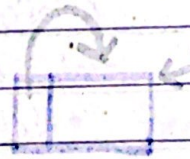
* So, in CIRCULAR LL whenever a Last node is "inserted" or "deleted", its next pointer must be set to the first node (head).

APPLICATIONS OF CIRCULAR L.L:

* Round Robin Schedulling in CPU.
  ↳ CPU schedulling Job; which process to run which time.

* To keep track of the turn in a multi-player game.
  → LUDO
  → CHESS

* To repeat videos in a playlist.



OPERATIONS OF CIRCULAR LINKED LIST:

* Here structure of all algorithm changes repectively.

* Node class architecture remains same.

// CODE

```
class Node {
    int data;
    Node* next;

public:
    Node(int data, Node* next) {
        this->data = data;
        this->next = next;
    }

    friend class CircularList;
};

class CircularList {
    Node* head;
```

public:

// CONSTRUCTOR

```
CircularList () {
        head = NULL;
}
```

// DESTRUCTOR

```
~CircularList () {
    if (head != NULL) {
        Node* temp = head;
        while ( temp->next != head)
            temp = temp->next;

        Node* cur = head;

        while (cur->next != head){
            head = head->next;
            temp->next = head;

            delete cur;
            cur = head;
        }

        delete head;
        head = NULL;
    }
}
```

// IS EMPTY (remains same)

```
bool isEmpty () {
    return !head;
}
```

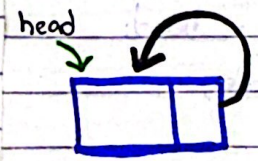
// INSERT AT START

```
void insertAtStart (int data) {
    Node* newNode = new Node(data, N

    if (head == NULL) {
        head = newNode;
        head->next = head;
    } else {
        Node* cur = head;

        while(cur->next != head)
            cur = cur->next;

        cur->next = newNode;

        newNode->next = head;
        head = newNode;
    }
}
```

// INSERT AT LAST

```
void insertAtLast( int data) {
    Node* temp = new Node(data, NULL);

    if (head == NULL) {
        head = temp;
        head->next = head;
    } else {
        Node* cur = head;

        // Locating Last node

        while (cur->next != head){
            cur = cur->next;

        cur->next = temp;
        temp->next = head;
    }
```

★ INSERTING FIRST NODE



(self - refrencing)

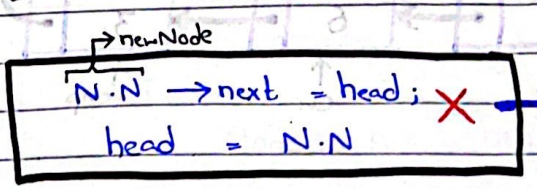
WRONG

WRONG

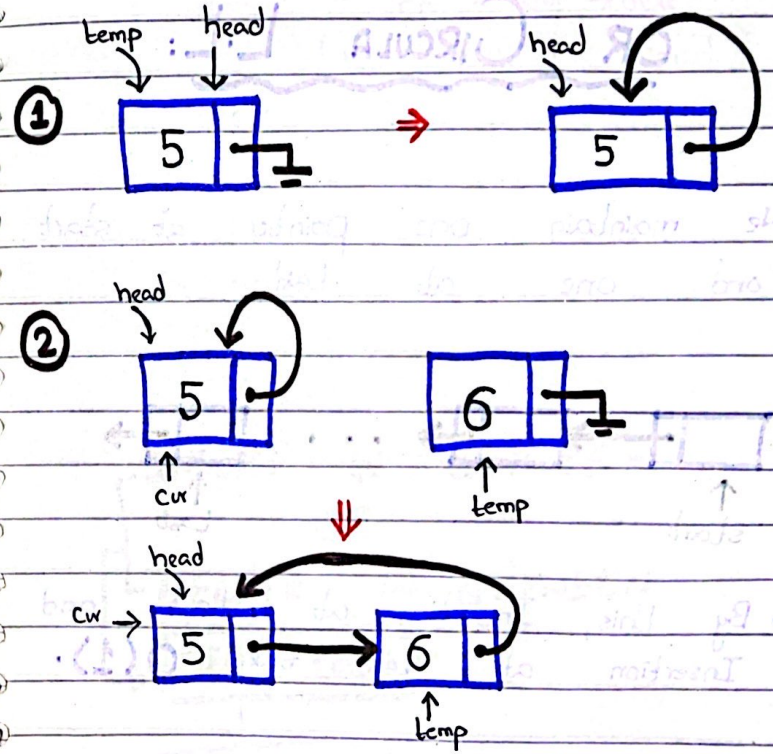
- Direct Not Valid:

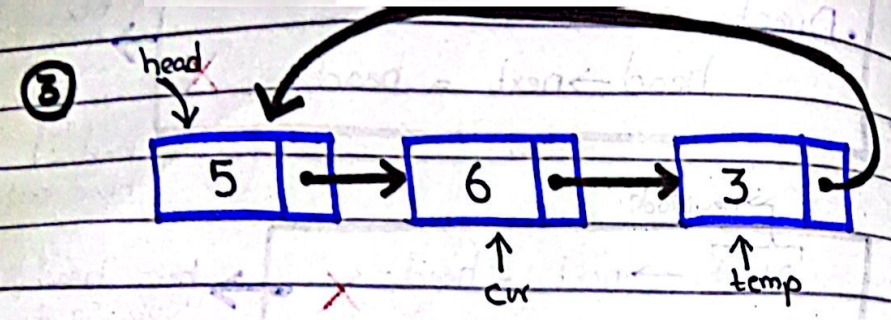
```
head->next = head;   X
```

bcz head is at NULL So, there is not next of head ex



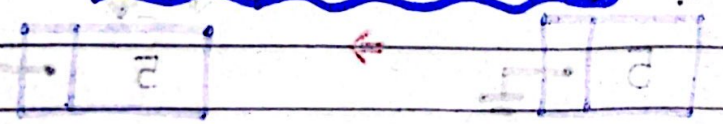
★ RUN : INSERTING AT LAST



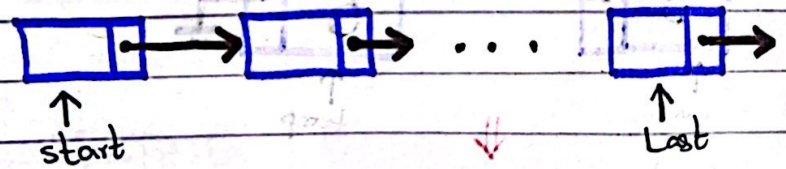


NOTE:

OPTIMIZATION OF SINGLY LL OR CIRCULA L.L:



★ We maintain one pointer at start and one at Last.



→ By this, Insertion at Start and Insertion at Last takes O(1).

```
//TRAVERSING CIRCULAR L.L
void display () {
    if (head == NULL) {
        cout << "List is Empty";
    } else {
        Node * cur = head;

        do {
            cout << cur → data;
            cur = cur → next;
        } while ( cur != head);
    }
}
```

INSERT IN BETWEEN
OR
DELETE IN BETWEEN

SEE CODE

↓

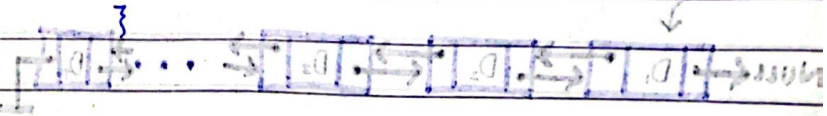
circular_link_list_all_operations.cpp

// DELETE FIRST NODE

```
void deleteFirst () {

    if (head == NULL)
        cout << "List is already empty";
    else {

        if (head->next == head) {
            delete head;
            head = NULL;
        } else {
            Node* cur = head;

            do {
                cur = cur->next;
            } while (cur->next != head);

            Node* temp = head;

            head = head->next;
            cur->next = head;
            delete temp;
        }
    }
```

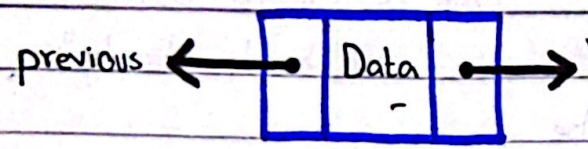
// DELETE LAST NODE

```
void deleteLastNode() {
    if (head == NULL)
        cout << "Circular List is already Empty";
    else {
        if (head->next == head) {
            delete head;
            head = NULL;
        } else {
            Node* pre = head;
            Node* cur = head->ne

            while (cur->next != head)
                pre = cur;
                cur = cur->next;
            }

            pre->next = head;
            delete cur;
        }
    }
```



③ DOUBLY LINK LIST

- Architecture of Node

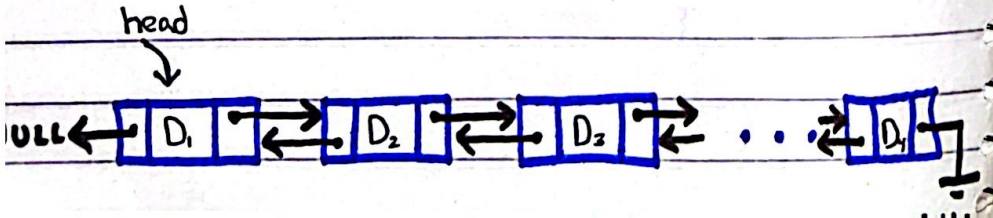


* It has two pointers ; "next" and "previous".

* It is forward as well as backward.

* Each Node apart from storing its data has two links.
  - → The "FIRST LINK" points to the "PREVIOUS NODE" in the list.
  - → The "SECOND LINK" points to the "NEXT NODE" in the List.

* It is used mostly bcz it is most powerful.



APPLICATIONS / EXAMPLS OF DOUBLY LINKED LIST

* To implement Undo and Redo functionality in various applications.
* DATA EXPLORER
  - ↳ backward
  - ↳ forward
* Web browser to implement backw and forward navigation of visite web pages.
* In systems where both front and back navigation is require

CODE OF DOUBLY LINK LIST

- Node class Arctecture will be changed.

```
Class DNode {

    int data;    // FOR DATA

    Node* pre;   // FOR PREVIOUS ADRESS

    Node* next;  // FOR NEXT ADRESS

    public:

        DNode (int data, DNode* pre, DNode*
                                    next) {

            this->data = data;
            this->pre  = pre;
            this->next = next;
        }

        friend class DList; // DNode class
                            became public
                            to DList.
};
```

```
class DList {

    DNode* head;

    public:
        // CONSTRUCTOR
        DList () {
            head = NULL;
        }

        // DESTRUCTOR

        ~DList () {
            if (head != NULL) {
                Node* cur = head;

                while (cur != NULL) {
                    head = head->next;
                    delete cur;
                    cur = head;
                }
                delete head;
            }
        }
```

// IS EMPTY → will remain Same

```
bool isEmpty() {
    return !head;
}
```

// INSERT AT LAST

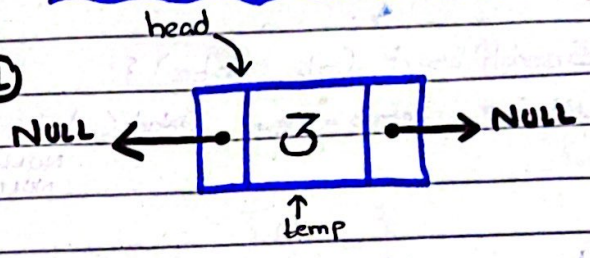
```
void insertAtLast(int data) {
    DNode* temp = new DNode(data, NULL, NULL);

    if (head == NULL)
        head = temp;
    else {
        DNode* cur = head;

        while(cur->next != NULL)
            cur = cur->next;

        cur->next = temp;
```

`cur->next->pre = cur` OR `temp->pre = cur;`

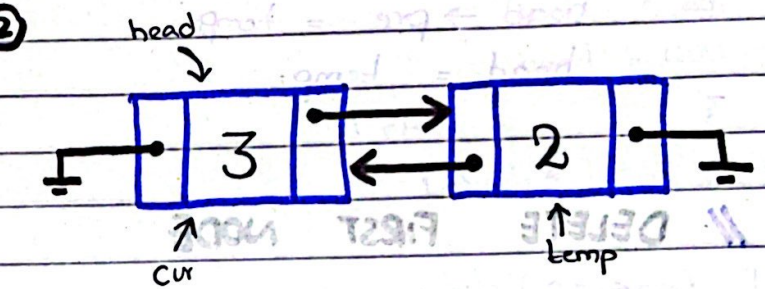
→ same work

```
    }
}
```

FOR FIRST NODE



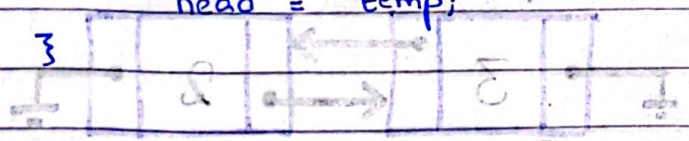
★ For first Node pre and next both points to NULL



★ In Doubly Link List 1st Node's pre is NULL bcz 1st Node does not have any predecessor and Last node's next is NULL bcz Last node does not have any successor

// INSERT AT START

```
void InsertAtStart (int data) {
    DNode* temp = new DNode(data, NULL, NULL);

    temp->next = head;
    head->pre = temp;
    head = temp;
}
```



// DELETE FIRST NODE

```
void deleteFirstNode () {
    if (head == NULL)
        cout << "Doubly List is Already Empty";
    else {
        Node* cur = head;

        head = head->next;
        head->pre = NULL;
        delete cur;
        cur = NULL;
    }
}
```

// DELETE LAST NODE

```
void deleteLastNode () {
    if (head == NULL)
        cout << "Doubly List is Already Empty";
    } else {
        if (head->next == NULL) {
            delete head;
            head = NULL;
        } else {
            DNode* cur = head;

            while (cur->next != NULL)
                cur = cur->next;

            cur->pre->next = NULL;

            delete cur;
        }
    }
}
```

// TRAVERSING FORWARD

```
void displayForward() {
    if (head == NULL)
        cout << "Doubly List is Empty";
    else {
        DNode* cur = head;
        while (cur != NULL) {
            cout << cur->data;
            cur = cur->next;
        }
    }
}
```

// TRAVERSE BACKWARD

```
void displayBackward() {
    if (head == NULL)
        cout << "Doubly List is Empty";
    else {
        DNode* cur = head;
        while (cur->next != NULL)
            cur = cur->next;

        while (cur != NULL) {
            cout << cur->data;
            cur = cur->pre
        }
}
```

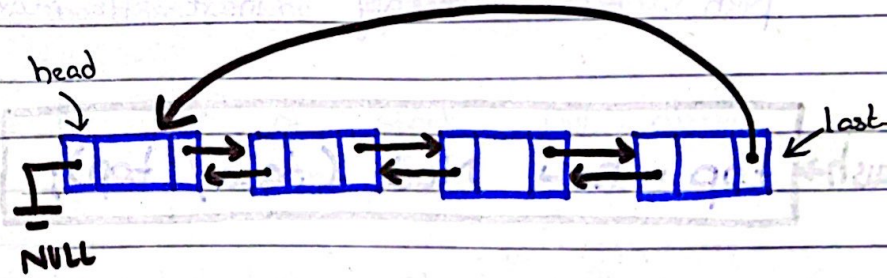
INSERTION IN BETWEEN
OR
DELETION IN BETWEEN

→ SEE CODE

doubly_link_list_all_operations.cp

④ CIRCULAR DOUBLY LINK LIST

* Same architecture as doubly Link List only last node pointing towards first node (head).



* In Circular Doubly Link list 1st Node previas will always pointing to NULL.

STACK USING SINGLY LINK LIST

* **LIFO :** Last In First Out.
* In stack we only hold / make top instead of head.

OPERATIONS:

- **PUSH :**
  - Insert the new node at the **FRONT**.
  - So, In Stack using Link List push is basically "insert At Start".

push→
```
top = new Node (data, top);
```

- **POP :**
  - Remove the first node.
  - So, In Stack using Link List pop is basically "Delete at first" or "Delete first Node".

```
temp = top;
top = top → next;
delete temp;
```
←pop

NOTE:

* In this way the most recently added node at the front / top will be removed first, hence maintains the **LIFO** structure.
* The "**TOP**" of stack will always points to the first node.
* In Link List, Stack will never full, Stack here will also not have any curSize or maxSize.

CODE ARCITECTURE OF STACK USING SINGLY LINK LIST:

```
class Node {
    int data;       // Holds Data
    Node* next;     // Holds Reference of next node
public
    friend class Stack; // Node class became public to stack
};

class Stack {

    Node* top;  // holds reference of first node of stack.
public:

    // all other functionalities such as push, pop, isEmpty etc.
}
```

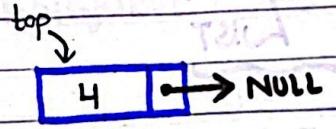
see Code

↳ stack_link_list.cpp

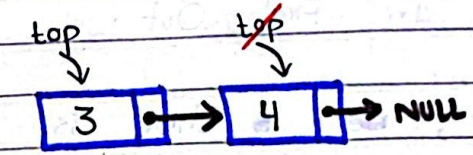
OPERATIONS

→ top == NULL

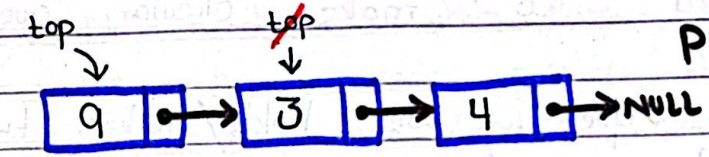
1) push(4)



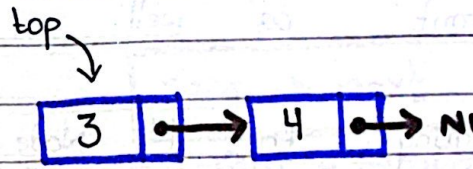
2) push(3)

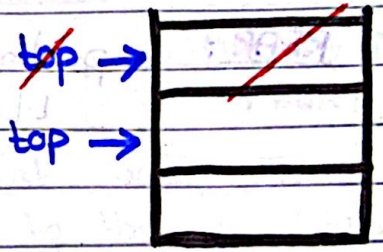


3) push(9)



4) pop()





FUNCTIONS OF STACK:

- → IsEmpty
- → Insert At Start (PUSH)
- → Delete At Start (POP)
- → Traverse (DISPLAY)

} O(1)

} O(n)

QUEUE USING SINGLY LINKED LIST

★ **FIFO:** First In First Out.

★ In Link List; there is no need to make circular queue.

★ In queue, we holds/make two pointers; front as well as rear.

FRONT: pointing to 1st Node of L.L.

REAR: pointing to Last Node of L.L.

OPERATIONS:

- **ENQUEUE:** → **REAR** → **INSERT AT LAST**
  - Insert the newNode at the end. The **REAR** points to the newly insated Last node.
  - No need of while loop boz we have rear pointer At fr We simply use/update rear poir

```
rear → next = temp;
rear = temp;
```

- **DEQUEUE:** → **FRONT** → **DELETE FI FRONT**
  - Remove the first node; The **FRONT** points to the first Nod
  - If only one Node left and w dequeue it then our front as wel as rear both points to **NULL**.

```
t = front;
front = front → next;
delete t;
```

★ In L.L, Queue will never full, Queue here will also not have any curSize or maxSize.

CODE ARCITECTURE OF QUEUE USING SINGLY LINK LIST.

```
class Node {
    int data;          // holds data
    Node * next;       // holds reference of next node

    friend class Queue;  // Node class became public to queue.
};

class Queue {

    Node * front;  // holds first node adress
    Node * rear;   // holds last node reference of queue.
public:

    // all other functionalities such as enQueue, deQueue etc.
};
```

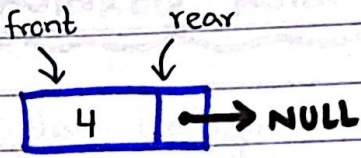
See Code → queue_link_list.cpp

NOTE: In Queue no IsFull Function exists.

OPERATIONS

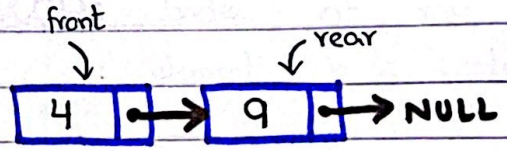
→ front == rear == NULL

1) If first node enQueue's then both front and rear points to that (first) node.



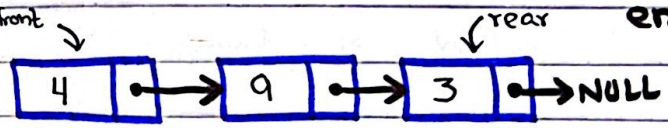
enQueue(4)

2)



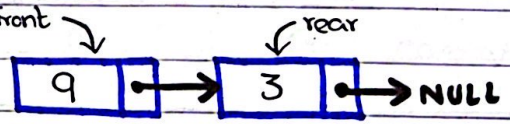
enQueue(9)

3)



enQueue(

4)



deQueue

FUNCTIONS OF QUEUE.

- → IsEmpty — Θ(1)
- → Insert At End — enQueue — Θ(1)
- → Delete from front — deQueue — Θ(1)
- → traverse — Θ(n)

SPARSE MATRIX

★ A sparse matrix is a matrix where majority of the elements have the value zero/null.

① ARRAY (TRIPLET) REPRESENTATION.

- Explained previously.

② LINKED LIST REPRESENTATION

→ In this technique each node has "FOUR" fields.

- **ROW:** Index of row, where non-zero element is located.
- **COLUMN:** Index of column, where non-zero element is located.
- **VALUE:** value of non-zero element located at index (row, column).
- **NEXT:** Address of next node.

First Node of Linked Sparse

→ 1st node is a "Meta Data". Data about Sparse Matrix.

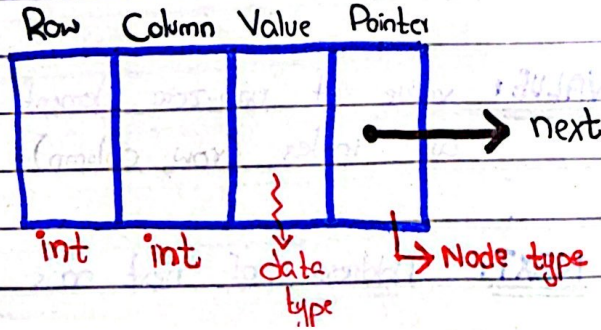
Ther first (HEAD) node of Linked List stores:

↳ total no. of rows of sparse matrix.

↳ total no. of columns of sparse matrix.

↳ total no. of non-zero values of sparse matrix.

Node Arcitecture of L.L Sparse Representation.

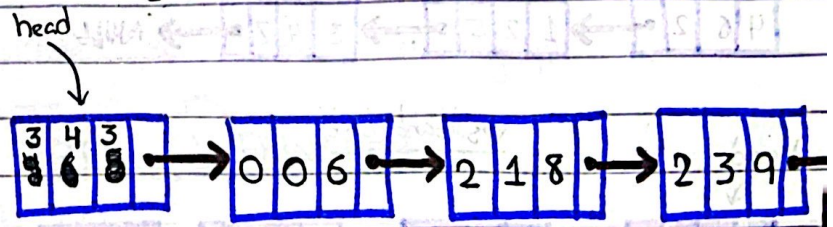


Code Structure

```
class Node {
    int row;      // row-index of non-zero e
    int column.   // column-index of non-zero ele
    int value;    // non-zero element of
                     index (row, column)
Its type can vary i.e char, float, sting etc.

    Node* next;   // reference of next

};
```

Example: * Consider a matrix of 3x4 containging 3 non-zero values.

	0	1	2	3
0	6	0	0	0
1	0	0	0	0
2	0	8	0	9



* Here, we no need to create any memory before/first. Bez here nodes are allocating on

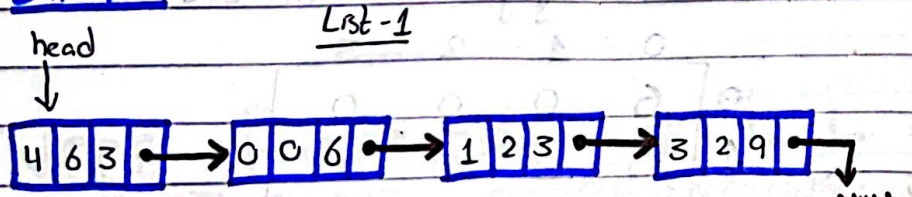
Addition of Sparse Matrix

→ In Linked List Sparse Representation while adding two Sparse Matrixes there will no need to allocate memory before, for resultant matrix. Simply add elements one by one and create node and put that element on node.

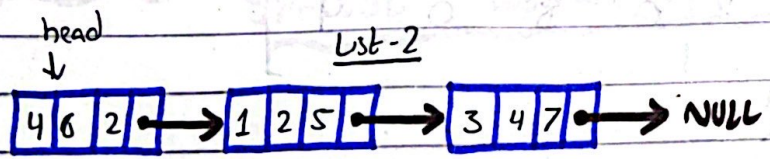
→ By this way there will be no wastage of memory in resultant list.

EXAMPLE:

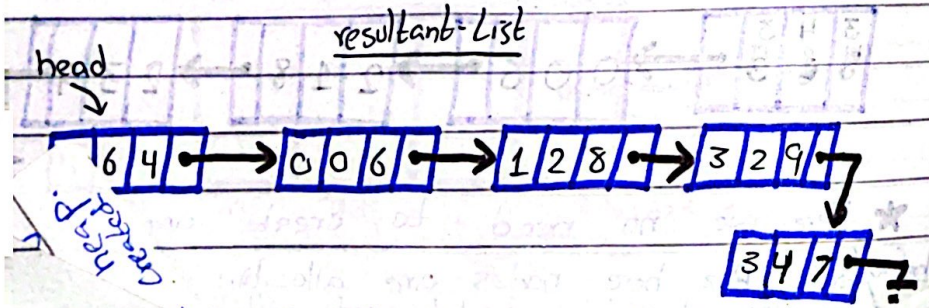
List-1



List-2



resultant-List



Heap created.

Non-Linear Data Structure

↳ **TYPES:**

- ★ Tree
- ★ Graphs
- ★ Heap
- ★ Tables

TREE

- Tree is a heranchal and non-linear data structure.
- Tree is recursively defined collection of nodes, starting at a **ROOT** node. where each node is a D.S consist of a value, together with a list of references to & nodes the **Childs/sub-t**
- Tree is also a graph.

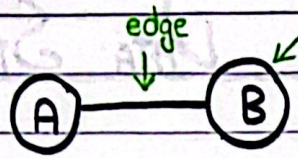
Tree Defination

It has:

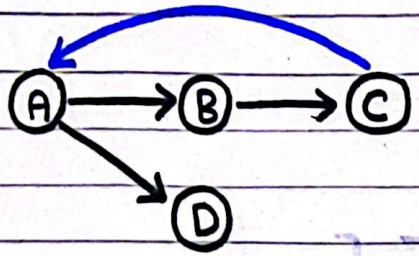
Nodes ↓ Edges ↓

$G = (N, E)$

EXAMPLE:



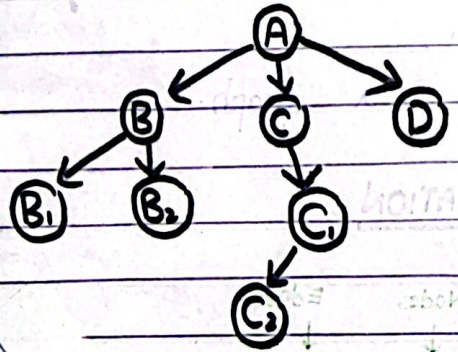
un-directed graph
(bi-directional graph)



directed-graph

NOTE:

- ★ Tree is basically directed graph.
- ★ Tree always start from "ROOT" node
- ★ After root node, tree defines itself recursively.
- ★ Tree is a "Acyclic" graph (No cylee exists like graph)



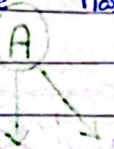
★ Tree is called "Directed graph" bcz when you go to child than you can't go to its parent back.

★ Like when you go to 'C' Node, then you can't go to 'A'.

DEGREE OF TREE

In tree every node has some degree

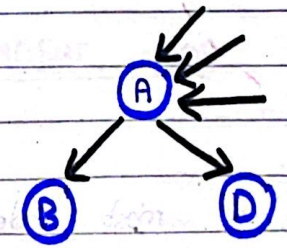
↳ DIRECTED GRAPH



IN-DEGREE & **OUT-DEGREE**

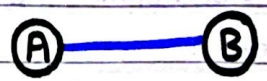
- The number of ~~childrens~~ edges arriving at a node.
- The number of edges leaving a node.

Example:



Here,
In-Degree of A =
Out-Degree of A =

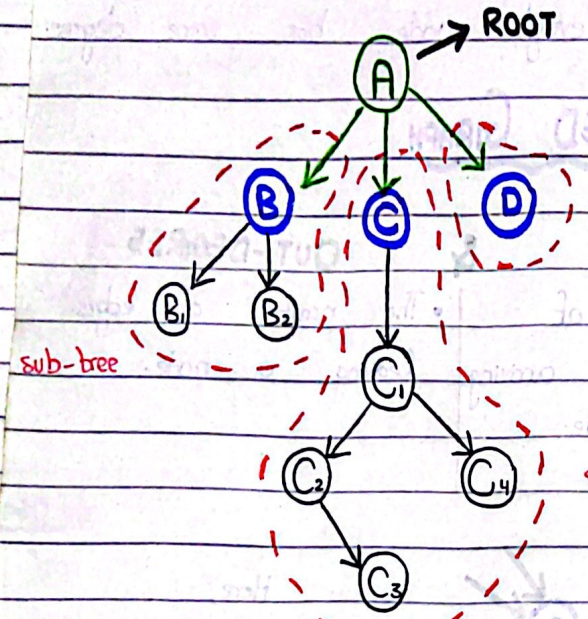
↳ UNDIRECTED GRAPH



- In-degree and Out-degree of 'A' is 1
- So, simply in undirected graph, we consi a "DEGREE", not "IN" or "OUT" bc here in-degree and out-degree are sam

DEGREE OF 'A' = 1.

EXAMPLE OF TREE



(Root is itself a sub-tree)

ROOT NODE: The top most node of a treee.

SUB-TREE: A descendant of a tree node.

PARENT NODE: The node which has a child from it.

- 'C3' is descendent of 'A'.
- 'C2' is direct parent of 'C3'.
- 'A' is not direct parent of 'C3'.

LEAF NODE: A node that has no children. (not have any branches or subtree)

- In above graph: B_1, B_2, C_3, C_4, D

These are end of tree (Leaf node). We st on leaf nodes in a t

ANCESTOR: A node that is connected t all lower level nodes.

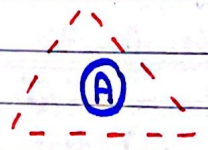
- 'A' is ancestor of 'C3'.

DESENDENT NODE: The connected lower nodes of the ancest node.

(Branches from a tree.)

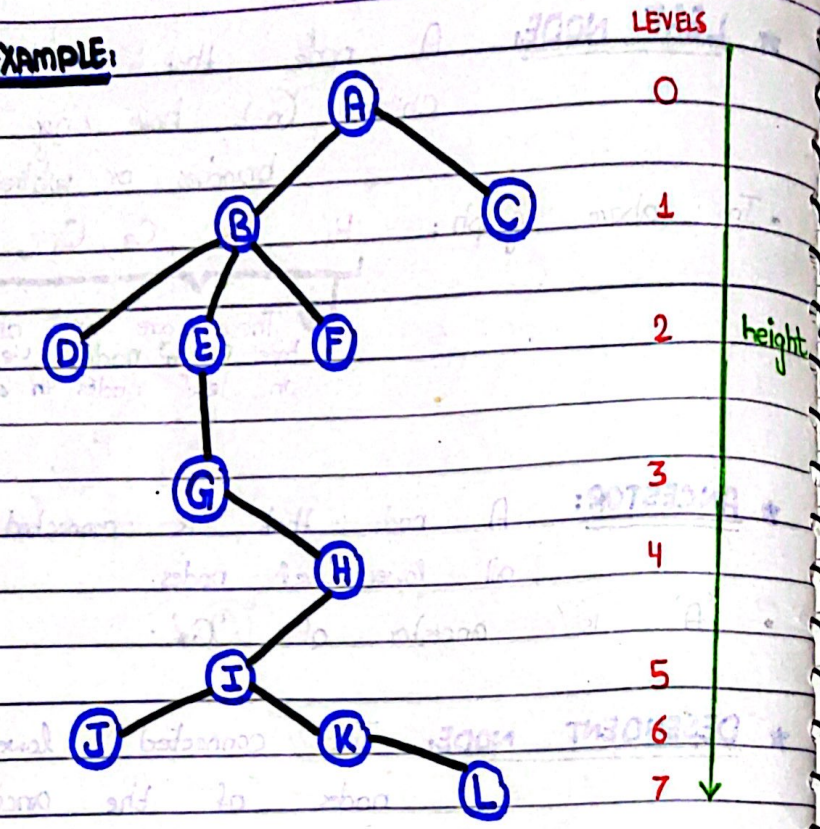
DEGREE: The numbr of children a no has.

NOTE:



- It has no left subtree and no right subtree.

EXAMPLE:



★ **LEVEL OF TREE:** Every node in the tree is assigned a level number in such a way that the **'ROOT NODE'** is at **'LEVEL 0'**. All the child nodes have level number given by parent's level number **+1**.

- where node exists.
- root always at '0' level.

- No. of levels are equal to no. of steps (edges).
- No. of levels = height of tree.

★ **PATH:** A sequence of consecutive edges. (Path is the way to go from one node to another node).
- No. of edges from one node to a destination.

Example: Path from B to L?

(B-L): B, E, G, H, I, K, L

★ **LENGTH:** length is also a path. (Count of Edges).

LENGTH (B-L): 6 (edges)

LENGTH (H-J): 2 (edges)

★ **DEPTH OF A NODE:** The depth of a node 'G' in the tree is the length of the path (no. of edges) from root of tree to node 'G'.

★ **HEIGHT OF A TREE:** A path length from root node to "DEPEST" leaf node.

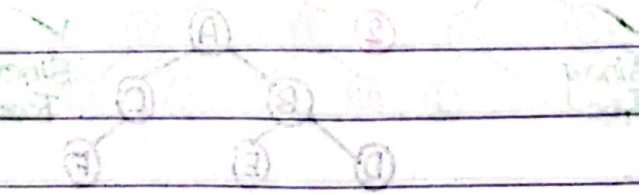
FOR ABOVE TREE:

(A - L)
↳ depest leaf node

height : 7

- height is '7' bcz level of tree is 7.

So, the height of tree is the length of the path (number of edges) from root to the "DEPEST" node in ~~thre~~ tree. ★

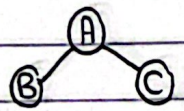


TYPES OF TREE

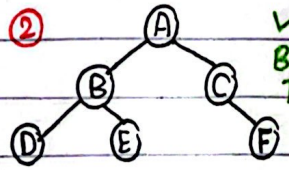
(1) BINARY TREE

* Tree whose every node would have AT MOST (maximum of) "TWO (2)" children is called binary Tree.
  - Each node has 0, 1 or atmost 2 children.
  - The children are typically named as left and right child.
  - The sub-trees of a node are typically called left-sub tree and right-sub tree.

EXAMPLES:

1. 

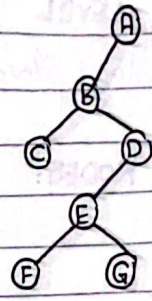
✓ Binary Tree

2. 

✓ Binary Tree

3. 

✓ Binary Tree

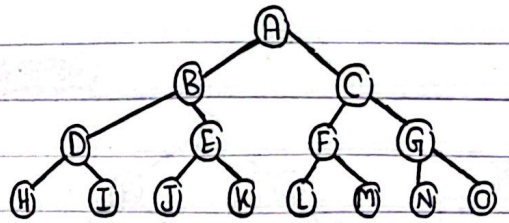
3. 

✓ Binary Tree

TYPES OF BINARY TREE:

1) FULL BINARY TREE:

* A tree in which every node, other th the leaves node have its maximum i.e 2 children.
* Every level is completed except the last level.



LEVELS	NOD
0	2^0 =
1	2^1 =
2	2^2 =
3	2^3 =
⋮	⋮
	2^k =

* It increases in multiple of '2'.

→ FOR 1st NODE AT LEVEL "0":

So, for node on particular level

FORMULA FOR CALCULATING NODES:

At Level 'k'

$2^k = n$

↳ no. of nodes

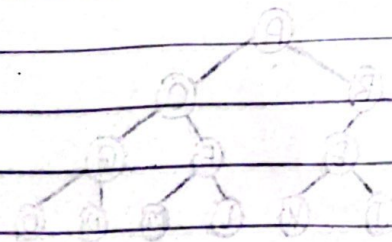
→ FOR 1st NODE AT LEVEL "1":

FORMULA FOR CALCULATING NODES:

$2^{k-1} = n$

k-1 → level

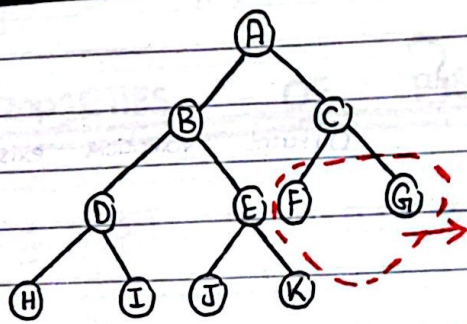
n → node



2) COMPLETE BINARY TREE

- Every level is complete except possibly the last level;
- And the last level is completed from left side.
- Last level completed itself from left side.

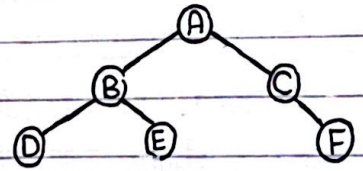
EXAMPLE:



✓ Complete Binary Tree

Last level is incomplete

EXAMPLE:



✗ Not Complete Binary Tree

NOTE:

- Every Full Binary Tree is a Complete Binary Tree
- But Every Complete Binary Tree is not a Full Binary Tree.

TRAVERSING

DISPLAY OF TREE

* Traversal is a process to visit all the nodes of a tree and may perform some operations on them too like print, update, delete etc.

* All nodes of a tree are connected via edges (Links). So, we always start from the root node.

* In tree we have multiple ends, so we cannot traverse it by only one by one.

* As tree is of recursive nature, so we move level by level.

METHODS OF TRAVERSING A TREE

A tree can be traversed in three ways:

1) In - Order Traversal
2) Pre - Order Traversal
3) Past - Order Traversal

* These traversals can also / always done in Binary Tree as well.

* In all these traversing we only see position of node in our algorithm.

1) IN-ORDER TRAVERSAL:

* In this traversal method, the "left-sub-tr is visited first, then the "root" and later the "right-sub-tree".

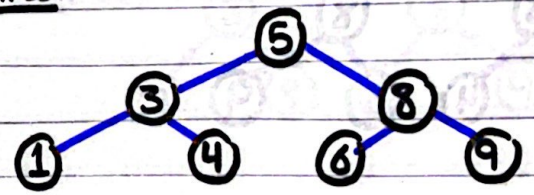
* "NODE" in mid.

Algo: LEFT (1st) → NODE (2nd) → RIGHT (3rd) (LN

↳ apply this algo recursively on each node.

* also start from root node or node from where we want to start printing.

EXAMPLE:



By this traversing, root Node will al in centre / betwe

OUTPUT: 1 3 4 5 6 8 9

NOTE: * By In Order Traversing the Binary Search Tree, The output we get is alwa "SORTED" in "ASCENDING ORDER".

2) PRE-ORDER TRAVERSAL:

* In this traversal method, the "root" is visited first, then the "left-sub-tree", and later the "right-sub-tree".

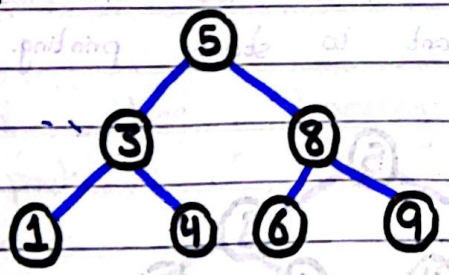
* "NODE" at start (pre).

Algo: 1st ← 2nd ← 3rd

NODE → LEFT → RIGHT (NLR)

↳ apply this algo recursively on each node.

EXAMPLE:



OUTPUT: 5 3 1 4 8 6 9

↳ By this traversing resultant/output gives root Node always at start.

3) POST-ORDER TRAVERSAL

* In this traversal method, the "left-sub-tree" is vis first, then "right-sub-tree" and later the "root".

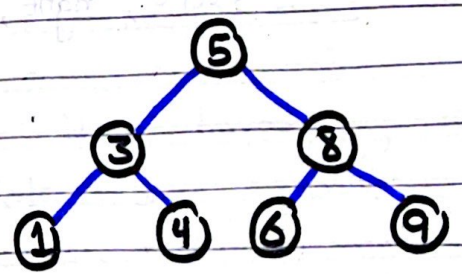
* "NODE" at End (post).

Algo: 1st 2nd 3rd

LEFT → RIGHT → NODE (LRN)

↳ apply this algo recursively a each n

EXAMPLE:



OUTPUT: 1 4 3 6 9 8 5

↳ root a at end

TREE TRAVERSAL

HOW TO REMEMBER?

* Keep (Left, right) order in your mind which is very natural. Now change the position of **ROOT**.

→ **PRE-ORDER**: (**Root**, left, right)

→ **IN-ORDER**: (left, **Root**, right)

→ **POST-ORDER**: (left, right, **Root**)

CODING POINT OF VIEW

→ To do all the above traversing, we do these by recursion. Bcz for every node after performing anything we have to go back.

So, best method is

- **RECURSNE NATURE**

or

- **STACK**

X Not Iteration

* For all these traversing, we have to write two versions:

1) **PRNATE**: Working Host (Not expose User)

2) **PUBLIC**: print (expose to User

```
class Node{
    .
    .
    .
};

class BST {
    private:
        Node * root;

    // Working Host to perform In-Order Traversal
    void inOrderTraversal ( Node * root)
    {
        // left -root -right

        if (root != NULL) {

            inOrderTraversal (root -> left);
            cout << root -> data;
            inOrderTraversal (root -> right);
        }
    }
```

Best Solution Because Tree has recursive nature.

```
    // Working Host to perform PreOrder Traversal
    void preOrderTraversal (Node * root) {
        // root - left - right

        if (root != NULL) {
            cout << root -> data;
            preOrderTraversal (root -> left);
            preOrderTraversal (root -> right);
        }
    }

    // Working Host to perform PostOrder Traversal
    void postOrderTraversal (Node* root) {
        // left - right - root

        if (root != NULL) {
            postOrderTraversal ( root -> left);
            postOrderTraversal (root -> right);
            cout << root -> data;
        }
    }
```

see file
↳ binary_search_tree_all_operations.cpp

EXPRESSION TREE

* A specific kind of **BINARY TREE** used to represents expressions (algebric and boolean)

* These trees can represent expressions that contain both unary and binay operations

RULES OF EXPRESSION TREE

- The "**OPERANDS**" always exists in the "**LEAVES**" of an expression tree.

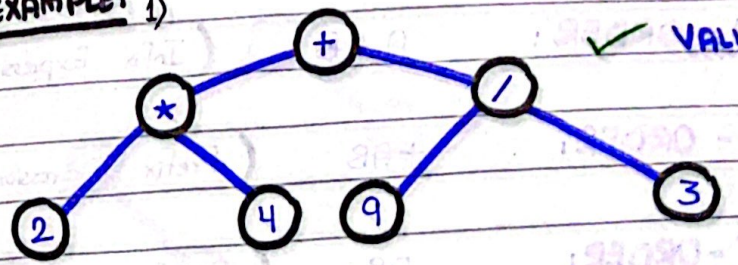
- The "**OPERATORS**" always exists in the "**NON-LEAF**" node of an expression tree.

Thus, one operand cannot be /never parent of another operand.

Similarly one operator can be attach with other operator in left sub tree

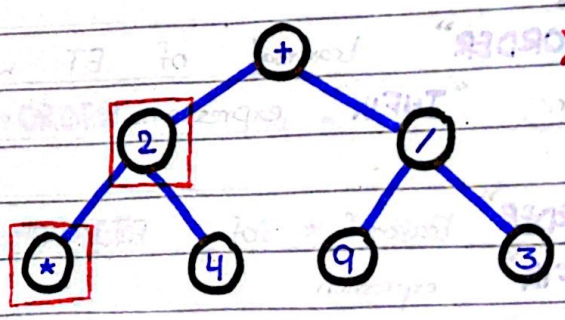
- Thus, operators can have other operators as its children in the left or right sub tree. "Moreover operator % can also have operands as its childrens."

EXAMPLE: 1)



✓ VALID ET

2)



✗ WRONG E

ADVANTAGES OF ET:

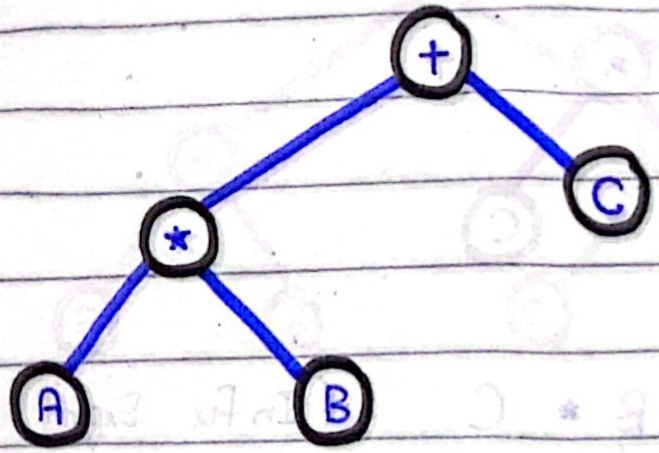
- Whenever we do In-Order Traversal o Expression Tree we always get "Infix Exp
- When we do Pre-Order Traversal of ET we get "Pre-Fix Expression."
- Post-Order Traversal of ET results i "Post-fix Expression."

EXAMPLE: A * B + C

* Here precedence and priority matters.

A * B + C

① ②



NOTE:

- The operation will be perf '1st' will de be in the lower leve tree.
- So, we will highest p operation

IN-ORDER: A * B + C (Infix E

PRE-ORDER: + * A B C (Prefix Expr

POST-ORDER: A B * C + (Postfix Expr