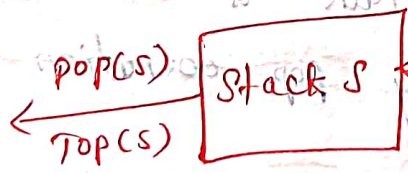
STACK

* A stack is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the top.

* The fundamental operations on a stack are

  * push – which is equivalent to an insert
  * pop – which deletes the most recently inserted element.

Stacks are sometimes known as Last in first out lists. The model of stack depicted in following fig.

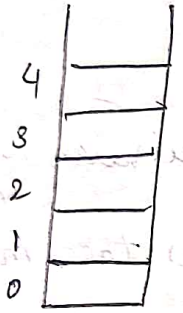


It signifies only that pushes are input operations and pops and Tops are output

Push operation :-

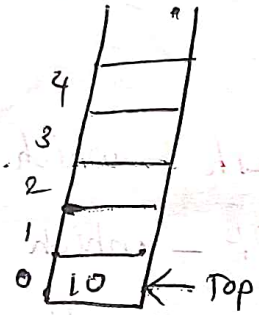
The process of inserting a new element to the top of the stack. For every push operation the top is incremented by 1.

Example :



Empty stack

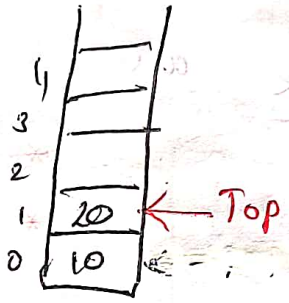
Top = -1



Insert (10)

Top = Top++

Top = 0



Insert 20

Top++

Pop Operation

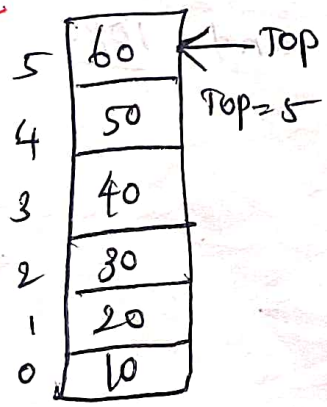
The process of deleting an element from top of the stack is called pop operation. For every pop operation Top decremented by 1.

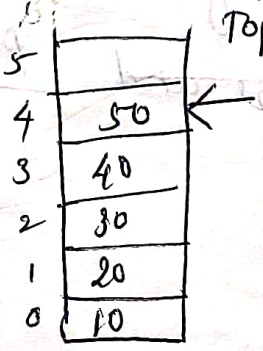
Pop(S) ;

Top -- ;

Top = 4

Example





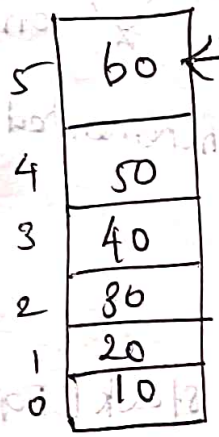
Exceptional Conditions

① overflow :-

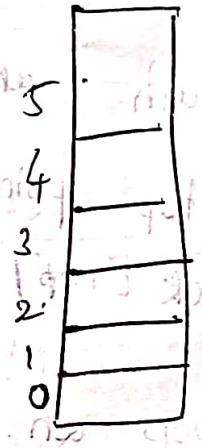
Attempt to insert an element when stack is full is said to be overflow.

② Underflow :-

Attempt to delete an element when the stack is empty.



Push (70) in above stack cases overflow ie) already the stack is full can't insert next item.



Top = -1

pop(5)

There is no items present in above stack, so can't delete an items This situation is called stack underflow

Stack can be implemented in two ways

* Array Implementation
* Linked List Implementation.

Array Implementation:-

Each Stack is associated with a Top pointer which is -1 for an empty stack.

To push an element 'x' on to the stack top pointer is incremented then set Stack[Top] = x

To pop an element Stack[Top] value is returned and the Top pointer is decremented.

Stack Declaration:-

```
Stack stackRecord;
typedef struct stackRecord *stack;
int IsEmpty (Stack S);
int Isfull (Stack S);
Stack CreateStack (int maxelement);
```

```
Void DisposeStack(Stack S);
Void makeempty(Stack S);
Void push(ElementType x, Stack S);
ElementType Top(Stack S);
Void Pop(Stack S)
ElementType TopandPop(Stack S);
```

```
#define EmptyTos (-1)
#define minstacksize (5)
struct StackRecord
{
  int capacity;
  int TopofStack;
  ElementType *Array;
};
```

Routine to create a stack

```
Stack CreateStack(int MaxElements)
{
  Stack S;
  if(maxElements < minstacksize)
  Error("stack size is too small");
  S = malloc(sizeof(Struct StackRecord));
```

```
if (S==NULL)
Fatal Error ("out of space");
S→Array = malloc (sizeof(Element Type) *
                  maxElements);
if (S→Array == NULL)
Fatal Error("out of space");
S→Capacity = maxElements;
makeEmpty (S);
return (S);
}
```

MakeEmpty Routine

```
Void MakeEmpty (Stack S)
{
  S→TopOfStack = EmptyTos;
}
```

EmptyTOS = -1 ie) S→TopOfStack = -1

Routine to dispose or to free a stack

```
Void disposeStack (Stack S)
{
  if (S!=NULL)
    {
```

```
        free (S -> Array);
        free (S);
    }
}
```

Routine IsEmpty

```
int IsEmpty (Stack S)
{
return S -> TopofStack == Empty TOS;
}
```

Routine to insert an element into stack

```
Void push (Element X, Stack S)
{
    if (Isfull (S))
    Error ("Full Stack");
    else
    S -> Array [++S -> TopOfStack] = X;
}
```

IsFull Routine

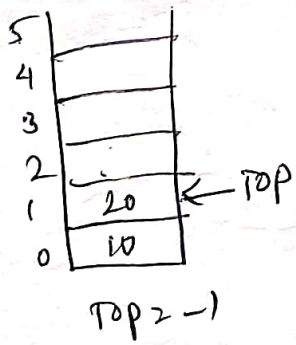
```
int IsFull (Stack S)
{
  if (S->capacity == S->Tos)
    return 1;
  else
    return 0;
}
```

Routine to return Top of the stack

```
ElementType Top (Stack S)
{
  if (!IsEmpty (S))
    return S->Array [S->TopOfStack];
  else
    Error("Empty Stack");
    return 0;
}
```

Example Insert Element (30) into the following stack

Push (30)



Top = 1

Push (30) X = 30 & S are arguments,

Before inserting an element into stack first check whether the stack is full or not. if it is full displays an error message, otherwise insert an element.

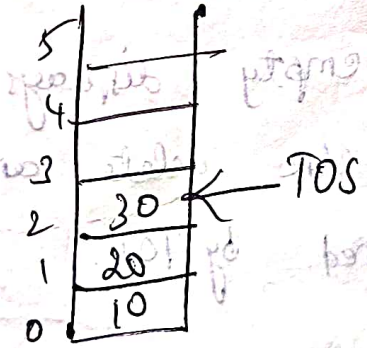
IsFull (S) capacity 6

check S → capacity == S → TOS

6 == 1 X

S → Array [1 + S → TopofStack] = 30

S → Array [2] = 30



Routine to delete an element from stack

```
Void Pop (stack S)
{
    if (Is empty (S))
        Error ("Empty stack");
```

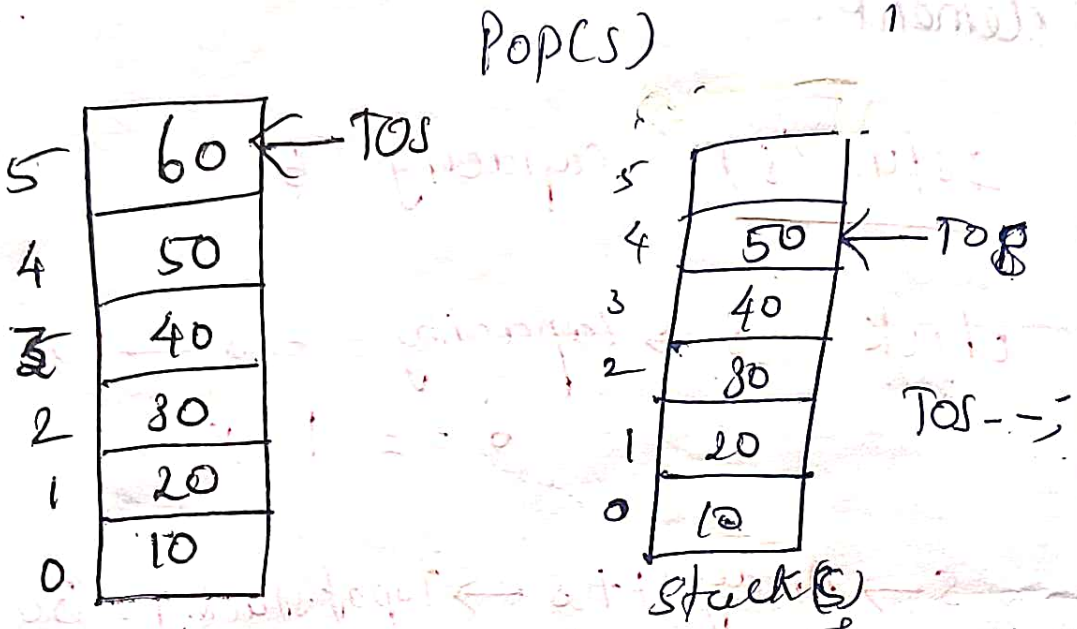
else

S → Top Of Stack - - ;

}

Example

Delete an Element from given stack.



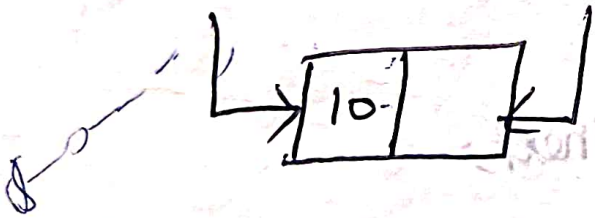
Before delete an element from stack check whether stack is empty (or) not. If its empty displays an error message or else delete an element currently pointed by TOS.

S → Array [S → Top of Stack - -] ;

Linked List Implementation of stack

* push operation is performed by inserting at front of the list.
* Pop operation is performed by deleting the front of the list.
* Top operation returns the element at front of the list.

Example :-



Routine to check whether the stack is empty

```
int Isempty (stack s)
{
  if (s -> next == NULL)
  return (1);
}
```

Routine to create empty stack

```
Stack CreateStack ()
{
  stack s;
  s = malloc (sizeof (struct node));
  if (s == NULL)
  Error ("out of space");
  Makeempty (s);
  return (s);
}
```

Make Empty ()

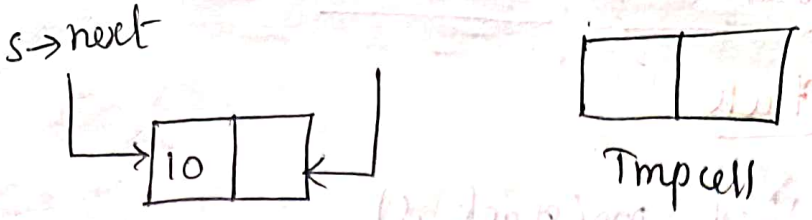
```
Void makeempty (stack s)
{
  if (s == NULL)
```

```
    Error ("Create stack first");
    else
    while ( !Isempty (S))
    pop(S);
}
```

Routine to push an element in to Stack

```
Void push (int x, Stack S)
{
 Struct node *Tmpcell;
 Tmpcell = malloc (sizeof (Struct Node));
 if(Tmpcell == NULL)
 Error ("out of space");
 else
 {
  Tmpcell -> Element = x;
  Tmpcell -> next = S -> next;
  S -> next = Tmpcell;
 }
}
```

Example

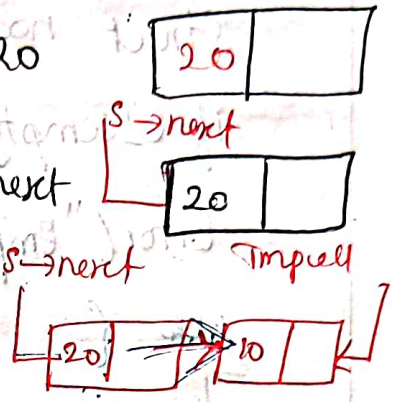


Push (20) X=20 Tmpcell != NULL

Tmpcell → Element = 20

Tmpcell → next = S→next

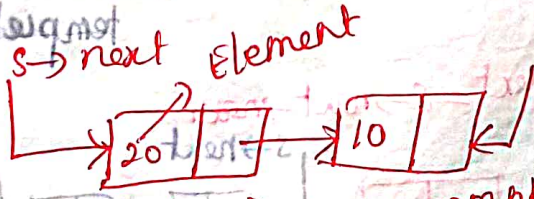
S→next = Tmpcell



Routine to return Top element of the stack

```
int Top (Stack S)
{
  if (!IsEmpty(S))
    return S->next->element;
  else
    Error("Empty stack");
  return 0;
}
```

Ex

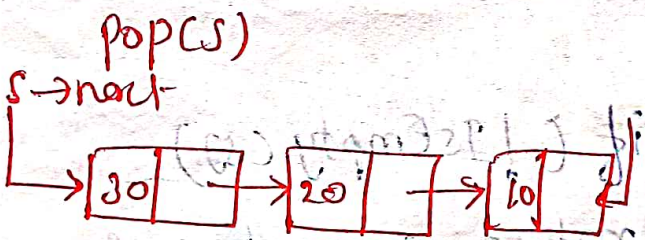


check if stack is not empty it returns S→next→element = 20

Routine to delete an element from stack

```
Void pop(Stack S)
{
  Struct node *tempcell;
  if (Isempty (S))
  Error("Empty stack");
  else {
  tempcell = S→next;
  S→next = S→next→next;
  free(tempcell);
  }
}
```

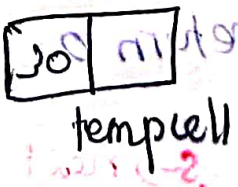
Example



check whether stack is empty or not

If its not "empty then

tempcell = S→next



S→next = S→next→next