→ Hello World

```
#include<stdio.h>
int main(){
    printf("Hello World!");
}
```

→ Dynamic memory allocation

1. malloc() method

The malloc is a method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax-

```
ptr=(cast-type*) malloc(byte-size)
```

Example-

```
ptr=(int*) malloc(100*sizeof(int));
```

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *ptr;
    int n,i;
    printf("Enter the number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements:%d\n",n);

    ptr=(int*) malloc(n*sizeof(int));
    if(ptr==NULL){
        printf("Memory not allocated");
        exit(0);
    }
```

```
else{
  printf("Memory successfully allocated using malloc");
  for(i=0; i<n; i++){
    ptr[i]=i+1;
  }
  printf("The elements of the array are");
  for(i=0; i<n; i++){
    printf("%d", ptr[i]);
  }
}
return 0;
}
```

2. calloc() method

calloc method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It is very much similar to malloc() but has two different points.

i) It initializes each block with default value 0.

ii) It has two parameters or arguments as compare to malloc.

Syntax-

```
ptr=(casttype*) calloc(n, elementsize);
```

Example-

```
ptr=(float*) calloc(25, sizeof(float));
```

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int *ptr;
  int n,i;
```

```
printf("Enter the number of element");
scanf("%d", &n);
printf("Entered number of elements is %d", n);

ptr = (int*) calloc (n, sizeof(int));
if(ptr == NULL){
    printf("Memory not allocated");
    exit(0);
}
else{
    printf("Memory allocated successfully using calloc");
    for(i=0; i<n; i++){
        ptr[i] = i+1;
    }
    printf("The elements of the array are");
    for(i=0; i<n; i++){
        printf("%d", ptr[i]);
    }
}
return 0;
}
```

→ Realloc method

realloc method in C is used to change the memory allocation of a previously allocated memory. If the memory is previously allocated with the help of malloc or calloc dynamically is insufficient realloc can be used to dynamically realloc memory. Reallocation of memory maintains the already present value and new block will be initialized with the default garbage value.

Syntax- ptr = realloc(ptr, newSize);

ptr = realloc(ptr, 10 * sizeof(int));

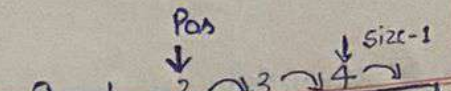
```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  int *ptr;
  int n, i;
  n=5;
  printf("Enter number of elements: %d\n", n);
  ptr= (int*) calloc(n, sizeof(int));
  if (ptr == NULL){
    printf("Memory not allocated\n");
    exit(0);
  }
  else{
    printf("Memory successfully allocated using calloc\n");
    for(i=0; i<n; i++){
      ptr[i]= i+1;
    }
    printf("The elements in the array are: ");
    for(i=0; i<n; i++){
      printf("%d,", ptr[i]);
    }
    n=10;
    printf("\n Enter the new size of the array: %d\n", n);
    ptr= (int*) realloc(ptr, n * sizeof(int));
    printf("Memory successfully reallocated using realloc\n");
    for(i=5; i<n; i++){
      ptr[i]= i+1;
    }
    printf("The elements in the array are: ");
    for(i=0; i<n; i++){
      printf("%d,", ptr[i]);
    }
  }
return 0;
}
```

→ Free method

free method is used in c to dynamically de-allocate the memory. The memory allocated using malloc & calloc function is not deallocated on their own. Hence the free method is used to when ever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it

Syntax- free(ptr);

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *ptr,*ptr1;
    int n,i;
    n=5;
    printf("Enter the number of elements: %d\n",n);
    ptr=(int*)malloc(n*sizeof(int));
    ptr1=(int*)calloc(n,sizeof(int));
    if(ptr==NULL||ptr1==NULL){
        printf("Memory not allocated\n");
        exit(0);
    }else{
     printf("Memory successfully allocated using malloc\n");
     free(ptr);
     printf("Malloc memory successfully freed\n");
     printf("Memory successfully allocated using calloc\n");
     free(ptr1);
     printf("Calloc memory successfully free\n");
    }
return 0;
}
```



Insertion

```
int a[50], size, i, num, pos;
printf("Enter the size of array");
scanf("%d", &size);
printf("Enter the elements of array");
for(i=0; i<size; i++){
    scanf("%d", &a[i]);
}
printf("Enter data you want to insert");
scanf("%d", &num);
printf("Enter position:");
scanf("%d", &pos);
for(i=size-1; i>=pos-1; i--){
    a[i+1] = a[i];
}
a[pos-1] = num;
size++;
for(i=0; i<size; i++){
    printf("%d", a[i]);
}
```

Worst case - O(n)
O(n-p)

Best - O(1)

i=3
a[5] = a[4]
a[4] = a[3]

Insertion at beginning

```
for(i=size-1; i>=0; i--){
    a[i+1] = a[i]
}
a[0] = num;
size++;
```

0

Best case

```
a[size] = a[pos-1]
a[pos-1] = num
```

→ Arrays in data structure

int, float, char

int a[60];

- Array is a collection of more than one data item but the datatype of those element is same

char b[10];

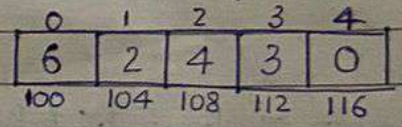
float c[5];

- All the data will be stored in a consiquctive memory locati

- Arrays can be initialized in 2 ways:-

at compile time (static) int a[5]={6,2,4,3,0};

at run time (dynamic)



i=0 (n-1) i=2;

Formula = base addrus + i * size of datatype

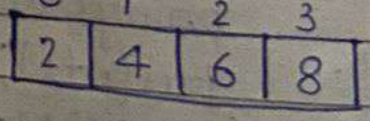
= 100 + 2 * 4

= 108

Operations on array

Traversal

```
int a[50], size;
printf("Enter size of array");
scanf("%d", &size);
printf("Enter elements of array");
for(int i=0; i<size; i++){
    scanf("%d", &a[i]);
}

printf("Elements in array are");
for(int i=0; i<size; i++){
    printf("%d", a[i]);
}
```



- Delete data from specific position

0	1	2	3	4
6	2	10	4	5

Pos = 2

```
printf("from which position you want to delete");
scanf("%d", &pos);
if(pos<=0 || pos > size)
    printf("invalid position");
else
for(i=pos-1; i<size-1; i++){
    a[i] = a[i+1];
}
size--;
for(i=0; i<size; i++){
  printf("%d", a[i]);
}
delete at beginning
for(i=0; i<size-1; i++){
  a[i] = a[i+1]
}
size--;
```

Unsorted array best case O(1)

~~a[size-1~~

```
a[pos-1] = a[size-1]
size--;
```

→ Array & pointers

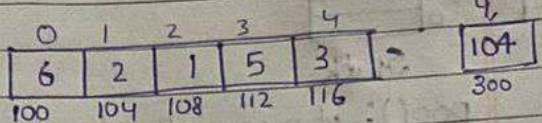
```
int a[] = {6,2,1,5,3};
int *p; it is a pointer to integer
int b=10;
p=&b;
printf("%d", b);
```

b	p
10	200
200	204

newNode = ---
printf("data")
newNode->next=NULL;
newNode->...

↱ value at the address of p variable

```
printf("%d", *p); //10
printf("%p", &b); //200
printf("%p", p); //200
```




```
printf("%d", a[2]); //1
          *(a+2); //1
          *(q+1); //1
```

-
```
int a[5], i;
int *q=a;
for(i=0; i<5; i++){
   scanf("%d", &a[i]); OR (a+i)
}
for(i=0; i<5; i++){
   printf("%d", a[i]), *(q+i)
}
```

→ Stack

Linear data structure.

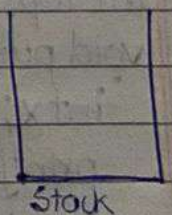
It is a ordered list which is going to follow a rule of insertion & deletion

Rule [Insertion, Deletion] It is possible it only from one end.

LIFO, FILO

push, pop, top O(1)

Operations display O(n)



push(x); // Insertion of an element

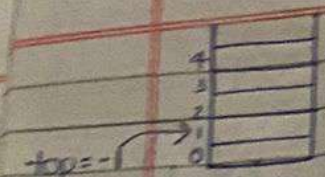
pop(); // deletion

peek()/top(); // it return top most element of stack without removing that element from the stack.

isEmpty();

isFull();

```
        top++;
        stack[top] = x;
    }
}
void pop(){
    int item;
    if(top == -1)
        printf("Underflow");
    else{
        item = stack[top]
        top--;
    }
    printf("the popped item is %d", item);
}
void peek(){
    if(top == -1)
        printf("stack is empty");
    else{
        printf("Topmost element is %d", stack[top]);
    }
}
void display(){
    int i;
    for(i=top; i>=0; i--){
        printf("%d", stack[i]);
    }
}
void main(){
    int ch;
    do{
        printf("Enter choice 1 push 2 pop 3 peek 4 display")
        scanf("%d", &ch);
```



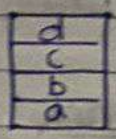
pop(); it means the top most element will be removed but our stack is empty so it shows an underflow condition.

push(2);
top++; first we have to increment the top then insert the 2 -> element

push(3);
top++;
pop();
top--;

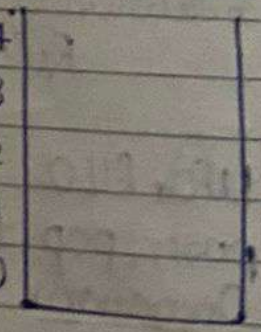
push(1)
push(5)
push(6)
push(7)
push(8) - overflow condition

Application of stack
Reverse a string abcd - dcba
Undo mechanism
Infix to postfix/prefix
stack using



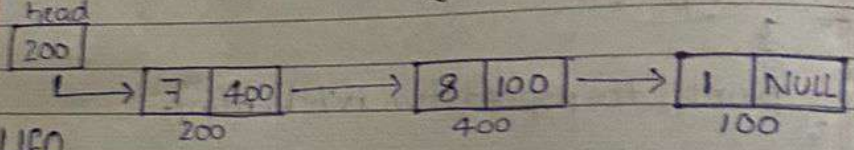
Implement array

```
#define N 5;
int stack[N];
int top=-1;
void push(){
  int x;
  printf("Enter the value");
  scanf("%d", &x);
  if(top==N-1)
     printf("Overflow");
else{
```



```
switch(ch){
  case1: push();
         break;
  case2: pop();
         break;
  default: printf("Invalid choice");
}
while(ch!=0);
}
```

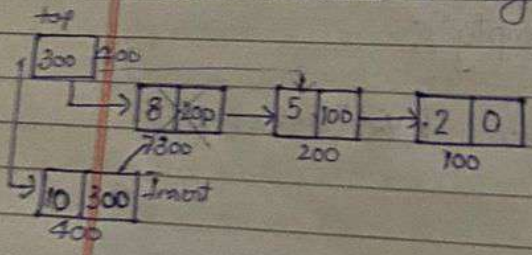
Implementation stack using linkedlist

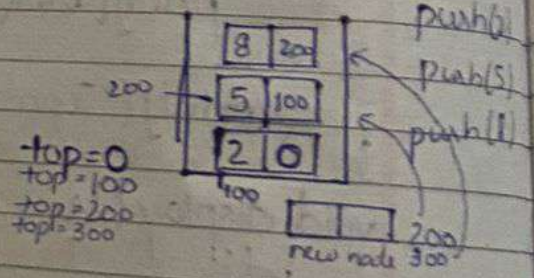


LIFO

Insertion & deletion at end time complexisity is O(n)

Insertion & deletion at beginning time comptex is O(1)



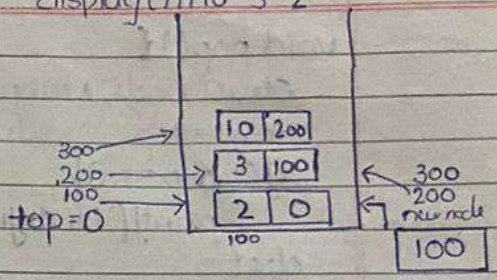



```
void main(){
  push(2); push(3); push(10);
  display();
  peek(); pop(); peek();
  display();
}
```

```
};
newNode = ---
printf("data")
newnode->next=NULL;
newNode->next = temp->next;
temp->next = .
```

PAGE NO.
DATE

display() 10 3 2




```
struct node{
    int data;
    struct node *link;
};
struct node *top=0;
void push(int x){
    struct node *newnode;
    newnode=(struct node*) malloc (sizeof(struct node));
    newnode -> data = x;
    newnode->next = top;
    top = newnode;
}
void display(){
    struct node *temp;
    temp = top
    if (top == NULL)
        printf("Linked list is empty");
    else{
        while(temp != NULL) {
            printf("%d", temp->data);
            temp = temp -> link;
        }
}
}
void peek(){
    if (top==NULL)
        printf("Stack is empty");
    else{
        printf("top element is %d", top->data);
    }
}
```

newNode = ____
printf("data")
newNode->next = NULL;
newNode->next = temp->next;
temp->next = n. n.

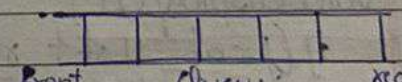
PAGE No.
DATE

Prefix +51
<operator> <operand> <operand>
a * b + c => *ab + c => ++abc
Postfix
<operand> <operand> <operator>
5 + 1 => 51 +
a * b + c => ab * + c => ab * c +

- Queue
FIFO LILO
Queue is a ordered list which follow FIFO Rear/tail
Rule insertion (it is performed by one end that called
deletion (it is performed on another end that is head/front
enqueue() - insertion
dequeue() - deletion

Front → ____ ← Rear

Operations dequeue enqueue
Enqueue(2)
Dequeue() - always dequ the first element from the front.
front() / peek() - what is the element is front in the queue.
isFull() - it will return true if the queue is full else false
isEmpty() -



Front = rear = -1

Application
printer
call centers
- Implementation:
#define N 5;
int queue[N];
int front = -1;
int rear = -1;

```
void pop(){
  struct node *temp;
  temp=top;
  if(temp==0)
    printf("Underflow");
  else{
    printf("the pop element is %d",top->data);
    top=top->link;
    free(temp);
  }
}
```

- Infix prefix postfix
  5+1 p+q a+5
  <operand><operator><operand>

Infix (a-1)+5
operand itself may be a expression

* When the operator is between operands so this form is known as infix expression

Example → 5+1*6 [6×6=36
                [5+6=11 ✓ ← associativity.

Infix expression follows operator precedence rule.

1. ()
2. ^ R-L
3. */ L-R
4. +- L-R

5+1*6 = 5+6 = 11
(5+1)*6 = 36

Example 1+2*5+30/5
→ ⇒ 1+10+30/5
⇒ 1+10+6
⇒ 11+6
⇒ 17

2^2^3
2^8 ←
=256

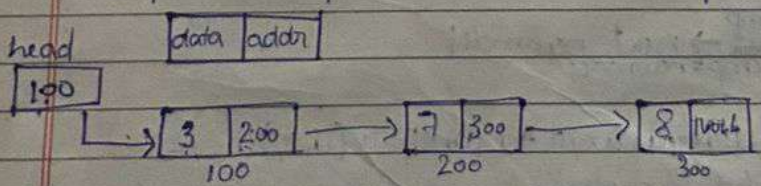
```
void peek(){
  if(f==-1 && r==-1)
    printf("Overflow");
  else{
    printf("%d", queue[front]);
  }
}
```

→ LinkedList

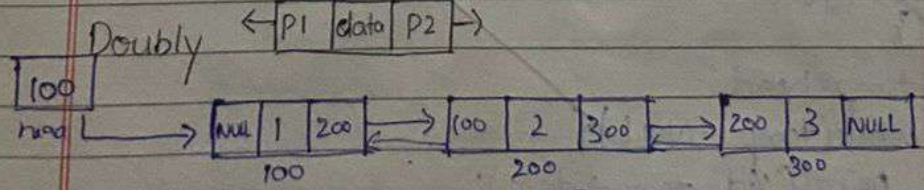
- It is also the collection of any data items and that are not stored in consiquite location.
- Some extra space is required to store the pointers with each value.
- Insertion & deletion is easier than array.
- Accessing of any element is going to take O(n) & in array it is O(1).
- Binary search is not possible in linked list.
- It is of dynamic size.

Types of linked list

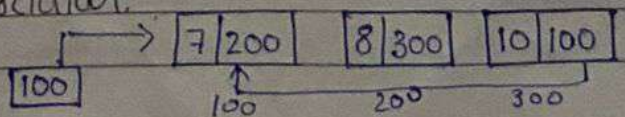
Singly Doubly Circular Doubly circular



Doubly

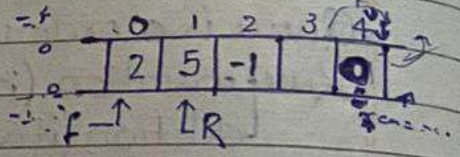


Circular

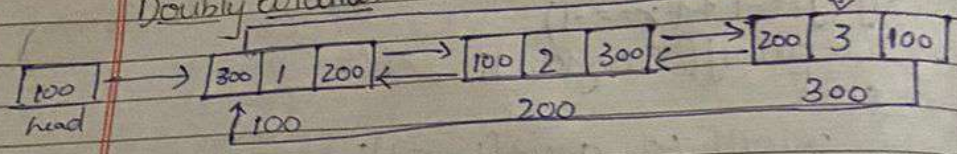


```
void enque(int x){
  if(rear == N-1)
    printf("Overflow");
  else if(front == -1 && rear == -1){
    front = rear = 0;
    queue[rear] = x;
  }
  else{
    rear++;
    queue[rear] = x;
  }
}

void deque(){
  if(front == -1 && rear == -1)
    printf("Underflow");
  else if(front == rear){
    front = rear = -1;
  }else{ printf("The deque element is %d", queue[front]);
    front++;
  }
}

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

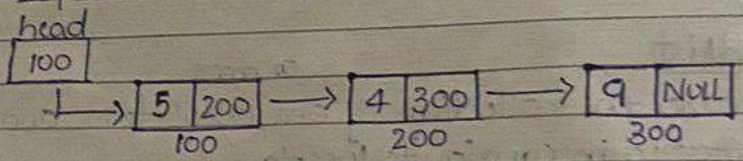
```
void main(){
  enqueue(2)
  enqueue(5)
  enqueue(-1)
  display()
  peek()
  dequeue()
  peek()
  display()
}
```



Doubly circular



Implemantation of linkedlist




```
struct node {
   int data;
   struct node *next;
};
Struct node *head; *newnode; *temp;
head=0; int choice, while(choice){
newnode head = (Struct node *) malloc (sizeof(struct node));
Scanf("%d", &newnode->data);
newnode->next=0;
if(head==0)
   head=newnode; temp=newnode;
else{
   head->next=newnode;
   temp=newnode;
}
printf("Do you want to continue (0,1));
Scanf("%d", &choice);
}
temp=head;
while(temp!=NULL){
 printf("%d", temp->data);
 temp=temp->next;
}
```

```
#include<stdio.h>
#include<stdlib.h>
void display();
struct node{
    int data;
    struct node *next;
};
struct node *head=NULL, *newnode, *temp;
int choice;
void insertionAtBeginning(){
    if(head==NULL)
        printf("No memory allocated");
    else{
        newNode=(struct node *)malloc(sizeof(struct node));
        printf("\nEnter the data you want to insert");
        scanf("%d",&newNode->data);
        newNode->next=head;
        head=newNode;
        display(head);
    }
}
void insertionAtMiddle(){
    int pos, i=1;
    printf("\nEnter the position you want to insert data:");
    scanf("%d",&pos);
    if(head==NULL){
        printf("\nNo memory allocated.");
    }else{
        newNode=(struct Node*)malloc(sizeof(struct node));
        temp=head;
```

newnode->next=NULL;
newNode->next=temp->next;
temp->next=

PAGE No.
DATE

- Implementation of singly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node{
  int data;
  struct node *next;
};
void main(){
  struct node *head=NULL, *newnode, *temp;
  int choice, count=0;
  while(choice){
    newnode=(struct node*) malloc(sizeof(struct node));
    printf("Enter data in a linked list: ");
    scanf("%d", &newnode->data);
    newnode->next=NULL;
    if(head==NULL){
      head=temp=newnode
    }else{
      temp->next=newnode;
      temp=newnode;
    }
    printf("Do you want to enter another element 1: continue
    scanf("%d", &choice);                          0:exit);
  }
printf("Elements of linked list are");
temp=head;
while(temp!=NULL){
  printf("%d", temp->data);
  temp=temp->next;
} count++;
printf("Total no of elements in linked list are %d", count);
}
```

```
        while(i<pos){
            temp = temp->next;
            i++;
        }
        printf("\nEnter the data you want to insert");
        scanf("%d", &newNode->data);
        newNode->next = temp->next;
        temp->next = newNode;
    }
    display(head);
}
void insertionAtEnd(){
    if(head == NULL){
        printf("\n No memory allocated");
    }
    else{
        newNode = (struct node*) malloc(sizeof(struct node));
        printf("\nEnter the data you want to insert:");
        scanf("%d", &newNode->data);
        temp = head;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = NULL;
    }
    display(head);
}
```

```
void deletionAtBeginning(){
    temp=head;
    head=temp->next;
    free(temp);
    display(head);
}
void deletionAtMiddle(){
    int pos, i=1;
    printf("Enter the position you want to delete");
    scanf("%d", &pos);
    struct node *nextNode;
    temp=head;
    while(i<pos-1){
        temp=temp->next;
        i++;
    }
    nextNode=temp->next;
    temp->next=nextNode->next;
    free(nextNode);
    display(head);
}
void deletionAtEnd(){
    struct node *preNode;
    temp=head;
    while(temp->next!=NULL){
        preNode=temp;
        temp=temp->next;
    }
    preNode->next=NULL;
    free(temp);
    display(head);
}
```

```
void display(struct node *head){
    int count=0;
    printf("Elements in the linked list are ");
    temp=head;
    while (temp !=NULL) {
        printf("%d", temp->data);
        temp=temp->next;
        count++;
    }
    printf("\nTotal number of elements in the linked
        list are %d", count);
}
void clearLinkedList(){
    while(head !=NULL){
        temp=head;
        head=head->next;
        display(head);
        free(temp);
    }
}
void main(){
    int num;
    printf("Enter the number of elements you want to
        enter in a linked list");
    scanf("%d", &num);
    for(int i=1; i<=num; i++){
        newNode=(struct node*) malloc(sizeof(struct node));
        printf("Enter %d element :", i);
        scanf("%d", &newNode->data);
        newNode->next=NULL;
```

```
        if(head==NULL){
            head=temp=newNode;
        }
        else{
            temp->next=newNode;
            temp=newNode;
        }
    }
    display(head);
    do{
     printf("\n Select an operation: \n\t 1. Insertion at
          beginning \n\t 2. Insertion at specific position
          \n\t 3. Insertion at end \n\t 4. Deletion from
          beginning \n\t 5. Deletion at specific position \n\t
          6. Deletion last node \n\t 7. Exit \n\t 8. Display
          \n\t 9. Clear linked list");
     printf("\n Enter the choice");
     scanf("%d", &choice);
     switch(choice){
     case 1:
        insertionAtBeginning();
        break;
     case 2:
        insertionAtMiddle();
        break;
     case 3:
        insertionAtEnd();
        break;
     case 4;
        deletionAtBeginning;
        break;
```

```
        case 5:
            deletionAtMiddle();
            break;
        case 6:
            deletionAtEnd();
            break;
        case 7:
            exit(0);
            break;
        case 8:
            clearLinkedList();
            break;
        default:
            printf("Invalid option");
        }
    } while(choice != 9);
}
```

newNode->next = NULL;
newNode->next = temp->next;
temp->next = ...

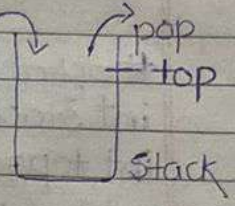
~~Python~~

PAGE No.
DATE

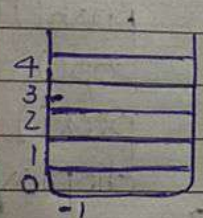
→ Introduction to stack

- It is a linear data structure

Rule- Insertion & deletion is possible only from one end.
Last in first out (LIFO)
First in last out (FILO)



Operations performed on stack
- push(x)
pop()
peek() / top()
isEmpty()
isFull()



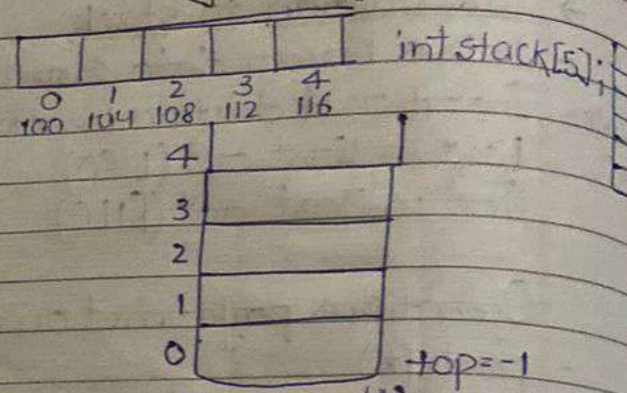
pop() - underflow condition
push(2)
└ top++; then insert 2
push(3)
└ top++; then insert 3
pop()
└ 3 will remove top--

push(1)
push(5)
push(6)
push(7)
push(8) - overflow condition

Application - reverse a string
undo
Recursion
Infix postfix prefix

→ Implementation of stack using arrays

int a[5];




```
#define size=5;
int stack[size];
int top=-1;
- void push(){ int x;
   printf("Enter data you want to insert");
   scanf("%d",&x);
   if(top==size-1){
      printf("Overflow");
   }else{
      top++;
      stack[top]=x;
   }
}

- void pop(){
   int item;
   if(top==-1){
      printf("Underflow");
   }else{
      item=stack[top];
      top--;
      printf("The popped item is %d", item);
   }
}
```

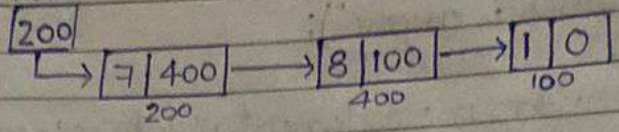
```
void main(){
   push()
   pop()
   peek()
   display()
}
```

newNode->next = temp->next;
temp->next = newNode;

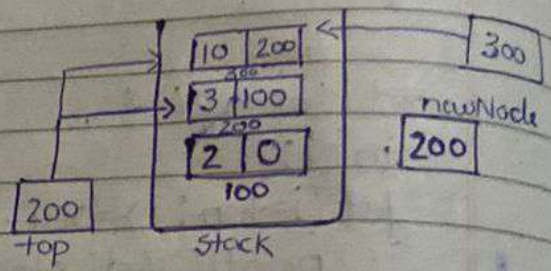
all operation done O(1)

```
void peek(){
  if(top==-1){
    printf("Stack is empty");
  }else{
    printf("The top most element is %d", stack[top]);
  }
}
void display(){
  int i;
  for(i=top; i>=0; i--){
    printf("%d", stack[i]);
  }
}
void main(){
  int ch;
  do{
    printf("Enter choice 1: Push 2: Pop 3: Peek 4: display");
    scanf("%d", &ch);
    switch(ch){
      case 1:
        push(); break;
      case 2:
        pop(); break;
      case 3:
        peek(); break;
      case 4:
        display(); break;
      default:
        printf("Invalid choice");
    }
  }while(ch!=0);
}
```

- Implementation of stack using linked list




```
struct node{
   int data;
   struct node *link;
};
struct node *top=0;
void push(int x){
   struct node *newNode;
   newNode=(struct node*) malloc (sizeof(struct node));
   newNode->data=x;
   newNode->next=top;
   top=newNode;
};
void display(){
  struct node *temp;
  temp=top;
  if(top==NULL){
     printf("Stack is empty");
  }else{
    while(temp!=NULL){
      printf("%d",temp->data);
      temp=temp->link;
    }
  }
}
```




```
void main(){
   push(2)
   push(3)
   push(10)
   display()
   peek()
   pop()
   peek()
   display()
};
```

```
void peek(){
  if(top == NULL){
    printf("Stack is empty");
  }else{
    printf("Top element is %d", top->data);
  }
}
void pop(){
  struct node *temp;
  temp=top;
  if(top == NULL){
    printf("Underflow");
  }else{
    printf("The popped element is %d", top->data);
    top=top->link;
    free(temp);
  }
}
```

→ Infix prefix postfix

1	() [] {}	
2	^ R-L	
3	*/ L-R	
4	+- L-R	

$5 + 1 * 6 = 6 \times 6 = 36$
↳ $= 5 + 6 = 11$ ✓

- $1 + 2 * 5 + 30/5$
  $1 + 10 + 30/5$
  $1 + 10 + 6$
  $11 + 6$
  $\Rightarrow 17$

$2 \wedge 2 \wedge 3$
$2^8 = 256$

Prefix
<operator><operand><operand>

Infix to prefix
$5 + 1 \quad +5 1$

$a * b + c = *ab + c$
$+*abc$

Postfix
<operand><operand><operator>

5 + 1 = 51 +

a*b+c = ~~+c ab*~~ ab*+c

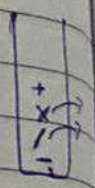
ab*c+

→ Infix to postfix conversion

A+B/C ABC/+

Stack | / | + |

A-B/C*D+E ABC/D*-E+



()

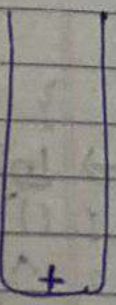
^

/* L-R

+- L-R

Infix K+L-M*N+(O^P)*W/U/V*T+Q

Input	Stack	Postfix
K		K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M
N	-*	KL+MN
+	+	KL+MN*-
(	+(	KL+MN*-
O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^W
W	+*	KL+MN*-OP^W
/	+/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V*
T	+*	KL+MN*-OP^W*U/V*T
+	+	KL+MN*-OP^W*U/V*T*
Q	+	KL+MN*-OP^W*U/V*T*Q
		KL+MN*-OP^W*U/V*T*Q+



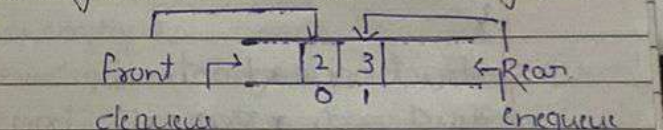
newNode = ____
printf("d.In")
newNode->next NULL;
newNode->next = temp->next;
temp->next = newN.

V	+/	KL+MN*OP^W*U/V
*	+*	KL+MN*OP^W*U/V |
T		KL+MN*OP^W*U/V | T
+	+	KL+MN*OP^W*U/V|T*+
Q	Postfix	KL+MN*OP^W*U/V|T*+Q+

→ Queue

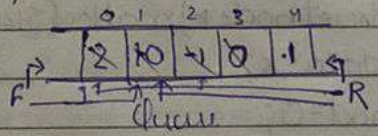
linear data structure (FIFO)(LILO)

Rule- Insertion can be perform from one end that is rear
Deletion can be perform from another end that is front.



Operations- (2) enqueue, dequeue
front()/peek()
isfull isEmpty

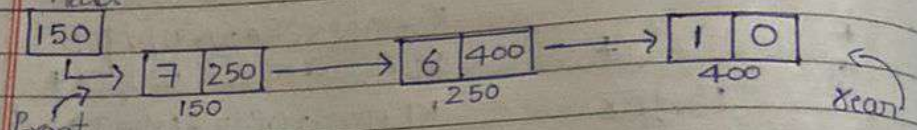
SIZE=5



F=R=-1 - queisempty
enque(2) F=R++;
enque(10 O(1)
enque(-1)
dequeue() Front++;
enque(0)(1)(5) - overflow condition
peek() - 10
deque() deque() deque
Front = Rear = pointing to 4 index
if(F==R)
F=R=-1

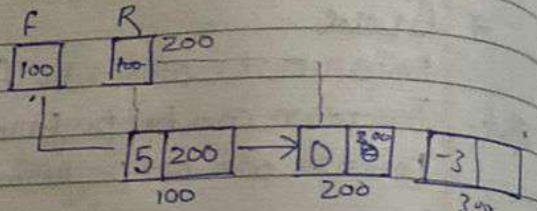
Application - printer

→ Queue linkedlist



FIFO

enequ dequeue O(1)

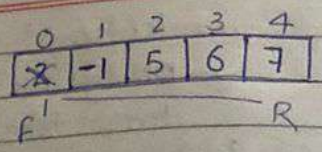


- struct node{
  int data;
  struct node *next;
}

```
struct node *front=-1;
struct node *rear=-1;
void enequ(int x){
  struct node *newNode;
  newNode=(struct node*)malloc(sizeof(struct node));
  newNode->data=x;
  newNode->next=0;
  if(front==0 && rear==0){
      front=rear=newNode;
  }else{
      rear->next=newNode;
      rear=newNode;
  }
}
void display(){
  struct node *temp;
  if(front==0 && rear==0){
      printf("List is empty");
  }else{
    temp=front;
```

```
        while(temp!=0){
            printf("%d", temp->data);
            temp=temp->next;
        }
    }
}
void dequeue(){
    struct node *temp;
    temp=front;
    if(front==0 && rear==0){
        printf("List is empty");
    }else{ printf("The dequeued element is %d", front->data);
        front=front->next;
        free(temp);
    }
}
void peek(){
    if(front==0 && rear==0){
        printf("List is empty");
    }else{
        printf("The front data is %d", front->data);
    }
}
```

→ Circular queue




```
#define N 5
int queue[N];
int front=-1;
int rear=-1;
void enqueue(int x){
  if(front==-1 && rear==-1){
      front=rear=0;
      queue[rear]=x;
  }
  else if((rear+1)%N==front){
      printf("queue is full");
  }
  else{
     rear=(rear+1)%N;
     queue[rear]=x;
  }
}
void dequeue(){
  if(front==-1 && rear==-1){
     printf("Queue is empty");
  }else if(front==rear){
     front=rear=-1;
  }else{
     printf("The dequed element is %d" queue[front]);
     front=(front+1)%N;
  }
}
```

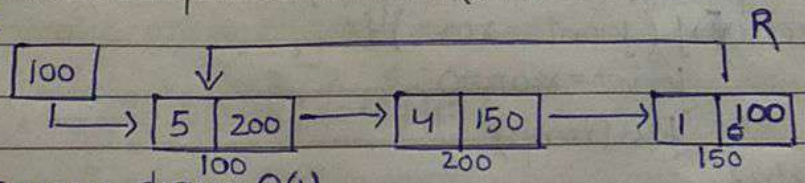
enqueue
2, 1, 5, 6, 7
display
dequeue
12
-1
enqueue
en- 10
en- 11

```
}
newNode = ____
printf("data")
newNode->next=NULL;
newNode->next = temp->next;
temp->next = newNode;
}
```

PAGE No.
DATE

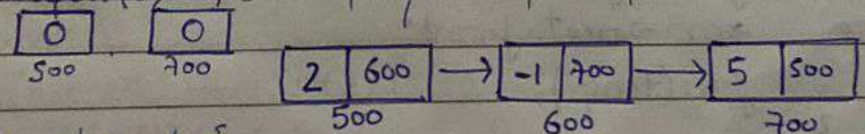
```
void display(){
  int i=front;                                  i=2 3 4 0 1
  if(front==-1 && rear==-1){                    (2+1)%5
    printf("Queue is empty");                   (3+1)%5
  }else{                                        (4+1)%5
    printf("Queue is:");                        5 6 7 0 10
    while(i!=rear){
      printf("%d", queue[i]);
      i=(i+1)%N;
    }
    printf("%d", queue[rear]
}
```

→ Circular queue with linked list



enqueue dequeue O(1)

enqueue(2) -1, 5 display() dequeue() peek()



-
```
struct node{
  int data;
  struct node *next;
};
struct node *front=0;
struct node *rear=0;
void enqueue(int x){
  newNode=(struct node*) malloc(sizeof(struct node));
  newNode->data=x;
  newNode->next=0;
```

```
if(rear==front==0 && rear==0){
    front = rear = newNode;
    rear -> next = front;
}else{
    rear -> next = newNodi;
    rear = newNode;
    rear -> next = front;
}
}
void dequeue(){
  struct node *temp;
  temp = front
  if(front==0 && rear==0){
     printf("Queue is empty");
  }else if(front==rear){
     front = rear = 0;
     free(temp);
  }else{
     front = front -> next;
     rear -> next = front;
     free(temp);
  }
}
void peek(){
  if(front==0 && rear==0){
     printf("Queue is empty");
  }else{
     printf("The peak element is %d", front -> data);
  }
}
```

```
printf("%d\n", temp->data);
newNode->next = NULL;
newNode->next = temp->next;
temp->next = newNode;
}
```

PAGE No.
DATE

```
void display(){
    struct node *temp;
    temp=front;
    if(front==0 && rear==0){
        printf("Queue is empty");
    }else{
        while(temp->next != front){
            printf("%d", temp->data);
            temp=temp->next;
        }
        printf("%d", temp->data);
    }
}
```

→ Linear search algorithm

0	1	2	3	4	5	6	7
15	5	20	35	2	42	67	17

n=8
data=42
└ location is 6
index is 5

element is present → location
not present
found=0;

```
- for(int i=0; i<n; i++){
      if(a[i]==data){
          printf("Element found at index %d", i);
          found=1
          break;
      }
}
```

Case1

```
if(found==0){
    printf("Element not found");
}
```

```
if(i==n){
    printf("Element not found");
}
```

Time complexity – O(n) Worst case
Best case O(1)

Sum of All cases / no of cases
$\frac{1+2+3\ldots n}{n} = \frac{n(n+1)}{2n}$
$O(\frac{n+1}{2})$

```
        else
            l = mid + 1
    }
    return -1;
}
```

Time complexity - O(logn) Worst case
O(1) Best case

→ Sorting

Bubble sort

n = 5

A → | 15 | 16 | 6 | 8 | 5 |
0 1 2 3 4

Basic principle of bubble sort is the adjacent two element will be compared if those element are in correct order then it will be fine will move further & if those element are not in correct order then we will swap the elements.

Pass-1

15 16 6 8 5
15 16 6 8 5
15 6 16 8 5
15 6 8 16 5
15 6 8 5 16

Pass-2

15 6 8 5 16
6 15 8 5 16
6 8 15 5 16
6 8 5 15 16

* The largest element is bubbled up to the end of the array.

Pass-3

6 8 5 15 16
6 8 5 15 16
6 5 8 15 16

Pass4

6 5 8 15 16
5 6 8 15 16

* How many passes required = no of element - 1;

```
for(i=0; i<n-1; i++){        flag=0;
    for(j=0; j<n-1; j++){
        if(a[j] > a[j+1]){
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }  flag = 1;
    }
}   if(flag == 0){
        break;
    }
```

Time complexity is- O(n) in best case
Worst case - $O(n^2)$

→ Insertion sort

A- | 5 | 4 | 10 | 1 | 6 | 2 | n=6
0 1 2 3 4 5

In this technique the given array is divided into two parts one is the sorted ~~path~~ sublist unsorted sublist

then we take one value from unsorted sublist and we have to insert that value in sorted sublist not simply inserted we have to find the appropriate place then insert that element in sorted sublist

S.L | U.S.L →
5 | 4 | 10 | 1 | 6 | 2

temp | 4 |

In sorted sublist there will be one element so compare 4 in sorted sublist & find appropriate place then insert.

| ?4 | 5 | 10 | 1 | 6 | 2 |

temp | 4 |

- Binary search

n=10

data=59

5	9	17	23	25	45	59	63	71	89
0	1	2	3	4	5	6	7	8	9
l				m					r

m1 m3 r' m2

* Requirement of binary search element is sorted array If not so first you have to sort the array then apply binary search.
* Divide & conquer technique

data=59 $\frac{l+r}{2}$ =

l	r	mid
0	9	4
5	9	7
5	6	5
6	6	6

Case I data == a[mid]

Case II data < a[mid] - r = m-1

Case III data > a[mid]

l left = mid+1

data=6

l	r	m
0	9	4
5	9	7
5	6	5
6	6	6
→ 7	6	

if (l > r) stoping condition

```
int binarySearch(arr, n, data){
    l=0, r=n-1;
    while(l<r){
        mid=(l+r)/2;
        if(data==a[mid])
            return mid;
        else if(data<a[mid])
            r=mid-1;
```

```
if(min != i){
    swap(a[i], a[min]);
  }
}
```

Time complexity $O(n^2)$

```
int temp = arr[i];
arr[i] = arr[min];
arr[min] = temp;
```

min 0 1 2 3 4 5

j = 1 2 3 4 5 6

a[1] < a[0] ✓

a[2] < a[1]

a[3] < a[1]

a[4] < a[1] ✓

a[5] < a[4] ✓

→ Quick sort

A

10	15	1	2	9	16	11
0	1	2	3	4	5	6

So the basic idea behind the quick sort is that in this sorting algorithm is we have to choose one pivot element.

Pivot element is 10

Divide & conquer technique.

We have to partion the array in such that all the elements less than 10 will be left side of array & all the elements greater than the pivot element will be right side.

Key

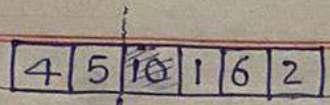
Partion1	Pivot	Partion2

val < pivot val > pivot

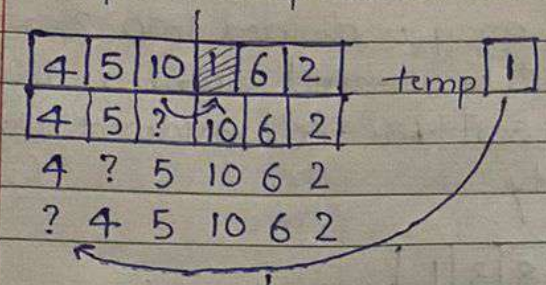
0-2

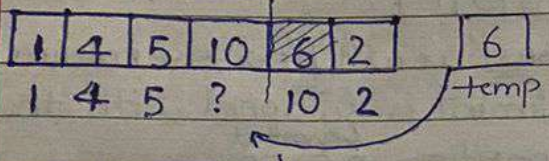
4-6

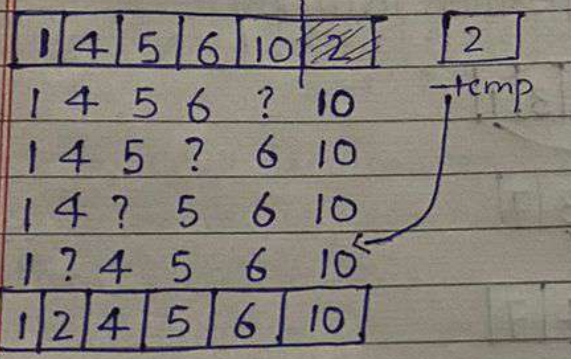
0	1	2	3	4	5	6
2	1	9	10	15	11	16



compare temp value with sorted sublist





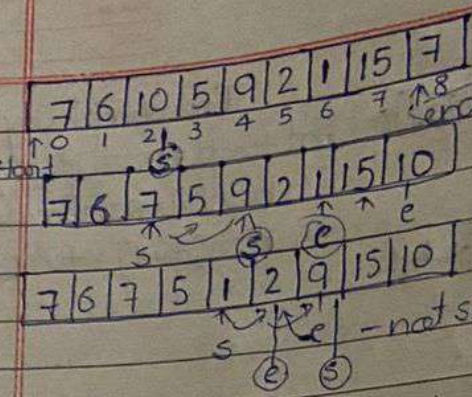


i = 5
j = ~~5~~ 4 3 2 1
a[5] = a[4]
a[4] = a[3]
a[3] = a[2]
a[2] = a[1]
false

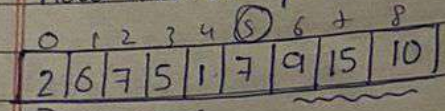
```
for(i=1; i<n; i++){
    temp=a[i];
    j=i-1;
    while(j>=0 && a[j]>temp){
        a[j+1]=a[j];
        j--;
    }
    a[j+1]=temp;
}
```

Worst case - $O(n^2)$

Best case - $O(n)$



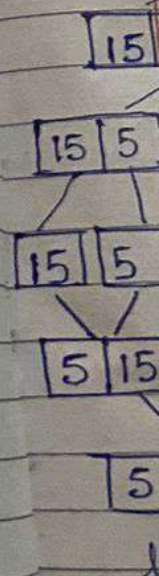
now then swap end with pivot element




```
partion (a, lb, ub){
   pivot = a[lb];
   start = lb;
   end = ub;   while(start<end){
   while(a[start]<=pivot){
        start++;
   }
   while(a[end]>pivot){
        end--;
   }
   if(start<end){
      swap(a[start], a[end]);
}
swap(a[lb], a[end];
return end;
}
QuickSort(A, lb, ub){
   if(lb<ub){
   loc = partion(a, lb, ub);
   QuickSort(a, lb, loc-1);
   QuickSort(a, loc+1, ub);
}
}
```

$O(n^2)$
$O(n \log n)$

$\frac{0+4}{2} = 2$



- Selection sort

n=6

a

7	4	10	8	3	1
0	1	2	3	4	5

In this case also the array is divided into 2 categories one is sorted & another is unsorted same as insertion sort but in selection sort ~~the~~ we considered the sorted sublist is empty.

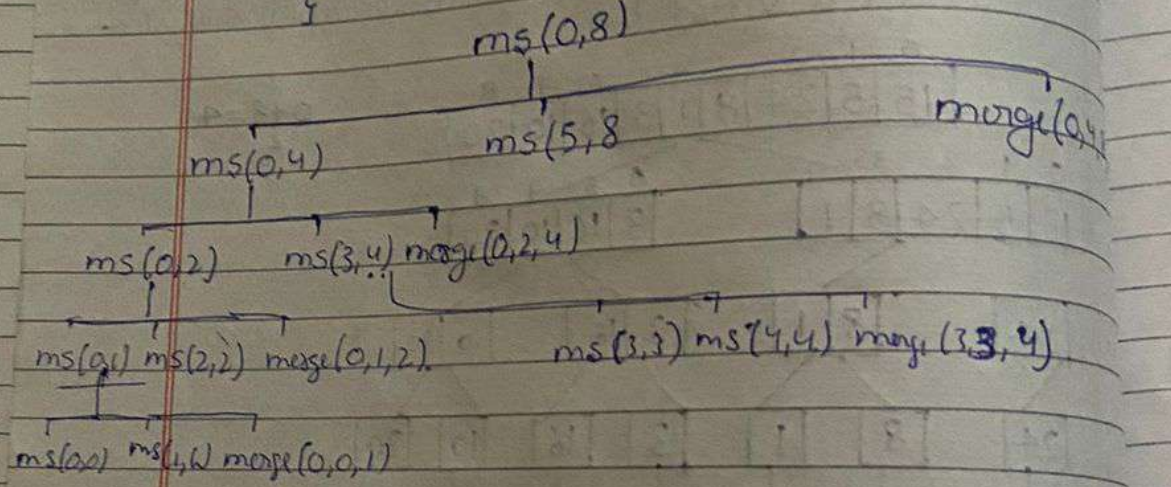
Sorted	unsorted					
	7	4	10	8	3	1

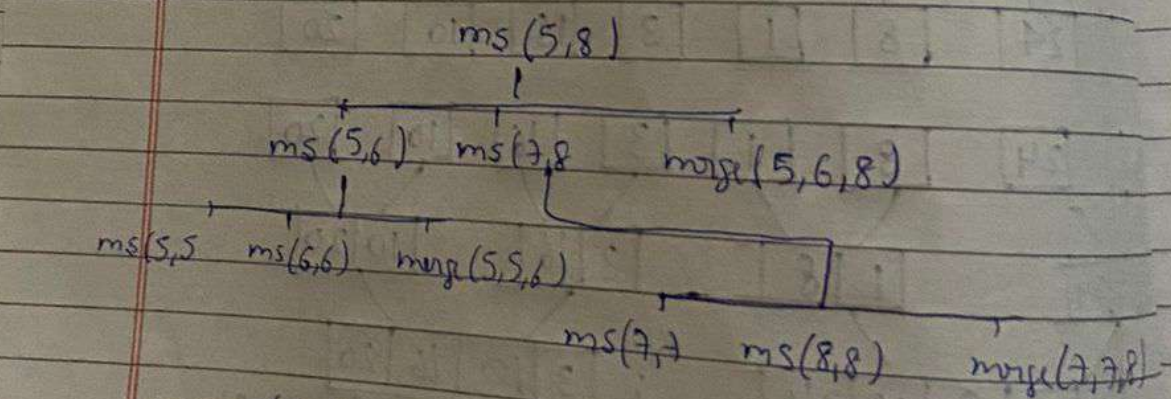
From the unsorted sublist we will find the minimum element and that minimum element is swapped with the element which is at starting position in unsorted array.

	7	4	10	8	3	1
Pass 1	1	4	10	8	3	7
Pass 2	1	3	10	8	4	7
Pass 3	1	3	4	8	10	7
Pass 4	1	3	4	7	10	8
Pass 5	1	3	4	7	8	10

```
for(i=0; i<n-1; i++){
    int min=i;
  for(j=i+1; j<n; j++){
      if(a[j]<a[min]{
          min=j;
      }
}
```

```
mergeSort(a, lb, ub){
  if(lb<ub){
    mid=(lb+ub)/2;
    mergeSort(a, lb, mid);
    mergeSort(a, mid+1, ub);
    merge(a, lb, mid, ub);
  }
```






```
merge(a, lb, ub){
  i=lb
  j=mid+1
  k=lb;
  while(i<=mid && j<=ub){
    if(a[i]<=a[j]){
      b[k]=a[i]
      i++;
      k++;
```

```
printf("data");
newNode->next=NULL;
newNode->next=temp->next;
temp->next=newNode;
}
```

```
    }
    else{
        b[k]=a[j];
        j++;
        k++;
    }
    k++;
  }
  if(i>mid){
    while(j<=ub){
      b[k]=a[j];
      j++;
      k++;
    }
  }else{
    while(i<=mid){
      b[k]=a[i];
      i++;
      k++;
    }
  }
  for(k=lb;k<=ub;k++){
    a[k]=b[k];
  }
}
```

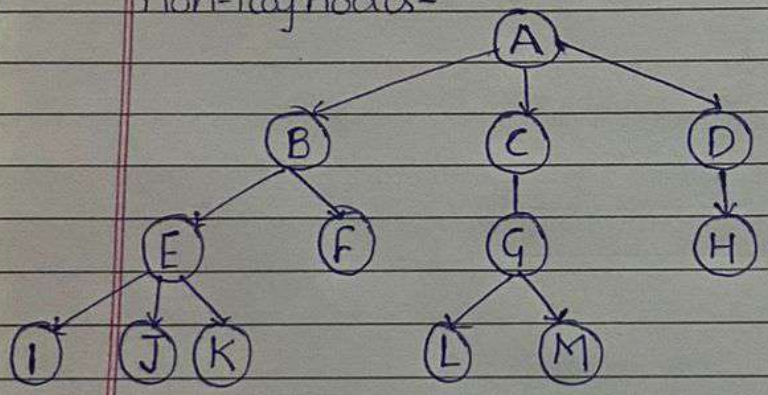
void

PAGE No.
DATE

Introduction to trees

- It is a non linear data structure.
- The data in the form of hierarchial data.
* Tree can be defined as a collection of entities (nodes) linked together to simulate a hierrachy.

Root- the node which not having any parent is known as root node of a tree.
child node.
leaf node. - the node which not having any child.
non-leaf nodes-



root = A
nodes = A B C D E F G H I J K L M
Parent node = G is parent of L & M
child node = L & M are children of G
leaf node = I J K F L M H
non leaf node = A B C D E G
path- sequence of consecutive edges from source node to destination node.
ancestor - any predecessor node on the path from root to that node.

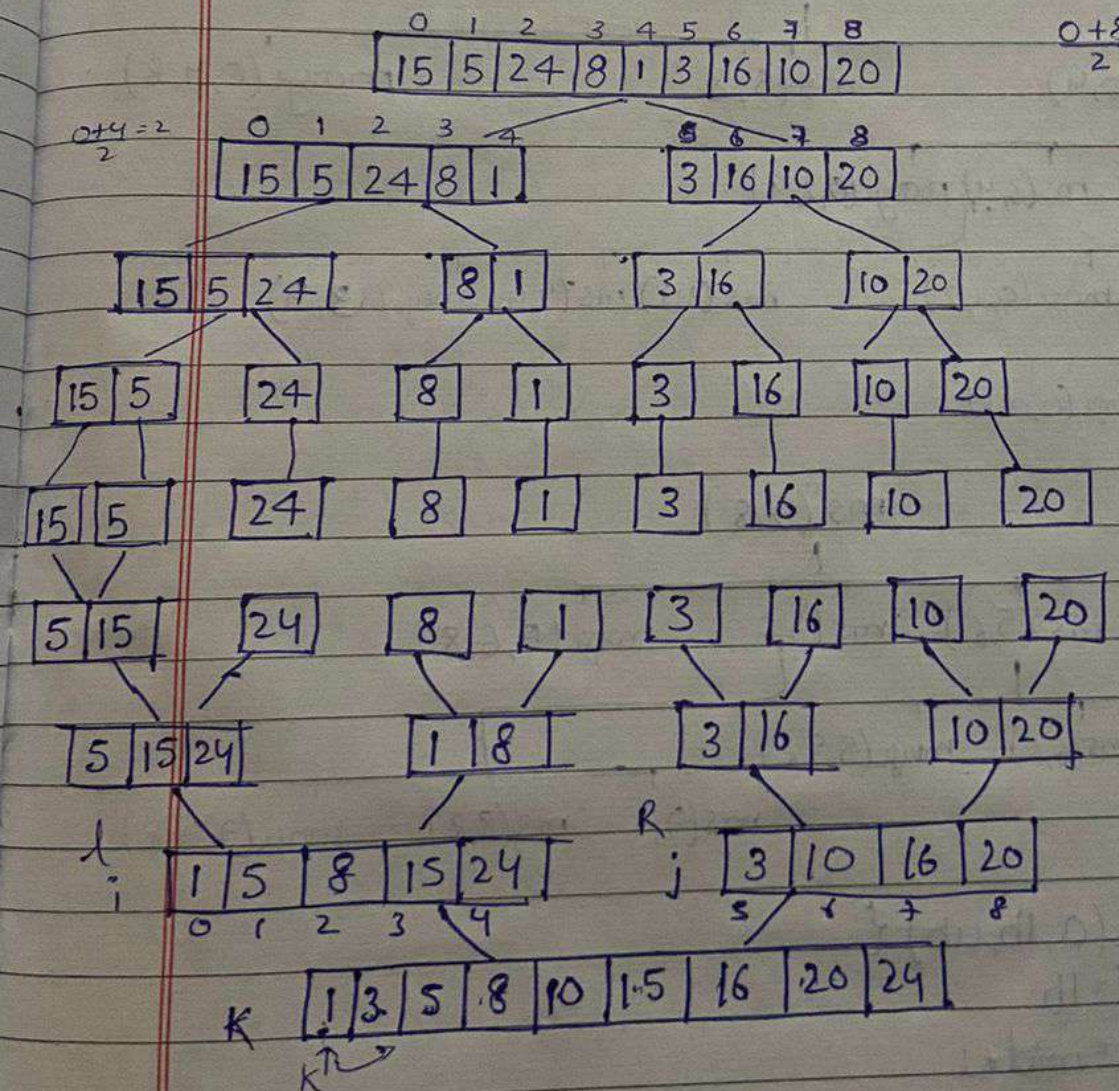
L = A C G
H = A D

newNode->next = temp->next;
temp->next = newNode;

PAGE No.
DATE

- Merge sort

This also works on divide & conquer technique
In this case the complete list is divided into sublist & each sublist is having one element.
After that we have to merge the sublist & we have keep merging the sublist to produce a sorted sublist. & keep on merging the sublist until we get one complete sublist & that sublist is sorted list.



Compare L[i] with R[j] which one is smaller put into array jo ⁕ small hoga usme plus 1 ho jyega.

PAGE No.
DATE

decendant - any successor node on the path from that node to leaf node.
C = G, L, M

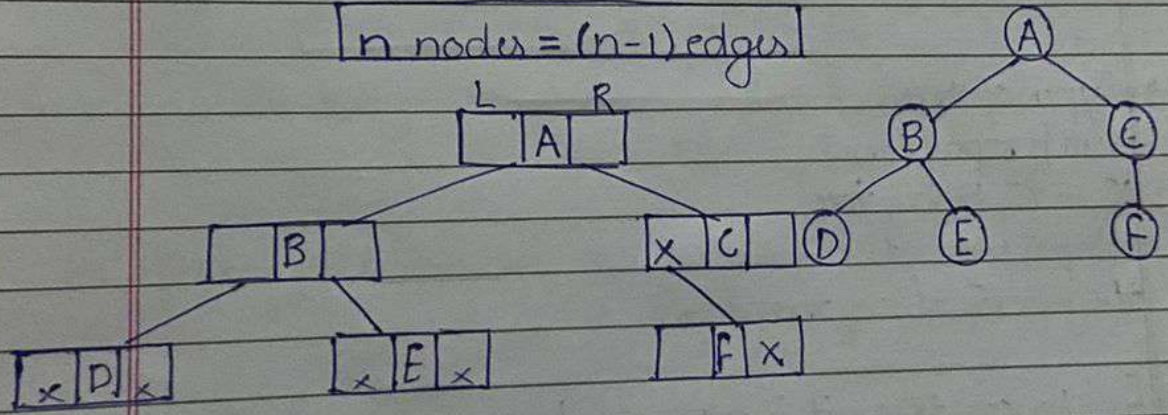
sibling - all the node of same parent.
E & F are children of B so E & F are sibling

degree - the no. of children that node.
degree of A = 3

dept of node - length of path from root to that node.
dept of A to I = 3

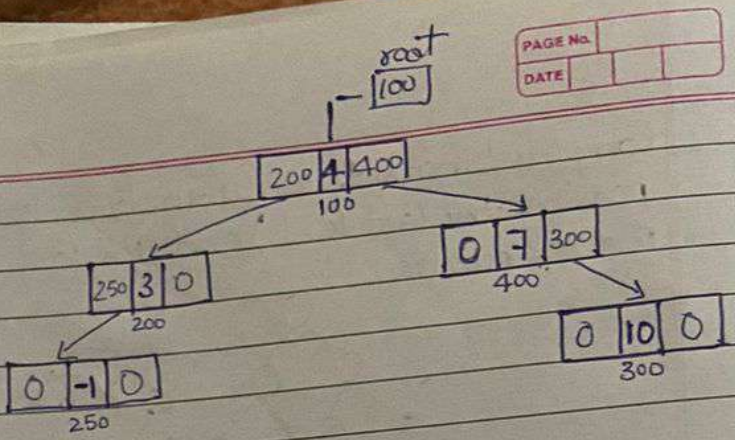
height of node - no of edges in the longest path from that node to leaf.
B to I = 2

n nodes = (n-1) edges

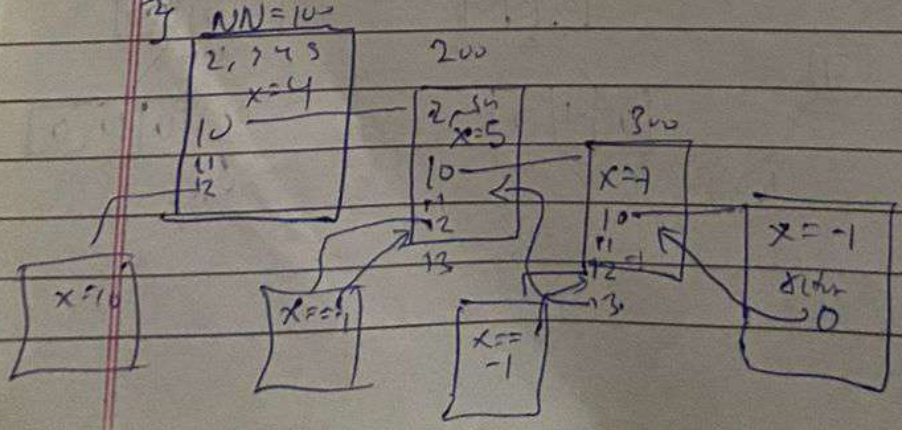



```
struct node {
    int data;
    struct node *left;
    struct nod *right;
};
```

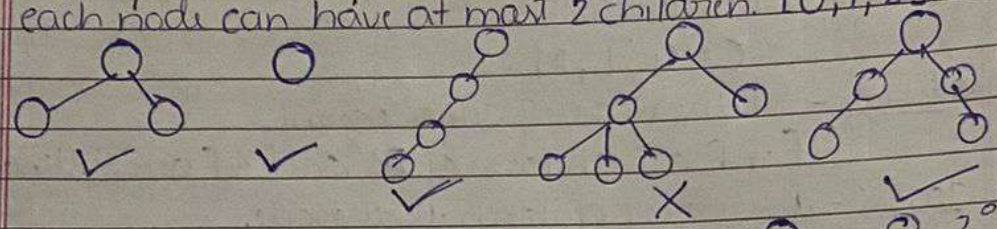
→



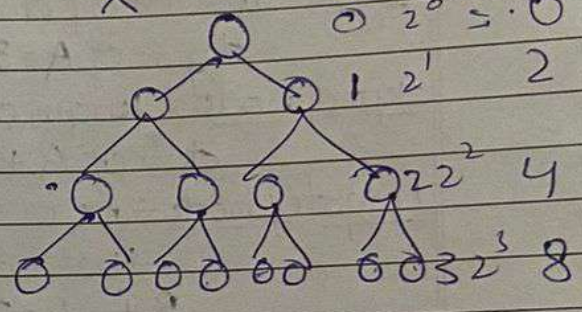

```
struct node{
    int data;
    struct node *left, *right;
};                          it will return pointer to a node
struct node *creat(){
1   int x;
2   struct node *newNode;
3   newNode=(struct node*)malloc(sizeof(structNode));
    data & (-1 for no node);
4   printf("Enter the value you want to insert");
5   scanf("%d", &x);
6   if(x==-1){
7       return 0;
    }else{
8       newNode->data=x;
9       printf("Enter left child of %d", x);
10      newNode->left=creat();
11      printf("Enter right child of %d", x);
12      newNode->right=creat();
    }
13  return newNode;
```



- Binary tree & its types
- each node can have at most 2 children. [0,1,2]



* max no of nodes possible at level i is 2^i

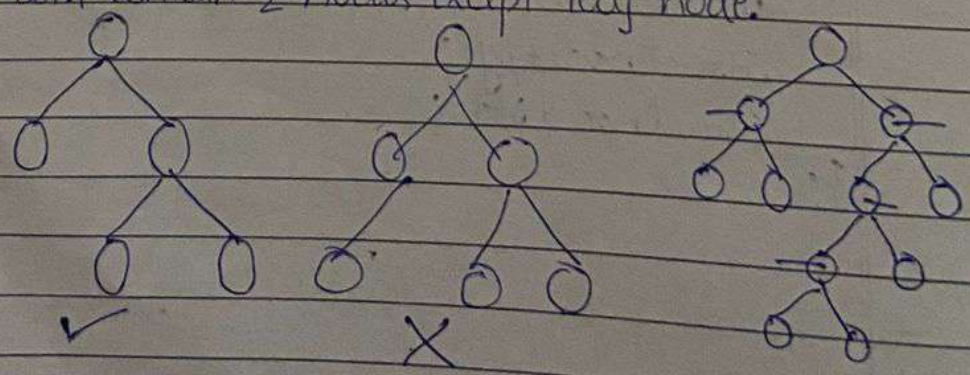


* max no of nodes of height h
  $2^{h+1} - 1$
  min no of nodes = h + 1

$0+2+4+8 = 14$

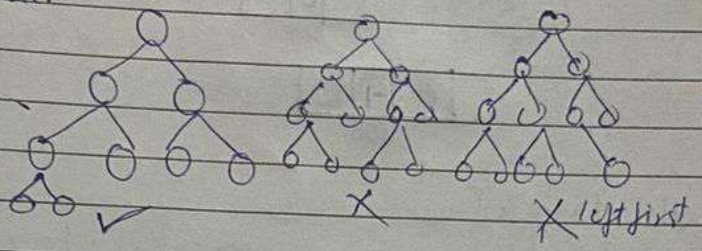
→ Types of trees
→ Full / proper / strict
→ Complete binary tree
→ Perfect binary tree
→ Degenerate binary tree.

1. Full binary tree - it is a binary tree where each node contains either zero or 2 children
   Each node will contain 2 nodes except leaf node.

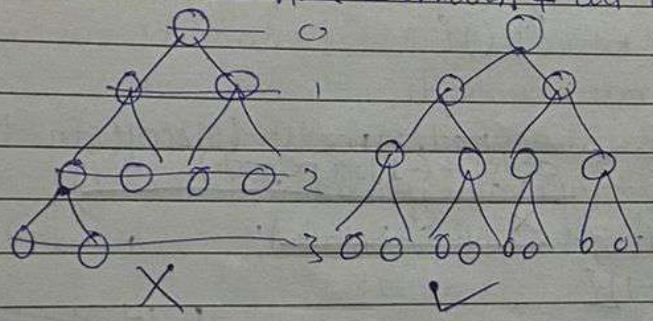


No. of leaf node = no of internal nodes + 1
= 4 + 1
= 5

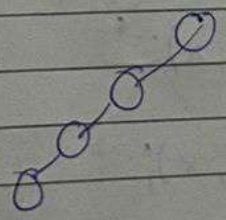
2. Complete binary tree - all the levels are completly filled (except possibly the last level) and the last level has nodes as left as possible.



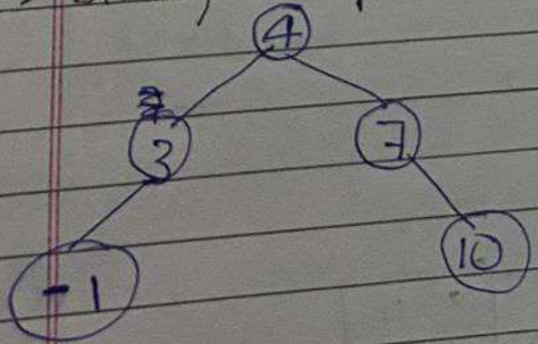
3. Perfect binary tree - a tree can be a perfect binary tree if all the internal nodes have 2 children & all leaves are at same level.

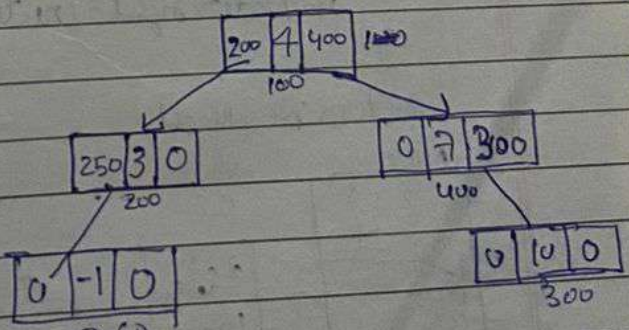


4. Degenerate binary tree - all the internal nodes have only one child. (left skewed binary tree)



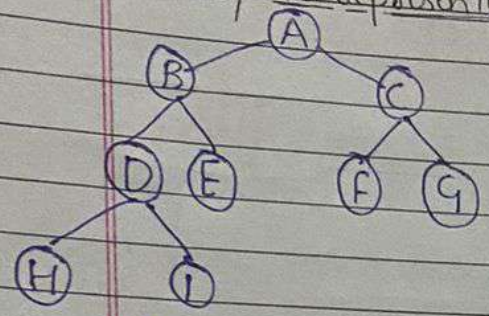
→ Binary tree implementation





PAGE No.
DATE

- Binary tree representation using array.



A	B	C	D	E	F	G	H	I	Case I
0	1	2	3	4	5	6	7	8	

A	B	C	D	E	F	G	H	I	Case II
1	2	3	4	5	6	7	8	9	

If a node is at i^{th} index-

→ left child would be at $[(2*i)+1]$

Right child would be at $[(2*i)+2]$

Parent would be at $\left[\frac{(i-1)}{2}\right]$ floor value.

i=4

$\frac{(4-1)}{2} = \frac{3}{2} = 1.5 = 1$

LC $[2\times4+1] = 9$

$[2\times4+2] = 10$

Case II node is at i^{th} index

left would be $= [2*i]$

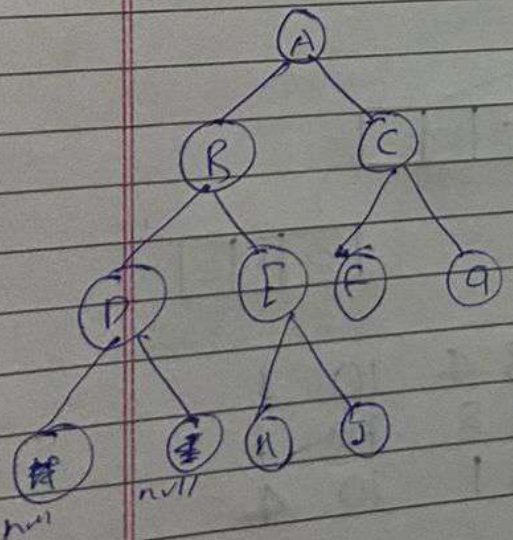
right would be $[(2*i)+1]$

Parent at $= \left[\frac{i}{2}\right]$ floor value.

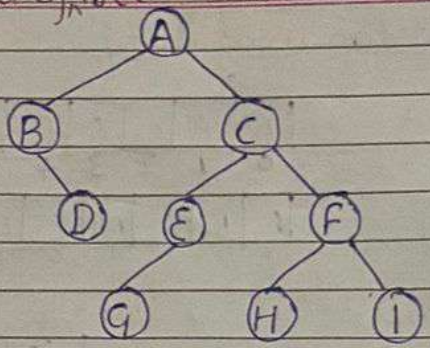
i=4

LC $= 2\times4 = 8$

$2\times4+1 = 9$

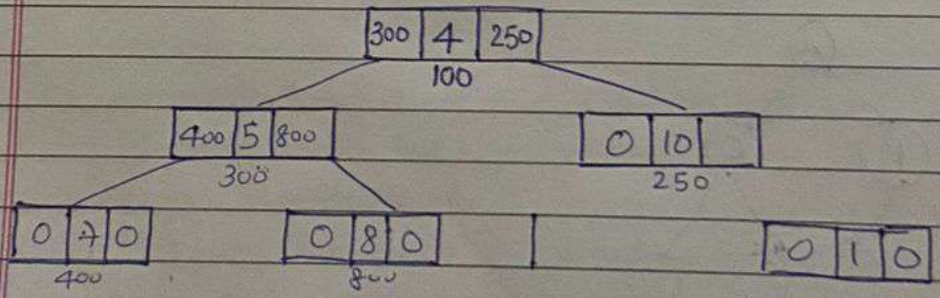


Traversal of tree
binary.



Inorder (Left, Root, Right)
Preorder (Root, Left, Right)
Postorder (Left, Right, Root)

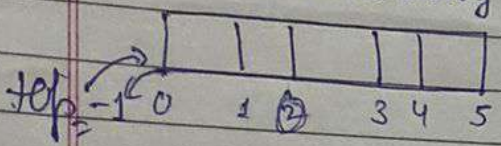
Inorder = B D A G E C H F I
Preorder = A B D C E G F H I
Postorder = ~~B~~ ~~D~~
D B G E H I F C A



Inorder = Left, Right, Root) 7 5 8 4 10 1
Preorder = Root, Left, Right) 4 5 7 8 10 1
Post = Left Right, Root) 7 8 5 1 10 4

```
void preorder(struct node *root){

    printf("%d", root->data);
    preorder(root->left);
```

Stack array




```
#include <stdio.h>
#include <stdbool.h>

bool isEmpty(){

    if(top == -1){
        return true;
    }else{
        return false;
    }
}

bool isFull
```

top ++;
stack[top] = num;