UNIT-2

→ Introduction to stacks :-

* Stack is a data structure that works based on principle of last-in-First-out (LIFO)
* It means the last element that was added to the stack must be the first one to be removed.
* In stack all operations such as Insertion and Deletion are permitted at only one end called TOP

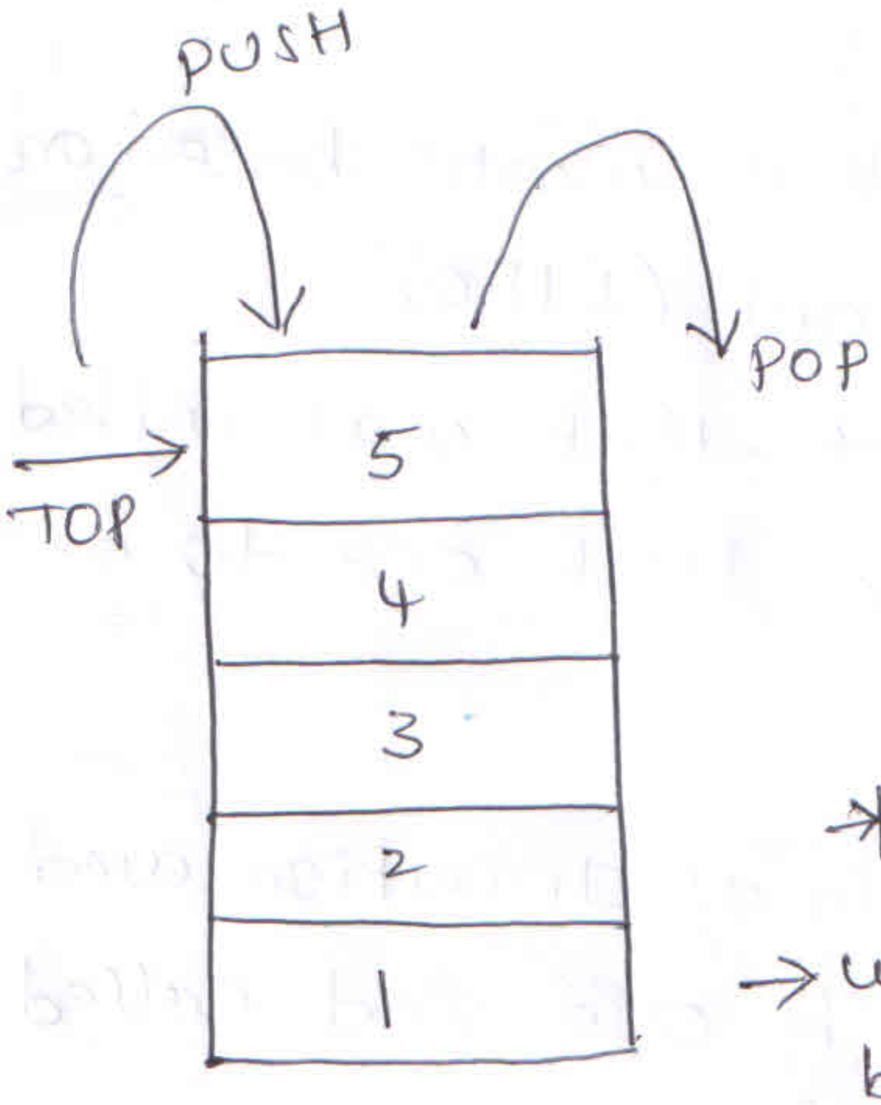
Def of stack :- stack is collection of items in which the data are inserted and remove from One end called the top of the stack

Example :-

Books are kept one above the other and the books which is inseted First is taken out at last. Book which is inseted lastly is seved first.

→ Operations on stack :-

* there are two main operations that can perform in the stack. 1, PUSH 2, POP
* PUSH: allows you to insert an element into stac
* POP: allows you to remove an element from the sta

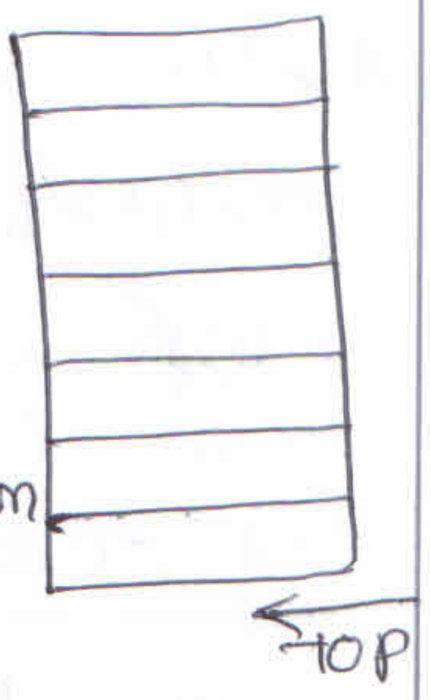


∴ Stack ∴

* In additional a stack can be
  - → Empty : Checks if the stack is Empty
  - → Full : Checks if the stack is Full
  - → Display : Displays the content of the stack.

* when stack is Empty :-

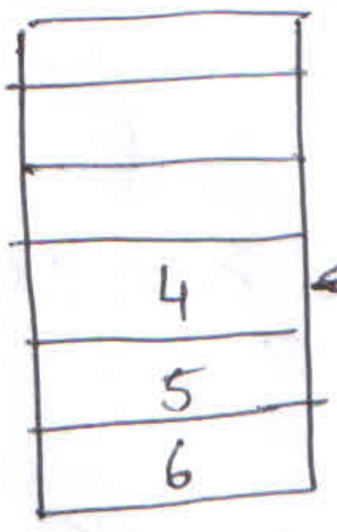
→ when stack is said to be empty then it does not contain any element inside it.



→ when ever the stack is empty position of the topmost element is -1 or 0

→ If top = -1 then array index position start with "0 - - - - ∞" (or) If top = 0 then array index position start with "1 - - - - ∞"

* when stack is not Empty



→ whenever we add very first element then topmost position will be incremented by '1'

→ Representation of a stack :-

* stack may be represented in the memory in various ways 1, Array
  2, linked list

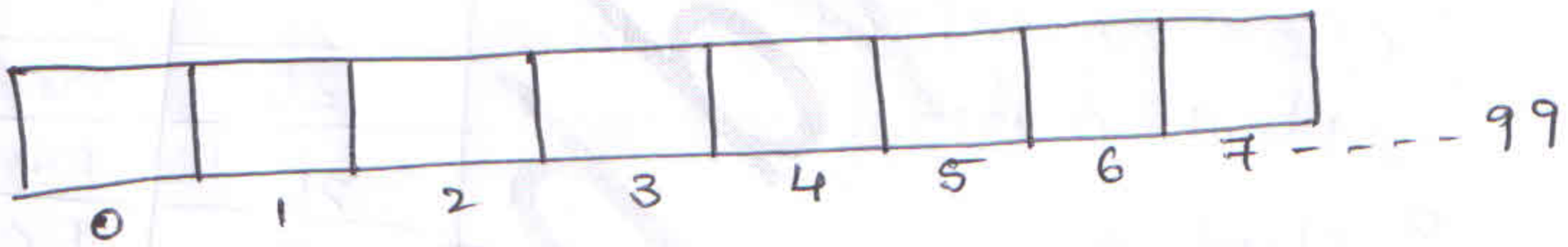
Array Representation of a Stack :-

* It is ordered list with some restrictions on the way in which we perform various operations on a list
* An array of some MAXIMUM size and we need an integer variable top which will keep track of the top of the stack as more and more elements are inserted into and delete from the stack.
* There are two types of declarations are there

Declaration1 :-

```
# define Size 100
  int stack[Size];
  top=-1;
```

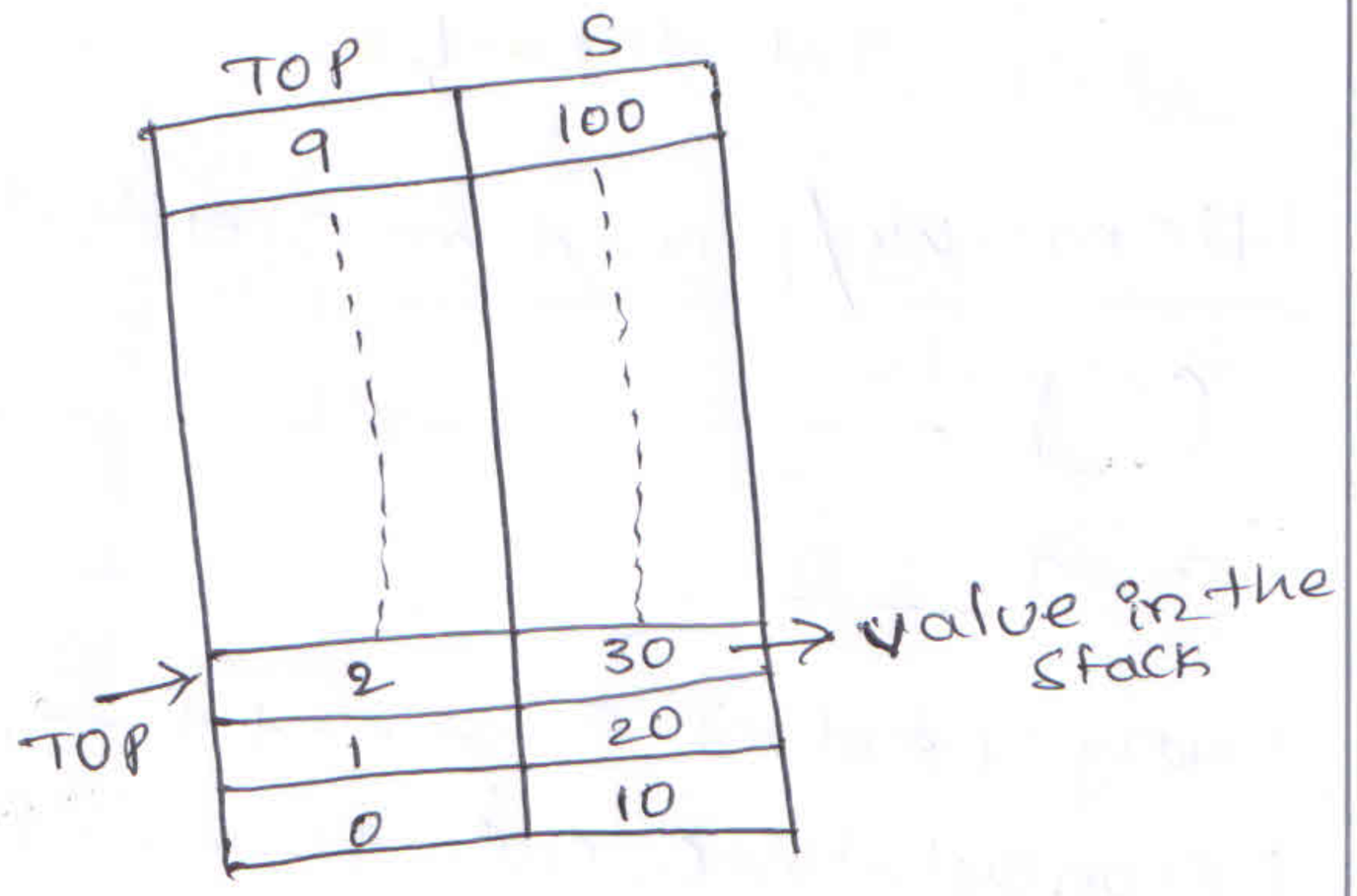
* In the declaration stack is nothing but an array of integers and most recent index of that array will act as top



* Note :- Here top=-1

Declaration2 :-

```
# define Size 5
  Struct Stack
  {
    int S[Size];
    int top;
  } St;
```



* Above two declarations 1, 2 are for stack only but the 2nd declaration will always petered.

Because in the 2nd declaration we have used a struct for stack element and top

* TOP & stack are associated with each other by putting them together in a structure.

* Ex:- Stack can also be used in the db. If want to store ~~structur~~ student marks in db and declaration of stack as shown in below.

```
# define size 60
typedef struct student
  {
    int rollno;
    char name[30];
    Float marks;
  } stud;
stud S1[size];
  int top=-1;
```

	Roll_no	Name	Marks
59			
top=2			
2	30	madhu	88
1	20	Bindu	98
0	10	Deepu	96

Hierarchy/precedence Operators :-

( )	(left to Right) L → R
[ ]	L → R
Unary Operator's	++, --, !, *, &, ~ ---- R → L
Exponentation Operator's	$, ^ ---- L → R
Arithmetic operator's	*, /, %, ---- L → R
Arithmetic operator's	+, - ---- L → R.

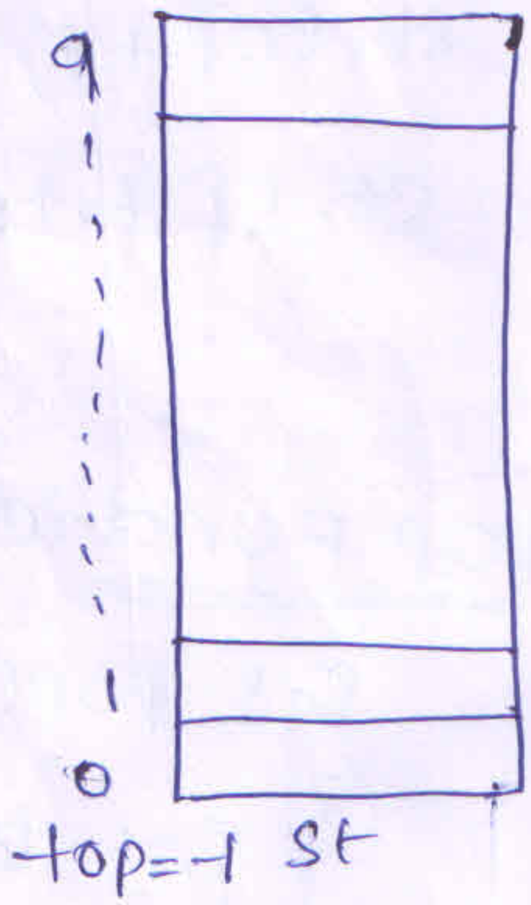
→ Stack Empty Operation :-

* Initially stack is Empty
* top should be initialized to -1 or 0
* If we set top to -1 initially then stack will contain the elements from 0th position and if we set top to '0' initially. the elements will be stored from 1st position. in the stack.
* Elements may be pushed onto the stack and there may be a case that all the elements are removed from the stack. then the stack becomes Empty
* Then top reaches to -1 we can say the stack is Empty

Note :-

top=-1 means stack is Empty

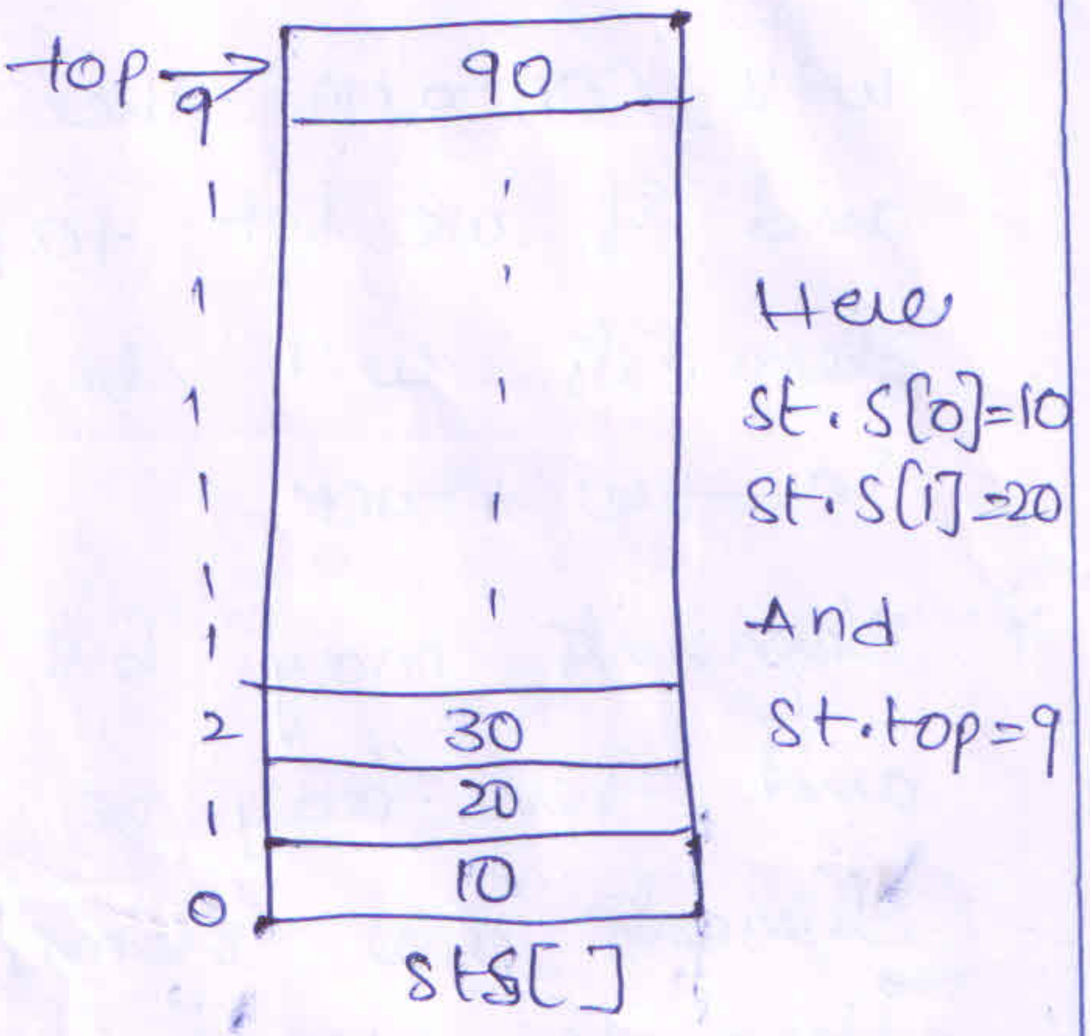
```
int StEmpty()
 {
  if (St.top==-1)
   return 1;
  else
   return 0;
 }
```



→ Stack Full operation :-

-the representation of stack using arrays size of array means size of stack. Stack Full condition is achieved when stack reaches to maximum size of array.

```
int STFull()
{
  if(st.top >= size-1)
     return 1;
     else
      return 0;
  }
```



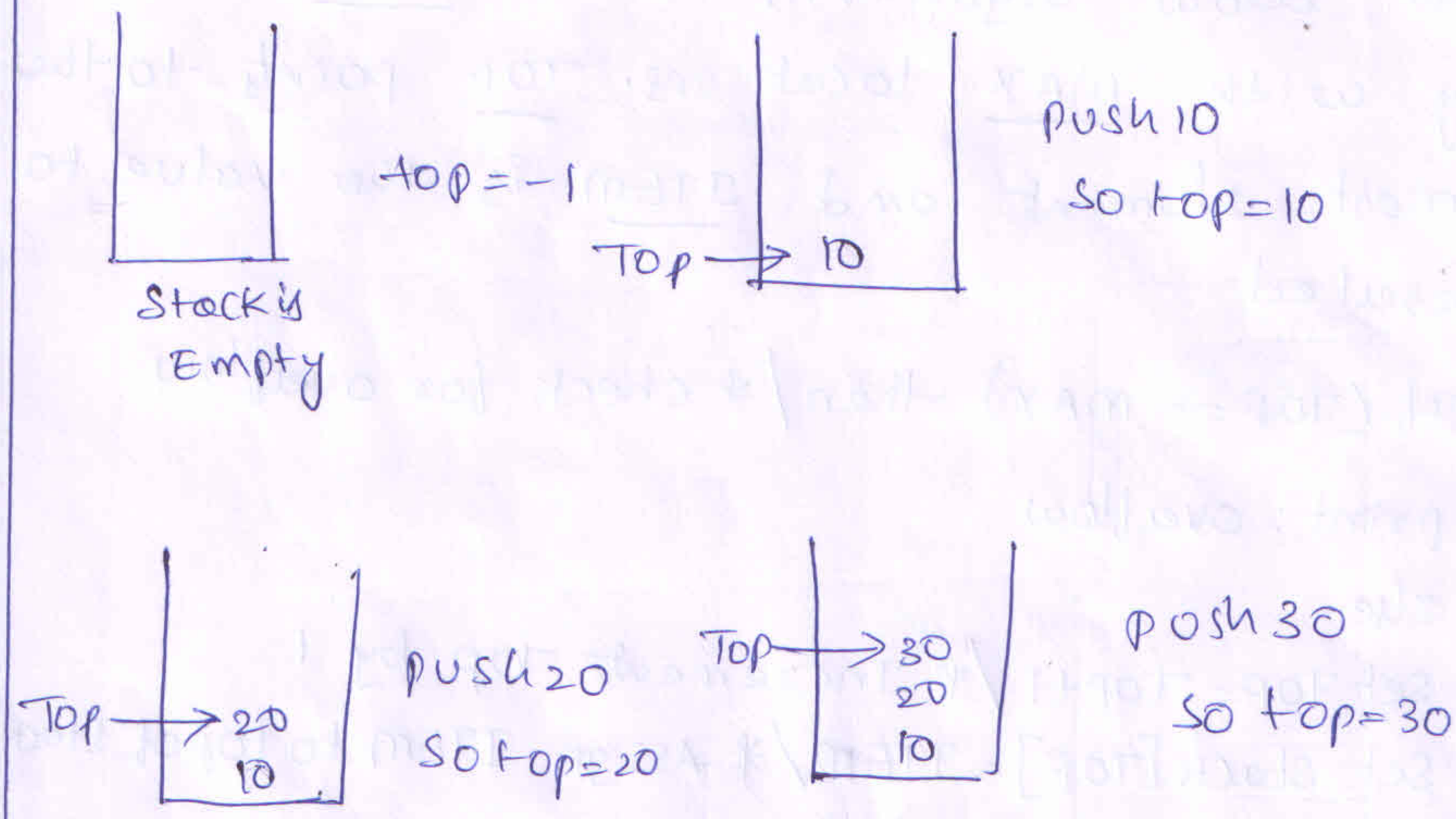
→ push Function :-

```
void push(int item)
{
  st.top++; /* top pointer is set to next location
   st.s[st.top] = item; /* placing the elements at
                           that location.
```

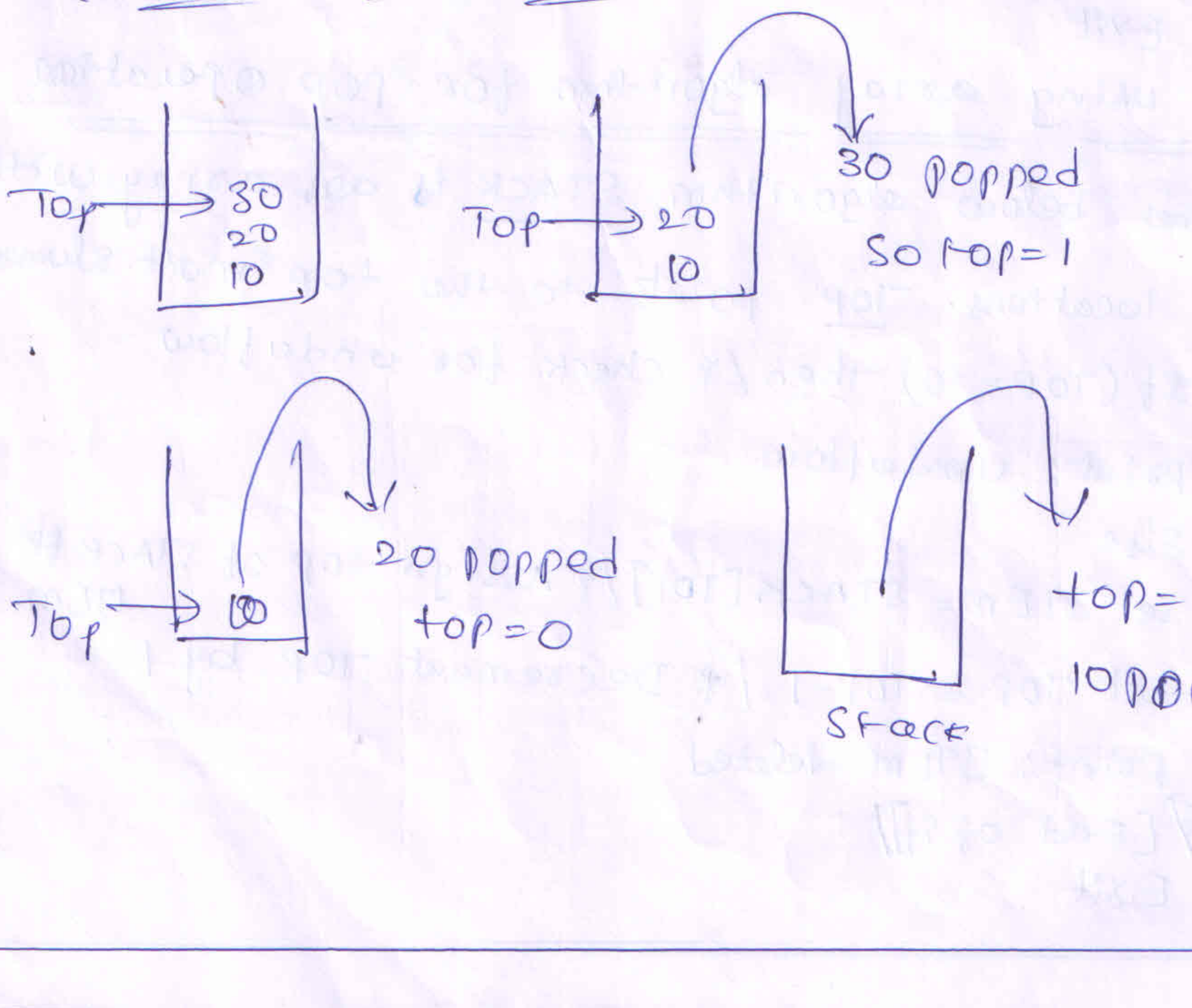
→ pop Function :-

```
int pop()
{ int item;
  item = st.s[st.top];
   st.top--;
   return(item);
}
```

Performing push operation :-



Performing pop operation :-



→ Stack using array algorithm for push operation

In the below algorithm. Here Stack is an array with MAX locations. TOP point to the top most element and ITEM is the value to be inserted

```
If (TOP == MAX) then /* check for overflow
print: overflow
else
set TOP = TOP+1 /* Increment top by 1
Set stack [TOP] = ITEM /* Assign ITEM to top of Stack
print: ITEM inseated
*/[End of if]
Exit
```

→ Stack using array algorithm for POP operation

In the below algorithm STACK is an array with MAX locations. TOP points to the top most element

```
If (TOP == 0) then /* check for underflow
print: underflow
else
Set ITEM = STACK [TOP] /* Assign top of STACK to ITEM
Set TOP = TOP-1 /* Decrement TOP by 1
point: ITEM deleted
*/[End of if]
Exit
```

Stack using array algorithm for Display operation

1. If top = -1
   print " underflow"
2. Repeat step 3 for i = TOP to i >= 0
3. print Stack[i]
4. Return

→ **Stack Applications :-**
1. Expression conversion
2. Expression Evaluation
3. parsing well formed parenthesis
4. Decimal to binary Conversion
5. Storing function calls.
6. Reversing list

Reversing list :-

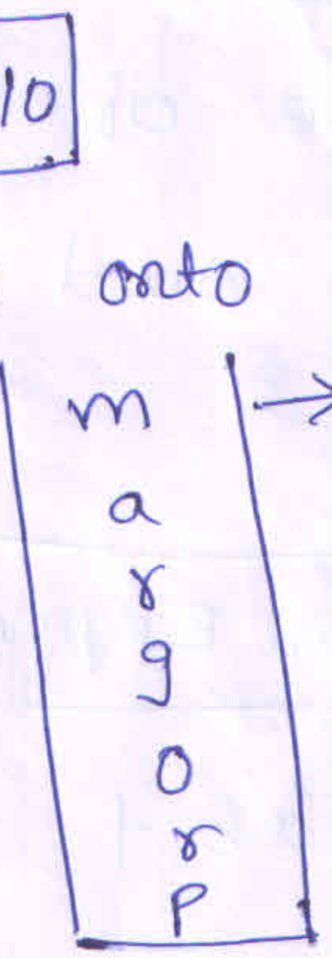
To reverse a string stack can be used. The simple mechanism is to push all the characters of a string onto the stack and then pop all the characters from the stacks and print them

Example :-

input string

p	r	o	g	r	a	m	10

Then push all the characters onto the stack till '10' is not encountered



→ Expressions :-

Expression is a string of operands and operators Operands are some numeric values. ~~and operators~~ ~~are of types~~ operators are '+', '-', '*', '%' etc there are 3 types of expressions !, Infix Expression
2, postfix "
3, prefix "

1. Infix Expression :-

In this type of expression the arrangement of operands and operators are shown in below

Infix Expression = operand1 operator operand2

Example :- A+B , (a+b) * (c-d)

postfix Expression :-

In this type of Expression the arrangment of operands and operators is as shown in below

postfix Expression = operand1 operand2 operat

Example :- ab+ , ab+cd-*

prefix Expression:-

In this Expression the arrangement of operands and operators are shown in below

prefix Expression = operator operand1 operand2

Example :- +ab , */+ab-cd.

→ Algorithm for Infix to postfix Expression

In the below algorithm "I" is an arithmetic Expression written in infix notation and "P" is the equivalent postfix Eqprenion generated by this algorithm.

1. Push "(" left parenthesis onto stack
2. Add ")" right parenthesis to the end of Expression "I"
3. Scan "I" from left to right and repeat Step 4. for each element of "I" until the stack becomes empty
4. If the scanned Element is:
   a) an operand then add it to "P"
   b) a left parenthesis then push it onto stack
   c) An operator then
      i, pop from stack and add to "P" each operator

which has the same or higher precedence then the scanned operator

ii) Add newly scanned operator to stack

d) A right parenthesis then:

i) pop from stack and add to "P" each Operator until a left parenthesis is encountered

ii) Remove the left paranthesis

/*[End of step4 if]

/*[End of step3 for loop]

5) Exit

Examples for Infix to postfix Expression:-

A+B*C+D

① | | +

A

② | * | + |

AB

③ | + | * | + |

ABC

④ | + |

ABC*+

⑤ | + |

ABC*+D+

O/P.

ii) (A+B) * (C+D) → (I/P) AB+CD+* (O/P)

iii) A*B+C*D → AB*CD*+

iv) A+B+C+D → AB+C+D+

→ Evaluation of postfix Expression :-

Example :- AB+C-BA+C$-

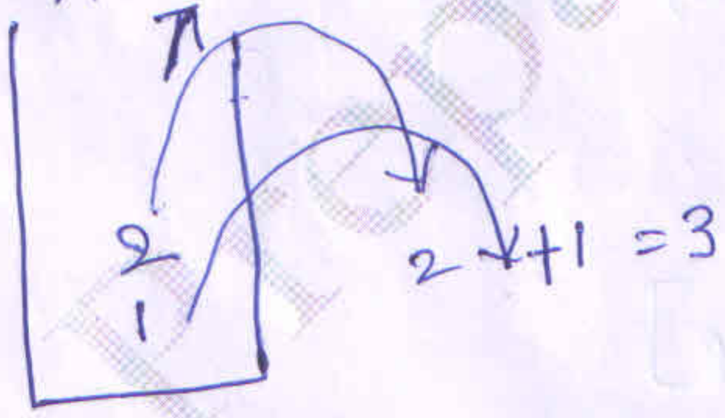
A = 1, B = 2 and C = 3

AB+C-BA+C$-

① AB+C-BA+C$-

② AB+C-BA+C$-

③ AB+C-BA+C$-



④

= 3-3 = 0

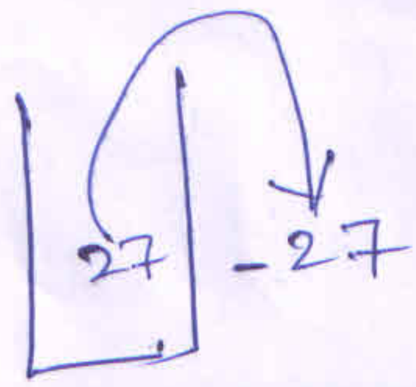
⑤ AB+C-BA+C$-

⑥ Here $ = x^y

$3^3 = 27$

⑧

⑦

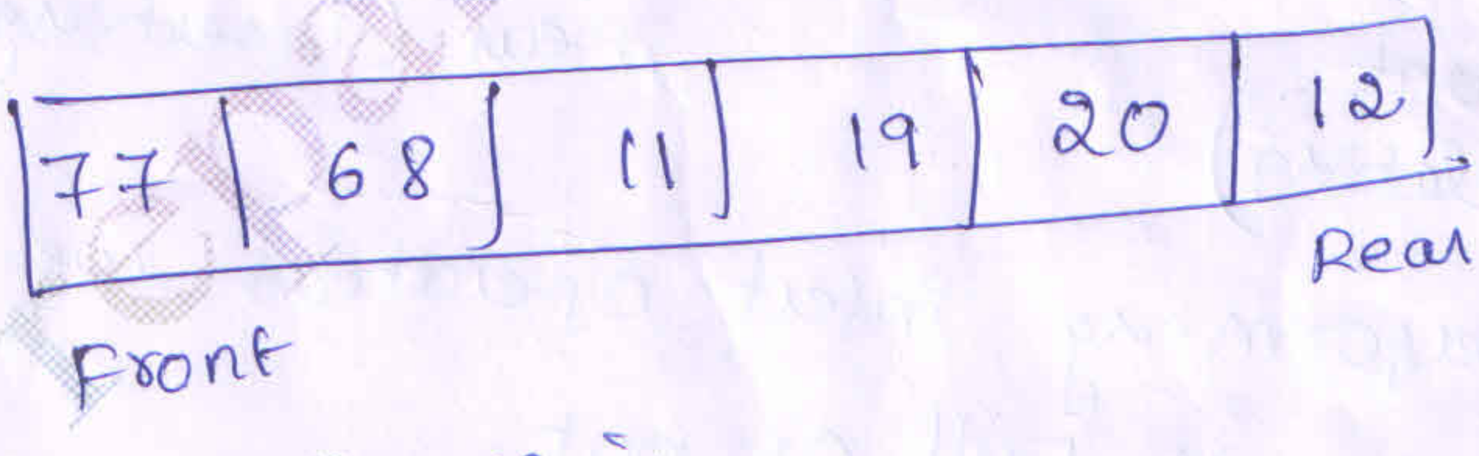


→ Def Queue :- Queue is defined as ordered collection of elements that has 2 ends named as front end and Rear end

* From the front end one can delete the elements
* and from the Rear end one can insert the elements

Example :- people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue

Representing the Queue with few elements as below



→ Operations on Queue :-

Queue is nothing but collection of items. Both ends of the queue are having their own functionalities. The Queue is also called as FIFO (First-in-First-out)

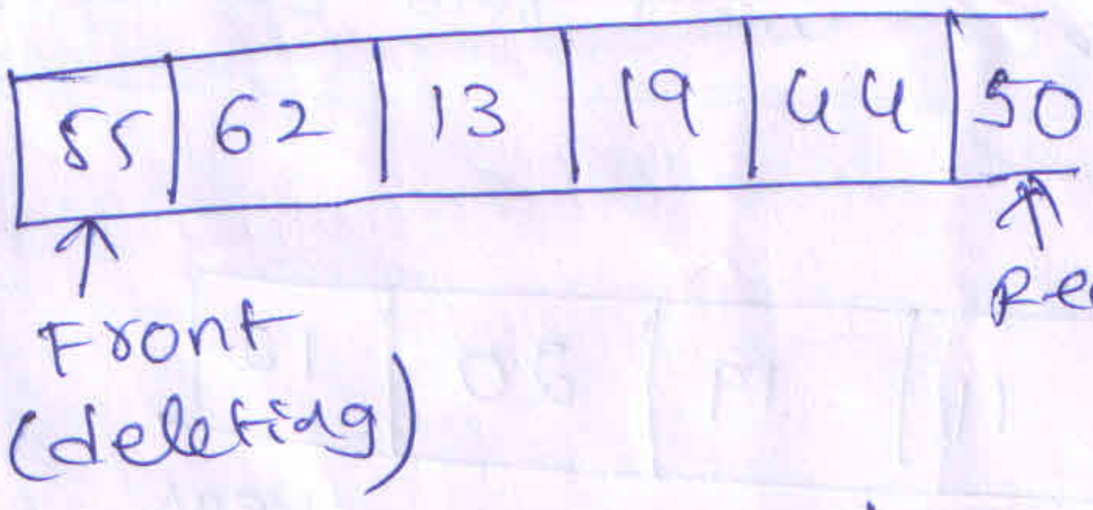
1, Queue overflow 2, Queue underflow
3, Insertion of the element into the Queue
4, Deletion of elements from the Queue 5, Display of Queue

Structure of Queue :-

```
struct Queue
{
    int que[size];
    int front;
    int rear;
} Q;
```

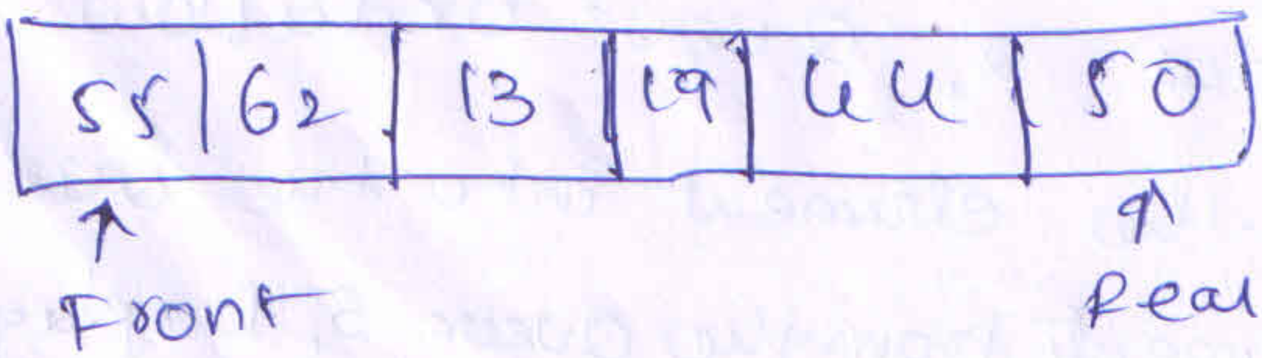
Insertion of element into the Queue :-

Insertion of element in the queue will always takes place from the rear end



Before performing insert operation we must check queue is Full or not

If rear pointer is going beyond the maximum size of the queue then the queue overflow occurs.



→ Algorithm to Insert Item into Queue :

Here QUEUE is an array with "N" locations FRONT and REAR points to the front and rear of the QUEUE. ITEM is the value to be inserted

1. If (REAR == N) then /* check for overflow
2. print: Overflow
3. Else
4. If (FRONT and REAR == 0) then /* check if Queue is empty
   a) Set FRONT = 1
   b) Set REAR = 1
5. Else
6. Set REAR = REAR+1 /* Increment REAR by 1
   */ [End of step 4 if]
7. QUEUE[REAR] = ITEM
8. print: ITEAM inserted
   */ [End of step 1 if]
9. Exit

→ Algorithm to delete Item from Queue :

Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear of the queue.

→ 1. If (FRONT == 0) then /* Check for underflow
2. print: underflow
3. Else
4. ITEM = QUEUE[FRONT]
5. If (FRONT == REAR) then /* check if only one element is left
a). Set FRONT = 0
b). Set REAR = 0
6. Else
7. Set FRONT = FRONT + 1 /* increment FRONT by 1
/* [End of step 5 if]
8. print: ITEM deleted
/* [End of step 1 if]
9. Exit

→ Applications of Queue :-
1, Q's are used in OS & for scheduling the jobs
2, Q's are for categorizing data
3, Q's are ~~for~~ used for simulation and modeling

→ Round Robin Algorithm :- (RR)
RR algorithm is an emporant scheduling algorithm. It is used especially for new time sharing system. The circular Queue is used to implement such algorithms.

For Example there are 'N' procedures or tasks such as $P_1, P_2, P_3 \text{---} P_N$. All these tasks are to be executed by the CPU of the computer system. the execution times of the tasks or processes are different. The tasks are executed in sequence P_1, P_2 P_3 --- and P_N.

In time sharing mode the tasks are executed one by one the algoritem forms a small unit of time say from 10 to 100 milliseconds for each task. This time is called time slice or time quantum of a task. the CPU executes tasks from P_1 to P_N allocating fixed amount of time for each process on allocating time to all tasks, CPU resumes the first task. That is, when all tasks are completed, it returns to P_1. In the time sharing system if any task is completed before the Estimated time, the next task is taken up for execution immediately.

Tasks	Expiry Time (unit)
P_1	10
P_2	19
P_3	8
P4	5

these 4 tasks and the total time to complete all the tasks would be (10+19+8+5) 42 unit

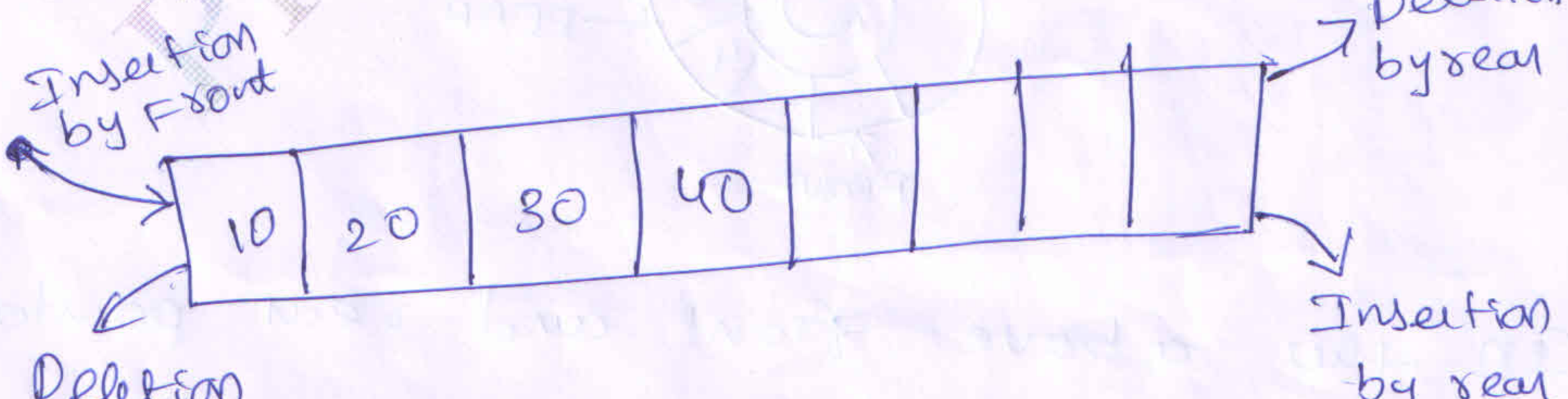
Suppose, the time slice is of 7 unit, the RR scheduling for the above case would be as given below:

In the first pass each task takes 7 unit of time task P4 is executed in the first pass whereas task P_1, P_2, P_3 requires more than 7 unit of time Hence in the second round task P_1 and P_3 are executed at least, P_2 is Executed.

→ Enqueue :- enqueue means to add an item to the queue, generally at the "back" of the queue. this can sometimes be called as "push".

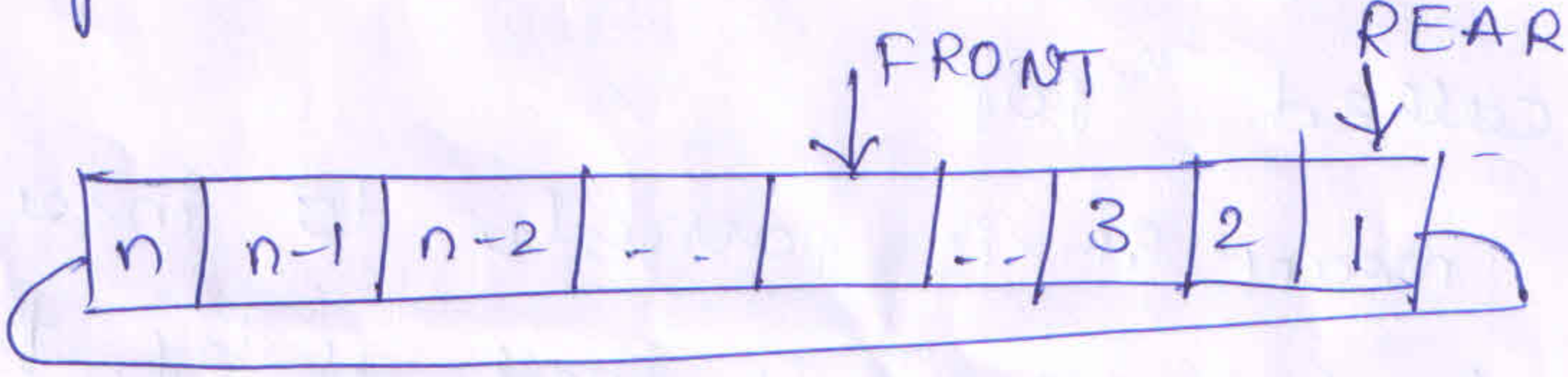
→ dequeue :- dequeue means to remove an item from the queue, generally from the "front" of the queue. this can sometimes be called "pop"

that means it is possible to insert the elements by rear as well as by front. similarly it is possible to delete the elements from front as well as from rear this concept is called double ended Queue

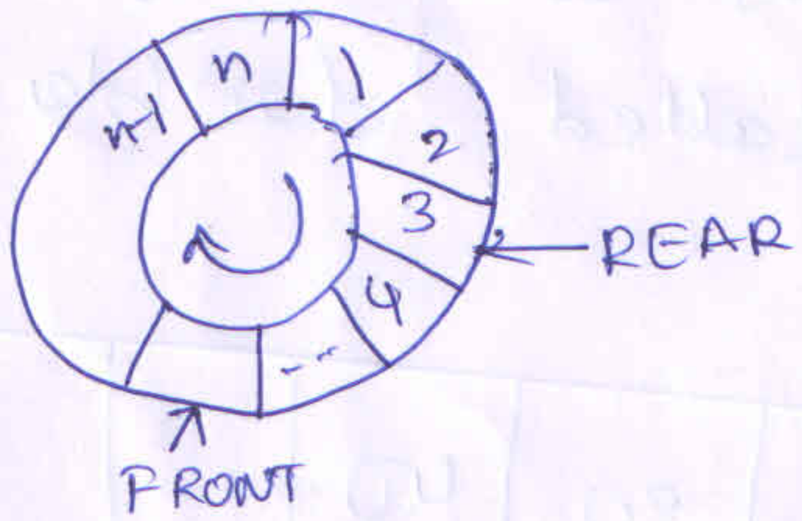


→ Circular Queue :-

Representation of a Queue using an array has the disadvantage that when the rear pointer reaches the end of the array, insertion operation is defined even if there is some space at the front of the array. this advantage can be overcome by using the circular Queue.



Circular Queue using Array representation



In the above front and rear pointer move in the same direction (clockwise) when the Queue is at the i^{th} position, the next position will be found out using the relation $i \bmod (length + 1)$ where length is the maximum of element that can be placed on the Queue

Delete from Circular Queue

Delete Circular( ):

Description : Here QUEUE is an array with N locations. FRONT and REAR points to the front and rear elements of the QUEUE

```
1.  If (FRONT == 0) Then                    /* check for underflow */
2.      print : Underflow
3.  Else
4.      ITEM = QUEUE[FRONT]
5.      If (FRONT == REAR) Then             /* If only element is left */
            a) Set FRONT = 0
            b) Set REAR = 0
6.      Else if (FRONT == N) then           /* if front reaches end if QUEUE */
7.          set FRONT = 1
8.      Else
9.          set FRONT = FRONT + 1           /* increment FRONT by 1 */
        [End of Step 5 if]
10.     print : ITEM deleted
        [End of step 1 if]
11. Exit
```

Circular Queue

Insert into circular queue

Insert circular():

Description: Here queue is an array with N locations. FRONT and REAR points to the front and rear elements of the QUEUE. ITEM is the value to be inserted.

```
1. If (FRONT==1 and REAR==N) or (FRONT == REAR+1) Then
2.    print: Overflow
3.    Else
4:    If (REAR==0) Then                    /*check if queue is empty*/
          a) set FRONT = 1
          b) Set REAR = 1
5.    Else if(REAR==N) Then                /* if REAR reaches end
                                              if queue */
6.        Set REAR = 1
7.    Else
8.        Set REAR = REAR+1                /* increment rear by 1 */
          [End of step 4 if]
9.    Set QUEUE[REAR] = ITEM
10.   print: ITEM inserted
      [End of step 1 if]
11.   Exit
```

→ Priority Queue:

Def:- A priority Queue is a datastructure in which each element has assigned a value called the priority of the element and an element can be inserted (or) deleted not only at the Ends but at any position on the queue

There are 2 types of priority queues are there 1, ascending priority Queue
2, Descending priority Queue.

1, Assending priority Queue:- (APQ)

An APQ is a collection of items in which items can be inserted arbitrarily and from which only the smallest item can be removed

2, Descending priority Queue:- (DPQ)

A DPQ is a collection of items in which items can be inserted arbitrarily and from which only the largest item can be removed

Applications of priority Queue :-

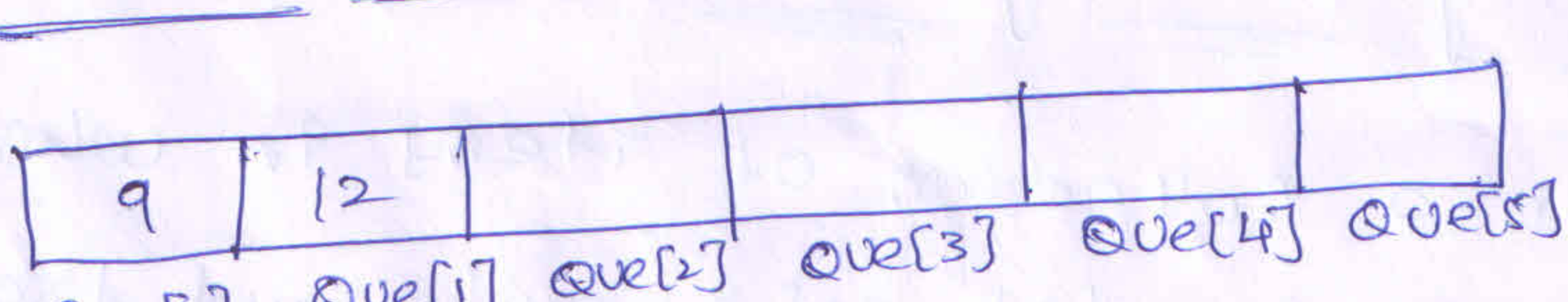
1, The typical example of priority Queue is scheduling the jobs in os. The Jobs are placed in the Queue and position of the Job in priority Queue determines their priority.

2. There 3 kinds of Jobs are there
   * real Time Jobs
   * Foreground Jobs
   * background Jobs.

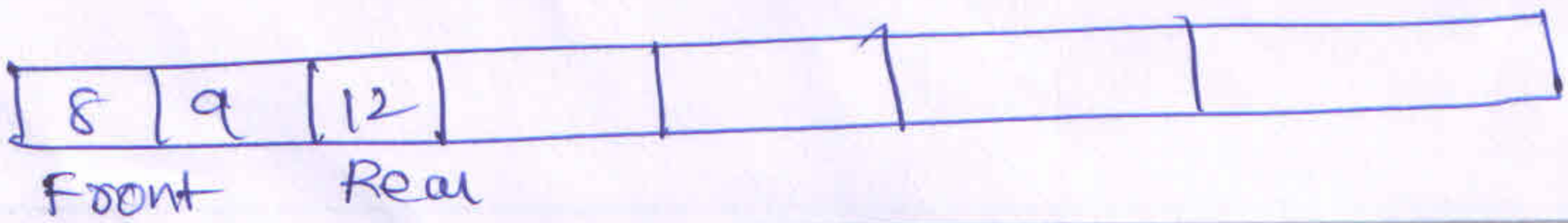
3, In network communication, to manage limited bandwidth for transmission the priority Queue is used

4, In simulation modelling to manage the discrete events the priority Queue is used.

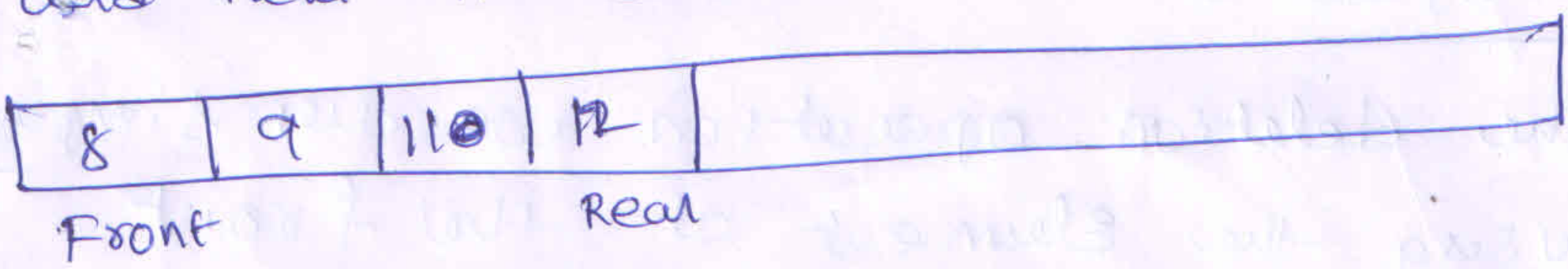
Insertion Operation :-



If we want to insert '8'



and next ith element

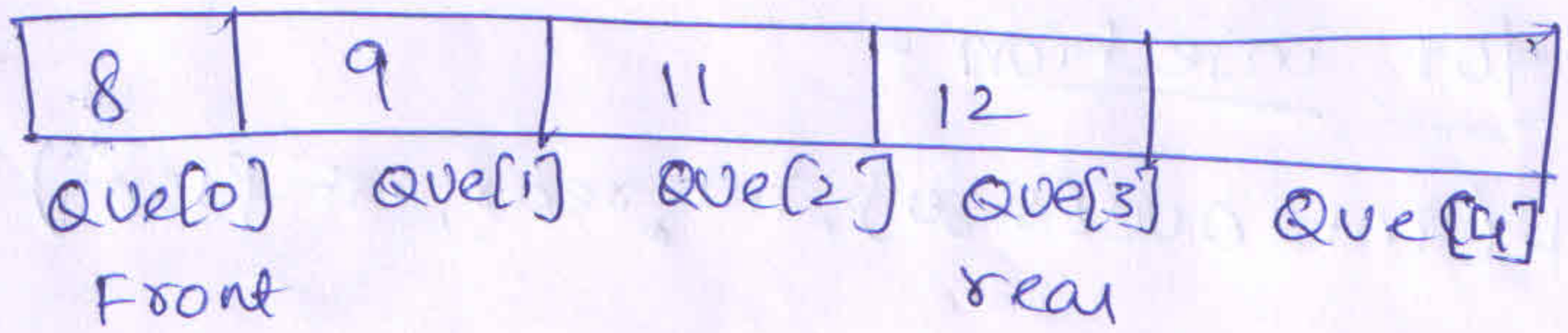


Algorithm for insertion :-

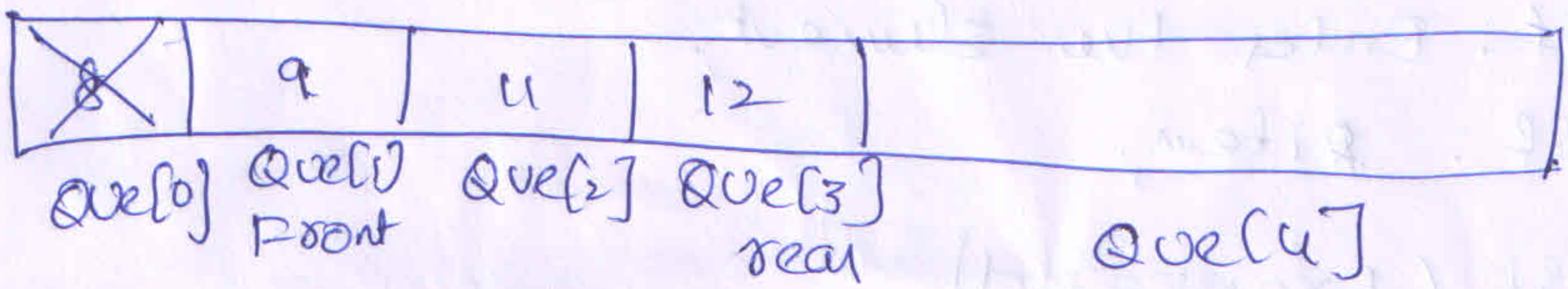
```
int insert(int que[size], int rear, int front)
{
    int item, j;
    pf: Enter the element;
    sf: &item;
    if (front==-1)
        front++;
    j=rear;
    while (j>=0 && item<que[j])
    {
        que[j+1]=que[j];
        j--;
    }
    que[j+1]=item;
    rear=rear+1;
    return rear;
}
```

Deletion operation:

In the deletion operation we are simply removing the element at the front.



then the Element at Que[0] will be deleted



Algorithm for Deletion:

```
int delet(int Que [SIZE], int front)
  {
     int item;
     item = Que[front];
     Pf: The item deleted
       front ++;
     return front;
  }
```

Write a program for Stack using array

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define size 5
struct stack {
  int s[size];
  int top;
} st;
int stfull() {
  if (st.top >= size - 1)
    return 1;
  else
    return 0;
}
void push(int item) {
  st.top++;
  st.s[st.top] = item;
}
int stempty() {
  if (st.top == -1)
    return 1;
  else
    return 0;
}
int pop() {
  int item;
  item = st.s[st.top];
  st.top--;
  return (item);
}
void display() {
  int i;
  if (stempty())
    printf("\nStack Is Empty!");
  else {
    for (i = st.top; i >= 0; i--)
        printf("\n%d", st.s[i]);
  }
}
int main() {
  int item, choice;
  char ans;
  clrscr();
  st.top = -1;
printf("\n\tImplementation Of Stack");
do {
  printf("\nMain Menu");
  printf("\n1.Push \n2.Pop \n3.Display \n4.exit");
  printf("\nEnter Your Choice");
  scanf("%d", &choice);
  switch (choice) {
  case 1:
      printf("\nEnter The item to be pushed");
      scanf("%d", &item);
```

```
        if (stfull())
          printf("\nStack is Full!");
        else
          push(item);
        break;
case 2:
        if (stempty())
          printf("\nEmpty stack!Underflow !!");
        else {
          item = pop();
          printf("\nThe popped element is %d", item);
        }
          break;
        case 3:
              display();
              break;
        case 4:
              exit(0);
        }
        printf("\nDo You want To Continue?");
        ans = getche();
    } while (ans == 'Y' || ans == 'y');
return 0;
}
```

WRITE C PROGRAM THAT IMPLEMENTS STACK OPERATIONS USING LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int Data;
    struct Node *next;
}*top;
void popStack()
{
    struct Node *temp, *var=top;
    if(var==top)
    {
        top = top->next;
        free(var);
    }
    else
    printf("\nStack Empty");
}
void push(int value)
{
    struct Node *temp;
    temp=(struct Node *)malloc(sizeof(struct Node));
    temp->Data=value;
    if (top == NULL)
    {
        top=temp;
```

```
        top->next=NULL;
    }
    else
    {
        temp->next=top;
        top=temp;
    }
}
void display()
{
    struct Node *var=top;
    if(var!=NULL)
    {
        printf("\nElements are as:\n");
        while(var!=NULL)
        {
            printf("\t%d\n",var->Data);
            var=var->next;
        }
    printf("\n");
    }
    else
    printf("\nStack is Empty");
}
int main(int argc, char *argv[])
{
int i=0;
top=NULL;
printf(" \n1. Push to stack");
printf(" \n2. Pop from Stack");
printf(" \n3. Display data of Stack");
printf(" \n4. Exit\n");
while(1)
{
    printf(" \nChoose Option: ");
    scanf("%d",&i);
    switch(i)
    {
        case 1:
        {
        int value;
        printf("\nEnter a valueber to push into Stack: ");
scanf("%d",&value);
push(value);
display();
break;
}
case 2:
{
popStack();
display();
break;
}
case 3:
```

```
{
display();
break;
}
case 4:
{
struct Node *temp;
while(top!=NULL)
{
    temp = top->next;
    free(top);
        top=temp;
        }
        exit(0);
        }
        default:
        {
        printf("\nwrong choice for operation");
        }
    }
  }
}
```

WRITE A C PROGRAM THAT USES STACK OPERATIONS TO CONVER INFIX EXPRESSION INTO POSTFIX EXPRESSION

```
#include<stdio.h>
#include<string.h>
typedef struct stack
 {
  char data[20];
  int top;
 }st;
void push(st *s,char x)
 {
 s->data[++s->top]=x;

}
char pop(st *s)
 {
   return(s->data[s->top--]);
 }
int pred(char);
int isoprtor(char);
int isoprnd(char);
void main()
{
 st s;
 char s1[20], post[20]="(",ch;
 int i,j=-1;
 s.top=-1;
 printf("\n enter infix notation:");
 gets(s1);
```

```
strcat(post,s1);
strcpy(s1,post);
strcat(s1,")");
strcpy(post,"");
for(i=0;s1[i]!='\0';i++)
{
  if(isoprnd(s1[i]))
  post[++j]=s1[i];
  else if(isoprtor(s1[i]))
    {
     while((pred(s1[i]))<=(pred(s.data[s.top])))
     post[++j]=pop(&s);
     push(&s,s1[i]);
    }
   else if(s1[i]=='(')
   push(&s,s1[i]);
   else if(s1[i]==')')
    {

     ch=pop(&s);
     while(ch!='(')
      {
        post[++j]=ch;
        ch=pop(&s);
      }
    }
}
  post[++j]='\0';
  printf("\n\npostfix notation is :%s",post);
}
int isoprtor(char ch)
{
    if(ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='%')
      return 1;
    return 0;
}
int pred(char ch)
{
    if(ch=='-'||ch=='-')
      return 1;
    else if(ch=='(')
      return 0;
    return 2;
}
int isoprnd(char ch)
{
    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z')||(ch>='0'&&ch<='9'))
       return 1;
    return 0;
}
```

WRITE A PROGRAM THAT IMPLEMENT QUEUE OPERATIONS USING ARRAYS

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
void insert(int);
int del();
int queue[MAX], rear=0, front=0;
void display();
int main()
{
 char ch , a='y';
 int choice, token;
 printf("1.Insert");
 printf("\n2.Delete");
 printf("\n3.show or display");
 do
 {
     printf("\nEnter your choice for the operation: ");
     scanf("%d",&choice);
    switch(choice)
    {
        case 1: insert(token);
        display();
        break;
        case 2:
        token=del();
        printf("\nThe token deleted is %d",token);
        display();
        break;
       case 3:
        display();
        break;
        default:
        printf("Wrong choice");
        break;
}
printf("\nDo you want to continue(y/n):");
    ch=getch();
   }
   while(ch=='y'||ch=='Y');
   getch();
}
void display()
{
   int i;
 printf("\nThe queue elements are:");
```

```
    for(i=rear;i<front;i++)
    {
        printf("%d ",queue[i]);
    }
}
void insert(int token)
{
    char a;
    if(rear==MAX)
    {
        printf("\nQueue full");
        return;
    }
    do
    {
        printf("\nEnter the token to be inserted:");
        scanf("%d",&token);
        queue[front]=token;
        front=front+1;
        printf("do you want to continue insertion Y/N");
        a=getch();
    }
    while(a=='y');
}
int del()
{
    int t;
    if(front==rear)
    {
        printf("\nQueue empty");
        return 0;
    }
    rear=rear+1;
    t=queue[rear-1];
    return t;
}
```

WRITE A PROGRAM THAT IMPLEMENT QUEUE OPERATIONS USING LINKED LISTS

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;
int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();
int count = 0;
void main()
{
   int no, ch, e;
  printf("\n 1 - Enque");
   printf("\n 2 - Deque");
 printf("\n 3 - Front element");
   printf("\n 4 - Empty");
printf("\n 5 - Exit");
printf("\n 6 - Display");
printf("\n 7 - Queue size");
create();
while (1)
{
   printf("\n Enter choice : ");
   scanf("%d", &ch);
   switch (ch)
   {
   case 1:
      printf("Enter data : ");
      scanf("%d", &no);
      enq(no);
      break;
 case 2:
    deq();
    break;
case 3:
  e = frontelement();
   if (e != 0)
      printf("Front element : %d", e);
   else
```

```
    count++;
}
/* Displaying the queue elements */
void display()
{
    front1 = front;
    if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty");
        return;
    }
    while (front1 != rear)
    {
        printf("%d ", front1->info);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("%d", front1->info);
}
 /* Dequeing the queue */
void deq()
{
    front1 = front;
    if (front1 == NULL)
    {
        printf("\n Error: Trying to display elements from empty queue");
        return;
    }
    else
        if (front1->ptr != NULL)
        {
            front1 = front1->ptr;
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = front1;
        }
        else
        {
            printf("\n Dequed value : %d", front->info);
            free(front);
            front = NULL;
            rear = NULL;
        }
        count--;
}
/* Returns the front element of queue */
int frontelement()
```

```
{
   if ((front != NULL) && (rear != NULL))
      return(front->info);
   else
      return 0;
}
/* Display if queue is empty or not */
void empty()
{
   if ((front == NULL) && (rear == NULL))
     printf("\n Queue empty");
   else
    printf("Queue not empty");
}
```