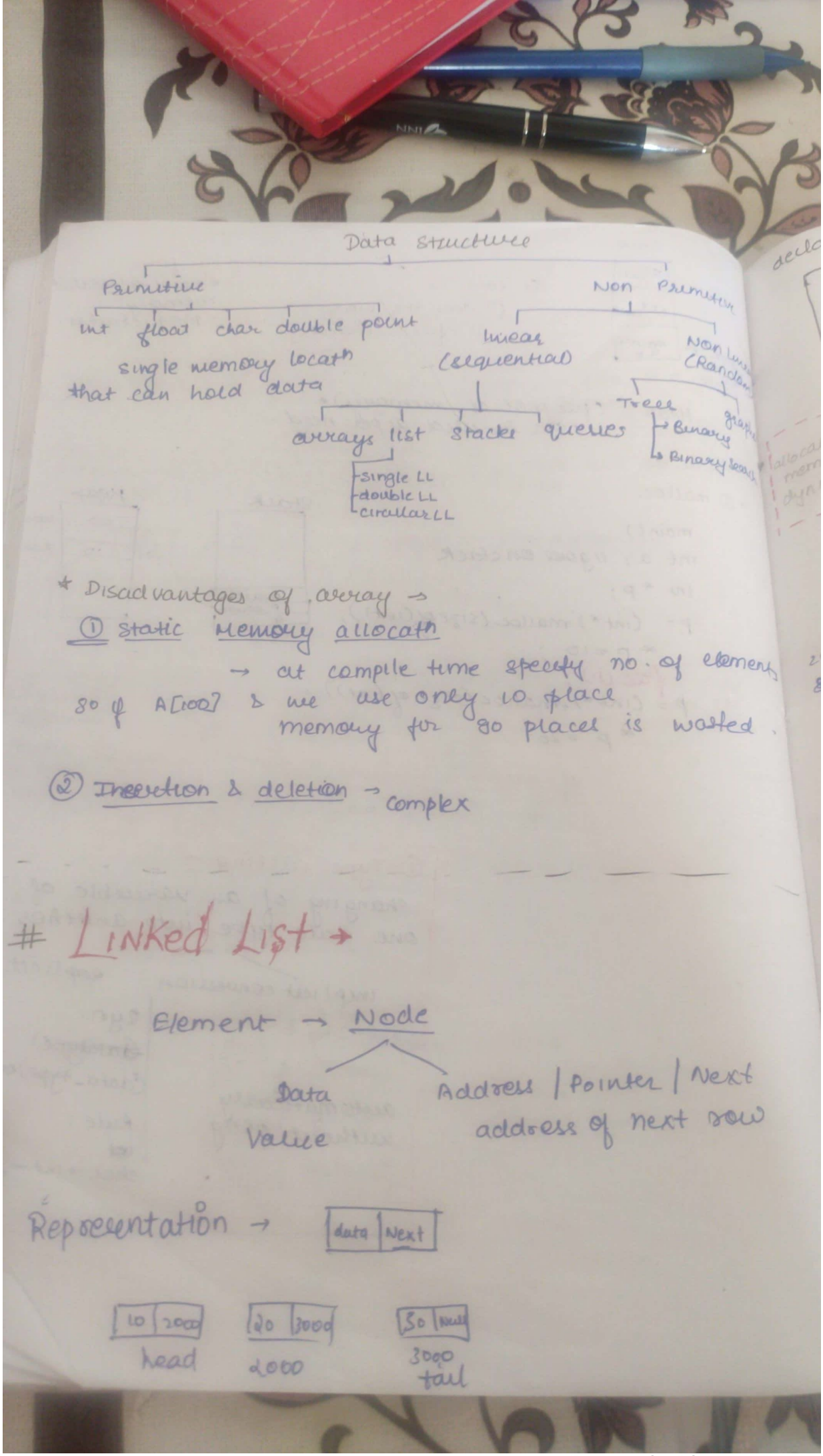


* Disadvantages of array →

① Static memory allocath

→ at compile time specify no. of elements

so if A[100] & we use only 10 place memory for 90 places is wasted.

② Insertion & deletion → complex

LINKed List →

Element → Node

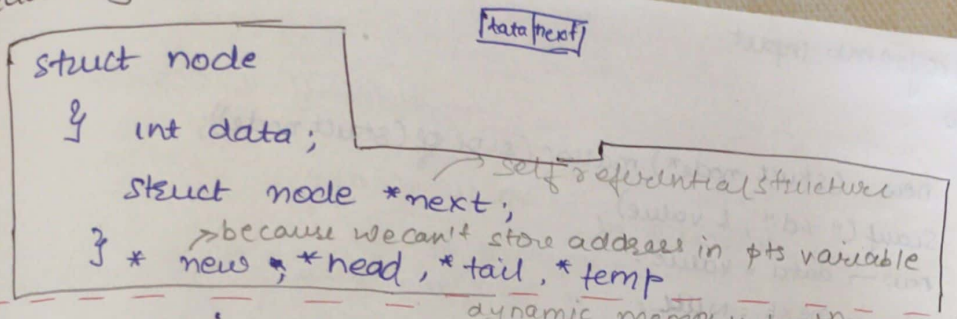
Data Value

Address / Pointer / Next address of next row

Representation →

Scanned with CamScanner

declaring Node ....



dynamic memory location

allocate memory dynamically

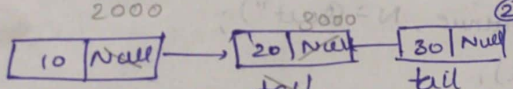
new = (struct node *) malloc (size of (struct node));

datatype (typecast) — to allocate memory — size

creating list →

10 → New
2000 20 → New
3000 30 → New

① establish link
② Create node



fixed ← head
tail

tail → next = new
tail = new

tail of next will be assigned to new

tail → next = new
tail = new

new = (struct node*) malloc (size of (struct node));

Program input

```
do
{
  new = (struct node*) malloc(sizeof(struct node));   // Memory allocation
  scanf("%d", &value);                                // → value intake
  new → data = value;                                 // → data me 20
  new → next = NULL;                                  // → next me NULL
  if (head == NULL)
  {
    head = new;                                       // head or tail bnaaya
    tail = new;
  }
  else
  {
    tail → next = new;                                // tail of next = new
    tail = new;                                       // tail ki location me jo next h usme,
  }                                                   //  new ki location daldi
  printf("Y = continue, N = Quit");
  fflush(stdin);
  scanf("%c", ch);
} while (ch == 'Y');
```

Explanation -

20 → 2000
30 → 3000
50 → 5000

value = 20

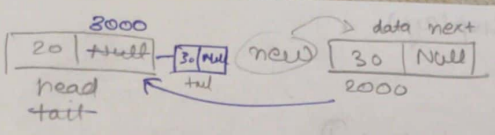
new | 20 | Null |
data Next

As not initialised automatically head | GB | Null tail | GB | Null

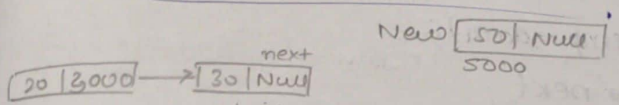
if head == null head | 20 | Null ← tail

if pressed y

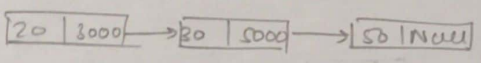
next loop



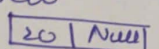
head x null
else ps



Next = 5000
tail = new

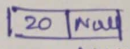


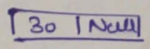
① New



→ head, tail

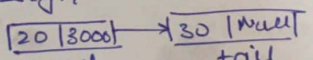
② ~~New~~ agla value lia





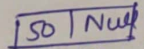
isko 'new' bola ab

(i) tail of 'next' ko assign kra 'new' ka address

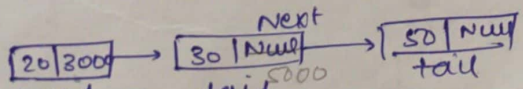


(ii) tail ab 'new' bna

③ agla value lia



isko 'new' bola



tail of Next is assigned ~~to~~ add of new

tail ab new ko bna dia

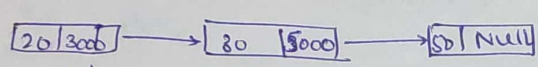
Scanned with CamScanner

Program output (display)

```
temp = head;
while (temp != NULL)
{
    printf("%d", temp->data);
    temp = temp->next;
}
```

access the address of 'next' k pointer ko access krra = agla data access kr

explain →

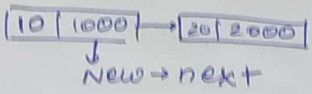


① temp → ① Node (head) ko assign kia
② temp is not null
③ data print (20)
④ temp assigned 'next' address ki value = (30)
⑤ 30 print hua
⑥ temp = 50
⑦ 50 print hua
⑧ temp = 'Next' address per value (NULL)
⑨ condn false exit.

Scanned with CamScanner

Insertion operation →

① Insertion At Beginning → (element is 10)



scanf ("%d", &value);

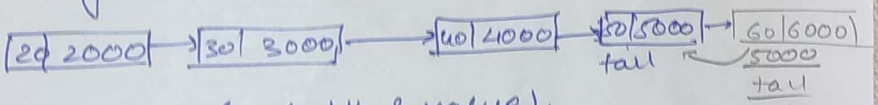
new → data = value — new ka data me 10

new → next = head — new ka next me head ka address

head = new

new ko head bnana

② Inserting at Ending



scanf ("%d", &value);

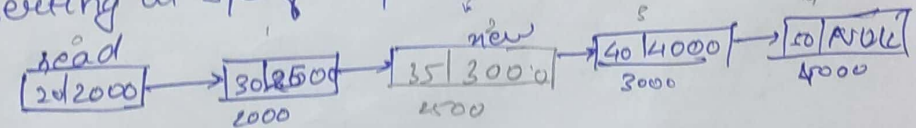
new → data = value; — new k data me value dalo

tail → next = new; — tail k next me new ko dalo

new k next me NULL — new → next = NULL;

tail = new → tail ab new ko bnaya

③ Inserting at specific pos → (pos = 2, value = 35)



scanf ("%d", pos);

scanf ("%d", value);

temp = head

for (i = 0; i <= pos - 1; i++) — habi

temp = temp → next → jab tak i pos na pauche

new k next me temp ka next bhra ←

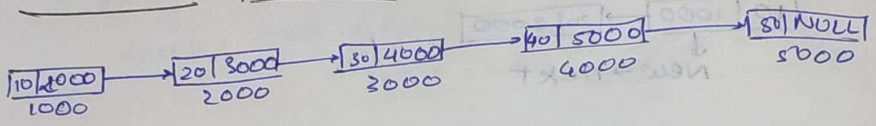
new → data = value — new mg data value

new → next = temp → next

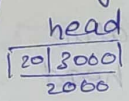
temp k next me new ka address bhr dia

temp → next = new

Deletion Operation →



① Delete At Beginning →




```
temp = head;
temp head = head → next;
temp → next = NULL;
```

② Delete At End →

```
temp = head;
while (temp → next != tail)
  { temp = temp → next; }

temp → next = NULL;
tail = temp;
```

③ Deletion any where

```
for (i=0; i< pos-1; i++)
  { temp = temp → next; }
temp → next = temp → next → next;
```

Scanned with CamScanner

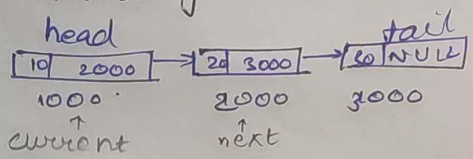
Counting no of nodes →

```
c = 0
temp = head;
while (temp != NULL)
{ c++;
  temp = temp → next;
}
```

Reversing →

```
struct node *current, *next,
            *prev = NULL;
current = head;
while (current != NULL)
{ next = current → next;
  current → next = prev;
  prev = current;
  current = next;
}
head = prev;
```

Reversing →



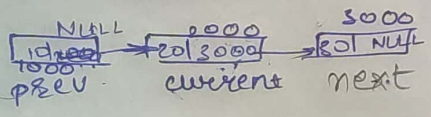
Iteration true

next = 2000

current → next = NULL

prev = 10

current = 20



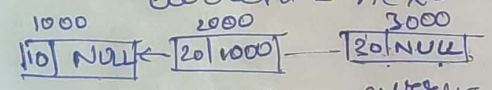
Iteration true

next = 3000

current → next = 1000

prev = current

current = next

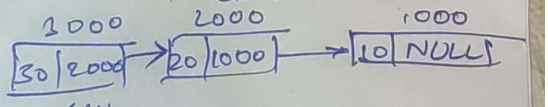


Iteration true

next = NULL

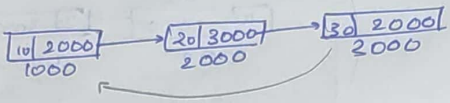
current → next = 2000

prev = 30, current = 30, next = NULL

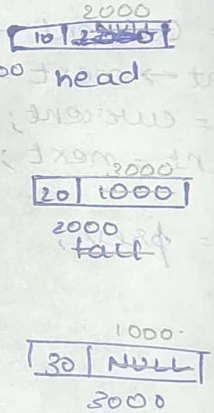


Scanned with CamScanner

CIRCULAR LINKED List




```
struct node
{
  int data;
  struct node *next;
} *new, *head, *tail, *temp;

main()
{
  new = (struct node*) malloc (sizeof (struct node));
scanf ("%d", &value);
  new -> data = value;
  new -> next = NULL;
  if (head == NULL);
    { head = new;
      tail = new;
    }
  else
  {  tail
     head -> next = new;
      tail = new;
      tail -> next = head;
  }

  while (ch = '4');

// display
temp = head;
while (temp -> next != head)
{ printf ("%d", temp -> data);
      temp = temp -> next; }
printf ("%d", temp -> data);
```



last wla nhi hoga isliye

Scanned with CamScanner

Insertion at Beginning →

```
new → data = value;
new → next = head;
tail → next = new;
tail → head = new;
```

Insertion at End

Insertion at anywhere

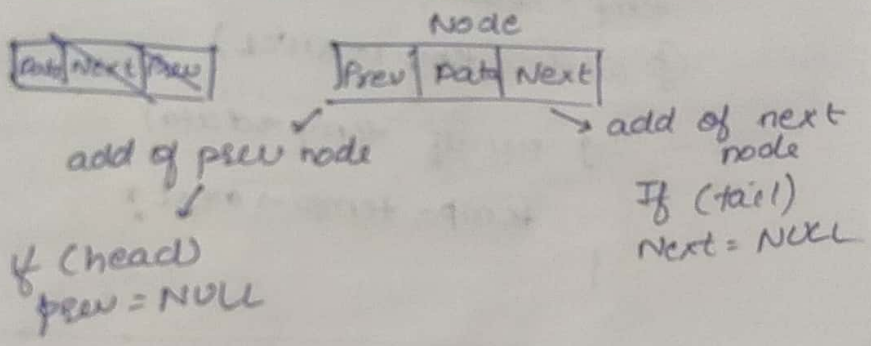
Prog written (simple)

~~Insertion~~

Deletion

① At Beginning, end, any where (simple logic see laptop)

Double Linked List

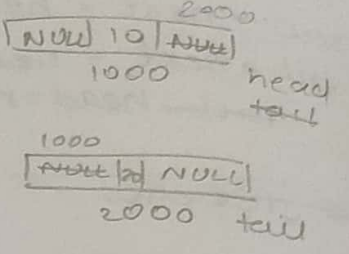



```
struct node
{ int data;
  struct node * prev;
  struct node * next;
};
{ new*, temp*, head*, tail*;

main()
do
{
  new = (struct node *) malloc (sizeof (struct node));
  scanf ("%d", &value);
  new → data = value;
  new → prev = NULL;
  new → next = NULL
```

NULL 10 NULL

Scanned with CamScanner

```
If (head == NULL)
{
    head = new;
    tail = new;
}
else
{
    tail → next = new
    new → prev = tail
    tail = new
}
while (ch == 'y')
temp = head;
while (temp != NULL)
{ printf temp → data)
    temp = temp → next;
}
```



Insertion

① At Begin →

```
new → prev = NULL;
new → next = head;
head → prev = new;
head = new;
```

② At end →

```
tail → next = new;
new → prev = tail;
new → next = NULL;
tail = new
```

③ Anywhere →

```
new → prev = temp
new → next = temp → next
temp → next = new
```

Deletion →

① At Begin

```
head = head → next
head → prev = null
```

② At end

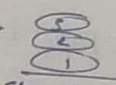
```
tail = tail → prev;
tail → next = NULL;
```

③ Anywhere

```
temp → next = temp → next → next
temp → next → prev = temp
```

Scanned with CamScanner

STACKS using ARRAYS

① Linear DS (sequential)
② Follows LIFO (Last in First out) →  ③ removed last
③ push → inserting an element into stack
pop → deleting " " " "
Top → top of stack < Push / Pop
★ Top = -1 (initial value → stack is empty)

Implementing in two ways →
① Arrays ② LL

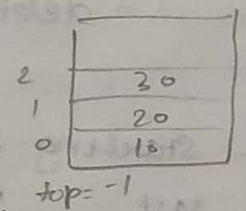
- overflow condn (Push)
  → stack had 5 element can only have
  Insert 6th element
  → inserting an element to a stack which is already having max elements.

- underflow condn (Pop)
  → Deleting an element from an empty stack.

- implementation → stack[10], size = 10, top = -1

PUSH

```
push()
{
  top++;
  scanf("%d", element);
  If (top == size-1)
    { printf("overflow"); }
  else
    { top++
      stack[top] = element;
    }
}
```

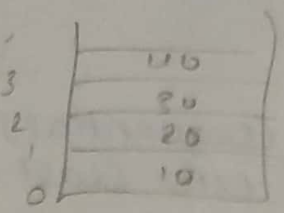


POP

```
pop()
{ if(top == -1)
    printf("underflo
  else
  { element = stack[top]
    top--; }
```

Scanned with CamScanner

```
display()
{ if (top==-1)
  { for (i=top; i>=0; i--)
    { printf ("%d", stack[i]); }
  else
    printf ("empty stack");
```



i = 3 3>=0
stack[3] = 40

USING ARRAYS — QUEUE — implement <arrays / LL

- Linear DS (sequential)
- Follows FIFO (1st in 1st out)

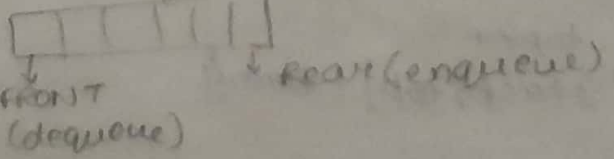
Standing in line for ticket, come 1st get 1st

- operations →
  - → ENQUEUE → inserting
  - → DEQUEUE → deleting

- → CONDITIONS →
  (enqueue) → OVERFLOW → Insertion in a Queue that's full
  (dequeue) → UNDERFLOW → deletion from empty Queue

- → two ends →
  - → FRONT → starting element
  - → REAR → last "

- → enqueue will be done at REAR end.
- → dequeue " " " " FRONT "

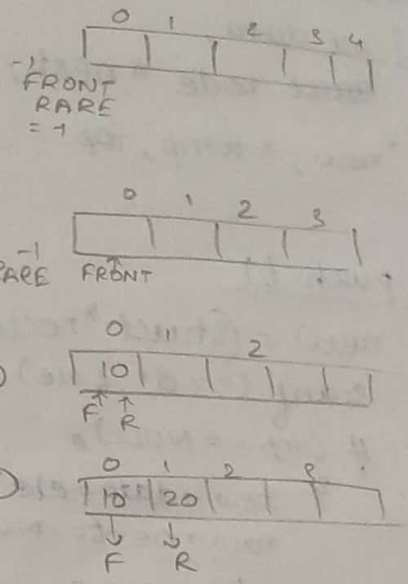


- ★ Initially FRONT = REAR = -1

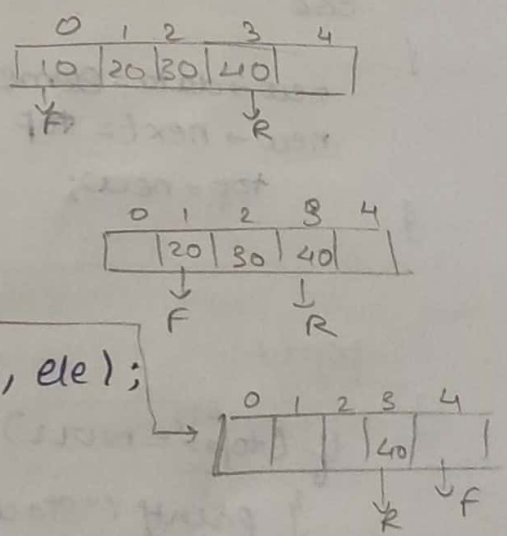
Scanned with CamScanner

Implementn →

```
SIZE = 10
FRONT = RARE = -1
QUEUE [];

ENQUEUE()
{  if (REAR == SIZE-1)
      { printf ("overflow");
   else
{ scanf ("%d", ele);
    if (FRONT == -1)
     { FRONT = 0; }
     RARE++;
   {  QUEUE [RARE] = elet;
```




```
DEQUEUE ()
{
   if (REAR == -1 || FRONT > REAR)
    { printf ("underflow"); }
else
   { ele = QUEUE [FRONT];
     printf ("element %d is deleted", ele);
     FRONT++;
   }
```

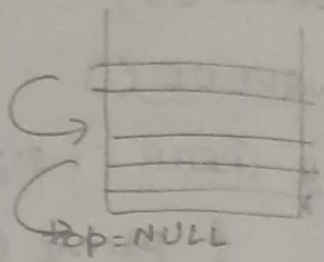



```
DISPLAY()
{
   if (REAR == -1 || FRONT > REAR)
{ printf ("empty queue"); }
else
{ for (i=FRONT ; i<=REAR; i++)
    { printf ("%d", Queue [i]);
   }
}
```

Scanned with CamScanner

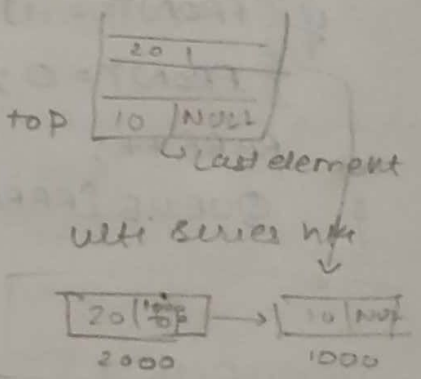
STACK using LL

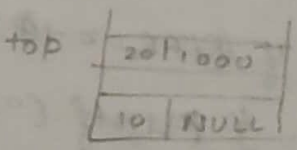
```
struct node
{ int data;
  struct node *next;
}*new, *temp, top
```



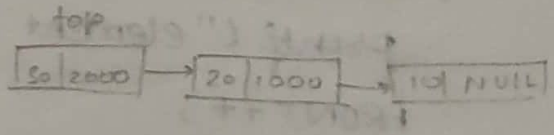
push ()

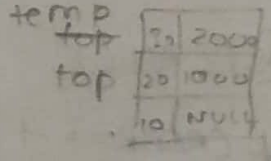
```
{ new = (struct *node) malloc (sizeof (struct node));
  scanf ("%d", &ele);
  if (top == NULL)
    { new->data = ele;
      new->next = NULL;
      top = New;
    }
  else
{
    new->data = element;
    new->next = top ; new k next mx
                      top
    top = new;
}
```





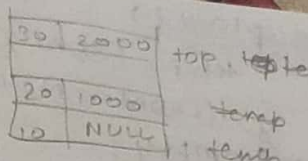

```
pop()
{
  if (top == NULL)
    { printf ("stack empty"); }

  else
    { temp = top;
      ele = top->data ; (elem. is deleted)
      top = top -> next;
  or[ temp -> next = NULL;
      free (temp);
    }
```





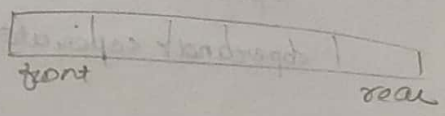
Scanned with CamScanner

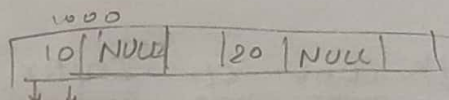
```
display()
{ if (top==NULL)
  { printf("stack is empty"); }

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

}
```

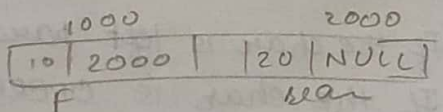


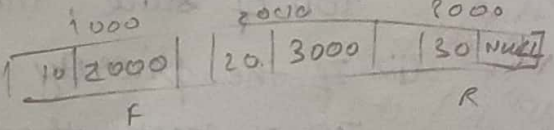
QUEUE USING LL

```
struct node
{ int data;
  struct node *next;
};*new,*temp,*front,*rear;

enqueue()
{ new = ( ) malloc( );
  scanf("%d",&ele);
  if (FRONT==rear)
  new -> data = ele;
  new -> next = NULL;
  if (rear==NULL)
  { front = new;
    rear = new;
  }
  else
  { rear -> next = new;
    rear = new;
  }
```

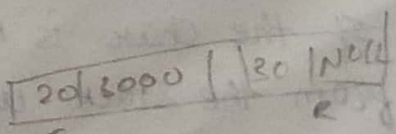






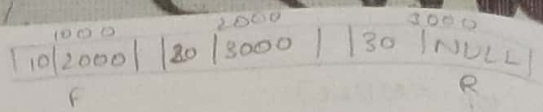



```
dequeue()
{
  if (FRONT==NULL)
    printf("empty");
  else
  { temp=front;
    front = front->next;
    free(temp);
  }
```



Scanned with CamScanner

```
display ()
{
    temp = FRONT;
    if (FRONT == NULL)
        write empty;
    else
        while (temp != NULL)
            printf ("%d", temp->data);
        temp = temp->next;
}
```



Applications of Stack

① Infix expression →

operands → A, B, -- operators → + - * / ^

<operand> <operator> <operand> → A+B

② Postfix expression

(A+B) AB+

③ Prefix exp →

+AB (A+B)

If Conv. of Infix to Postfix exp (Algorithm) :-

STEPS →

① If the char is left paranthesis, push to the stack.

② If the char is OPERAND, add to the POSTFIX exp.

③ If the char is OPERATOR, check whether stack is empty.
If empty - push to stack. If not, check priority
(i) If priority of OPERATOR > OPERATOR present at TOP of stack, then PUSH the operator in the stack.
(ii) If the priority of operator ≤ operator present at TOP of stack, then pop the operator from stack and add to POSTFIX exp and goto step (i).

If its left para don't check just push in stack

④ If the char is RIGHT Paranthesis, then pop all the operato from the stack untill it reaches left par & add to postfi

⑤ After reading all char, if stack is not empty then pop and add to postfix.

Scanned with CamScanner

for eg A+B*C

A → char (operand) by step 2

+ → operator step3

→ stack is empty - push to stack

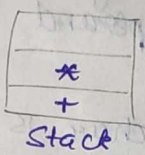
B → operand step 2

* → operator step 3 (i)

C → operand step 2

step 5 ABC*+

postfix: A B C



Stack

eg② a-(b*c-d)/e → Infix

char	stack	postfix
a		a
-	-	a
(	-(	a
b	-(	ab
*	-(*	ab
c	-(*	abc
-	-(-	abc*
d	-(-	abc*d
)	-	abc*d-
/	-/	abc*d-
e	-/	abc*d-e
		abc*d-e/-

(→ push

A → postfix

+ → empty - Push

not empty

is of ↑ priority → push

↓ pop before & postfix it & check again

) → pop all till (add to postfix

last → pop, add to postfix

we don't insert (,) in postfix

Scanned with CamScanner

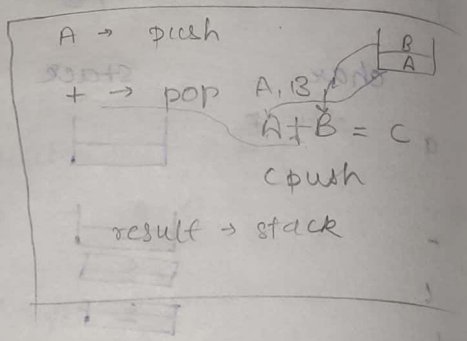
```
* Create()
  int x;
  newnode = (struct node *)
  printf("Enter data");
  scanf("%d", &x);
```

2) Evaluation of Postfix expression →

① only stack is used.
② operand stack

① If char is operand, PUSH into stack.
② If char is operator, pop top 2 operands from the stack, perform calculation, push the result back into stack.

After reading all the char from postfix exp. STACK will be only having RESULT.



Eg → 562 * * +

char	stack
5	5
6	6, 5
2	2, 6, 5
*	12, 5
+	17

remove 2, write 6 on left side, put * & perform 6*2 = 12 calculat

5 + 12 = 17

Result = 17

Eg. 4325 * - +

char	stack
4	4
3	5, 4
2	2, 3, 4
*	3, 4
5	4

3) Balancing

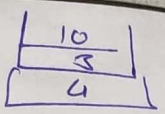
Balancing symbols: ( ) [ ] { }

Algorith
① Read c
② If char into sta
③ If ch stack is
④ After

Examp

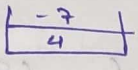
Scanned with CamScanner

*



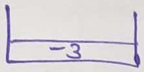
2 * 5 = 10

-



3 - 10 = -4

+



4 + -7 = -3

Ans = -3

3) Balancing Symbols →

Balancing symbols: ( ) > exp.
[ ]
{ } → block of statements

→ Balancing every open symbol should have closed symbol

Algorithm →

① Read char from exp.
② If char is open symbol '(', '[', '{', push symbol into stack.
③ If char is closed symbol),], }, check if stack is empty
- If empty, exp is unbalanced
- If not, pop the symbol from stack and compare with the symbol which is read.
  - If matches, repeat the process
  - If not, unbalanced.

④ After reading all the exp, stack - not empty → unbalanced.

Example →

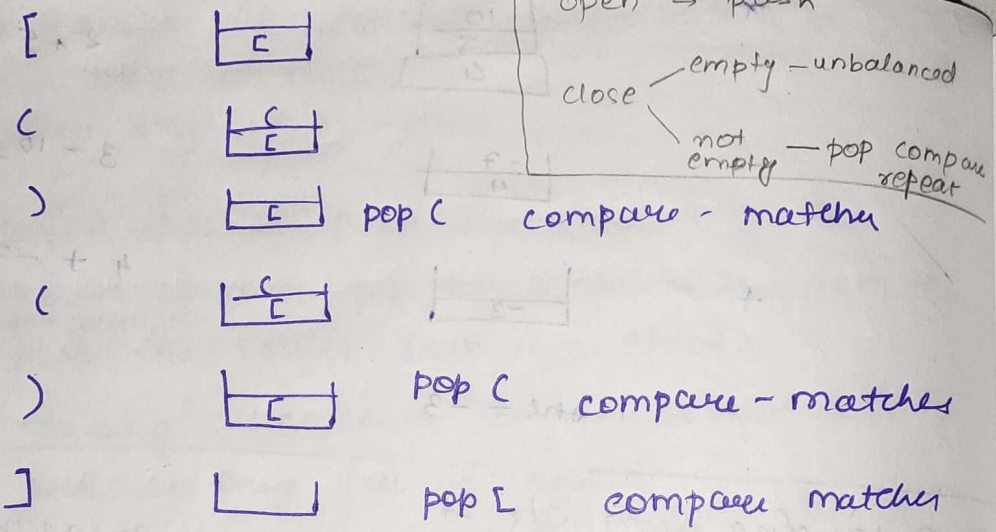
[(a+b)(a-b)]

Scanned with CamScanner

= create;

create will return pointer to a node (eg 100)

* create()
int x;
D newnode = (struct node *) malloc(sizeof
typecast
printf("Enter data");
("%d", &x);



BALANCED EXPRESSION

Node →

Root node

Edge →

Parent

CHILD

SIBLINGS

Leaf

Interna

DEGRE

Scanned with CamScanner

push

pty — unbalanced

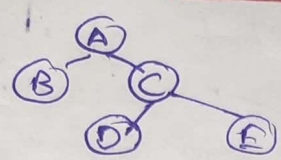
pty — pop compare repeat

matches

- matches

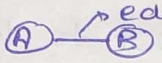
matches

TREE non linear hierchal fashion



Node → element
A, B, C

Root node → starting node
A is root node (only 1)

Edge → 

→ link/connection b/w 2 nodes.
N nodes − (N−1) edges.

Parent → A, C Node with branches

CHILD → Node with edge from bottom to top.
Branches of parent.
B, C, D, E

SIBlings → child nodes of same parent.
B, C & D, E

Leaf → Node without ~~chain~~ child
B, D, E

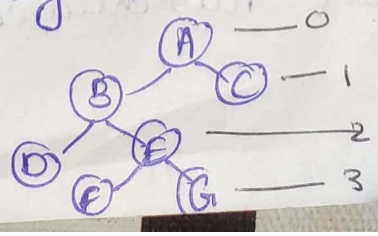
Internal nodes → All nodes other than leaf
Node with child.
A, B, C

DEGREE → no. of ~~child~~ child nodes.

degree (A) = 2
degree (B) = 2
degree (C) = 2
Max degree = Degree of tree = 2

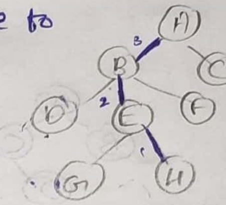
Level → every step in tree
starts from 0.
For every step / hierachy level +1

level of tree = 3
level of root node = 0



Scanned with CamScanner

Hieght → longest path from leaf node to that node is height



Hieght (B) = 2
hiegh (A) = 3

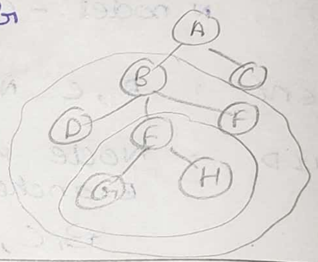
depth = longest path from root node to that node.

depth (E) = 2

Path → sequence of nodes from root to leaf

Path (A to G) = A – B – E – G

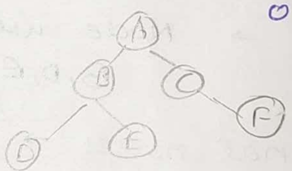
Subtree → Node with child node



Binary Tree

→ every node in a tree should have atmost 2 children

0, 1, 2 children



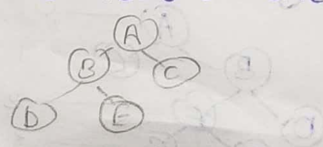
~~★ If a newly connected node value is greater than root then it goes on right side~~

Types →

- Full binary tree / strictly binary tree
- Almost complete binary tree | incomplete binary tree
- complete binary tree | Perfect bT
- left skewed B T
- right skewed B T

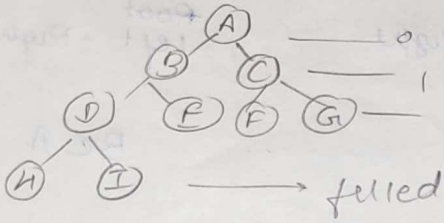
Full B | Strictly B →

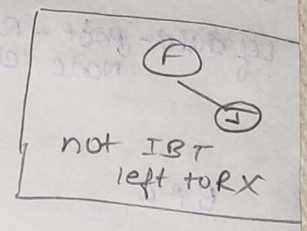
every node must have two children except leaf



Incomplete B →

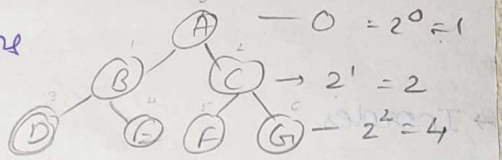
every node must have 2 children in all levels except in last level but filled from left to right.





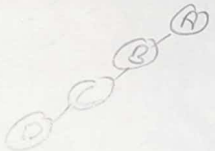
CBT →

every node must have 2 childrens in all levels.
each level 2^L nodes.



LSBT →

every node should have only left chain.



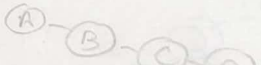
no. of nodes max in a tree with height h

$= 2^{n+1} - 1$

Height of complete binary tree $= \log_2 n$

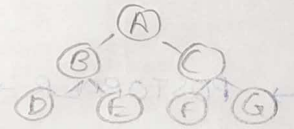
Max height $= n+1$

RSBT



Binary Tree Representh

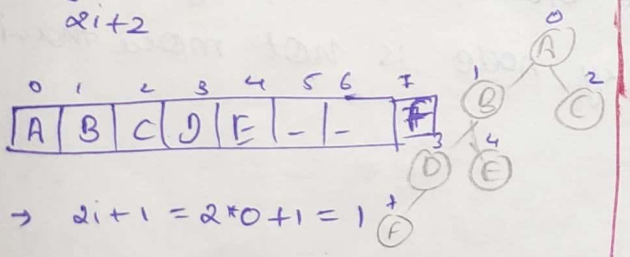
Array — LL



(1) Using Array →

Algo:

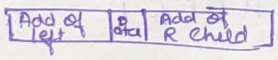
1. Consider root node at index 0
2. left child is placed at $2i+1$
   i → parent node
3. Right child is placed at $2i+2$

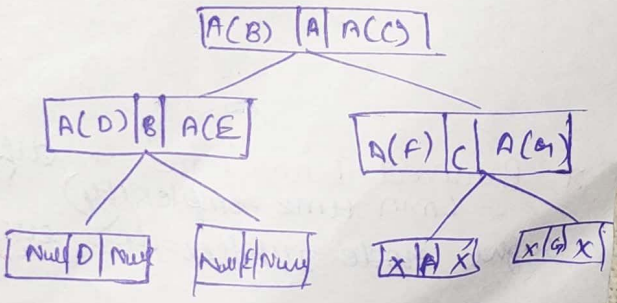


B → $2i+1 = 2*0+1 = 1$

F → $2i+1 = 2*3+1 = 7$

(2) Using LL →





A(B) → add of B

X → Null

Scanned with CamScanner

TREE TRAVERSAL

① Inorder

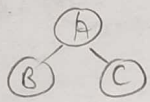
Left child - Root Node - Right child

BAC

② Preorder

Root Node - left - Right

ABC



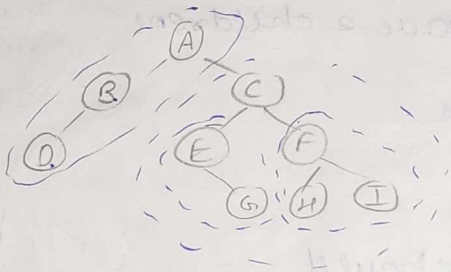
③ Post order

~~Root~~ Left - Right - Root

BCA

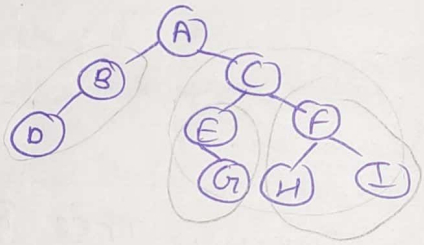
→ Inorder

DBAEGCHFI



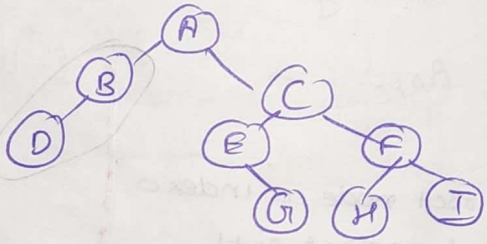
→ PREORDE

ABDCEGFHI



→ POSTORDER →

DBGEHIFCA



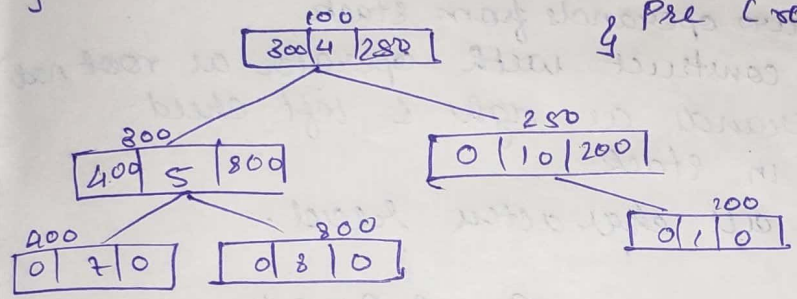
* Balanced binary tree → diff b/w weight of left & right ~~node~~ subtree for every node is not more than (min time complexity)

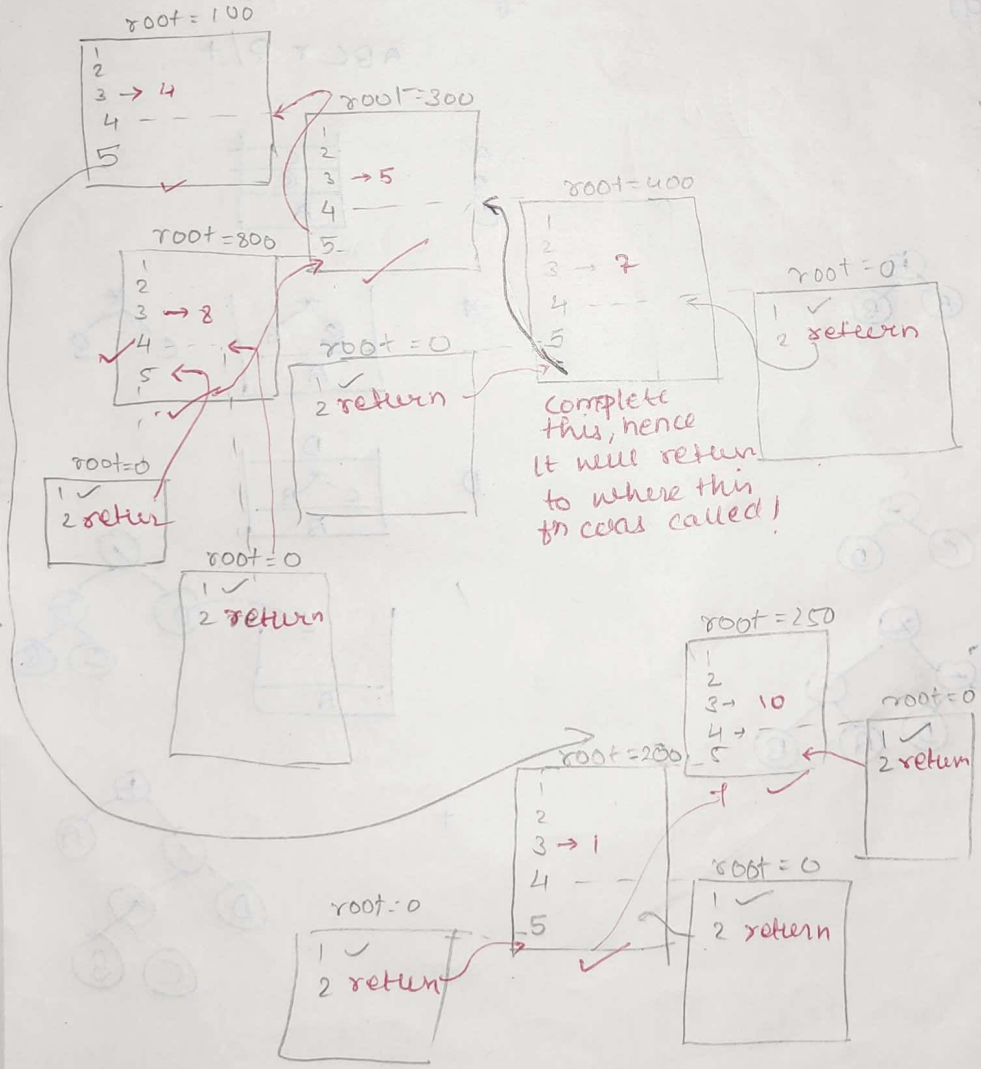
Scanned with CamScanner

Preorder Code & Memory allocatn in stack

```
main()
{ struct node *root;
  write Preorde is;
  Pre (root);
}
```

```
void Pre (struct node *root)
{ if (root==0)
    return;
  printf ("%d", root->data);
  Pre (root->left);
  Pre (root->right);
}
```





Scanned with CamScanner

```
main()
{ root = create;
```

create node
return pointer
to a node (for eg 100)

```
* create()
{ int x;
  newnode = (struct node*) malloc(sizeof(struct node));
  printf("Enter data");
  scanf("%d", &x);
```

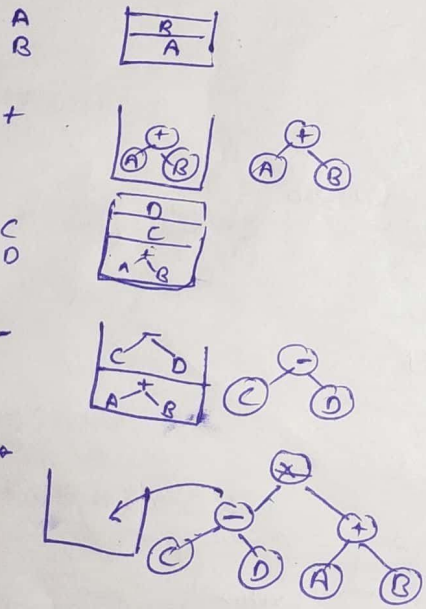
typecast

Expression Tree

1. Convert Infix to Postfix.
2. Read all character
3. If character is operand, push in stack.
4. If " " Operator
   (i) pop two operands from stack
   (ii) Tree construct with operator as root node & operands as right & left child
5. Push tree back in stack
6. Repeat until all characters read.

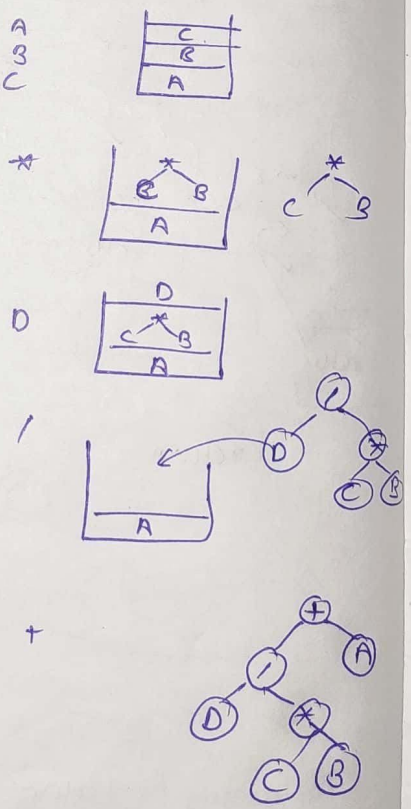
Eg. (A+B) * (C+d)

AB+ CD- *



Eg A+B * C/D

ABC * D/+



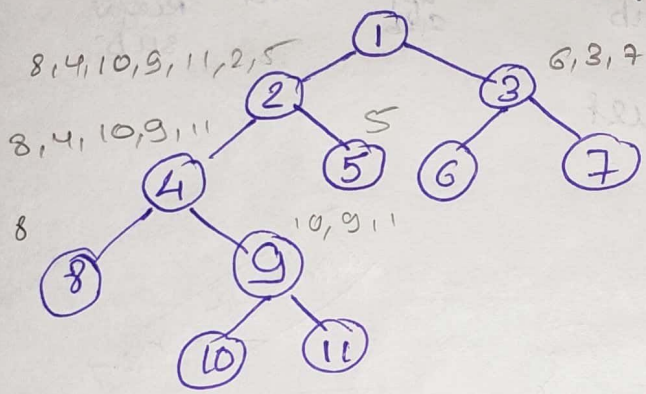
Scanned with CamScanner

Constructing BT from Preorder & Inorder

~~Pre → 1,2,~~ First complete left side

LRR Pre → 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7

LRR In → 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7
(left) (Right)



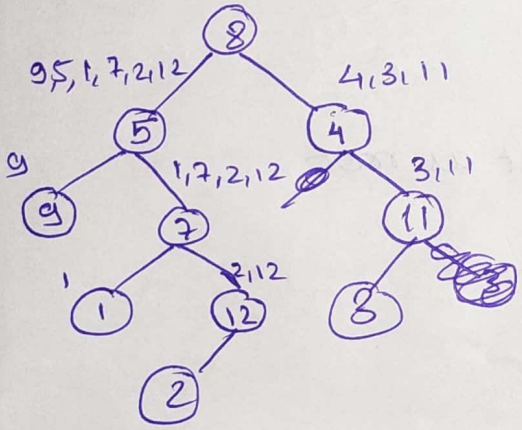
1) If 1 is 1st element in Pre (which is ROOT)
2) Mark 1 in inorder and left ← 1 → Right sub sub
3) Out of left ele, the first one coming in Pre is child

Post & In

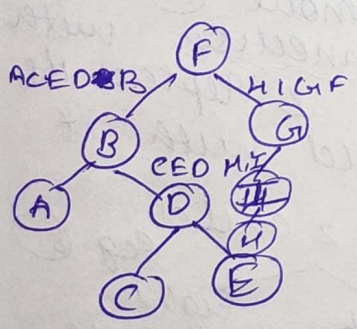
LRR Post → 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

LRR In → 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
(left) (right)

1) Last element in Post is root
2) Mark 8 in Inor, & left me left sub & right me right sub
3) In Pre from right to left whatever comes 1st from left sub ele is child
4) 1st complete left



Pre & Post (not possible to construct unique BT but possible to complete BT)



Pre → F B A D C E G I H ERoLR

Post → A C E D B H I G F LRRo

Pre B A D C E
Post A C E D B

D C E
C E D

1) 1st element in pre is root (F)
2) check next successor in pre & mark it in post
left sub + F + B → Right sub

G I H
H I G

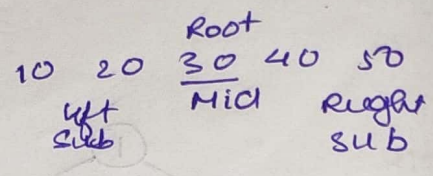
3) Predecessor of F in post Mark G in pre
left ← G → Right

Pre I H
Post H I

Scanned with CamScanner

Binary search tree

1. Every node → 0, 1, 2 child
2. sort elements
3. Binary search
4. All elements of left sub must be ≤ root
5. All ele of right sub must be > root.

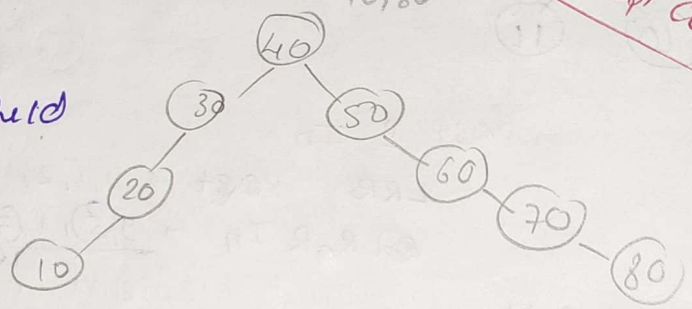


Operations →

1. Insertion
2. Delete → 1, 2 child
3. Search
4. Min & Max

40, 30, 20, 50, 60, 70, 10, 80

Heap + BST code



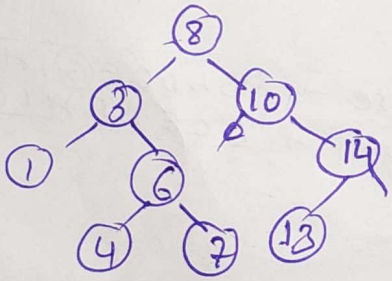
Insertion →

1. If tree is empty – set element as root
2. If element < root — Left sub
3. If " > " — Right sub.

Deletion →

→ Deletion a node ① having no child :

8, 3, 10, 1, 6, 14, 4, 7, 13



LEAF NODE → 4, 7, 13

① Remove connection b/w leaf node & parent

② having ① child

for eg ⑭

remove ⑭ node connection with ⑩ & replace its child with it

③ having 2 child eg ⑧

left sub replace ↑ ele

Right replace Smallest element.

Searching

① key == Root (Mid ele is root) key == NULL
return root; return root;

② key < Root
search space - left sub → search (root → left, key)

③ key > Root
search space - right sub
return (root → right, key)

Min / Max element

Min → left sub

Max → right sub

AVL Tree

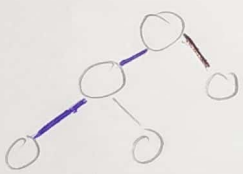
1) Type of Binary search tree.

2) self balancing binary search tree

Balancing factor = Height of left sub - Height of right sub
(0, 1, -1)

Balancing factor (-1) → Heavy Right
(0) → Balancing
(1) → Heavy left.

Eg. BF = 2 - 1 = 1



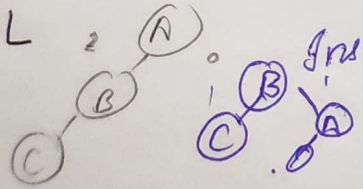
Left Heavy Tree / Heavy left.

Operations →

① Deletion ② Insertion

Rotations →

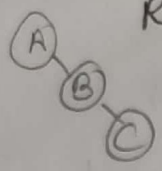
① LL :



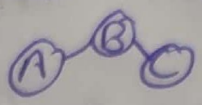
Inserting node on left of left sub to balalance

Scanned with CamScanner

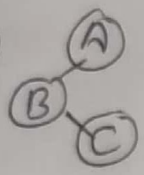
(2) RR



Inserting node on right of right sub



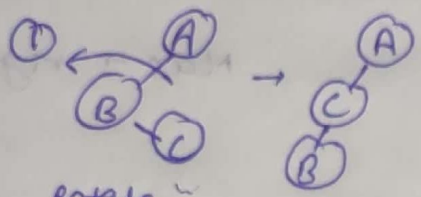
(3)

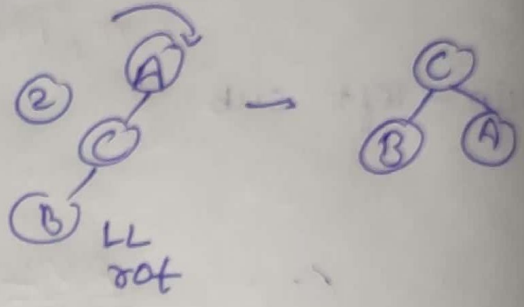


Inserting a node on right of left sub

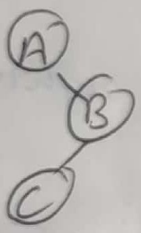
LR

2 steps



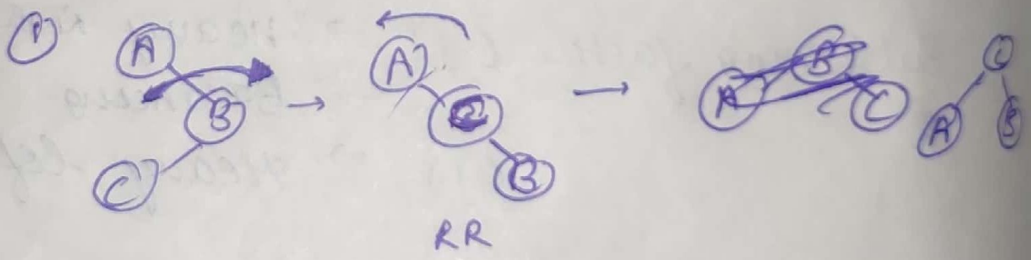


(4)



RL

Inserting node on left of right sub



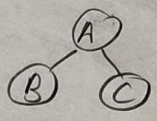
Red - Black Tree

(1) AVL type

Scanned with CamScanner

Graphs

→ set of (V, E) pairs
V → vertices set
E → set of edges

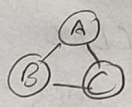


Graph / Tree

→ vertices → Represented as circles also known as nodes.

→ Edges → Represented as lines connecting 2 vertices/nodes

A, B, C → vertices
{A, B}, {A, C} → edges

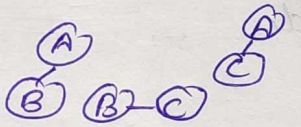


Graph

AB
BC → edges
AC

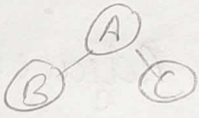
Terminology →

① Adjacent nodes →

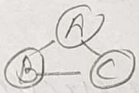


A & B
B & C
C & A which connect same edge

② Degree of Node → No. of edges connected to that node

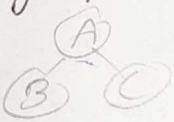


A → 2
C → 1



A
B → B 2
C

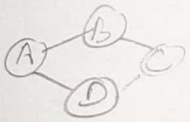
③ Size of graph → Total no. of edges in graph



Size = 2

Size → 3

④ path → sequence of vertices from source node to destination node. (vertices & edges not repeat)



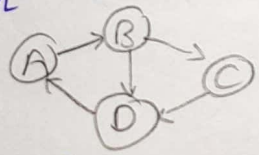
A → C →
A → B → C
or
A → D → C

<A, B, C>

⑤ Trail → a walk in which no edge is repeated

Types of Graphs →

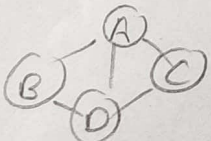
① Directed or digraph



(A, B) ≠ (B, A)
in 1 direction

(B, C)

Undirected

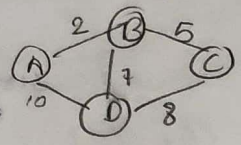


Bidirectional
{B, C}

(A, B) = (B, A)
(B, C) = (C, B)
can go in any direction

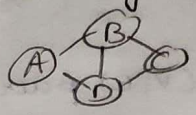
Scanned with CamScanner

② Weighted

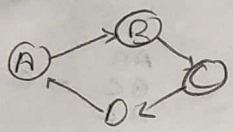


cost/weight is specified for every edge

Unweighted



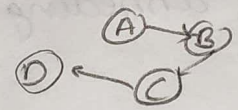
③ CYCLIC

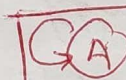


starting node = end node

A → B → C → D → A

ACYCLIC





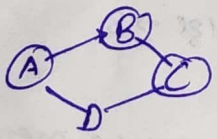
self loop

A can have edge with itself

↓ can be used to refresh a page

Representation →

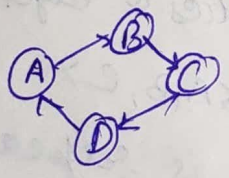
① By using Multidim. Array → Adjacency Matrix



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

0 → if edge not
1 → if edge yes

A to A → 0
A to B → 1



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

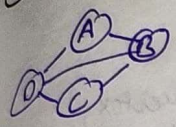
* V are represented as → { V_1, V_2, V_3 }

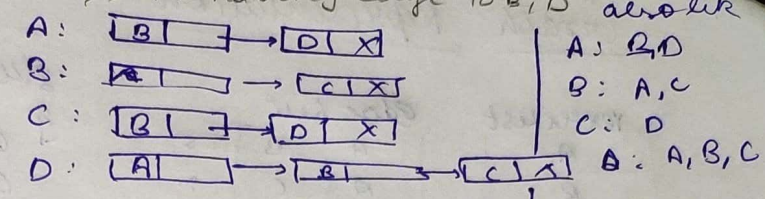
E as → { {V_1, V_2}, {V_3, V_4} }

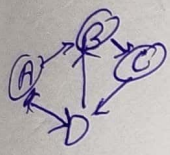
Scanned with CamScanner

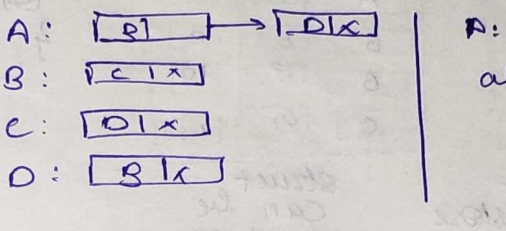
② using list →

→ A is having edge to B, D also like



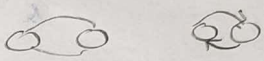






Properties →

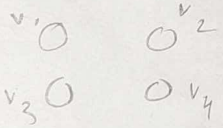
① Multiedge / Parallel edge →



like in airlines H 200 100 Delhi could have diff cost

② No. of edges →

min → 0



$V = \{v_1, v_2, v_3, v_4\}$

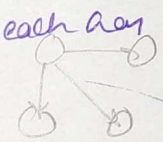
$|V| = 4$

$|E| = \emptyset$

max in directed → Node × edge each can

$= 4 \times 3$

$= 12$



if $|V| = n$

then $0 \leq |E| \leq n(n-1)$ (directed)

$0 \leq |E| \leq \frac{n(n-1)}{2}$ (undirected)

Graph is called dense → too many edges

called sparse → less edges

Scanned with CamScanner

Graph Representation →

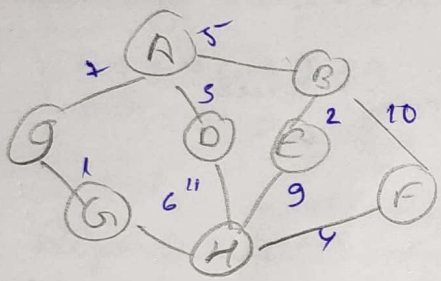
Vertex List
A
B
C
D
E
F

Array can be used to store vertices

Edge list		
&A	&B	5
&A	&C	7
&A	&D	3
&B	&E	2
&B	&F	10
&C	&G	1

struct can be used to store end & starting vertex

```
struct edge
{
    char *startvertex;
    char *endvertex;
    int weight;
};
```



Scanned with CamScanner