बनाने के लिए सबसे पहले **Link List** के **Structure** को बनाया जायेगा।

```
struct node
{
        int info;
        struct node * link;
};
```

Characteristics of Link List

(1) इस प्रकार की **Link List** में एक **Node** में दो अलग **Data Type** की **Values** को **Store** किया जाता हैं।
   - **(i) Information Part : -** जिसमें किसी भी **(int, char, double, long** आदि **Data Type** की **Values** को **Store** किया जा सकता हैं**)**
   - **(ii) Pointer Part: - Pointer Part** जिसमें **Next Node** का **Address** होता हैं। अतः **Link Part** के द्वारा ही **Next Node** को **Access** किया जा सकता हैं।

(2) **List** के जिस **Node** के **Link Part** में **NULL** होता हैं। वह **List** का **Last Node** कहलाता हैं।
(3) यदि **List** खाली है तो **Information** और **Link Part** दोनो **NULL** होगें।
(4) यदि **Information Part** में **Value** है तथा उस **Node** का **Link Part NULL** हैं अतः **List** में एक **Item** हैं।
(5) यदि **Link Part** में दूसरे **Node** का **Address** हैं अतः **List** खाली नहीं हैं।
(6) **Link List** के अन्तर्गत एक **Pointer List** के **First Node** को **Point** करता हैं। जिससे **List** के **Last Node** तक पहुंचा जा सकता हैं।

Operations on Link List

(1) Create an Empty Link List: -

(1) एक **Node Pointer head** बनाया जायेगा जिसमें किसी **Value** को **Initialize** नहीं किया गया हैं। यह **Pointer List** के **First Element** को **Point** करने के लिए **Use** में लिया जाता हैं।

(2) शुरूआत में **List Empty** होगी अतः **Head Pointer** को **NULL** से **Initialize** किया जायेगा।

(3) यह प्रदर्शित करता हैं कि **List Empty** हैं इसके लिए **C** में निम्न **Function** को बनाया जायेगा।

Function: -

```
void emptylist (struct node*head)
{
        head = NULL;
}
```

(2) Display List: - List को शुरूआत से **Last** तक **Traverse** किया जाता हैं। इसके लिए एक **Head Pointer** की आवश्यकता होती हैं। साथ ही एक और **Temporary Pointer** की आवश्यकता होती हैं जो **List** के **Last Node** तक जाता हैं। **List** का **Last Node** वह **Node** हैं जिसके **Link Part** में **NULL** होता हैं अतः दो **struct Node Pointer** की आवश्यकता होगी जिसमें पहला **List** के **First Node** को **Point** करेगा तथा दूसरा एक के बाद एक **Next Node** पर जायेगा और **Information Part** को **Print** करेगा।

Function: -

```
void printlist (struct node*head)
{
        struct node*t;
        t = head;
        if (t == NULL)
        {
                printf ("\n list is empty");
                return;
        }
        else
        while  (t → link != NULL)
        {
                printf ("%d", t → info);
                t = t → link;
        }
}
```

Insertion in Link List: - किसी भी **Element** को **List** में **Insert** करने के लिए सबसे पहले **Free Node** की आवश्यकता होती हैं। जब **Free Node** बना लिया जाता हैं तो **Node** के **Information Field** में **Data Value** को **add** कर देते हैं और इस नये **Node** को उसकी जगह पर **Add** कर दिया जाता हैं। किसी भी **List** में **Insertion 3** प्रकार से किया जा सकता हैं।

(1) List के **Beginning** में

(2) List के **End** में

(3) किसी दिये गये **Element** के बाद

Insertion at the Beginning: - किसी भी **Element** को **List** के **First** में **Add** करने के लिए सबसे पहले **List Empty** हैं या नहीं इसे **check** किया जाये। यदि **List Empty** हैं तो नया **Node** ही **List** का **First Node** होगा इसके लिए निम्न **Steps** को **Perform** करेंगे।

(1) New Node बनायेगें।

(2) New Node के Information Part में Data Value Enter करेंगे।
(3) New Node के Link Part NULL Value Insert करेंगे और अब
(4) List का Pointer Head उस नये Node को Point करेगा।

यदि List Empty नहीं है तो नये Node को List के सबसे आगे Add कर दिया जायेगा। इसके लिए निम्न Steps को Follow करेंगे।

(1) New Node बनायेगें।
(2) New Node के Information Part में Data Value Enter करेंगे।
(3) New Node के Link Part में Head (List Pointer जो List के First Item को Point कर रहा है) का Address डालेंगे।
(4) अब Head New Node को Point करेंगा

Function: -

```
void insert (struct node ** head, int item)
{
    struct node*new;
    new = malloc (sizeof (struct node));
    new → info = item;
    if (*head = = NULL)      //if the list is empty
    new → link = NULL;
    else
    new → link = *head;
    * head = new;
}
```

Insert at The End of the list: - यदि किसी Element को List के End में Insert करना हो तो सबसे पहले List Empty हैं या नहीं Condition को Test किया जायेगा यदि List Empty हैं तो नया Node List का First Node भी होगा और List का Last Node भी होगा। अतः निम्न Steps को Perform करेंगे।

(1) New Node बनायेंगे।
(2) New Node की Information Part में Data Value Insert करेंगे।
(3) New Node के Link Part में NULL डालेंगे।
(4) अब Head Pointer New Node को Point करेंगा।

यदि List Empty नहीं हैं तो List को Traverse कर Last Element तक पहुँचेंगे और बाद में Last में नये Node को Insert कर दिया जायेगा। इसके लिए निम्न Steps को Follow करेंगे।

(1) New Node बनायेगें।
(2) New Node की Information Part में Data Value Insert करेंगे।
(3) New Node के Link Part में NULL डालेंगे।
(4) अब Head Pointer के अलावा एक नये Pointer की आवश्यकता होगी जो List के Last Node तक जायेगा।
(5) List के Last Node के Link Part में नये Node का Address डालेंगे

Function:-

```
void insertatlast (struct node**head, int item)
{
        struct node * new, *temp;
        new = malloc (sizeof (struct node));
        new → info = item;
        new →  link = NULL;
```

```
        if (*head = = NULL)          //if list is empty
        *head = new;
        else
        {
                temp = *head;
                while (temp → link != NULL)
                temp = temp → link;
                temp → link = new;
        }
}
```

Insert After The Given Element Of The List: - यदि नये **Node** को किसी दिये गये **Element** के बाद **Insert** करना हो तो सबसे पहले उस **Location** को **Findout** किया जायेगा जहां **Node** को **Add** करना हैं और इसके बाद **List** में नये **Node** को **Add** कर दिया जायेगा। इसके लिए निम्न **Steps** को **Perform** करते हैं।

(1) **New Node** बनायेंगे।
(2) **New Node** की **Information Part** में **Data Value Insert** करेंगे।
(3) एक **Temporary Pointer** लेंगे जो **List** के उस **Element** को **Point** करेगा जिसके बाद नया **Node Add** करना हैं। यदि
(4) यदि **Temporary Pointer NULL** तक पहुँच जाता हैं अतः **List** में **Item** नहीं हैं जिसके बाद नये **Node** को **Add** करना हैं
(5) यदि **Temporary Pointer** उस **Item** को ढूँढ लेता हैं तो निम्न **Steps** को **Follow** करेंगे।
  (i) नये **Node** के **Link Part** में **Temporary Pointer** के **Link Part** को डाल देंगे।
  (ii) **Temporary Pointer** के **Link Part** में **New Node** को डाल देंगे

Function: -

```
void insertatmiddle (struct node **head, int loc, int item)
{
        struct node * new, * temp;
        int i;
        temp = *head;
        for (i = 0; i < loc; i++)
        {
                temp = temp →  link;
                if (temp = = NULL)
                {
                        printf ("item not found");
                        return;
                }
        }
        new = malloc (sizeof (struct node));
        new → info = item;
        new → link = temp → link;
        temp → link = new;
}
```

Delete The Element From The List

किसी भी **Element** को **List** से **Delete** करने के लिए सबसे पहले **Pointers** को **Properly Set** किया जाता हैं। बाद में उस **Memory** को **Free** कर दिया जाता हैं। जिसे **Node** द्वारा **Use** में लिया जा रहा था। **List** का **Deletion 3** जगह से किया जा सकता हैं।

(1) List के **Beginning** से
(2) List के **End** से
(3) दिये गये **Element** के बाद वाले **Node** को **Delete** करना

(1) Delete the Element from the beginning of the List: - जब **List** के **First Element** को **Delete** करना हो तो इसके लिए निम्न **Steps** को **Perform** किया जाता हैं।

(i) **Head** की **Value** को किसी **Temporary Variable** में **Assign** करना।
(ii) अब **head** के **Next Part** को **Head** में **Assign** कर दिया जायेगा।
(iii) जो **Memory ptr** द्वारा **Point** की जा रही है उसे **Deallocate** कर दिया जायेगा।

Function: -

```
void deletefromfirst (struct node **head)
{
        struct node*ptr;
        if (*head == NULL)
        return;
        else
        {
                ptr = *head;
                head = (*head) → link;
                free (ptr);
        }
}
```

Delete From End Of The List: - किसी भी **Element** को **List** के **End** से **Delete** करने के लिए सबसे पहले **List** को **Second Last Element** तक **Traverse** किया जायेगा। इसके बाद **Last Element** को **Delete** करने के लिए निम्न **Steps** को **Follow** किया जायेगा।

(1) यदि **head NULL** हैं तो **List** में कोई **Data Item** नहीं हैं अतः किसी **Item** को **Delete** नहीं किया जा सकता।
(2) एक **Temporary Pointer** लेंगे जिसमें **head** की **value** को **assign** करेंगे।
(3) यदि **head** का **link part NULL** हैं तो **List** में एक **Item** हैं जिसे **Delete** करना हैं।
(4) इसे **Delete** करने के लिए **head** में **NULL Assign** कर देंगे और **Pointer ptr** को **Free** करे देंगे।
(5) यदि **List** में एक से ज्यादा **Item** है अतः पहले **NULL** तक जायेगे और एक **Pointer Last Node** के पहले वाले **Node** को **Point** करेगा।
(6) IInd Last Node के **Link Part** में **NULL Value Assign** कर देंगे।
(7) वह **Pointer** जो **Last Node** को **Point** कर रहा था उसे **Free** कर देंगे।

Function: -

```
void deletefromlast (struct node **head)
{
        struct node *ptr, *loc;
        if (*head = = NULL)
        return;
        else if ((*head) → link = = NULL)
        {
```

```
            ptr = *head;
            *head = NULL;
            free (ptr);
        }
        else
        {
            loc = *head;
            ptr = (*head) → link;
            while (ptr → link != NULL)
            {
                loc = ptr;
                ptr = ptr → link;
            }
            loc → link = NULL;
            free(ptr);
        }
}
```

Delete After A Given Element Of The List: - जब किसी दिये गये **Element** के बाद वाले **Element** को **Delete** करना हो तो सबसे पहले **Location** को **Find out** किया जायेगा जिसके बाद वाले **Element** को **Delete** करना हैं इसके लिए निम्न **Steps** को **Follow** करते हैं।

(1) इसके लिए **2 Struct Node Pointers** लेगें।

(2) **Head Pointer List** के **first element** को **Point** करता हैं एक और **Pointer** वहीं **Point** करेगा जहां **Head Point** कर रहा है।

(3) यदि **Head NULL** है अतः **List Empty** है।

(4) यfn **List Empty** नहीं है तो एक अन्य **Pointer** लेगें जो **specific location** जिसे **Delete** करना है के पीछे रहेगा।

(5) अब पीछे वाले **Pointer** के **Link Part** में आगे वाले **Pionter** का **Link Part** डाल देंगे और **List Pointer** को **NULL** तक बढ़ाते रहेंगे।

Function: -

```
delete (struct node*head, int loc)
{
        struct node *ptr, *temp;
        ptr = head;
        if (head = = NULL)
        {
            printf ("list is empty");
            return;
        }
        else
        temp = ptr
        while (ptr != NULL)
        {
            if (ptr → info = = loc)
            {
                temp → link = ptr → link;
                free (ptr);
                return;
```

```
                }
                temp = ptr;
                ptr = ptr → link;

                }
        }
}
```

Doubly Link List

Doubly Link List को **Two way List** भी कहा जाता है जिसमें हर **Node** को **3 Parts** में **Divide** किया जाता है।

(1) **List** का पहला **Part Previous Pointer Field** कहलाता है जो **List** के पिछले **Element** का **address** रखता है।
(2) **List** का दूसरा **Part Information** को **Contain** करता है।
(3) **List** का **IIIrd Part Next Pointer** कहलाता है जो **List** के अगले **element** को **Point** करता है।

दो **Pointer Variable** को **Use** में लिया जाता है जो **List** के **First Element** तथा **List** के **Last Element** के **Address** को **Contain** करते हैं। **List** के **First Element** के **Previous Part** में **NULL** रहता है जो यह बताता है कि इस **Node** के पीछे कोई अन्य **Node** नहीं है। **List** के **Last Element** के **Next Part** में **NULL** रहता है जो यह बताता है कि इसके आगे कोई **Node** नहीं है। **doubly link list** को दोनो **Directions** में **Travers** किया जा सकता है।

Representation of Doubly Link List

माना की **user Link List** में **Integer Values** को **Store** करना चाहता है। इसके लिए **Doubly Link List Structure** का **memory** में होना आवश्यक है। अतः **Structure define** करेंगे।

Node Structure: -

```
struct node
{
        int info;
        struct node * prev, *next;
};
```

Insert A Node At The Begning

Doubly Link List के शुरू में किसी **Node** को जोड़ने के लिए यह **check** करते हैं कि **List** खाली तो नहीं है। यह देखने के लिए **head Pointer** को **check** करते हैं। यदि **Head NULL** है तो नये **Node** को अब **Head Point** करने लगेगा। नहीं तो **Head** के **Previous Pointer** में नये **Node** का **Address** डाल दिया जाता है। अतः किसी **Node** को जोड़ने के लिए निम्न **Algorithm** लिखेंगे।

Step 1:- नये **Node** को **Memory Allocate** करेंगे।
struct node * new = malloc (sizeof (struct node))

Step 2: - एक **Temporary Variable** में **Head** का **Address** डालेंगे।
struct node * temp = head

Step 3: - **New Node** के **Information Part** में **Data Item Insert** करेंगे।
new → info = item

Step 4: - **New Node** के **Previous Part** को **NULL** करेंगे।
new → prev = NULL

Step 5: - **New Node** के **Next Part** में **head** का **address** डालेंगे।
new → next = head

Step 6: - यह **check** करेंगे कि **head NULL** है या नहीं यदि **head NULL** है तो **Step 7** को **Execute** करेंगे अन्यथा **Execute** करेंगे।

Step 7: - **head** में नये **Node** का डाल देंगे अतः अब **head** नये **Node** को **Poin** करेगा।

Step 8: - **head** के **prev part** को **New node** का **Address** डालेंगे।
head → prev = new
head = new (अब head नये node को Point करने लगेगा)

Step 9: - **Exit**

Function: -

```
Insertatfirst (sturct node ** head, int item)
{
        struct node*new = malloc (sizeof (struct node));
        new → info = item;
        new → prev = NULL;
        new → next = head;
        if (head = = NULL)
        head = new;
        else
        {
                head → prev = new;
                head = new;
        }
}
```

Insert The Item At The End Of The Doubly Link List

Link List के **Last** में किसी नये **Node** को जोड़ने के लिए पहले यह **check** करेंगे कि **List** खाली तो नहीं है यदि **List** खली है तो पहला **Node** ही **First Node** होगा तथ वही **Last Node** होगा। यदि **List** में और भी आइटमस है तो नये **Node** की **Information Part** में **Data Item** तथा नये **Node** के **Next Part NULL** डालेंगे तथा उसे **List** के **Last** में **Add** कर देंगे।

At a last

Step 1: - नया **Node** बनयेंगे
stuct node * new = malloc (sizeof (struct node));

Step 2: - एक **Temporary *** लेंगे जिसमें **head** का **address** डाल देंगे
struct node * temp = head

Step 3: - नये **Node** के **Information Part** में **Data Item Insert** करेंगे।
new → info = item

Step 4: - **check that head = NULL**
यदि **head NULL** है तो **step 5** को **Run** करेंगे या अन्यथा **Step6** को **Run** करेंगे।

Step 5: - **Insert at first function** को **call** करेंगे
और **return** हो जायेगे।

Step 6: - **(1)** नये **Node** के **Information Part** में **Data Item Insert** करेंगे।
new → info = item
(2) नये **Node** के **Next Part** में **NULL Insert** करेंगे
new → next = NULL
(3) जब तक **T** का **Next part NULL** नहीं हो जाता है। **T** को आगे बढ़ाऐंगे।
while (temp → next != NULL)
temp = temp → next
(4) temp के **Next Part** में नये **Node** का **Address** डालेंगे।
temp → next = new

(5) नये Node के Prev Part में temp का address डालेंगे।
new → prev = temp

Step 7: - Exit

Function: -

```
addatlast (struct node **head, int item)
{
        struct node * temp = *head;
        struct node * new = malloc (sizeof (struct node));
        if (head = = NULL)
        {
                insertatfirst (& head, item);
                return;
        }
        new → info = item;
        new → next = NULL;
        while (temp → next != NULL)
        temp = temp → next;
        temp → next = new;
        new → prev = temp;
}
```

Traversing the Doubly link List

(1) In order Traversal: - इसके लिए निम्न **Steps** को **Follow** किया जाता है।

Step 1: - जब तक **head NULL** नहीं हो जाता तब तक **List Pointer** को आगे बढ़ायेंगे।
while (head != NULL)

Step 2: - **List** के हर **Information Part** को **print** करेंगे व **head** को आगे बढ़ाते जायेंगे।
print head → Info
head = head → next

(2) Reverse Order Traversal: - इसके लिए निम्न **Steps** को **Follow** किया जाता है।

Step 1: - जब तक **tail NULL** नहीं हो जाता तब तक **List Pointer** को आगे बढ़ायेंगे।
while (tail != NULL)

Step 2: - **List** के हर **Information Part** को **print** करेंगे व **tail** को आगे बढ़ाते जायेंगे।
print tail → Info
tail = tail → next

Deletion From Doubly Link List: -

(1) **Delete from Begning of List: -** यदि **doubly link list** खाली है तो **return** हो जायेंगे अन्यथा जिस **Node** को **head pointer point** कर रहा है उसे **Delete** करेंगे। इसके लिए निम्न **Steps** को **Follow** किया जायेगा। यहां **Head Pointer Struct Node Pointer** है जो **List** के **first node** को **Point** कर रहा है।

Step 1:- एक **Temporary Pointer** लेंगे जिसमें **Head** का **address** डालेंगे
struct node *temp = head

Step 2: - **Check** करेंगे कि **head NULL** है या नहीं यदि **head NULL** है तो **return** हो जायेगें।
if (head = = NULL)
return

Step 3: - यदि **head NULL** नहीं है तो **head** में **head** के **next part** को **assign** कर देंगे।

head = head → next
free (temp)

Step 4: - **head** जिस **Node** को **point** कर रहा है उसके **prev part** में **NULL Assign** करेंगे।
head → prev = NULL

Step 5:- **Exit**

Function: -

```
deletefromfirst (struct node *head)
{
        struct node *temp = head;
        if (head = = NULL)
        return;
        head = head → next;
        free (temp);
        head → prev = NULL;
}
```

Delete From Last: -

List के **Last Node** को **Delete** करने के लिए सबसे पहले **3 Conditions** को **Check** करेंगे।

(1) यदि **List** खाली है तो **return** हो जायेगें

(2) यदि **List** में केवल एक ही **Node** है तो उसे **delete** कर देंगे और **head** में **NULL Insert** कर देंगे।

(3) यदि **List** में एक से ज्यादा **Items** है तो पहले **Last Node** तक पहुंचेगे बाद में **Last Node** को **Delete** कर देगें। इसके लिए निम्न **Algorithm** हैं।

Step 1: - एक **Temporary Pointer** लेगें जिसमें **hand** का **address** डालेंगे
struct node * temp = head

Step 2: - यह **check** करेंगें कि **head NULL** है या नही यदि **head NULL** है तो **return** हो जायेंगे।

Step 3: - यह **check** करेंगें कि **head** का **Next NULL** है या नही। यदि **head** का **Next NULL** है तो **4** से **6** तक के **steps** को **repeat** करेंगे।

Step 4: - **head** में **NULL insert** करेंगें।
head=NULL

Step 5: - **Temporary Pointer** को **Free** कर देंगे
free(t)

Step 6: - **return**

Step 7: - जब तक **temp** के **Next part** में **NULL** नहीं आ जाता तब तक **temp** को आगे बढ़ाएगें।
while(temp->next==NULL)
temp=temp->next

Step 8: - **temp** के **prev** के **next part** में **NULL insert** करेंगे।
temp->prev->next=NULL

Step 9: - **Temp** को **Free** कर देगें।
free(temp)

Step 10: - **exit**

Function: -

```
deletefromlast (struct node *head)
{
        struct node * temp = head;
```

```
        if (head = = NULL)
        return;
        if (head → next = = NULL)
        {
                head = NULL;
                free (temp);
                return;
        }
        while (temp → next != NULL)
        temp = temp → next;
        temp → prev → next = NULL;
        free (temp);
}
```

Circular Link List: - **Simple Link List** में **Last Node** का **Link Part** हमेशा **NULL** होता है लेकिन **Circular Link List** में **List** के **Last Node** के **Link Part** में हमेशा **head** का **address** रहता है। अतः **Circular Link List** का आखरी **Node** हमेशा **List** के **First Node** को **Point** करता है।

Insert the New Node In the Circular Link List

Insert at First:-

जब **Circular Link List** के प्रारम्भ में किसी नये **Node** को जोड़ना हो तो इसके लिये सबसे पहले यह **Check** करेंगे कि **List** खाली तो नहीं है। यदि **head NULL** है तो नये **Node** को **Insert** कर दिया जायेगा और **Head Pointer** उसे **Point** करेगा यदि **Head NULL** नहीं है तो **head** पर पहुंच कर नये **Node** को **add** कर दिया जायेगा। उसके लिये निम्न **Algorithm** हैं।

Step1: - **New Node** बनायेंगे
struct node*new = malloc (sizeof (struct node))

Step2: - एक **Temporary Pointer** बनाएगें। जिसमें **head** का **address** डालेंगे।
struct node * temp = *head

Step 3: - **New Node** के **Information Part** में **Item Insert** करेंगे
new → info = item

Step 4: - **Check** करेंगे कि **head NULL** है या नहीं यदि **head NULL** है तो **5th step** को **execute** करेंगे नहीं तो **6th Step execute** होगा।

Step 5:- **Head** में नया **Node insert** करेंगे
head = new node
new node → next = head

Step 6: - **new node** के **next part** में **head** का **address** डालेंगे
new node → next = head
temp के **next part** में जब तक **head** का **address** नहीं आ जाता तब तक **temp** को आगे बढ़ाएगें।
while (temp → next != head)
temp = temp → next
head में नये **node** का **address** डालेंगे
head = new node
temp के **next part** में **head** का **address** डालेंगे
temp → next = head

Step 7: - **Exit**

Function: -

```
struct node
{
        int data;
        struct node *next;
};
void append(int x,struct node **p)
{
        struct node *temp=*p,*n=malloc(sizeof(struct node));
        n->data=x;
        printf("\nappending:%d",x);
        if(temp==NULL)/*list empty*/
        {
                *p=n;
                n->next=n;
                return;
        }

        while(temp->next !=*p)
                temp=temp->next;
        temp->next=n;
        n->next=*p;
}
```

Insert at Last: -

इसके लिए सबसे पहले यह **check** करेंगे की **List** खाली है या नहीं यदि **List** में एक भी **Item** नहीं है तो **appedn function** को **call** करेंगे नहीं तो **while loop** की सहायता से उस **Node** तक पहुंचेगे जहां **temp** के **next** में **Head** होगा अर्थात् **Last Node** तक पहुचेंगे अब इस **Node** के **Info Part** में **data item insert** करेंगे और **Node** के **Next Part** में **head** का **address** डाल देंगे। इसके लिए निम्न **Algorithm** हैं।

Step 1: - नये **Node** को **memory allocate** करेंगे
struct node*new = malloc (sizeof (struct node))

Step 2: - एक **Temporary Pointer** बनाएगें। जिसमें **head** का **address** डालेंगे।
struct node * temp = *head

Step 3: - **New Node** के **Information Part** में **Item Insert** करेंगे
new → info = item

Step 4: - **Check** करेंगे कि **head NULL** है या नहीं यदि **head NULL** है तो **5th step** को **execute** करेंगे नहीं तो **6th Step execute** होगा।

Step 5:- **appedn function** को **call** करेंगे
और **return** हो जाऐंगे

Step 6: - **new node** के **next part** में **head** का **address** डालेंगे
new node → next = head
temp के **next part** में जब तक **head** का **address** नहीं आ जाता तब तक **temp** को आगे बढ़ाएगें।
while (temp → next != head)
temp = temp → next
temp के **next part** में **head** का **address** डालेंगे
temp → next = new

Step 7: - **Exit**

Circular List की Traversing

Step 1: - एक **Temporary Pointer** बनाएगें। जिसमें **head** का **address** डालेंगे।
struct node * temp = head

Step 2: - **if head = NULL**
return

Step 3: - **Repeat Step 4 to 5 while temp → next != head**

Step 4: - **print temp → info**

Step 5:- **temp = temp → next**
(end of while loop)

Step 6: - **print temp → info**

Step 7: - **Exit**

Delete an Item from the Circular Link List

Delete from first: - Circular Link List के प्रारम्भ से किसी भी **Node** को **delete** करने के लिए **3 conditions** को **check** करेंगे

(1) List खाली है।

(2) List में केवल एक **Node** हैं तो उस **Node** को मिटाकर **head** में **NULL** डाल देंगे

(3) List में एक से ज्यादा **Node** है तो **Loop** की सहायता से **first Node** पर पहुंचेगें और उस **Node** को मिटाकर **head** को अगले **node** पर ले जायेंगे। इसके लिए निम्न **Algorithm** हैं:–

Step 1: - एक **Temporary Pointer** लेगें जिसमें **head** का **address** डालेंगे
struct node * temp = head

Step 2: - यह **check** करेंगें कि **head NULL** है या नही यदि **head NULL** है तो **return** हो जायेंगे।

Step 3: - यह **check** करेंगें कि **head** का **Next head** है या नही। यदि **head** का **Next head** है तो **4** से **6** तक के **steps** को **repeat** करेंगे।

Step 4: - **Temp** को **free** कर देंगे।
head=NULL

Step 5: - **head** में **NULL** डालेंगे।
head = NULL

Step 6: - **return**

Step 7: - जब तक **temp** के **Next part** में **head** का **adres** नहीं आ जाता तब तक **temp** को आगे बढ़ाएगें।
while(temp->next != head)

Step 8: - **temp=temp->next**

Step 9: - **head = head → next (head** में **head** का **next Insert** करेंगे।**)**

Step 10: - **temp** के **next** में **head** डालेंगे।
temp → next = head

Step 11: - **free(temp)**

Step 12: - **Exit**

Delete from Last: -

Circular List के अन्त में से किसी **Node** को हटाने के लिए **3 conditions** को **check** करेंगे

(1) यदि **List** खाली है

(2) यदि **List** में केवल एक **Node** है तो **head** में **NULL** डाल देंगे

(3) यदि **List** में एक से ज्यादा **Node** है तो **Loop** की सहायता से **List** के अन्तिम **Node** पर पहुंचेगे उस **Node** को मिटाकर उस **Node** के पिछले **Node** में **head** का **address** डालेंगे। इसके लिए निम्न **Algorithm** हैं।

Step 1: - एक **Temporary Pointer** लेगें जिसमें **head** का **address** डालेंगे
struct node * temp = head

Step 2: - यह **check** करेंगें कि **head NULL** है या नही यदि **head NULL** है तो **return** हो जायेंगे।

Step 3: - यह **check** करेंगें कि **head** के **Next** में **head** है या नही। यदि **head** का **Next head** है तो **4** से **6** तक के **steps** को **repeat** करेंगे।

Step 4: - **head** में **NULL** डालेंगे।
head=NULL

Step 5: - **temp** को **free** कर देंगे।
free (temp)

Step 6: - **return**

Step 7: - जब तक **temp** के **Next part** में **head** का **adress** नहीं आ जाता तब तक **temp** को आगे बढ़ाएगें।
while(temp->next != head)
temp=temp->next

Step 8: - **temp** के **next** को **free** कर देंगे।
free (temp → next)

Step 9: - **temp** के **next** में **head** डालेंगे।
temp → next = head

Step 10: - **Exit**

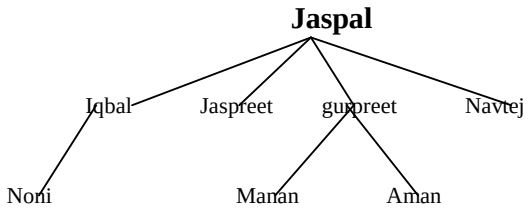
Application of Link List

Linked List एक **Primitive data Structure** है जिसे विभिन्न प्रकार की **Application** में **Use** में लिया जाता है। उनमें से कुछ **Applications** निम्न है।

(1) किसी अन्य **Data Structure** को **Implement** करने के लिए जैसे **Stack queue, tree** तथा **graph**
(2) **Name** की **Directory** को **maintain** करने के लिए
(3) **Logn Integers** पर **Arithmetic Operation Perform** करने के लिए
(4) **Polynomials** को **Manipulate** करने के लिए
(5) **Sparts Matrix** को **Represent** करने के लिए

UNIT-III
TREES

Arrary,Linked list,stacks and queues आदि को **Linear data structure** के व्दारा प्रदर्शित किया जाता है। इन **structure** के व्दारा **hierarchical data** को प्रदर्शित नही किया जा सकता। **Hierarchical data** में **ancestor-descendant,superior-subordinate,whole part** व **data elements** के बीच **similar relationship** होती है।



उपरोक्त **figure** में **jaspal** के **descendents** को **diagram** मे **hierarchical order** में प्रदर्शित किया गया है। जिसमें **jaspal top of the hierarchy** को प्रदर्शित कर रहा है। **Jaspal** के **children iqbal,jaspreet,gurpreet and navtej** है। **Iqbal** के एक **child Noni** है,**gurpreet** के दो व **jaspreet and navtej** के एक भी नही है। इस प्रकार यह कहा जा सकता है कि **Iqbal** के **siblings Noni, Jaspal** के **descendents iqbal,jaspreet,gurpreet and navtej** है तथा **Noni** के **ancestors Manan and Aman** है।

The tree is an ideal data structure for representing the hierarchical data. As there are many
types of trees in the forest,likewise there are many types of trees in data structure-tree,binary tree,expression tree,tournament tree,binary search tree,threaded tree,AVL tree and B-tree.

Tree एक **ideal data structure** है जिसके द्वारा **hierarchical data** को प्रदर्शित किया जाता है। जैसा कि हम जानते हैं कि **forest** में बहुत से **tree** होते हैं। उसी प्रकार **data structure** में भी कइ प्रकार के **tree** होते हैं जैसे :-

Tree, binary tree, expression tree, tournament tree, binary search tree, threaded tree, AVL tree and B-tree.

Tree Terminology

Level of the Eement: - Tree में सबसे ज्यादा प्रयोग में आने वाली **term level** है। **Tree** के **first element** को **root** कहा जाता है। **Tree** के **Root** को हमेशा **Level 1** में रखा जाता है। उपरोक्त **Diagram** में **Jaspal Level1** में **iqbal,jaspreet,gurpreet and navtej Level 2** में तथा **Noni, Aman and Manan Level 3** में हैं।

Degree of Element: - किसी भी **Element** की **Degree Number of Children** पर **Depend** करती हैं। **Leaf Node** की **Degree** हमेशा **0** होती हैं क्योंकि **Leaf Node** का **Left** और **Right Part** नहीं होता हैं।

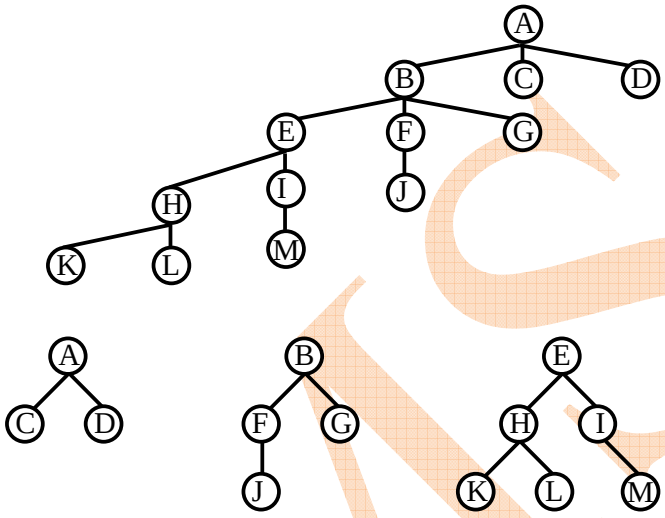
Degree of The Tree: - किसी भी **tree** की **Degree Tree** में उपस्थित **Maximum Elements** पर **Depend** करती हैं। उपरोक्त **diagram** में **tree** की **Degree 4** हैं। क्योंकि उसमें एक **Nodes** से अधिक से अधिक **4 Elements** जुड़े हैं।

Properties Of Tree : - Tree के निम्न गुण हैं।

(1) कोई भी **Node Tree** की **Root** हो सकती हैं। **Tree** में जुड़ी हर **Node** का एक गुण होता हैं कि प्रत्येक **Node** को किसी दूसरी **Node** से जोड़ने के लिए केवल एक ही **Path** होता हैं। एक **Tree** जिसमें किसी **Root** को **Identify** नहीं किया गया हैं वह **Free Tree** कहलाता हैं।

(2) जिस **Tree** में **n No. of Nodes** हैं उसमें **n-1 Edges** होंगे।

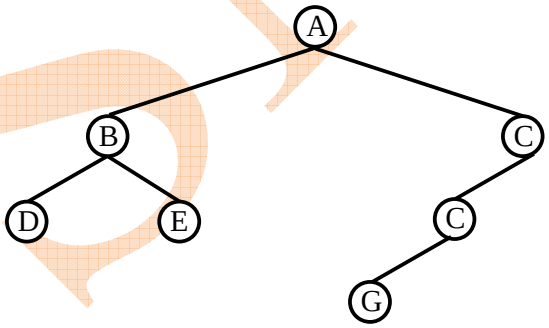
(3) वे **Nodes** जिनका कोई **Child Node** नहीं होता **leaves** या **terminal node** कहलाता हैं।
(4) वे **Node** जिनके पास कम से कम एक **Child** होता हैं **Nonterminal Nodes** कहलाते हैं।
(5) **Nonterminals Node** को **Internal Node** तथा **Terminal Node** को **External Node** कहते हैं।
(6) एक **Level** पर उपस्थित सभी **Nodes** के लिए **Depth** और **Height** का मान वही होता हैं जो उसके **Level** का मान होता हैं।
(7) एक **Node** की **Depth** को **Node** का **Level** भी कहा जाता हैं। **Tree** के एक संग्रह को **Forest** कहते हैं। नीचे दिये गए **tree** के **root** और **edges** जो कि उसे **tree** से जोड़ रहें हैं को हटा दिया जाए तो हमारे पास **forest** बचेगा जिसमें अन्य **3 Trees** होंगे



Forest

Binary Tree

ऐसा **Tree** जिसकी प्रत्येक **Node** के पास अधिकतम दो **Child Nodes** हो सकती हैं, **Binary Tree** कहलाता हैं। इसे हम इस प्रकार भी कह सकते हैं कि **Binary Tree** में किसी **Node** के पस के दो अधिक **Child Nodes** नहीं हो सकती हैं।



Binary Tree

Binary Tree सामान्य **Tree** के एक विशेष **Class** में आते हैं।

Properties of Tree

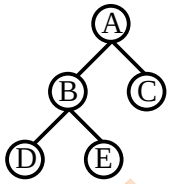
Binary Tree के निम्नलिखित **Properties** हैं –

(1) एक **Binary Tree** जिसमें **n** आन्तरिक **Nodes Internatl Nodes** हैं, उसमें अधिकतम् **n+1 External Nodes** हो सकती हैं। यहां पर **Root Node** को भी **Internal Node** के रूप में गिना गया हैं।
(2) एक **Binary Tree** जिसमें **n Internal Nodes** हैं, के **External Path** की लम्बाई, **Internal Path** की लम्बाई की दो गनुी होती हैं।

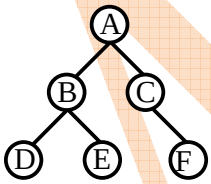
(3) एक **Binary Tree** जिसमें **n Internal Nodes** हैं की उंचाई **height** लगभग $\log_2 n$ होती हैं।
प्रत्येक **Binary Tree** एक **Tree** होता है परन्तु प्रत्येक **Tree** एक **Binary Tree** नहीं होता।
एक पूर्ण **Binary Tree (Full** अथवा **Complete Binary Tree)** की समस्त **Internal Nodes** की **Degree** होती हैं और समस्त **Leaves** एक ही **Level** पर होती हैं।
Binary Trees को **3 Parts** में बांटा गया हैं।

(1) Strictly Bianry Tree
(2) Complete Binary Tree
(3) Full Binary Tree

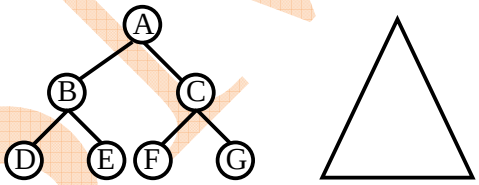
(1) **Strictly Binary Tree: -** वह **Tree** जिसमें कम से कम **0** या **2 Nodes** का होना आवश्यक हैं अर्थात् एक **Tree** में एक **Node** से या तो **Left** और **Right** दोनो **Values** को **Insert** किया जायेगा या एक भी नहीं।



(2) **Complete Binary Tree: -** वह **Tree** जिसका **Level Same** हो **All most Complete Binary Tree** कहलाता हैं। जैसे :–

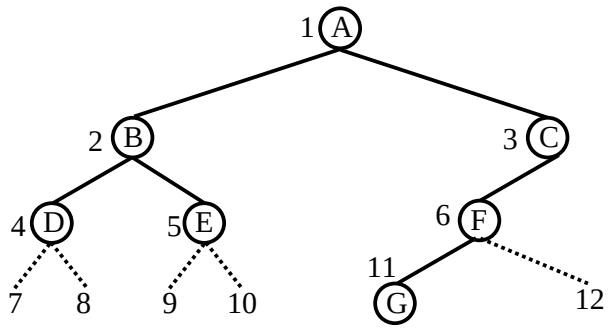


(3) **Full Binary Tree: -** वह **Tree** जो **Strictly Binary** व **Complete binary** दोनो हो **Full Binary Tree** कहलाता हैं। यह **Triangle Property** को प्रदर्शित करता हैं।



Tree Representation as Array

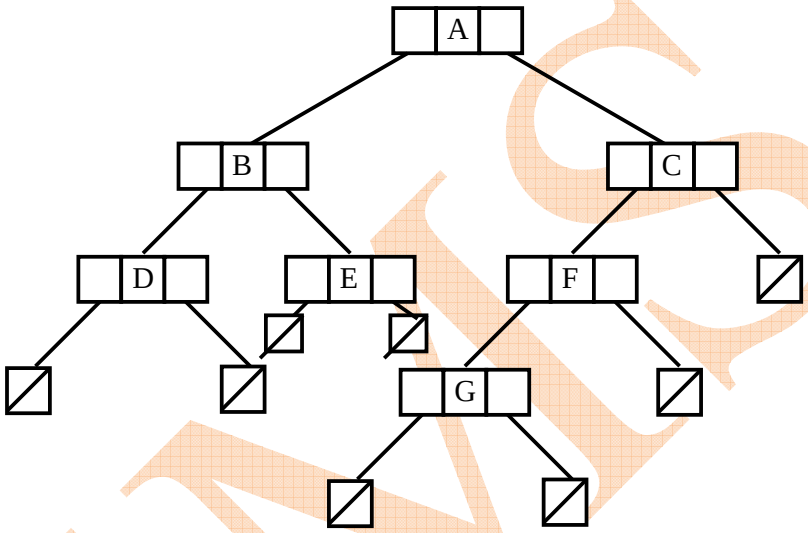
Array के द्वारा **Binary Tree** को प्रदर्शित करने के लिए **Tree** के **Nodes** की **Numbering** करनी होती हैं। किसी भी **Tree** के **Nodes** की **Numbering** करने के लिए हम **Binary Tree** को **Full Binary Tree** मानकर हर **Node** को एक **Number** दे देते हैं और जिस **Number** पर वह **Node** होता हैं उसे **Array** में **Store** कर देते हैं।
इसका **Example** निम्न हैं।



1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F					G	

Array Representation of a Binary Tree

Link List Representation of Binary Tree: - किसी भी **Tree** को **Linked List** द्वारा सही से प्रदर्शित किया जा सकता हैं। हम जानते हैं कि एक **Binary Tree** में अधिकतम दो **Child Node** ही होते हैं। अतः इस प्रकार के **Representation** में **node Value** के दोनो ओर एक–एक **Pointer Maintain** करते हैं। जिससे उसकी **Child Nodes** को प्रदर्शित किया जाता हैं। **Tree** का वह भाग जिसमें किसी सूचना को **Store** नहीं किया जाता उन्हें **NULL** के द्वारा सुरक्षित रखा जाता हैं।



Linked List Representation of a Binary Tree

Basic Operations on Binary Tree

Binary Search Tree एक विशेष प्रकार का **Binary Tree** है जो या तो खाली है या **Left Sub Tree** की सारी **Information Root** से छोटी है और **Right Sub Tree** की सारी **Information Root** से बड़ी हैं **Left** और **Right** दोनों ही **Sub Trees Binary Search Tree** हैं।

Binary Search Tree में निम्न **Operations** को **Perform** करना होता हैं।

(1) Insertion
(2) Delition
(3) Traversal

Link List द्वारा जब **Tree Representation** किया जाता हैं तो **Node** के तीन भाग होते हैं

(1) Left (2) Info (3) Right

इसके लिए निम्न **Strcutre** को बनाया जाएगा

```
struct node
{
        struct node * left;
        int info;
        struct node * right;
};
```

Binary Search Tree में **Node** को जोड़नाः – इसके लिए सबसे पहले एक **Temprarory Variable T** को **Root** पर और एक **Variable Prev** को **NULL** पर भेज देते हैं। **Prev Variable T** से पहले वाली **Node** को प्रदर्शित करेगा अब एक नये **Node** को **Memory Allocate** करेंगे उसके **Information Part** में **Data** तथा **left** और **Right** में **NULL Value Insert** करेंगे इसके बाद **Loop** के द्वारा **Prev** को **T** पर भेजा जाएगा और यह **Check** किया जाएगा कि क्या **Data** की **Value T** की **Value** से छोटी हैं।

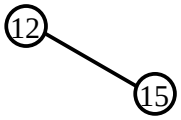
यदि **Data** की **Value T** से छोटी है तो उसे **T** के **Left** में **Insert** कर दिया जाएगा यदि **Data** की **Value T** से बड़ी है तो उसे **T** के **Right** में **Insert** किया जाएगा। अब यह **Check** करेंगे कि **Previous Variable** जो कि **T** के पीछली **Node** को प्रदर्शित कर रहा हैं **NULL** तो नहीं हैं। यदि **NULL** है अतः **Tree** खाली हैं तो नये **Node** को **Root** बना देंगे।

Step 1: - एक **Temporary Pointer** लेगें जिसमें **root** का **address** डालेंगे
struct node * t = root
Step 2: - **struct node * prev = NULL**
Step 3: - **struct node * n = malloc (sizeof (struct node))**
Step 4: - **n → info = data**
n → left = NULL
n → right = NULL
Step 5: - **while (t != NULL and data != t → info)**
set prev = t
check whether (data < t → info)
if yes then
t = t → left
else
t = t → right
Step 6: - **check whether (data = t → info)**
if yes then return
Step 7: - **check whether (prev = NULL)**
if yes then
root = n
return
Step 8: - **check whether (data < prev → info)**
if yes then
prev → left = n
else
prev → right = n
Step 9: - **Exit**

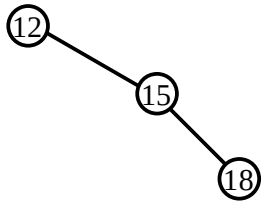
निम्न **Numbers** को यदि **Binary Tree** में **Insert** करना हो तो उसके लिए निम्न **Steps follow** करेंगे
12, 15, 18 10, 11, 14, 22

(1) सबसे पहले 12 को **add** करना है चुंकि अभी तक कोई **Node Add** नहीं हुई है, इसीलिए यह साधारण रूप से **Add** हो जाएगी।

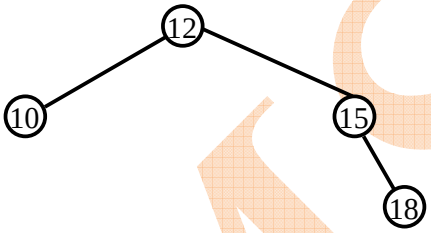
(2) इसके बाद **15** को **Add** कराना है। चूंकि **15, 12** से बड़ा है इसीलिए **15, 12** के दाईं ओर **Add** होगा।



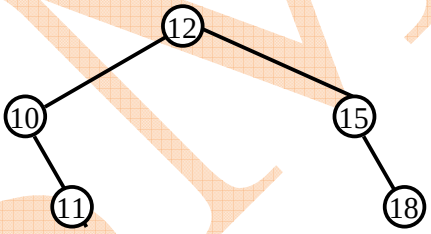
(3) **18** को **Add** करते हुए हमने पाया कि **18, 12** से बड़ा है परन्तु चूंकि **12** के दाईं ओर **15** पहले से ही है इसीलिए हम इसकी **15** से भी तुलना करेंगे। तुलना करने पर हमने पाया कि **18, 15** से भी बड़ा है। इसीलिए **18, 12** के भी दाईं ओर **Add** होगा।



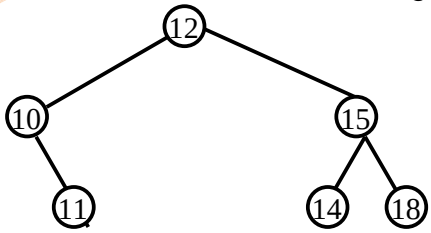
(4) **10** को **Add** करते हुए हमने पाया कि **10, 12** से छोटा है, इसीलिए **10, 12** के बाईं ओर **Add** होगा।



(5) **11** को **Add** करते हुए हमने पाया कि **11, 12** से तो छोटा है, इसीलिए **12** के बाईं ओर **Add** होगा परन्तु **12** के बाईं ओर पहले से सुरक्षित सूचना **10** से बड़ा है, इसीलिए **11, 12** के बाईं ओर परन्तु **10** के दाईं ओर **Add** होगा।



(6) **14** को **Add** करते हुए हमने पाया कि **14, 12** से बड़ा है इसिलये यह **12** के दाईं ओर **Add** होगा। परन्तु यहां पर पहले से ही **18** सुरक्षित है। इसलिये हमने **14** की तुलना **15** से की और पाया कि **14, 15** से छोटा है इसीलिये **14, 12** के दाईं ओर परन्तु **15** के बाईं ओर **Add** होगा।

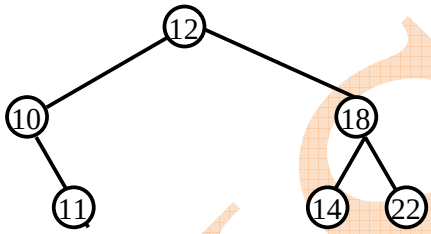


(7) अब हमें अन्तिम सूचना अर्थात् **22** को **Add** करना है। **22** को **Add** करते हुए हमने देखा कि **22, 12** से बडऋा है, इसलिये **22, 12** के दाईं ओर **Add** होगा। परन्तु **12** के दाईं ओर पहले से **15** सरुक्षित है, इसलिये हमने इसकी तुलना **15** से की और पाया **22, 15** से भी बड़ा है। इसलिये **22, 15** के भी दाईं ओर जाएगा, परन्तु **15** के दाईं ओर **18** सुरक्षित है, अतः **22** की तुलना **18** से भी

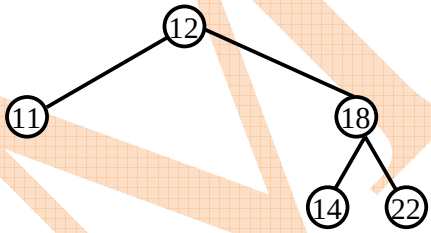
(8) की और पाया कि **2, 18** से भी बड़ा है, इसलिये **22, 18** के दाईं ओर **Add** होगा। अतः हम कह सकते हैं कि **22, 15** के दाईं ओर सुरक्षित **18** के दाईं ओर **Add** होगा।

Delete a Node from Binary Tree.

(1) 15 को **Delete** करते हुए हमने पाया कि **15** के **left** और **right child** दोनों भरे हुए हैं, इसलिये हमने अपनी **Algorithm** के अनुसार इसका **Inorder (14, 15, 18)** निकाला और पाया कि **18** इसका **Inorder sucessor** हैं इसलिये **15** को **delete** करके **18** को **Promote** कर दिया। अतः अब **Tree** निम्नानुसार रह जाता है –



(2) 10 को **Delete** करते हुए हमने पाया कि **10** का केवल **right child** ही है, इसलिये हमने इसके **child** को **promote** करके **10** को **delete** कर दिया। अतः अब **tree** निम्नानुसार रह जाता है –



Traversal of Binary Tree

Binary Tree को **Traverse** करना सबसे **Important Operation** हैं। **Binary Tree** पर बहुत सारे **Operations Perform** किये जाते हैं। **Traversing** करना **tree** के हर **Node** पर होता हैं। **Traversing** एक **Process** हैं। जिसमें **Tree** के हर **Nodes** से गुजरना होता हैं। किसी भी **Tree** को **Traverse** करने के लिए सबसे पहले उसके **root** को **check** किया जाता हैं। इसके बाद **left Sub Tree** को **Traverse** करते हैं तथा अन्त में **Right sub tree** को **traverse** करते हैं। **Normally Binary Tree** के **Traversal 3** प्रकार के होते हैं।

(1) In order Traversal: - (Left, Root, Right)

- **(i)** सबसे पहले **Left Sub Tree** को **Traverse** करो।
- **(ii)** **Root** को **Traverse** करो।
- **(iii)** **Last** में **Right Sub Tree** को **Traverse** करो।

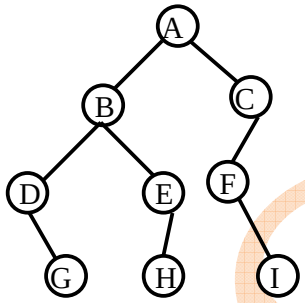
(2) Pre Order Traversal: - (Root, Left, Right)

- **(i)** **Root** को **Traverse** करो।
- **(ii)** **Left Sub Tree** को **Traverse** करो।
- **(iii)** **Last** में **Right Sub Tree** को **Traverse** करो।

(3) Post Order Traversal: - (Left, Right, Root)

- **(i)** **Left Sub Tree** को **Traverse** करो।
- **(ii)** **Right Sub Tree** को **Traverse** करो।

(iii) Root को Traverse करो।

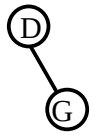


उपरोक्त **Tree** में यदि तीनों प्रकार कर **Traversal** करना हो तो इसके लिए **Tree** को **Root** से **Access** किया जाएगा।

In Order Traversal of Tree: - (Left, Root, Right)

सबसे पहले **Root Node A** पर जाऐंगे। अब देखेंगे कि **root** का **Left Full** है अतः **A** के **Left** में **B** हैं। अब **B** के **Left** को **Check** करेंगे **B** का **Left** भी **Full** हैं। इसके **Left** में **D** हैं। पर **D** का **Left** खाली हैं अतः यहां से **Left Root** और **Right** को **Traverse** करेंगे। **D** का **Left** खाली हैं अतः **D** और **G (Root, Right)** को **Traverse** करेंगे।

(1) इस **Tree** का **In order Traversal D** और **G** होगा।



(2) D खुद **B** का **Left** हैं अतः अब **Root** को लिखेंगे **DGB**

(3) B का **Right E** हैं तथा **E** का **Left H** हैं अतः **EH** वाले **Tree** के अर्न्तगत **Traversal HE** होग अतः **Tree**

D G B H E

(4) Left sub Tree complete हो चुका हैं अब **root** को लिखेंगे

D G B H E A

(5) अब **Right Sub Tree** को **Access** करेंगे। **F** का **Left NULL** है तथा **Right** में **I** हैं। **L, Root, R** के आधार पर **Traversal FI** होगी।

D G B H E A F I

(6) F का **root C** में और **C** का **Right NULL** हैं। अतः **Traversal**

D G B H E A F I C

Pre Order Traversal: - (Root, Left, Right)

सबसे पहले **Root Node A** पर जाऐंगे। अब देखेंगे कि **root** का **Left Full** है अतः **A** के **Left** में **B** हैं। अब **B** के **Left** को **Check** करेंगे **B** का **Left** भी **Full** हैं। इसके **Left** में **D** हैं। पर **D** का **Left** खाली हैं अतः यहां से **Root, Left** और **Right** को **Traverse** करेंगे।

(1) सबसे पहले **root** को देंगे **Tree** का **Root A** हैं अतः **Tree Traversal** में **First Element A** है।

A

(2) A के **Left Sub Tree** को **Check** करेंगे **A** के **Left** में **B** अतः **Tree Traversal**

A B

(3) B का **Left D** है और **G** का **Root D** है अतः **Tree Traversal**

A B D

(4) **D** का **left NULL** है और **Right** में **G** है अतः **root Left Right** के आधार पर **Tree Traversal**
A B D G
(5) **B** के **right** में **E** है और **E H** का **Root** है अतः **Root left right** के आधार पर **Tree Traversal**
A B D G E H
(6) अब **Right Sub Tree** को **Traverse** करेंगे **A** का **Right C** है अतः **Tree Traversal**
A B D G E H C
(7) **C** के **Left** में **F** है तथा **F I** का **Root** है अतः **Tree Traversal**
A B D G E H C F I

Post Order Traversal: - (Left, Right और **Root)**
सबसे पहले **Root Node A** पर जाऐंगे। अब देखेंगे कि **root** का **Left Full** है अतः **A** के **Left** में **B** हैं। अब **B** के **Left** को **Check** करेंगे **B** का **Left** भी **Full** हैं। इसके **Left** में **D** हैं। पर **D** का **Left** खाली हैं अतः यहां से **Left, Right, Root** को **Traverse** करेंगे।

(1) **A** का **Left B** है **B** का **Left D** है और **D** का **Left NULL** है पर **D** का **Right G** है अतः **Left Right Root** के आधार पर **Traversal**
G D
(2) **B** का **Left Sub Tree Complete** है और **B** के **right sub tree** पर जऐंगे। **B** के **right** पर **E** है और **E** के **Left** में **H** अतः **Traversal**
G D H E B
(3) **A** का **Left Sub Tree traverse** हो चुका है अब **A** के **Right Sub Tree** को **traverse** करेंगे।
(4) **A** के **Right** में **C** है **C** के **Left** में **F** है तथा **F** का **Right I** है अतः पहले **IF** को लेंगे अतः **Tree Traversal**
G D H E B F I
(5) **C** का **Left Sub Tree complete traverse** हो चुका है अतः अब **Right Sub Tree NULL** है। अतः **Tree Traversal**
G D H E B F I C
(6) सबसे अन्त में **Root Node** को लिखा जाएगा।
G D H E B F I C

Threaded Binary Tree

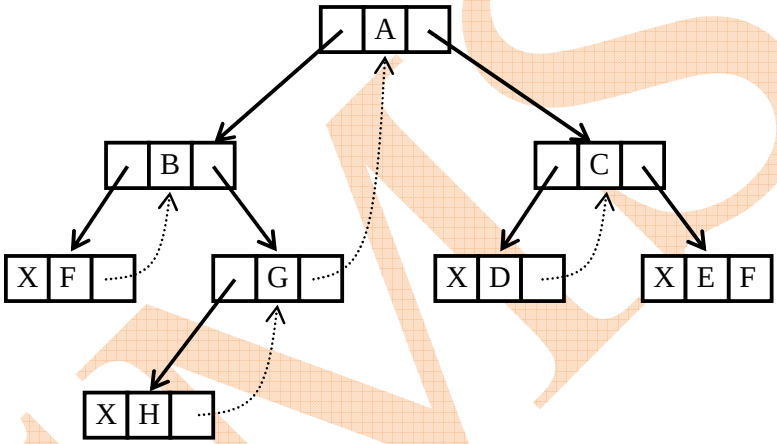
जब किसी **binary tree** को **link list** के रूप में प्रदर्शित किये जाते हैं तो **Normally** आधे से ज्यादा **Pointer Field NULL** होते हैं। वह **Space** जिसे **NULL Entry** द्वारा **Occupy** किया जाता हैं। इसमें **Valuable Information** को भी **Store** कर सकते हैं। इसके लिए एक **Special Pointer** को **Store** किया जाता हैं जो अपने **Higher Nodes** को **Point** करता हैं। इन **Special Pointer** को **Threades** कहते हैं। तथा ऐसा **Tree** जिसमें इस प्रकार के **Pointers** को रखा जाता हैं **Threaded Binary Tree** कहलाते हैं।
Threads Normal Pointer से अलग होते हैं। **Threaded Binary Tree** में **Threades** को **dotted line** के द्वारा प्रदर्शित किया जाता हैं इसके लिए एक **Extra Field** को **Memory** में **Store** करते हैं जिसे **Tag** या **Flag** कहा जाता हैं। **Threaded Binary Tree** को प्रदर्शित करने के बहुत से तरीके हैं।

(1) **Tree** के **Right NULL Pointer** के हर **Node** को **Successor** से **Replace** करवाया जाएगा। यह **In Order Traversal** के अन्तर्गत आता हैं। इसलिए इस **Thread** को **Right Thread** कहा जाता हैं और इस प्रकार का **Tree Right Threaded Binary Tree** कहलाते हैं।
(2) **Tree** के **Left NULL Pointer** के हर **Node** को **Predecessor** से **Replace** करवाया जाएगा। यह **In Order Traversal** के अन्तर्गत आता हैं। इसलिए इस **Thread** को **Left Thread** कहा जाता हैं और इस प्रकार का **Tree Left Threaded Binary Tree** कहलाते हैं।
(3) जब **Left** और **Right** दोनों **Pointers** को **Predecessor** और **Successor** से **Point** करवाया जाता हैं तो इस प्रकार के **Tree** को **Full Threaded Binary Tree** कहलाते हैं।

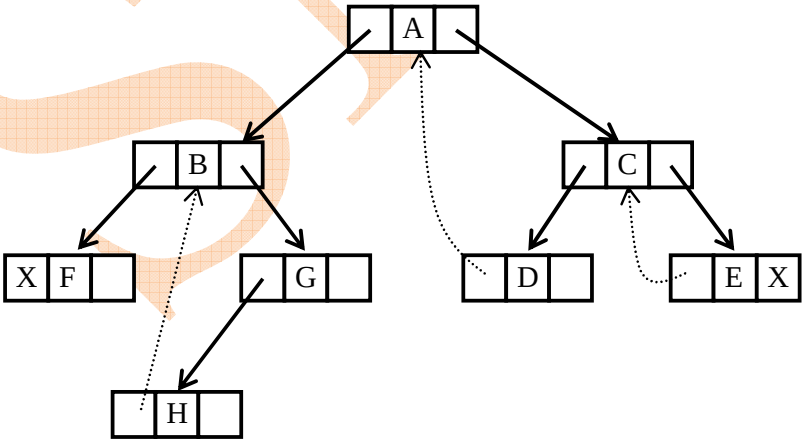
Threaded Binary Tree दो प्रकार के होते हैं

(1) One-way Threaded Binary Tree
(2) Two-way Threaded Binary Tree

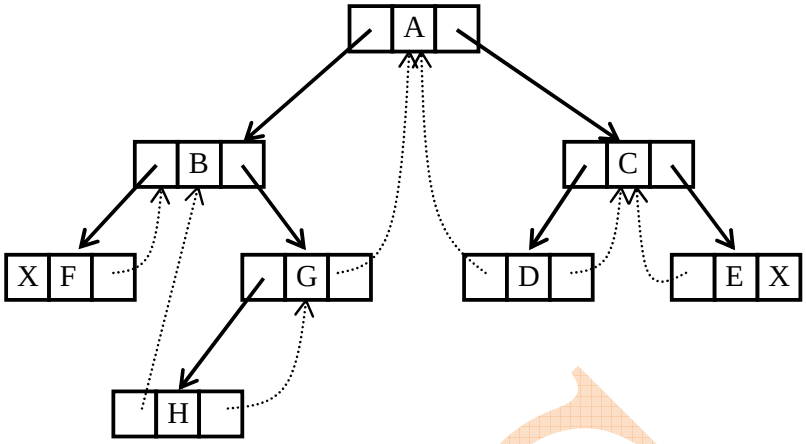
One-way Threading में केवल उस **Node** से जोड़ देते हैं जो **traversal** में उसके बाद आती हैं।
Two-way Threading में **Node** को उन दोनों **Node** से जोड़ देत हैं जो **Traversal** में उसके दोनो ओर हैं।



Right Threaded Binary Tree (One-Way Threading)



Left Threaded Binary Tree (One-Way Threading)



Full Threaded Binary Tree (Two-Way Threading)

B Tree

यह एक विशेष प्रकार का **tree** है जिसके **root** में हम एक से अधिक **Node** रखकर उसमें से एक से अधिक **Branches** निकाल सकते हैं। इस प्रकार के **tree** को **M-Way tree** भी कहा जाता हैं यहां एक **Balanced M-way tree** को **B Tree** कहते हैं। इस **Tree** के **Node** एक से अधिक या बहुत सारे **Records** और **childrens** के **Pointers** को रख सकते हैं। **B Tree** को **Balanced Sort Tee** भी कहा जाता हैं। यह **Binary Tree** नहीं होता हैं। एक **B Tree** के निम्न गुण हैं :–

(1) प्रत्येक **Node** अधिक से अधिक **N Childrens** रख सकती हैं व कम से कम **N/2** या किसी अन्य संख्याओं के **Children** जो **2** और **n** के बीच कहीं पर है।
(2) हर **Node** में **children** से **1** कम की आ सकती हैं।
(3) **Node** में **Keys** को परिभाषित क्रम में व्यवस्थित किया जाता हैं।
(4) **Left Sub Tree** की सभी **Keys, Key** की **Predecessor** होती हैं।
(5) **Right Sub Tree** की सभी **Keys, Key** की **Successor** होती हैं।
(6) जब एक पूरे भरे हुए **Node** में कोई **Key** जोड़ी जाती हैं तो **Key** दो **Nodes** में टूट जाती हैं और वह **Key** जो बीच में होगी उसे **Parent Node** में रखा जाएगा तथा छोटी **Value** वाली **Key Parent** के **Left** में तथा बड़ी **Value** वाली **Key Parent** के **Right** में रहेगी।
(7) सभी **Leaves** एक ही **Level** पर होती हैं अतः **V** के **Level** के उपर कोई भी खाली **Sub Tree** नहीं होगा।

B Tree में जोड़ना :–

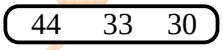
B-Tree में किसी भी **Key** को जोड़ने से पहले यह देखा जाता हैं कि वह **Key** कहां जोड़ी जानी है। यदि **Key** पहले से निर्मित **Node** में जा सकती है, तो **Key** को जोड़ना आसान है, वह **Node** को दो भागों **(Nodes)** में बांट देती हैं। वह **Key** जिसका मान माध्यमान **(**बीच का**)** होता है, वह **Node** में रहता है और इसके बाएं **child** में इस **Key** के दाएं स्थित समस्त **Keys** आ जाती हैं। **B-Tree** में **Insertion (**जोड़ना**)** निम्नलिखित उदाहरण से विस्तार से समझाया जा रहा है –

माना हमें निम्नलिखित **Keys** का **4 Order** का **Tree** बनाना है –

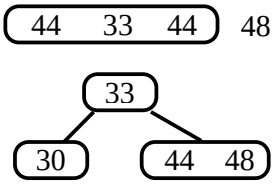
44 33 30 48 50 20 22 60 55 77

सबसे पहली **key 44** है जिसे हमें **tree** में जोड़ना है। चूंकि यह पहली **key** है इसलिए इस **Key** के द्वारा ही **Root Node** का निर्माण होगा –

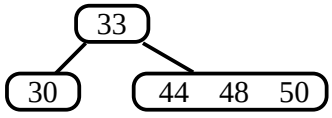
इसी प्रकार हम **33** और **30** को भी उनके स्थान के अनुरूप **Node** में जोड़ देंगे –



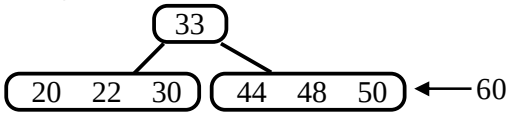
चूँकि हमने यह **4 order** का **tree** बनाया है इसीलिए इसकी एक **Node** में अधिकतम् **3 Keys** ही रख सकते हैं। अतः अब जो भी **key** इसमें जोड़ी जायेगी वह इस **(root) Node** को दो भागों में विभक्त कर देगी। यहां हमें **48** जोड़ना है–



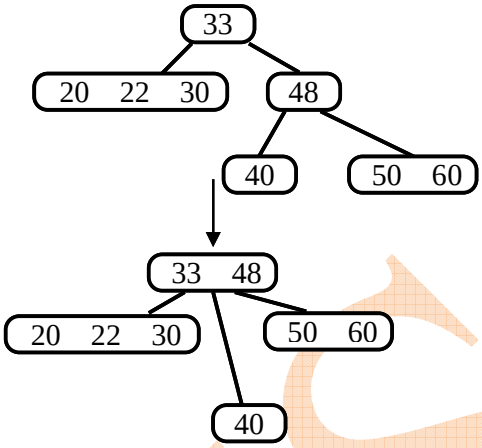
अब हम **50** को इसके स्थान के अनुरूप जोड़ देंगे –



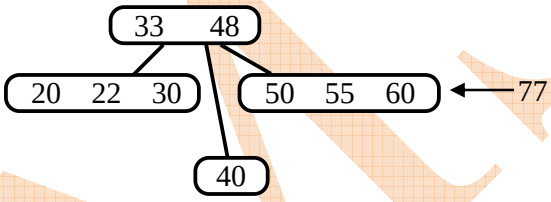
अब हम **20** और **22** को इसके स्थान के अनुरूप जोड़ देंगे –



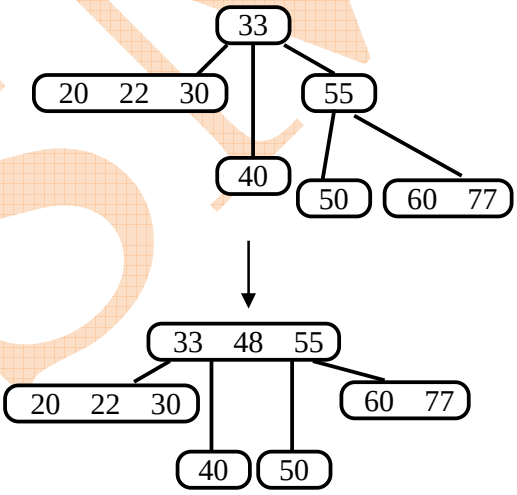
अब हमें **60** को जोड़ना है। जैसा कि हम देख सकते हैं **60, 50** के बाद जुड़ेगा और **50** वाली **Node** पूर्णतया भर चुकी हैं। इसीलिये **60** के जुड़ने पर यह **Node** दो भागों में विभक्त हो जाएगी और यह **Node** उन दोनों का **Parent Node** का मान **root node** में **merge** हो जाएगा –



अब हमें **55** को इसके स्थान के अनुरूप जोड़ देंगे –



अब हमें **77** को जोड़ना हैं। जैसा कि हम देख सकते हैं कि **77, 60** के बाद जुड़ेगा, अतः यहां पर भी **Node** विभक्त होगी और चूंकि अभी भी **Root Node** में स्थान रिक्त है इसलिए इस **Node** का मान **Root** में चला जाएगा –

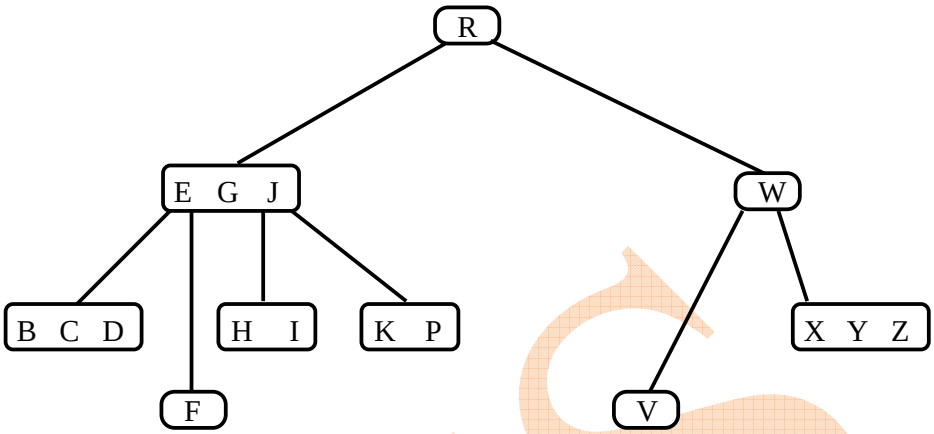


अब यह **Tree Belanced** है, और इस प्रकार बनाया गया है कि इसकी समस्त **Leaves** एक ही **Level** पर हैं।

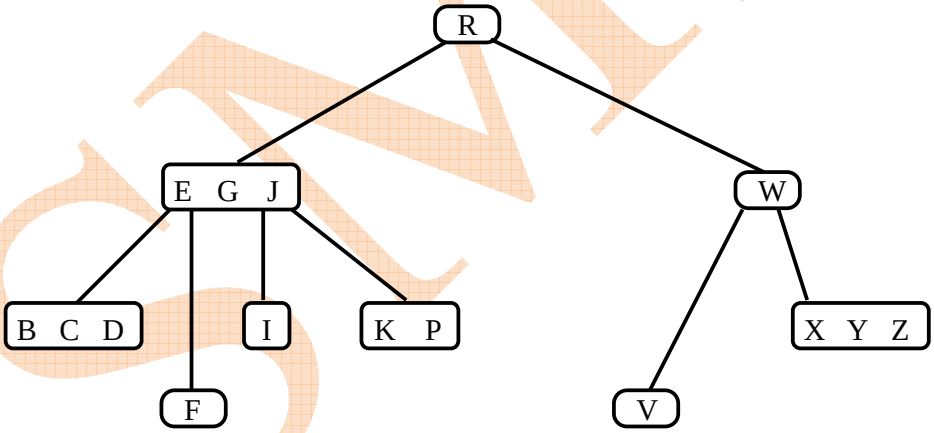
B – tree में से मिटाना

जैसा कि हम जोड़ने में पहले **key** तथा उसका स्थान देखते हैं उसी प्रकार मिटाने में भी पहले **Key** तथा उसके स्थान को ढूंढते हैं।

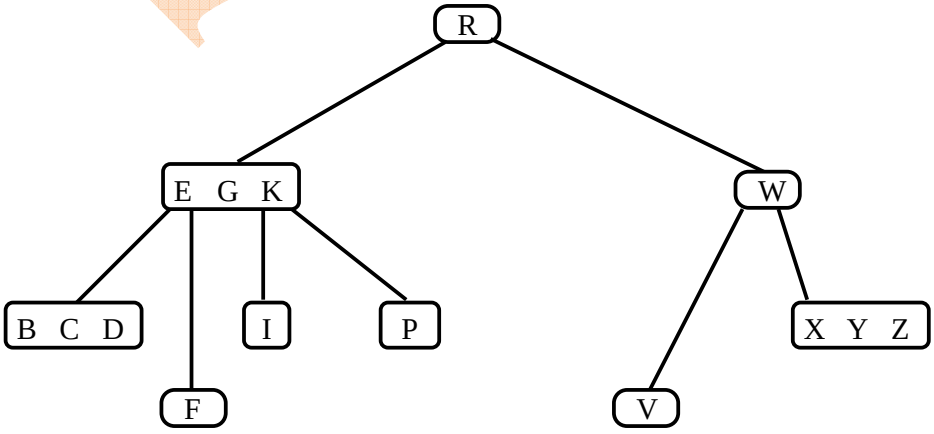
यदि मिटाई जाने वली **key** एक **terminal (leaf)** है, जो मिटाना सरल है, अर्थात् केवल उस **key** को **Node** में से मिटा देना होगा।
यदि मिटाई जाने वाली **key** एक **terminal (leaf)** नहीं है ता, इसे इसके **successor** के मान से बदल दिया जाता हैं। इस प्रकार यदि **successor, terminal Node** में हैं, तो वहां से वह मिटा दिया जाएगा।



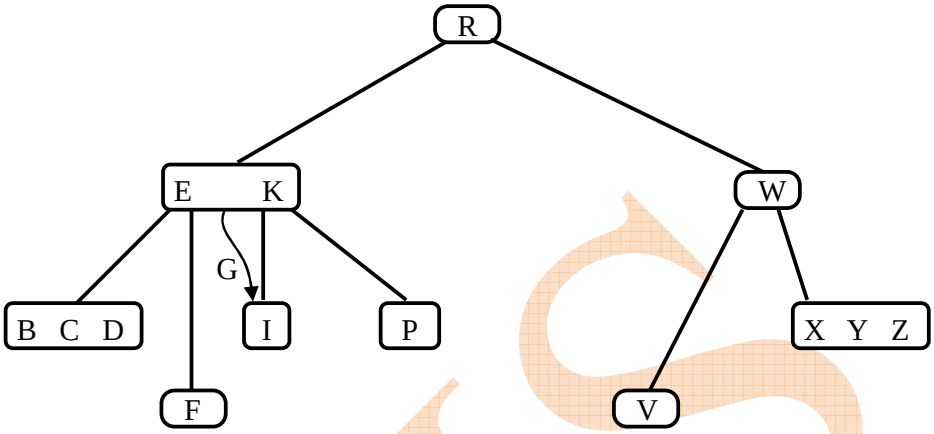
इस प्रकार यदि **sucessor, non-terminal node** में है, तो उसका **sucessor, replace** होगा और यही सारी शर्तें इस **sucessor** के साथ भी लगेंगी। अतः यह देखा जाए तो प्रत्येक प्रकार के **deletion** में **terminal node** में से एक **key** अवश्य मिटेगी।
यदि हम '**H**' को **Delete** करते हैं तो बहुत आसान है अर्थात् केवल '**H**' को **Node** में से **Delete** कर देंगे –

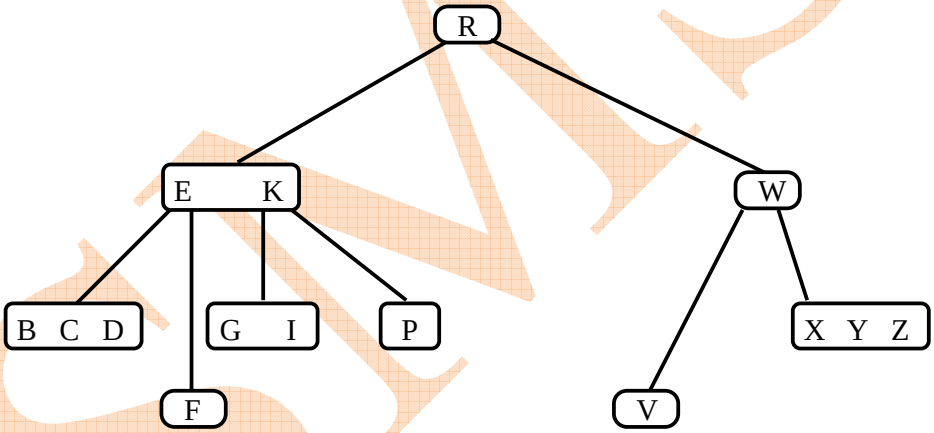


यदि हम '**J**' को मिटाते हैं तो हमें इसके **sucessor, K** को इसके स्थान पर कॉपी करना होगा –



यदि हम 'F' को मिटाते हैं, तो यहां **underflow** की स्थिति उत्पन्न हो जाएगी। देखा जाए तो यहां **F** के खत्म होते ही **G** का एक **child** खत्म हो जायेगा। अतः **G** को **remove** करके **E** के **child node** में भेज दिया जाएगा, तो इसमें निम्न स्थिति उत्पन्न हो जाएगी –





इस प्रकार दिए गए क्रम का **B-Tree** बनाया जा सकता है और किसी **B-Tree** में से किसी वांछित **key** को मिटाया भी जा सकता हैं।

AVL Tree (Height Balanced Tree)

एक **Binary Tree,** जिसकी **height, h** है, पूर्ण रूप से सन्तुलित **(balanced)** तभी होता है यदि उसकी समस्त **leaves** या तो **level** या फिर **h-1 level** पर हैं और यदि **h-1 level** से कम **level** पर स्थित समस्त **Nodes** के दो **children** हैं।

हम यह भी कह सकते हैं कि यदि किसी **Binary Tree** की प्रत्येक **Node** की **left** से सबसे लम्बे **Path** की **Length, right** में सबसे लम्बे **Path** की **length** के लगभग बराबर होती है, तो वह **Binary Tree** एक **Balanced Tree** कहलाता हैं।

यदि किसी **Binary Tree** की प्रत्येक **Node** के लिए **Left Sub – Tree** की **height** और **Right Sub-Tree** की **height** में यदि केवल एक का अन्तर ही है, तो ऐसा **Binary Tree Height Balanced Tree** कहलाता हैं।

एक लगभग **Height Balance Tree** को **AVL Tree** कहा जाता हैं।

Height Balance Tree का निर्माण

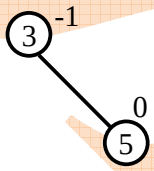
Height Balance Tree के लिए प्रत्येक **Node** का यह गुण होता हैं कि इसके **Left Sub-Tree** की **Height Right Sub-Tree** की **Height** से या तो **1** ज्यादा या बराबर या फिर **1** कम होती हैं। इसे **Balancing Factor (bf)** भी कह सकते हैं।

यदि दोनों **Sub – Trees** बराबर है तो, **bf=0**
यदि **Right Sub-Tree** की **height** बड़ी है, तो **bf = -1 (H (left)<H(Right))**
यदि **Left Sub-Tree** की **Height** बड़ी है तो **vf = +1 (H(left)>H(Right))**
अर्थात् **bf = left height – right height**
जब **Balancing Factor – 2** होता है तो हम **tree** को वामावर्त **(Anti-Clockwise)** घुमा देते हैं और जब **Balancing Factor+2** होता हैं तो हम **tree** को दक्षिणावर्त **(Clockwise)** घुमा देते हैं।

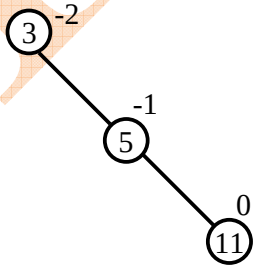
उदाहरण – **3 5 11 8 4 1 12 7 2 6 10**
को हमें **height balanced tree** बनाना हैं।

हम इस क्रिया को **root** अर्थात् **List** की पहली सूचना से प्रारम्भ करते हैं –

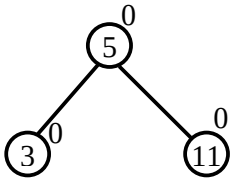
अब **Tree** में जोड़ने के लिए **5** है। यह पहले **tree** के **right** में जाएगा –



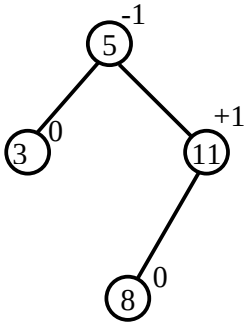
परिणामस्वरूप प्राप्त **tree** अभी भी **balanced** है। क्योंकि **bf = -1** है। अब हामरे पास **tree** में जोड़ने के लिए **11** है। यह **5** के **Right** में जाएगा–



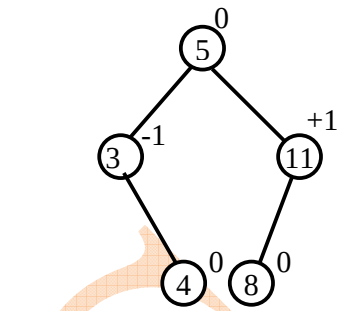
चूँकि **bf** का मान **–2** हो गया है, इसलिए हम **Tree** को **Anti-clockwise** घुमा देंगे –



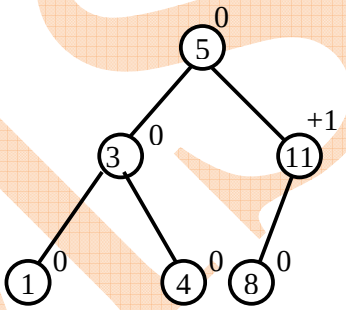
अब हमारे पास जोड़ने के लिए **8** है, यह **11** के **left** में जाएगा –



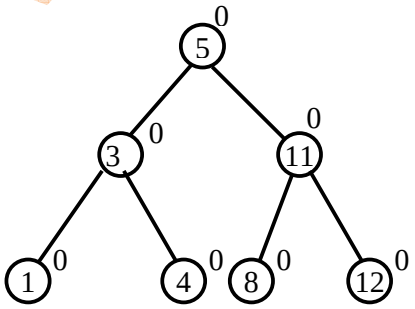
परिणमस्वरूप प्राप्त **tree** अभी भी **balanced** है। अब हमें **4** को जोड़ना है यह **3** के **right** में जाएगा –



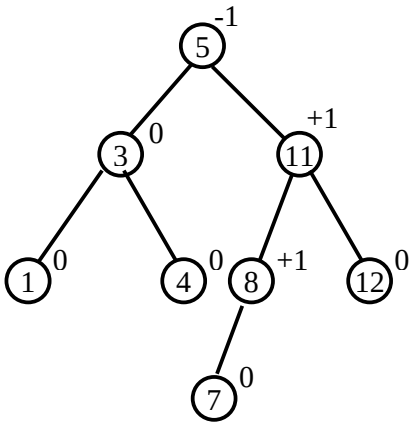
परिणामस्वयप प्राप्त **tree** अभी भी **balanced** है। अब हमें **1** जोड़ना है। यह **3** के **left** में जाएगा –



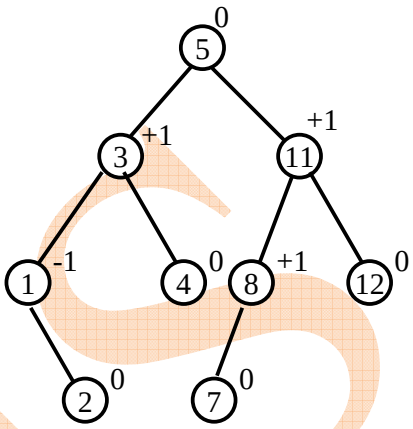
परिणामस्वरूप प्राप्त **tree** अभी भी **balanced** है। अब हमें **12** को जोड़ना है। यह **11** के **right** में जाएगा –



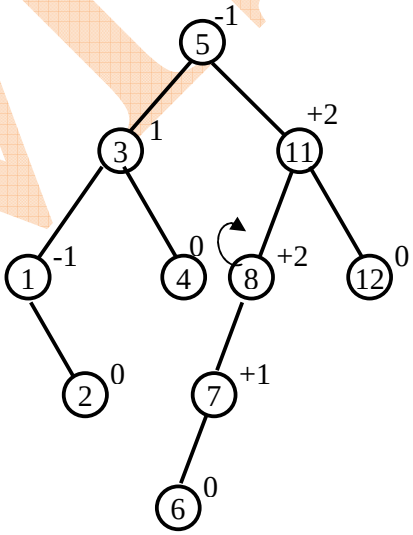
परिणामस्वरूप प्राप्त **tree** अभी भी **balanced** है। अब हमें **7** को जोड़ना है। यह **8** के **left** में जाएगा –



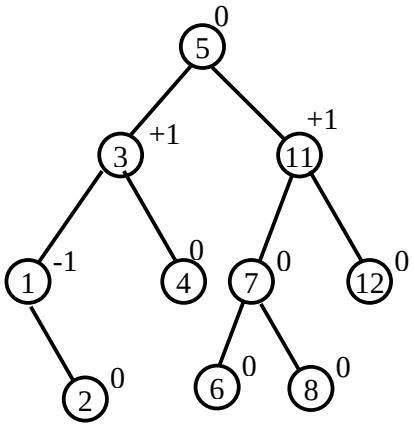
परिणामस्वरूप प्राप्त **tree** अभी भी **balanced** है। अब हमें **2** को जोड़ना है। यह **1** के **right** में जाएगा –



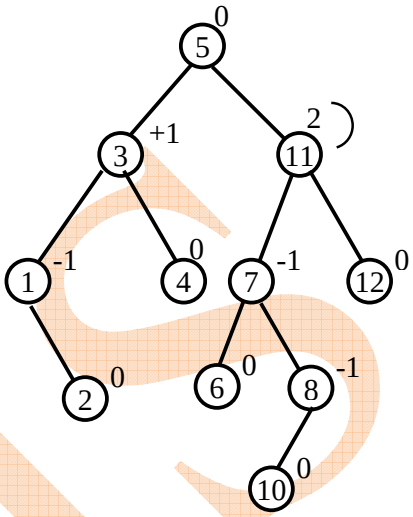
परिणामस्वरूप प्राप्त **tree** अभी भी **balanced** है। अब हमें **6** को जोड़ना है। यह **7** के **Left** में जाएगा –



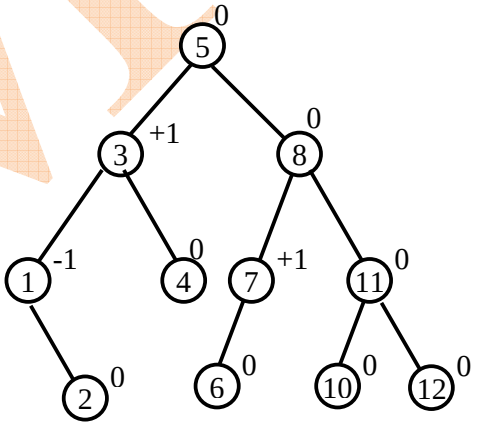
चूँकि **bf** का मान **2** हो गया है इसलिये हम **tree** को **clockwise** घुमा देंगे –



परिणामस्वरूप प्राप्त **tree** अब **balanced** है। अब हमें **10** को जोड़ना है। यह **8** के **right** में जाएगा –



चूँकि **bf** का मान **2** हो गया है इसीलिए हमें **tree** को **clock wise** घुमाना होगा। यदि हम **7** को **Promote** करते हैं, तो **7** के **right** वाला **sub tree** काफी लम्बा हो जाएगा और हमें अनेक बार **balancing** करनी पड़ेगी। इसीलिए हम **8** को **promote** करेंगे –



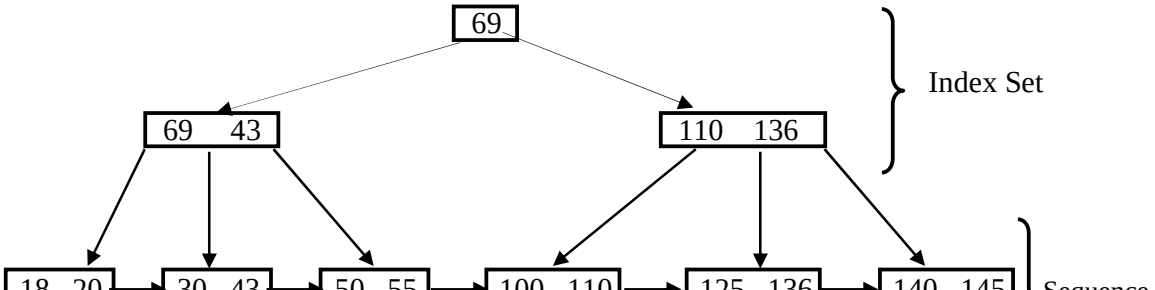
इस प्रकार प्राप्त **tree** ही अभीष्ट **Height Balanced Tree** है।

B+ Tree

B+ Tree Data Structure B Tree का **Extension** है। इसे **Index File Organization** की **Technique** को **Implement** करने के लिए **Use** में लिया जाता है। **B+ Tree** के **2 Parts** होते हैं।

(1) Index Set
(2) Sequence Set

B+ Tree के अर्न्तगत **Leaf Node** का **Right Pointer Next Leaf Node** को **Point** करता हैं।



Insert into B+ Tree: - B Tree की तरह ही **B+ Tree** में भी नई **Value** को **Insert** किया जाता हैं जब **Leaf Node** को **2 Nodes** में **Split** कर देते हैं।
Delition from B+ Tree: - B+ Tree के अर्न्तगत किसी भी **Key** को **Delete** करना **B Tree** से आसान है। जब किसी भी **Key Value** को **Leaf** से **Delete** किया जाता हैं तो **Index Set** में से किसी **Key** को **Delete** करने की आवश्यकता नहीं होती हैं।

B* Tree :– इस **Data Structure** को **Knuth** द्वारा बनाया गया था इसको बनाने का मुख्य उद्देश्य **Insertion** और **Deletion** में होने वाले **Overhead** को **Reduce** करना है। एक अन्य **Primary** उद्देश्य **Search Performance** को **Improve** करना भी था **B* Tree** एक **B Tree** है जिसमें हर **Node Half Full** न होकर **2/3 Full** होता हैं। **B* Tree** के द्वारा **Node Splitting** को **Reduce** किया जाता हैं। जब **Node Full** हो जाते हैं तो उन्हें **Split** करने से पहले **Re-Distribution Scheme** को **Use** में लेते हैं। **Insertion** तथा **Deletion Operation B Tree** की तरह ही होते हैं।

UNIT – IV

Sorting: - दिये गये **elements** को **Ascending (**आरोही**)** या **descending (**अवरोही**) Order** में जमाना **sortring** कहलाता हैं।
Searching: - दिये गये **Item** में से किसी एक **Item** को **Findout** करना **Searching** कहालाता हैं।
Sorting और **Searching** सामान्यतः **file** के **records** पर **Perform** की जाती हैं। इसलीए कुछ **Standard Terminology** को **Use** में लेते हैं।

Sorting

Sotring एक **Important Operation** है और **Normally** इसके लिए बहुत सारी **Applications** को **Use** में लिया जाता हैं। **Sorting** प्रायः एक **List Of Elements** पर **Perform** की जाती हैं। ऐसी **List** की **Sorting** को **Internal Sorting** कहते हैं जिसमें **List** के छोटेःछोटे भागों को **Sort** किया जाता हैं।
List पूर्ण रूप से **Computer** की **Primary Memory** में रहती हैं।
इसके विपरीत जब **Sorting** को **Secondary Memory** में **Perform** किया जाता हैं तो इस प्रकार की **Sorting External Sorting** कहते हैं।

Classification of Sorting Methods: - Sorting का सबसे **Easy** तरीका सबसे छोटी **Key** के साथ एक **Element** को चुनना होता हैं चुना गया **Element Array** से अलग हो जाता हैं। और बचे हुए **element** में से **Again** छोटे **Element** को चुना जाता हैं। इस तरीके को **Choosing Element With Sorting** कहा जाता हैं। इस तरह हम कई **Sorting Family** जैसे **Sorting With Merging, Sorting with insertion** व **Sorting with exchange** को में में लेते हैं। इन सभी तरीकों को **Comparative Sort** कहा जाता हैं। **Sorting** एक **Alternate Approach** हैं। जिसमें **Elements** को **Arrays** में **Collect** किया जाता हैं।

Advantages of Sorting: -

(1) **Data Sensitiveness: -** कुछ **Sorting** की तरीको में **Sequence** का ध्यान रखा जाता हैं। इस तरीके के द्वारा **data** की **Sensitiveness** को प्रदर्शित किया जाता हैं। वहीं दूसरी ओर जो **Data Sensitive** नहीं होता हैं। वह **Working Type** का कुछ समय **Waste** कर देता हैं।

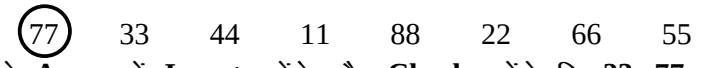
(2) **Stability: -** जब कोई भी प्रारम्भिक **List** जिसमें **Element** होते हैं। जो कि **Sequence** में नहीं होते हैं के द्वारा कोई भी **Sorting** को **Use** में लिया जा सकता हैं।

(3) **Storage Requirement: -** कुछ **Sorting** तरीके, **Original Array** के द्वारा **Re – Arrangement of Element** पर निर्भर करते हैं।

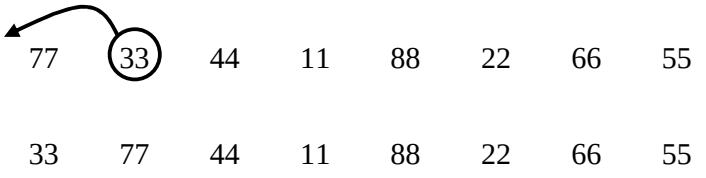
Insertion Sort: - इस **Sorting** में **Set of Value** से नहीं बल्कि उपलब्ध **Sort file** में **Elements** को **Fill** करते हैं। जैसे **Array** जिसमें **N Element A[1], A[2].... AN Memory** में है तो इसके लिए निम्न प्रकार के **Steps** को **Follow** किया जायेगा।
Pass1: A[1] पहले से **Sorted** है
Pass2 : A[2] को **A[1]** से पहले या बाद में **Insert** किया जायेगा इस प्रकार **Array** को **Sort** करेंगे।
Pass3 : A[3] का **A[1]** व **A[2]** से **Comparision** किया जायेगा यदि **A[3], A[1]** व **A[2]** से छोटा है तो **A[3]** को सबसे पहले **Insert** कर दिया जायेगा। यदि **A[3], A[1]** से बड़ा और **A[2]** से छोटा हो तो **A[3]** को **A[1]** और **A[2]** के बीच में **Insert** कर दिया जाएगा। यदि **A[3], A[1]** और **A[2]** दोनों से बड़ा है तो उसे दोनो के बाद **Insert** कर दिया जायेगा। अतः
Pass4 : इस प्रकार यह क्रम निरन्तर चलता रहेग।
Pass5 : इस प्रकार **Array** के **Elements** को **Proper Place** पर **Insert** कर दिया जायेगा।

Example: - Insertion Sort को समझने के लिए निम्न **Method** को **Use** में लेते हैं।
निम्न **Array** दिया गया हैं।
77 33 44 11 88 22 66 55
इस **Array** को **sort** करने के लिए निम्न **Steps** को **Follow** करेंगे।

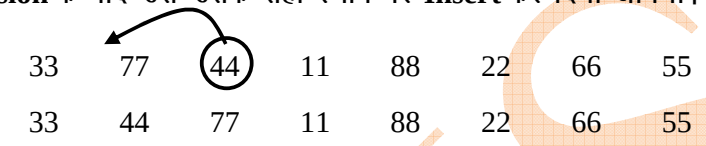
1. सबसे पहले **Array** में **77** को **Store** किया जाये।



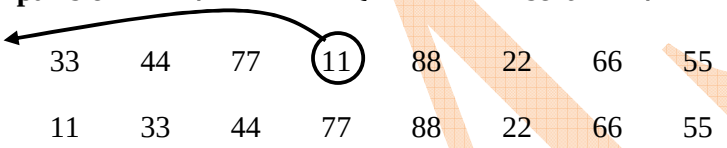
2. अब **33** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **33, 77** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा।



3. अब **44** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **44, 33** और **77** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा।

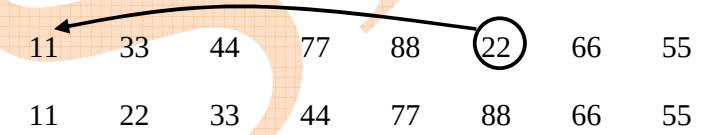


4. अब **11** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **11- 33, 77** और **44** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा।

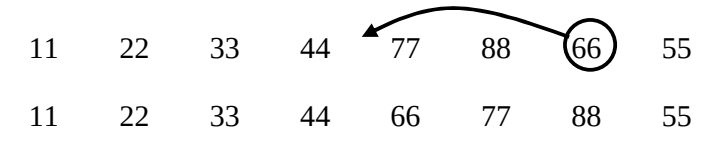


5. अब **88** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **88- 33, 77, 44** और **11** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा। **88 Array** में **Inserted** सभी **elements** से बड़ा है। अतः इसे **Array** के **Last** में **Insert** किया जायेगा।

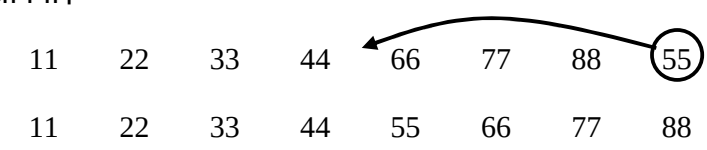
6. अब **22** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **22- 33, 77, 44, 11** और **88** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा। **22 Array** में **Inserted Value 11** से बड़ा और **33** से छोटा है अतः इसे **11** और **33** के बीच में **Insert** कर दिया जायेगा।



7. अब **66** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **66- 11, 22, 33, 44, 77** और **88** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा। **66 Array** में **Inserted Value 44** से बड़ा और **77** से छोटा है अतः इसे **44** और **77** के बीच में **Insert** कर दिया जायेगा।



8. अब **55** को **Array** में **Insert** करेंगे और **Check** करेंगे कि **55- 11, 22, 33, 44, 66, 77** और **88** से छोटा है या बड़ा और **comparision** के बाद उसे उसके सही स्थान पर **Insert** कर दिया जायेगा। **55 Array** में **Inserted Value 44** से बड़ा और **66** से छोटा है अतः इसे **44** और **66** के बीच में **Insert** कर दिया जायेगा।



Algorithm of Insertion Sort

Algorithm -

1. **Set K = 1**
2. **For k = 1 to (n-1)**
3. **Set temp=a (k)**
4. **Set j = (k-1)**

While temp<1 (j) and (j>=0) perform the following steps; set a(j+1) =a (j)
[End of loop structure]
Assign the value of temp to a (j+1)
[End of for loop structure]

5. **Exit**

Selection Sort: - ये **Technique Minimum Maximum Element** पर निर्भर करती हैं। इसके लिए सबसे पहले **Minimum Value** को **Select** किया जायेगा। और उसे **Array** की **First Position** पर रख दिया जायेगा। उसके बाद **2nd Minimu Element** को **find** करेंगे और उससे **2nd Position** से **Exchange** कर दिया जाएगा

Pass1 :Find the location LOC of the smallest in the list of elements A[1], A[2],....,A[N], and then interchange A[LOC] and A[1]. Then:

Pass2 :Find the location LOC of the smallest in the sublist of N – 1 elements A[2], A[3],..., A[N], and then interchange A[LOC] and A[2]. Then:
A[1], A[2] is sorted, since A[1] ≤ A[2].

Pass3 :Find the location LOC of the smallest in the sublist of N – 2 elements A[3], A[4],..., A[N], and then interchange A[LOC] and A[3]. Then:
A[1], A[2],...., A[3] is sorted, since A[2] ≤ A[3].

... .

... .

Pass N – 1 Find the location LOC of the smaller of the elements A[N-1], A[N], and then interchange A[LOC] and A[N-1]. Then:

Thus A is sorted after N – 1 passes.

Example: - Selection Sort को समझने के लिए निम्न **Method** को **Use** में लेते हैं।
निम्न **Array** दिया गया हैं।

77 33 44 11 88 22 66 55

इस **Array** को **sort** करने के लिए निम्न **Steps** को **Follow** करेंगे।

1. सबसे पहले **Array** के **Minimum Element** को **Findout** करेंगे।
2. **Array** का **Minimum Element 11** हैं। इसे **Array** के **0 Index** से **Replace** कर दिया जाएगा।

77	33	44	(11)	88	22	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	33	44	77	88	22	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

3. अब **a[1]** से **a[7]** तक में से **Minimum Element** को **Find Out** करेंगे। **Minimum Element 22** हैं उसे **a[1]** से **Replace** कर दिया जाएगा।

11	33	44	77	88	22	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	22	44	77	88	33	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

4. अब **a[2]** से **a[7]** तक के **Minimum Element** को **Find Out** करेंगे **Minimum Element 33** हैं। इसे **a[3]** से **Replace** कर दिया जायेगा।

11	22	44	77	88	33	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	22	33	77	88	44	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

5. अब **a[3]** से **a[7]** तक के **Minimum Element** को **Find Out** करेंगे **Minimum Element 44** हैं। इसे **a[3]** से **Replace** कर दिया जायेगा।

11	22	33	77	88	44	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	22	33	44	88	77	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

6. अब **a[4]** से **a[7]** तक के **Minimum Element** को **Find Out** करेंगे **Minimum Element 55** हैं। इसे **a[4]** से **Replace** कर दिया जायेगा।

11	22	33	44	88	77	66	55
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	22	33	44	55	77	66	88
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

7. अब **a[5]** से **a[7]** तक के **Minimum Element** को **Find Out** करेंगे **Minimum Element 66** हैं। इसे **a[5]** से **Replace** कर दिया जायेगा।

11	22	33	44	55	77	66	88
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

11	22	33	44	55	66	77	88
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

8. अब **a[6]** से **a[7]** तक के **Minimum Element** को **Find Out** करेंगे **Minimum Element 77** हैं। यह अपनी स्थान पर ही हैं। अतः इसे **Sort** करने की आवश्यकता नहीं हैं।

11	22	33	44	55	66	77	88
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

Algorithm of Selection Sort: - यह एल्गोरिथम आसानी से प्रारम्भ की जा सकती हैं।

जैसे – **1. Repeat steps 2 and 3 for k=1, 2 ..... n-1:**
2. Call min(a,k,n,loc).
3. [Interchange a[k])and a(loc)]
Set temp : a (k) a (k) : = a (loc) and a (loc) : = temp.
[End of step 1 loop]
4. Exit.

Heap Sort

Heap एक **Binart Tree** है, जिसकी **Parent Node** हमेशा **Child Node** से बड़ी होगी। **Heap Sorting** में एक दी हुई **List** को **Heap** में बदलते हुए इस प्रकार से **Arrange** करते हैं कि वह **List** स्वतः ही **Sort** हो जाए।

Heap Sort की तकनीक में हम **Element** को एक–एक करके **tree** में **Insert** कर देते हैं और यदि **Insert** किया गया **Element, Parent Node** से बड़ा है तो उस **Node** को **Parent Node** के साथ **swap** कर देते हैं। इस विधि को अच्छी तरह समझने के लिए हम एक उदाहरणकी सहायता लेंगे। नीचे एक **Unsorted List** के **Elements** को दर्शाया गया है –

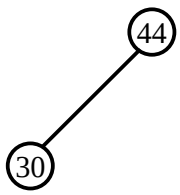
44 30 50 22 60 55 77

Heap Sorting के लिए **tree** बनाते हुए **Node** को **preorder** के रूप में **Insert** करते हैं अर्थात् पहली **Node** को **root** पर, दूसरी **Node** को **Left** में, तीसरी **Node** को **Right** में तथा इसी क्रम में आगे तक **Nodes** को **Tree** में **Insert** करते हैं।

Step 1 - **44** को हम **Tree** में **Insert** करेंगे, चूँकि **Tree** अभी तक ही नहीं बना है इसीलिये **44** ही **parent Node** बन जाएगी।

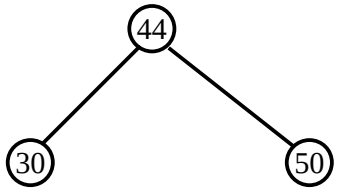
अतः अब **List –44**।

Step 2 - अब हमें **30** को **Heap Tree** में **Insert** करना है चूँकि हम यह पहले ही बता चुके हैं **Heap Tree** में **Insertion** करते हुए पहले **Root,** फिर **Left** में **Node** को **Insert** करते हैं। इसीलिये हम **30, 44** के **Left** में डालेंगे।

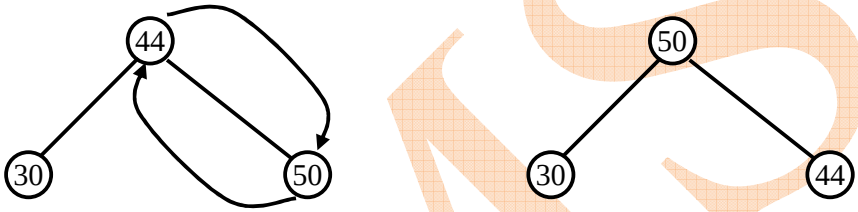


अतः अब **List – 44, 30**।

Step 3 - अब हमें **50** को **Heap Tree** में **Insert** करना है, **Insertion** के नियम के अनुसार **root** अर्थात् **44** के **right** में **Insert** करेंगे।

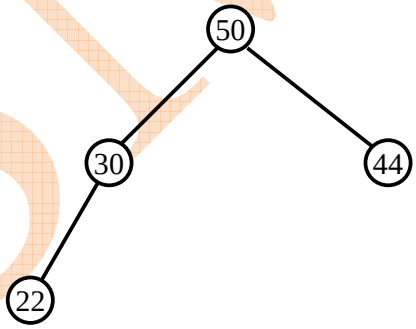


यहाँ पर हमने देखा कि उपरोक्त **Tree** के **Child Node** का मान **Parent Node** के मान से बड़ा है इसीलिए हम इन दोनों की **Swapping** कर देंगे।

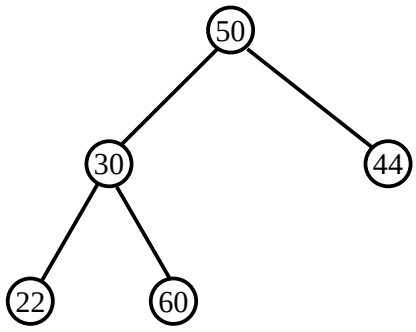


अतः अब **List – 50, 30, 44** ।

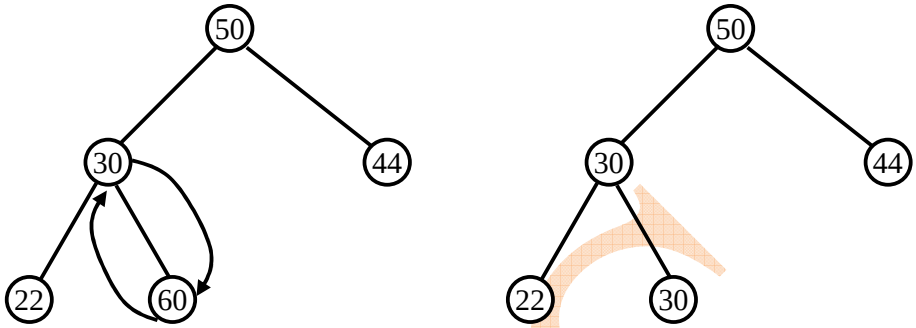
Step 4 - अब हमें प्राप्त **Tree** में **22** को **Insert** करना है। चूंकि **Root (50)** के दोनों ओर **(left** एवं **right** में**) Data Save** है इसीलिए हम **Root** के **Left child** को एक **sub-tree** का **root** मानकर इसके **Left** में **22** को **Insert** करेंगे।



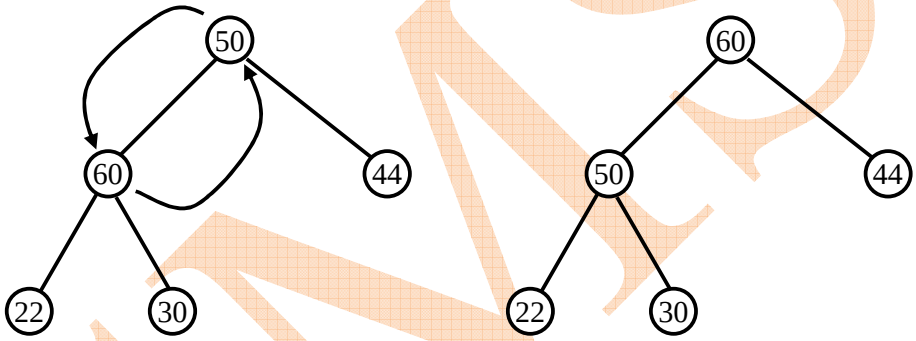
Step 5 - यहां पर हमें **60** को **Insert** करना है, चूंकि हम यहां **Root** के **Left Tree** में **Insertion** कर रहे हैं इसीलिए **60** को **30** के **Right** में **Save** करेंगे।



यहां पर हमने देखा **child node** का मान **parent Node** के मान से बड़ा हो गया है इसीलिए हम इन दोनों की **Swapping** कर देंगे।

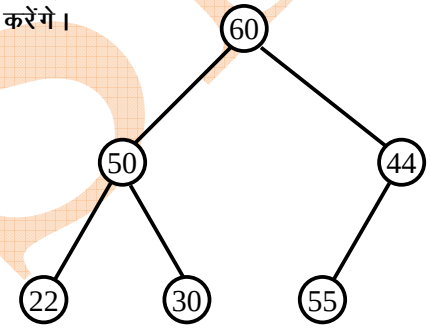


दोबारा हमने यहां देखा कि **child node (60)** का मान **Parent Node (50)** से बड़ा है इसीलिये हम इन दोनों की **Swapping** कर देंगे।

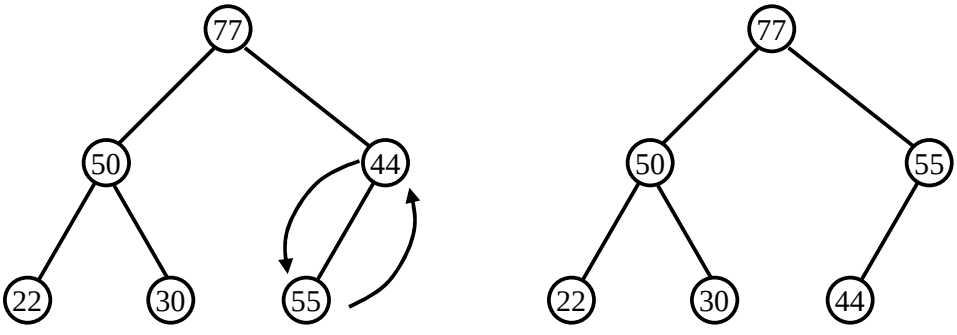


अतः अब **List – 60, 50, 44, 22, 30** ।

Step 6 - अब हमें **55** को **Insert** करना है, चूंकि हम **Root** के **Left sub-tree** के दोनों ओर **Insertion** का चुके हैं इसीलिए अब हम **Root** के **Right sub-tree** के **Left** में **Node** को **Insert** करेंगे।

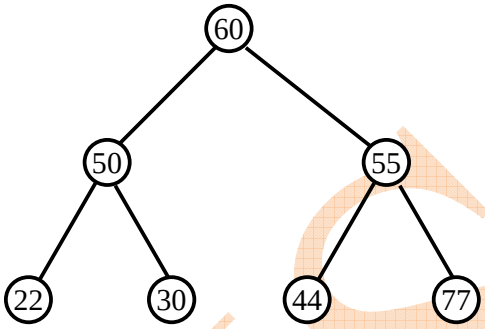


यहां पर हमने देखा **child node** का मान **Parent Node** के मान से बड़ा हो गया है इसीलिए हम इन दोनों की **Swapping** कर देंगे।

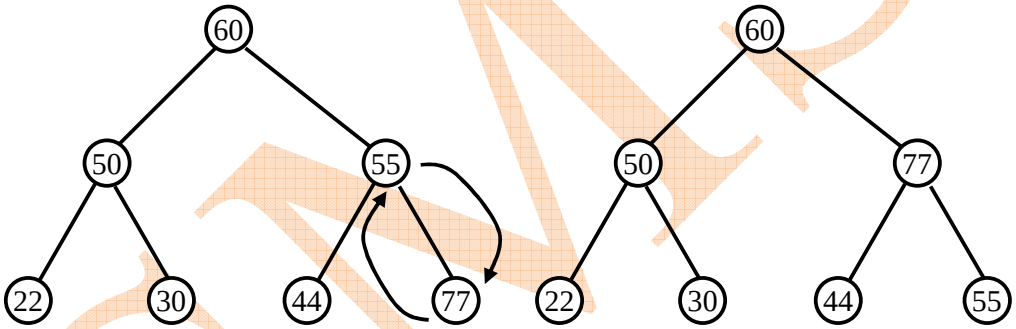


अतः अब **List – 60, 50, 55, 22, 30, 44।**

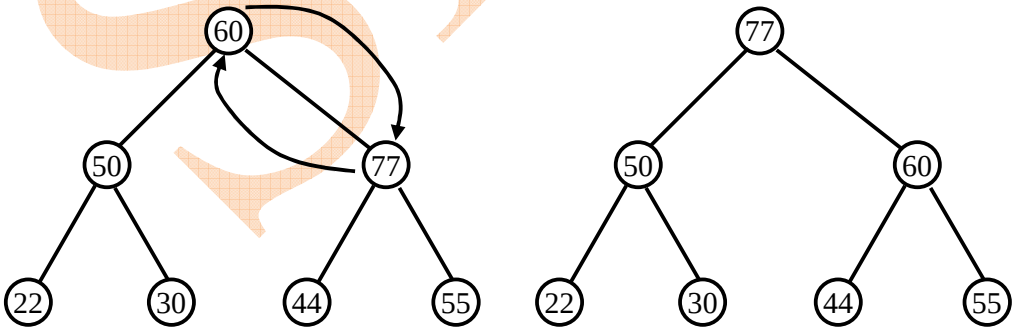
Step 7 - अब हमें **77** को **Insert** करना है, चूंकि हम **root** के **right sub-tree** के **Left** में **Insertion** कर चुके हैं इसीलिए अब हम **root** के **Right sub-tree** के **right** में **Node** को **Insert** करेंगे।



यहां पर हमने देखा **child node** का मान **Parent Node** के मान से बड़ा हो गया है इसीलिए हम इन दोनों की **Swapping** कर देंगे।



दोबारा हमने यहां देखा कि **Child Node (77)** का मान **Parent Node (60)** से बड़ा है इसीलिये हम इन दोनों की **Swapping** कर देंगे।



अतः अब **List – 77, 50, 60, 22, 30, 44, 55।**

अब, चूंकि **List** को **Increasing/Accending Order** में **sort** करना है, अतः हमें **List** के पहले **Element (77)** और अन्तिम **Element (55)** की **Swapping** करनी होगी।

अतः अब **List – 55, 50, 60, 22, 30, 44, 77।**

अब **77 Sort** हो चुका है, **List** के अन्य **element** को **Sort** करने के लिए **List** के इस **Sorted Element** को छोड़कर, अन्य **Unsorted Elements** पर उपरोक्त प्रक्रिया पुनः दोहराई जाती हैं।

Step 1- **set in R, root, par, son, temp** **(R, root, par, son व temp को initialize करें।)**

Step 2-	**Reapeat for k = n, k<=1**	**(k** का मान **n** सुरक्षित करें और **step 3** से **ste 15** तक दोहराएं जब तक **k** का मान **1** के बराबर या बड़ा हो।**)**
Step 3-	**call wapt with &A[1], &A[k]**	
	(A[1] व **A[k]** को **Swap** करने के लिए **function** को कॉल करें।**)**	
Step 4-	**set par = 1**	**(part** का मान **1** सुरक्षित करें।**)**
Step 5-	**set son = par x 2**	**(Son Variable** में **par** का दुगुना मान सुरक्षित करें।**)**
Step 6-	**set roo = A[1]**	**(root variable** में **A[1]** अर्थात् **array** का पहला **element** सुरक्षित करें।**]**
Step 7-	**check whether (son + 1) < l**	**(**जांचे कि क्या **(son + 1)** का मान **K** से छोटा है।**)**
	if yes then	**(**यदि हां तो
	check whether A[son+1] > A[son]	**(**जांचे कि क्या **A[son+1]** का मान **A]son]** से बड़ा है।**)**
	if yes then	
	son++	**(**यदि हां तो **son** में **1 Increment** करें।**)**
Step 8-	**Repeat while [son < = (k-1) and A[son] > root]**	
	(step 9 से **step 15** तक दोहराएं जब तक **son** का मान **k-1** से छोटा या बराबर है और **A[son]** का मान **root** से बड़ है।	
Step 9-	**set A[par] = A[son]**	**(A[par]** में **A[son]** का मान सुरक्षित करें।**)**
Step 10-	**set par = son**	**(par** में **son** का मान सुरक्षित करें।**)**
Step 11-	**set son = par x 2**	**(son** में **par** के दुगुने मान को सुरक्षित करें।**)**
Step 12-	**Check whether (son+1)<k**	**(**जांचे कि क्या **(son+1) k** से छोटा है।**)**
	if yes then	**(**यदि हां तो
	Perform Step 13	**Step 13** का अनुसरण करें
	else	अन्यथा
	Perform step 14	**Step 14** का अनुसरण करें।**)**
Step 13-	**check whether A[son+1] > A[son]**	
	(जांचे कि क्या **A[son+1]** का मान **A[son]** से बड़ा है।**)**	
	if yes then	
	son ++	**(**यदि हां तो **son** में एक का **increment** करें।**)**
Step 14-	**check whether son > n**	**(**जांचे कि क्या **son** का मान **n** से बड़ा है।**)**
	if yes then	
	set son = n	**(**यदि हां तो **son** में **n** सुरक्षित करें।**)**
	[End of If condition]	
Step 15-	**set A[par] = root**	**(A[par]** में **root** का मान सुरक्षित करें।**)**
	[End of loop of step 8]	
	[end of loop of step 2]	
Step 16-	**Exit**	

Bubbles Sort

Algorithm – माना कि **n Elements** का एक **Linear Array** है, **temp Element** की **Interchanging** के लिये अस्थायी **Variable** है –

1. **Input n elements of an array a.**
2. **Initialize i=0**
3. **Repeat through step 6 while (i<n)**
4. **Set i=0**
5. **Repeat through step 6 while (i<n-i-1)**
6. **If (a[i]>a(j+1)**
   - **(i) temp = a (j).**
   - **(ii) a [j] = a (j+1).**

(iii) a (j+1) = temp.

7. **Display the sorting elements of array a.**
8. **Exit.**

For Example:- The following array is given:-

इस प्रकार के **Array** में **adjacent number** को **compare** किया जाता है।

यदि a[i]>a[i+1]=exchange

if a[i]<a[i+1]=no change

11	15	2	13	6
a[0]	a[1]	a[2]	a[3]	a[4]

इसमें अलग अलग **passes** को **perform** किया जाता है। जब तक **array** के **last element** तक ना पहुँच जाए।

PASS 1:

1.दिए गए **Array a[0]** को **a[1]** से **compare** करवाऐंगें।

if a[0]>a[1]=exchange and if a[0]<a[1]=no change

11,15 से छोटा है।अतः **no change**

11 15 2 13 6

a[0]<a[1]=no change

2. दिए गए **Array a[1]** को **a[2]** से **compare** करवाऐंगें।

if a[1]>a[2]=exchange and if a[1]<a[2]=no change

15,2 से बड,ा है।अतः **exchange**

11 15 2 13 6

a[1]>a[2]=exchange

11 2 15 13 6

3. दिए गए **Array a[2]** को **a[3]** से **compare** करवाऐंगें।

if a[2]>a[3]=exchange and if a[2]<a[3]=no change

15,13 से बड,ा है।अतः **exchange**

11 2 15 13 6

a[2]>a[3]=exchange

11 2 13 15 6

4. दिए गए **Array a[3]** को **a[4]** से **compare** करवाऐंगें।

if a[3]>a[3]=exchange and if a[3]<a[4]=no change

15,6 से बड,ा है।अतः **exchange**

11 2 13 15 6

a[3]>a[4]=exchange

11 2 13 6 15

PASS 2:

Now the array is:

11	2	13	6	15
a[0]	a[1]	a[2]	a[3]	a[4]

1.दिए गए **Array a[0]** को **a[1]** से **compare** करवाऐंगें।

if a[0]>a[1]=exchange and if a[0]<a[1]=no change

11,2 से बड,ा है।अतः **exchange**

11 2 13 6 15

a[0]>a[1]=exchange

2 11 13 6 15

2. दिए गए Array a[1] को a[2] से compare करवाऐंगें।

if a[1]>a[2]=exchange and if a[1]<a[2]=no change

11,13 से छोटा है।अतः no change

2 11 13 6 15

a[1]<a[2]= no change

3. दिए गए Array a[2] को a[3] से compare करवाऐंगें।

if a[2]>a[3]=exchange and if a[2]<a[3]=no change

13,6 से बड,ा है।अतः exchange

2 11 13 6 15

a[2]>a[3]=exchange

2 11 6 13 15

4. दिए गए Array a[3] को a[4] से compare करवाऐंगें।

if a[3]>a[4]=exchange and if a[3]<a[4]=no change

13,15 से छोटा है।अतः no change

2 11 6 13 15

a[3]<a[4]=no change

PASS 3:-

Now the array is:

2 11 6 13 15

a[0] a[1] a[2] a[3] a[4]

1.दिए गए Array a[0] को a[1] से compare करवाऐंगें।

if a[0]>a[1]=exchange and if a[0]<a[1]=no change

2,11 से छोटा है।अतः no change

2 11 6 13 15

a[0]<a[1]=no change

2. दिए गए Array a[1] को a[2] से compare करवाऐंगें।

if a[1]>a[2]=exchange and if a[1]<a[2]=no change

11,6 से बड,ा है।अतः exchange

2 11 6 13 15

a[1]>a[2]= exchange

2 6 11 13 15

3. दिए गए Array a[2] को a[3] से compare करवाऐंगें।

if a[2]>a[3]=exchange and if a[2]<a[3]=no change

11,13 से छोटा है।अतः no change

2 6 11 13 15

a[2]<a[3]=no change

4. दिए गए Array a[3] को a[4] से compare करवाऐंगें।

if a[3]>a[4]=exchange and if a[3]<a[4]=no change

13,15 से छोटा है।अतः **no change**

2 6 11 13 15

a[3]<a[4]=no change

Now the Array is sorted.

QUICK SORT

(1) दिए गए **Array** की **left boundary** पर **left pointer** तथा **right boundary** पर **right pointer assign** करेंगें।

(2) Array के **1st element** को **pivot** मानेंगें और इसे इसकी सही **position** पर पहुँचाऐंगें। अतः **Pivot** के **left** में **Pivot** से छोटे **Element** और **Pivot** के **right** में **Pivot** से बडें **element**

(3) इसके लिए सबसे पहले **Pivot** और **left pointer** दोनो एक ही **element** को पोंइट करेंगे अब **left** का **comparision right pointer** से करेंगे

If (pivot or left is less than right then no change)

if (pivot or left is greater than right then exchange the element with pivot)

(4) यदि **Pivot Right** में पहुँच गया है तो **left** को **increase** करेंगें

if pivot or right is less then left then exchange

if pivot or right is greater then left then no change

for example :-

25 10 30 15 20 28

(1) दिए गयें **array** में **25** को **left** और **pivot** दोनों **point** करेंगें तथा **right pointer array** के **last element** को **point** करेंगा अब **pivot** को **right** से **compare** करेंगे यदि **pivot right** से छोटा है तो **no change** नही तो **element** को **exchange** कर दिया जायेगा

25 10 30 15 20 28

Pivot
left

Pivot or left < right = no change

right

(2) right pointer को आगे बढायेंगे और उसे **pivot** से **compare** करेंगे

If (pivot or left is less than right then no change)

if (pivot or left is greater than right then exchange the element with pivot)

25 10 30 15 20 28

Pivot
left

right

20 10 30 15 25 28

left

Pivot
right

(3) क्योंकि **pivot right side** में आ चुका है अतः **pivot** के **left** में **pivot** सें छोटी **values** आनी चाहिए अतः अब **left pointer** को आगे बढायेंगे ।

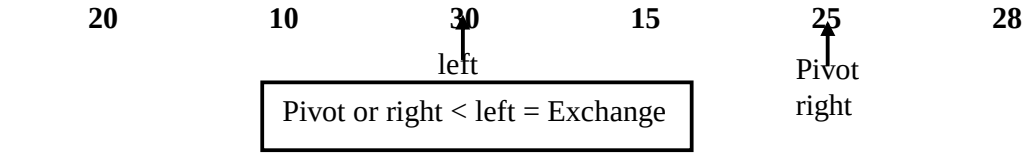
20 10 30 15 25 28

left

Pivot or right > left = Nochange

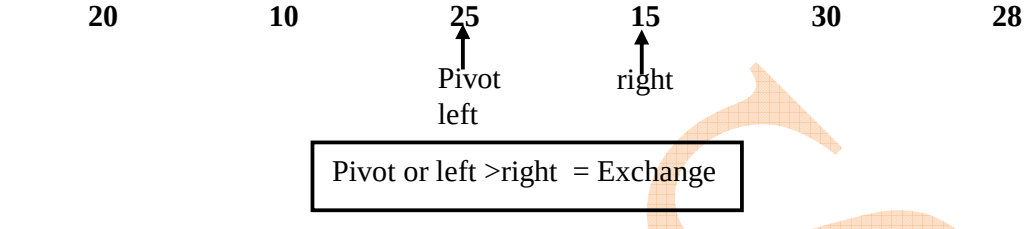
Pivot
right

(4) क्योंकि **pivot right side** में आ चुका है अतः **pivot** के **left** में **pivot** सें छोटी **values** आनी चाहिए अतः अब **left pointer** को आगे बढायेंगे ।



20 10 25 15 30 28

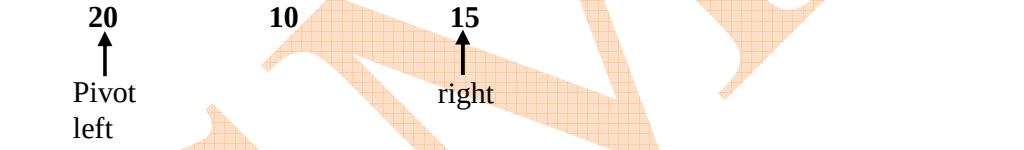
(5) क्योंकि **pivot left side** में आ चुका है अतः **pivot** के **right** में **pivot** सें बडी **values** आनी चाहिए अतः अब **right pointer** को आगे बढायेंगे ।



20 10 15 25 30 28

(6) अतः अब **pivot** अपनी सही **position** पर आ चुका है । **pivot** के **left** में सारी **pivot** से छोटी **values** है और **pivot** के **right** में **pivot** से बडी **values** है ।अतः अब **array** को तीन **parts** में **divide** कर देंगे । **first array** में **pivot** से छोटी **values** को रखेंगे **second array** में **pivot** को रखेंगे तथा **third array** मे **pivot** से बडी **values** को रखा जायेगा । अब **first** और **third array** की अलग अलग **sorting** करेंगे और उसे बाद में **pivot** के साथ **add** कर दिया जायेगा ।

(7) **pivot** से छोटे **elements** के **array** को **sort** करेंगे । **Pivot** से छोटे **elements** का **array** निम्न है :–



(8) उपरोक्त **array** मे 20 को **pivot** माना है अब **Pivot** और **Rght Pointer** को **Compare** करेंगे। यदि **Right Pointer Pivot** से छोटा है तो **Right** वाले **Element** को **Pivot** से **Exchange** कर दिया जायेगा।

Pivot, right Pointer से बड़ा है अतः इसे **Exchange** करेंगे।

15 10 20
↑ ↑
Left Pivot
Right

(9) इस प्रकार **Pivot Right** में आ चुका हैं। और **Right** के **Left** वाली सभी **values right** से छोटी होनी चाहिए अतः **Left** को आगे बढ़ाएगें।

15 10 20
↑ ↑
Left Pivot
Right

Left Pointer वाला **Element Pivot** से छोटा हैं। अतः **No Exchange** अब इसे **Pivot** वाले **Array** में **Add** कर देंगे। क्योंकि जब **Left** को आगे बढ़ाएगें तो **Left Pivot** और **right** तीनों एक ही **element** को **Point** करेंगे। अतः **Array Sorted** हैं।

15 10 20 25

(10) अब **Pivot** के **right side** वाले **Array** को **sort** करेंगे

30 28

(11) **30** को **Pivot** मानेंगे और **Left Pointer** भी **30** को **Point** करेगा। **Right Pointer 28** को **Point** करेगा।

30 28
↑ ↑
Pivot Right
Left

अब **Pivot** को **right Pointer** से **Compare** करेंगे। यदि **Pivot right** से बड़ा है तो उसे **exchange** कर दिया जाएगा।

28 30
↑ ↑
Left Pivot
Right

क्योंकि **Pivot Right** में आ चुका हैं अतः **Pivot** के **Left** में **Pivot** से छोटी **Values** होनी चाहिए अतः **Left** को आगे बढ़ाएगें।

28 30
↑
Left
Right
Pivot

अतः **Array Sorted** हैं अब इसे **Pivot** के साथ **Pivot** के **right** में **add** कर देंगे।

15 10 20 25 28 30

(12) इस प्रकार **quick sort** के द्वारा **Array** को **sort** किया जाता हैं

Algorithm For Quick Sort

Step 1 : **[Initially]**
Left = l
Right = R
Pivot = a[(l+r)/2]
Step 2 : **Repeat through step 7 while (left <= high)**
Step 3 : **Repeat step 4 while (Left<=Right)**
Step 4 : **Left = Left + 1**
Step 5 : **Repeat step 6 while (a {[Right] < pivot})**
Step 6 : **Right = Right – l**
Step 7 : **If (left < = Right)**
(i) **Temp = a (left)**
(ii) **a(left) = (Right)**
(iii) **a (Right) = temp**
(iv) **left = left +1**
(v) **Right = Right + 1**
Step 8 : **If (l, right) quick_sort (a, l , right)**
Step 9 : **If (left<R) quick_sort (a, left, r)**
Step 10: **Exit**

Searching

Searching का अर्थ है ढूंढना। **Searching** द्वारा किसी भी **List** में सुरक्षित किसी **element** को ढूंढा जा सकता हैं। **Searching** किसी भी **List** अर्थात् कुछ **Elements** का समुह हैं जो **Link** के द्वारा जुड़े रहते हैं में से किसी **Element** की **Location** को ढूंढने के लिए की जाती हैं। **Searching** का **Successful** या **Unsuccessful** होना पूर्णतः **Element** के मिलने या न मिलने पर निर्भर करता हैं अर्थात् यदि **element list** में मिल जाता हैं तो **Searching successful** कहलाती हैं और यदि **element list** में नहीं मिलता हैं तो **Searching Unsuccessful** कहलाती हैं **Searching** की **algorithm list** में **stored elements** के **Data Structure** के **type** पर निर्भर करती हैं। अतः यदि एक **array** में **searching** करते हैं तो उसकी **algorithm** अलग होगी और यदि एक **List** में **Searching** करते हैं तो उसकी **algorithm** अलग होगी। **Fast Searching** के लिए दो **Searching methods** को **Use** में लिया जाता हैं।
(1) Linear या **Sequential Search** **(2) Binary Search**

(1) Linear या **Sequential Search: -** यह सबसे **Easiest Searching Method** हैं **Sequential Search** की तुलना **single key** के उपर की जाती हैं। **Linear Search** में हम क्रम से एक–एक करके **Array** के हर **Element** तक पहुँचते हैं और देखते हैं कि वह **element** दिया गया **element** हैं या नहीं यदि **element** मिल जाता हैं तो वह **Successful Search** कहलाएगी और यदि **Element** नहीं मिलता हैं तो वह **Unsuccessful Search** कहलाएगी। यह एक ऐसी तकनीक हैं जो दिए गए **Item** को **Locate** करने के लिए **Squentially Array** को **Traverse** करता हैं।
Effeciency of Sequential Search: - इसमें दिया गया समय अथवा **comparision** के **No. Searching** में **Record** बनाते हैं जो **Technique** की **Capacity** पर आधारित होता हैं यदि दिया गया **element first position** पर हैं तो केवल एक **Search Table** का **Comparision** बनेगा। यदि दिया गया **Element Last** में है तो **N Comparisions** होंगे और यदि **element array** के बीच में कहीं पर हैं तो **(N+1)/2 comparision** होंगे। इसके लिए निम्न **algorithm** हैं
Algorithm - यह **Algorithm a** में **Item** के **Loc, Location** को प्राप्त करता हैं, या **Sets loc = 0** यदि **Search** असफल हैं।

1. **[insert item at the end of data] set data [n+1] = item**
2. **[initialize counter] set loc = 1.**
3. **[search for item]**
   Repeat white data [loc] = item;
   set loc = loc+1.
4. **[successfull] if loc = n+1, then set loc = 0**
5. **Exit.**

Binary Search: - यह **Data Structure Sorted Array** और **Link List** दोनों का **Combination** हैं इस **Search** के कुछ **Important Points** निम्न हैं।

(1) Bianry Search Recursive हैं। यह निर्धारित करता हैं कि **Search Key Array** के नीचे या उपर वाले आधे भाग पर स्थित हैं या नहीं
(2) यहां एक **Termination Condition** हैं
   (i) यदि **Low > high** तब **partition** यह **Search** करता हैं कि कोई **element** इसमें नहीं हैं।
   (ii) यदि यहां पर **Current Partition** के मध्य में **Element** के साथ एक **Match** हैं तो **Easily Element** से वापस जा सकते हैं।
(3) Array को तब तक **Search** करना जब तक कि सही **Point** द्वारा **Insert** करवाया गया नया **Item** प्राप्त न हो जाए।
(4) सभी **Items** को एक स्थान पर **Move** करना
(5) नये **Item** को खाली स्थान में **Insert** करना।
(6) Binary Search को **Static** घोषित किया जाता हैं यह एक **Local Function** हैं। जो सभी **Programs** के द्वारा **Access** किया जा सकता हैं।
(7) दिये गए **Element** को **Array** में **Search** करने के लिए निम्न **Steps** को **Follow** करेंगे।
   (i) **Beg Array** के **first element** को **Point** करेगा।
   (ii) **End Array** के **Last Element** को **Point** करेगा।

(iii) अब **Begning** और **end** वाले **Index** से **Middle Value** को **Find out** किया जाएगा।

(iv) अब **Check** करेंगे कि जिस **Item** को **Search** करना हैं वह **Middle Value** से बढ़ा हैं या छोटा।

(v) यदि दिया गया **Item Middle Value** से छोटा हैं तो उसे **Middle** से पहले वाले **Array** में **Search** करेंगे।

(vi) यदि दिया गया **Item Middle Value** से बड़ा हैं तो उसे **Middle** से बाद वाले **Array** में **Search** करेंगे।

इस प्रकार हर बार **Begning** और **End** को **Set** किया जाएगा और **Specific element** को **Find out** करेंगे।

Example: - एक **Sorted Array** दिया गया हैं जिसमें से **15 Element** को **Search** करना हैं

Array = 3, 10, 15, 20, 35, 40, 60

Solution: -

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
↑						↑
Beg						End

mid = (beg + end)/2

(0 + 6)/2 = 3

(1) अतः **mid a[3] Element** को **point** करेगा। अब यह **Check** करेंगे कि **a[3]** पर जो **element** हैं वह **15** से छोटा है या बड़ा

(i) यदि **a[3]** पर जो **Element** है वह **15** से छोटा है तो **15** के **right** वाले **array** में **search** करेंगे।

(ii) यदि **a[3]** पर जो **element** है वह **15** से बड़ा है तो **15** के **Left** वाले **Array** में **Search** करेंगे।

(2) **a[3]** पर **20** है जो **15** से बड़ा है अतः **15** को उसके **left** में **search** किया जाएगा इसके लिए **end** में **mid – 1** को **Assign** करेंगे। और पुनः **Begning** और **End** के **Middle Value** को **find** करेंगे।

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
↑		↑	↑			
Beg		End	mid			

mid = (beg + end)/2

(0 + 2)/2 = 1

(3) **a[1]** पर **10** है जो **15** से छोटा है अतः **15** को उसके **Right** में **search** किया जाएगा इसके लिए **Beg** में **mid + 1** को **Assign** करेंगे। और पुनः **Begning** और **End** के **Middle Value** को **find Out** करेंगे।

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
	↑	↑				
	Beg	End				

mid = (beg + end)/2

(1 + 2)/2 = 2

(4) **Begning** और **end** दोनो **a[2]** को **point** करेंगे अतः अब **check** करेंगे कि **15 a[2]** पर तो नहीं हैं **array** देखने से पता चलता हैं कि **a[2] position** पर **15** है। इस प्रकार **Binary Search Perform** किया जाएगा।

Algorithm for Binary Search

Begin

set beg = 0

```
set end = n – 1
set mid = (beg+end)/2
while (   (beg ≤ end) and (a[mid] # item )    ) do
        if (item < a [mid]) then
                set end = mid-1
        else
                set beg = mid + 1
        endif
        set mid = (beg + end) / 2
endwhile
if ( beg > end) then
        set loc = -1
else
        set loc = mid
        endif
End.
```

UNIT - V

Hash Table:- Memory में कुछ **Data** को **Store** करने के लिए बहुत से **Program** की आवश्यकता होती हैं। **Data Structure** को कई प्रकार के **Operations Perform** करने के लिए बनाया गया है। जैसे **Array, Linked List, Binary Search Tree** आदि के अर्न्तगत किसी भी **Element** को आसानी से **Add** या **Delete** किया जाता हैं। एक **Hash Table** उन **Target** को **Include** कर सकते हैं जिनके पास **Key** और **Value** है तथा जो **Objects** को भी **Include** करते हैं। इसमें **Hash Table** का एक **Array** बनाय जाता हैं जो कि **Hashing Function** होता हैं। **Hash Function** एक **Argument** के रूप में **Key** लेता हैं और **Table** के **Array** की **Range** में कुछ **Index** की **Calculation** करता हैं।

जब हम **Hash Table** में एक **element** को **Include** करना चाहते हैं तो सबसे पहले इसकी **Hash Value** की **Calculation** की जाती हैं। हम **Array** में **Location** को तब तक **Findout** करते हैं जब तक इसका **index hash value** के बराबर नहीं हो जाता और वहां **Item Insert** कर दिया जाता हैं। यदि **Location Table** में पहले से ही **use** में आ रही हैं तो पास वाली **Free Location** को **Check** करते हैं। इस प्रकार की **hashing simple hashing** कहलाती हैं।

जब **Array** में एक **Element** को प्रदर्शित करना चाहते हैं। तब सबसे पहले **Hash value** की गणना की जाती है और तब **Location** पर **Table** की **Checking Start** होती हैं जिनकी **Index** इस **hash value** के बराबर होती है।

यदि **Hash Table** से **element** को **Delete** करना चाहते हैं तो सबसे पहले उस **Element** की **Location** को **Find out** किया जाता हैं। जब **Element** मिल जाता हैं तो उस पर **Mark** लगा दिया जाता हैं कि **Location Free** हो।

यदि कोई **Element "C"** को **Findout** करने की कोशीश करते हैं तो सबसे पहले इसकी **Hash Value** की **Calculation** की जाती हैं। फिर इस **Location** में जाकर देखते हैं यह **Location 2** है। जब **C** मिल जाता हैं तो इसकी **Hash Value** की **calculation** करते हैं। यदि किसी **Element** को **Hash Table** से हटाया गया हो तो उसके स्थान पर एक **Mark** लगा देते हैं। जैसे निम्न चित्र में **B** को हटाया गया है तो उसे निम्न प्रकार प्रदर्शित किया जाता हैं।

		a	*	c	

Hash Table पर **Operations** को **Find** तथा **View** किया जा सकता हैं। यह दो **Factor** का **Function** है।

Hash Table की **complexity** (वास्तव में इस **Cell** में कितने **Percent Element** को **Include** किया गया हैं।) **Example: - Hash Function** हमेशा किसी भी **Element** के लिए **Return** होगा तो यह देखा जाता हैं कि एक **Link Lised** की तरह **Hash Table** के कार्य से मिला है जो **Operation** को ढूढने में ज्यादा **Important** नहीं हैं। **Key** को **Use** में लेने का **General Item** यह हैं कि किसी भी **Record** के **Address** को **Store** कर लिया जाये। लेकिन इसी **Modify** किया जाना चाहिये ताकि ज्यादा **Memory Waste** न हो इस **Modification** के लिये एक **Function H** को **K Set of Keys** पर **Define** किया जाता हैं। यह **Function** निम्न प्रकार हैं।

H : K → L

इस **Function** को **Hash Function** कहा जाता हैं हो सकता हैं कि दो **Keys K1** तथा **K2** दोनो **Same Hash Address** को **Point** करती हैं। इस प्रकार की **Condition** को **Collision** कहलाती हैं और इस **Collision** को दूर करने के लिए कुछ **Methods** को **Use** लिया जाता हैं। **Hashing** को **2 Parts** में **Divide** करते हैं।

(1) Hash Function
(2) Collision Resolution

(1) Hash Function: - Hash Function को **Select** करने के लिए **2 principles** को **Use** में लिया जाता हैं। एक **Function H** बहुत **Easy** तथा **Quick Computation** को **Perform** करता हैं। **Hash Function Hash Addresses** को **Uniformly distribute** करता हैं। जिससे कम से कम **Collision Occur** होते हैं।

(2) Collision Resolution: - माना कि हम एक नये **Record R** को **Key K** के साथ **File F** में **Add** करना चाहते हैं लेकिन **Memory Location** जहां पर नये **Record** को **Add** करना है वह

Full हैं तो इस प्रकार की **condition Collision** कहलाती हैं। **Collision** को दूर करने के लिए कुछ **Techniques** को **Use** में लिया जाता हैं।
Normally Collision को **Avoid** करना **Impossible** है जैसे एक **Class** में **24 Studnets** हैं और एक **Table** में **365 Records** का **Space** हैं। एक **Random Hash Function** के द्वारा **Student** के **Birthday** को **Choose** किया जाता हैं।

Hash Function की **Efficiency** को **Collision** के **Solution** के **Procedure** के आधार पर **Major** किया जाता हैं। इसके लिए **Key Comparision** की आवश्यकता होती हैं। जिससे **Specific Record** की **Location Find** किया जाता हैं। **Efficiency Load Factor** λ पर निर्भर करती हैं। **Collision Resolution** के लिए **2 Procedure** को **use** में लिया जाता हैं।

(1) Open Addressing: - माना एक नये **Record R** को **Key K** के साथ **Memory Table T** में **Add** करना है लेकिन **Memory Address** या **Memory Location With Hash Address** पहले से भरी हुई हैं। इस प्रकार के **Collision** को दूर करने के लिए **R** को **First Available Location** में **Assign** कर दिया जायेगा। इसके लिए सबसे पहले **Record R** को **Search** किया जायेगा इसके लिए **Linear Search** को **Use** में लेते हैं। इस प्रकार के **collision resolution** को **linear probing** कहा जाता हैं।
Example: - माना एक **Table T** में **11 Memory Locations** हैं। **T[1].... T[11]** तथा **File F** में **8 Records** हैं। **A, B, C, D, E, X, Y** तथा **Z** जिसके **Hash Address** निम्न हैं।

Record :	**A,**	**B,**	**C,**	**D,**	**E,**	**X,**	**Y,**	**Z**
H(k) :	**4,**	**8,**	**2,**	**11,**	**4,**	**11,**	**5,**	**1**

माना **8 Records** को **Table T** में उपरोक्त **Order** के आधार पर **Enter** कर दिया गया हैं। अब **File F** को **meory** में निम्न प्रकार देखा जा सकता हैं।

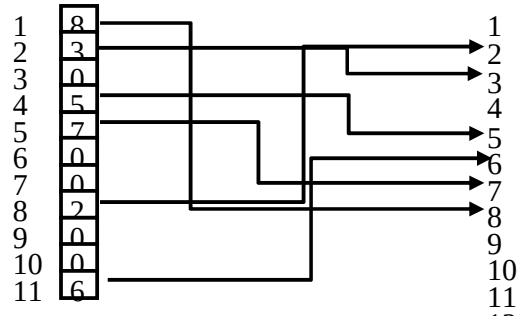
Table *T*:	**X,**	**C,**	**Z,**	**A,**	**E,**	**Y,**	**__,**	**B,**	**_,**	**_,**	**D**
Address:	**1,**	**2,**	**3,**	**4,**	**5,**	**6,**	**7,**	**8,**	**9,**	**10,**	**11**

Linear Probing का **Main Disadvantage** यह हैं कि **Records** का **Clusterization** हो जाता हैं। जब **Load Factor > 50%** हो **Clustering** किसी भी **Record** के **Average Search Time** को **Increase** कर देती हैं।

(2) Chaining: - Chaining Memory में **2 Tables** को **Maitain** करती हैं। एक **Table T Memory** में **F Records** को **Contain** करती हैं। **T** एक **Additional Field** को **Maintain** करता हैं जिसमें **Links** को रखा जाता हैं सभी **records** जो **Table T** में हैं **Same Hash Address H** के द्वारा एक दूसरे से **Linked** रहते हैं। **For Example: -** एक **Data Table T** में **8 Records** है जिनका **Hash Address** निम्न **Records** हैं।

Record :	**A,**	**B,**	**C,**	**D,**	**E,**	**X,**	**Y,**	**Z**
H(k) :	**4,**	**8,**	**2,**	**11,**	**4,**	**11,**	**5,**	**1**

Chaining को **Use** में लेकर **Records** को **Memory** में निम्न प्रकार से **Represent** किया जा सकता हैं।



GRAPHS

Graph भी एक **Non-Linear Data Structure** है। इस अध्याय में हम **Graph** के साथ किए जा सकने वाले सभी **Operations** के बारे में जानकारी प्राप्त करेंगे।

एक **G Graph Vertices (Nodes)** के **Set V** और **Edges** के **Set E,** से निर्मित होता है। इसे हम इस प्रकार भी कह सकते हैं कि **Vertices (Nodes)** का **Set V** और **edges** का **Set E** मिलकर एक **Graph** बनाते हैं। हम **Graph = (V,E)** लिखते हैं। **V, Nodes** का एक सीमित और भरा हुआ **Set** है और **E, Nodes** के जोड़ों का **Set** है, ये जोड़े ही **Edges** कहलाते हैं।

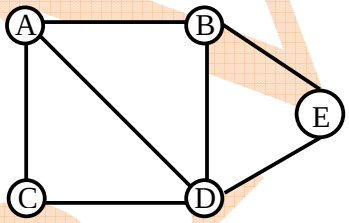
Graph को परिभाषित करना

V(G) को **Graph** की **Nodes** पढ़ा जाता है और **E(G)** को **Graph** की **Edge** पढ़ा जाताप है।

एक **Edge E=(V,W)** दो **Nodes, V** और **W** का एक जोड़ा है। जहां **V** और **W Incident** है।

इन दोंनो **Sets** को समझने के लिए निचं चित्र में दर्शाए **Graph** का अध्ययन करें।

इस चित्र में एक साधारण या दिशाहीन **Graph** को दर्शाया गया है, जिसके **Nodes** को क्रमशः **A, B, C, D** और **E** नाम दिया गया है इसलिए –



V(G) = (A, B, C, D, E)

E(G) = {(A,B), (B,D), (D,C), (C,A), (A,D) (B,E), (D,E)}

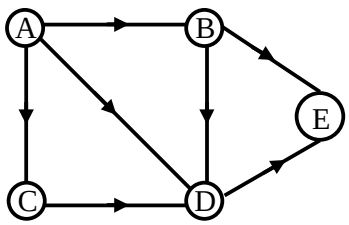
आप देख सकते हैं कि एक **Edge A** और **B** को जोड़ती है और हमने इसे **(A,B)** लिखा है, इसे **(B,A)** भी लिख जा सकता था। यही बात बाकी सभी **Edges** पर भी लागू होती है। इसीलिए हम कह सकते हैं कि चूंकि **Graph** में **Nodes** को क्रम नहीं दिया गया है, यह एक दिशाहीन ग्राफ के लिये सही है।

एक दिशाहीन **Graph** में **Nodes** को जोड़ा एक बिना क्रम की **Edge** को प्रस्तुत करता है। इसीलिए **(V,W)** और **(W,V)** दोनों एक ही **Edge** को प्रस्तुत करते हैं।

एक दिशा वाले **Graph** में प्रत्येक **Edge, Nodes** का एक क्रमिक जोड़ा होता है, अर्थात् प्रत्येक **edge** एक दिशा वाले जोड़े से प्रस्तुत होती है।

यदि **E = (V,W)** तो **Edge, V** से प्रारम्भ होगी और **W** पर समाप्त हो रही है। यहां **V** को **Tail** अथवा **Initial Vertex** तथा **W** को **Head** अथवा **Final Node Vertex** कहा जाएगा। अतः **(V,W)** और **(W,V)** दो भिन्न **Edges** को प्रदर्शित करेंगी।

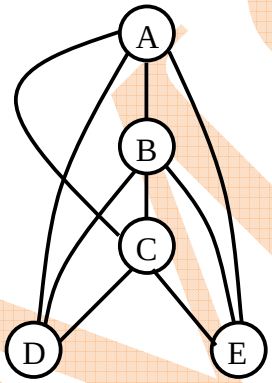
एक दिशा वाला अथवा निर्दिष्ट **Graph** निम्नांकित चित्र में दर्शाया गया है –



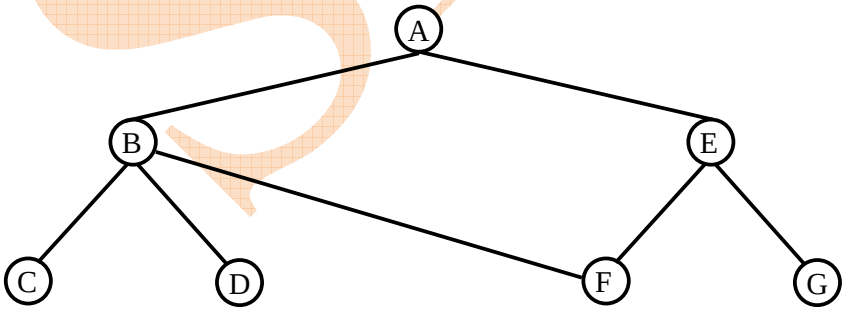
उपरोक्त चित्र में दर्शाए गए **Graph** के लिए –
V (G) = (A,B,C,D,E,)
E (G) = {(A,B), (A,C), (A,D), (C,D), (B,E), (B,D), (D,E)}

आधारभूत परिभाषिक शब्दावली
(Graph Terminology)

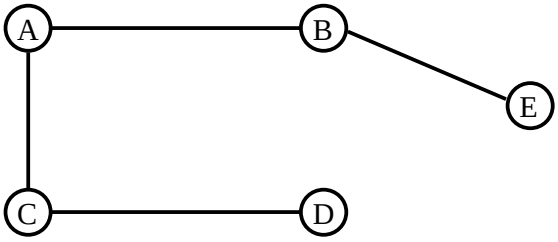
दिशाहीन और निर्दिष्ट **Graph** हम पहले ही बता चुके हैं, कुछ **Graphs** निम्नलिखित हैं –
एक **Graph G,** पूर्ण **Graph** कहा जाएगा, यदि **Graph** की प्रत्येक **Node Graph** की प्रत्येक दूसरी **Node** से जुड़ी हो। एक पूर्ण **Graph** जिसमें **n Vertices** हैं उसमें, **n(n-1)/2 Edges** होंगी। एक पूर्ण **Graph** को निम्नांकित चित्र में दर्शाया गया है –



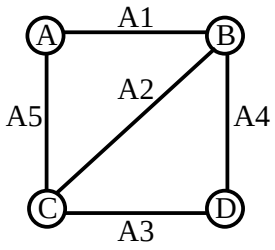
एक ट्री, जिसमें **cycle** बनी हो, उसे **Graph** कहते हैं। इसे इस प्रकार भी कह सकते हैं कि यदि एक **tree** की किसी **Node** को दो प्रकार से **accesds** किया जा सकता है, यह **tree** नहीं बल्कि **Graph** होता है। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –



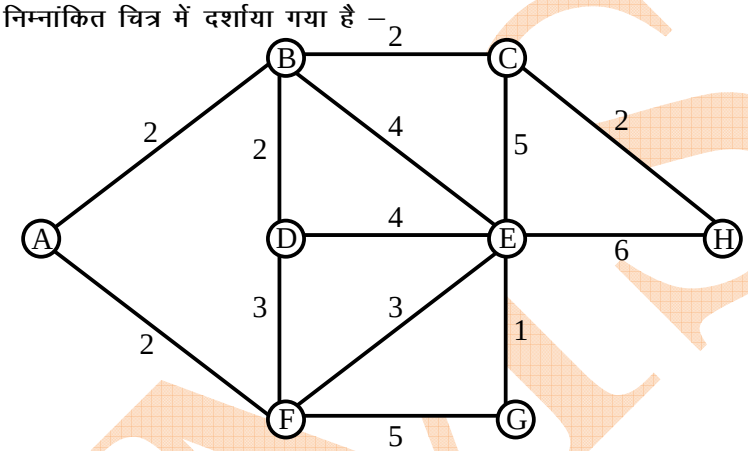
एक जुड़ा हुआ **Graph** जिसमें **cycle** न हो, उसे **Tree Graph** अथवा **Free Tree** अथवा साधारण **Tree** कहा जाता है। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –



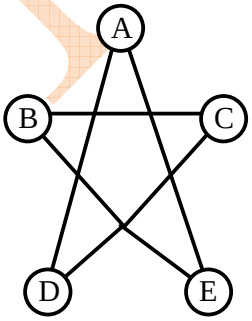
एक **Graph** जिसकी **Edges** पर **Data** लिखा होता है, **Labeled Graph** कहलाता है। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –



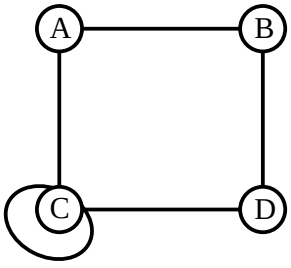
एक **Graph** जिसकी **Edges** पर धनात्मक अंक लिखे होते हैं, उन्हें **Weighted Graphs** कहते हैं। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –



वह **Graph** जिसके हर **Node** की **Degree** समान होती है, **Regular Graph** कहलाते हैं। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –

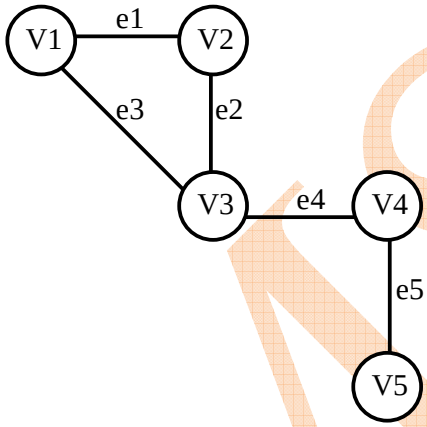


वे **Edges** जिनका **Intial** और **End Point** समान होता है, उन्हें **Parallel Edges** कहते हैं तथा वह **Edge** जिसका **Intial** और **End Point** एक ही होता है, उसे **Self Loop** कहते हैं।
ऐसा **Graph,** जिसमें न तो **Self Loop** हो या फिर **Parallel Edge** हो अथवा दोनों ही हों, **Multigraph** कहलाता है।



ऐसा **Graph,** जिसमें न तो **Self Loop** हो और ना ही **Parallel Edge** हो, **Single Graph** कहलाता है। इसे इस प्रकार भी कहा जा सकता है कि कोई **Graph,** जो कि **Multigraph** नहीं है, **Single Graph** कहलाता है।
ऐसी **Vertex,** जो किसी अन्य **Vertex** के **Contact** में नहीं होती, वह **Isolated Vertex** कहलाती है। ऐसा **Graph** जिसमें, **Isolated Vertex** होती है, उसे **NULL Graph** कहते हैं। ऐसा ही एक **Graph** निम्नांकित चित्र में दर्शाया गया है –

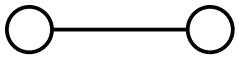
Adjacent Vertices (Neighbours)
निम्नांकित चित्र में **vertex V1, vertex V2** के **adjacent** कहलाएगी, यदि एक **edge (V1, V2)** अथवा **(V2, V1)** है –



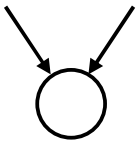
यहां **V1** और **V2 adjacent** हैं, **V1** और **V3 adjacent** हैं, **V2** और **V3 adjacent** हैं, **V3** और **V4 adjacent** हैं तथा **V4** और **V5 adjacent** हैं क्योंकि ये सभी **vertices** किसी न किसी **edge** के द्वारा आपस में जुड़े हुए हैं। प्रत्येक **Vertex** को उससे सम्बद्ध **edges** के आधार पर डिग्री **(Degree)** प्रदान की जाती है। यहाँ पर **V1 Vertex,** दो **Vertices V2** और **V3** से जुड़ा हुआ है, अतः **V1** की डिग्री **2** होगी।
इसी प्रकार **Vertex V2** की डिग्री **= 2**
Vertex V3 की डिग्री **= 3**
Vertex V4 की डिग्री **= 2**
Vertex V5 की डिग्री **= 1**
वह **Vertex** जिसकी डिग्री केवल **1** होती है, उसे **Pandent Vertex** कहा जाता है।
वह **Vertex** जिसकी डिग्री शून्य **(0)** है, उसे **Isolated Vertex** कहा जाता है। पिछे दिए गए चित्र में इसी प्रकार की **Vertex** को दर्शाया गया है।

Path: - closed path वह **Path** है जहां **Path** के **End Points Same** हो
वह **Path Simple Path** कहलाते हैं यदि सारे **Vertices** एक **Sequence** में हो और सभी अलग–अलग हो।

Connected Graph: - एक **Graph G Connected** कहलाता हैं यदि **2 Vertices** के बीच में एक **Path** हो



Multigraph: - एक **Graph** जिसकी **Multiple Edges** होती हैं। उन्हें **Multigraph** कहा जाता हैं।
Out Degree & In Degree of vertex: - एक **Vertex** से जितनी **Edges** बाहर की तरफ निकलती हैं। उन्हें **Vertex** की **Outdegree** कहा जाता हैं। एक **Vertex** पर जितनी **Edges** आकर खत्म होती हैं। उन्हें **Vertex** की **In Degree** कहा जाता हैं।


In Degree


Out Degree

Source & Sink: - एक **Vertex Source** कहलाता हैं जब उसकी **Outdegree Greater than 0** हो तथा **In degree = 0** हो।
एक **Vertex Sink** कहलाता हैं। जब उस **Vertex** की **In Degree Greater than 0** और **Out degree = 0** हो।
Parallel Edges: - अलग–अलग **Edges E** तथा **E- Parallel Edges** कहलाऐंगे यदि वे **same source** तथा **Terminal Vertices** से **Connected** हो।
Direct Acyclic Graph: - Directed Acyclic Graph एक **Directed Graph** है जिसमें **Cycles** नहीं बनती हैं।

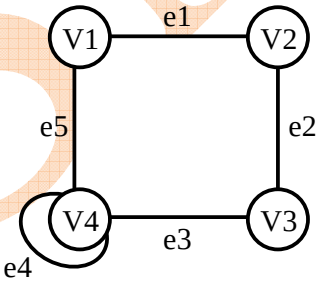
Incedency
एक **Edge** जिन **Vertices** को जोड़ती है वे **Vertices** उस **Edge** की **Incedent** होती है।
Edge e1 की incident – Vertices V1 और V2
Edge e2 की incident – Vertices V2 और V3
Edge e3 की incident – Vertices V1 और V3
Edge e4 की incident – Vertices V3 और V4
Edge e5 की incident – Vertices V4 और V5

Representation of Graph in Memory

Graph को **Memory** में **Representation** करने के निम्नलिखित तरीके हैं –

(i) Adjacency Matrix
(ii) Incedency Matrix
(iii) Adjacency List Representation
(iv) Multilist Representation

Adjacency Matrix
इस प्रकार के **Representation** में **vertex** का दूसरी **vertex** से सम्बन्ध **(Relation)** को एक **Matrix** के द्वारा प्रस्तुत करते हैं। निम्नांकित चित्र में दर्शाए गए **Graph** का **Adjacency Matrix Representation** चित्र के नीचे दर्शाया गया है –



	V1	V2	V3	V4
V1	0	1	0	1
V2	1	0	1	0
V3	0	1	0	1
V4	1	0	1	1

यहां हमने उन **vertices** के नीचे **1** लिखा है, जो सामने लिखी **vertex** की **adjacent** हैं। शेष के नीचे **0** लिखा है। जैसे **V1, V2** और **V4** की **adjacent** है इसलिए हमने **V2** और **V4** के नीचे **1** लिखा है और **V1** और **V3** के लिये **0** लिखा है।

Indedency Matrix

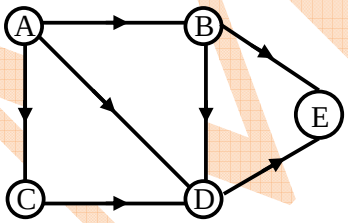
इस प्रकार के प्रस्तुतीकरण में **Vertex** का विभिन्न **Edges** से **Relation** को एक **Matrix** के द्वारा प्रस्तुत करते हैं। निम्न चित्र में दर्शाए गए **Graph** का **Incedency Matrix Representation** को नीचे दर्शाया गया है –

	e1	e2	e3	e4
V1	1	0	0	1
V2	1	1	0	0
V3	0	1	1	0
V4	0	0	1	1

यहां हमने उन **Vertices** को आगे **1** लिखा है, जो उपर दी गई **Edge** के **Incedent** है – जैसे **V1, e1** और **e5** की **Incident** है, इसलिए **e1** और **35** के नीचे **1** लिखा गया है और **e2, e3** और **e4** के नीचे **0** लिखा है।

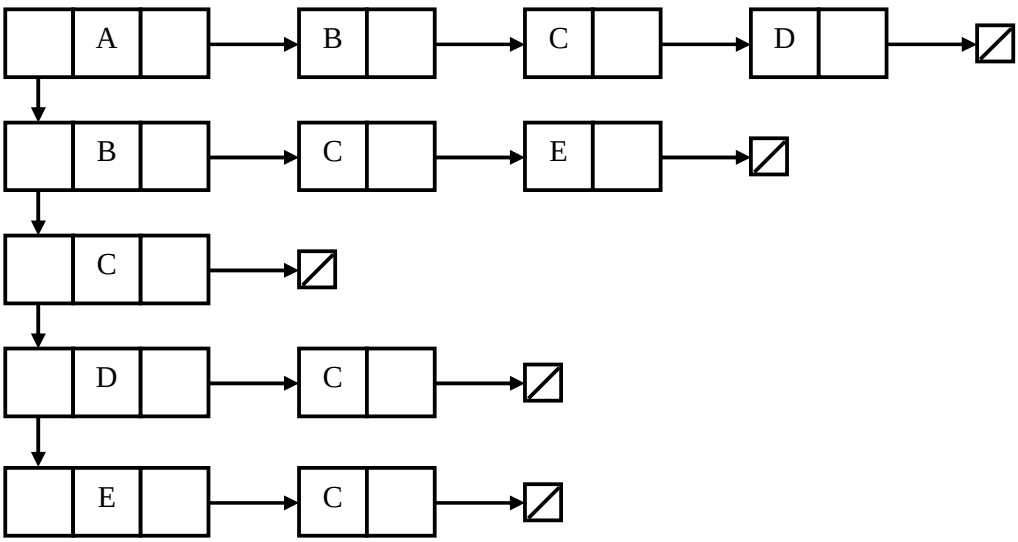
Adjacency List Representation

निम्न चित्र में दर्शाए गए **Graph** के लिए हम **Adjacency List** प्रस्तुतीकरण करेंगे। इसके लिए सर्वप्रथम एक **Table** बनाएंगे, जिसमें प्रत्येक **Node** की **Adjacent Node,** उसके सामने लिखी हो –



A	**B**	**C**	**D**
B	**C**	**E**	
C			
D	**C**		
E	**C**		

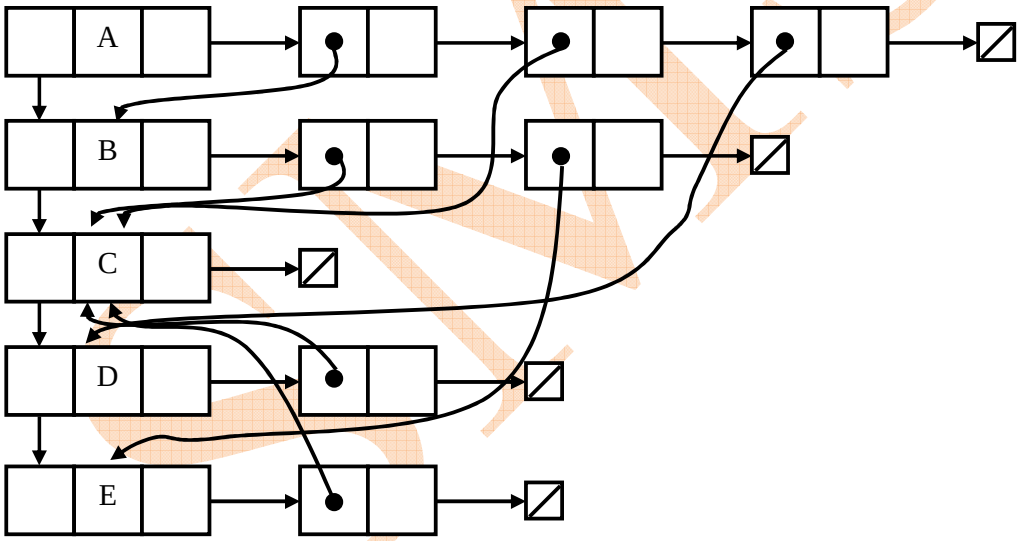
अब इस **Table** को निम्न रूप में **List** में परिवर्तित करेंगे –



यहां पर जिस **Node** की **adjacent Node** बताई गई है, उसमें दो **Pointer** प्रयुक्त किए हैं, पहला **Pointer** तो अगली **Node** को दर्शा रहा है और दूसरा **Pointer Adjacent Node** को।

Multilist Representation

इस प्रकार के **Representation** में हम **List** के द्वारा ही **Graph** को प्रस्तुत करते हैं, परन्तु **Adjacent Node** में उनकी सूचना नहीं बल्कि उनके **Pointer** को रखा जाता है।



Graph Traversal

Graph Traversal का अर्थ है – **Graph** की प्रतयेक **Node** को **Visit** करना। एक **Graph** की **Vertices** को **Visit** करने के अनके तरीके हो सकते हैं। जो दो तरीके हम आपको यहां बताने जा रहे हैं, वो बहुत ही प्रचुर मात्रा में प्रयोग किए जाते हैं और ये तरीके **Traversal** के अत्यन्त सरल तरीके सिद्ध हुए हैं। ये तरीके निम्नलिखित हैं–

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

Breadth First Search (BFS)

जैसा कि नाम से प्रतीत होता है कि इस तरीके में **Nodes** को चौड़ाई से **Visit** करना है। **BFS** में पहले हम **Start Vertex** की समस्त **Vertices** को **Visit** करते हैं और फिर इनकी सभी **Unvisited Vertices** को **Visit** करते हैं।

इसी प्रकार आगे तक समस्त **Vertices** को **Visit** किया जाता है। **BFS** के लिए निम्नलिखित **Algorithm** को ध्यान में रखते हैं –

status 1 = ready
status 2 = waiting
status 3 = process

Step 1- **Graph** की सभी **Nodes** को **visit** के लिए तैयार रखें। **(Status1)**

Step 2- **Start Vertex A** को **Queue** में डालकर इसका **Status Waiting** करें। **(Status 2)**

Step 3- **Step 4** से **Step 5** तक दोहराये जब तक **Queue** खाली न हो जाए।

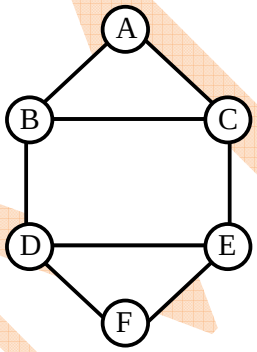
Step 4- **Queue** की पहली **Node n** को **Queue** से **Remove** करके **n** को **Process** कर दें। **(Status 3)**

Step 5- **Queue** के अन्त में **n** के सभी **Adjacent Vertices** जो कि **ready – state** में है, को डाल दें। **(Status 1)**

Step 6- **Step 3** का लूप खत्म हुआ।

Step 7- **Exit**

इस पूरे **Process** को समझने के लिए नीचे दिए गए उदाहरण का अध्ययन करें –



इस चित्र के लिए पहले **Adjacent Table** बना लेते हैं –

A	**B**	**C**	
B	**A**	**C**	**D**
C	**A**	**B**	**E**
D	**B**	**E**	**F**
E	**C**	**D**	**F**
F	**D**	**E**	

अब हम किसी भी **Vertex** को **Start Vertex** बनाकर कार्य शुरू कर सकते हैं।
इस उदाहरण में हमने **B** को **Start Vertex** माना है। अब हमें इसे **Queue** में डालना हैं।

Queue	**B**
Origin	**0**

इस **Queue** के दो भाग है – पहला भाग तो साधार **queue** है जिसमें हम **Vertex** की **Adjacent Vertices** डालेंगे, जबकि दूसरे भाग में यह **Entry** की गई है कि वे किसकी **Adjacent Vertices** हैं। जो–जो **Vertices, visit** हो जाए उन्हें **Queue** के नीचे लिख लें। अब हमें **B** की समस्त **Adjacent Vertices** को **Queue** में डालना है –

Queue	**A**	**C**	**D**

Origin	B	B	B

B A C D

अब इन **Vertices** की **Unvisited Adjacent Vertices** को **Visit** करना है। जैसा कि हम देख सकते हैं कि **A** की समस्त **Adjacent Vertices Visit** हो चुकी हैं, इसलिए अब हम **C** की बची **Adjacent Vertices** को **Queue** में डालेंगे।

Queue	D	E
Origin	B	C

B A C D E

अब हमें **D** की बची **Adjacent Vertices** को भी **Queue** में डालना है –

Queue	E	F
Origin	B	D

B A C D E F

अब जैसा कि हम देख सकते हैं कि **Graph** की समस्त **Nodes Visit** हो चुकी हैं। अतः **Graph** का **traversal** सम्पन्न हो चुका है और **Queue** के नीचे लिखी **List** ही दिए गए **Graph** का **BFS** हैं।

एक **Graph BFS** भिन्न हो सकता है चूँकि यह ***Start Vertex*** पर निर्भर करता है।

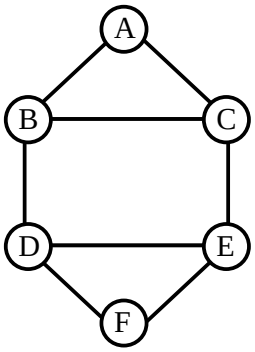
Depth First Search (DFS)

इसके नाम से ही प्रतीत हो रहा है कि यह तरीका **Graph** को गहराई से **Visit** करता है। **DFS** में हम सबसे पहले **Start Vertex** को **Stack** के द्वारा **Visit** करते हैं, फिर इसकी समस्त **Adjacent Vertices** को **Stack** में डालकर, **Stack** के **Top** पर स्थित क्रिया तब तक दोहराते हैं जब तक कि **Stack** खाली न हो जाए। **DFS** करने के लिए अग्रलिखित **Algorithm** को ध्यान में रखते हैं –

status 1 = ready
status 2 = waiting
status 3 = process

Step 1- **Graph** की सभी **Nodes** को **visit** के लिए तैयार रखें। **(Status 1)**
Step 2- **start vertex** को **stack** में डालकर इसका **status Waiting** कर दें। **(Status 2)**
Step 3- **Step 3** से **5** तक तब तक दोहराएं जब तक कि **Stack** खाली न हो जाए।
Step 4- **Stack** के **Top** में से **Node** को निकालकर उसे **Process** करें। **(Status 3)**
Step 5- **n** की **Adjacent Vertices** को **Stack** में डालकर उनका **Status 1** से **2** करें।
Step 6- **Step 3** का **Loop** खत्म हुआ।
Step 7- **Exit**

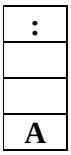
इस पूरे **process** को समझने के लिए नीचे दिए गए उदाहरण का अध्ययन करें –



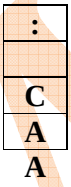
इस चित्र के लिए पहले **Adjacent Table** बना लेते हैं –

A	**B**	**C**	
B	**A**	**C**	**D**
C	**A**	**B**	**E**
D	**B**	**E**	**F**
E	**D**	**C**	**F**
F	**D**	**E**	

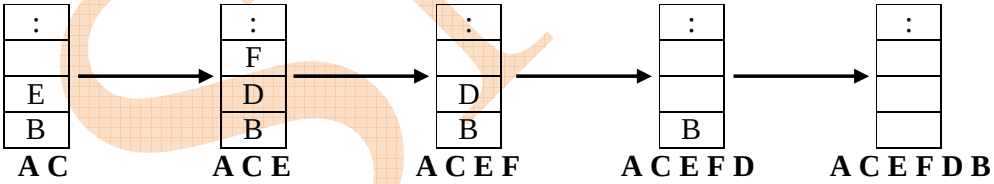
अब **Start Vertex** को **Stack** में डाल दें –



अब सबसे पहले **Visited Node** को **stack** के नीचे लिख लें। **Visited Node** की समस्त **Adjacent Nodes** को **Stack** में डाल दें –



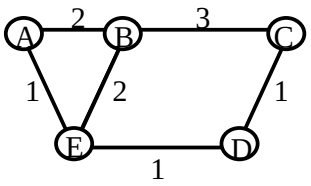
अब पुनः **Top of the Stack** को **stack** से बाहर निकालकर इसकी **Adjacent** को **Stack** में डाल दें और इसी प्रकार तक आगे करते रहिए जब तक **stack** खाली ना हो जाए।



अब जैसा कि हम देख सकते हैं कि **stack** पूरा खाली हो चुका है अतः **DFS** सम्पन्न हो चुका है। खाली **Stack** के नीचे **list** ही **Graph** का **DFS** है।
एक **Graph DFS** भी भिन्न हो सकता है चूँकि यह भी **Start Vertex** पर निर्भर करता है।

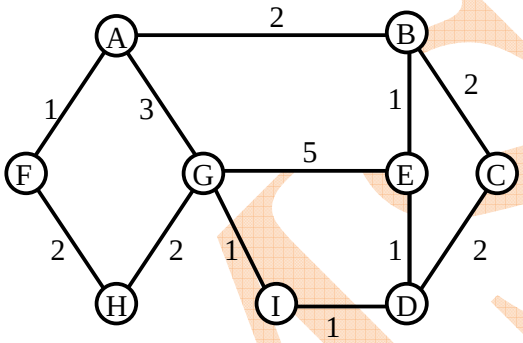
Shortest Path Problem

हमने **Graph** को **Traverse** करते समय देखा कि हम **Graph** को **Graph** की **Edges** के द्वारा **travel** करते ही किसी – किसी **case** में ऐसा भी हो सकता है कि इन **Edges** पर कुछ **Weight** दिया गया है। यह **Weight** की दूरियों के उपर असर डाल सकता है।



निम्न चित्र में दर्शाए गए **Graph** का अध्ययन करें। हम **Graph** की प्रत्येक **Node** से **Graph** की दूसरी प्रत्येक **Node** पर जा सकते हैं। माना हमें **A Node** से **C Node** पर जाना है तो हमारे पास दो रास्ते है – **A→B→C** या **A→E→D→C**। देखने में तो दूसरा रास्ता लम्बा लगता है, परन्तु यदि हम **edge** पर लिखे **Weight** से गणना करें तो दूसरे रास्ते को चुनेंगे। इस प्रकार हम सदैव एक छोटे–छोटे **Path** को ही चुनेंगे।

यहाँ पर हमें **Graph** को दाहरा लेना चाहिए। **Graph** में **Path, Vertices** का एक क्रम है, जिस प्रकार वे **Edge** में हैं। **Path** की लम्बाई उसकी **Edge** पर लिखे **Weight** के जोड़ के बराबर होती है। **Path** की पहली **Vertex, Path** की **Source Vertex** तथा अन्तिम **Vertex, Path** की **Destination Vertex** कहलाती हैं।



अब हम उपरोक्त चित्र में दर्शाए **Graph** के लिए **Shortest Path** निकालेंगे।

यहां पर हम देख सकते हैं **A** से **D** तक जाने के अनेक रास्ते हैं –

पहले **Path** की लम्बाई **A → B → C → D ⇒ 2 + 2 + 2 = 6**

दूसरे **Path** की लम्बाई **A → G → I → D ⇒ 3 + 1 + 1 = 5**

तीसरे **Path** की लम्बाई **A → B → E → D ⇒ 2 + 1 + 1 = 4**

हम देख सकते हैं कि **A → B → E → D Path** की कीमत **(cost)** सबसे कम है, इसलिए हम इसी रास्ते को अपनाएंगे।

किसी भी **Graph** में **Shortest Path** को खोजने के लिए प्रयोग किए जाने वाला तरीका **Dijkastra** का **Method** कहलाता है। इस तरीके को प्रयोग करने के लिए निम्नलिखित **Algorithms** को ध्यान में रखते हैं –

status 1 = ready state

status 2 = coloured state

Step 1- **Graph** की सभी **Nodes** को **ready state** में **initialize** करा दें। **(Status 1)**

Step 2- **Starting Vertex** का **Status** बदल दें। **(Status 2)**

Step 3- **Loop 4** को तब तक दोहराएं जब तक कि सभी **Nodes Coloured Status** में नही परिवर्तित कर दें।

Step 4- **n** की **Adjacent Node,** जिसका सबसे कम **Weight** हो, का **Status** बदलकर **Coloured Status** कर दें।

Step 5- **Step 2** का **Loop** खत्म हुआ।

Step 6- **Exit**

इसे अब उदाहरण की सहायता से समझेंगे।

	A	B	C	D	E	F	G	H	I
	{	2	∞	∞	∞	1	3	3	∞

हम यहां देख सकते हैं कि सबसे कम **Weight** वाली **Univisited Edge AB** है, इसलिए हम इसकी **Unvisited Incedent Vertices** को पुरानी शर्तों के अनुसार ही **Table** में जोड़ देंगे।

	A	B	C	D	E	F	G	H	I
	{	2	4	∞	3	1	3	3	∞

हम यहाँ देख सकते हैं कि सबसे कम **Weight** वाली **Unvisited Edges AE, AG** और **AH** हैं। इन तीनों में से किसी भी **Edge** को पहले चुना जा सकता है। यहां हम **AE** ले रहे हैं और इसकी **Unvisited Incedent Vertices** को पुरानी शर्तों के अनुसार **Table** में जोड़ लेंगे।

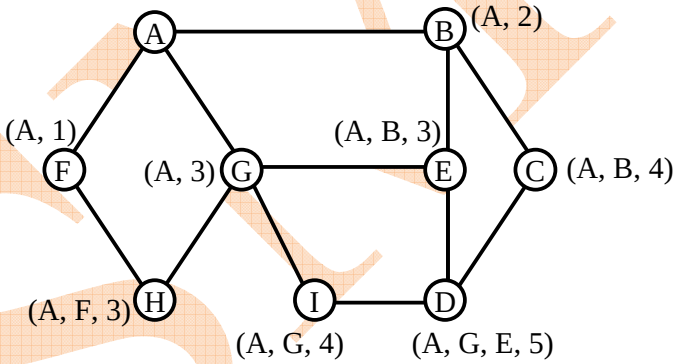
	A	B	C	D	E	F	G	H	I
	{	2	4	4	3	1	3	3	∞

अब हम **AG** की सारी **unvisited incident vertices** को पुरानी शर्तों के अनुसार लिस्ट में जोड़ देंगे।

	A	B	C	D	E	F	G	H	I
	{	2	4	4	3	1	3	3	4

अतः हम देख सकते है यदि **A** को **I** पर पहुंचाना है, तो उसकी कीमत **(Cost)** केवल **4** है। **A** से अन्य **Nodes** पर पहुंचने के लिए **Path** निम्नलिखित हैं –

A → B ⇒ 2
A → B → G ⇒ 4
A → F ⇒ 1
A → G ⇒ 3
A → F → H ⇒ 3
A → G → I ⇒ 4
A → G → E → D ⇒ 4
A → B → E ⇒ 3



यहां पर प्रत्येक **Node** उस पर **A Node** से पहुंचने का रास्ता **(Path)** व कीमत **(Cost)** बता रही है।

Minimal Spanning Tree

Spanning Tree वह **Tree** है, जो कि एक **Graph** द्वारा बनाया जाता है और एक **Minimal Spanning Tree ,** वह **Spanning Tree** जिसकी **Edges** की कीमत **(Cost** सबसे कम है।

जिस **Graph (G)** से **Spanning Tree** का निर्माण किया जा रहा है, उसका **Connected** होना आवश्यक है। यदि **Graph Connected** नहीं है, तो उससे **Spanning Tree** का निर्माण नहीं किया जा सकता क्योंकि **Unconnected Graph** एक **Tree** कहलाता है।

यदि कोई **Tree (T)** जो कि किसी **Connected Graph** का **Spanning Tree** है, तो –

(1) Graph (G) की प्रत्येक **Vertex Tree (T)** की **Edge** को **Belong** करेगी, और

(2) Tree (T) में **Graph (G)** की **Edges** ही **Tree** बनाऐं।

हम किसी **Graph** को **Minimal Spanning Tree** में निम्नलिखित दो **Methods** से परिवर्तित कर सकते हैं –

(1) Krushkal's Method

(2) Prim's Method

Kurushkal's Method

J. Krushkal का सन् **1957** में प्रतिपादित यह **Method, Minimal Cost Spanning Tree** बनाने का बहुत ही जाना माना तरीका है। यह **Method Graph** की **Edges** की एक **List** पर कार्य करता है।
यहां पर **Input** एक **Connected Weight Graph G** है, जिसमें **n Vertices** हैं।

Step 1- **Graph** की **Edges** को बढ़ते क्रम में व्यवस्थित कर लें।

Step 2- **Graph G** की **Vertices** से शुरू होकर बारी – बारी से **Process** करें और प्रत्येक **Edge** को जोड़े जो एक **Cycle** नहीं बनाती अथवा जब तक **n-1 Edges** नहीं जुड़ जाती।

Step 3- **Exit**

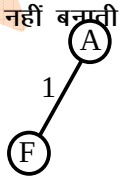
यह तरीका तब तक तो लाभदायक हे जब तक कि **Graph** छोआ है क्योंकि इस प्रकार से **Tree** बनाने में शुरू में **Nodes Unconnected** रहती हैं।
इसे अब उदाहरण की सहायता से समझेंगे।

Step 1- **Graph** की **Edges** को बढ़ते क्रम में व्यवस्थित कर लें।

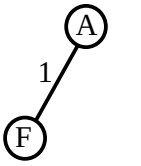
AF	**1**
BE	**1**
ED	**1**
ID	**1**
GI	**1**
AB	**2**
BC	**2**
CD	**2**
HG	**2**
FH	**2**
AG	**3**
GE	**5**

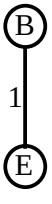
अब पहली **Edge** से शुरू करते हुए **Tree** को तब तक बनाऐं, जब तक कि **n-1** अर्थात् **(9-1=8) Edges** नहीं हो जाती तथा उन्हीं **Edges** को जोड़ेंगे जो कि **Cycle** नहीं बनाती हैं।

(1) हम **Tree** का निर्माण किसी भी **1 Weight** वाली **Edge** से प्रारम्भ कर सकते हैं। यहां पर हमने **AF Edge** से यह निर्माण प्रारम्भ किया है।

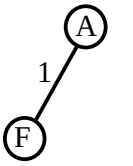


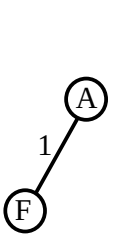
(2) यहां पर **Tree** में एक अन्य **1 Weight** वाली **Edge BE** को जोड़ लिया है।





(3) यहां पर **Tree** में एक अन्य **1 Weight** वाली **Edge ED** को जोड़ लिया है।

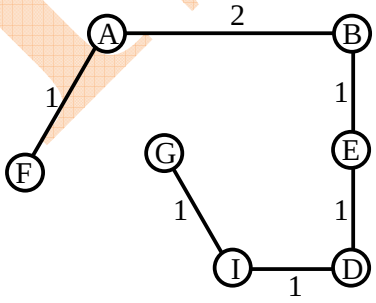




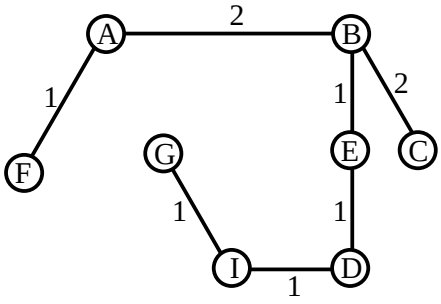
(4) यहां पर **Tree** में एक अन्य **1 Weight** वाली **Edge D1** को जोड़ लिया है।

(5) यहां पर **tree** में एक अन्य **1 Weight** वाली **Edge IG** को जोड़ लिया है।

(6) यहां पर **Tree** में एक **2 Weight** वाली **Edge AB** को जोड़ लिया है, क्योंकि **1 Weight** वाली सभी **Nodes Visit** हो चुकी हैं।

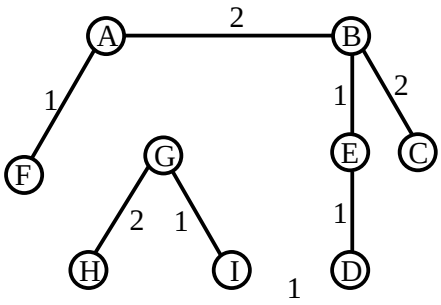


(7) यहां पर **Tree** में एक अन्य **2 Weight** वाली **Edge BC** को जोड़ लिया है।



(8) यहाँ चूँकि **CD** एक **cycle** बना रही है इसलिये हम इसे नही जोडेंगे।

(9) यहां पर **tree** में एक अन्य **2 Weight** वाली **Edge GH** को जोड़ लिया है।



जैसा कि हम देख सकते हैं कि **n-1 (8) Edges** जुड़ चुकी है। इस **Tree** की कीमत **(cost) = 1+2+2+1+1+1+1+2=11** हैं।

Prim's Method

Prim का **Method** भी **connected graph** को **Spanning Tree** में परिवर्तन करने के लिए प्रयोग होता है।

Step 1- इसमें एक टेबल साथ – ही – साथ प्रयोग की जाती है, जिसमें प्रत्येक **Vertex** की **Adjacent Vertices** उनके **Weight** के साथ लिखी जाती हैं।

Step 2- **Step 3** को तब तक दोहराएं जब तक कि **n-1 Edges** न जुड़ जाएं।

Step 3- सबसे कम **Weight** वाली **Edge** को **Tree** में जोड़े, परन्तु यह ध्यान रखा जाना चाहिए कि कोई भी **edge cycle** ना बनाए।

Step 4- **Step 2** का **Loop** समाप्त हुआ।

Step 5- **Exit**