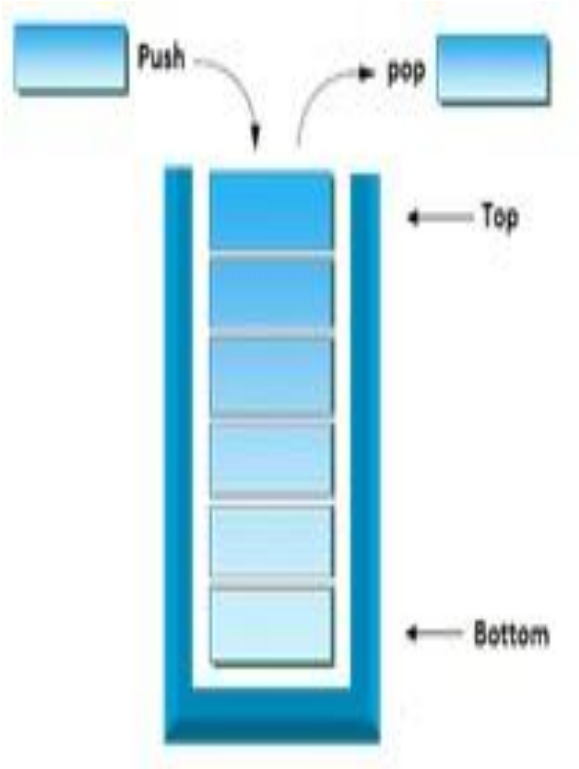
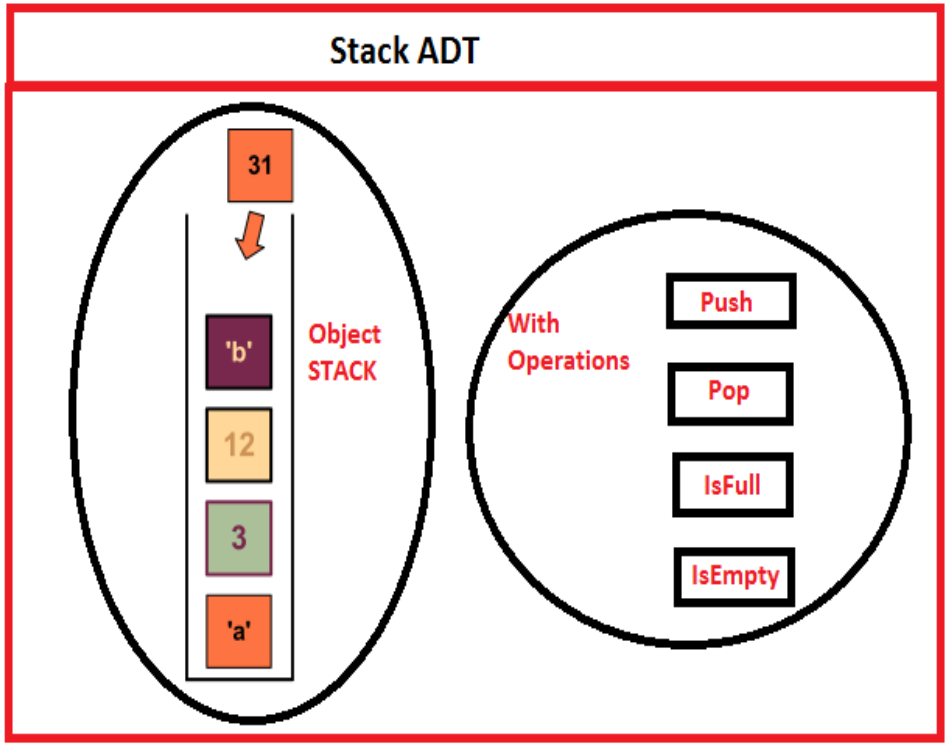
Guess??????

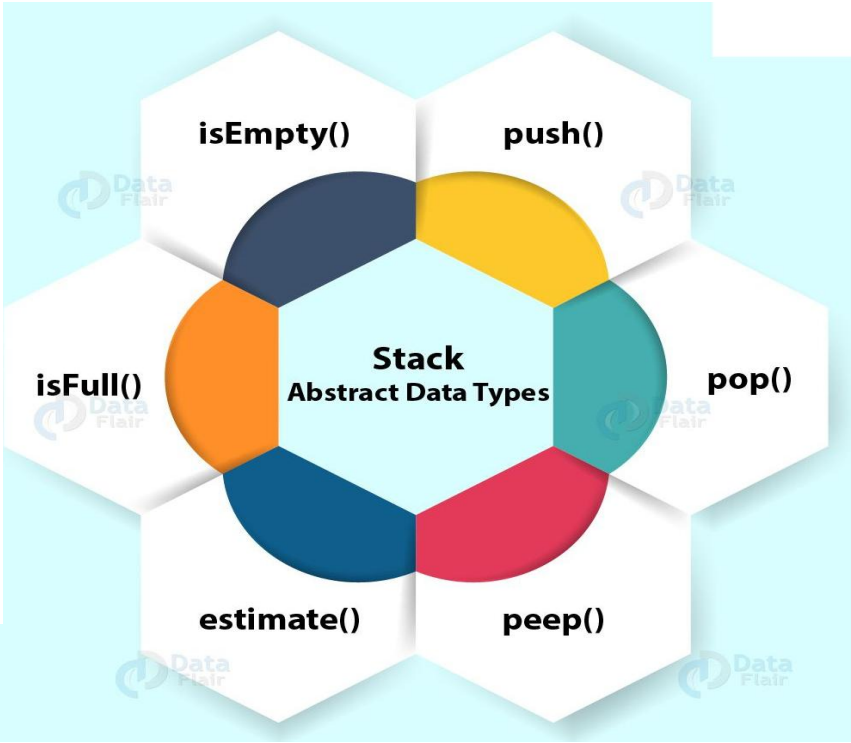
Stack


A stack of cafeteria trays
A stack of pennies
A stack of shoe boxes
A stack of neatly folded shirts


Push
pop
Top
Bottom

Stack ADT


Stack ADT
31
'b'
12
3
'a'
Object
STACK
With
Operations
Push
Pop
IsFull
IsEmpty


isEmpty()
push()
isFull()
Stack
Abstract Data Types
pop()
estimate()
peep()

Operation on Stack

- PUSH
- POP
- IsFull
- IsEmpty

Exceptional Conditions

- Overflow
- Underflow

Implementation of Stack Using

- Linked List
- Array

Implementation of Stack
Using
Linked List

Type Declaration For Stack using Linked List

```
#ifndef _Stack_h

struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

int IsEmpty( Stack S );
Stack CreateStack( void );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );

#endif  /* _Stack_h */


/* Place in implementation file */
/* Stack implementation is a linked list with a header */
struct Node
{
    ElementType Element;
    PtrToNode   Next;
};
```

Figure 3.39 Type declaration for linked list implementation of the stack ADT

To check Whether Stack is Empty

```
int
IsEmpty( Stack S )
{
    return S->Next == NULL;
}
```

Figure 3.40 Routine to test whether a stack is empty—linked list implementation

To Create an Empty List

```
Stack
CreateStack( void )
{
    Stack S;

    S = malloc( sizeof( struct Node ) );
    if( S == NULL )
        FatalError( "Out of space!!!" );
    MakeEmpty( S );
    return S;
}

void
MakeEmpty( Stack S )
{
    if( S == NULL )
        Error( "Must use CreateStack first" );
    else
        while( !IsEmpty( S ) )
            Pop( S );
}
```

Figure 3.41 Routine to create an empty stack—linked list implementation

To Push onto Stack

```
void
Push( ElementType X, Stack S )
{
    PtrToNode TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
        FatalError( "Out of space!!!" );
    else
    {
        TmpCell->Element = X;
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
    }
}
```

Figure 3.42 Routine to push onto a stack—linked list implementation

To Return TOP element in a Stack

```
ElementType
Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Next->Element;
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.43 Routine to return top element in a stack—linked list implementation

To POP from a stack

```
void
Pop( Stack S )
{
    PtrToNode FirstCell;

    if( IsEmpty( S ) )
        Error( "Empty stack" );
    else
    {
        FirstCell = S->Next;
        S->Next = S->Next->Next;
        free( FirstCell );
    }
}
```

Figure 3.44 Routine to pop from a stack—linked list implementation

Implementation of Stack

Using

Array

Stack Declaration For Stack using Array

```
#ifndef _Stack_h

struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty( Stack S );
int IsFull( Stack S );
Stack CreateStack( int MaxElements );
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 );

#endif  /* _Stack_h */

/* Place in implementatioin file */
/* Stack implementation is a dynamically allocated array */
#define EmptyTOS ( -1 )
#define MinStackSize ( 5 )

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
```

Figure 3.45 *Stack* declarations—array implementation

Stack Creation

```
Stack
CreateStack( int MaxElements )
{
    Stack S;

/* 1*/      if( MaxElements < MinStackSize )
/* 2*/          Error( "Stack size is too small" );

/* 3*/      S = malloc( sizeof( struct StackRecord ) );
/* 4*/      if( S == NULL )
/* 5*/          FatalError( "Out of space!!!" );

/* 6*/      S->Array = malloc( sizeof( ElementType ) * MaxElements );
/* 7*/      if( S->Array == NULL )
/* 8*/          FatalError( "Out of space!!!" );
/* 9*/      S->Capacity = MaxElements;
/*10*/      MakeEmpty( S );

/*11*/      return S;
}
```

Figure 3.46 Stack creation—array implementation

Freeing & Checking Empty - Stack

```
void
DisposeStack( Stack S )
{
    if( S != NULL )
    {
        free( S->Array );
        free( S );
    }
}
```

Figure 3.47 Routine for freeing stack—array implementation

```
int
IsEmpty( Stack S )
{
    return S->TopOfStack == EmptyTOS;
}
```

Figure 3.48 Routine to test whether a stack is empty—array implementation

To Create an Empty stack & To Push onto a stack

```
void
MakeEmpty( Stack S )
{
    S->TopOfStack = EmptyTOS;
}
```

Figure 3.49 Routine to create an empty stack—array implementation

```
void
Push( ElementType X, Stack S )
{
    if( IsFull( S ) )
        Error( "Full stack" );
    else
        S->Array[ ++S->TopOfStack ] = X;
}
```

Figure 3.50 Routine to push onto a stack—array implementation

Top & POP

```
ElementType
Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack ];
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.51 Routine to return top of stack—array implementation

```
void
Pop( Stack S )
{
    if( IsEmpty( S ) )
        Error( "Empty stack" );
    else
        S->TopOfStack--;
}
```

Figure 3.52 Routine to pop from a stack—array implementation

To Give TOP element & POP a Stack

```
ElementType
TopAndPop( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack-- ];
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.53 Routine to give top element and pop a stack—array implementation