Write a program :) void main() ✓ jenny
Write a Function :) void main() ✗ love babbar ✓
code with harry ✓

j 3 5 + x 1

Stack A + B (AB+)

→ Array Imp of Stack.
(Icon, push, pop, peek, display functions)
→ LL Imp. of stack (option)
→ Paranthesis checker ✓✓✓ (5 marks)
→ Infix to Postfix sums (using stack) ✓
→ Postfix Eval (Sum) ✓
→ Recursion using stack ✗ options

LL

→ Singly (Disp) Doubly Circular

Singly: Ins, Del, Ins, Del, Display
Doubly: Dis, Ins, Del
Circular: Ins, Del, Display

leng leng leng

Ins: end | beg
pos (5 marks)

Queue
linear queue
Circular queue
priority
Deque

} Insert deletion design } Optional

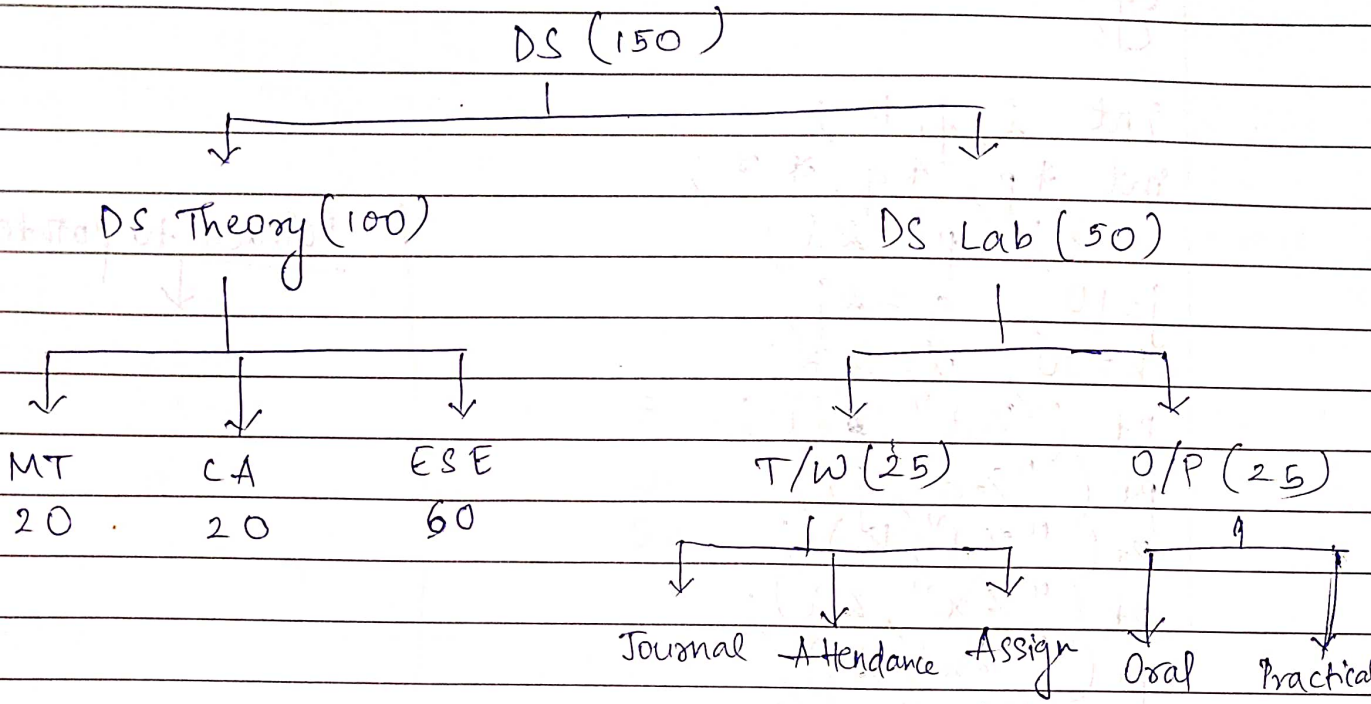
Queue using stack
LL using Queue
LL using stack

Structure & pointer

(* head). next head → next.

16/20 20/20

Data structures. (Lang - C)

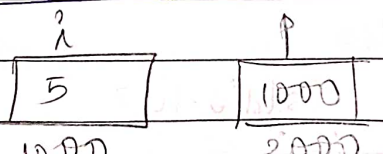


Reema Thareja
+
NPTEL
+
Online links in Syllabus

- Stack
- Queues
- Linked list
- Trees
- Graphs
- Searching.

Pointers

*	Value at
&	address of

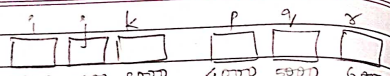



```
int i;
i = 5;
int *p;
p = &i;
printf ("%d", i);      5
printf ("%x", &i);     1000
printf ("%x", p);      1000 → (if %d instead of %x then error)
printf ("%x", &p);     2000
printf ("%d", *p);     5
```

```
int j;
j = 10;
int *q;
q = &j;
```

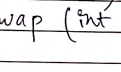
*p
Value at (p)
Value at (1000)

eg.




```
int i, j, k;
int *p, *q, **r;
i=5;  p=&i;
j=10;  q=&j;
k=20;  r=&k;
pf("%d", i);   5
pf("%d", j);   10
pf("%d", k);   20
pf("%x", &i);  1000
pf("%x", &j);  2000
pf("%x", &k);  3000
pf("%x", &r);  6000
pf("%x", &q);  5000
pf("%d", *p);  5
pf("%d", *q);  10
pf("%d", *r);  20
pf("%x", &p);  4000
pf("%x", p);   1000
pf("%x", q);   2000
```

Pointer to pointer




```
int **pp;
pp = &p;
```

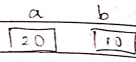
*pp → Value at (pp) = Value at (4000) = 1000

*(*pp) → Value at 1000 = 5

Q. Write a program to swap two integers using a swap function. Take values from the user.

→

```
#include <stdio.h>
void swap (int*, int*);
void main ()
{
  int a, b
  pf("Enter 2 values");
  sf("%d %d", &a, &b);
```




```
  swap(&a, &b);
  pf("After swapping")
  pf("a=%d  b=%d", a, b);
}
void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

temp = *a = Value at 1000

temp = 10

*a = *b

Value at 1000 = Value at 2000 = 20

*b = temp;

Value at 2000 = 10.

Structures.

struct Employee ← real world entity. (OOP in C using structure)

```
{
  int Emp_id;
  float salary;
};  ← indicates end.
```

```
void main ()
{
  struct Employee el;   ← structure variable.
  pf("Enter Emp id & salary");
  sf("%d %f", &Emp_id, &el.salary);
  pf("Employee data");
  pf("Emp-id = %d", el.Emp_id);
  pf("Emp sal = %f", el.salary);
}
```

2024/8/30 08:45

Q. Write a program to take employee id and salary for three employees from the user and print the same.

```
struct Employee
{
  int emp_id;
  float salary;
};

void main ()
{
  struct Employee e1, e2, e3;
  pf("Enter details for 1st employee:");
  sf("%d %f", &e1.emp_id, &e1.salary);
  pf("Enter details for 2nd employee:");
  sf("%d %f", &e2.emp_id, &e2.salary);
  pf("Enter details for 3rd employee:");
  sf("%d %f", &e3.emp_id, &e3.salary);
  pf("Employee data");
  pf("Employee 1: %d %f", e1.emp_id, e1.salary);
  pf("Employee 2: %d %f", e2.emp_id, e2.salary);
  pf("Employee 3: %d %f", e3.emp_id, e3.salary);
```

Q. Write a program to sort the data of 10 students. Take the data from the user. (using array of structure)

```
struct student
{
  int Roll_no;
  char name[10];
  float total_marks;
};
struct student c[10];   // array of structures

void main ()
{
  int i, j;
  pf("Enter data for 10 students");
  for(i=0; i<10; i++)
  {
    scanf("%d", &c[i].Roll_no);
    gets(c[i].name);
    scanf("%d", &c[i].total_marks);
  }

  int temp
  printf("Sorting the data");
  for(i=0; i<10; i++)   Rows
  {
    for(j=0; j<10; j++)   columns
    {
      if(c[j].total_marks < c[j+1].total_marks)
      {
        temp = c[j];
        c[j] = c[j+1];
        c[j+1] = temp;
      }
    }
  }

printf("After sorting");
for(i=0; i<10; i++)
{
  printf("Roll-no. %d", c[i].Roll_no);
puts printf("Name" c[i].name);
  printf("Salary" pf("Total marks %d", c[i].total_marks);
}
}
```

Data (any fact / value / etc.)
Information (When data is organized it becomes information)
Need of Data Structures.

Data Structures

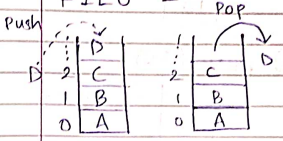
- Primitive (Basic ones like int, float, etc.)
- Non primitive (Made using primitive datatype) eg; Array. int arrray → Primitive, Non primitive
  - Linear
    - Array
    - Linked list (Similar but arrays are static while LL are dynamic)
    - Stack (Plate in a box, first one out; Movie ticket)
    - Queue
  - Nonlinear
    - Tree
    - Graph

Operations on DS

i) Create
ii) Insert
iii) Delete
iv) Search
v) Sort
vi) Traversing / Display. (visiting each element)

1) Stack

Linear DS
LIFO –
FILO –



Push – Insert
Pop – Delete
Peek – Knowing the topmost element

Display / Traverse
↓
For loop (top to bottom)

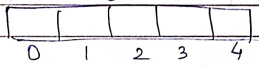
Stack
- Static → Array Representation
- Dynamic → Linked List Representation

Array Representation

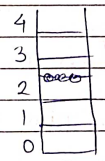
PUSH Operation

1) Initial Scenario

MAX = 5 top = -1
Stack [MAX]



OR

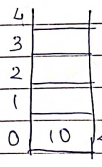


top = -1

Stack Empty if (top == -1)

2) Insert 10

∴ push (10)

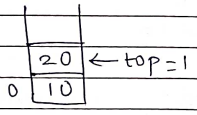


top = 0
top++
st[top] = 10

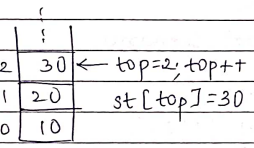
top → -1 before where array starts

3) Push 20

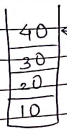
top ++ ; top = 1
st [top] = 20



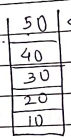
4) push (30)



5) push (40)

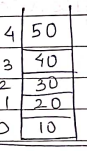


6) push (50)



7) push (60)

MAX = 5



~~stack overflow~~

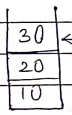
if (top == MAX-1)
↓ after this stack overflow.

Function (for push)

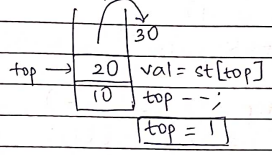
```
void push (int st[], int val)
{
  if (top == MAX-1)        → condition for full; after this happened stack overflows
    pf ("Overflow")
  else                     → Inserting after maxm limit reached
    top++;
    st[top] = val;
}
```

Pop Operation (delete)

① Initially.

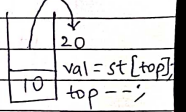


② Pop



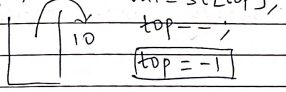
top = 1

③ pop



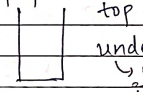
top = 0

④ pop



top = -1

⑤ pop



top == -1
underflow
↳ no element in stack.

Function (for pop)

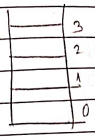
```
int pop (int st[], ~~int val~~)
{
  if (top == -1)
    { pf ("Underflow");
      return -1
    }
  else
    {
      val = st[top];
      top--;
      return val;
    }
}
```

Function (for peek)

```
int peek (int st[])
{
  if (top == -1)
  pf ("Stack is empty");
  else
    return st[top];
}
```

```
void Display (int st[])
{
   int i;
if (top == -1)
   pf ("Stack Empty");
else
for (i=top; i>=0; i--)
   pf ("%d \n", st[i]);
}
```

like this → representation



Q. WAP to perform ~~Array~~ Implementation of Stack.

```
#include <stdio.h>          ← preprocessor directive
#define MAX 3               ← constant global variable.

// declare global variable
int st[MAX];        ⟵ As it's global it will also work if we dont pass it to function.
int top = -1;

// function prototypes      ↓ If passed then also will work.
void push (int st[], int val);
int pop (int st[]);
int peek (int st[]);
void display (int st[]);

void main ()
{
   int val, option
```

```
do
{
   printf ("---- MAIN MENU ----");
   printf ("\n 1. PUSH");
   printf ("\n 2. POP");
   printf ("\n 3. PEEK");
   printf ("\n 4. DISPLAY");
   printf ("\n 5. EXIT");

   pf ("Choice = ?");
   sf ("%d", &option);

switch (option)
{
   case 1:
      pf ("Enter element");
      sf ("%d", &val);
      push (st top
      push (st[], val);
      break;

   case 2:
      val = pop (st[]);
      if (val != -1)
      pf ("Deleted element %d", val);
      break;

   case 3:
      val = peek (st[]);
      if (val != -1)
      pf ("Topmost = %d", val);
      break;
```

```
        case 4:
                display (st)
                break;
    } // Switch
} while (option != 5); }

void push (int st[], int val)
{
    if (top == MAX - 1)
    {
        pf ("\n Stack Overflow");
    }
    else
    {
        top++;
        st[top] = val;
    }
}

int pop (int st[], int val)
{
    if (top == -1)
    {
        pf ("\n Stack underflow");
        return -1
    }
    else
    {
        val = st[top];
        top--;
        return val;
    }
}
```

```
void display (int st[])
{
    int i;
    if (top == -1)
    pf ("\n Stack is empty");
    else
    {
        for (i = top; i >= 0; i--)
        {
            pf ("\n %d", st[i]);
            pf ("\n");
        }
    }
}

int peek (int st[])
{
    if (top == -1)
    {
        pf ("\n Stack is empty");
        return -1;
    }
    else
        return (st[top]);
}
```

Infix to Postfix.

Operators : +, −, *, %, /, ^ (^ associated to E)

Higher precedence

*, %, /, ^ (L → R)

^ : Highest precedence of all.

Associativity of precedence same for left to right whichever comes first.

Lower Precedence

+, −

Manual conversion

1) (A−B) * (C+D)

e (AB−) * (CD+)

AB− CD+ *

2) (A+B) / (C+D) − (D * E)

AB+ / CD+ − DE *

AB+CD+ / − DE *

AB+CD+ / DE * −

3) (A+B ^ C) * D+E ^ F

(A+BC^) * D+E F^

(ABC^+) * D+EF^

ABC^+ D * EF^+

4) a+b * C / D−E

a + bc * / D − E

a + bc * D / − E

abc * D / + E −

a + bc * / D − E

a + bc * D / − E

abc * D / + − E

abc * D / + E −

((A−B) + D / ((E+F) * G))

(AB−) + D / ((EF+G*))

(AB−) + D / (EF+G*))

(AB−) + DEF+G*) /

AB− DEF+G*) / +

((A+B) * C − (D−E)) ^ (F+G)

((AB+) * C − (DE−)) ^ (FG+)

(AB+C* − (DE−)) ^ (FG+)

AB+C* DE− − ^ FG+

AB+C* DE− − FG+ ^

A − (B / C + (D % E * F) / G) * H

A − (B / C + (DE % * E) / G) * H

A − (B / C + DE % F * / G) * H

A − (BC / + DE % F * G /) * H

A − BC / DE % F * G / + * H

A − BC / DE % F * G / + H *

ABC / DE % F * G / + H * −

POP till Stack empty → if element ...

If equal precedence then pop first then push

Using Stack:

~~Rules~~ :-

1) Without brackets -

Scan (L → R)

Rules:

1) Operands - Postfix String (PS)
2) Operators - Stack.
3) If (precedence of i/p char operator) ≤ (Precedence of operator on top) top most element.

then pop

↓

Bada operator hamesha chote k upar rahega

Infix - A | * | B | + | C

SR No.	I/p char	Stack	Postfix String	Remark (Write remark only if you are sure)
1)	A	-(empty)	A	
	*	*	A	
	B	*	AB	
	+	+	AB*	precedence(*) > precedence(+) ∴ pop *; push +
	C	+	AB* C	
			AB* C +	If elements are over pop stack elements until it gets empty.

Infix : A + B * C / D

I/p	Stack	Postfix string
A	- (empty)	A
+	+	A
B	+ (open end of stack ↓)	AB
*	closed end → + | *	AB
C	+, *	ABC
/	+, /	ABC*
D	+, /	ABC*D
↓		ABC*D/
		ABC*D/+

Infix : a + b * c / d - e

I/p	Stack	Postfix String
a	- (empty)	a
+	+	a
b	+	ab
*	+, *	ab
c	+, *	abc
/	+, /	abc*
d	+, /	abc*d
-	-	abc*d/+
e	-	abc*d/+e
		abc*d/+e-

2) With Paranthesis

Rule 1) Push opening paranthesis into the stack
2) Pop all operators till opening paranthesis if you get i/p char as closing paranthesis. And discard opening paranthesis.
3) Rest all remains the same.

eg; (A-B) *(C+D) = AB-CD+*

opening parenthesis is always greater than any operator

Sr. No.	I/P char	Stack	Postfix String	Remarks
1.	(	(	—	
2.	A	(	A	
3.	−	(−	A	
4.	B	(−	AB	
5.	)	(− → pop	AB −	pop until (& discard (
6.	*	*	AB −	
7.	(	*(	AB −	
8.	C	*(	AB − C	
9.	+	*(+	AB − C	
10.	D	*(+	AB − CD	
11.	)	*	AB − CD +	
12.			AB − CD + *	pop until empty

Solve -
1) A+(B* C − (D/E − F) * G) * H
2) (A * B) + (C/D) − (D+E)
3) (A−B) + C * D/E − C

```
#include <stdio.h>
#include <string.h>
#define MAX 50
char st[MAX], top = -1;
void push (char st[], char);
char pop (char st[])
void Infix to Postfix (char source[], char target[]
int main ()
{
    char infix[100], Postfix[100];
    pf ("Enter Infix Exp:")
    gets (infix);
    strcpy (postfix, "");
    pf ("u postfix exp=");
    puts (postfix);
    return 0;
}
```

(A−B) + C*D/E−C
(AB−) + C*D/E−C
(AB−) − CD*/E−C
(AB−) − CD*E/−C
AB−CD*E/+−C
+B−CD*E/+C−

Infix : (A−B) + C * D / E − C

Sr No.	I/p char	Stack	Postfix String	Remarks.
1.	(	(		
2.	A	(	A	
3.	−	(−	A	
4.	B	discard ← (− →pop	AB	
5.	)		AB −	
6.	+	+	AB −	
7.	C	+	AB − C	AB−CD*E
8.	*	+*	AB − C	/+C−
9.	D	+*	AB−CD	
10.	/	+/	AB−CD *	
11.	E	+/	AB−CD*E	
12.	* → −	−	AB−CD*E/+	
13.	C	−	AB−CD*E/+C	

2024/8/30 08:45

Infix : (A * B) + (C / D) − (D + E)

SR. No.	I/p char	Stack	Postfix string	Remarks
1.	(	(	−	
2.	A	(	A	
3.	*	(*	A	
4.	B	(*	AB	
5.	)	discard (*) pop	AB*	
6.	+	+	AB*	
7.	(	+(	AB*	
8.	C	+(	AB*C	
9.	/	+(/	AB*C	
10.	D	+(/	AB*CD	
11.	)	+	AB*CD/	
12.	−	−	AB*CD/+	
13.	(	−(	AB*CD/+	
14.	D	−(	AB*CD/+D	
15.	+	−(+	AB*CD/+D	
16.	E	−(+	AB*CD/+DE	
17.	)	−	AB*CD/+DE+	
			AB*CD/+DE+−	

University Q.

Source − stack
target → PS

Infix : A + (B * C − (D/E − F) * G) * H

SR. No.	I/p Char	Stack	Postfix string
1.	A		A
2.	+	+	A
3.	(	+(	A
4.	B	+(	AB
5.	*	+(*	AB
6.	C	+(*	ABC
7.	−	+(−	ABC*
8.	(	+(−(	ABC*
9.	D	+(−(	ABC*D
10.	/	+(−(/	ABC*D
11.	E	+(−(/	ABC*DE
12.	−	+(−(−	ABC*DE/
13.	F	+(−(−	ABC*DE/F
14.	)	+(−	ABC*DE/F−
15.	*	+(−*	ABC*DE/F−
16.	G	+(−*	ABC*DE/F−G
17.	)	+	ABC*DE/F−G*−
18.	*	+*	ABC*DE/F−G*−
19.	H	+*	ABC*DE/F−G*−H
			ABC*DE/F−G*−H*+

* Program for ~~Postfix~~ Infix to Postfix

```
#include <stdio.h>
#include <string.h>
# define MAX 50
char st[MAX], top = -1;
void push (char st[], char);
char pop (char st[]);
void Infix to Postfix (char source[], char target[]);
int main(){char infix[100], postfix[100];
printf ("Enter Infix Expression ");
gets (infix);
strcpy (postfix, "")
Infix to Postfix (infix, postfix);
printf ("Postfix expression = ");
puts (postfix);
return 0;
}

void Infix to Postfix (char source[], char target[]){
{
int i = 0, j = 0;
char temp;
strcpy (target, "");

while (source[i] != '\0')
{
// till the input expression ends

Case 1: // Opening bracket
if (source[i] == '(')
{
push (st, source[i]);
i++;
}
else if (source[i] == ')')

Case 2: // closing bracket else if (source[i] == ')')
{ while ((top != -1) && (st[top] != '('))
{
target[j] = pop(st);
j++;
}

if (top == -1)  // closing bracket does not get opening bracket eg; ((a+b))
{
printf ("Incorrect Expression");
exit(1);
}
// remove left paranthesis
temp = pop(st);
i++;
} // else if

Case 3: if char is alphabet or digit
else if (isdigit(source[i]) || isalpha(source[i]))
{
target[j] = source[i];
j++;
}
i++;
```

Case 4: Operators

```
else if ( source[i] == '+' || source[i] == '-'
        || source[i] == '*' || source[i] == '/'
        || source[i] == '%' )
{
  while ( (top != -1) && (st[top] != '(')
  && getPriority (st[top]) >= getPriority (source[i])
  {
      target[j] = pop(st);
      j++;
  }
  push ( st, source[i] );
  i++;
}
else
{
   pf ( "Incorrect element" );
   exit(1);
}

// Empty stack (if ip char is ), $, etc.)
while ( (top != -1) && st[top] != '(' )
{
   target[j] = pop(st);
   j++;
}

   target[j] = '\0'
} // function ends.

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

```
char pop (char st[], char x)
{
   if (top == -1)
   {
      pf ("\n Stack underflow");
      return -1;
   }
   else
   {
      x = st[top];
      top--;
      return x;
   }
}

void push (char st[], char x)
{
   if top
```

```
void push (char st[], char val)
{
   if (top == MAX-1)
      pf ("Stack Overflow");
   else
   {
      top++;
      st[top] = val;
   }
}
```

```
char pop (char st[])
{
    char val = ' ';
    if top (== -1)
    { pf ("stack underflow");
    else
    {
        val = st[top];
        top--
    }
    return val;
}
```

Postfix Evaluation

→ Scanning : L → R

→ If (char == operand)
push

→ if (char == operator)
{
✓ pop last two operands
✓ apply operator & calculate result
✓ push result in the stack.

→ set final result = st[top]

String :- 934 * 8 + 4 / -

I/p char	Stack	Result
9	9	
3	9, 3	
4	9, 3, 4 (operand 1, operand 2)	3 * 4 = 12
*	9, 12	
8	9, 12, 8	
+	9, 20	operand 2 = 8, operand 1 = 12, 12 + 8 = 20
4	9, 20, 4	
/	9, 5	~~4 / 20~~ 20/4 = 5
-	4	9 - 5 = 4

2) 653 +9 * + = 78 = 78

3) 562 + * 124 / − = 37

This is 1 char → only consider. in real, only one character is considered

I/p char	stack	Result.
6	6	
5	6,5	op1
3	6,5,3 → op 2	5+3 = 8
+	6,8	
9	6,8,9	8*9 = 72
*	6,72	6+72 = 78
+	78	

PS = 562 + * 124 / −

I/p char	Stack	Result.
5	5	
6	5, 6	
2	5, 6, 2	
+	5, 8	6+2 = 8
*	40	5*8 = 40
12	40, 12	
4	40, 12, 4	
/	40, 3	12/4 = 3
−	37	40−3 = 37.

99.99% may come → This one and the previous program.

Q. → WAP for Postfix Evaluation.

```
#include <stdio.h>
#define MAX 100
float st[MAX];
int top = -1;
void push (float st[], float val);
float pop (float st[]);

float evaluate_PS (char exp[]);

void main ()
{
    pf ("Enter expression");
    gets (exp);
float val = evaluate_PS (exp)
  pf ("value = %f", val);

float Evaluate_PS (char exp[])
 {
  int i=0;
  float op1, op2, value;
while (exp[i] != '\0')
{
  if (isdigit (exp[i]))
    {
      push (st, (float) (exp[i] - '0'));
    }
else
 {
      op 2 = pop (st);
      op 1 = pop (st);
```

```
switch (exp[i])
{
  case '+' :
            value = op1 + op2;
            break;
  case '-' :
            value = op1 - op2;
            break;
  case '*' :
            value = op1 * op2;
            break;
  case '/' :
            value = op1 / op2;
            break;
  case '%' :
            value = op1 % op2;
            break;
  default : pf ("Invalid");
  } // switch

  push (st, value);
} // else

  i++;
} // while

  return (pop(st));
}

void push (float st[], float val)
{
  if (top == MAX-1)
    pf("\n Stack overflow");
else
    {  top++;
       st[top] = val;
    }
}

float pop (float st[])
{
    float val = -1;
  if (top == -1)
     pf ("Stack Underflow);
else
    val = st[top];
    top--;
  }
   return val;
}
```

Recursion

Do write eg. in difference.

Iteration	Recursion
Top down approach	Has base recursive case
Stack not involved	Used when same task repeated
Normal for loop	Top-down then down-top
	stack involved
	No for loop

1) Difference between iteration & Recursion.
2) Define a function that calls itself to solve the smaller version of its task.
3) Base case - stopping condition
Recursive case - fuction call to itself.

(Required for recursive function and not for other function)

eg; factorial of 5

5! = 5 × 4 × 3 × 2 × 1
≡ 5! = 5 × 4!
= 4 × 3!
= 3 × 2!
= 2 × 1!
= 1 × 0!
0! = 1

Recursive Function

```
int fact (int n)
{
   if (n==0)
      return 1;
   else
      return (n * fact (n-1))
```

First the elements are pushed one by one into the stack and then the result is popped out till the final result comes.

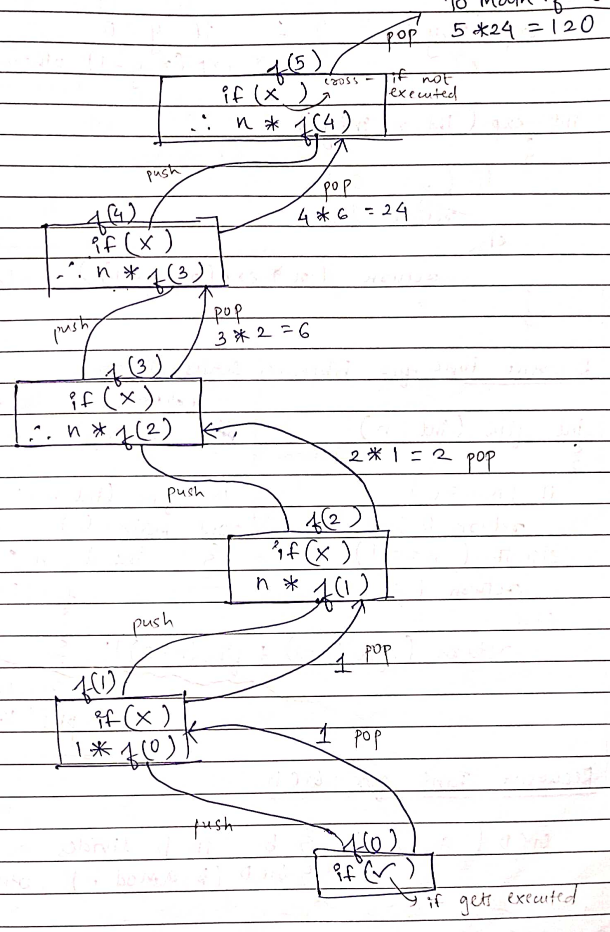
↓ winding ↓ push

pop unwinding aka (backtracking) ↑

in viva → winding → push
unwinding → pop → result

Dry run ↓ Paper and pen execution

Application of stack in Recursion -



Recursive Func for Exponent

$$Exp(x,y) = \begin{cases} 1 & \text{if } y = 0 \quad \text{Base case, stopping cond.} \\ x * exp(x, y-1) & \text{otherwise} \quad \text{Recursive case.} \end{cases}$$

x^y

$2^{10} = 2 \times 2^9$

```
int exp (int x, int y)
{
    if (y == 0)
        return 1;
    else
        return (x * exp(x, y-1))
}
```

Recursive Func for Fibonacci Series - nth term

```
int fibo (int n)        // n → nth term
{
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return (fibo(n-1) + fibo(n-2));
}
```

If all terms are also to be printed then use for loop -

```
int fibo (int n);
void main ()
{   int i, n;          // n → no. of terms
    int result;
    for(i=0; i<n; i++)
    {
      result = fibo(i);
      pf("%d", result);
```

Imp for comp exam

Recursive Func for GCD

$$GCD(a,b) = \begin{cases} b & \text{if } b \text{ divides } a \quad \text{Base case, stopping condition} \\ GCD(b, a \bmod b) & \text{otherwise} \quad \text{Recursive case} \end{cases}$$

Here $a > b$

If $a < b$ then interchange a and b in formula

```
int GCD (int a, int b)
{
    if (a % b == 0)
        return b;
    else
        return (b, a % b);
}
```

Imp for MT

WAP to check nesting of Parantheis

to check expression valid or not

```
#include <stdio.h>
#include <string.h>
#define MAX 10
int top = -1;
char st [MAX];
void push (char);
char pop ();
void main ()
{
    char exp [MAX];
    char temp;
    int i, flag = 1;        // we consider exp is OK
    pf ("Enter expression");
    gets (exp);
    for (i = 0; i < strlen (exp); i++)
    {
```

For diff in topmost element

valid exp if flag = 1; else 0.

/* Case 1: Opening bracket we are considering all type of brackets */

Infix → parenthesis present
Prefix/Postfix → No parenthesis

```
if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
        push (exp[i]);

/* case 2: Closing brackets */
if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}')
{
 * We have closing bracket but stack is empty
   i.e closing bracket but no opening bracket */
if (top == -1)
        flag = 0;
        break; its or our wish to break
else
 {
        // pop topmost element
        temp = pop ( );

// check with the i/p char
if ((exp[i] == ')') && (temp == '{' || temp == '['))
        flag = 0;
        break;
if ((exp[i] == ']') && (temp == '(' || temp == '{'))
        flag = 0;
        break;
if ((exp[i] == '}') && (temp == '[' || temp == '('))
        flag = 0;
        break;
```

A + { Deleting after top = -1 is underflow
Inserting after top = Max - 1 is overflow

```
} // else
} // for
// case 3: Mismatch in no. of paranthesis
    if (top >= 0)
        flag = 0;
if (flag == 1)
    pf ("Valid");
else
    pf ("Not valid");
} // main
```

Imp Question

Difference Stack and Array.

1) Explain Stack Overflow and Underflow. (Explain with eg:)
2) Explain difference between Array and Stack.
3) Differentiate btwn working of Stack using LL and Stack using array.
4) Difference between peek and pop function. (write in pa
5) Convert foll. exp from Infix to Postfix.
6) Evaluate Postfix Exp.
7) Differentiate btwn iterative and recursive function. Which one would you use in which condition.
8) Write a function that accepts 2 stacks. Copy the contents of first stack into 2nd stack. Order of elements should be preserved.

Queues

→ linear DS — In Non-linear we have branching structures.
→ FIFO
→ Front end & rear end

→ Elements added from rear end
→ Elements deleted from front end.

Static → Array
Dynamic → Linked list

Array Implementation of Queue of linear q.

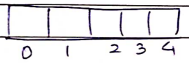
→ Variables used Stack → Top
Queue → front, rear
f r

Operations

1) enqueue → Insert
2) Dequeue → Delete
3) Traversing → Display
4) Peek → knowing the frontmost element.

1) Enqueue

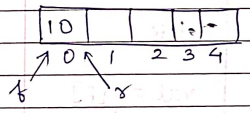
MAX = 5 q[MAX]
f = r = -1



∴ 1st empty condition
f = r = -1

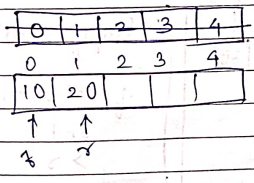
2) Insert 10

1st element as
f = r = -1
∴ f = r = 0
q[r] = 10



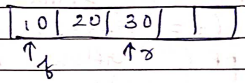
3) Insert 20

r++ , q[r] = 20

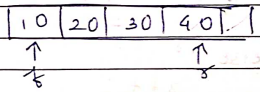


4) Insert 30

r++ ; q[r] = 30

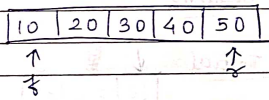


5) Insert 40



r++ q[r] = 40

6) Insert 50



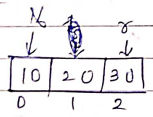
r = MAX - 1 (full condition)

After step 6, if we attempt Insert 60
∴ OVERFLOW
(Insertion after r = MAX - 1)

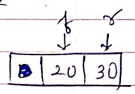
```
void Enqueue ()
{
  int num;
pf ("Enter val");
sf ("%d", &num);
if (r == MAX-1)
  pf ("Overflow");
    exit (1);
else if (f==-1 && r==-1)
    f=r=0;
else
    r++;
  q[r] = num;
}
```

Dequeue

Initial



Delete
val = q[f] = 10
f++;

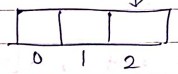


Delete
val = q[f] = 20
f++;

Condition for 1 element
f==r && f&r != -1

Delete

val = q[f]
f++; ∴ f > r



∴ Empty condition,
f = r = -1
and
f > r

Display → Always front to rear. Not 0 to rear

Traverse empty space then we can't access.

```
int delete ()
{
  int val;
if (f==-1 || f>r)
{
  pf ("Underflow"); f > r
    return -1;
}
else
  {
    val = q[f];
    f++; }

if (front > rear)
    f=r=-1
```

for further insertions.

→ This happens after full queue dequeued and dequeued. and then if insertion is tried to be made.

```
  return val;
}
```

```
int peek ()
{
  if (f==-1 || f>r)
  {
    pf("Q is empty");
      return -1;
  }
else
    return (q[f]);
}
```

```
void display ()
{
   int i;
if (f==-1 || f>r)
   pf ("Q is empty");
else
   for (i=f; i<=r; i++)
      pf ("%d \t", q[i]);
}
```

Q. WAP to perform Queue Operations (Linear Q)

```
#include <stdio.h>
#define MAX 10
int q[MAX]
int f=r=-1
void enqueue ()
int delete ()
int peek ()
void display ()
void main ()
{
   do {
      printf ("---- MENU ----")
      printf ("1. Insert");
      printf ("2. Delete");
      printf ("3. Peek");   → No peek operation
      printf ("4. Display the queue");
      printf ("5. Exit");
      printf ("Enter your option");
      scanf ("%d", &option);
```

```
switch (option)
{
 case 1: enqueue ();
         break;

 case 2:
         val = delete ()
         if (val != -1)
         pf ("Delete in Q = %d", val);

 case 3:
         val = peek ();
         if (val != -1)
         pf ("1st element in Q = %d", val);

 case 4:
         display ();
         break;
}
} while (option != 5);
```

Here write functions of enqueue, delete, peek and display.

to be updated again and again

Date:

Circular Queue

→ The first index comes right after the last index.

1) Initially

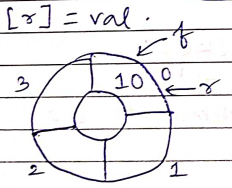
MAX = 4 , q[MAX]

f = -1 , r = -1 — already initialized

f = r = 0

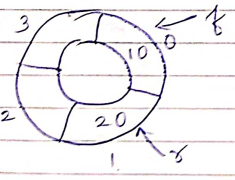
Insert 10

q[r] = val.



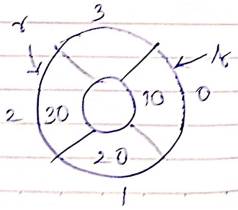
Insert 20

r++ , q[r] = 20



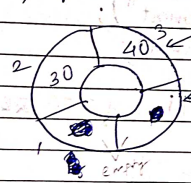
Insert 30

r++ , q[r] = 30



Insert 40

r++ , q[r] = 40

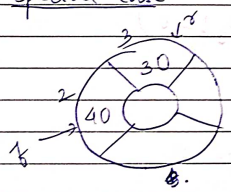


∴ r = MAX − 1 and f = 0

1st condition

r = MAX − 1 for overflow along with f == 0

Special Case

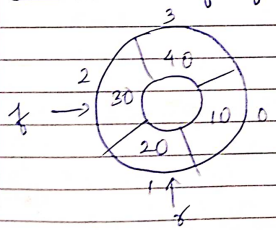


r = MAX − 1 & f != 0

r = 0

∴ q[r] = 100.

2nd condition for full



r = f − 1

Indices problem
↓
Queue.

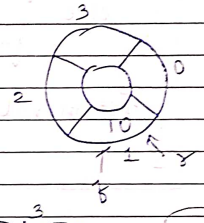
```
void Insert()
{ int num;
  pf("Enter Element");
  sf("%d", &num);
  if (r==f-1) || (f==0 && r=MAX-1)
      pf("Overflow");
  else if (f==-1 & r==-1)
      {
          f=r=0;
          q[r] = num;
      }
  else if (r=MAX-1 && f!=0)
      {
          r=0;
          q[r] = num;
      }
  else
      { r++;
        q[r] = num;
      }
}
```

Delete Operation -

i) Underflow / Empty

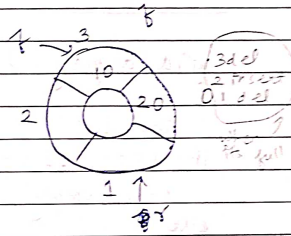
$f = r = -1$

Case 1 - Only 1 element



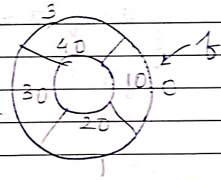

```
f = r
val = q[f]
f=r=-1
```

Case 2 - Special Case




```
f = MAX-1
∴ val = q[f]
f = 0
```

Case 3 - Normal Deletion.

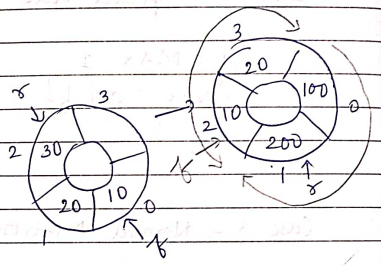



```
val = q[f]
f++;
return val;
```

```
int delete()
{
   int val;
if (f==-1 & r==-1)
{
   pf("Underflow")
   return -1
}
```

```
val = q[f];
if (f == r)
    f = r = -1
else
{ if (f == MAX-1)
    f = 0;
  else
    f++;
}
return val;
}
```

```
void display()
{
int i;
if (f == -1 && r == -1)
  pf ("Empty");
else
  if (front <= rear)
{
  for (i = f; i <= r; i++)
    pf("%d", q[i]);
}
else
  {
    for (i=f; i<MAX; i++)
      pf ("%d", q[i];
    for (i=0, i<=r; i++)
      pf("%d", q[i]);
  }
```



Q. WAP to implement a circular queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front = -1, rear = -1;
void insert();
int delete();
int peek();
void display()
int main()
{
   int option, val;
   clrscr();
   do
   {
      printf("Main Menu");
      printf("\n 1. Insert an Element");
      printf("\n 2. Delete an element");
      printf("\n 3. Peek");
      printf("\n 4. Display the queue");
      printf("\n 5. Exit");

   printf("Enter your option");
   scanf("%d", &option);
switch (option)
{
     case 1:
          insert();
          break;
     case 2:
          val = delete_element();
```

```
        if (val != -1)
            printf ("\nNumber deleted is : %d", val);
            break;
    case 3:
            val = peek ();
            if (val != -1)
                printf ("\n The first value in queue is
                        %d", val);
                break;
    case 4:
                display ();
                break;
        }
}while (option != 5);
getch();
return 0;
}

void insert ()
{
    int num;
    pf ("Enter number to be inserted");
    scanf ("%d", &num);
    if (front == 0 && rear == MAX-1)
        printf ("\n Overflow");
    else if (front = -1 && rear = MAX-1)
    {
        front = rear = 0;
        queue[rear] = num;
    }
    else if (rear = MAX-1 && front != 0)
    {
        rear = 0;
        queue[rear] = num;
    }
    else
    {
        rear++;
        queue[rear] = num;
    }
}

int ~~dele~~ delete_element ()
{
    int val;
    if (front == -1 && rear == -1)
    {
        printf ("\n Underflow");
        return -1;
    }
    val = queue[front];
    if (front == rear)
        front = rear = -1;
    else
    { if (front = MAX-1)
        front = 0;
      else
        front ++;
    }
    return val;
}

int peek ()
{
    if (front = -1 && rear = -1)
    { printf ("\n Queue is Empty");
```

```
        return -1;
    }
    else { return queue[front]; }
}

void display()
{   int i;
    printf("\n");
    if (front == -1 && rear == -1)
        printf("\n Queue is Empty");
    else
    {
        if (front < rear)
        {
            for (i = front; i <= rear; i++)
                printf(" %d", queue[i]);
        }
        else
        {
            for (i = front; i < MAX; i++)
                printf(" %d", queue[i]);

            for (i = 0; i <= rear; i++)
                printf(" %d", queue[i]);
        }
    }
}
```

Circular Q Updated

Insertion / Enqueue Operation –

```
if (rear + 1) % MAX == front
    pf ("Overflow");
if (front == -1 & rear == -1
    set front = rear = 0
else
    if rear == MAX-1 & front != 0
        rear = 0
    else
        rear = (rear + 1) % MAX
        set q[rear] = val
```

Deque

Double Ended Queue.

- pronounced as 'deck'
- Insertion & Deletion from either ends
- Input restricted Q

↓

1) Insert from the rear
2) Deletion from both ends

- Output restricted Q.

↓

i) Insertion from both ends
ii) Deletion from front end

rear → right	Insert R()	✓ Already done
front → left	delete L()	✓ usual way

Insert L() & ~~date~~ delete R()

Usually Deck not Dmp

Dequeue
↓
Double ended queue

Dequeue
↓
Delete

```
void InsertL()
{
  int val;
  pf("Enter value");
  sf("%d", &val);
  if (left==0 && right==MAX-1) || (left==right+1)
  {
    pf("Overflow");
    exit(1);
  }
  if (left==-1)  // Q is empty
  {
    left = right = 0;
  }
  else {
    if (left==0)
      left = MAX-1;
    else
      left = left-1;
  }
  deque[left] = val;
}
```

left ↓ — right

10	20	30	40	30
0	1	2	3	4

10	50	40	30	20	← l
0	1	2	3	4	

l = r

10	50	40	30	20
0	1	2	3	4

Insert

right pointer used to delete

```
void deleteR()
{
  if (left==-1)
  {
    pf("Underflow");
    exit(1);
  }
  pf("Deleted ele=%d", q[right]);
  if (left==right)  /* Only one element */
  {
    l = r = -1
  }
  else
  {
    if (right==0)
      right = MAX-1
    else
      right = right-1;
  }
```

left = -1

0	1	2	3	4

r

10				

One element

10	20	30	40	50

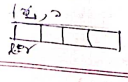
r

300	20	30	40	50

~~10~~	~~20~~	~~30~~	~~40~~

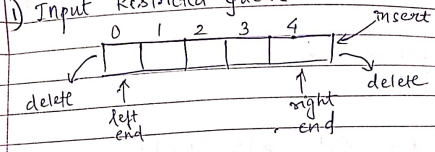
1) Priority Queue → Only for Theory
2) ~~Applications~~ of Queue.

delete from right →



Deque

1) Input Restricted queue



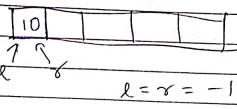
Operations on IR Deque

1) Insert R ( ) → same as ~~CQ~~ CQ.
2) delete L ( ) → Same as LQ
3) delete R ( ) (*)

```
void ~~dot~~ delete_right ( )
{
    if ( left == -1)
    {
        pf ("Underflow") ;
        exit ( );
    }

pf ( "Deleted element = %d", d[right] );
if (left == right) // one element
    left = right = -1
else
    {
        if (right == 0)
            right = MAX-1;
        else
            right --;
    }
```

(1) (2) Applications of Deque → Google, GPT
Given function from Insert L ( ), delete R ( ). program on deck
In phone and in machine for processes. No changes.

Case 1: Only one element



l = r = -1

Burst time → Time required to complete task

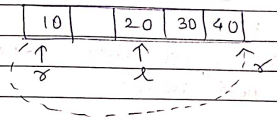
Throughput → How many work done in unit time

SJF → Shortest job first

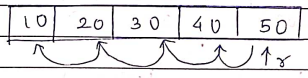
Priority Queue → Rearranging of elements. (Huffman coding)

Output restricted.

Case 2: (A particular case)

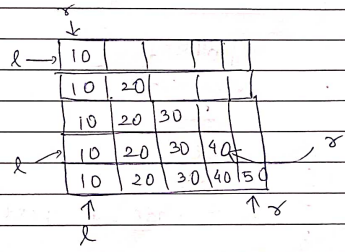


Case 3: Normal deletion from right.

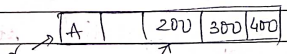


Sequence of Operations

1) Insert R (10)
(20)
(30)
(40)
(50)



delete R( )
delete R( )
delete R( )
delete R( )
Insert R(100)
Insert R(200)
delete L ( )



delete R ( )

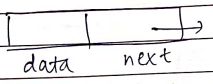
start/head → same.
-1 → End of LL

~~Ex 5~~

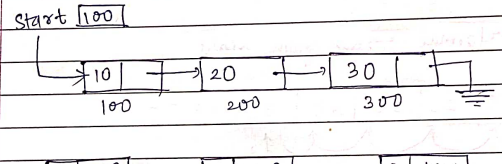
3) Linked list

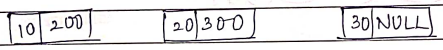
→ Linear collection of data
→ Data element – nodes
→ Sequence of nodes

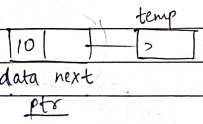
→ Node



→ Data :- any int, float, char data
→ Next :- add of next node.







ptr → data = 10
ptr → next = temp

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

⇒ Self - Referential structure.

Stack → top = -1
queue → front & rear = -1
LL → start ~~FRONT~~ *

Struct node * start

start = NULL

Linked list is Empty.

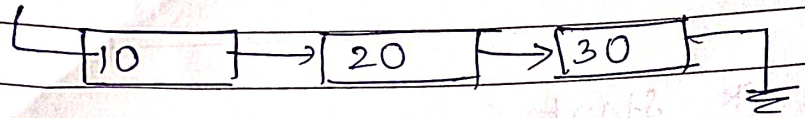
LL in Memory

start	data	next
→ 0	4	4
1		
2	0	-1
3		
4	E	7
5		
6		
7	L	8
8	L	2

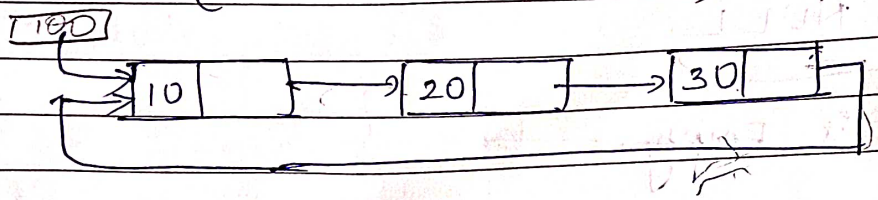
Page No.:
Date:

Types of LL

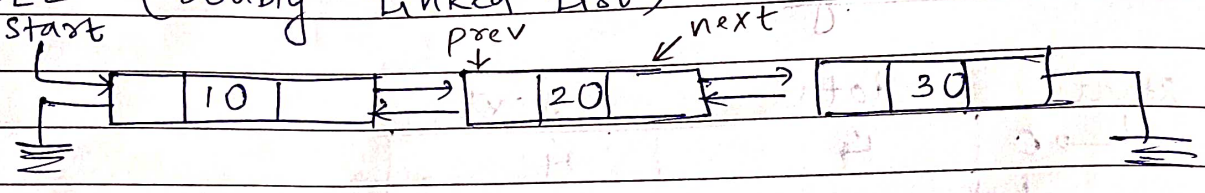
SLL (Simple Linked List)



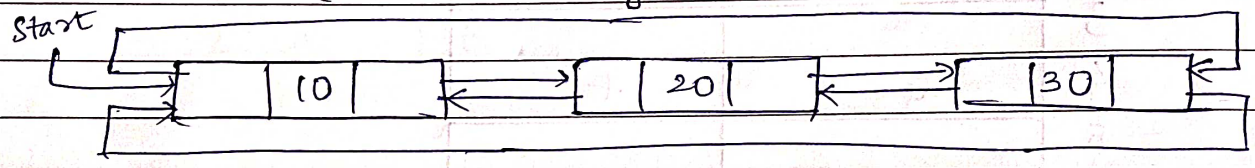
CLL (Circular Linked List)



DLL (Doubly Linked List)



C-DLL (Circular Doubly Linked List)



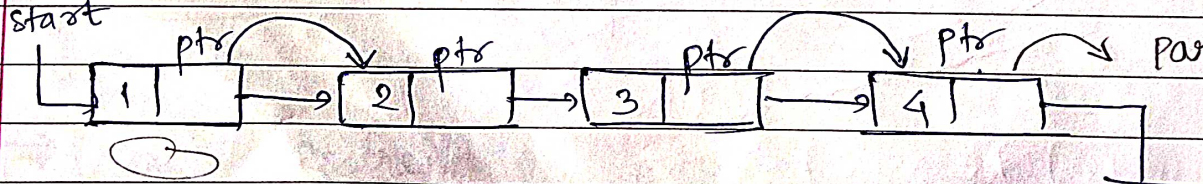
Singly LL

sl → name = "abc";

ptr = ptr → next

✓ Simplest

* Traversing (Display)



pf("%d", ptr → data);

2024/8/30 08:49

Counting no. of nodes

```
struct node *count (struct node * start)
{
    struct node *ptr;
    int count=0;
    ptr = start;
    while (ptr != NULL)
    {
        c++;
        printf ("%d", ptr -> data);
        ptr = ptr -> next;
    }
    pf ("%d", c);
    return start;
}
```

Searching for a given node -

```
struct node * search (struct node * start, int val)
{
    int c = 0;
    struct node *ptr;
    ptr = start;
    while (ptr != NULL)
    {
        c++;
        if (ptr -> data == val)
        {
            pf ("val found at %d", c);
            break;
        }
        else { ptr = ptr -> next; }
    }
    return start; {
```

Insert

i) beg ii) End iii) Before any node iv) After any node

Allocate memory dynamically in C
↳ malloc

Syntax -
ptr = (cast type *) malloc (size)

i) Insert at the beginning

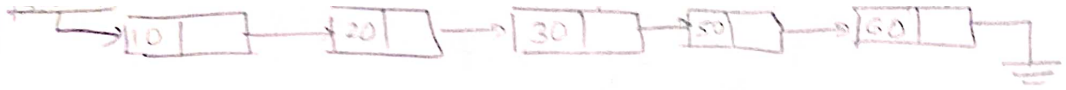
```
struct node * insert beg (struct node * start)
{
    struct node * nn;        // new node
    int num;
    pf ("Enter data");
    sf ("%d", &num);
    nn = (struct node *) malloc (size of (struct node));
    nn → data = num;
    nn → next = start;
    start = nn;
    return start;
}
```

2 (can't write reverse as it would become a circular LL

ii) Inserting at end

```
struct node * insert end (struct node * start)
{
    struct node * nn;
    int num;
    pf ("Enter data");
    sf ("%d", &num);
nn = (struct node *) malloc (size of (struct node));
    nn → data = num;
    nn → next = NULL;
    ptr = start;
```

2024/8/30 08:49



Page No.:
Date:

```
while (ptr → next != NULL)
  {
    ptr = ptr → next;
  };
    ptr → next = nn;
   return start;
}
```

Insert before : (any node) → temp humesh ptr k piche in before and after a bhagne wale - ptr and temp

```
struct node * insert_before (struct node * start)
{
   struct node *ptr, *nn, * temp;
  int num, val;
  pf("Enter data");
  sf("%d", &num);
  pf("Enter value before which we need to insert");
  sf("%d", &val);
 nn = (struct node *) malloc (size of (struct node));
  nn → data = num;
  ptr = start
  while (ptr → data != val)
  {                          → temp (for inserting after)
    temp = ptr;
    ptr = ptr → next;
  }
  nn → next = ptr;
  temp → next = nn;
  return start;
}
```

Reversing it won't make any difference

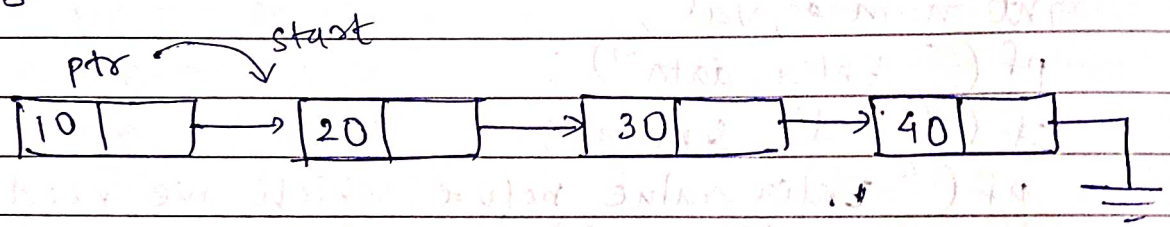
sert
r if
k jgh
ap
heck once

ptr != NULL
ptr → next != NULL

Deleting a Node

1) Delete from beg -

```
struct node * delete_beg (struct node * start)
{
    struct node * ptr;
    ptr = start;
    start = start → next;
    free (ptr);
    return start;
}
```



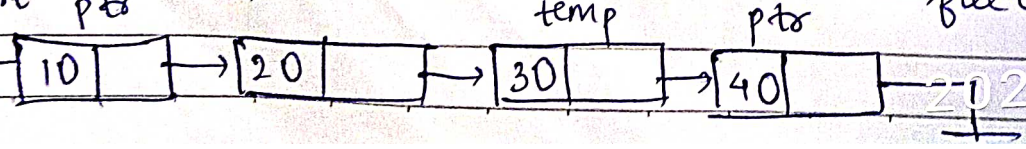
2) Delete from End -

```
struct node * delete_end (struct node * start)
{
    struct node * ptr, * temp;
    ptr = start;
    while (ptr → next != NULL)
    {
        temp = ptr;
        ptr = ptr → next;
    }
    temp → next = NULL;
    free (ptr);
    return start;
}
```

30 → next = NULL

temp → next = NULL

free (ptr)

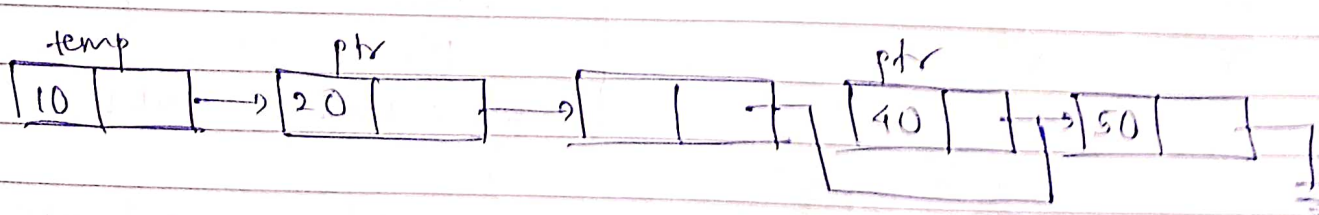


2024/8/3

3) Delete a specified node -

```
struct node * delete_node (struct node * start)
{
    struct node *ptr, *temp;
    int val;
    pf ("Enter the data to be deleted");
    sf ("%d", &val);
    ptr = start;
    if (ptr -> data == val)
    {
        start = delete_beg (start);
        return start;
    }
    else
    {
        while (ptr -> data != val)
        {
            temp = ptr;
            ptr = ptr -> next;
        }
        temp -> next = ptr -> next;
        free (ptr);
        return start;
    } // else
```

while (ptr->data == val)
temp->next = ptr->next;



4) Delete after a given node

```
struct node * delete_after (struct node * start)
{
    struct node * ptr, *temp;
int *p;
pf("Enter val after which node is to be deleted");
sf("%d", &val);
ptr = start;
temp = start;
while (temp -> data != val)
  {
    temp = ptr;
    ptr = ptr -> next;
  }
temp -> next = ptr -> next;
free(ptr);
return start;
}
```

Struct student *s1

Delete
ptr = temp (initialize temp and ptr to start)
after = ptr
before -> ptr

Insert
after = temp
before = ptr

Display the entire list

```
struct node * display (struct node *start)
{
  struct node * ptr;
    ptr = start;
  while (ptr != NULL)
  {
    pf("\t %d", ptr -> data);
    ptr = ptr -> next;
  }
return start;
}
```

In last fun, start = nn not possible else same.

!cmp → data != val
↘ for ~~delete~~ isert_after

Page No.:

Single LL != NULL

In circular != start

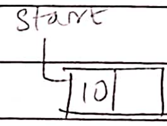
Create a linked list

~~Inserted before~~

Created before →

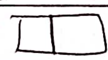
```
struct node * create_LL (struct node * start)
{
    struct node * nn;
    int num;
    pf ("Enter -1 to End");
    pf (" Enter the data");
    sf ("%d", & num);
    while (num != -1)
    {
        nn = (struct node *) malloc (size of
                (struct node));
        nn → data = num;      // very 1st node
        if (start == NULL)
        {   nn → next = NULL;
            start = nn;
        }
        else
        {   nn → next = start;
            start = nn;
        }
        pf (" Enter data");
        sf ("%d", & num);
    }
    return start;
}
```

Reverse not allowed



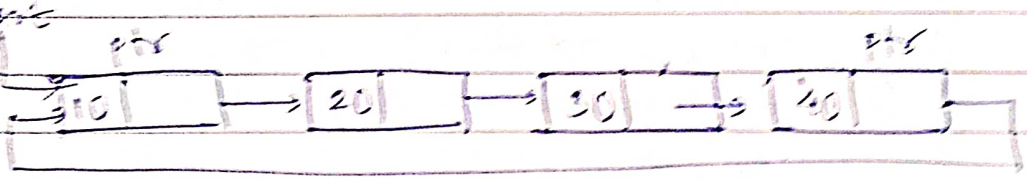
~~Ins~~ Either 1st node is created and after that nodes are added before in the sequence.

created before



Circular linked list

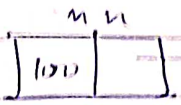
```
struct node * display (struct node * last)
{
    struct node * ptr;
    ptr = last;
    while ( ptr -> next != last)
    {
        printf (" %d ", ptr -> data);
        ptr = ptr -> next;
    }
    printf (" %d ", ptr -> data);  // last node
    return last;
}
```

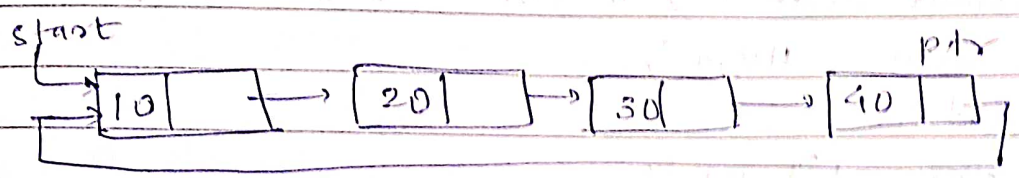


```
struct node * insert_beg (struct node * last)
{
    struct node * nn, * ptr;
    int num;
    printf (" Enter data to insert");
    scanf (" %d ", &num);
    ptr = last;
    nn = (struct node *) malloc (sizeof (struct node));
```

```
nn -> data = num;
while (ptr -> next != start)
    { ptr = ptr -> next; } // while.
    ptr -> next = nn;
    nn -> next = start;
    start = nn; // first node assigned to nn
    return start;
}
```

```
40 -> next = nn;
nn -> next = start;
start = nn;
```





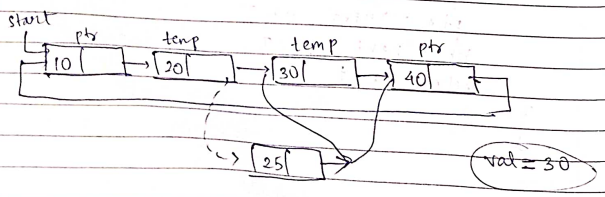

```
struct node * insert_end ( struct node * start )
{
    struct node * nn , * ptr , * temp ;
    int num ;
    pf ( " Enter data in num " );
    sf ( " %d ", & num );
    ptr = start ;
nn = (struct node *) malloc (size of (struct node));
nn -> data = num ;
while (ptr -> next != start)
    {
        ptr = ptr -> next ; }
    ptr -> next = nn ;
    nn -> next = start ;
    return start ;
}
```

Only start = nn; not present from insert_beg func else all same.

insert_before : reverse node

```
struct node * insert_before (struct node * start)
{
    struct node * nn, * ptr, * temp;
    int num, val;
    pf ("Enter data");
    sf ("%d", &num);
    pf ("Enter node b4 ....");
    sf ("%d", &val);
    nn = (struct node *) malloc (sizeof (struct node));
    nn -> data = num;
    ptr = start;
    // b4 the 1st node
    if (ptr -> data == val)
        start = insert_beg (start);
    else
    {
        while (ptr -> data != val)
        {
            temp = ptr;
            ptr = ptr -> next;
        }
        nn -> next = ptr;
        temp -> next = nn;
    }
    return start;
}
```

will do if up nd down




```
struct node * insert_after (struct node * start)
{
    struct node * nn, * ptr, * temp;
    int num, val;
    pf ("Enter data");
    sf ("%d", &num);
    pf ("Enter node after ....");
    sf ("%d", &val);
    nn = (struct node *) malloc (size of (struct node));
    nn -> data = num;
    ptr = start;
    No cond^n for after the 1st node
    while (temp -> data != val)
    {
        temp = ptr;
        ptr = ptr -> next;
    }
    nn -> next = ptr;
    temp -> next = nn;
    return start;
}
```

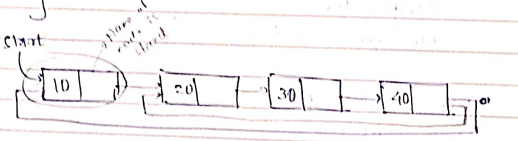
Delete Specified → LL
Insert_before → Circular LL

Circular linked list

Delete -

1)

```
struct node * delete_beg (struct node * start)
{
    struct node *ptr;
    ptr = start;
    while (ptr -> next != start)
        ptr = ptr -> next;
    ptr -> next = start -> next;
    free (start);
    start = ptr -> next;
    return start;
}
```

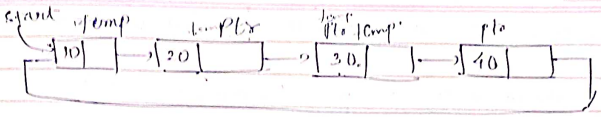


2)

```
struct node * delete_end ( struct node * start)
{
    struct node *ptr, *temp;
    ptr = start;
    while (ptr -> next != start)
    {
        temp = ptr;
        ptr = ptr -> next;
    }
    temp -> next = ptr -> next; OR temp -> next = start;
    free (ptr);
    return start;
}
```

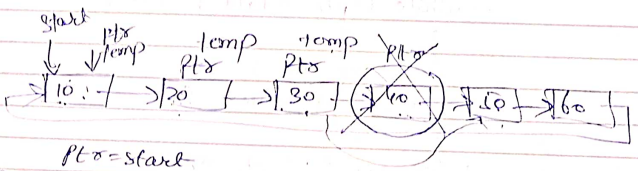
val = 30

ptr -> next -> data != val .



3)

```
struct node * delete_node ( struct node *)
{
    struct node
```



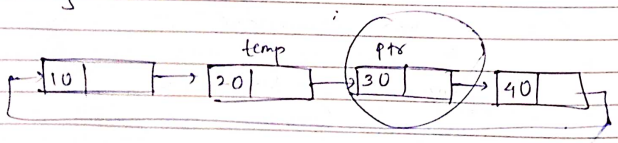
delete before →

```
ptr = start
while (ptr -> next -> data != val)
{
    temp = ptr;
    ptr = ptr -> next;
}
temp -> next = ptr -> next;
    free (ptr);
return start;
```

val = 30

In delete after ~~first~~ ~~prev~~ ~~start~~ ~~to~~ jo delete karna h udhr tk ptr jayga

3)
```
struct node delete_node (struct node * start)
{
  struct node * ptr, * temp;
  int val;
  pf ("Enter data to be deleted");
  sf ("%d", & val);
  ptr = start;
  if (ptr -> data == val)
  {
    start = delete_beg (start);
    return start;
  }
  else
  {
    while (ptr -> data != val)
    {
      temp = ptr;
      ptr = ptr -> next;
    }
    temp -> next = ptr -> next;
    free (ptr);
    return start;
  }
```



4)
```
struct node * delete_after (struct node * start)
{
  struct node * ptr, * temp;
  int val;
  pf ("Enter data after which node to be deleted");
  sf ("%d", & val);
  ptr = start;
  temp = start;
  while (temp -> data != val)
  {
    temp = ptr;
    ptr = ptr -> next;
  }
  temp -> next = ptr -> next;
  free (ptr);
  return start;
}
```

For delete_before only stopping condition for while changes. else all remains same.

```
while (ptr -> next -> data != val)
{
  temp = ptr;
  ptr = ptr -> next;
}
temp -> next = ptr -> next;
free (ptr);
return start;
```

Menu → 1) Create
2) Insert
3) Delete
4) Display

LL Basic requirement

Circular linked list

(Create)

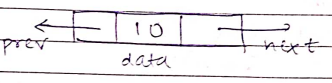
```
struct node *  create_CLL (struct node * start)
{
  struct node *nn,*ptr;
  int num;
  pf ("Enter -1 to End");
  pf ("Enter data");
  sf ("%d", &num);
  while (num != -1)
    {
      nn = (struct node *) malloc (sizeof (struct node));
      nn -> data = num;
    if (start == NULL)
    {
      start = nn;                    ✗ reverse not allowed
      nn -> next = start;
    }
  else
    {
      ptr = start;
      while (ptr -> next != start)
        {
          ptr = ptr -> next;
        }
      nn -> next = start;
      ptr -> next = nn;
  } // else
  pf ("Enter data");
  sf ("%d", num);
} // while
  return start;
} // function
```

End of valid ptr -> next != NULL
End & head value ptr != NULL

Page No.:
Date:

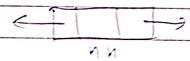
Doubly linked list

Node of DLL –

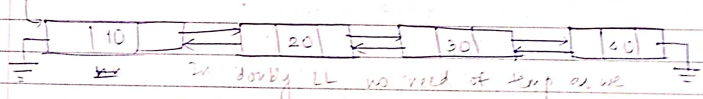



```
struct node
{
  struct node *prev, *next;
  int data;
};
```

struct node * start = NULL



nn → data
nn → prev
nn → next

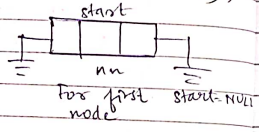


In doubly LL no need of temp as we have previous.

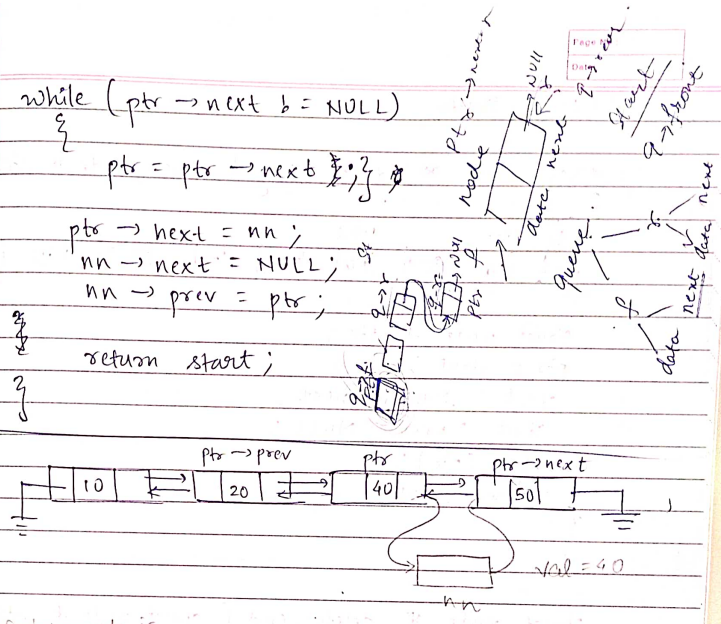
```
struct node * display (struct node * start)
{
  struct node *ptr;
  ptr = start;
  while (ptr != NULL)
  { pf ("%d", ptr -> data);
    ptr = ptr -> next; }

  return start;
}
```

TRY for * ptr

```
struct node * insert_beg (struct node * start)
{
    struct node * nn;
    int num;
    pf ("Insert data");
    sf ("%d", &num);

nn = (struct node *) malloc (size of (struct node));
nn -> data = num;
if (start == NULL)
{
    start = nn;
    nn -> next = NULL;
    nn -> prev = NULL;
}
else
{
    nn -> next = start;
    start -> prev = nn;
    nn -> prev = NULL;
    start = nn;
}
    return start;
}
```




```
struct node * insert_end (struct node * start) {
struct node * nn, * ptr; int num;
printf ("Enter val");
sf ("%d", &num);
nn = (struct node *) malloc (size of (struct node));
nn -> data = num;
    ptr = start;
while (ptr -> next != NULL)
{
    ptr = ptr -> next;
}
ptr -> next = nn;
nn -> next = NULL;
nn -> prev = ptr;
}
    return start;
```



check if before node is start then call insert_beg

```
ptr = start;
```

Insert before

```
while (ptr -> data != val)
{ ptr = ptr -> next; }
```

Always perform operations logically

```
(1) ptr -> prev -> next = nn;
(2) nn -> next = ptr;
(3) nn -> prev = ptr -> prev;
(4) ptr -> prev = nn;

return start;
```

check if after node is last one if true then call insert_end

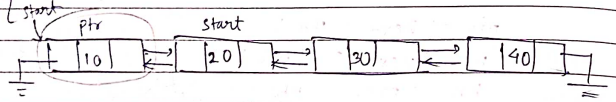
Insert After

```
while (ptr -> data != val)
{ ptr = ptr -> next; }
```

```
nn -> prev = ptr;
nn -> next = ptr -> next;
ptr -> next -> prev = nn;
ptr -> next = nn;

return start;
```

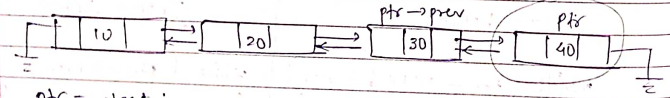
DLL → Delete

```
struct node * delete_beg (struct node * start)
{
```



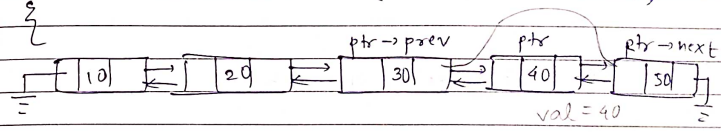

```
    struct node * ptr;
    ptr = start;
    start = start → next;
    start → prev = NULL;
    free (ptr);
    return start;
}
```

```
struct node * delete_end (struct node * start)
{
```



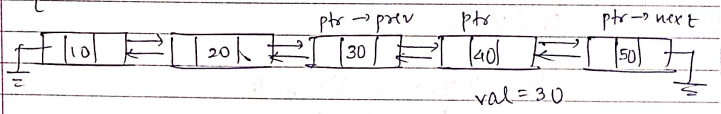

```
    ptr = start;
    while (ptr → next != NULL)
    {
        ptr = ptr → next;
    }
    ptr → prev → next = NULL;
    free (ptr);
    return start;
}
```

```
struct node * delete_node (struct node * start)
{
```




```
    while (ptr → data != val)
    {
        ptr = ptr → next;
    }
    ptr → prev → next = ptr → next;
    ptr → next → prev = ptr → prev;
    free (ptr);
    return start;
}
```

```
struct node * delete_after (struct node * start)
{
```




```
    while (ptr → prev → data != val)
    {
        ptr = ptr → next;
    }
    ptr → prev → next = ptr → next;
    ptr → next → prev = ptr → prev;
    free (ptr);
    return start;
}
```

Array in Java are dynamic → space at runtime
↳ using new keyword
Array in C are static → size fixed

In C everything static unless new operator or malloc, calloc used.

Implementation of Stack:

```
struct stack
{
  int data;
  struct stack *next;
};

struct stack *top = NULL;
struct stack *push (struct stack *top, int val)
{
  struct stack *ptr;   (ptr is ptr)
  ptr = (struct stack *) malloc (size of
        (struct stack));
  ptr -> data = val;
  if (top == NULL)
  {
    // Insert 1st node
    top = ptr;
    ptr -> next = NULL;   can do
  }
  else
  {
    ptr -> next = top;
    top = ptr;
  }
  return top;
}
```

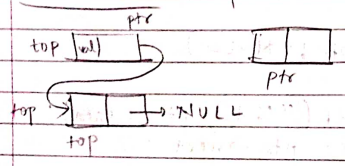
No need of if condition as space is not allocated and elements are added as a node

```
struct stack display (struct stack *top)
{
  struct stack *ptr;
  ptr = top;
  if (top == NULL)
    pf ("Stack Empty");
  else
  {
    while (ptr != NULL)
    {
      printf ("%d", ptr -> data);
      ptr = ptr -> next;
    }
  }
  return top;
}

struct stack pop (struct stack *top)
{
  struct stack *ptr;
  ptr = top;
  if (top == NULL)
  {
    printf ("Underflow");
  }
  else
  {
    top = top -> next;
    pf ("Val deleted = %d", ptr -> data);
    free (ptr);
  }
  return top;
}
```

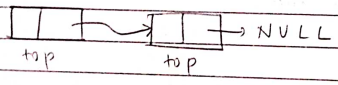
2024/8/30 08:50

```
int peek (struct stack * top)
{
    return top -> data;
}
```

V1 Vertical ptr = top



V2 Horizontal

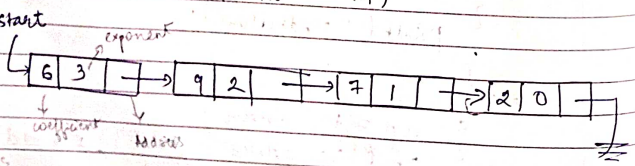


Queues using LL → Do Yourself.

Polynomial Representation (Application of LL

$P(x) = 6x^3 + 9x^2 + 7x + 1$

start



4 > 3 The one with higher exponent will go to resultant. The exponents will be compared for addition. If same they are added.

```
struct node
{
    int coeff;
    int exponent;
    struct node * next;
};
```

LL → dynamic (No fix size) → so no overflow

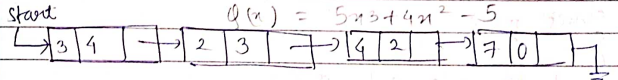
Addition

$P(x) = 3x^4 + 2x^3 - 4x^2 + 7$

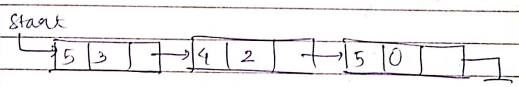
$Q(x) = 5x^3 + 4x^2 - 5$

$3x^4 + 7x^3 + 0x^2 + 7$

start

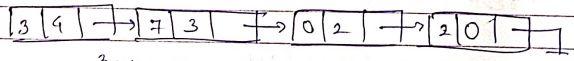


start



can also see multiply in more notes

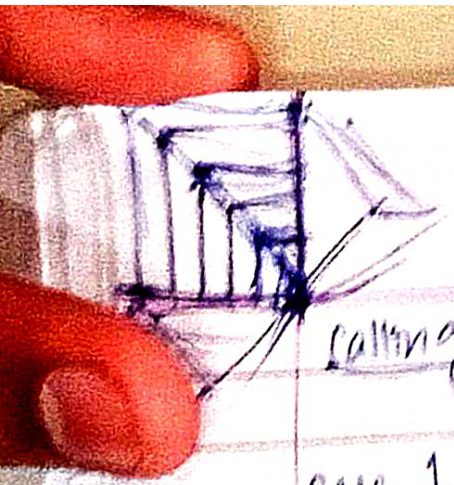
Resultant Polynomial -



$3x^4 + 7x^3 + 0x^2 + 2$ ✓

Linked Queue

```
#include <stdio.h>
struct node
{
   int data;
   struct node *next;
};
struct queue
{
   struct node *f, *r;
};
struct queue *q;

q -> front = NULL;
q -> rear = NULL;

struct queue * insert (int val)
{
 struct node * ptr;
 ptr = (struct node *) malloc (size of (struct node));

 ptr -> data = val;
 if ( q -> f == NULL)
 {
      q -> f = ptr;
      q -> ~~rear~~ ~~rear~~ r = ptr;
      q -> f -> next = NULL;
      q -> r -> next = NULL;
 }
 else
 {
      q -> r -> next = ptr;
      q -> rear = ptr;
      q -> rear -> next = NULL;
      return q;
 }

struct queue *del ( )
{
   struct node * ptr;
   ptr = q -> front;
   if ( q -> front == NULL)
      pf ("Underflow");
   else
      {
        q -> f = q -> f -> next;
        pf ("Deleted element = %d", ptr -> data);
        free (ptr);
      }

      return q;
}

int peek ( )
{
 return q -> front -> data;
}
```



calling

```
case 1:
    pf ("enter data");
    sf ("%d", &val);
    q = insert (val);
    break;
```