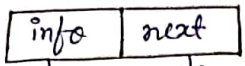
18/7/19

Linked List :

Linked list is an order collection of finite homogeneous data element called nodes where the linear order is maintained by means of link or pointer.

A linked list node has two parts.



What data/ info want to store.

Address of next node.

Single linked list

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

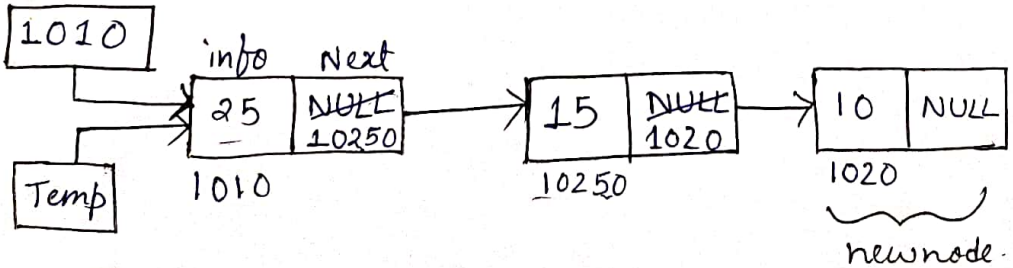
Linked list are three types—

1) Single linked list
2) Double linked list
3) Circular linked list.

In single linked list :—

Head

NULL → Store address of 1st node



Temp = info.

getnode() → memory address.

Procedure

I/P → he
nu
O/P → Poi
var
Begin,
if he
pri
retu
else
en
retu

Alg

Sing
I/P –
O/P –

Procedure_create_single(head/num) :-

I/p → head - Pointer to the first node
num - number of element to be inserted.

o/p → Pointer to the head node stored in the head variable.

```
Begin,
  if head ≠ NULL, then
    print "Linked list already created"
  return head.
  else
    for i ← 1 to num do
    item ← getinfo( ) ⟹ Printf & scanf part
    newnode ← getnode( ) ⟹ Created.
    info[newnode] ← item
    next[newnode] ← NULL
      if head = NULL then
        head ← newnode.
      else
        next[temp] ← newnode
      end if
        temp ← newnode
    end for.
  end if.
  return (head)
```

Algorithm

Single linked list traverse (head)

I/p – head - Pointer to the first node

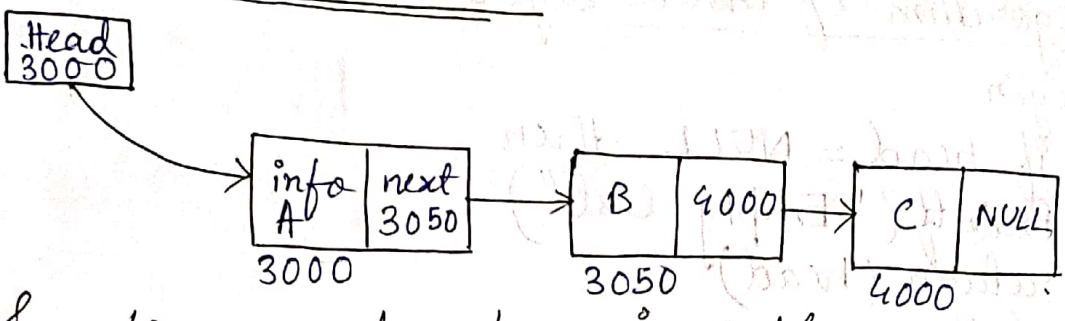
o/p – Print the content of the linked list.

```
Begin
    loc ← head
    while loc ≠ NULL do
    print (info [loc])
     loc ← next [loc]
  end while
end.
```

loc = Pointer variable

Linked list insertion

19/7/19



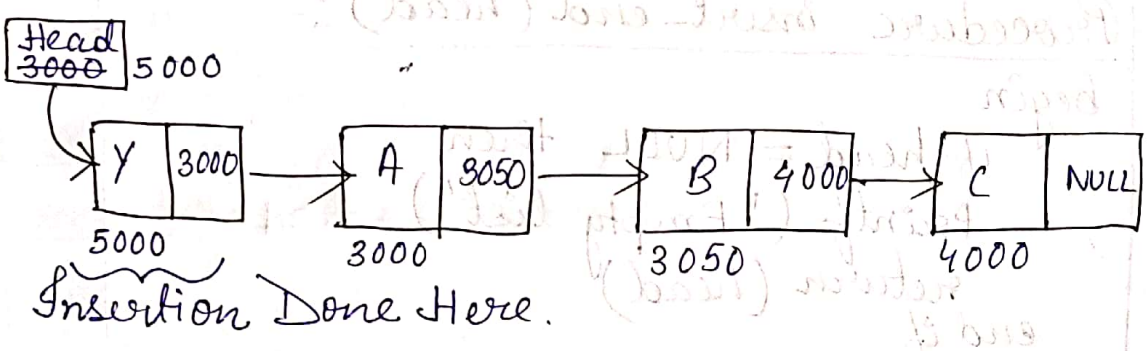
Insertion can be done in 3 places:-

1. 1ST Position
2. Last Position
3. Any Position.

Insertion at 1ST Position

next[newnode] ← Head

Head ← newnode.

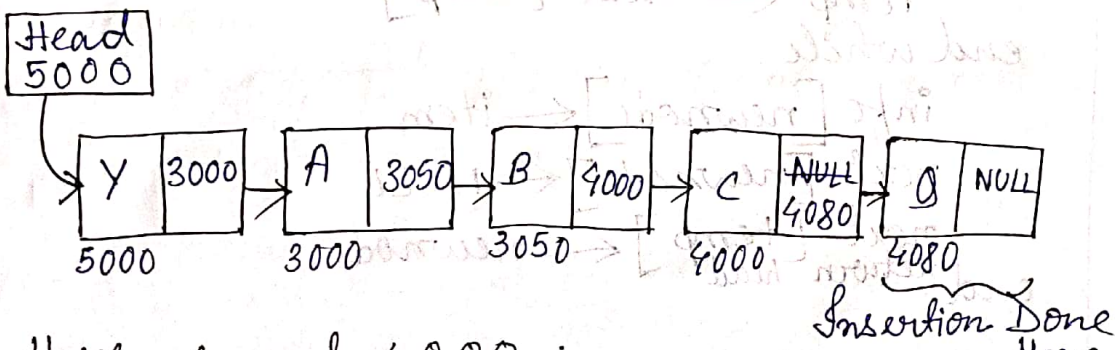


Insertion at last Position

next[newnode] ← NULL

next[temp] ← newnode

We check, temp → next == NULL.



Here, at temp 4000.

temp → next == NULL.

Algorithm of insert_begin (head):-

```
begin
  if head = NULL then
  Printf ("Empty list")
  return (head)
end if
item ← get info ( )
newnode ← getnode ( )
info [newnode] ← item
next [newnode] ← head
   head ← newnode
return (head)
end.
```

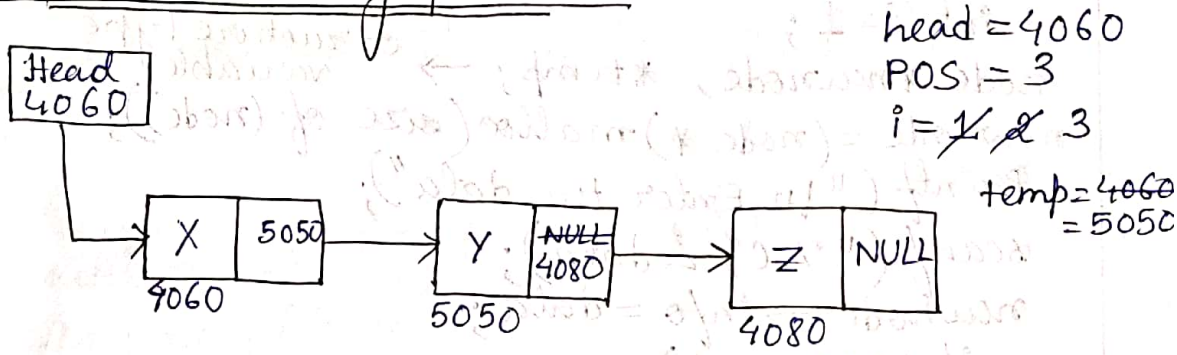
Procedure insert_end (head):-

```
begin
   if head = NULL then
    Printf ("Empty list")
     return (head)
  end if
   item ← getinfo( )
  newnode ← getnode ( )
    temp ← head
 while
      next [temp] ≠ NULL
      temp ← next [temp]
 end while.
      info [newnode] ← item
     next [newnode] ← NULL
     next [temp ] ← newnode
end return head
```

23/7/19

Typedef is a keyword which helps to rename the datatype.

Insertion at any position :-



info [newnode] = Z
next [newnode] = NULL
newnode = 4080

Algorithm

Procedure insert_any (head, POS) :-

```
begin
    set i ← 1.
    newnode ← getnode ( )
    item ← get info ( )
    info [newnode] ← item
    if POS = 1, then
        next [newnode] ← head
        head ← newnode.
    else
        temp ← head
        while i < POS - 1 and temp ≠ next.
        temp ← next [temp]
        i ← i+1.
        end while.
        next [newnode] ← next [temp]
        next [temp] ← newnode
    end if
    return head
    end.
```

Program

```
node * insert_any (node *head, int POS)
{  char data;
   int i=1;
 node *newnode, *temp;  → structure type variable
newnode =(node *)malloc(size of (node));
 Printf ("\n Enter the data");
 scanf ("%.c", &data);
 newnode → info = data;
   if (POS==1)
    { newnode → next = head;
      head = newnode;
    }
   else
    { temp = head;
      while (i< POS-1)
       { temp = temp → next;
         i++;
       }
      newnode → next = temp → next;
      temp → next = newnode;
    }
  return head;
}
```

Create linkedlist fun" :-

```
node * create (node * head, intn) → no. of node.
{ int i;
  node *temp; * newnode;
  for (i=0; i<n; i++)
   { newnode=(node *)malloc(size of (node));
     Printf ("Enter the data");
     scanf ("%.c", &newnode → info);
     newnode → next = NULL;
       if (head == NULL)
```

```
        head = newnode;
    else
        temp -> next = newnode;
        temp = newnode;
    }
return head;
}
```

25/7/19

Algorithm of Single linkedlist delete_any (head, POS)

```
begin
  set i ← 1.
  temp ← head.
  if POS = i, then
    head ← next (temp)
    return (head)
  end if.
  else
      while i < POS and temp ≠ NULL, do.
        Ptr ← temp
        temp ← next [temp]
        i ← i+1.
  end while
  next [Ptr] ← next [temp]
  call free (temp)
  return (head)
  end.
```

C function for traverse single linkedlist

```
Void traverse (node *head)
{ node *temp;
  temp = head;
  while (temp != NULL)
    printf ("%d", temp -> info);
    temp = temp -> next;
}
```

Algorithm of Traversing

```
begin
  temp ← head
  while temp ≠ NULL, do
  printf (info[temp])
  temp ← next[temp]
  end while
end.
```

26/7/19

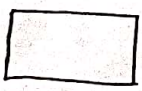
Algorithm of Single linked list reverse

```
begin
  if head = NULL, then
    print "empty list"
  end if
  loc ← head
  locn ← next[loc]
  locp ← NULL
  while locn ≠ NULL, do
        locp ← loc
        loc ← locn
        locn ← next[loc]
  end while
     head ← loc.
  return (head)
  end.
```

~~loc→next = locp~~ next[loc] ← NULL

~~loc→next = NULL~~

next[loc] ← locp.



Advantage of linked list :-

1) linked list are dynamic data structure that they can grow & shrink during the execution of a program

2) Efficient memory utilization. Here memory is not preallocated. It is allocated when it's required & it is deallocated or remove when it is no longer needed.

3) Insertion & deletion are easier & efficient linked list provide flexibility in inserting a data item at a special possition & deletion of a data item from a given possition.

Disadvantage :-

1) It will consume more memory.

2) Access to an arbitary data item is little bit difficult & also time consuming.

Singly Circular linked list :-

Circular Linked list is a special type of linked list. In this list the next field of the last node points to the 1st node of the list. This type of linked list is mainly used in list allow access to nodes in middle of the list without starting from the begining.

Algorithm of Circular linked list Create:

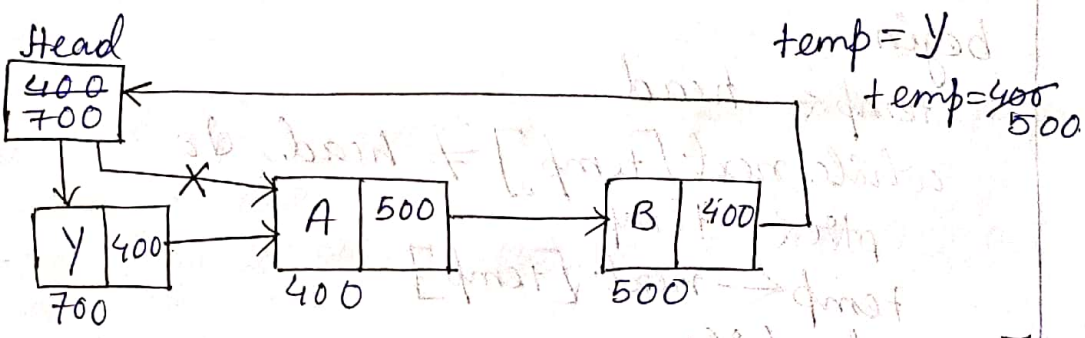
```
Procedure create_list (head, n)
begin
  if head ≠ NULL, then
  Print "Already Created"
  return (head)
  end.if
  for i ← 1 to n do
   item ← get info ( )
  newnode ← getnode ( )
   info [newnode] ← item
  if head = NULL, then
      temp ← head ← newnode
      next [newnode] ← head
  else
      next [temp ] ← newnode
      next [newnode] ← head.
  end if
       temp ← newnode.
  end for.
   return (head)
   end.
```

Procedure Traverse (head) :-

```
begin
   temp ← head
  while temp ≠ head do
   Print " info [temp]"
   temp ← next [temp]
 end while
end.
```

- Procedure insert_begin list(head) [circular]

```
begin
   item ← getinfo( )
 newnode ← getnode( )
 info[newnode] ← item
 temp ← head.
while next[temp] ≠ head, do
     temp ← next[temp]
end while.
     next[temp] ← newnode.
     next[newnode] ← head.
     head ← newnode
  return (head)
end.
```



- Procedure insert_end list(head) [circular] :-

```
begen
   item ← getinfo( )
 newnode ← getnode( )
  info[newnode] ← item
   temp ← head.
while next[temp] ≠ head, do
      temp ← next[temp]
end while
      next[temp] ← newnode
      next[newnode] ← head
 reteurn (head)
end.
```

- Procedure delete_begin_list (head) [circular]

```
begin
  temp ← head
  while next [temp] ≠ head, do
        temp ← next [temp]
  end while
     next[temp] ← next [head]
  call free (head)
   head ← next [temp]
   return (head)
end
```

- Procedure delete_end_list (head) [circular]

```
begin
  temp ← head
  while next [temp] ≠ head, do
     ptr ← temp
     temp ← next [temp]
  end while
      next [ptr] ← next [temp]
  call free (temp)
  return (head)
end.
```

30/7/19

Stack :- It is a linear data structure where insertion & deletion take place only at one end called top of the stack here insertion is called push operation & deletion is pop operation. Stack is also named as PUSH down list or LIFO [Last in fast out]

Algorithm insert_element in Stack using array.

```
begin
    if TOP >= STACKMAX
        print "stack is full"
    else
        TOP = TOP + 1.
        stack[TOP] = ITEM
    end if
end.
```

Deletion using array ( )

```
begin
    if TOP = NULL, then
        print "underflow"
    else
        Item = stack[TOP]
        TOP = TOP - 1
    end if
end.
```

Q. The following sequence of the operation is performed on a stack

PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP

The sequence of POP of values are 20, 20, 10, 10, 20.

- **Uses of Stack**

1) Evaluation of arithmatic eq^n's.
2) Incrementation of recursion. (calculation of factorial value, quick sort, tower of Honoi)

Infix → A+B, A-C+B
Prefix → +AB
Postfix → AB+

Algo: Evaluation of postfix expression.
This algorithm finds the value of arithmatic expression P written in postfix notation.

Start

STEP 1: Add a ")" at the end of P ["4)" This work as sentinel (last position)]

STEP 2: Scan P from L → R and repeat step 3 & 4 for each element of P until ")" is encountered.

STEP 3: If an operand is encountered PUSH it on STACK

STEP 4: If an operator (⊗) is encountered, then

(a) Remove the two TOP elements of STACK, where A is the TOP element & B is the next-to-TOP element.

(b) Evaluate B ⊗ A.

(c) PUSH the result of (b) on STACK

[End of If structure]

[End of step 2 loop]

STEP 5: Set VALUE equals to the TOP element of STACK

STEP 6: Exit

P: 5 6 2 + * 12 4 / −)

1 2 3 4 5 6 7 8 9 10

Scaned Symbol		STACK
1.	5	5
2.	6	5, 6
3.	2	5, 6, 2 → A, → B
4.	+	5, 8
5.	*	40
6.	12	40, 12
7.	4	40, 12, 4
8.	/	40, 3
9.	−	37
10.	)	

1/8/19

1) If TOP points at the TOP of the STACK & STACK is the array (STACK[ ]) containing STACK elements, then which of the following statement correctly reflect POP operation.

a) TOP = TOP - 1 ; item = Stack [TOP]

b) item = Stack[TOP] ; TOP = TOP - 1 ;

c) item = Stack [-- TOP]

d) Both (b) & (c) are correct.

2) If 2, 1, 5, 8 are the stack content with element 2 being at the top of the stack then what will be the

STACK USING LINKED LIST

2/8/19

Algorithm :- [PUSH( )]

```
Begin
   item ← getinfo( )
newnode ← getnode( )
 info[newnode] ← item
  if top = NULL, then
       next [newnode] ← NULL
          top ← newnode
   else
        next [newnode] ← top
        top ← newnode
   end if
 return (top)
end.
```

POP( )

```
Begin
    if top = NULL, then
       print "underflow"
    else
       temp ← top
       item ← info[top]
       top ← next [top]
       next [temp] ← NULL
       call free(temp)
   end if
 return (top)
end.
```

Traverse ( )

```
if (top == NULL)
 printf ("Underflow");
else
  {
    temp = top;
    while (temp != NULL)
     {
       printf ("%d", temp -> info);
       temp = temp -> next;
     }
  }
```

6/8/19

Postfix expression evaluation.

Q.1) P: 2 3 4 * + 8 −

Scaned symbol	Stack
1. 2	2
2. 3	2, 3 → B
3. 4	2, 3, 4 → A
4. *	2, 12
5. +	14
6. 8	14, 8
7. −	6
8.)	

Q.2) P: 9 3 4 * 8 + 4 /−

Scaned symbol	Stack
1. 9	9
2. 3	9, 3 → B
3. 4	9, 3, 4 → A
4. *	9, 12
5. 8	9, 12, 8
6. +	9, 20
7. 4	9, 20, 4
8. /	9, 5
9. − 10.)	4

Algorith

Step 1:

Step 2:

Step 3:

Step 4:

Step 5

Step 6

Step

Inf

P

A

A

Transforming Infix to Postfix

Algorithm: REVERSE POLISH (Q, P)
Q is the arithmatic expression written in infix notation. This algo. finds the equiva-lent postfix notation.

Step 1: PUSH "(" onto STACK and add ")" to the end of Q.

Step 2: Scan Q from L → R and repeat step 3 to 6 for each element of Q until the STACK is empty.

Step 3: If an operand is encountered add it to P.

Step 4: If a left parenthesis "(" is encountered PUSH it on STACK.

Step 5: If an operator ⊗ is encountered, then:
(a) Repeatedly pop from STACK and ADD to P each operator (on the TOP of STACK) which has the same precedence as or higher precedence than ⊗.
(b) ADD ⊗ to STACK.
End if.

Step 6: If a right parenthesis ")" is encountered
(a) Repeatedly POP from STACK and ADD it to P each operator (on the TOP of STACK) until a "(" is encountered.
(b) Remove left parenthesis "(" [DO not add it to STACK]
End if
End of step 2 loop.

Step 7: end.

Infix → A+B*C

Postfix	Prefix
A+[BC*]	A+[*BC]
ABC*+	+A*BC

Infix → (A+B)*C

Postfix	Prefix
[AB+]*C	[+AB]*C
AB+C*	*+ABC

Infix → A * (B+C) * D

Postfix → A * [BC+] * D

[ABC+*] * D

ABC+ * D *

Prefix → A * [+BC] * D

[*A+BC] * D.

** A+BCD.

L → R

Q: A+(B*C-(D/E↑F)*G)*H)

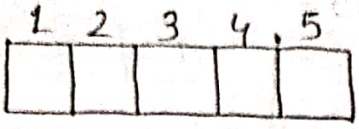
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Scaned Symbol	STACK	Expression P
1) A	(	A
2) +	(+	A
3) (	(+(	AB
4) B	(+(	AB
5) *	(+(*	ABC
6) C	(+(*	ABC*
7) -	(+(-	ABC*D
8) (	(+(-(	ABC*DE
9) D	(+(-(	ABC*DE↑
10) /	(+(-(/	ABC*DEF
11) E	(+(-(/↑	ABC*DEF↑/
12) ↑	(+(-(/↑	ABC*DEF↑/G
13) F	(+(-(/↑	ABC*DEF↑/G*-
14))	(+(-	ABC*DEF↑/G*-H
15) *	(+(-*	ABC*DEF↑/G*-H*
16) G	(+(-*	
17))	(+	
18) *	(+*	
19) H	(+*	
20))	EMPTY	

Q: (A+B) * C -(D-E) / (F+G))
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Scaned Symbol	STACK	Expression P
	(	
1) (	((	
2) A	((	A
3) +	((	A
4) B	((+	AB
5))	(	AB+
6) *	(*	AB+
7) C	(*	AB+C
8) -	(-	AB+C*
9) (	(-(	AB+C*
10) D	(-(	AB+C*D
11) -	(-(-	AB+C*D
12) E	(-(-	AB+C*DE
13))	(-	AB+C*DE-
14) /	(-/	AB+C*DE-
15) (	(-/(	AB+C*DE-
16) F	(-/(	AB+C*DE-F
17) +	(-/(+	AB+C*DE-F
18) G	(-/(+	AB+C*DE-FG
19))	(-/	AB+C*DE-FG+
20))	EMPTY	AB+C*DE-FG+/-

Queue



FORNT = Delete
REAR = Insert
If Forent & Rear are equal then only one value is present.
At 1st FORNT & REAR = NULL
When we want to delete one & then it will take place from FORNT & if we want to insert element REAR value will be incremented by 1.

8/8/19

- **Algorithm insersion**

Step 1: if REAR = & MAX
Step 2: Print "Overflow"
Step 3: Exit
Step 4: else
Step 5: If (REAR = NULL) and (FRONT = NULL), then
Step 6: FRONT = REAR = 1.
Step 7: else
Step 8: REAR = REAR + 1
Step 9: end if
Step 10: & [REAR] = Item
Step 11: End if
Step 12: End.

- **Algorithm of Deletion.**

Step 1: If (FRONT = NULL), then
Step 2: Print "underflow"
Step 3: exit
Step 4: else
Step 5: Item ← & [FRONT]
Step 6: If (FRONT = REAR)
Step 7: REAR ← 0, FRONT ← 0
Step 8: else
Step 9: FRONT ← FRONT + 1.
Step 10: end if.
Step 11: end if
Step 12: end

What is Queue?

Queue is a linear data structure where insersion take place at one end called REAR & deletion take place in another end called FRONT. Here which element come first that element dileted first also i.e., Queue is called FAST IN FAST OUT or FIFO.

Application > Application of queue infront of an counter.

2) Traffic signal

3) Printer.

4) Resource sharing in computer centre.

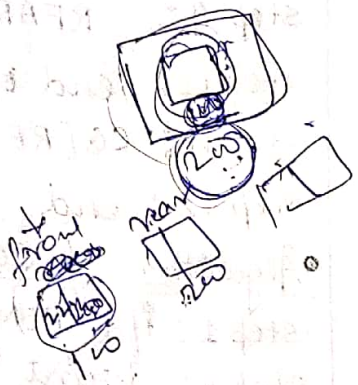
```
Struct node
{ int info;
  Struct node *next;
} rear, front;
```

If in a queue there are more than one element then always REAR > FRONT.

8) QMAX = 8

1	2	3	4	5	6	7	8
4	5	-9	66				

FRONT=? REAR=?



① insert 16
② delete
③ delete
④ insert 7
⑤ delete
⑥ insert -2.

→ ~~FRONT~~ FRONT=1, REAR=4

① F=1, R=4+1=5

Q[R]=16

1	2	3	4	5	6	7	8
4	5	-9	66	16			

② F=~~1~~ 2 1+1=2

1	2	3	4	5	6	7	8
	5	-9	66	16			

③

1	2	3	4	5	6	7	8
		-9	66	16			

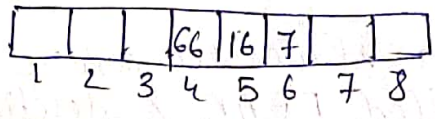
F=3, R=5

④

1	2	3	4	5	6	7	8
		-9	66	16	7		

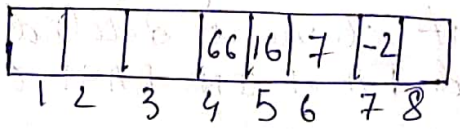
F=3, R=6 Q[R]=7

⑤



F = 4 R = 6.

⑥



F = 4 , R = 7.

3/8/19

Circular Queue:

• Algorithm of CQ Insert

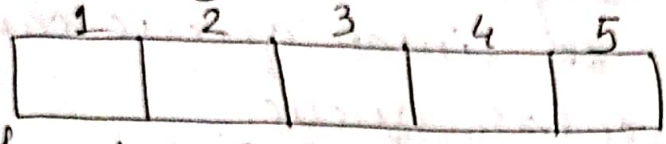
Step 1: If (FRONT = 1 and REAR = QMAX) OR (FRONT = REAR + 1) then
Step 2: Print "overflow"
Step 3: else
Step 4: If (FRONT = NULL), then
Step 5: FRONT ← 1, REAR ← 1
Step 6: else if (REAR = QMAX), then
Step 7: REAR ← 1.
Step 8: else
Step 9: REAR ← REAR + 1
Step 10: end of if structure
Step 11: CQ[REAR] ← Item
Step 12: end

• Algorithm of CQ Delete

Step 1: if (FRONT = NULL), then
Step 2: Print "Underflow"
Step 3: else
Step 4: Item ← CQ[FRONT]
Step 5: If (FRONT = REAR), then
Step 6: FRONT ← NULL, REAR ← NULL.
Step 7: else if (FRONT = QMAX), then
Step 8: FRONT ← 1.
Step 9: else
Step 10: FRONT ← FRONT + 1.

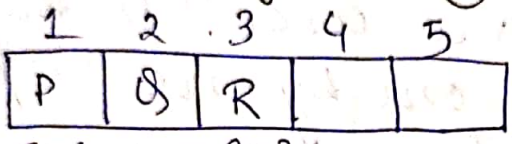
Step 11: End of if structure.
Step 12: End

Q) QMAX = 5



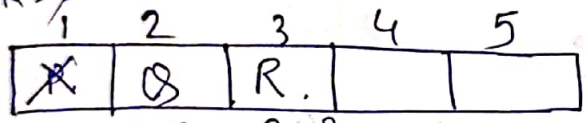
① Insert P, Q, R
② delete 1 element
③ Insert w, x y, z.
④ delete two element
⑤ insert G, H.
⑥ Delete two element

→ ①



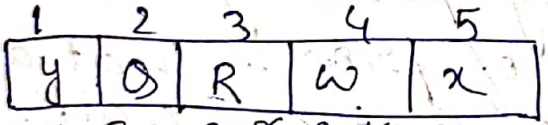
F=1 ~~R=2~~ R=3
~~R=1~~

②



F=2 R=3

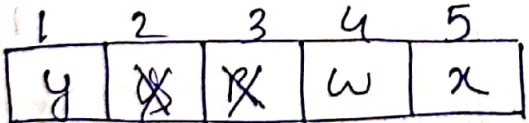
③



R=1 F=2 ~~R=3~~ ~~R=4~~ ~~R=5~~

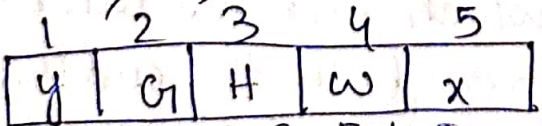
z will be show overflow.

④



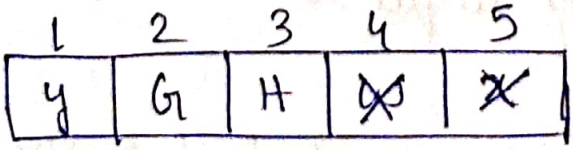
R=1 ~~F=2~~ ~~F=3~~ F=4

⑤



~~R=1~~ ~~R=2~~ R=3 F=4

⑥



F=1 R=3

Application of circular Queue:

① Memory mangement :-

The unused memory location in the case of ordinary queue or linear queue can be utilize in circular queue.

② Traffic system :-

In the computer control traffic system circular queue are used to switch on the traffic lights one by one repetedly as per the time set

③ CPU scheduling :-

CPU scheduling operating system often maintain a queue of process that are ~~maintain~~ ready to execute or that are waiting for a event occur.

13/8/19

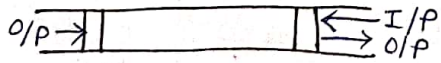
Double ended queue

A double ended queue or deque is the collection of homogeneous queue in which elements can be added & deleted or removed from both the end (FRONT end/ REAR end)

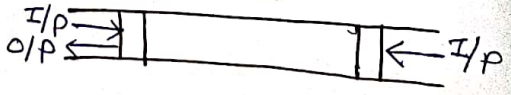
There are two type of deque —

1. Input restricted
2. Output restricted.

Input restricted



Output restricted.



Application of Deque :-

1) The undo & redo operation.

2) A steal-Job scheduling algo.

V.V.I

Left_Insert(DQ, QMAX, data)

Step 1: If (left = 1 and right = QMAX) or (left = right + 1), then
Step 2: print "overflow DQ".
Step 3: exit.
Step 4: else
Step 5: if left = NULL, then
Step 6: set, left ← right ← QMAX
Step 7: else if left = 1, then
Step 8: set left ← QMAX
Step 9: else
Step 10: set, left ← left − 1
Step 11: end if
Step 12: DQ[left] ← data
Step 13: End if
Step 14: End.

Right_Insert(DQ, QMAX, data)

Step 1: if (left = 1 and right = QMAX) or (left = right + 1), then
Step 2: print "overflow"
Step 3: exit.
Step 4: else
Step 5: if right = NULL, then
Step 6: set, left ← right ← 1.
Step 7: else if right = QMAX, then
Step 8: set, right ← 1.
Step 9: else
Step 10: set, right ← right + 1.
Step 11: end if
Step 12: set, DQ[right] ← data.
Step 13: end if
Step 14: end.

left delete(DQ, QMAX)

Step 1 : If (left = NULL), then
Step 2 : print "overflow" and exit
Step 3 : else
Step 4 : set data ← DQ[left]
Step 5 : if left = right
Step 6 : set, left ← right ← NULL
Step 7 : else if left = QMAX, then
Step 8 : set, left ← 1
Step 9 : else
Step 10 : set, left ← left + 1
Step 11 : end if
Step 12 : end if
Step 13 : end.

Right delete(DQ, QMAX)

Step 1 : If (right = NULL), then
Step 2 : print "underflow" and exit
Step 3 : else
Step 4 : set, data ← DQ[right]
Step 5 : if right = left, then
Step 6 : set, right ← left ← NULL
Step 7 : else if right = 1, then
Step 8 : set, right = QMAX
Step 9 : else
Step 10 : set right ← right − 1
Step 11 : end if
Step 12 : end if
Step 13 : end

Q. Consider the following deque of character where deque as a CQ which has allocated 6 memory space.
(a) F is added to the right
(b) Two letter deleted from right
(c) K, L, M are added to the left.
(d) One letter deleted from left
(e) R added to the left
(f) S added to the left
(g) T added to the right → left = right + 1
overflow.

left = 2, right = 4.
QMAX = 6.
Deque : —, A, C, D, —, —

R=3 L=4

K	A	C	S	R	L

Q. Deque QMAX = 6
left = 3, right = 5
Deque : —, —, R, T, S, —

1. insert X at left end. L=2, R=4
2. insert Y at right end
3. Delete two from right.
4. Insert G, Q, M at left.
5. Insert J at right → left = Right + 1, overflow
6. Delete two from left.
L=1, R=4

L=5 ~~L=6~~
R=5 L=6
L=X ~~L=2~~ ~~L=3~~ R=4 R=5 ~~R=6~~

G	X	R	T	S/M	Y/S

Priority Queue :-

It is another variation of queue structure here each element has been assigned a value for the priority of the element and an element can be inserted or deleted not only at the end but at any position on the queue.

FRONT ↓ REAR ↓

	P_1	P_2	P_3	P_i	P_4	
	A	B	X	y	D	

With this structure an element "y" of priority P_i may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority i.e., instead adding an element at REAR end it may be inserted at an intermediate position dedicated by its priority value.

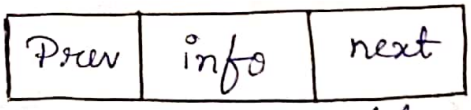
- A priority queue doesn't strictly follow FIFO principle.

1. An element of higher priority is processed before an element of lower priority.
2. Two elements with the same priority are process according to the order in which they were added to the queue.

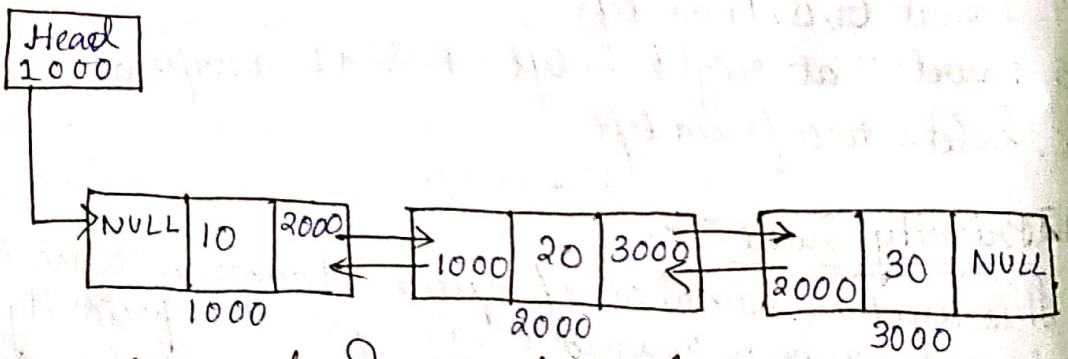
16/8/19

Doubly linked list:-

Doubly linked list is called two way data structure. It has three part prev, info, next. It is linear data structure.



Prev contains the address of previous node.
Info contains the information to be stored in the node.
Next contains the address of the next node.



Structure of Doubly linked list:-

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

- Operations can be performed —

1. Create.
2. Insertion
3. Deletion
4. Traverse [Forward & backward]
5. Reverse

Advantage of Doubly linked list :-

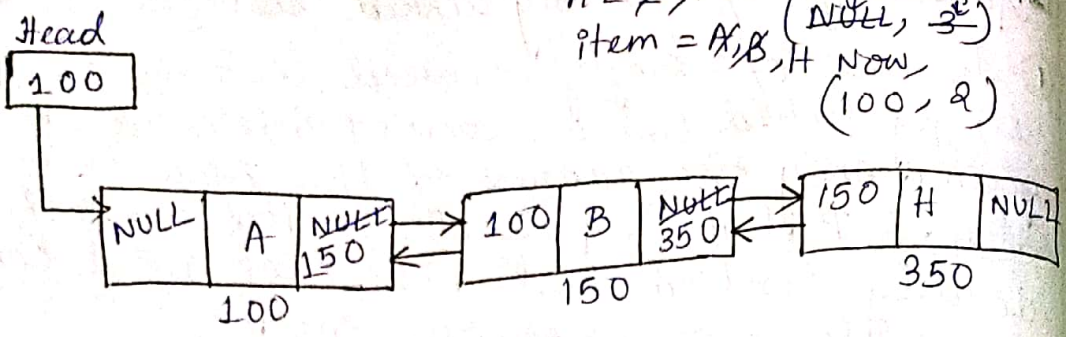
① In case of singly linked list or circular singly linked list we cannot delete a node giving only the address of the node to be deleted. But it is possible in the case of doubly linked list bcz each node contains the address of previous node.

② We can traverse the list backward as well as forward fo doubly linked list but which is not posseble for singly & circular singly linked list.

- Algorithm for create - double linked (head, n).

```
Begin
  if head ≠ NULL, then
    print "Already Created"
  return (head).
  end if
 for i ← 1 to n do
   item ← get info( )
  newnode ← get node( )
   info[newnode] ← item.
   next[newnode] ← NULL.
  if head = NULL, then
    head ← newnode.
    Prev[newnode] ← NULL
  else
    next[temp] ← newnode
    Prev[newnod] ← temp
  end if
  temp ← newnode
end for
  return (head)
end.
```

n = ~~1~~ ~~2~~ 3 (head, n)
item = A, B, H (NULL, ~~2~~ 3)
Now (100, 2)



newnode = ~~100~~ . ~~150~~ 350.

temp = ~~100~~ ~~150~~ 350.

20/8/19

Procedure insert_anyposition_dlist(head, POS)

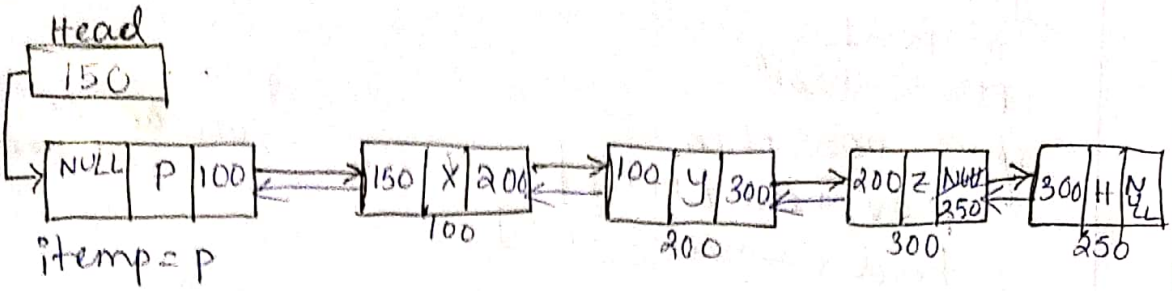
```
Begin
  set i ← 1
  newnode ← getnode()
    item ← getinfo()
  info[newnode] ← item
    if i = POS, then
        Prev[newnode] ← null
        next[newnode] ← head
        Prev[head] ← newnode
            head ← newnode
        return (head)
    end if
  temp ← head
  while i < POS AND temp ≠ NULL
        loc ← temp
        temp ← next[temp]
        i ← i+1
  end while
  if (next[loc] = NULL), then
        next[loc] ← newnode
        next[newnode] ← NULL
        Prev[newnode] ← loc
  else
        next[loc] ← newnode
        Prev[newnode] ← loc
        next[newnode] ← temp
        Prev[temp] ← newnode.
```

→ for 1st position.

→ for last POS

→ for any POS

```
    end if
    return (head)
end.
```



Traverse dlist (head) [forward]

```
Begin
  if (head == NULL), then
     print "list is empty"
  else
     temp ← head
     while temp ≠ NULL do
        print (info [temp])
        temp ← next [temp]
     end while
  end if
end.
```

Procedure traverse_dlist (head) [Reverse]

```
Begin
   temp ← head
   while temp ≠ NULL, do
      q ← temp
      Print " info [temp] "
      temp ← next [temp]
   end while
   while q ≠ NULL, do
      print " info [q] "
      q ← prev [q]
   end while
end.
```

→ forward

→ for backward we just have to remove this print.

→ backward.

Procedure delete_any position_dlist (head, POS)

```
Begin
  set i ← 1
  ptr ← head
  if i = POS, then.
     prev [next [ptr]] ← NULL  → use double pointer
     head ← next [ptr]
  end if.
  while i < POS AND ptr ≠ NULL
       temp ← ptr.
       ptr ← next [ptr]
       i ← i+1.
  end while
  if (next [ptr] = NULL) ; then
       next [temp] ← NULL

  else
       next [temp] ← next [ptr]
       prev [next [ptr]] ← temp  ⟷ use double pointer.

  end if
  call free (ptr)
  return (head)
end.
```

22/8/19

Polynomial Representation:

Linked list are widely used to represent & manipulate polynomials. Polynomials are the expression containing no. of term with non-zero coefficient & exponent. General syntax of polynomial

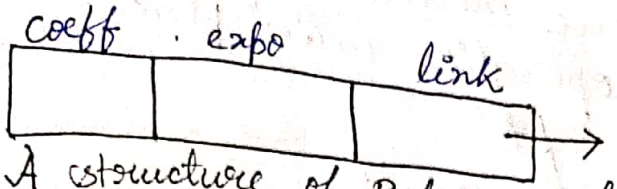
$$P(X) = a_n x_n^e + a_{n-1} x_{n-1}^e + \ldots + a_1 x_i^e + a.$$

where, a_i are non-zero coefficient

e_i " exponent.

In the linked list representation of polynomial each node contain their fields

- co-efficient field
- exponent field
- link field.

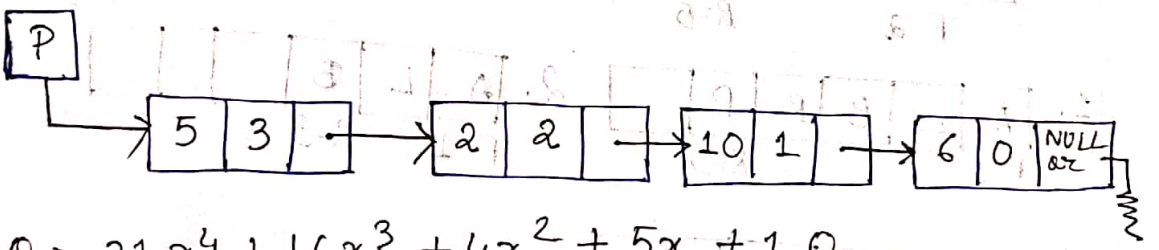


A structure of Polynomial node

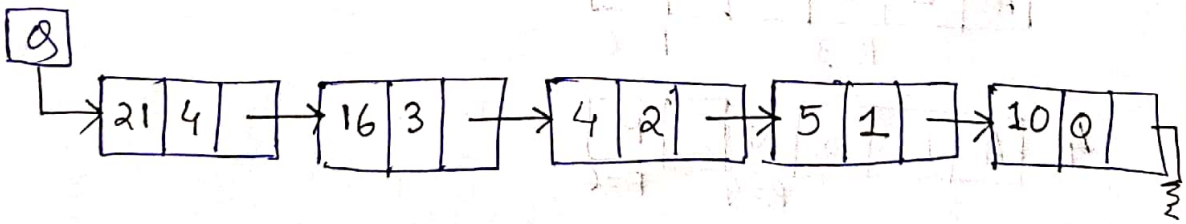
- Logical Representation of a node →

```
struct polynode
{
    int coeff;
    int expo;
    struct polynode *ptr;
};
typedef struct polynode PNODE;
```

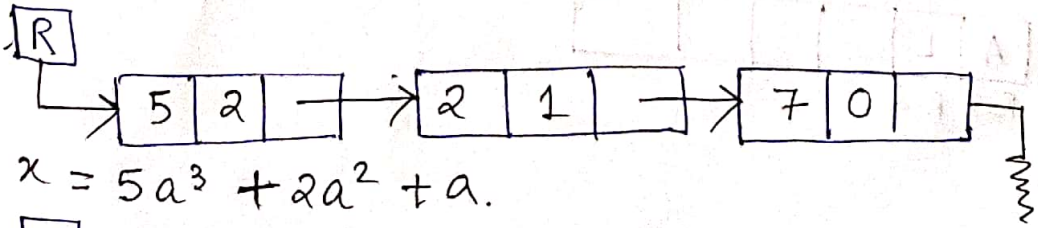
$P = 5x^3 + 2x^2 + 10x + 6$



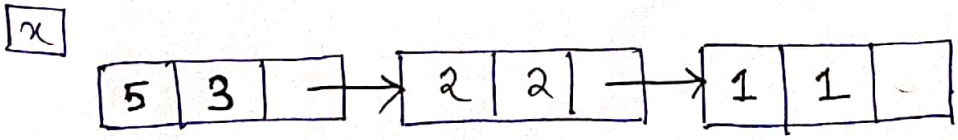
$Q = 21x^4 + 16x^3 + 4x^2 + 5x + 10$



$R = 5x^2 + 2x + 7$



$x = 5a^3 + 2a^2 + a.$



Q. Consider the following Queue which has six memory cell. Front = 2, Rear = 5.

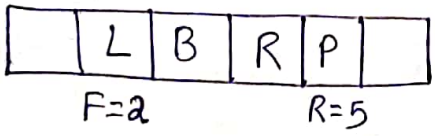
Q = —, L, B, R, P, —

Describe the queue including front & rear as the following operation takes place.

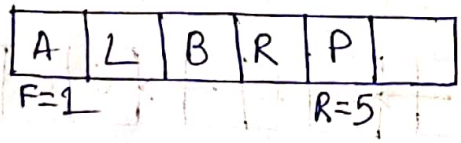
1) A is added to left.
2) Delete 2 info from right
3) M added to left.
4) N added to right
5) Delete 2 info from right.
6) Delete 1 info from left.

QMAX = 6

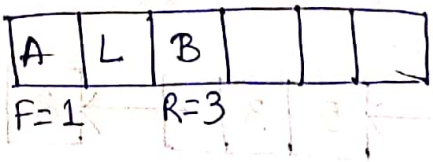
F = 2, R = 5



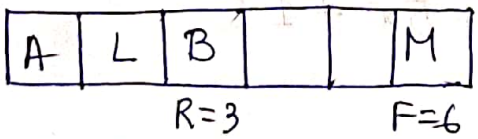
1.



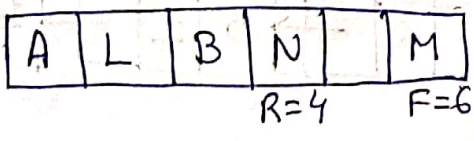
2.



3.

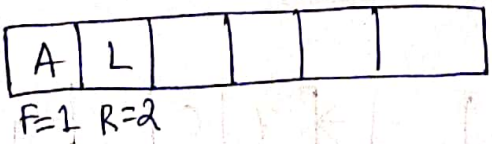


4.



5.

6.



Time Complexity:

The time Complexity is define as the process determine and formula for total time require to algorithm. This calculate will be independent details and programing language.

Assymetric notations:

1) They are used to make meaning full statement absolute the efficient of algorithm. commonly used Assymetric notations are —

i) O → Big-on / upper / max.

ii) Ω → Big-omega (lower bound / avg)

iii) Θ → Big-theta (tight bound / avg)

Big-Oh rotation:

The function $f(n) = O(g(n))$ if

Time Complexity Theorem :-

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$,
then $f(n) = O(n^m)$

Proof → let, $c = a_m + a_{m-1} + \cdots + a_1 + a_0$

$$= \sum_{i=0}^{m} a_i$$

then, $a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0 \leq n^m \times \sum_{i=0}^{m} a_i$ for all $n \geq 1$.

By Definition,

$$0 \leq f(n) \leq c \times g(n)$$

$c = \sum_{i=0}^{m} a_i$, $g(n) = n^4$ and $n_0 = 1$.

So, $f(n) = O(n^m)$

Predefined func^n :- like scanf, printf need constant time to run

Example :-

1.
```
main()
{
  for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    x = y+3;
}
```
→ $O(n^2)$

2.
```
main()
{
  a = a+1;
  for(j=0; j<n; j++)
  for(i=0; i<n; i++)
    x = y+3;
}
```
→ $O(n^2)$

[It can be $O(n^2+1)$ but here 1 is very small so it will be discarted]

3.
```
main()
{
  while(n>1)
    n = n-1;
}
```
→ $O(n)$

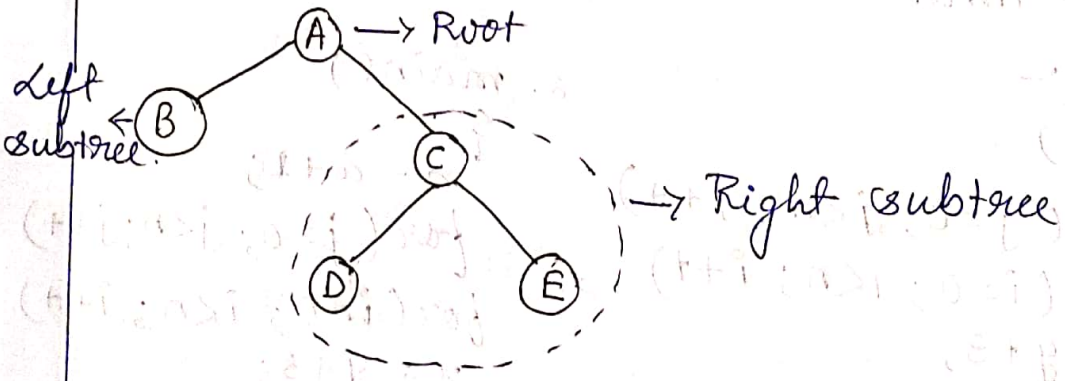
```
main( )
{
  while (n>=1)
  Printf ("%d", n);
} n = n-2;   [Ans, n=8]
```

→ $O(n/2)$

[Coz here n=8 & n=n-2 so n will be decremented by 2 so it will print 4 times so the time complexity becomes half]

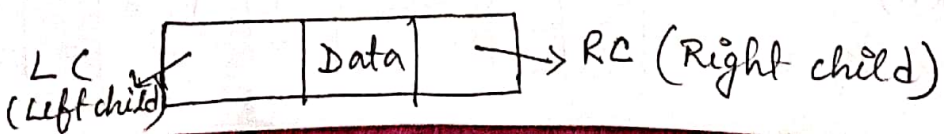
Tree:

A Tree is a finite set of vertices connected by edge such that i) there is one special designated node or vertices or vertex/node is called Root. ii) The remaining vertices as partition into a collection of sub-tree. ($I_1, I_2, \ldots I_n$) each of which is also a tree.



Node of Tree:

This is the main component of any tree structure. The concept of node is same as that use in the linked list. A node of a tree store actual data & linkes to the other node.



Parent →
The parent of a node is the immidiate predecessor of a node.

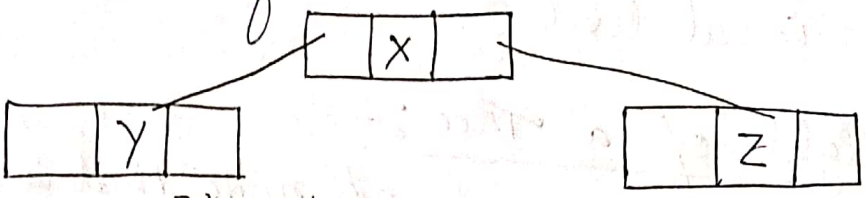


Fig : 1.

Here X is the parent of Y & Z

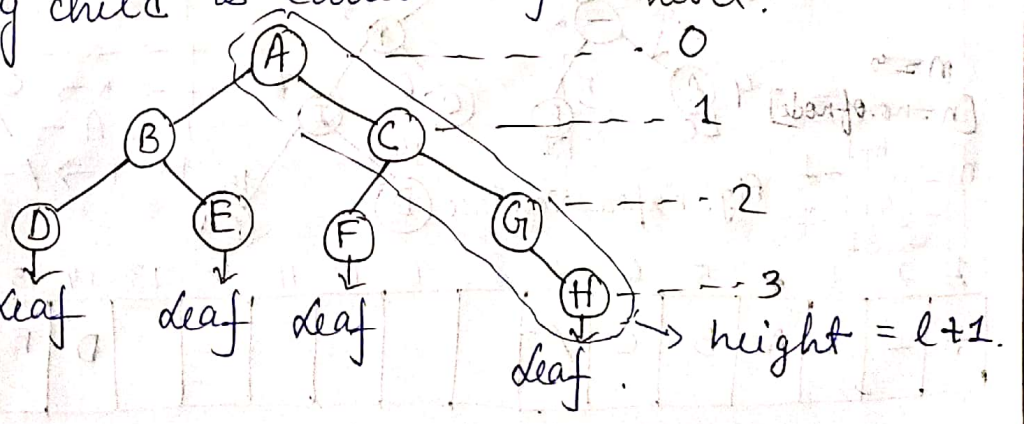
Child :–
If the immidiate predecessor of a node is the parent of the node then all the immidiate successor of node known as child node.

From the Fig: 1 Y & Z are the two child of X. The child which is in left side is called left child (LC) & which in the right side is called Right child (RC).

- What is LINK?
This is the pointer to a node in a tree

- **Root :→**
This is the special designated node which has no Parent is called Root.

- **Leaf :→**
The node which is at the end does not have any child is called leaf.



- **Level of tree :→** Level is the rank of hierarchy. The root is at level 0.

- **Height of a Tree :→** The maximum no. of node that is possible in a path starting from the root node to a leaf node is called height of a tree.

  $h = l + 1$ [l = level]

- **Degree of a tree :→** Every node can have maximum no. of child.

[For any binary tree degree is 2.]

12/9/19

Array representation of a Binary Tree.

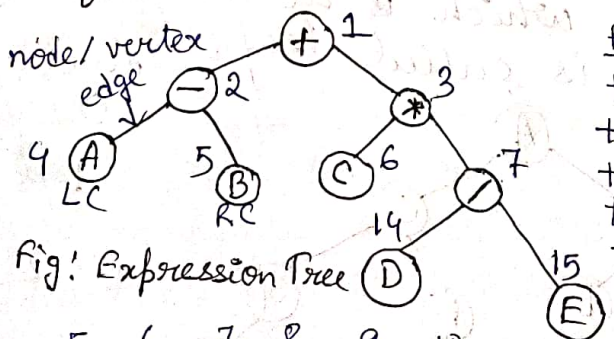
Rule 1: The root node is at location 1.

2: For any node with index i, $1 < i \leq n$

(a) PARENT $(i) = [i/2]$, for the node when $i = 1$, there is no Parent.

(b) LCHILD $(i) = 2 \times i$
if $2 \times i > n$, then i has no left child.

(c) RCHILD $(i) = 2 \times i + 1$
if $2 \times i + 1 > n$, there has no right child



Path
+ → - → A
+ → - → B
+ → * → C
+ → A → / → D
+ → * → / → E

Fig: Expression Tree

$n = 2^{h+1} - 1$
$= 2^{4+1} - 1$
$= 31$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
+	-	*	A	B	C	/	.	.	.	.	.	.	D	E	.	.	.

• Binary Tree:

A binary tree is finite set of node or data items which is either empty or consists of a single data item called root & two disjoint binary tree called leftsubtree & rightsubtree In binary tree the max^m degree of any node is 2.

Max size of array for any tree $2^{h+1} - 1$.

V.I

Traverse of a Binary Tree

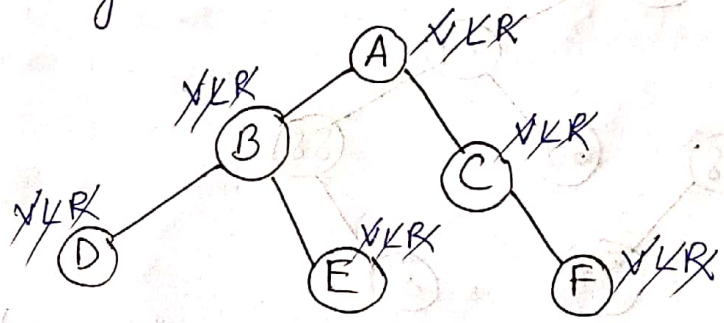
Three types —

① Pre-order Traverse (VLR)
② Post-order Traverse (LRV)
③ In-order Traverse (LVR)

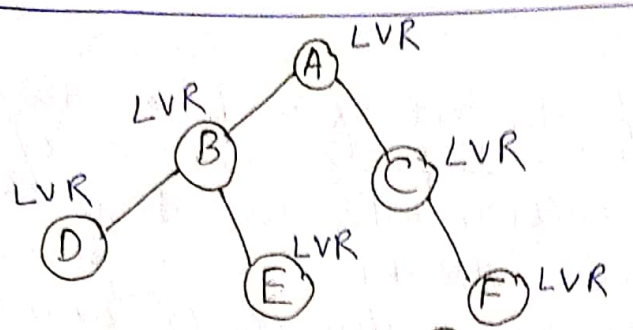
① Pre-order Traverse →
VLR means visit the node, Go to left, Go to right

② Post-order Traverse →
LRV means Go to left, Go to right, visit the node

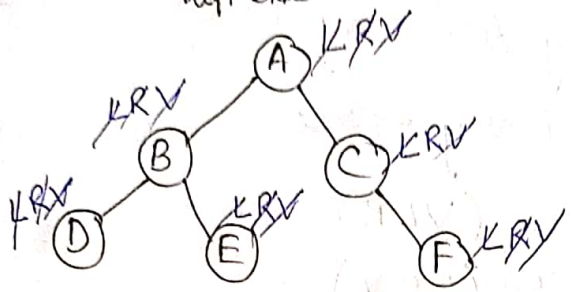
③ In-order Traverse →
LVR means Go to left, visit the node, Go to right.



Preorder → A B D E C F
↓
Root node (First position)



Inorder → D B E A C F → Right side node
Left side node, Root node (any position)

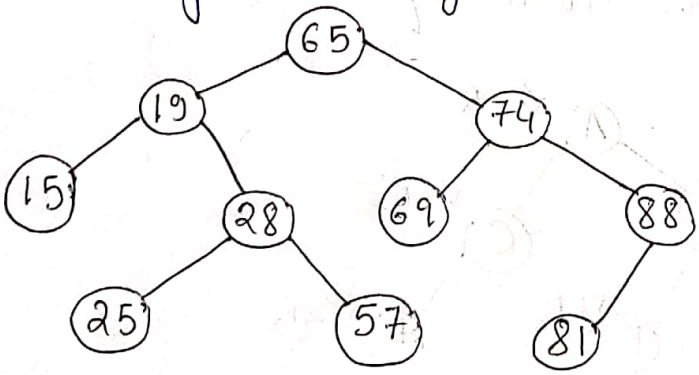


Post-order → D E B F C A
↘ Root node (last)

Binary Search Tree:

A binary tree T is termed binary search tree (or binary sorted tree) [BST]. If each node N of T satisfies the following property.

The value of N is greater than every value in the left sub-tree of N and is less than every in the right subtree of N



[A binary search tree with numeric data]

- Possible operations on any BST:—
  1) Searching data
  2) inserting data
  3) Deleting data
  4) Traversing the tree.

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the location of ITEM in the BST, or insert ITEM as a newnode in its appropriate place in the tree.

(a) Compare ITEM with the root node IV of the tree.

i) If ITEM < N, proceed to the left child of N.

ii) If ITEM > N, proceed to the right child of N.

(b) Repeat step (a) until one of the following occurs:

i) We meet a node N such that ITEM = N. In this case the search is successful.

ii) We meet an empty subtree, which indicates that the search is unsuccessfuly & we insert ITEM in place of the empty subtree.

Following six numbers are inserted in order into an empty BST : 40, 60, 50, 33, 55, 11.

☑ Non recursive function for traversal:

Pre order Algorithm :>

Initially ptr contains the address of root.

1. PUSH the address of root node on the stack.
2. POP an address from stack.
3. If the popped address is not NULL.
   (a) Traverse the node (Display the node)
   (b) PUSH right child of node on stack.
   (c) PUSH left child of node on stack.
4. Repeat step 2, 3 until the stack is no empty.

C-function:-

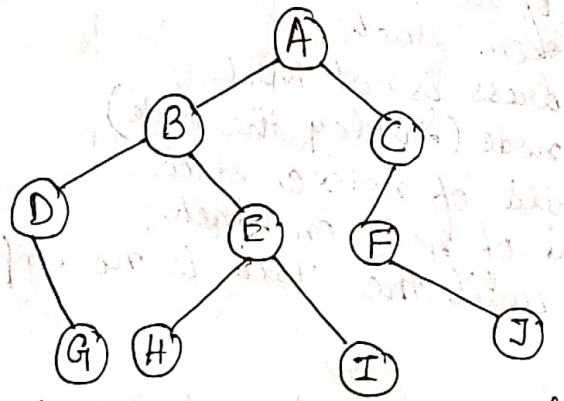
```
non-rec-preorder (struct node *ptr)
{
  stack[++top] = ptr;
  while(top != -1)
  {
    ptr = stack[top--];
    if(ptr != NULL)
```

```
        printf("%d", ptr -> info);
        stack[++top] = ptr -> rchild;
        stack[++top] = ptr -> lchild;
    }
  }
}
```

An array STACK is used to hold the addresses of nod
TOP pointer points to the top most element of the stack.
ROOT is the root node of tree to be traversed - PN
is the address of the present node. Info(PN) if the
information contained in the node PN.

1. Initialize TOP = NULL, PN = ROOT
2. Repeat step 3 to 5 until (PN = NULL)
3. Display Info (PN)
4. If (Right (PN) not equal to NULL)
   (a) TOP = TOP+1
   (b) STACK [TOP] = Right (PN);
5. If (Left (PN) not equal to NULL)
   (a) PN = Left (PN)
   else
   (a) PN = STACK [TOP]
   (b) TOP = TOP - 1.
6. Exit.

Preorder



In-order non-recursive traversal:-

Initially ptr contains the address of root:-

1. Repeat step 2,3 while stack is not empty or ptr is not equal to NULL
2. If ptr is not equal to NULL
   (a) PUSH ptr on stack:
   (b) Ptr = ptr -> lchild.

3. if ptr is equal to NULL
(a) pop an address from stack.
(b) traverse the node at the address (display).
(c) ptr = ptr → rchild

C function

```
nrec_inorder(struct node *ptr)
{
  while (top != NULL || ptr != NULL)
  {
    if (ptr != NULL)
    {
      Stack [++top] = ptr;
      ptr = ptr → lchild;
    }
    else
    {
      ptr = stack [top--];
      printf("%d", ptr → info);
      ptr = ptr → rchild;
    }
  }
}
```

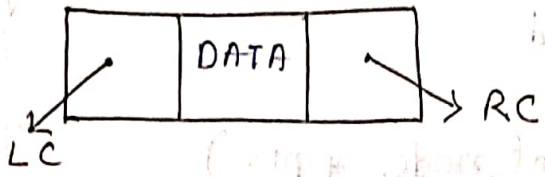
12. An array STACK is used to temporarily stored the address of the nodes.
TOP pointer always points to the top most element of the STACK.

STEP1: Initialize TOP = NULL and PN = ROOT
STEP2: Repeat the step 3 to 5 unit (PN = NULL)
STEP3: TOP ← TOP+1
STEP4: STACK [TOP] ← PN
STEP5: PN ← left (PN)
STEP6: PN ← STACK [TOP]
STEP7: TOP = TOP - 1
STEP8: Repeat steps 9 to 12 until (PN = NULL)
STEP9: Display info (PN)
STEP10: If (Right (PN) is not equal to NULL)
STEP10: (a) PN = Right (PN)
STEP10: (b) Go to step 9
STEP11: PN = STACK [TOP]
STEP12: TOP = TOP - 1

Inorder.

STEP 13: Exit

VI. Linked list representation of binary tree :>



Here, LC and RC are the two link fields used to store the address of left child and right child of a node.

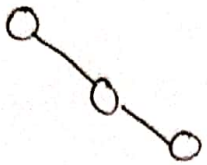
DATA is the information content of the node.

Data	Left child info	Right child info
+	2	3
−	4	5
+	6	7
A	—	—
B	—	—
C	—	—
/	14	15
D	—	—
E	—	—

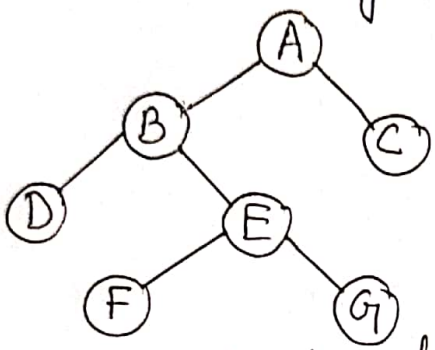
Fig: Linked representation of a binary tree

```
Struct node {
    Struct node * left;
    int data;
    Struct node * right;
};
```

Skewed Binary tree:— A skewed binary tree is a binary tree in which each node has only one left child or a binary tree in which each node has only right child.



Strictly Binary tree :- If every non-terminal node in a binary tree consist of non-empty left subtree and right subtree, then such a tree is called strictly binary tree.



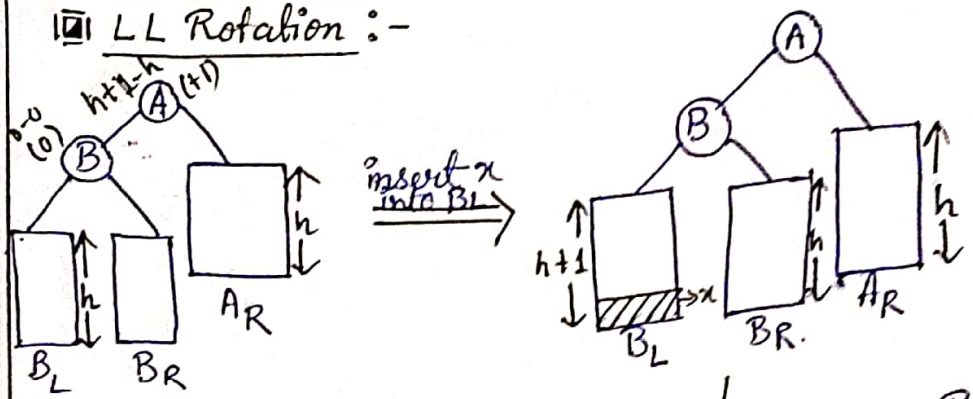
Non terminal node B, E having empty left child & right sub tree.

- Some property of a Binary tree :>

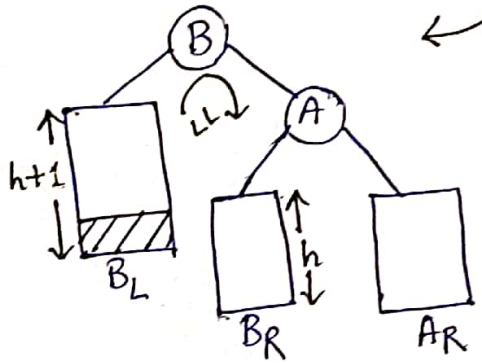
1. A tree with 'n' node has exactly (n-1) edges or branch.
2. The maximum number of node on level (i) of a binary tree is 2^i, $i \geq 0$.
3. Maximum number of node in a binary tree of height n is $2^n - 1$.
4. Relation b/w number of leaf nodes & degree is $\Rightarrow$ for any non empty binary tree, if no represent the number of leaf nodes and n_2 the number of nodes of degree 2 then $n_0 = n_2 + 1$.

POST ORDER TRAVERSAL (NON-RECURSIVE) 19/9/19

LL Rotation :-

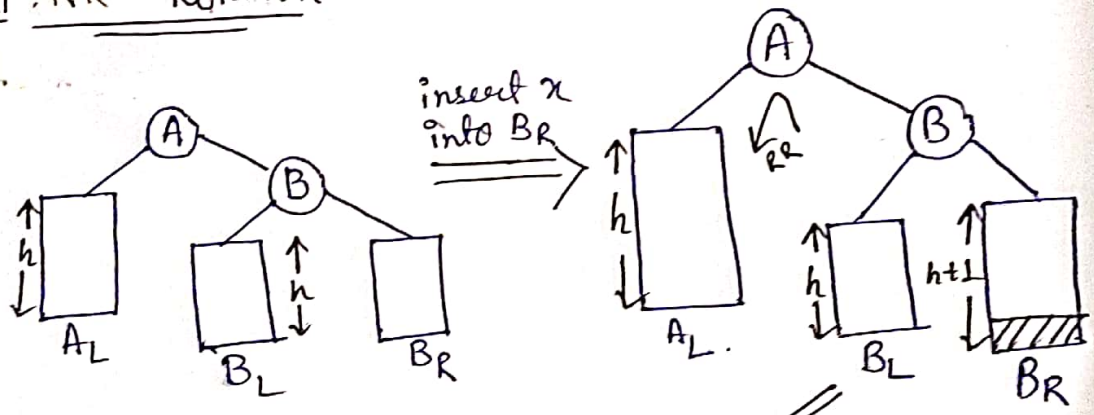


After LL Rotation.

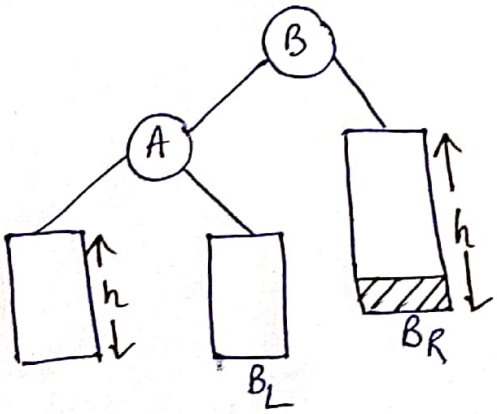


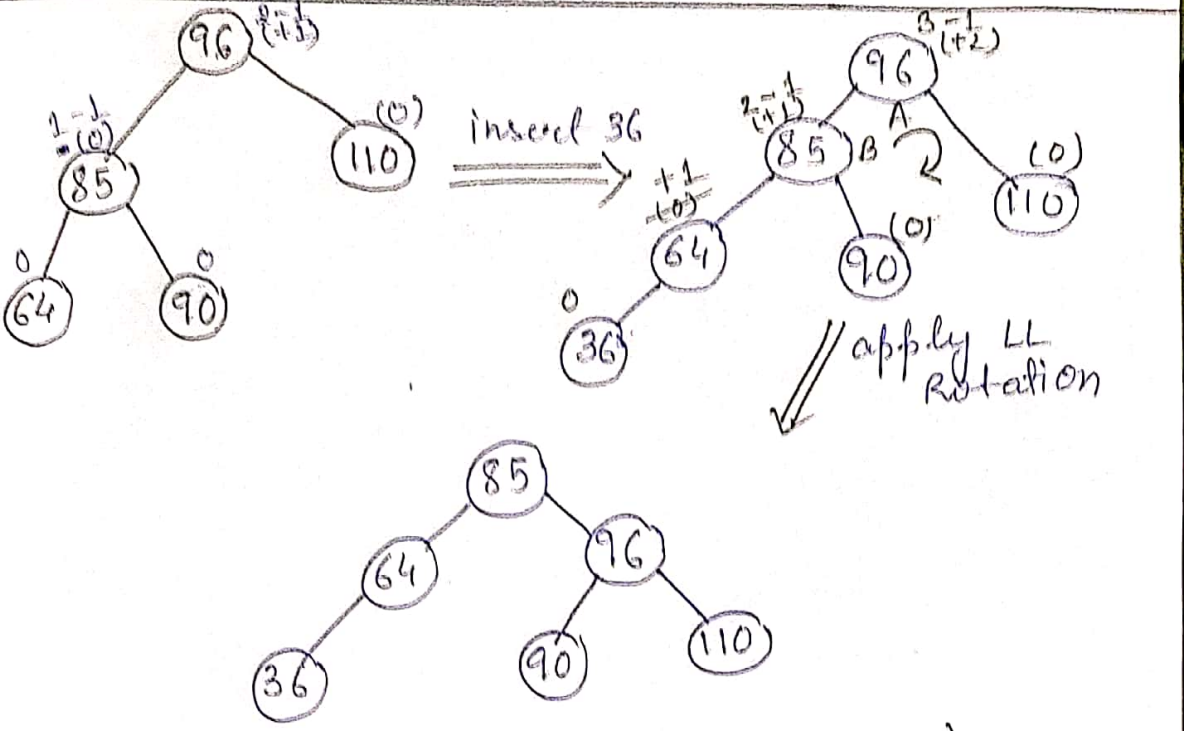
LL Rotation is clock... Rotation.

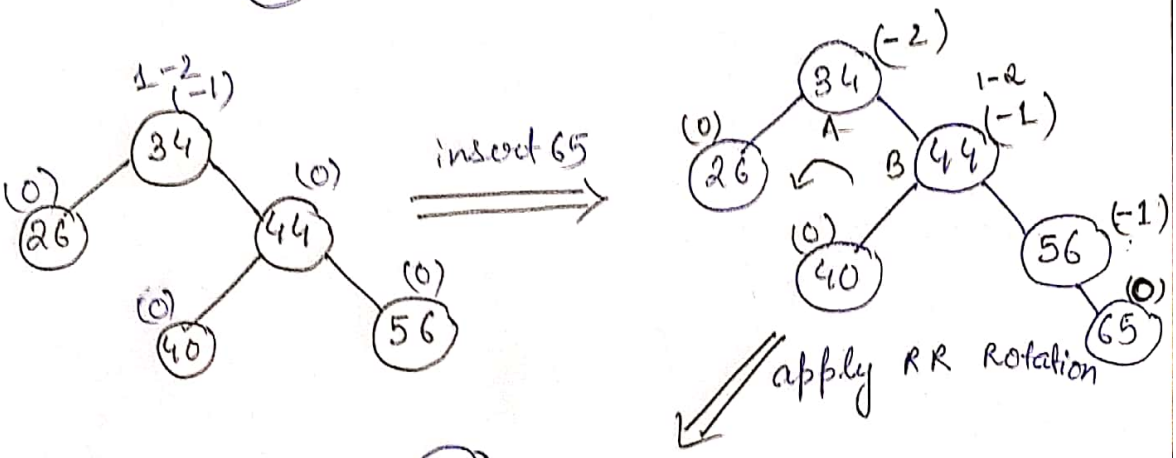
RR Rotation

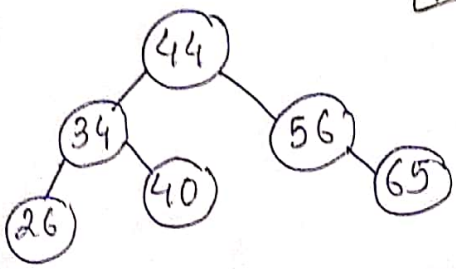


after RR Rotation



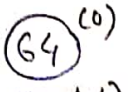






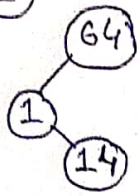
Q. Construct a AVL Tree by inserting the following elements in the order of their occurrence.
64, 1, 14, 26, 13, 110, 98, 85.

insert 64

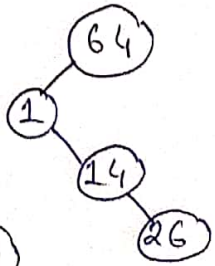


insert 1

insert 14



insert 26



insert 13

LR Rotation