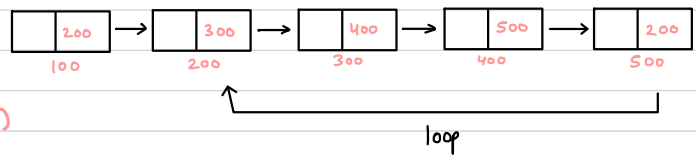
MERGING TWO LINKED LIST

We have two sorted linked list and we want to combine it into a single list.

$\Theta(m+n)$

```
if ( first -> data < second data)
{
        third = last = first;
        first = first -> next;
        last -> next = NULL;
}
else
{
        third = last = second;
        second = second -> next;
        last -> next = NULL;
}

while ( first != NULL && second != NULL)
{
        if ( first -> data < second -> data)
        {
                last -> next = first;
                last = first;
                first = first -> next;
                last -> next = NULL;
        }
        else
        {
                last -> next = second;
                last = second;
                second = second -> next;
                last -> next = NULL;
        }
}
if ( first != NULL )
        last -> next = first;
else
        last -> next = second;
```

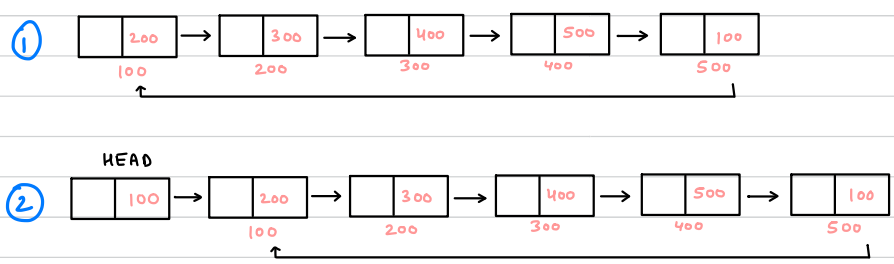
CHECK FOR LOOP IN LINKED LIST



O(n)

```
int isloop (struct Node *f)
{
    struct Node *p, *q;
    p=q=f;

    do
    {
        p = p->next;
        q = q->next;
        q = q ? q->next : q;
    } while ( p && q && p!=q);

    if (p==q)
        return 1;
    else
        return 0;
}
```

CIRCULAR LINKED LIST



DISPLAY CIRCULAR LINKED LIST

```
void Display (Node *p)
{
    do
    {
        printf (" %d", p->data);
        p=p->next;
    } while (p!= Head);
}
```

Display (Head);

DISPLAY CIRCULAR LINKED LIST USING RECURSION

```
void Display (Node *p)
{
        static int flag = 0 ;
        if ( p != Hea  ||  flag = 0)
        {
                flag = 1;
                printf("%d", p -> data);
                Display ( p -> next);
        }
}
```

CREATION OF CIRCULAR LINKED LIST

```
Void  create (int A[], int n)
{
        int  i;
        Struct Node *t, * last ;
        Head = (Struct Node *) malloc (sizeof (Struct Node));
        Head -> data = A[0];
        Head -> next = Head;
        last = Head;

        for (i= 1; i <n ; i++)
        {
                t = (struct Node *) malloc (sizeof (Struct Node));
                t -> data = A[i];
                t -> next = last -> next;
                last -> next = t;
                last = t;
        }
}

int main()
{
        int A[] = { 2,3,4,5,6};
        create (A , 5);
}
```

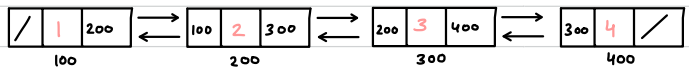
INSERTING IN A CIRCULAR LINKED LIST

```
void Insert ( struct Node *p, int index , int x)
{
        struct Node *t;
        int i;

        if (index < 0 || index > length())
                return;

        if (index == 0)
        {
                t = (struct Node *) malloc ( sizeof (struct Node));
                t -> data = x;

                if (Head == NULL)
                {
                        Head = t;
                        Head -> next = Head;
                }
                else
                {
                        while ( p -> next != Head )
                                p = p -> next;
                        p -> next = t;
                        Head = t;
                        t -> next = Head;
                }
        }

        else
        {
                for (i = 0; i < index-1 ; i++)
                        p = p -> next;
                t = (struct Node *) malloc ( sizeof (struct Node));
                t -> data = x;
                t -> next = p -> next;
                p -> next = t;
        }
}
```

DELETION IN CIRCULAR LINKED LIST

```
int Delete (struct Node *p, int index)
{
        Struct Node *q;
        int i, x;

        if (index<0 || index > length (Head))
                return -1;

        if (index == 1)
        {
                while (p->next != Head)
                        p=p->next;
                x = Head->data;

                if (Head == p)          // If there is only one node
                {
                        free (Head);
                        Head = NULL;
                }

                else
                {
                        p->next = Head->next;
                        free (Head);
                        Head = p->next;
                }
        }

        else
        {
                for (i=0 ; i < index-2; i++)
                        p = p->next;
                q = p->next;
                p->next = q->next;
                x = q->data;
                free (q);
        }
}
```

INSERT IN A DOUBLY LINKED LIST DOUBLY LINKED LIST




```
Struct Node
{
        Struct Node * prev;
        int data;
        Struct Node * next;
} * first = NULL;


void  create (int A[], int n)
{
        Struct Node *t, * last;
        int i;

        first = (Struct Node *) malloc ( sizeof ( Struct Node));
        first -> data = A[0];
        first -> prev = first -> next = NULL;
        last = first;

        for (i=1 ; i<n ; i++)
        {
                t = (Struct Node *) malloc ( sizeof ( Struct Node));
                t -> data = A[i];
                t -> next = last -> next
                t -> prev= last;
                last -> next = t;
                last = t
        }
}
```

INSERT IN A DOUBLY LINKED LIST

(1) Insert at first node

```
Node *t = new Node;
t -> data = x;
t -> prev = NULL;
t -> next = first;
first -> prev = t;
first = t;
```

(2) Insert at any given position

```
Node *t = new Node
t -> data = x;

for (i = 0; i < pos-1; i++)
        p = p -> next;
t -> next = p -> next;
t -> prev = p;
if (p -> next)
        p -> next -> prev = t
p -> next = t;
```

// To check if a node is available after p or not

DELETE FROM A DOUBLY LINKED LIST

(1) Deleting first node

```
p = first;
first = first -> next;
x = p -> data;
delete p;
if (first)      // If first is not NULL
    first -> prev = NULL;
```

(2) Deleting Node from given index

```
p = first;
for (i=0; i<pos-1; i++)
        p = p -> next;
p -> prev -> next = p -> next;

if (p -> next)
        p -> next -> prev = p -> prev;

x = p -> data;
delete p;
```

if p -> next is not NULL

REVERSING A DOUBLY LINKED LIST

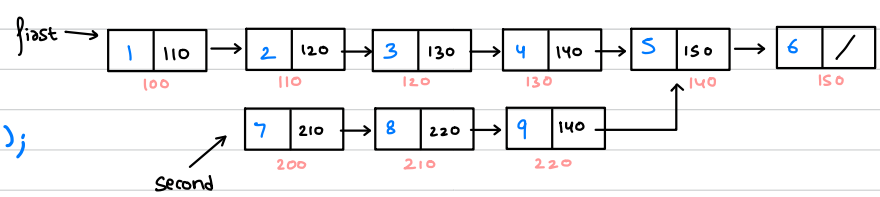
```
p = first;

while (p)
{
        temp = p->next;
        p->next = p->prev;
        p->prev = temp;
        p = p->prev

        if ( p != NULL && p->next = NULL)
                first = p;
}
```

FINDING MIDDLE NODE IN A LINKED LIST

```
p = q = first;

while (q)
{
        q = q->next;
        if (q)
            q = q->next;
        if (p)
            p = p->next;
}
```

Two pointers p and q will move at the same time.
q will move 2 nodes and p will move 1 node.

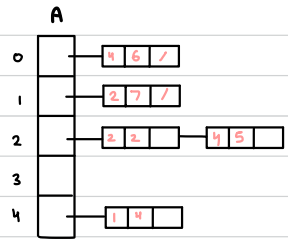
FINDING INTERSECTION POINT OF TWO LINKED LIST




```
p = first
while ( p != NULL)
        push (&stk1, p);
p = second;
while ( p != NULL)
        push (&stk2, p);
while ( stacktop (stk1) == stacktop (stk2))
{
        p = pop(&stk1);
        pop (&stk2);
}
```

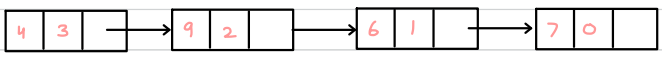
SPARSE MATRIX USING LINKED LIST

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 0 & 0 & 6 & 0 & 0 \\
2 & 0 & 7 & 0 & 0 & 0 & 0 \\
3 & 0 & 2 & 0 & 5 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 4 & 0 & 0 & 0 & 0 & 0
\end{array}
$$



POLYNOMIAL REPRESENTATION USING LINKED LIST

$p(x) = 4x^3 + 9x^2 + 6x + 7$



Recursion uses stack

STACK

↳ LIFO : Last In First Out

It is a collection of elements that follow LIFO for insertion and deletion.

ADT Stack (Abstract data type)

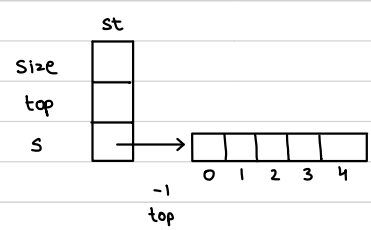
Data :
1. Space for storing elements
2. Top pointer

Implementation of stack using
1. Array
2. Linked List

Operations:
1. push(x)
2. pop()
3. peek(index)
4. StackTop()
5. isEmpty()
6. isFull()

IMPLEMENTATION OF STACK USING ARRAY

```
Struct stack
{
    int size;
    int top;
    int *s;
};

int main()
{
    Struct stack st;
    printf("Enter size of stack");
    scanf("%d", &st.size);
    st.s = new int[st.size];   for heap memory
    st.top = -1;
}
```



Stack empty — if (top == -1)
Stack full - if (top == size-1)

push() O(1)

```
void push (Stack *st, int x)
{
        if ( st -> top == st -> size -1)
                printf (" Stack Overflow");
        else
        {
                st -> top++;
                st -> s [st -> top] = x;
        }
}
```

pop() O(1)

```
int pop (Stack *st)
{
        int x = -1;

        if ( st -> top == -1)
                printf (" Stack Underflow");
        else
        {
                x = st -> s[st -> top];
                st -> top--;
        }
        return x;
}
```

peek()

```
int peek( Stack st, int pos)
{
        int x = -1;

        if ( st top - pos +1 < 0)
                printf (" Invalid Position");
        else
                x = st.s [st.top - pos+1];

        return x;
}
```

pos	Index = Top - pos +1
1	3
2	2
3	1
4	0

Stacktop()

```
int Stacktop (Stack st)
{
        if (st.top == -1)
                return -1;
        else
                return st.s [st.top];
}
```

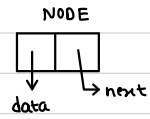
isEmpty()

```
int isEmpty (Stack st)
{
        if (st.top == -1)
                return 1;
        else
                return 0;
}
```

isFull()

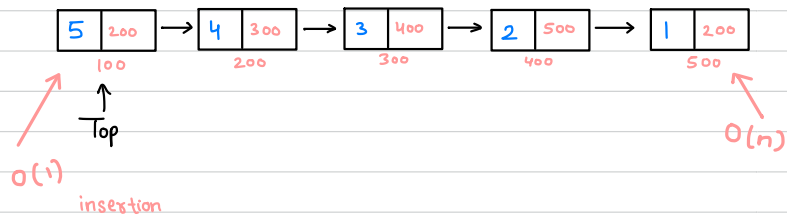
```
int isFull ( Stack st)
{
        if (st.top == -1)
                return 1;
        else
                return 0;
}
```

STACK USING LINKED LIST



Empty : if (top == NULL)
Full : Node *t = new Node;
if (t==NULL)

```
Struct Node
{
    int data;
    Struct Node *next;
}
```



push()

```
void push( int x)
{
    Node * t = new Node;

    if (t==NULL)
        printf("Stack Overflow");
    else
    {
        t -> data = x;
        t -> next = top;
        top = t;
    }
}
```

pop()

```
int pop()
{
    Node *p;
    int x=-1;

    if (top == NULL)
        printf("Stack is empty");
    else
    {
        p = top;
        top = top->next;
        x = p -> data;
        free(p);
    }
    return x;
}
```

peek()

```
int Peek (int pos)
{
    int i;
    Node *p = top;

    for (i=0 ; p != NULL && i<pos-1 ; i++)
        p = p->next;

    if ( p != NULL)
        return p->data;
    else
        return -1;
}
```

Stacktop()

```
int  Stacktop()
{
        if(top)
            return  top->data;
        return  -1;
}
```

isFull

```
int  isFull()
{
        Node *t = new  Node;
        int  r = t ? 1 : 0;
        free(t);
        return  r;
}
```

isEmpty

```
int  isEmpty()
{
        return  Top ? 0 : 1;
}
```

False

↳ True

Paranthesis Matching

exp

(	(	a	+	b	)	*	(	(	−	d	)	)	\0
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
int  isBalance(char *exp)
{
        Struct  Stack  st;
        St.size = Strlen(exp);
        St.top = -1;
        St.s = new  char[st.size];

        for(i=0; exp[i]!='\0'; i++)
        {
                if(exp[i]=='(')
                        push(&St, exp[i]);
                else if(exp[i]==')')
                        if(isEmpty(st))
                                return false;
                        else
                                pop(&St);
        }
        return isEmpty(St)? true : false;
}
```

initializing of stack

ASCII

(	40
)	41
[	91
]	93
{	123
}	125

INFIX TO POSTFIX CONVERSION

1. What is postfix
2. Why postfix
3. Precedence
4. Manual Conversion

1. **Infix:** Operand Operator Operand
   a + b

2. **Prefix:** Operator Operand Operand
   +ab

3. **Postfix:** Operand Operand Operator
   ab +

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, −	1	L-R
*, /	2	L-R
^	3	R-L
—	4	R-L
()	5	L-R

Unary minus

Eg a + b*c

(a + (b*c)) First fully paranthecise it.

prefix

(a + [*bc])
[+a*bc]

postfix

(a + [bc*])
[abc*+]

ASSOCIATIVITY

Left to Right

a+b+c−d

(((a+b)+c)−d)

Right to Left

a = b = c = 5

(a = (b=(c = 5)))

Power operator Example

a^ b ^c

(a^(b^c))

Postfix: (a^[bc^])

abc^^

Unary Operators Example

(1) −a negation of a

pre : −a

post : a−

(−(− a))

(2) *p

pre : *p

post : p*

(*(* p))

(3) n!

pre : !n

post : n!

(4) logx

pre : logx

post : xlog

Example : −a + b * logn !

−a + b * log [n!]

− a + b * [n! log]

[a−] + b * [n! log]

[a−] + [bn! log*]

a − bn! log* +

INFIX TO POSTFIX CONVERSION

a + b * c − d /e

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, −	1	L-R
*, /	2	L-R

Symbol	Stack	Postfix
a		a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
−	−	abc *+
d	−	abc *+d
/	/, −	abc *+d
e	/, −	abc *+ de

abc * +de /− Ans

PROGRAM

infix	a	+	b	*	c	−	d	/	e	\0
	0	1	2	3	4	5	6	7	8	9

```
char *convert( char *infix)
{
        struct stack st;      // Initialized
        char *postfix = new char[strlen(infix+1)];
                                                  → for null string
        int i=0; j=0;

        while( infix[i] != '\0')
        {
                if (isOperand(infix[i]))
                        postfix[j++] = infix[i++];

                else
                {
                        if (pre(infix[i]) > pre(stacktop(st))
                                push(&st, infix[i++]);
                        else
                                postfix[j++] = pop(&t);
                }
        }

        while ( ! isEmpty(st))
                postfix[j++] = pop(&st);

        postfix[j] = '\0';
        return postfix;
}
```

```
int pre(char x)
{
        if ( x == '+' || x == '-')
                return 1;
        else if (x == '*' || x == '/')
                return 2;
        return 0;
}
```

```
int isOperand(char x)
{
        if (x == '+' || x == '-' || x == '*' || x == '/')
                return 0;
        else
                return 1;
}
```

Q $((a+b)*c)-d\wedge e\wedge f$

$([ab+]*c)-d\wedge e\wedge f$

$[ab+c*]-d\wedge e\wedge f$

$[ab+c*]-d\wedge[ef\wedge]$

$[ab+c*]-[def\wedge\wedge]$

$ab+c*def\wedge\wedge-$

SYMBOL	OUT STACK PRE	IN STACK PRE
+, −	1	2
*, /	3	4
∧	6	5
(	7	0
)	0	?

because of R-L associativity

closing bracket cannot be pushed into Stack

EVALUATION OF POSTFIX

$35*62/+4-$

SYMBOL	STACK	OPERATION
3	3	
5	5, 3	
*		5*3
	15	
6	6, 15	
2	2, 6, 15	
/		6/2
	3, 15	
+		15+3
	18	
4	4, 18	
−		18−4
	14	

$x = 6+5+3*4$

$x = 65+34*+$

* Here + gets executed first instead of * because presedence and associativity are meant for parenthesisation, they don't decide which operator gets executed first.

PROGRAM FOR EVALUATION OF POSTFIX

postfix	3	5	*	6	2	/	+	4	−	\0
	0	1	2	3	4	5	6	7	8	9

```
int Eval( char *postfix)
{
        struct stack st;
        int i, x1, x2, r;

        for ( i=0; postfix[i] != '\0'; i++)
        {
                if ( isOperand ( postfix[i]))
                        push(&st, postfix[i]-'0');
                else
                {
                        x2 = pop(&st);
                        x1 = pop(&st);

                        switch ( postfix[i])
                        {
                                case '+' : r = x1+x2 ;  push(&st,r ; break;
                                case '-' : r = x1 - x2 ;  push(&st,r ; break;
                                case '*' : r = x1 * x2 ;  push(&st,r ; break;
                                case '/' : r = x1 / x2 ;  push(&st,r ; break;
                        }
                }

                return pop(&st);
}
```

because operand will be pushed into the stack in its ASCII value because postfix expression is in char.

For eg : 3

'51' − '48' = 3

QUEUE

↳ FIFO
(First In First Out)

Queue ADT

Data: (1) Space for storing elements
(2) Front : for deletion
(3) Rear : for insertion

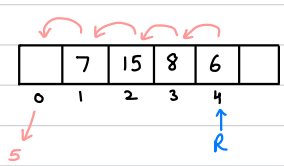
Operations: (1) enqueue (x)
(2) dequeue ()
(3) isEmpty()
(4) isfull ()
(5) first()
(6) last()

(1) Array
(2) Linked List

QUEUE USING ARRAY

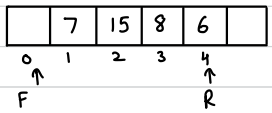
1. Queue using single pointer
2. Queue using front and rear.
3. Drawbacks of queue using array.

1. Queue using single pointer



Insert – O(1)
Delete – O(n)

2. Queue using front and rear



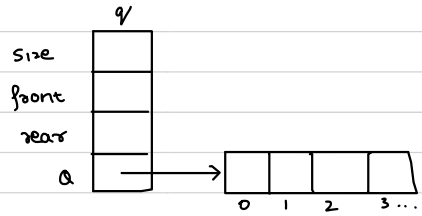
initially front = rear = -1

Insertion : increment rear and insert O(1)
Deletion : increment front and delete O(1)

Empty : if (front == rear)
Full : if (rear = size-1)

PROGRAM FOR QUEUE USING ARRAY

```
Struct Queue
{
    int size;
    int front;
    int rear;
    int * Q;
}
```

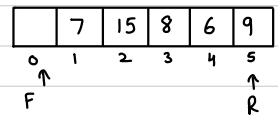



```
int main ()
{
    Struct Queue q;
    printf(" Enter Size: ");
    scanf (" %d", &q.size);
    q.Q = new int[q size];
    q.front = q.rear = -1;
}

void enqueue (Queue *q , int x)
{
    if ( q->rear == q->size-1)
        printf(" Queue is full");
    else
    {
        q->rear++;
        q->Q[q->rear] = x;
    }
}
```

```
void dequeue (Queue *q)
{
    int x = -1;

    if (q->front == q->rear)
        printf(" Queue is empty");
    else
    {
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
```

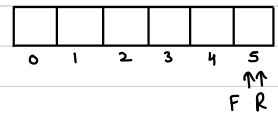
DRAWBACK OF QUEUE USING ARRAY

1. We cannot utilize space of deleted element.



2. Every location can be used only once

3. A situation where queue is empty and full



USING SPACE AGAIN SOLUTION

(1) Resetting Pointers

Whenever Front and Rear are pointing at same place, initialize them as -1

(2) Circular Queue

In circular queue, front and rear initialize with array's first position.

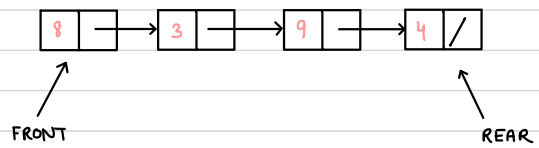
```
void enqueue( struct Queue *q , int x)
{
    if ((q -> Rear+1) % q -> Size == q -> front)
            printf(" Queue is Full ");
    else
    {
            q -> Rear = (q -> rear +1) % q -> size;
            q -> Q [ q -> rear] = x;
    }
}
```

To check if rear's next is front

rear =	(rear +1) % size
0	(0 +1) % 7 = 1
1	(1+1) % 7 = 2
2	(2+1) % 7 = 3
3	(3+1) % 7 = 4
4	(4+1) % 7 = 5
5	(5+1) % 7 = 6
6	(6 +1) % 7 = 0

```
void dequeue ( struct Queue *q)
{
    int x = -1
    if ( q -> front == q -> rear)
            printf(" Queue is Empty");
    else
    {
            q -> front = (q -> front+1) % q -> size;
            x = q -> Q [ q -> front];
    }
    return x;
}
```

QUEUE USING LINKED LIST



Empty : if (front == NULL)

Full : Node *t = new Node;
if (t == NULL)
// No more nodes can be created
i.e heap is full

```
void enqueue (int x)      O(1)
{
        Node *t = new Node;
        if (t == NULL)
                printf(" Queue is Full");
        else
        {
                t -> data = x;
                t -> next = NULL;
                if (front == NULL)
                        front = rear = t;
                else
                {
                        rear -> next = t
                        rear = t;
                }
        }
}

int dequeue()      O(1)
{
        int x = -1;
        Node *p;
        if (front == NULL)
                printf(" Queue is Empty");
        else
        {
                p = front;
                front = front -> next;
                x = p -> data;
                free (p);
        }
        return x;
}
```

DEQueue

↳ Double Ended Queue

QUEUE

	Insert	Delete
front	✗	✓
rear	✓	✗

DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	✓

INPUT RESTRICTED DEQUEUE

	Insert	Delete
front	✗	✓
rear	✓	✓

OUTPUT RESTRICTED DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	✗

PRIORITY QUEUES

1. Limited Set of priorities
2. Element Priority

1. Limited Set of priorities

Priorities = 3

Element →	A	B	C	D	E	F	G	H	I	J
Priority →	1	1	2	3	2	1	2	3	2	2

Priority Queues

Q_1	A	B	F			
Q_2	C	E	G	I	J	
Q_3	D	H				

INSERTION

DELETION

For deletion, elements of Q_1 will be deleted first, then Q_2 and then Q_3

Elements will be deleted in FIFO

2. Element Priority

Elements → 6, 8, 3, 10, 15, 2, 9, 17, 5, 8

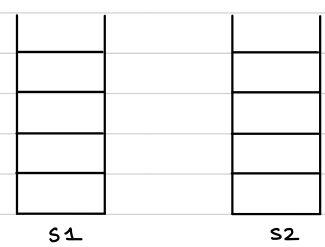
- Element is itself a priority
- Smaller number higher priority

1. Insert in same order O(1)
   Delete max priority by searching it O(n)

2. Insert in increasing order of priority O(n)
   Delete the last element of array O(1)

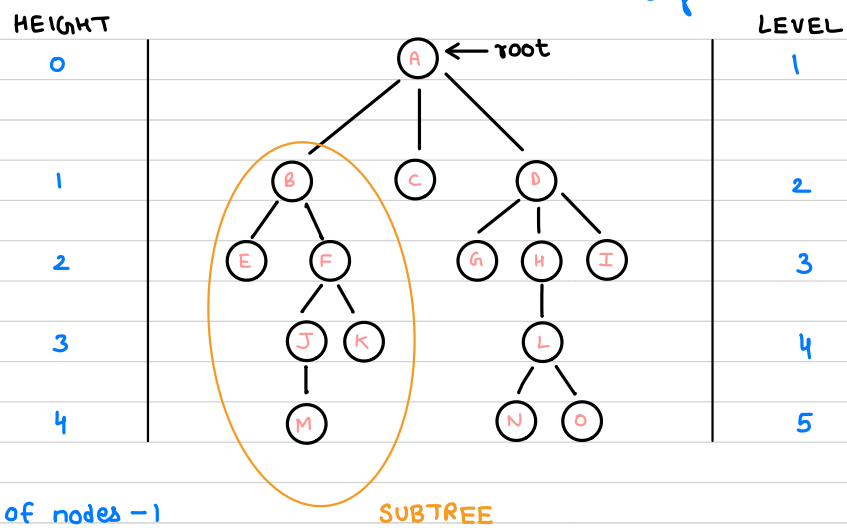
QUEUE using 2 STACKS

assume stacks to be implemented using linked list




```
enqueue ( int x)
{
        push ( &S1, x);
}


int  dequeue()
{
        int  x = -1;
        if ( isEmpty (S2))
        {
                if ( isEmpty S1)
                {
                        printf (" Queue Empty");
                        return  x;
                }
                else
                {
                        while (! is Empty (S1))
                                push (&S2, pop (&S1));
                }
        }
        return  pop ( &S2);
}
```

TREES → Tree is a collection of nodes and edges.



No of edges = No of nodes − 1

TERMINOLOGY

(1) **Root** : The first node on the top

(2) **Parent** : A node is a parent to its very next descendents.

(3) **Child**

(4) **Sibling** : Children of same parent G, H, I

(5) **Descendents** : Set of all those nodes which can be reached from a particular node.
E, F, J, K, M are descendents of B.

(6) **Ancestors** : For any node, all the nodes along the path from that node to root node
For M, J, F, B, A are ancestors.

(7) **Degree of a node** : No of children it is having L − 2

(8) **Internal Nodes / Non-Leaf Nodes / Non-Terminal Nodes** : Nodes with degree > 0

External Nodes / Leaf-Nodes / Terminal Nodes : Nodes with degree 0

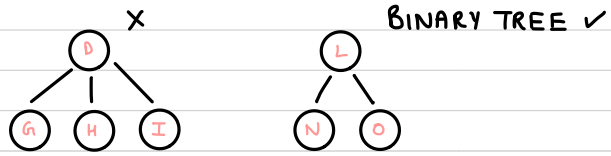
(9) **Levels** (10) **Height**

(10) **Forest** : Collection of trees.

BINARY TREE

degree {2}

children – 0, 1, 2

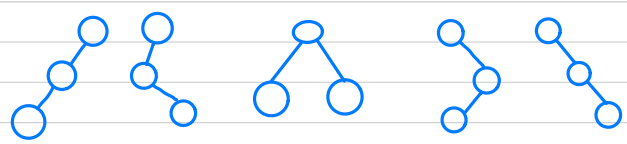


NUMBER OF BINARY TREES USING N NODES

(1) Unlabelled Nodes
(2) Labelled Nodes

No of trees with maximum height $= 4 = 2^2$
(2)

$n = 3$



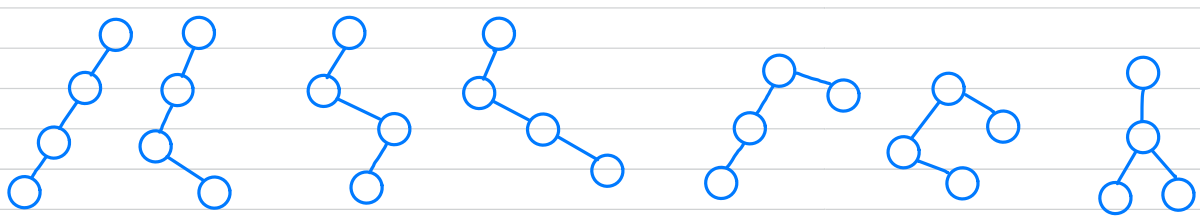
height = 2

mirror images

$T(3) = 5$

$n = 4$

No of trees with maximum height $= 8 = 2^3$



$T(4) = 14$

+ other 7 mirror images

$$T(n) = \frac{{}^{2n}C_n}{n+1}$$

Catalan Number

No of trees with maximum height $= 2^{n-1}$

2nd Method for finding Catalan Number

n	0	1	2	3	4	5	6
$T(n) = \frac{{}^{2n}C_n}{n+1}$	1	1	2	5	14	42	

$$T(6) = 1 \times 42 + 1 \times 14 + 2 \times 5 + 5 \times 2 + 14 \times 1 + 42 \times 1$$

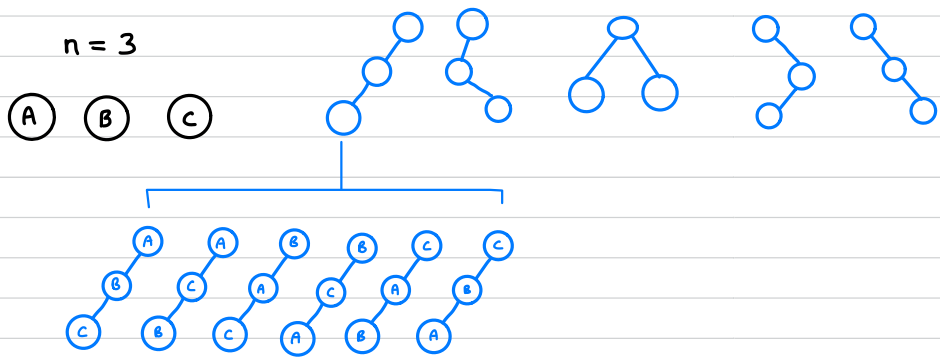
$$T(6) = T(0) \times T(5) + T(1) \times T(4) + T(2) \times T(3) + T(3) \times T(2) + T(4) \times T(1) + T(5) \times T(0)$$

$$= 132$$

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

(2) Labelled Nodes

$n = 3$



$6 = 3!$

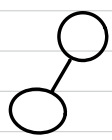
$$T(n) = \frac{{}^{2n}C_n}{n+1} * n!$$

Shapes Filling

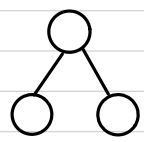
Height vs Node

Height
$h = 1$

min

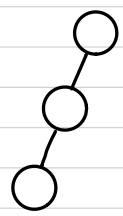


max

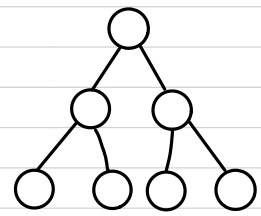


Height
$h = 2$

min

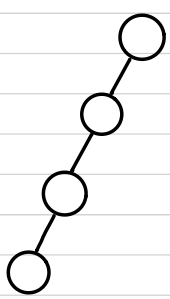


max

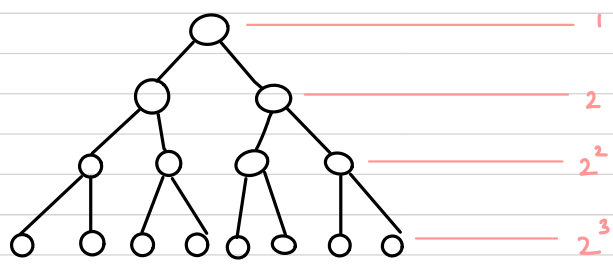


Height $h = 3$

min



max



$1 + 2 + 2^2 + 2^3 = 15$

Minimum Number of Nodes

min nodes $= h+1$

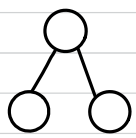
Maximum Number of Nodes

$$a + ar + ar^2 + ar^3 + ar^k = \frac{a(r^{k+1} - 1)}{r-1}$$

$$1 + 2 + 2^2 + 2^3 + \ldots\ldots 2^h = \frac{1 \cdot (2^{h+1} - 1)}{2-1} = 2^{h+1} - 1$$

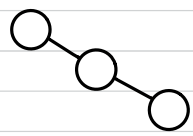
Nodes $n = 3$

min



$h = 1$

max



$h = 2$

If nodes are given

(1) Max height $h = n-1$

Can be obtained by previous formula

(2) For minimum height

$$\Rightarrow n = 2^{h+1} - 1$$

$$\Rightarrow 2^{h+1} = n+1$$

$$\Rightarrow h+1 = \log_2(n+1)$$

$$\Rightarrow h = \log_2(n+1) - 1$$

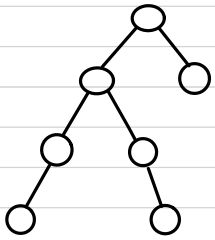
Height of a binary tree

$$\underbrace{\log_2(n+1) - 1}_{O(\log n)} \le h \le \underbrace{n-1}_{O(n)}$$

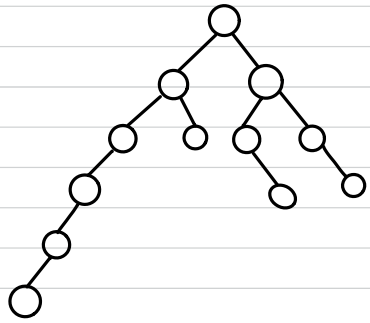
Number of Nodes in a binary tree

$$h+1 \le n \le 2^{h+1} - 1$$

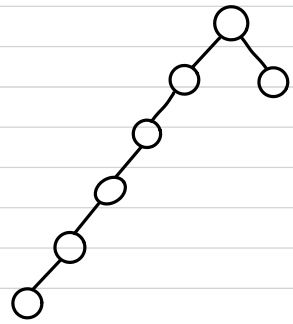
INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE



$deg(2) = 2$
$deg(1) = 2$
$deg(0) = 3$



$deg(2) = 3$
$deg(1) = 5$
$deg(0) = 4$

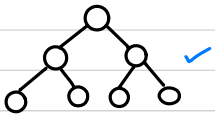


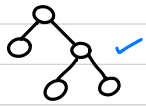
$deg(2) = 1$
$deg(1) = 4$
$deg(0) = 2$

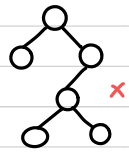
$\boxed{deg(0) = deg(2) + 1}$ True in binary tree

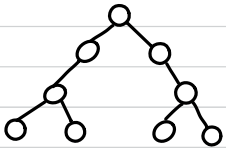
STRICT / PROPER / COMPLETE BINARY TREES

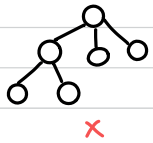
$deg = \{ \overset{\checkmark}{0}, \overset{\times}{1}, \overset{\checkmark}{2} \}$

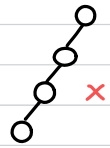








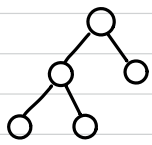




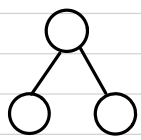
Height vs Node (Strict Binary Tree)

Height
$h = 1$

min

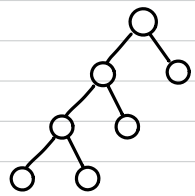


max

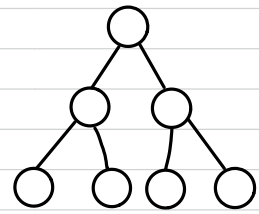


Height
$h = 2$

min

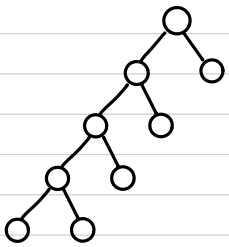


max

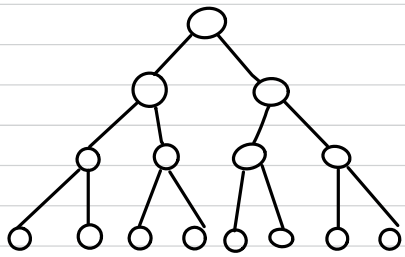


Height $h = 3$

min



max



Only minimum number changed

If height 'h' is given

Min Nodes $n = 2h+1$

Max Nodes $n = 2^{h+1} - 1$

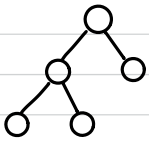
If 'n' nodes are given

Minimum height $h = \log_2(n+1) - 1$

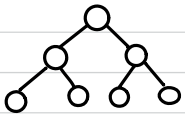
Max height $h = \frac{n-1}{2}$

$$\log_2(n+1) - 1 \le h \le \frac{n-1}{2}$$

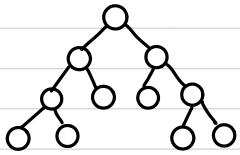
INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE (STRICT)



$i = 2$
$e = 3$



$i = 3$
$e = 4$

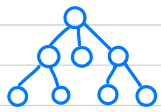


$i = 5$
$e = 6$

$\boxed{e = i + 1}$

n-ary Trees

3-ary Tree $\{0,1,2,3\}$



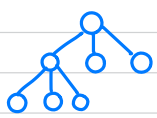
4-ary Tree $\{0,1,2,3,4\}$

Strict 3-ary Tree $\{0,3\}$

Strict n-ary trees

$h = 2$

min



$n = 3+3+1$

$n = 2 \times 3 + 1$

$i = 2$

$e = 5$

max

$i = 5$

$e = 9$

If height 'h' is given

Min Nodes $n = nh+1$

Max Nodes $n = \dfrac{m^{h+1}+1}{m-1}$

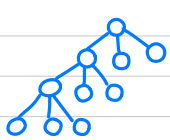
If 'n' nodes are given

Minimum height $h = \log_m[n(m-1)+1]-1$

Max height $h = \dfrac{n-1}{m}$

$h = 3$

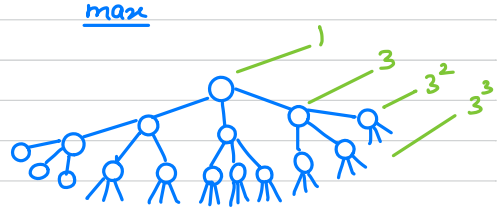
min



$i = 3$

$e = 7$

max



$\Rightarrow 1 + 3 + 3^2 + 3^3$

$\Rightarrow 1 + n + n^2 + n^3$

$\Rightarrow \dfrac{(n^{h+1}-1)}{n-1}$

$i = 13$

$e = 27$

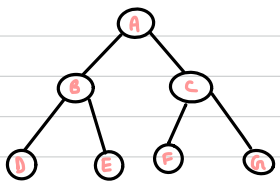
$e = 2i+1$

$\boxed{e = (n-1)i+1}$

REPRESENTATION OF BINARY TREE

1. Array Representation
2. Linked Representation

1. Array Representation



A	B	C	D	E	F	G
1	2	3	4	5	6	7

Element	Index	L. Child	R. Child
A	1	2	3
B	2	4	5
C	3	6	7
	i	$2*i$	$2*i+1$

Element $\longrightarrow$ i

L Child $\longrightarrow$ $2*i$

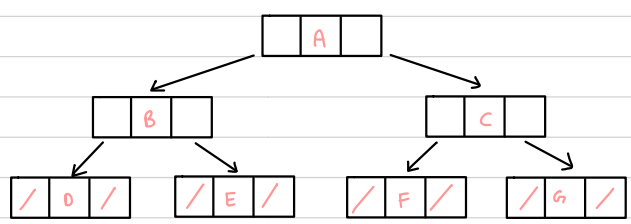
R Child $\longrightarrow$ $2*i+1$

Parent $\longrightarrow$ $\left\lfloor \frac{i}{2} \right\rfloor$

2 Linked Representation

NODE

lchild	NODE	rchild



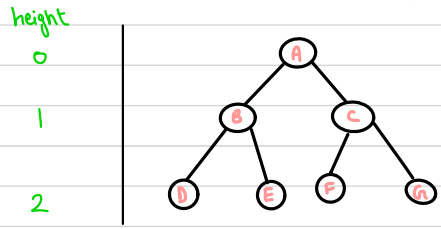

```
Struct Node
{
    Struct Node *lchild;
    int data;
    Struct Node *rchild;
};
```

$n = 7$

nulls $= n+1$

Full Binary Tree: A binary tree of height h having maximum number of nodes.

Eg

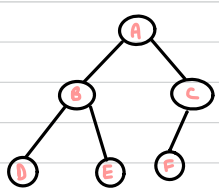


A	B	C	D	E	F	G
1	2	3	4	5	6	7

Complete Binary Tree: A complete binary tree of height h will be a full binary tree upto height h-1

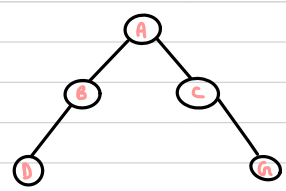
A full binary tree is always a complete binary tree.

Eg



A	B	C	D	E	F	
1	2	3	4	5	6	7

NO BLANK SPACES BETWEEN ELEMENTS



NOT A COMPLETE BINARY TREE

A	B	C	D			G
1	2	3	4	5	6	7

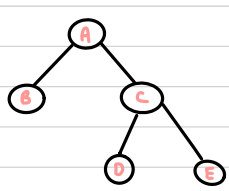
BLANK SPACES

degree
{0, 2}

STRICT VS COMPLETE

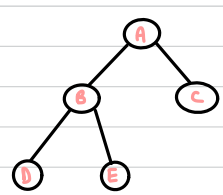
↓ complete

↳ almost complete



A	B	C			D	E
1	2	3	4	5	6	7

Strict ✓
complete ✗



A	B	C	D	E		
1	2	3	4	5	6	7

Strict ✓
complete ✓

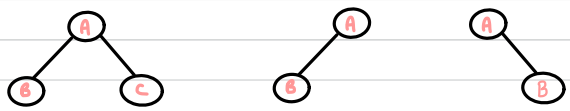
TREE TRAVERSALS

Preorder : NLR
Node , Left , Right

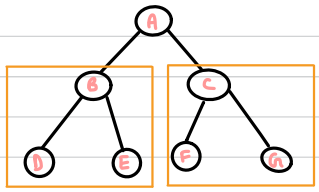
Inorder : LNR
Left , Node , Right

Postorder : LRN
Left , Right , Node

Level Order : Level by Level



Preorder	ABC	AB	AB
Inorder	BAC	BA	AB
Postorder	BCA	BA	BA
Level	ABC	AB	AB



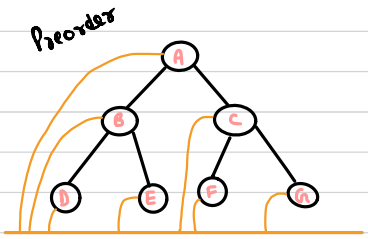
Preorder A , (B, D, E) , (C, F, G)
A, B, D, E , C , F, G

Inorder (D, B, E), A, (F , C , G)
D, B, E , A , F , C, G

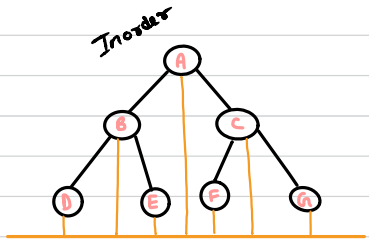
Postorder (D, E , B) , (F, G , C) , A
D, E , B, F , G , C, A

Level A , B, C , D, E , F , G

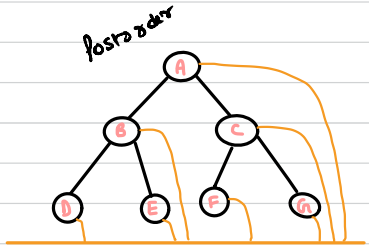
BINARY TREE TRAVERSAL EASY METHOD 1



A, B, D, E, C, F, G

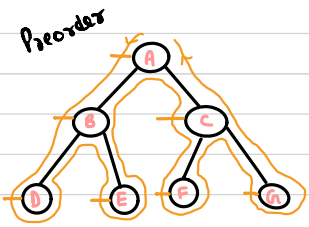


D, B, E, A, F, C, G

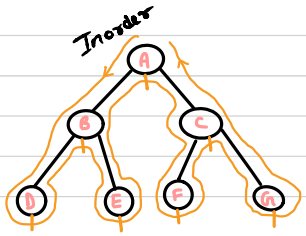


D, E, B, F, G, C, A

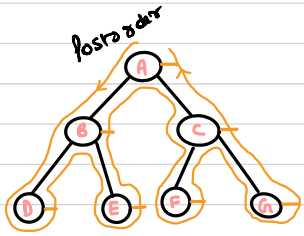
BINARY TREE TRAVERSAL EASY METHOD 2



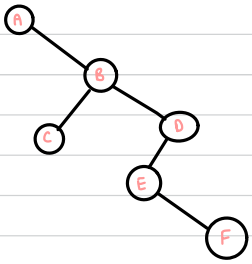
A, B, D, E, C, F, G



D, B, E, A, F, C, G



D, E, B, F, G, C, A



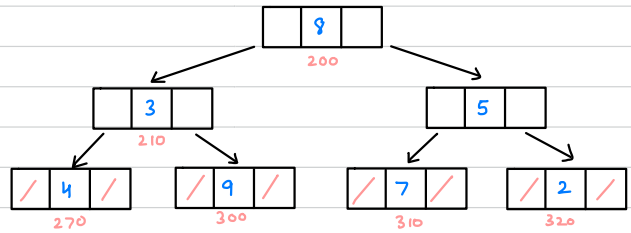
Preorder : A, B, C, D, E, F
Inorder : A, C, B, E, F, D
Postorder : C, F, E, D, B, A

PROGRAM TO CREATE BINARY TREE

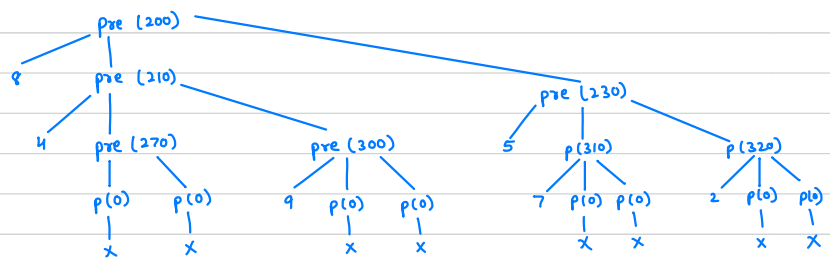
```
void create()
{
        Node *p, *t;
        int x;
        Queue q;        initializing a queue
        printf("Enter root value");
        scanf("%d", &x);
        root = malloc(.....);        initializing root
        root -> data = x;
        root -> lchild = root -> rchild = 0;
        enqueue(root);   Storing address of root in queue

        while (!isEmpty(q))
        {
                p = dequeue(&q);  take out a value from queue into p
                printf("Enter left child");
                scanf("%d", &x);

                if (x != -1)
                {
                        t = malloc(....);
                        t -> data = x; t -> lchild = t -> rchild = 0;
                        p -> lchild = t;      p -> rchild = t      for right child everything
                        enqueue(t);                                else will be same
                }
        }
}
```

RECURSIVE PREORDER TREE TRAVERSAL

```
void Preorder (Node *t)
{
    if (t != NULL)
    {
        printf(" %d", t -> data);
        Preorder (t -> lchild);
        Preorder (t -> rchild);
    }
}
```



TRACING TREE



ACTIVATION RECORDS ARE CREATED USING STACK

RECURSIVE INORDER TREE TRAVERSAL

```
void Inorder (Node *t)
{
    if (t != NULL)
    {
        Inorder (t -> lchild);
        printf(" %d", t -> data);
        Inorder (t -> rchild);
    }
}
```

RECURSIVE POSTORDER TREE TRAVERSAL

```
void Postorder (Node *t)
{
    if (t != NULL)
    {
        Postorder (t -> lchild);
        Postorder (t -> rchild);
        printf(" %d", t -> data);
    }
}
```

ITERATIVE PREORDER TRAVERSAL

```
void Preorder ( Node *t)
{
        Struct Stack st;
                                to check for empty stack
        while ( t! = NULL || isEmpty (st) )
        {
                if (t! = NULL)
                {
                        printf(" %d ", t -> data);
                        push (&st , t); push address of t in stack
                        t = t -> lchild;
                }

                else
                {
                        t = pop (&st); pop out address
                        t = t -> rchild; from stack
                }
        }
}
```

ITERATIVE INORDER TRAVERSAL

```
Inorder ( Node *t)

push (&st , t);
t = t -> lchild;

t = pop (&st);
printf(" %d ", t -> data);
t = t -> rchild;
```

ITERATIVE POSTORDER TRAVERSAL*

```
void Inorder ( Node *t)
{
        Struct Stack st;
        long int temp;
        while ( t! = NULL || isEmpty (st))
        {
                if (t! = NULL)
                {
                        push (&st, t);
                        t = t -> lchild;
                }
                else
                {
                        temp = pop (&st);
                        if( temp > 0)
                        {
                                push (&st, -temp);
                                t =((Node *) temp) -> rchild;
                        }
                        else
                        {
                                printf("%d",((Node *) temp)
                                                -> data)
                                t = NULL;
                        }
                }
        }
}
```

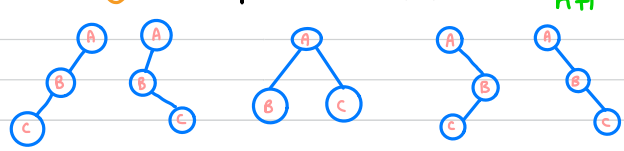
LEVEL ORDER TRAVERSAL

```
void Levelorder (Node *p)
{
        Queue q;
        printf("%d", p->data);
        enqueue (&q, p);

        while( ! isEmpty (q))
        {
                p = dequeue (&q);
                if (p -> lchild)
                {
                        printf("%d", p -> lchild -> data);
                        enqueue( &q, p -> lchild);
                }

                if (p -> rchild)
                {
                        printf("%d", p -> rchild -> data);
                        enqueue( &q, p -> rchild);
                }
        }
}
```

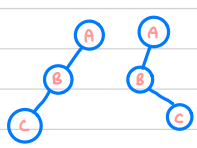
CAN WE GENERATE TREE FROM TRAVERSALS ?

$n = 3$

① Given preorder – A, B, C $\quad \frac{{}^{2n}C_n}{n+1}$



② Given preorder A, B, C
posorder C, B, A
more than one



CONCLUSION

If we are given either preorder, inorder, postorder we cannot generate unique tree.

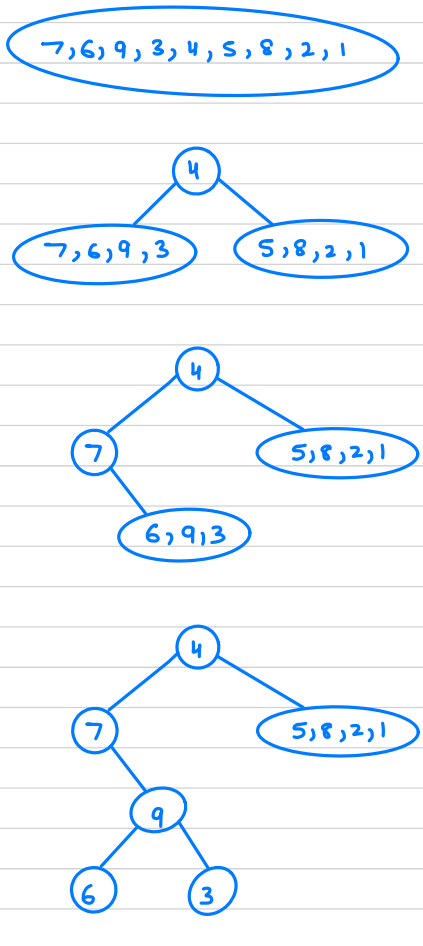
If we are given both preorder and postorder, then also unique tree cannot be generated

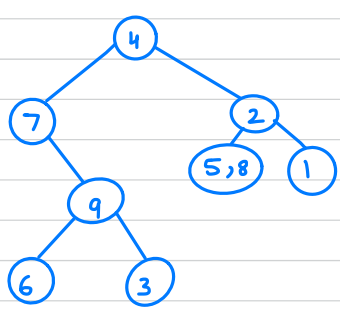
① preorder + inorder
② postorder + inorder

for generating unique tree

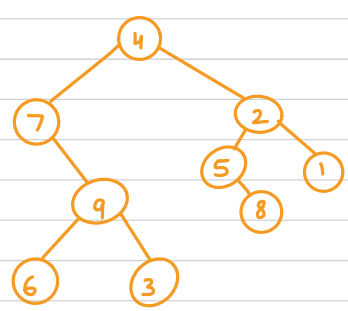
Q Preorder – 4, 7, 9, 6, 3, 2, 5, 8, 1
Inorder – 7, 6, 9, 3, 4, 5, 8, 2, 1

Sol



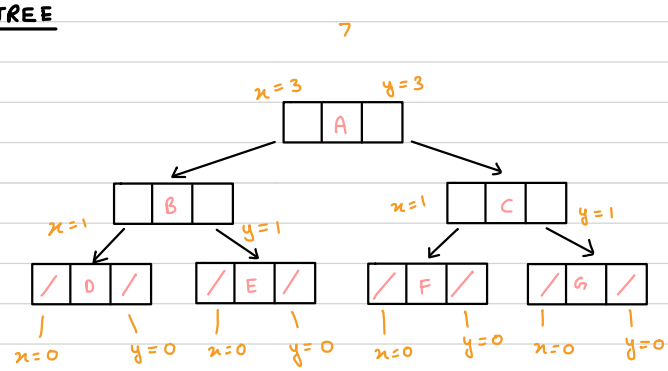


Unique Tree



COUNTING NO OF NODES IN A BINARY TREE

```
int  count (Node *p)
{
        int  x, y;

        if (p != NULL)
        {
            x = count (p -> lchild)
            y = count (p -> rchild)
            return x + y + 1;
        }
        return 0;
}
```



COUNTING LEAF NODES

COUNTING NODES WITH DEGREE 2

COUNTING NODES WITH DEGREE 2 OR 1

COUNTING NODES WITH DEGREE 1

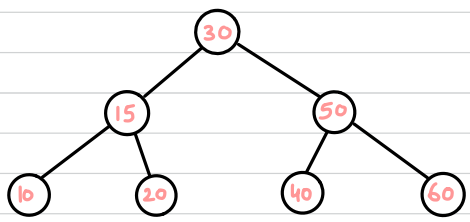
```
int  count (Node *p)
{
        int  x, y;

        if ( p != NULL)
        {
            x = count (p -> lchild);
            y = count (p -> rchild);

            if(p -> lchild == NULL && p -> rchild == NULL)
                return x + y + 1;
            else
                return x + y;
        }
        else
            return 0;
}
```

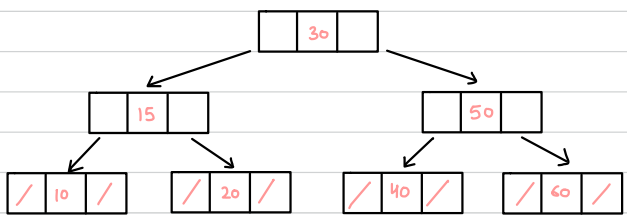
```
if(p -> lchild != NULL && p -> rchild != NULL)

if(p -> lchild != NULL || p -> rchild != NULL)

if ((p->lchild != NULL && p->rchild == NULL) ||
    (p->lchild == NULL && p->rchild != NULL))
```

XOR

```
if ( p -> lchild != NULL ^ p->rchild != NULL)
```

BINARY SEARCH TREE





- Left subtree is smaller
- Right subtree is greater
- Useful for searching (in less number of comparisons)
- No duplicates
- Inorder gives sorted order
- For n number of nodes, catalan number of trees can be generated

$$T(n) = \frac{{}^{2n}C_n}{n+1}$$

- BST is represented using linked representation.
- Can also be represented using arrays.

SEARCHING IN A BINARY SEARCH TREE

(searching takes maximum time same as height of tree) $O(\log n)$

(1) RECURSIVE SEARCH (tail recursion)

$O(h)$

$\log n \le h \le n$

```
Node * Rsearch(Node *t, int key)
{
    if (t == NULL)              // If element is not
        return NULL;            found
    if (key == t -> data)
        return t;
    else if (key < t -> data)
        return Rsearch(t -> lchild, key);
    else
        return Rsearch(t -> rchild, key);
}
```

(2) ITERATIVE VERSION $O(\log n)$

```
Node * Search(Node *t, int key)
{
    while (t != NULL)
    {
        if (key == t -> data)
            return t;
        else if (key < t -> data)
            t = t -> lchild;
        else
            t = t -> rchild;
    }
    return NULL;
}
```

INSERTING IN BINARY SEARCH TREE

```
void Insert(Node *t, int key)
{
    Node *r = NULL, *p;

    while (t != NULL)
    {
        if (key == t->data)
            return;        // To avoid duplication
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    p = new Node;
    p->data = key;
    p->lchild = p->rchild = NULL;

    if (p->data < r->data)
        r->lchild = p;
    else
        r->rchild = p;
}
```

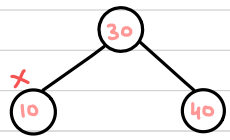
RECURSIVE INSERT IN BST

```
Node* Insert(Node *p, int key)
{
    if (p == NULL)
    {
        t = new Node;
        t->data = key;
        t->lchild = t->rchild = NULL;
    }
    if (key < p->data)
        p->lchild = insert(p->lchild, key);
    else if (key > p->data)
        p->rchild = insert(p->rchild, key);
    return p;
}
```

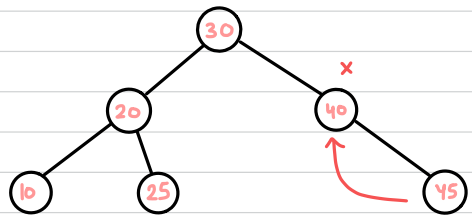
```
void main()
{
    Node *root = NULL;
    root = insert(root, 30);
    insert(root, 20);
    insert(root, 25);
}
```

DELETING FROM BST

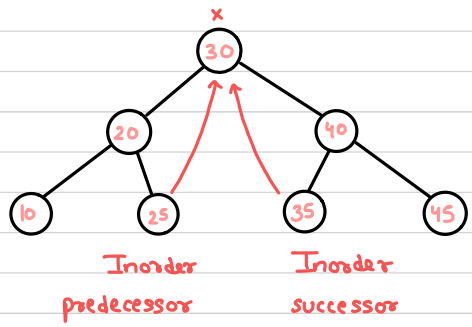
CASE 1 :



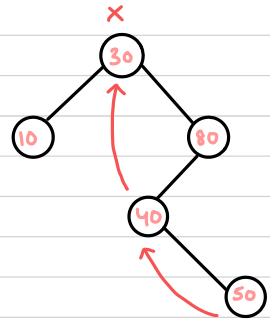
CASE 2 :



CASE 3 :



CASE 4 :



// Multiple changes to be made

```
int Height (struct Node *p)
{
        int x, y;
        if (p == NULL)
                return 0;
        x = Height (p->lchild);
        y = Height (p->rchild);
        return x>y? x+1 : y+1;
}
```

```
Struct Node * InPre (struct Node *p)
{
        while (p && p->rchild != NULL)
                p = p->rchild;
        return p;
}
```

```
struct Node *InSucc (Struct Node *p)
{
        while (p && p->lchild != NULL)
                p = p->lchild;
        return p;
}
```