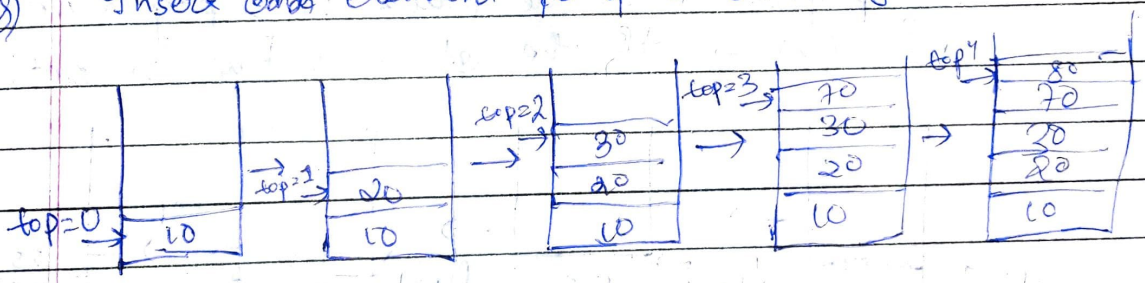
Stack

Stack is a linear data structure where elements can be added & deleted, inserted only one end i.e. top of stack.

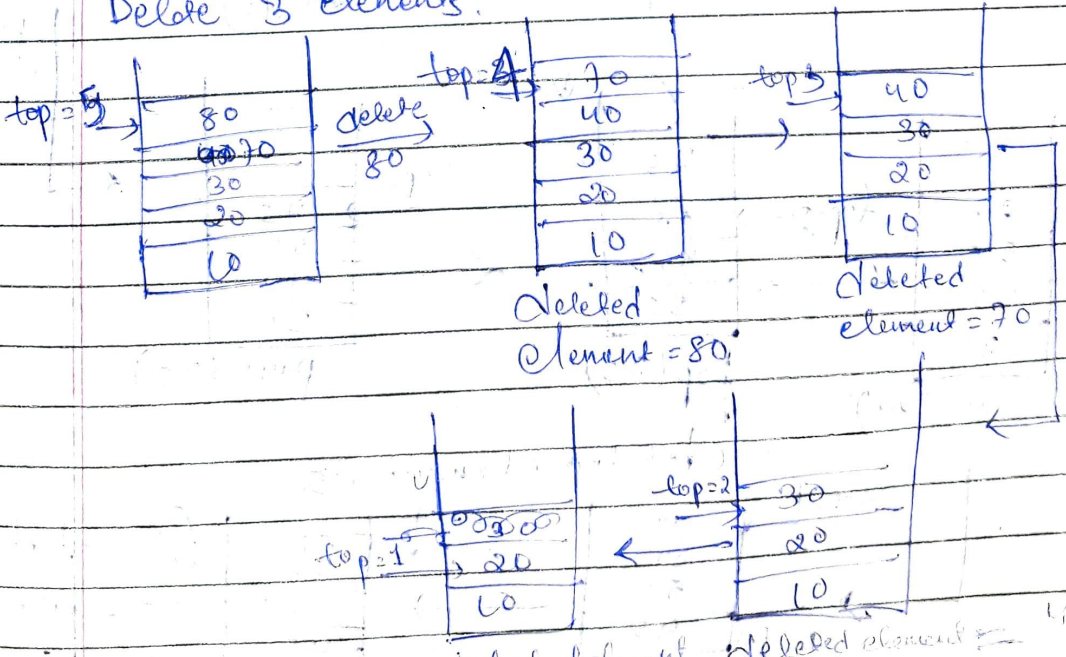
In array the programmers are allowed to insert & delete elements from the beginning from end & also from the middle.

As insert & delete operation isn't efficient, so it can't be used as a data structure in the application where frequent insert & delete are suppose to take place. An item or an element may be added or removed only from the top of the stack. It means the last element inserted to the stack is the first element to be removed from the stack.

Q) Insert element 70 & 80 in the given



Delete 3 elements.

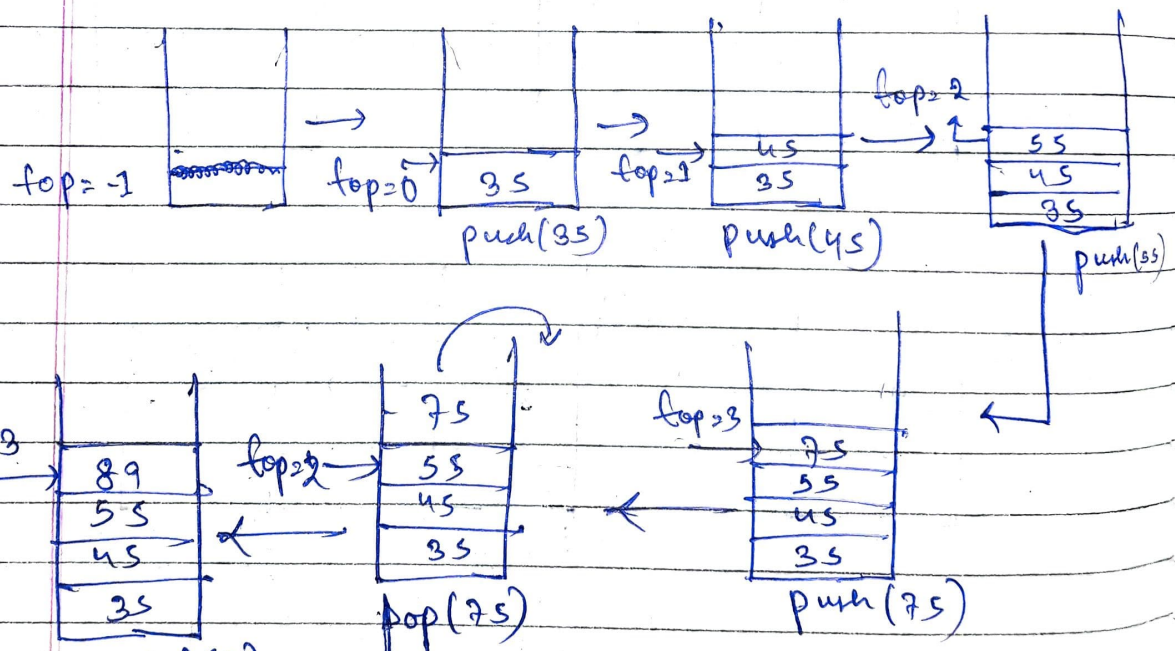


Stack follows LIFO (Last in first out) i.e. the element inserted last is the first element to be removed. It means that the elements are removed from the stack in the reverse order in which they were inserted into the stack.

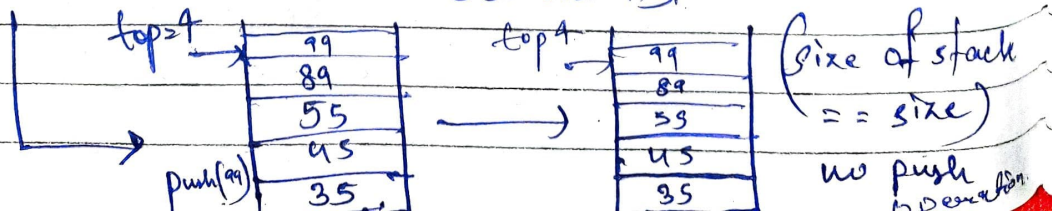
e.g. page visited history in a web-browser,
undo operations on a text editor (MS word)
function call

Terminology used in the stack

push → Insert an element to the stack.
pop → Delete an element from the stack
push(s, ele) → Insert an element ele to the top of a stack s.
pop(s) → delete & return the top element of the stack.
Create(s) → Create an empty stack s.
Isempty(s) → If the stack is empty or not.
isfull(s) → If the stack is full or not.



Deleted element = 75.



Representation of stack.

A stack can be represented in two ways:-
(i) Array representation (ii) Linked List representation

Array representation

We use an array & a variable top which holds the index of top element of the stack.
Initially the top is -1.
The limitation of this technique is that we have to declare the size of the array before performing any operation.
Stack is a linear & homogeneous type of DS.

Algorithms

```
Push(stack, ele);
1. If top == qsize.
        print OVERFLOW condition & return
2. Top = Top+1
3. Stack[TOP] = ele.
4. return.
```

```
POP(stack, )
1. If top == -1.
        print UNDERFLOW & return
2. del_ele = Stack[TOP].
3. Top = Top-1.
4. return del_ele.
```

```
#define size 100
int top;
void push(int stack[], int ele)
{
    if(top==size-1)
    {
        printf("Overflow");
        return;
    }
    top=top+1;
    stack[top]=ele;
}
int pop(int stack[])
{
    int del_ele;
    if(top==-1)
    {
        printf("Underflow");
        exit(0);
    }
    del_ele=stack[top];
    top--;
    return del_ele;
}
void display(int stack[])
{
    printf("Stack is:\n");
    if(top!=-1){for(i=top; i>=0; i--)
        printf("%d", stack[top]);
    }
    else
    printf("Stack is empty");
}
```

Output should be

1. Push
2. pop
3. display
4. Exit

Enter your choic
1 ↵

Enter the element
10 ↵
1 - push
2- pop
3- display
4- exit.
Enter your choice
1 ↵
Enter the element
30 ↵
1 - push
2- pop
3 - display
4- exist.
Enter the element.
20 ↵
1 -
2.
3 -
4-
Enter the choice
2 ↵
Deleted element.
20 ↵.

```
main()
{ int i, ele, stack[SIZE], ch; top=-1;
  for( ; ; )
  {
    printf("1.Push\n2.Pop\n3.display\n4.Exit");
    printf("\nEnter your choice");
    scanf("%c", ch);
    switch(ch)
    {
      case '1': printf("Enter the element:");
                scanf("%d", &ele);
                push(stack, ele);
                break;
      case '2': printf("Popped element: %d",
                       pop(stack));
                break;
      case '3': display(stack);
                break;
      case '4': exit(0);
      default : printf("Enter proper choice\n");
                break;
    }
  }
}
```

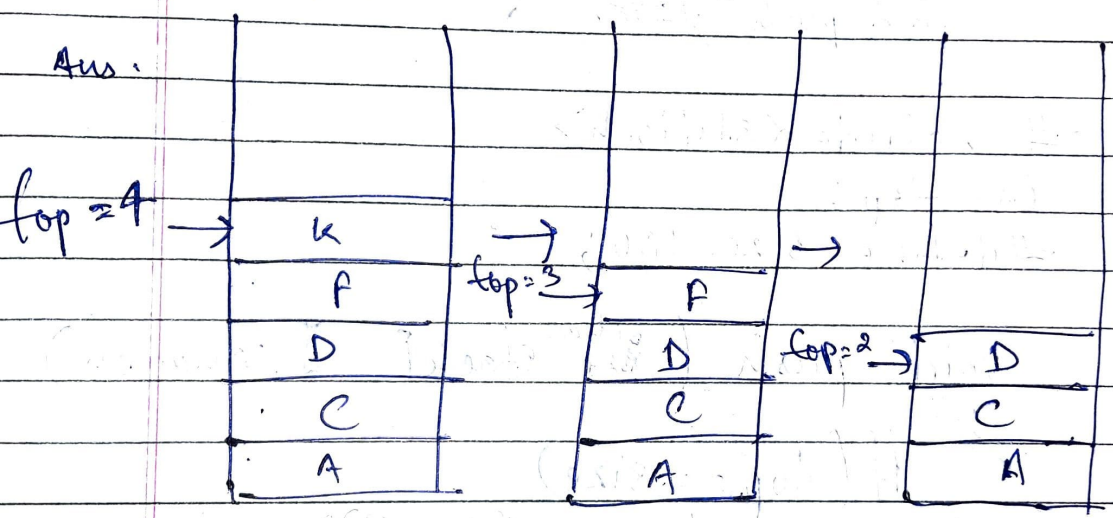
Q) Consider the following stack of characters the stack is represented as

STACK A, C, D, F, K, —, —, —, —.

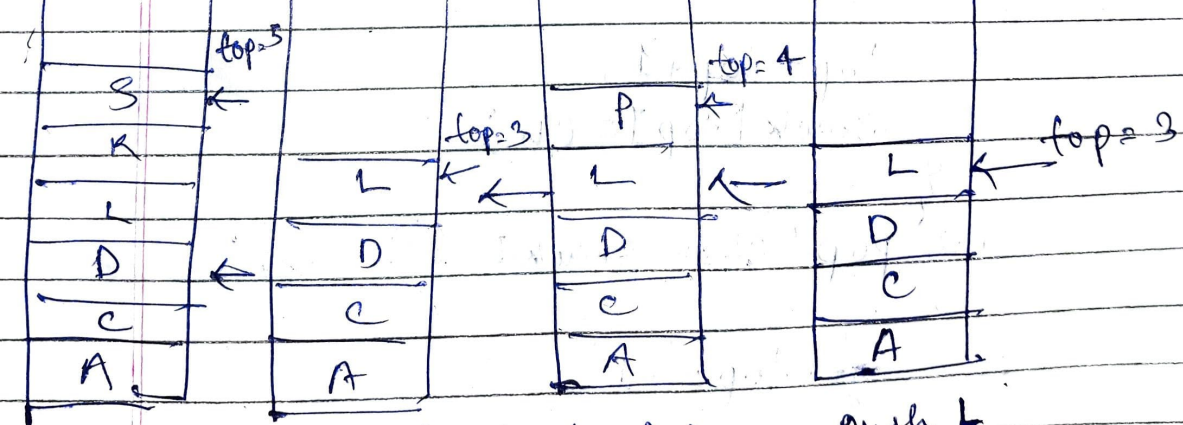
perform the following operation.

pop ( )
pop ( )
push (stack, L)
push (Stack, P)
pop ( )
push (stack, R)
push (Stack, S)
pop ( ).

Ans:



Deleted element=K. Deleted element=

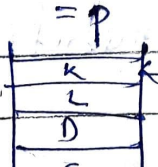


Push R Push S

Deleted element = P

push P

push L



Deleted element = S.

Q) WAP to reverse a string using Stack

Ans.

```
#include<stdio.h>
main()
{
  char str[50]

  printf("Enter the string : ");
  scanf("%s", str);
  for(i=0; i<sizeof(str); i++)
```

~~#include<stdio.h>~~
~~int top;~~
~~void push(char stack~~

```
#include<stdio.h>
int top;
#define size 100;

void push(char stack[ ], char ch)
{
  if(top==size)
  { printf("Overflow");
    return;
  }
  top=top+1;
  stack[top]=ch;
}
int pop(char stack[ ])
{
  if(top==-1)
  {
    printf("Underflow");
    return;
  }
```

```
    return stack[top];  top--;
}
main()
{
    char stack[size], ch; str[100];
    int i; top=-1;
    printf("Enter the string: ");
    scanf("%s", str);
    for(i=0; i<sizeof(str); i++)
    {
        push(stack, str[i]);
    }
    for(i=0; top>-1; i++)
    {
    str[i]=pop(stack);
    }
    str[i]='\0';
    printf("string is: %s", str);
}
```

```
#define size 100
struct stack
{
    int st[size];
    int top;
}
```

Application of stack

stack is an appropriate data structure for application in which information must be saved & later retrive in reverse order.

→ Evaluation of arithmatic expression.
  → Evalution of postfix expression.
  → " " prefix "
  → Conversion " infix to "
  → " " infix " prefix.

→ Check for balanced paranthesis.

→ Implementation of recursive function.

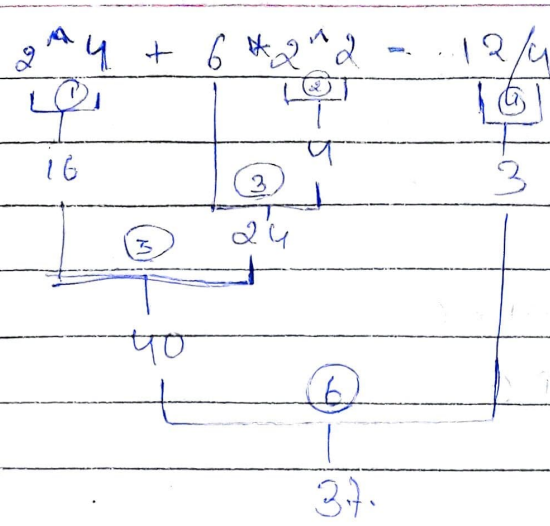
Arithmatic expression

→ It consist of operators & operands.

→ Binary operators have following presedence.

op	Precedence
↑	→ 3
*, /, %	→ 2
+, −	→ 1

* To evaluate an arithmatic exp. the problem is to find the order of evalution.
It is determined by the prescendence & associativity of the operator.



→ To evaluate the expression we must scan the expression from L → R, repeatedly.

→ This process is inefficient to the compiler. Hence convert given to postfix notation. Evaluate the postfix notation.

Notation for arithmetic expression

There are three notation to represent :-

(i) infix (ii) postfix (iii) prefix.

* In ~~prefix~~ infix operators are placed b/w the operand
* In postfix operator is placed after the operand.
* In prefix operator is placed before the operand.

A+B*C. A+B*C

A+BC* A+*BC

ABC*+. +A*BC.

(A+B)*C

AB+*C

AB

$(A+B)/(C-D)$ $(A+B)/(C-D)$

$AB+/CD-$ $+AB/-CD$

$AB+CD-/$ $/+AB-CD$.

$(A+B)*C$ $(A+B)*C$

$AB+C*$ $*+ABC$.

$A+B/C-D*B+F$.

$A+BC/-DB*+F$ | $A+/BC-*DB+F$

$+-+A/BC*DEF$.

$ABC/+DB*-F+$

$A+B/(C*D^B)-F*C^H$

$A+B/C*DE^*-F*C^H$. | $A+B/*C^DB-F*C^H$.

$A+B/CDE^**-F*CH^$ | $A+B/*C^DB-F*^CH$

$A+BCDE^*/-FCH^*$. | $A+/B*C^DB-*F^CH$

$ABCDE^*/+FCH^*-$ | $-+A/B*C^DBF^CH$.

The order of operator in the

→ The infix operation is Evaluated in two step

1. Convert infix to postfix.
2. Evaluate the postfix Expression

Evaluation of postfix expression.

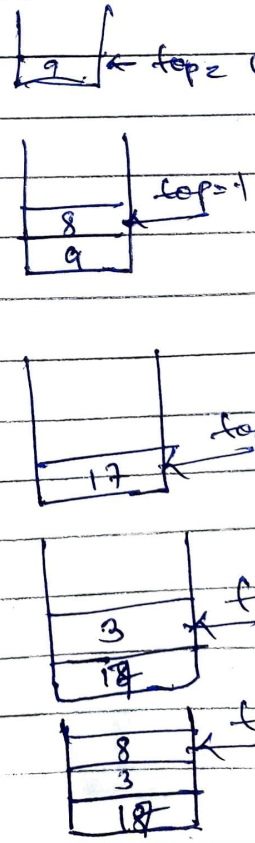
I/p → Arithmatic expression P
O/p → value of P.

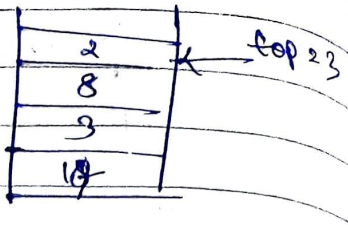
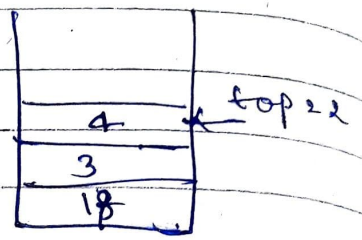
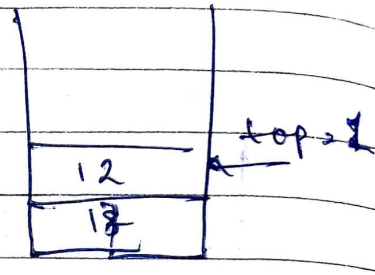
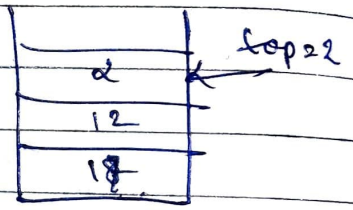
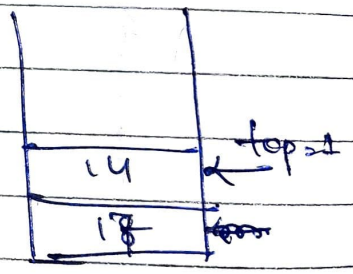
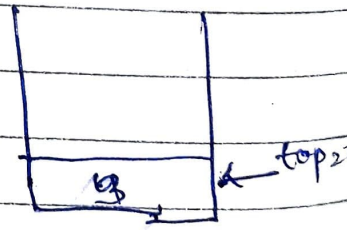
1. Take a empty stack.
2. while not the end of i/p.
   symbol = next i/p character.
   if (symbol is operand)
   Push the symbol to the stack.
   else if (symbol is operator)
   (1) pop two symbol from the stack where A is the top element B is the next top element
   (2) Evaluate B <symbol> A. & push the result to the stack.
3. Result = top element of stack.

eg - 9 8 + 3 8 2 / * 2 + =

Symbol Scanned
9
8
+
3
8

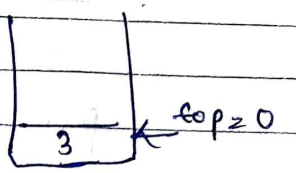
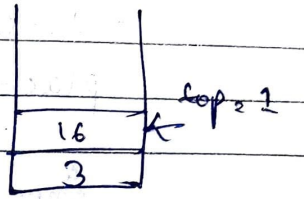
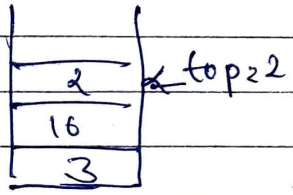
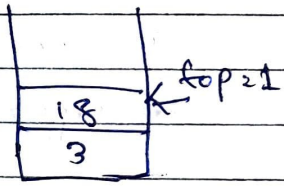
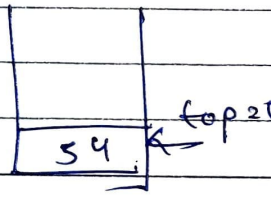
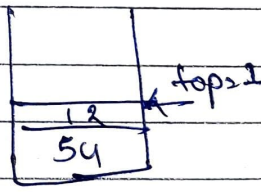
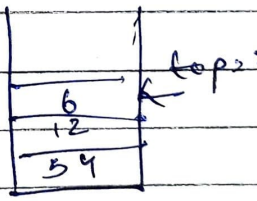
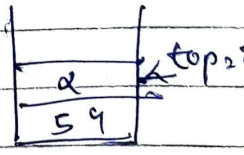
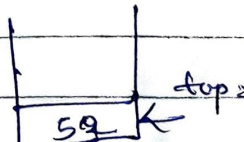
Stack



Symbol scanned	Stack
2	
/	
*	
2	
+	
-	

Result = 3.

3 16 2 + * 12 6 / -

Symbol scanned	Stack
3	
16	
2	
+	
*	
12	
6	
/	
-	

Converting an infix to postfix

while converting infix to postfix the parenthesis has to be elemented.

Stack will be used for temporary placement of operators

Input is infix expression
output is equivalent postfix expression P.

Algo

1. Take an empty stack.
2. Repeat while symbol isn't end of i/p.
   a- if the symbol is operand
      add the symbol to postfix string P.
   b- else if symbol is left '('
      push the symbol to stack.
   c- else if symbol is ')' then
      (i) Repeatedly pop operators from stack & add it to the postfix string P. until '(' is encountered.
      (ii) Remove '(' from the stack (don't add to the postfix string).
   d. else if symbol is operator.
      i. while (precedence of the top of the stack is >= precendence of the symbol)
      and stack is not empty.

Pop the stack & add it to postfix string P.

(iii) Push the symbol to the stack

3. Repeat while stack isn't empty.
Pop stack & add it to postfix string P.

(3) A + B/C * (D+E) - F.

	Symbol	Stack	Postfix string
1	A	Empty	A
2	+	+	A
3	B	+	AB
4	/	+/	AB
5	C	+/	ABC
6	*	+*	ABC/
7	(	+*(	ABC/
8	D	+*(	ABC/D
9	+	+*(+	ABC/D
10	E	+*(+	ABC/DE
11	)	+*	ABC/DE+
12	-	-	ABC/DE+*
13	F	-	ABC/DE+*F
14		-	ABC/DE+*F-
15		A	ABC/DE+F-*
16			ABC/DE+F-*+

Result = ABC/DE+*+F-

9) A+(B*C-(D/E^F)*G)*H.

Symbol	Stack	postfix
A	null	A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
^	+(-(/^	ABC*DE
F	+(-(/^	ABC*DEF
)	+(-	ABC*DEF^/
*	+(-*	ABC*DEF^/
G	+(-*	ABC*DEF^/G
)	+	ABC*DEF^/G*-
*	+*	ABC*DEF^/G*-
H	+*	ABC*DEF^/G*-H
	empty	ABC*DEF^/G*-H*+

Result : ABC*DEF^/G*-H*+.

8) WAP to convert a decimal no. to binary using stack.

Ans-

```
#include<stdio.h>
main()
{
    printf("Enter the number: ");
    scanf("%d", &n);
```

Ans

```
#include<stdio.h>
struct binary
{
  int st[10];
  int top;
};
main()
{ struct binary stack;
  int n;
  printf("Enter the number: ");
  scanf("%d", &n);
  while(n)
   {
     push(stack, x%2);
     x = x/2;
   }

void push(struct binary s);
{
```

* Check whether the expression is having valid parenthesis or not.

3 invalid case :- (i) More no. of left parenthesis
(ii) more no. of right parenthesis
(iii) mismatch left & right parenthesis.

Case1
Input is finished stack is not empty.

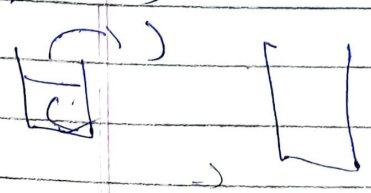
Case 2
Input is on going but stack is empty.

case3

Algo

(1) Take an empty stack.
(2) flag = 1.
(3) while not end of I/p & flag == 1 do
if (symbol == '(' || symbol == '{' ||
symbol == '[')
push (stack, symbol).
else if (symbol == ')' || symbol == '}' ||
symbol == ']').
if stack is empty.
print more no. of right parenthesis
flag = 0.
else
c = pop (stack).
if (symbol == ')' && (c == '[' || c == '{')
flag = 0;
print mismatch left & right parenthesis
else if (symbol == ']' && (c == '(' || c == '{')
flag = 0
print mismatch left & right parenthesis

(9+5)



else if (symbol == '}' && (c == '[' || c == '('))
    flag = 0;
    print mismatched parenthesis.

4- if (stack is not empty)
    flag = 0;
    print more no. of left parenthesis.

5- if (flag == 0)
    print Invalid parenthesis.

6- else
    print valid parenthesis

Queue

Queue is a linear data structure in which deletion can take placed only at one end called front & insertion can take place only a in on end called rear

Queue follows the principle of FIFO i.e. the order in which the elements enter in a queue, is the order in which they leave the queue.

Applications

* People waiting at the line in the bus stop.
* In time sharing system the program with same priority form a queue while waiting to be executed
* Access to the shared resources.
  e.g. printer.

Operations

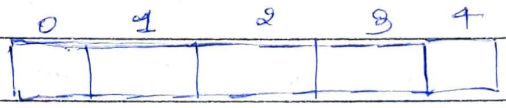
1- Insert 2- Delete 3- ~~trays~~ traverse.

Representation of the queue

1. Array implementation.
2. Linked List implementation.

Array implementation

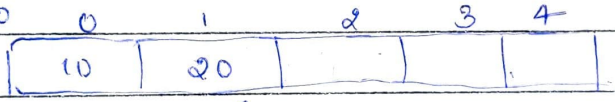
Let the size of the queue is 5, front = Rear = -1.



insert 10

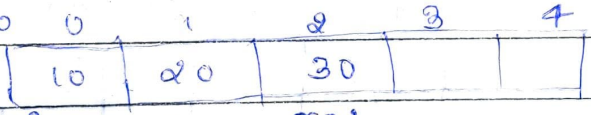
f=0, r=0

insert 20



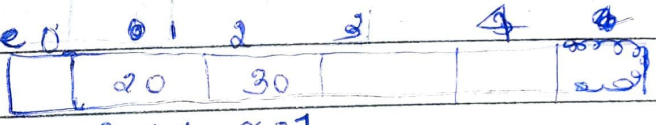
f=0 r=1

insert 30



f=0 r=2

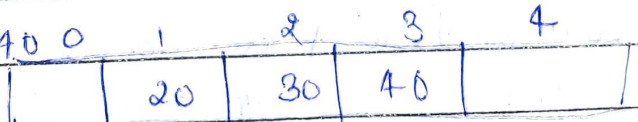
delete



f=1 r=1

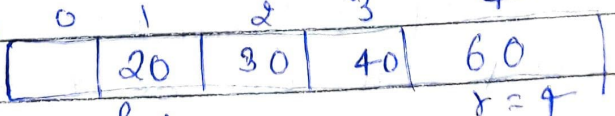
del-ele = 10.

insert 40



f=1 r=3

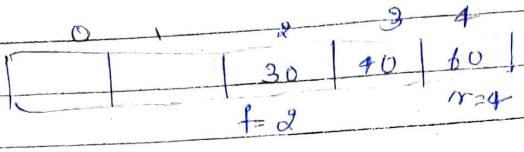
insert 60



f=1 r=4

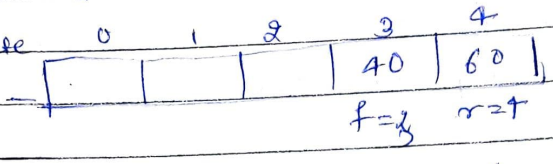
Condition to be queue is full

delete

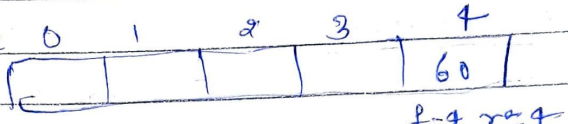


del_ele = 20.

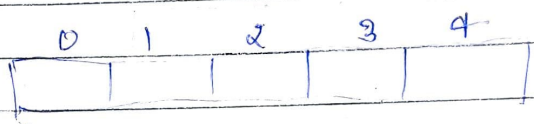
delete



delete



delete



size=5

insert (17)

insert (23)

insert (97)

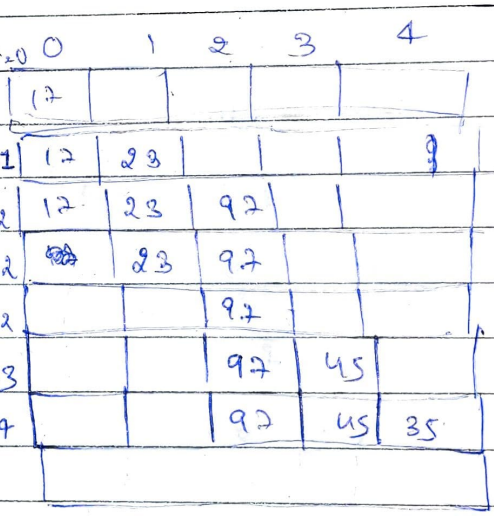
delete ( )

delete ( )

insert (45)

inser (35)

Overflow



* In array implementation we use an array two variables front & rear where rear hold the index of lastly added element in the queue & front holds the index of the element to be deleted next. Initially when the queue is empty both rear & front will be -1. When insert in an empty queue both front & rear increment by 1 whereas in non-empty only rear is increased & put the element in the new rear position.

When we delete from the queue having one element set f=r=-1. When front == rear ≠ -1 then the queue contain one element.

Limitations of Linear queue

* You have declare the size

Algo

insert (queue, ele)

1- if(rear == size - 1)
   then print queue overflow
   return;

2- if (front == -1)
   front = 0; rear = 0;

3- else
   rear++;

4- queue[rear] = ele;

delete (queue)
if (front == -1)
   then print queue underflow
   return;
if(front == rear)
   del-ele = queue [front];
if(front == rear)
   front = -1;
   rear = -1;
else
   front++;
return del-ele;

```
#include<stdlib.h>
#include<stdio.h>
#define SIZE 100
int f, r;
void insert (int q[ ], int ele)
{
  if(r==SIZE-1)
  {
    printf("Queue overflow");
    return;
  }
  else if(f==-1)
  {
    f=0;
    r=0;
  }
  else
    r++;
  q[r]=ele;
}
int delete (int q[ ])
{
  if(f==-1)
  {
    printf("Queue underflow");
    exit(0);
  }
  else if(f==r)
  { del_ele=q[f];
    f=-1;
    r=-1;
  }
  else
  {
    del_ele=q[f];
    f++;
  }
  return del_ele;
}
```

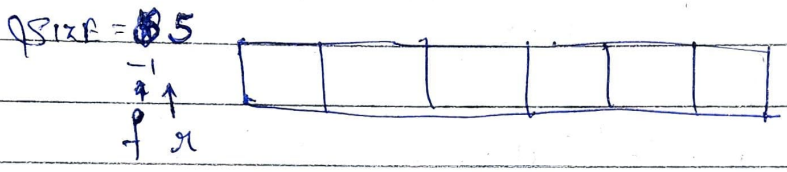
```
void display ( int q[ ])
{
    int i=0;
    if( f==-1)
    {
        printf ("Queue underflow ");
        return;
    }
    else
    { printf ("Queue is : \n");
      while (i<=r)
      {
          printf("%d", q[i]); i++;
      }
    }
}

main()
{   f=-1, r=-1;
    int ch, q[SIZE];
    for( ; ; )
    printf("1: INSERT \n2- DELETE \n 3 - DISPLAY \n 4- EXIT \n\n");
    printf("Enter your choice: ");
    scanf ("%d", &ch);
    switch (ch)
    {
      case 1: printf("\n Enter your element: ");
              scanf("%d", &ele);
              insert (q, ele);
              break;
      case 2: printf("Deleted element: %d", delete(q));
              break;
      Case 3: display (q);
              break;
      Case 4: exit(0);
              break;
      default: printf("Enter proper choice: \n");
    }}
```

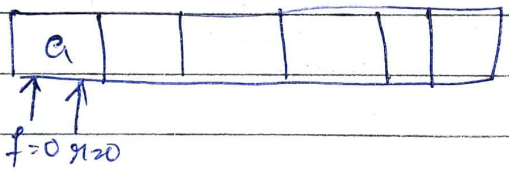
```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 100.
struct queue
{
    int q[SIZE];
    int f, r;
};
void insert(struct queue *p, int ele)
{
    if(p->r == SIZE-1)
    {
        printf("Queue overflow");
        return;
    }
    elseif(p->f == -1)
    {
        p->f = 0;
        p->r = 0;
    }
    else
        (p->r)++;
    p->q[p->r] = ele;
}
int delete(struct queue *p)
{   int del_ele;
    if(p->f == -1)
    {
        printf("Stack underflow");
        exit(0);
    }
    del_ele = p->q[p->r];
```

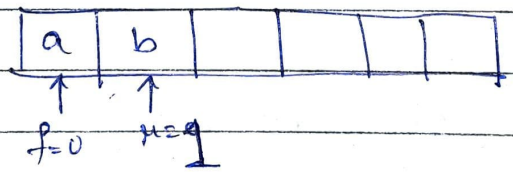
```
if(p->f==p->r)
{
    p->f=-1;
    p->r=-1;
}
else
    (p->f)++;
return del_ele;
}
void display (qstruct queue p)
{ int i;
  if(p.f==-1)
  {
      printf("Queue Underflow");
      return;
  }
  printf("Queue is:\n");
  for(i=0; i<=p.r; i++)
      printf("%d ", p.q[p.i]);
}
main()
{ int ch,
  qstruct queue s;
  s.f=-1;
  s.r=-1;
  for( ; ; )
  {
    printf("1-INSERT\n2-DELETE\n3-DISPLAY\n4-EXIT
    printf("Enter your choice: ");
    qscanf("%d", &ch);
    qswitch(ch)
    {
      case 1: printf("Enter the ele: ");
              qscanf("%d", &ele);
              insert(&s, ele);
              break;
```

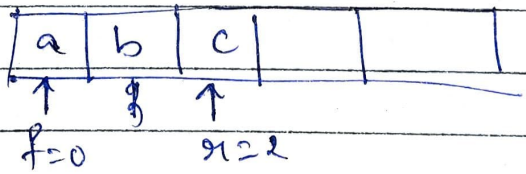
```
case 2: printf("Deleted element is: %d", delete(&s));
        break;
case 3: display(s);
        break;
case 4: exit(0);
        break;
default: printf("Enter proper choice\n");
}
}
}
```

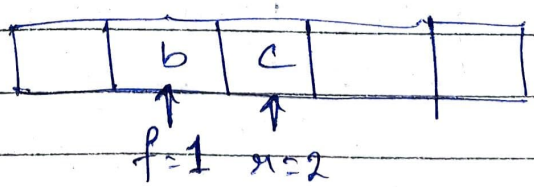
Limitation of Linear queue.

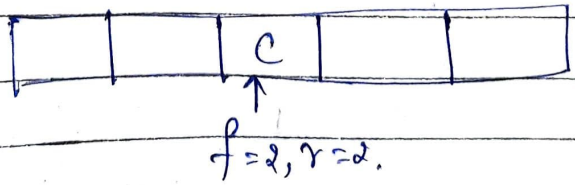


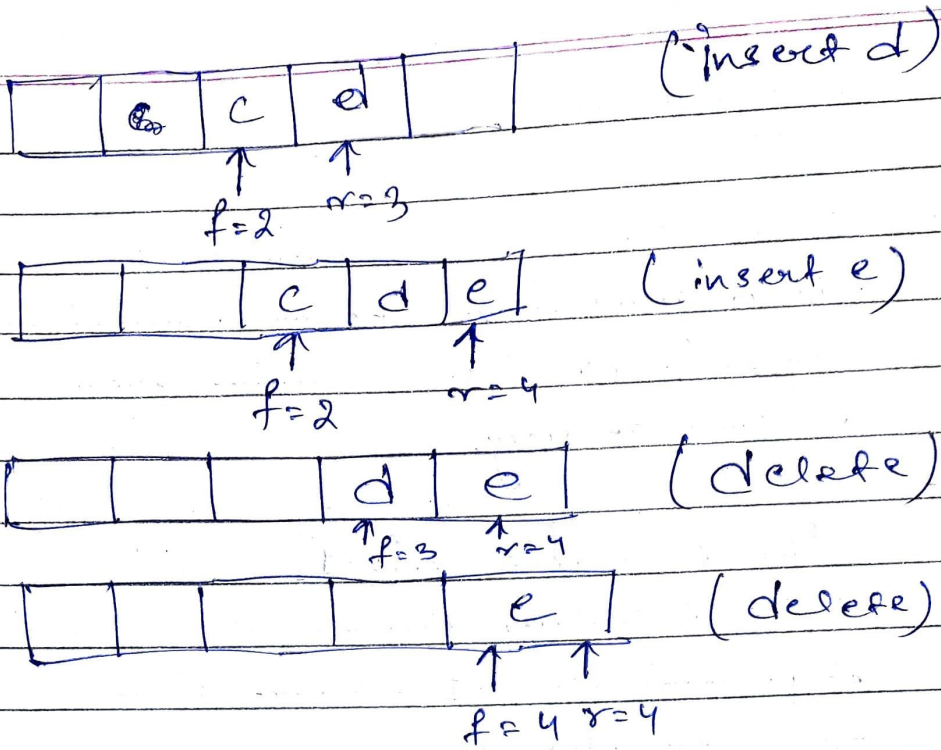






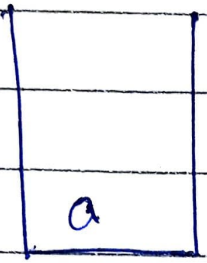






When rear = SIZE-1 & front is >0 (some element has been deleted) suppose at this point you want to insert a new element if we cannot insert because overflow condition satisfies even if there is some vacant space in the queue that means overflow condition of linear queue doesn't necessarily implies that queue is physically full. This leads to limitation of rejecting the insertion, despite the space available to accomodate it. This leads to circular queue.

(a+b)/(c-d)



	Stack	String
1	(	EMPTY
2	*(	a
3	*(+	ab
4	* NULL	ab+
5	/	ab+
6	/(	ab+
7	/c	ab+c
8	/(-	ab+c
9	/(-	ab+cd
10	/-	ab+cd-
11		ab+cd-/