Stack

A stack is formally defined as a list (a lin-ear Data Structure) in which all insertion and deletion are made at one end called the Top of Stack (TOS). The fundamental operations performed on a stack are :-

i) Push = To insert an element into the stack.

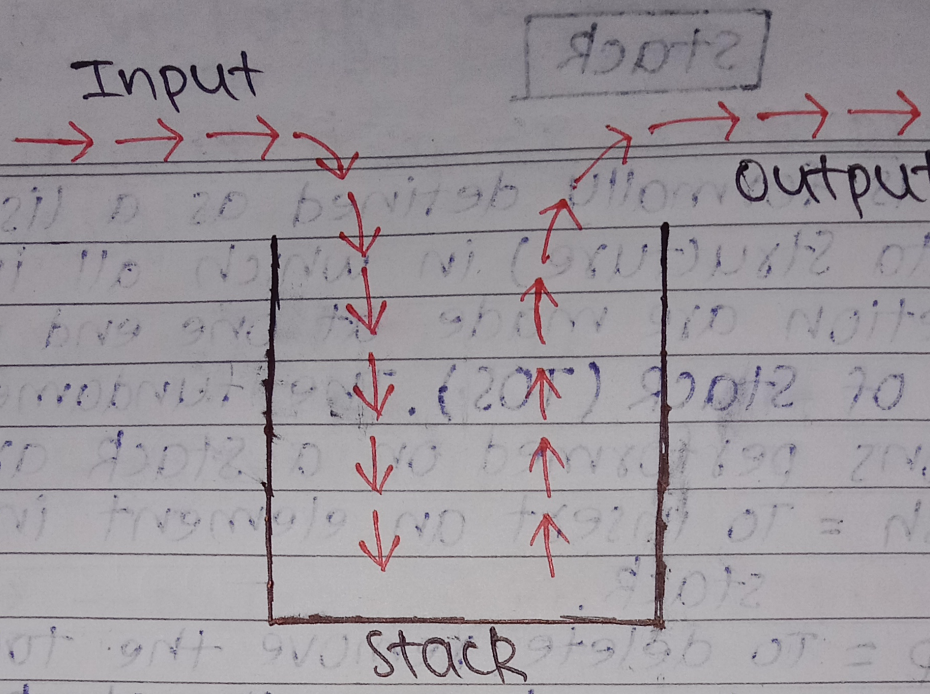
ii) Pop = To delete / remove the topmost element from the stack.

iii) Peek = To return the index's value / the value at given index.

iv) isempty / isfull = to determine whether the stack is empty or full to carry efficient push and pull operations.

The most recently pushed element are / can be checked prior to performing a pop operation. A stack is based on the Last In First Out (LIFO) order, in which insertion and deletion operations are performed at one end. Also, the information can only be removed in the opposite order in which is added to it. The most accessible information in a stack is at the top of the stack and least accessible information is at the bottom.

(2)



A pointer TOS (Top of Stack) keeps track of the top most information in the stack. Initially, when the stack is empty TOS has a value zero (if implementing through C or C++, this may be -1) and when the stack contains a single information TOS has a value 1, and so on. Before an item is pushed onto a stack, the pointer is incremented by one. The pointer is decreme-nted by one each time a deletion is made from the stack.

* Implementation of a stack

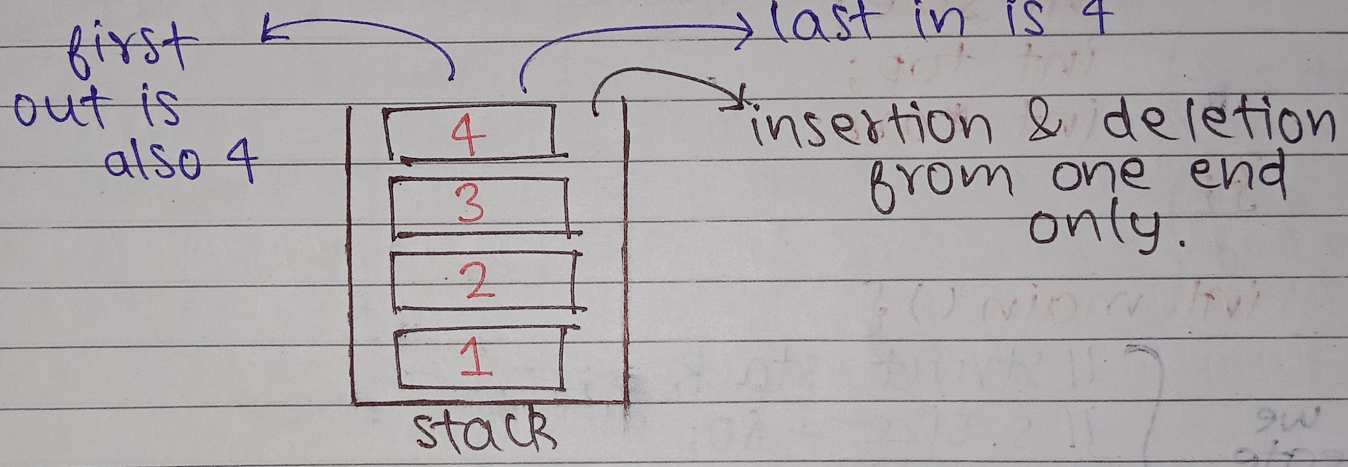
Stack is a list and hence any of the list implementation techniques may be used to implement a stack. Two common implementations are :-

i) Linked List (Pointer)
ii) Array

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

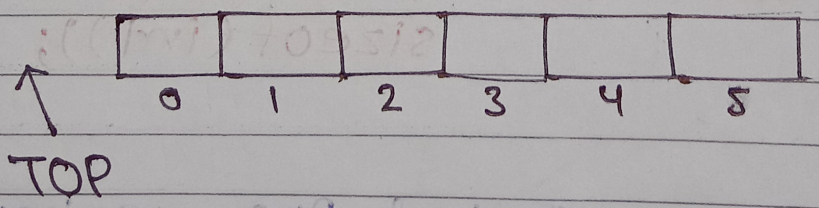
In stack :- Push means Insert an element and Pop means Delete an element

Stack is a collection of elements following LIFO. Items can be inserted or removed only from one end.



* Implementing stack using Array :-

In stack, we don't have only the Array, we also have the TOP (Topmost element) variable. Top is not the pointer, Top contains the index value.



If there is no element in the Array, then the value of TOP is -1. If there is 1 element in the array then the value of TOP is 0 & so on.

while implementing a stack using array, we have to remember 2 things :-

i) Fixed Size Array creation
ii) Top Element.

Stack Program :-

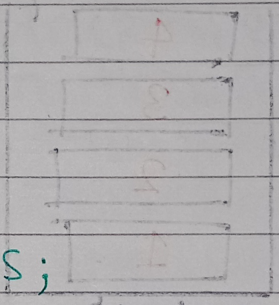
```
#include<stdio.h>
#include<stdlib.h>
struct stack{
    int size;
    int top;
    int *arr;
};

int main(){
```

if we create stack object then we write like this,

```
// struct stack s;
// s.size = 80;
// s.top = -1;
// s.arr = (int *)malloc(s.size *
            sizeof(int));
```

if we create stack pointer then we write like this.

```
struct stack *s;
s -> size = 80;
s -> top = -1;
s -> arr = (int *)malloc(s -> size *
            sizeof(int));
```



* **Note:-** Before Push & POP, we have to check that the stack is empty or the stack is full. So, we have to create 2 functions.

```
int isEmpty(struct stack *ptr)
{
    if(ptr -> top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
```

this check that the stack is empty or not.

```
int isFull(struct stack *ptr){
    if(ptr -> top == ptr -> size - 1){
        return 1;
    }
    else{
        return 0;
    }
}
```

this check that the stack is full or not.

these functions return integer value, return 1 means true and return 0 means false.

* Program for empty stack:

```
#include <stdio.h>
#include <stdlib.h>
struct stack
{
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack *ptr)
{
    if(ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
int isFull(struct stack *ptr)
{
    if(ptr->top == ptr->size-1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
int main()
{
    struct stack *s;
```

```
s -> size = 80;
s -> top = -1;
s -> arr = (int *) malloc (s -> size *
                           sizeof (int));

if (isEmpty (s))
{
   printf("stack is empty");
}
else
{
   printf("stack is full");
}
return 0;
}
```

when we run this code, it will print "the stack is empty" because there is no element in the stack at the moment.

```
s -> arr[0] = 7;
s -> top++;
```

Pushing an element manually in the stack before function calling

after writing this & run our code, it will print "the stack is not empty" because we manually pushed the element in the stack.

* Infix to Postfix conversion using stack

Eg:- x - y / z - R * d

* first we do it manually,

((x -(y / z))-(R * d))
((x -(yz/))-(Rd*))
((xyz/-)-(Rd*))
(xyz/-Rd*-) → Infix to Postfix manually.

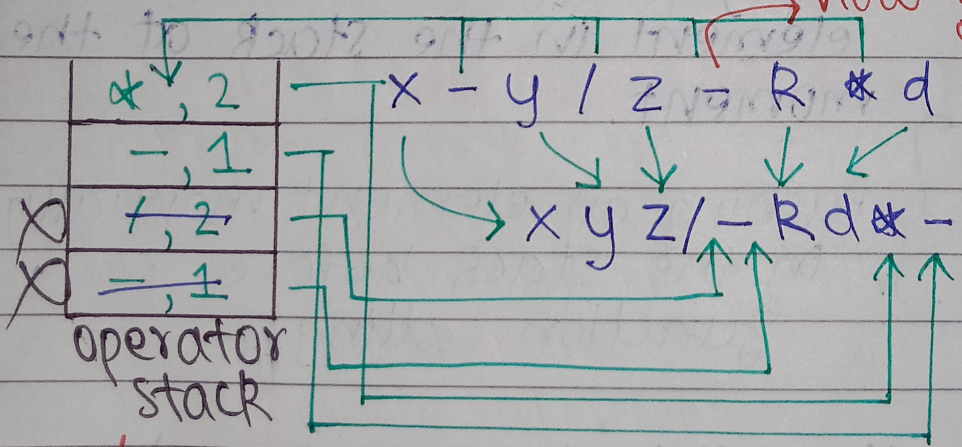
* now we do conversion using stack.

x - y / z - R * d

Push operator in stack & write operand as it is.

*, /	2
+, -	1

→ Precedence Table



now we have minus operand and we can't put it on the top of divide (/) operand, so we have to pop out the divide operator & add it in our expression, & we also have to pop out minus operator because same precedence can't sit together. After pop out both operator, then we add our latest minus operator.

after all the expressions pop out the remaining operators form the stack & add them in our expression.

* Algorithm to evaluate postfix expression.

i) create a stack to store operands.
ii) scan the given expression from left to right.
iii) a) If the scanned character is an operand, push it into the stack
b) If the scanned character is an operator, POP 2 operands from stack and perform operation and PUSH the result back to the stack.
iv) Repeat step 3, till all the characters are scanned.
v) when the expression is ended, the number in the stack is the final result.

* Algorithm to evaluate prefix expression.

i) Let opndstack be an operand stack.
ii) opndstack = emptystack.
iii) scan one char at a time from right to left in string.
iv) If scan char is operand, push it in stack
v) If scan char is operator, pop opnd1, opnd2 and perform the operand's operation specified by the operator. Push the result into stack.
vi) Repeat step 2, 3 and 4 until input prefix strings end.
vii) Pop opndstack & display, which is required value of given expression.

* Polynomial Addition Algorithm using LinRed List :-

(10)

Node Structure,

coefficient	Power	Address of next node

Algorithm,

i) If both the numbers are null then return.

ii) Else if compare the power, if same then add the coefficients and recursively call addPolynomials on the next elements of both the numbers.

iii) Else if the power of first number is greater, then print the current element of first number & recursively call addPolynomial on the next element of the first number & current element of the second number.

iv) Else print the current element of the second number and recursively call addPolynomial on the current element of first number & next element of second number.

—X———X———X———X——X—

Stack Program

```
#include <stdio.h>
#include <stdlib.h>
#define size 4
int top = -1, inp_array[size];
void push();
void pop();
void show();
int main()
{
  int choice;
  while (1)
  {
    printf("Operation performed by stack");
    printf("1. Push the element. \n 2.
      Pop the element. \n 3. show \n 4. End");
    printf("\n Enter the choice : ");
    scanf("%d", &choice);
    switch (choice)
    {
      case 1: push();
          break;
      case 2: pop();
          break;
      case 3: show();
          break;
      case 4: exit(0);
      default:
          printf("\n Invalid choice");
    }
  }
}
```

```
void push()
{
  int x;
  if(top == size-1)
  {
    printf("Overflow");
  }
  else
  {
    printf("enter element to be inserted
  to the stack : ");
    scanf("%d",&x);
    top = top+1;
    inp_array[top] = x;
  }
}

void pop()
{
  if(top == -1)
  {
    printf("underflow");
  }
  else
  {
    printf("popped element : %d",
        inp_array[top]);
    top = top-1;
  }
}
```

```
void show()
{
  if (top == -1)
  {
    printf("underflow");
  }
  else
  {
    printf("elements present in the stack ");
    for (int i = top; i >= 0; --i)
    {
      printf("%d", inp_array[i]);
    }
  }
}
```

* checking the stack is empty or not.

```
int isEmpty(struct stack *ptr)
{
  if(ptr -> top == -1)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
```

* checking the stack is full or not.

```
int isfull(struct stack *ptr)
{
  if(ptr -> top == ptr -> size - 1)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
```

* push element into stack.

```
void push(struct stack *ptr, int val)
{
  if(isFull(ptr))
  {
    printf("stack Overflow, cannot
      push %d to the stack \n", val);
  }
  else
  {
    ptr -> top ++;
    ptr -> arr[ptr -> top] = val;
  }
}
```

↳ element push krne se pehle hume check krna hoga k stack full to nahi h, agr stack full h to tum push nahi krskte.

* Pop element from the stack.

```
int pop(struct stack *ptr)
{
  if(isEmpty(ptr))
  {
    printf("stack underflow, cannot pop
      from the stack");
    return -1;
  }
  else
  {
    int val = ptr -> arr[ptr -> top];
    ptr -> top--;
    return val;
  }
}
```

↳ element pop krne se pehle ye check krege k stack khaali to nhi h, agr stack khaali hai to pop kr hi nhi skte.

* Stack Program with Push, Pop, Empty, Full Logic.

```
#include<stdio.h>
#include<stdlib.h>
struct stack
{
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack *ptr)
{
    if(ptr -> top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
int isFull(struct stack *ptr)
{
    if(ptr -> top == ptr -> size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

```
void push(struct stack *ptr, int val)
{
  if(isFull(ptr))
  {
    printf("stack overflow, cannot push
      %d to the stack", val);
  }
  else
  {
    ptr -> top++;
    ptr -> arr[ptr -> top] = val;
  }
}

int pop(struct stack *ptr)
{
  if(isEmpty(ptr))
  {
    printf("stack underflow, cannot pop
      from the stack.");
    return -1;
  }
  else
  {
    int val = ptr -> arr[ptr -> top];
    ptr -> top--;
    return val;
  }
}
```

```
int main()
{
    struct stack *sp = (struct stack *)
        malloc (sizeof(struct stack));
    sp -> size = 10;
    sp -> top = -1;
    sp -> arr = (int *) malloc (sp -> size
                * sizeof(int));
    printf("stack has been created");
    printf("Before pushing, Full: %d",
                        isFull(sp));
    printf("Before Pushing, Empty: %d",
                        isEmpty(sp));

    push(sp, 1);
    push(sp, 23);
    push(sp, 99);
    push(sp, 75);
    push(sp, 3);
    push(sp, 64);
    push(sp, 57);
    push(sp, 46);
    push(sp, 89);
    push(sp, 6);
    push(sp, 46); // stack overflow since the
                     size of the stack is 10.

    printf("after pushing, Full: %d",
                        isFull(sp));
    printf("after pushing, Empty: %d",
                        isEmpty(sp));
```

```
    printf("popped %d from the stack", pop(sp));
    printf("popped %d from the stack", pop(sp));
    printf("popped %d from the stack", pop(sp));
    return 0;
}
```

* peek() operation :- Peeking into something means to quickly see what's there at some place. In a similar way, it refers to looking for the element at a specific index, in a stack.

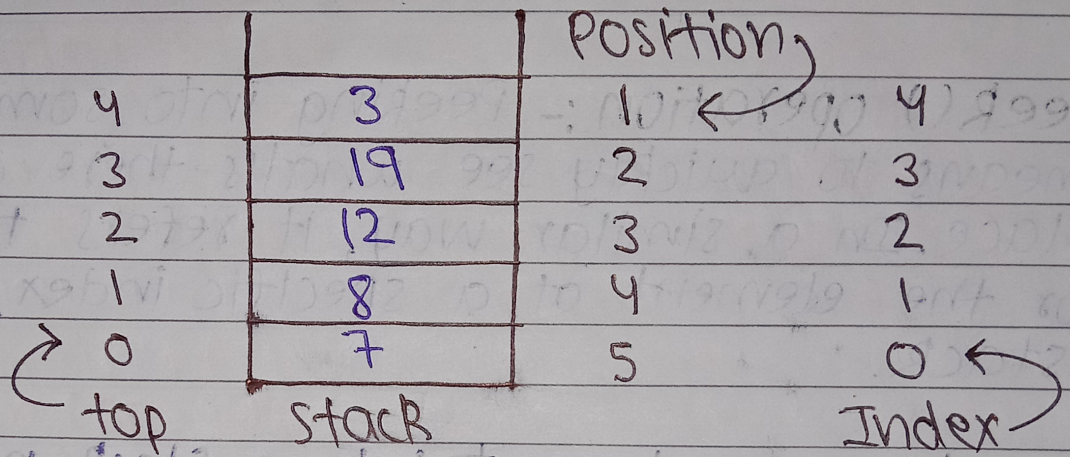
* pushing an element into a stack needs to first check if the stack is full or not, if the stack is not full, then we insert the element at the incremented value of the top.
* popping from a stack needs to first check if the stack is empty or not, if the stack is not empty, then we just decrease the value of the top by 1.

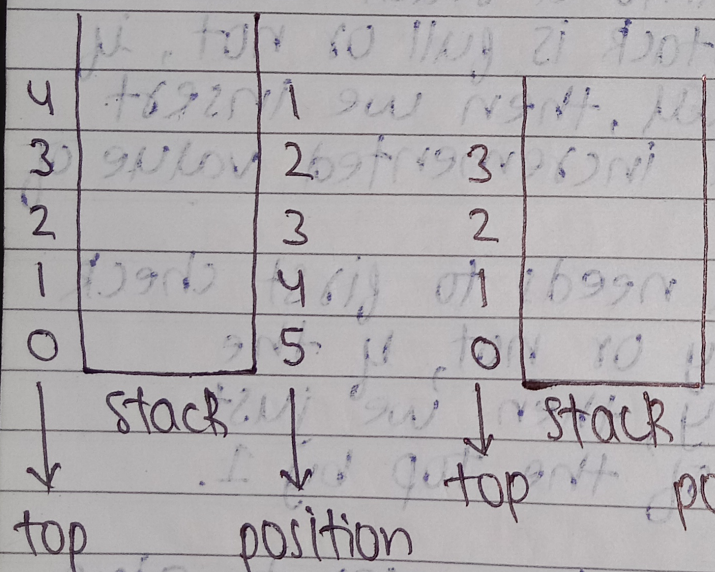
peek() operation requires the user to give a position to peek. Here, position refers to the distance of the current index from the top element + 1.

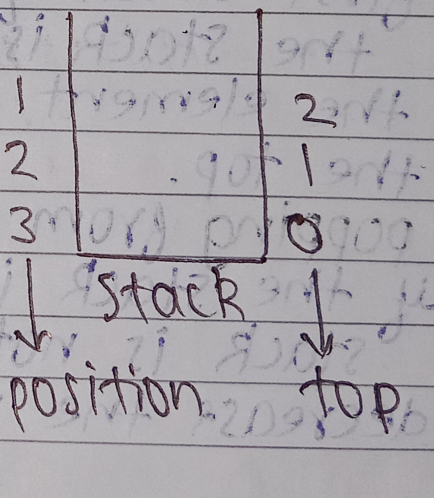
The index is,

(top - position + 1)

when user give some position, before we return the element from that position, first we'll check if the position asked is valid for the stack or not.







Jab top 4 hoga, to position 1 to 5 tk jyegi.

Jab top 3 hoga, to position 1 to 4 tk jyegi.

Jab top 2 hoga, to position 1 to 3 tk jyegi

and so on.

when user give some position, first we have to check that the position asked is valid or not. so, the logic to check the valid position is :-

```
int peek (struct stack *sp, int i)
{
  if(sp->top - i + 1 < 0)
  {
    printf("invalid position");
    return -1;
  }
  else
  {
    return sp->arr[sp->top - i + 1];
  }
}
```

peek() operation program :-

```
#include<stdio.h>
#include<stdlib.h>
struct stack
{
  int size;
  int top;
  int *arr;
};
int isEmpty(struct stack *ptr)
{
  if(ptr->top == -1){
    return 1;
  }
  else{
    return 0;
}}
```

```
int isFull(struct stack *ptr)
{
    if(ptr->top == ptr->size-1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
void push(struct stack *ptr, int val)
{
    if(isFull(ptr)){
        printf("overflow");
    }
    else{
        ptr -> top++;
        ptr -> arr[ptr->top] = val;
    }
}
int pop(struct stack *ptr)
{
    if(isEmpty(ptr)){
        printf("underflow");
        return -1;
    }
    else
    {
        int val = ptr -> arr[ptr -> top];
        ptr -> top--;
        return val;
    }
}
```

```
int peek(struct stack *sp, int i)
{
    int arrayInd = sp -> top - i + 1;
    if(arrayInd < 0)
    {
        printf("invalid position");
        return -1;
    }
    else
    {
        return sp -> arr[arrayInd];
    }
}
int main()
{
    struct stack *sp = (struct stack *) malloc
                       (sizeof(struct stack));
    sp -> size = 50;
    sp -> top = -1;
    sp -> arr = (int *) malloc (sp -> size *
                                sizeof(int));
    printf("stack created successfully\n");
    printf("before pushing, full : %d\n",
                                isFull(sp));
    printf("before pushing, empty : %d\n",
                                isEmpty(sp));
    push(sp, 1);
    push(sp, 25);
    push(sp, 79);
    push(sp, 99);
    push(sp, 86);
    push(sp, 71);
    push(sp, 13);
    push(sp, 69);
    push(sp, 90);
```

```
push(sp, 6);
push(sp, 5);
push(sp, 75);
for(int j=1; j<=sp->top+1; j++)
{
  printf("value at position %d is %d\n",
        j, peek(sp, j));
}
return 0;
}
```

We have 2 more operations in stacks :-
stackTop and stackBottom,

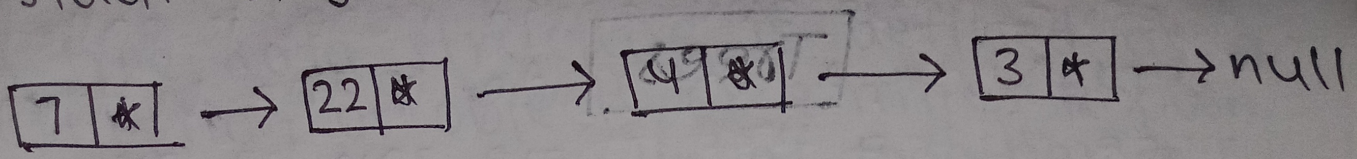
* stackTop = This operation is responsible for returning the topmost element in a stack. Retreiving the topmost element, we use the stack member top to fetch the topmost index & its corresponding element.

```
int stackTop(struct stack *sp)
{
  return sp->arr[sp->top];
}
```

* stackBottom = This operation is responsible for returning the bottom most element in a stack, i.e., the element at 0 index.

```
int stackBottom(struct stack *sp)
{
  return sp->arr[0];
}
```

* Stack using Linked List.



head side

Here, we consider the head side as the top of the stack, where we done our push(), pop() & all other operations.

3	7	top = 3
2	22	
1	4	
0	3	

stack

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

Structure of a node in a Linked List.