What is Data structure

- In computer science ,a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.

- A data structure is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees,.

What is the advantage of using Data structure?

- Control and organization the data inside the memory
- Build a strong and coherent programs
- it gives the user a lot of best way to write different programs
- Reduce the time of storing and retrieving the data it from the memory
- it tells how data can be stored and accessed at elementary level.

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced)	Deletion algorithm is complex.

A Real Life Example

Electronic Phone Book
Contains different **DATA**:
- names
- phone number
- addresses

Need to perform certain **OPERATIONS**:
- add
- delete
- look for a phone number
- look for an address

How to organize the data so to optimize the efficiency of the operations

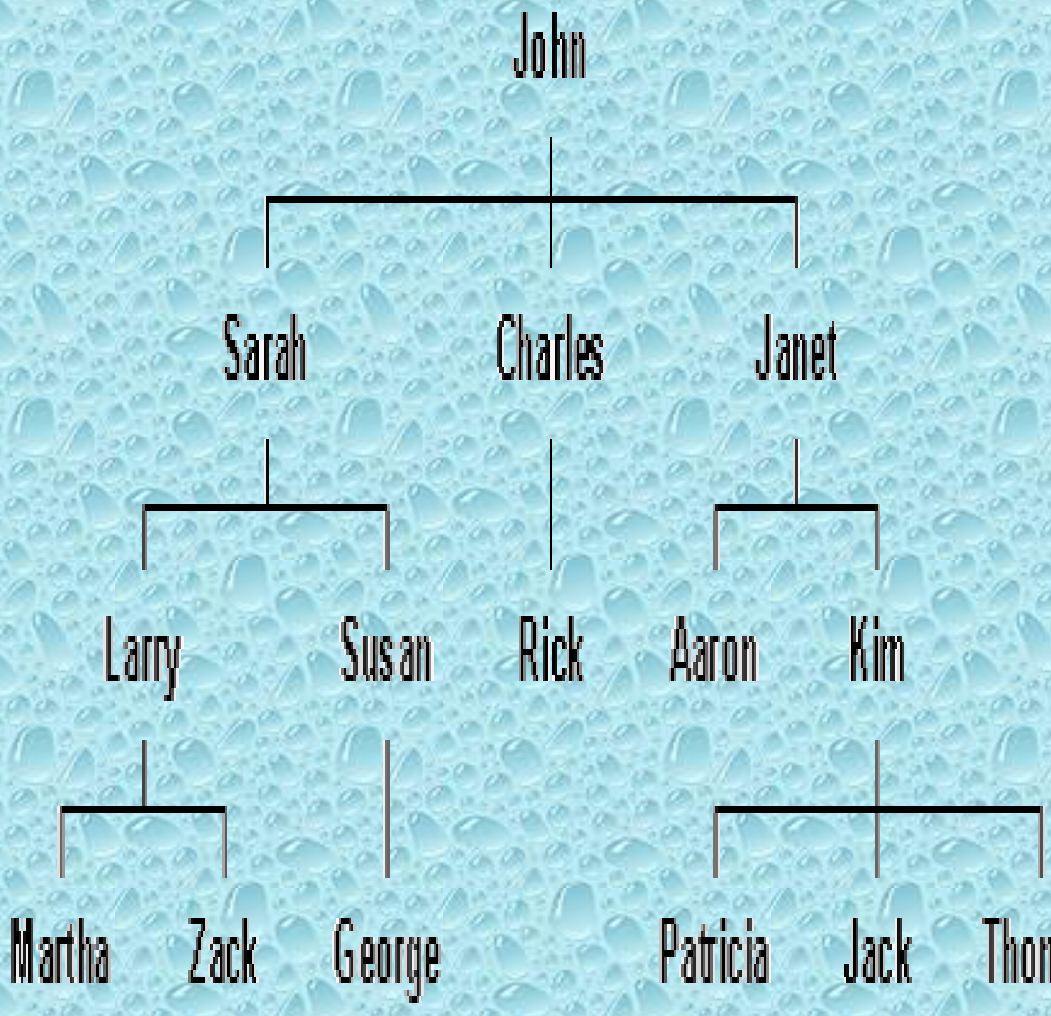
- **Lisa**
- **Michele**
- **John**
- **110**
- **622-9823**
- **112-4433**
- **75**
- **Bronson**
- **Paola**

Another Real Life Example

Algorithms, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting

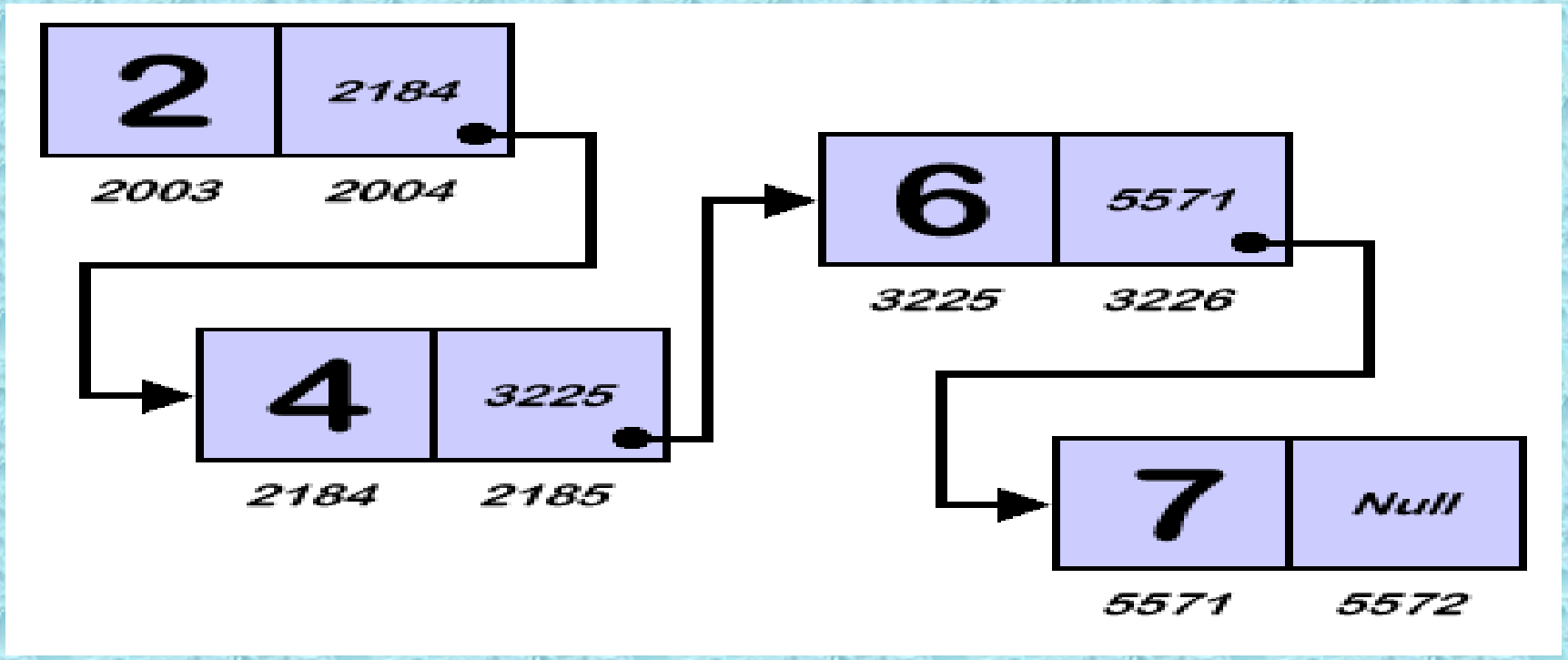
Name	Position
Aaron	Manager
Charles	VP
George	Employee
Jack	Employee
Janet	VP
John	President
Kim	Manager
Larry	Manager
Martha	Employee
Patricia	Employee

After thinking about the problem for a while. You decide that the tree diagram is much better structure for showing the work relationships at the ABC company.

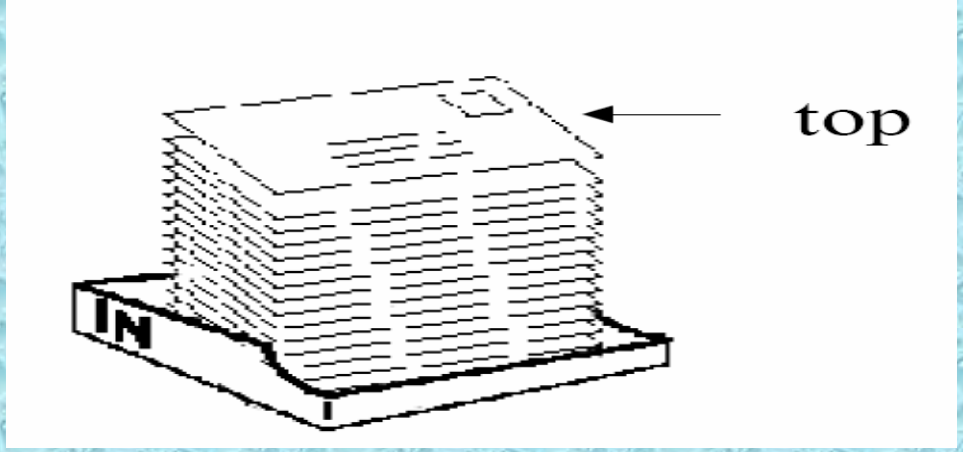


Pointer Implementation

This approach is to creating a list is to link groups of memory cells together using the pointers. Each group of memory cells is called as a node. With this implementation every node contains the data item and the pointer to the next item in the list. You can picture this structure as a chain of nodes linked together by the pointers. As long as we know where the chain begins, we can follow the links to reach any item in the list



The Stack



- A stack is a pile of item kept one over another, i.e. A stack only have a head or top no bottom.

- Any item added becomes the new top of the stack. And any item deleted or takeout from the stack is also taken out from the top & the top of Stack get reduced by one.

- That is the first item which goes in to the stack will come out of the stack the last as the last item. From this we conclude that a stack is a LIFO (last in first out) Stack is not the same as array.

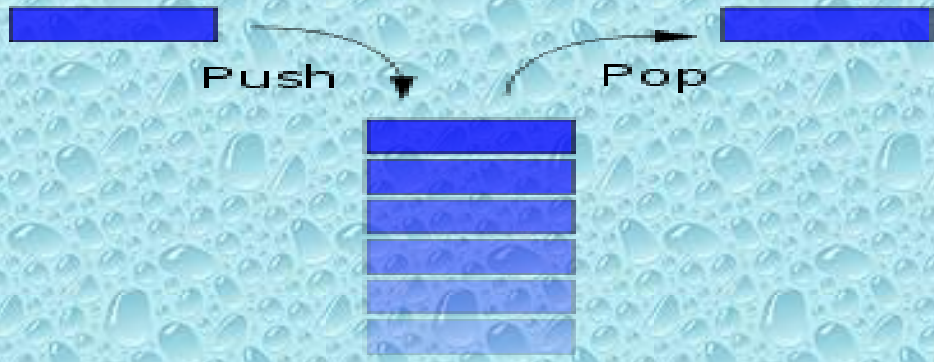
Cont. The Stack

- From the definition we found that stack provide provisions for adding of items and deleting of item from it, But it can be done from only with on end which is the top of the stack.

- A stack doesn't have tail or the end point. When we add any item to stack the head of stack goes up. And when we delete any items from a stack the head top) of the stack goes down.

- In Summary: A Stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. The other name of stack is Last-in -First-out list.

Operation of Stack



- From the above explanation we can fairly understand that stack can do only two work i.e. adding of item and deleting of item from it. We now points to the operation of stacks

  (1) Push : It is the operation by which we can add any item to a stack. This operation required the item to be added and a stack in which the item will be added .

  (2) Pop : It is the operation by which We can delete the upper item from the stack and this operation gives us the item that it is deleted from the stack. In other words this operation is used to take out the upper item from any stack .

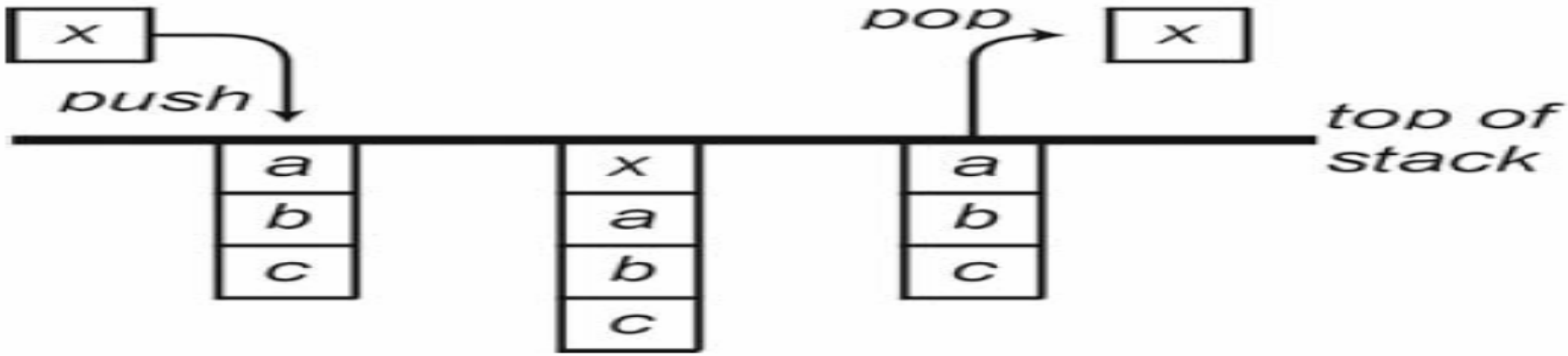
  (3) IsEmpty : This operation is used to check whether the stack we are using is empty or filled by any item. If the stack is empty this operation gives us as appropriate answer

.Code of Stack in C

- **Top():** return the top most element in the stack without removing it , if the stack is empty an errors return.

- We only check the top variable **if it is equal to zero then your stack is empty**. If it is **equal to the maximum** value of stack member then **the stack is full**.

- **Size():** return the number of element in the stack at any time

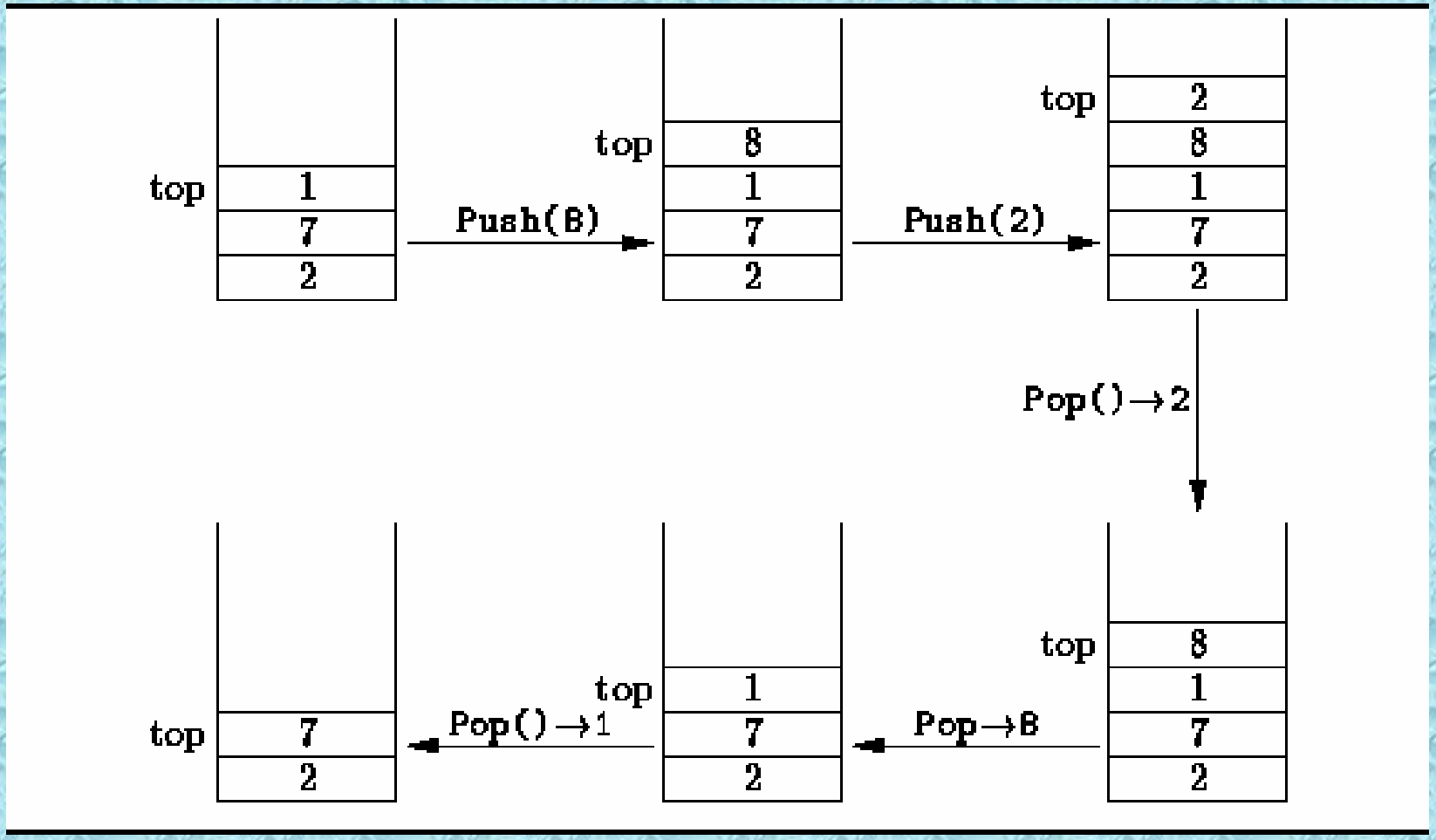


Increase and decrease Top element

- If we want to add any item to stack we **increase the value of top by one** and add any item to the stack array with the **increased top as the index of the array**.

- If we want **to delete or want to Pop** any item from the stack **we decrease the top by one and return the current or top item** of the stack array .

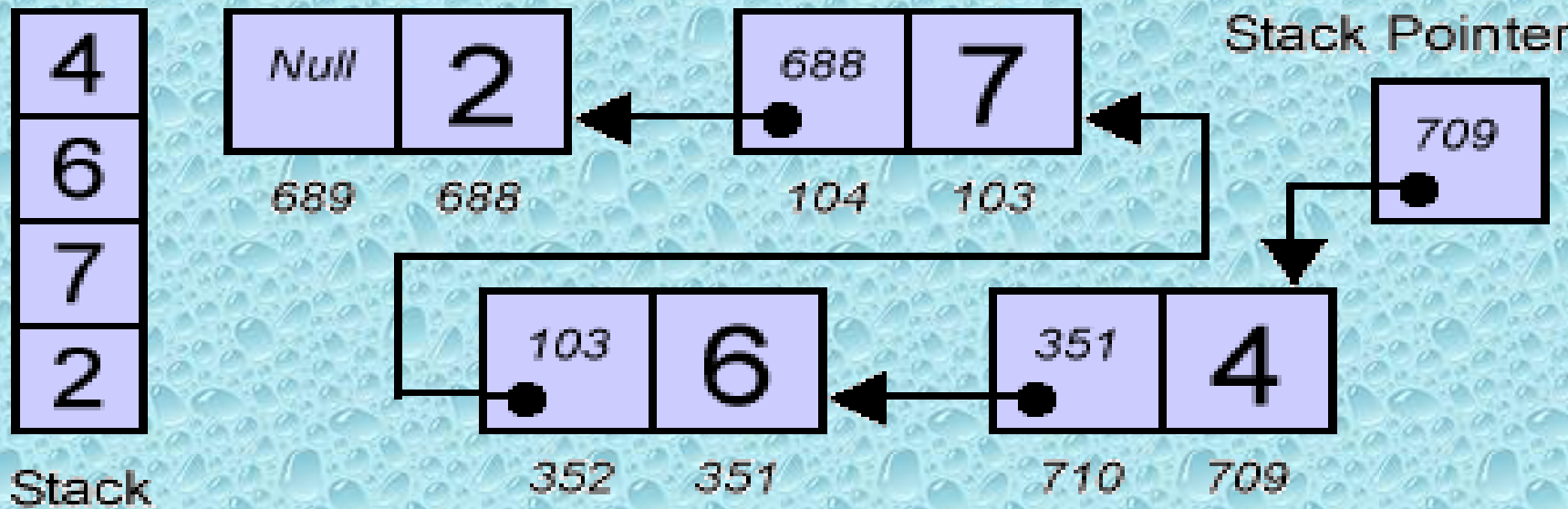
Example of inserting and deleting 5 numbers



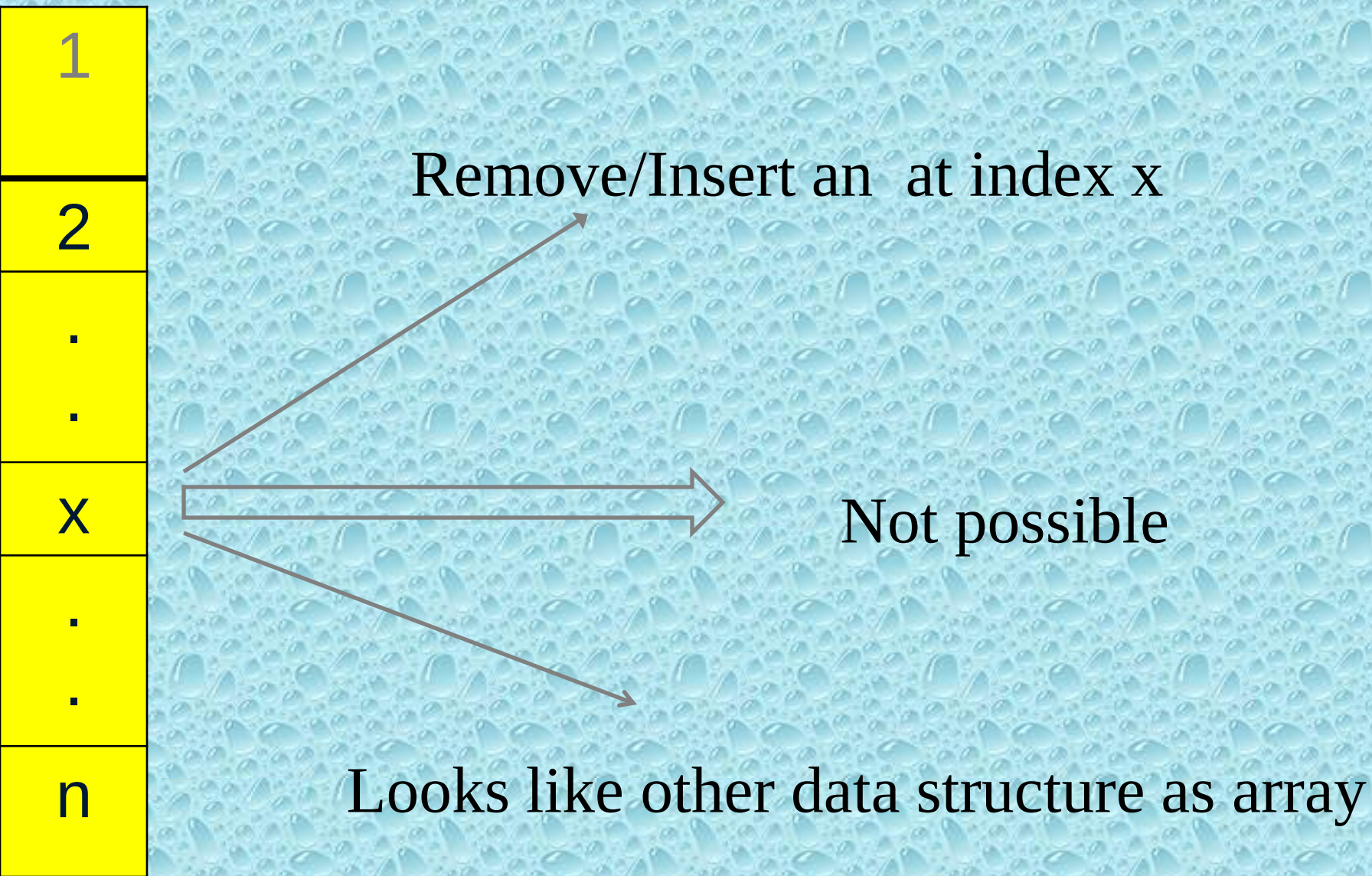
Pointer Implementation in the Stack

In order to implement a stack using pointers, we need to link nodes together just like we did for the pointer implementation of the list.

Each node contains the stack item and the pointer to the next node. We also need a special pointer(stack pointer) to keep track of the top of our stack.



Selective Removal Operation

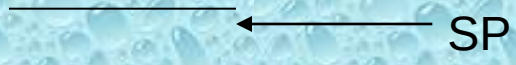


Stack Operations implemented by Array of size 3

Size()=0

Isempty()=1

Top()=-1, Null



Push(5)

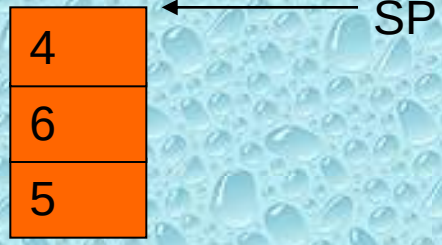
Size()=1
Isempty()=0
Top()=0
ack[top]=5

Push(6)

Size()=2
Isempty()=0
Top()=1
Stak[top]=6

Push(4)

Size()=3
Isempty()=0
Top()=2

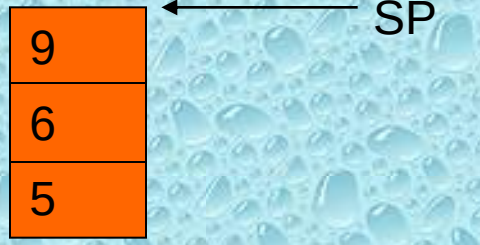


Pop()

Size()=2
Isempty()=0
Top()=1

Push(9)

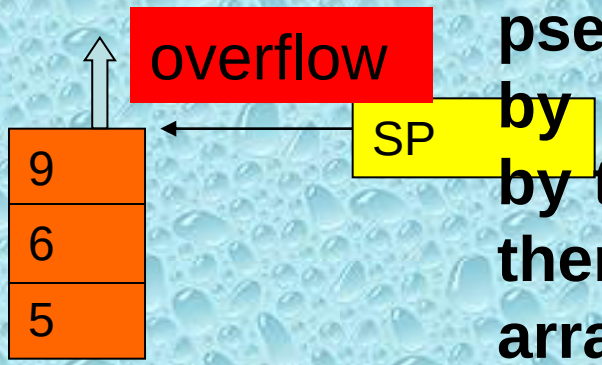
ize()=3
Isempty()=0
Top()=2



Cont. Stack Operations

Push(2)

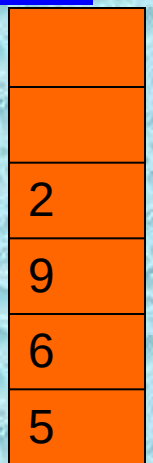
Size()=3
Isempty()=0
Top()=2



Push will return error, in this case pseducode will deal with this case by increasing the size of the array by the library function realloc() and then copy the elements to the new array and increase the new element

Push(2)

Size()=4
Isempty()=0
Top()=3
Stack[top]=2



- **The drawback of this method is the consuming time , and losing time and wastage of space**
- **There is other way to implement the stack using Linked list**

Stack Operations implemented by linked list

Size()=0

Isempty()=1

Top()=-1, Null

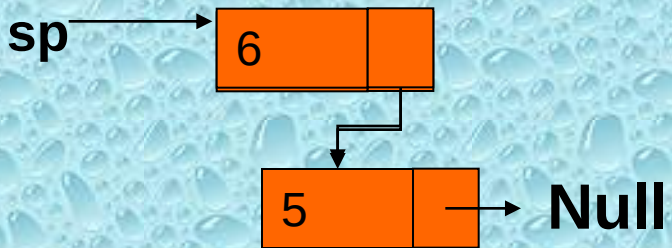
Push(5)

Size()=1

Isempty()=0

Top()=0

Push(6)



Size()=2

Isempty()=0

Top()=1

- **Advantage:**
  - **No wastage of space**
  - **No upper limit size**
- **Disadvantage**
- **The cost of the node creation**
  - **Push() and Pop() take time as node creation and deletion take more list**

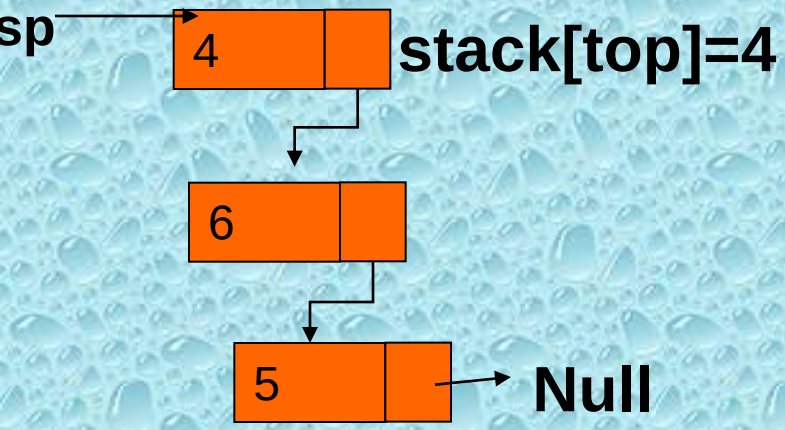
Stack Operations implemented by linked list

Size()=3

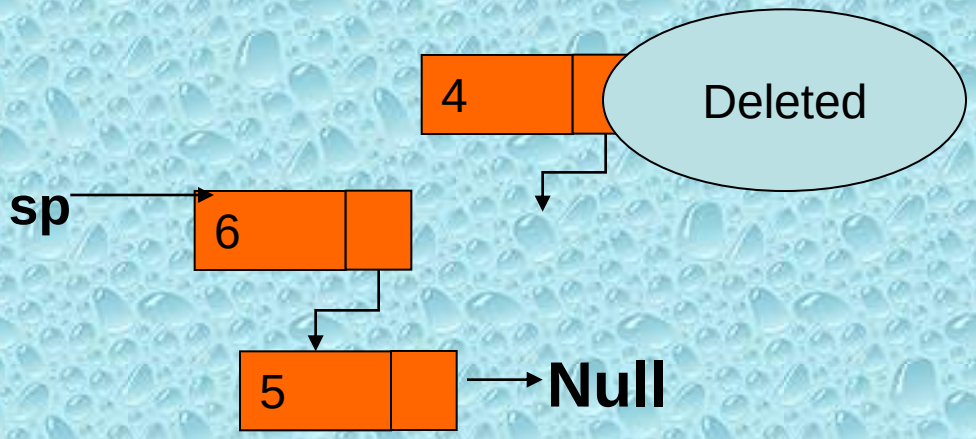
Isempty()=0

Top()=2

Push(4)



Pop()



Size()=2

Isempty()=0

Top()=1

STACK USING ARRAY

```
      /******    Program to Implement Stack using Array    ******/

#include <stdio.h>

#define MAX 50

void push();
void pop();
void display();

int stack[MAX], top=-1, item;

main()
{
     int ch;
     do
     {
          printf("\n\n\n\n1.\tPush\n2.\tPop\n3.\tDisplay\n4.\tExit\n");
          printf("\nEnter your choice: ");
          scanf("%d", &ch);

          switch(ch)
          {
               case 1:
                    push();
                    break;

               case 2:
                    pop();
                    break;
```

```
case 3:
        display();
        break;
```

```
            case 4:
                exit(0);

            default:
                printf("\n\nInvalid entry. Please try again...\n");
        }
    } while(ch!=4);
    getch();
}
 void push()
 {
      if(top == MAX-1)
            printf("\n\nStack is full.");
      else
      {
            printf("\n\nEnter ITEM: ");
            scanf("%d", &item);

            top++;
            stack[top] = item;
            printf("\n\nITEM inserted = %d", item);
      }
 }
```

```
void pop()
{
      if(top == -1)
            printf("\n\nStack is empty.");
      else
      {
            item = stack[top];
            top--;
            printf("\n\nITEM deleted = %d", item);
      }
}

void display()
{
      int i;
      if(top == -1)
            printf("\n\nStack is empty.");
      else
      {
            for(i=top; i>=0; i--)
                  printf("\n%d", stack[i]);
      }
}
```

In Conclusion

- **The simple algorithm uses a stack and is as follows:**
- Make an empty stack. Read characters until end of file. If the character is an open anything, push it onto the stack.
- If it is a close anything, then if the stack is empty report an error.
- Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error.
- At end of file, if the stack is not empty report an error.

Question?

Count the max number in stack, by C?

Write a program using Pointer?

Count the max number in stack, changing only in pop function

```
#include<iostream.h>
#include<conio.h>
int size=10;
int a[10],top=0;
int pop();
void push(int[],int);
main()
{     int i,k;
for(i=0;i<size;i++)
{cin>>k;
push(a,k);}
cout<<"THE MAX VAL = "<<pop();
getch();}
```

Cont. max number

```
void push(int a[],int k)
{
if(top==size-1)   cout<<" FULL STACK";
else
a[top++]=k;
}
int pop()
{
int i,  max=a[top--];
for (;;)
{    if(top=0)      break;
else
if(max<a[top])
max=a[top];
top--; }  return max;
}
```

Applications :**Balancing Symbols**

- Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.
- A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to their left counterparts. The sequence [()] is legal, but [(]) is wrong. Obviously, it is not worthwhile writing a huge program for this, but it turns out that it is easy to check these things.

- For simplicity, we will just check for balancing of parentheses, brackets, and braces and ignore any other character that appears.

Evaluating a Postfix Expression

- You may be asking what a stack is good for, other than reversing a sequence of data items. One common application is to find the value of a postfix expression. Another is to convert an infix expression to postfix. We will not look at the conversion algorithm here, but we will examine the algorithm to evaluate a postfix expression.

- First, let's explain the terminology. An infix expression is the type that we are used to in ordinary algebra, such as 3 + 9, which is an expression representing the sum of 3 and 9. Infix expressions place their (binary) operators between the two values to which they apply. In the above example, the addition operator was placed between the 3 and the 9.

- A postfix expression, in contrast, places each operator after the two values to which it applies. (*Post* means "after", right?) The above expression would be 3 9 +, when rewritten in postfix

Cont. Evaluating a Postfix Expression

- Here are a few more examples in the following table. The infix form is shown on the right, and the postfix form is given on the left .

Infix:	Postfix:
16 / 2	16 2 /
(2 + 14) * 5	2 14 + 5 *
2 + 14 * 5	2 14 5 * +
(6 - 2) * (5 + 4)	6 2 - 5 4 + *

Input String: (((2 * 5) - (1 * 2)) / (11 - 9))

Input Symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
(	(((	
2	(((2	
*	(((2 *	
5		(((2 * 5
)	((10	2 * 5 = 10 and *push*
-	((10 -	
(	((10 - (	
1	((10 - (1	

Input Symbol	Stack (from bottom to top)	Operation
*	((10 - (1 *	
2	((10 - (1 * 2	
)	((10 – 2	1 * 2 = 2 & *Push*
)	(8	10 - 2 = 8 & *Push*
/	(8 /	
(	(8 / (	
11	(8 / (11	
-	(8 / (11 -	
9	(8 / (11 - 9	
)	(8 / 2	11 - 9 = 2 & *Push*
)	4	8 / 2 = 4 & *Push*
New line	Empty	*Pop* & Print

- **Algorithm**
- **1. Read one input character**
- **2. Actions at end of each input**
- **Opening brackets (2.1) *Push* into stack and then Go to step (1)**
- **Number (2.2) *Push* into stack and then Go to step (1)**
- **Operator (2.3) *Push* into stack and then Go to step (1)**
- **Closing brackets (2.4) *Pop* from character stack**

(2.4.1) if it is closing bracket, then discard it, Go to step (1)
(2.4.2) *Pop* is used three times
The first popped element is assigned to op2
The second popped element is assigned to op
The third popped element is assigned to op1
Evaluate op1 op op2
Convert the result into character and
***push* into the stack**
Go to step (2.4)

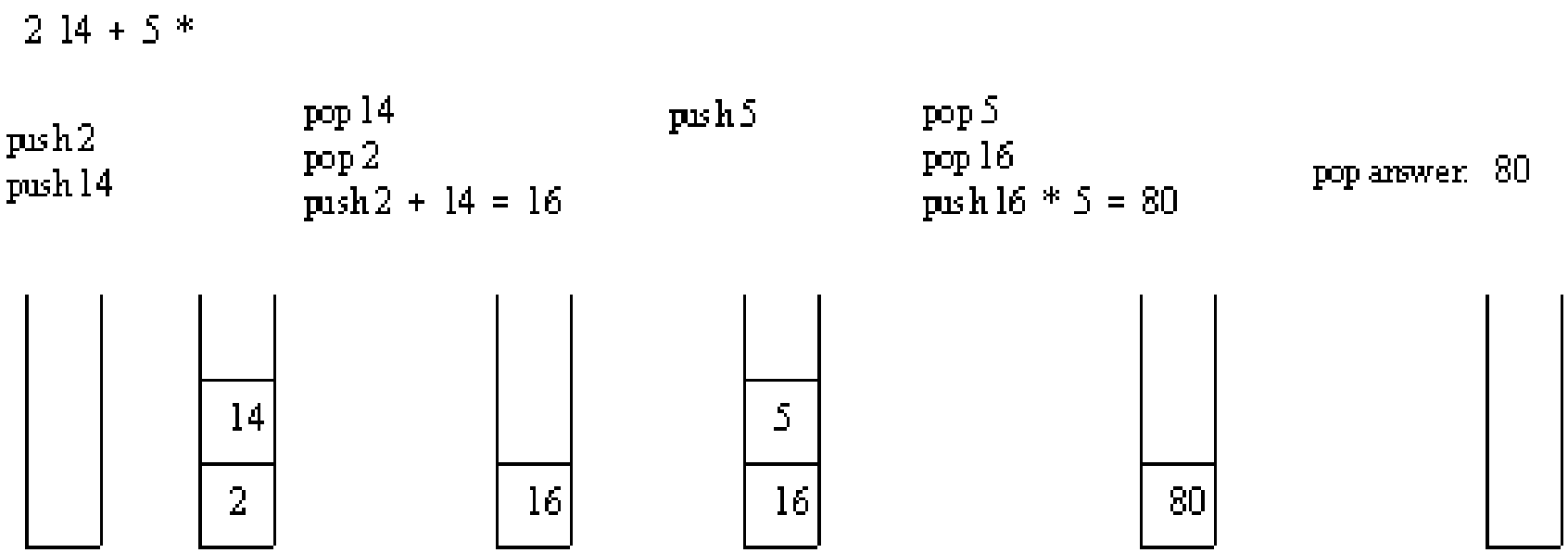
- **New line character (2.5) *Pop* from stack and print the answer**
- ***STOP***

Cont. Evaluating a Postfix Expression

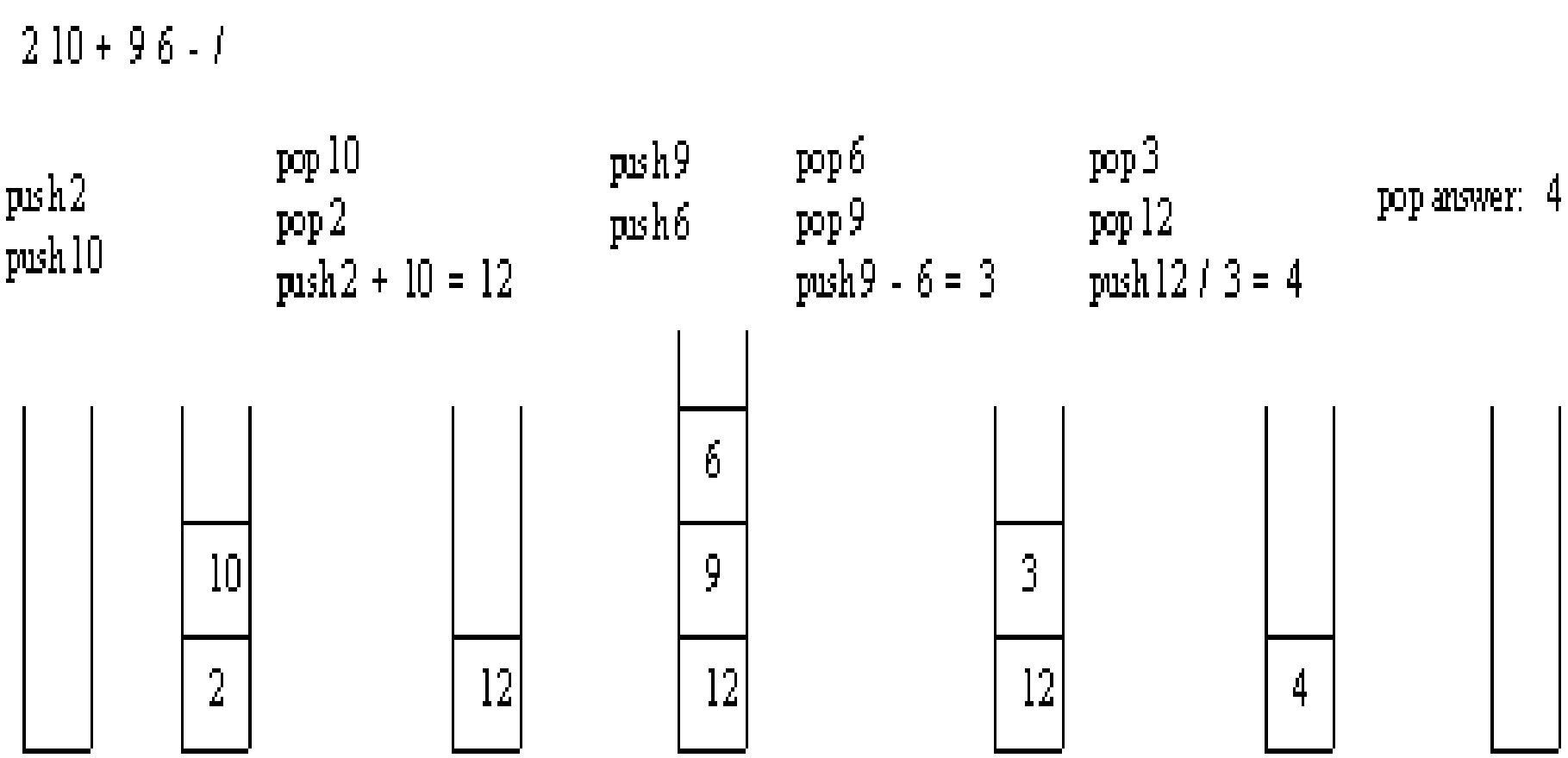
- The algorithm to evaluate a postfix expression works like this: Start with an empty stack of floats. Scan the postfix expression from left to right. Whenever you reach a number, push it onto the stack. Whenever you reach an operator (call it Op perhaps), pop two items, say First and Second ,and then push the value obtained using Second Op First .

- When you reach the end of the postfix expression, pop a value from the stack. That value should be the correct answer, and the stack should now be empty. (If the stack is not empty, the expression was not a correct postfix expression .

- Let's look at the postfix expression evaluation algorithm by way of example. Consider the postfix expression 2 14 + 5 * that was mentioned above.
- We already know from its infix form ,(2 + 14) * 5 ,that the value should be 16 * 5 = 80 .The following sequence of pictures depicts the operation of the algorithm on this example. Read through the pictures from left to right .



- Let's evaluate another postfix expression, say 2 10 + 9 6 - / which is (2 + 10) / (9 - 6) in infix. Clearly the value should work out to be 12 / 3 = 4 .
- Trace through the algorithm by reading the following pictures from left to right .

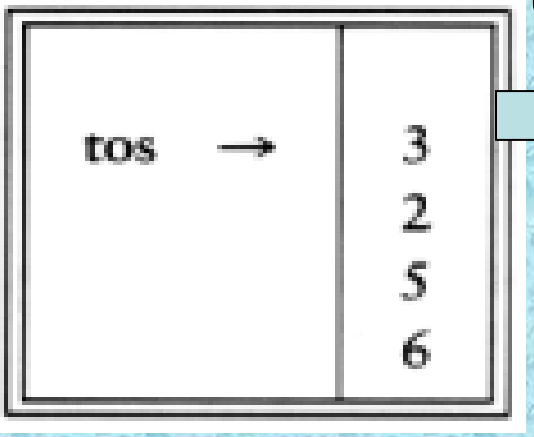


- When one reaches an operator in this algorithm, it is important to get the order right for the values to which it applies. The second item popped off should go in front of the operator, while the first one popped off goes after the operator. You can easily see that with subtraction and division the order does matter .

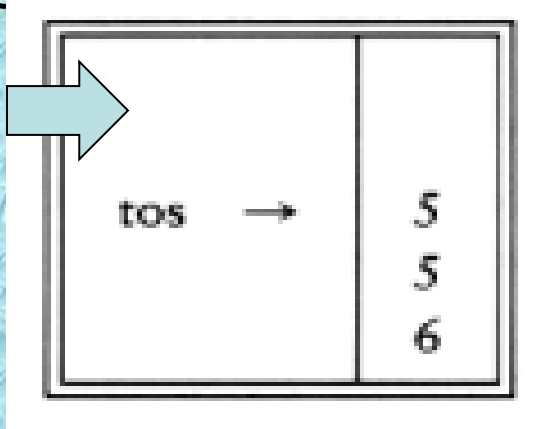
- A good exercise for the reader is to develop a program that repeatedly evaluates postfix expressions. In fact, with enough work, it can be turned into a reasonable postfix calculator .

Another Example

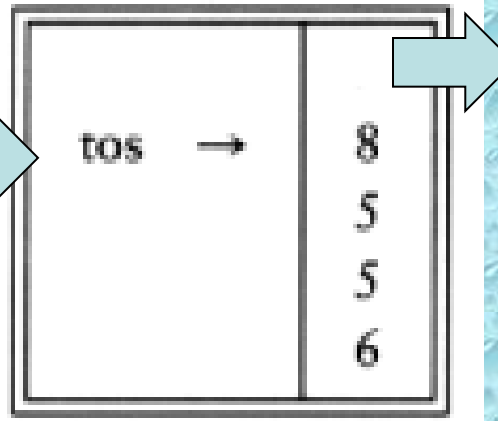
- the postfix expression 6 5 2 3 + 8 * + 3 + * where its infix was: 6*((5+(2+3)*8)+3)
- is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is



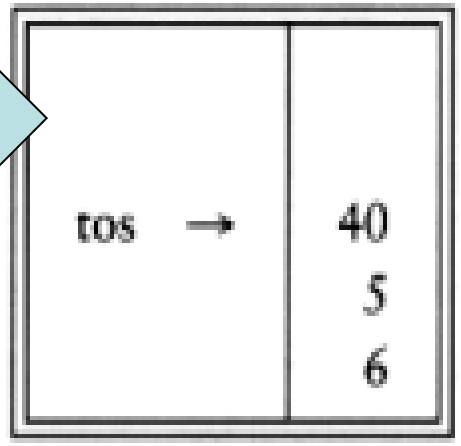
Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.



Next 8 is pushed.

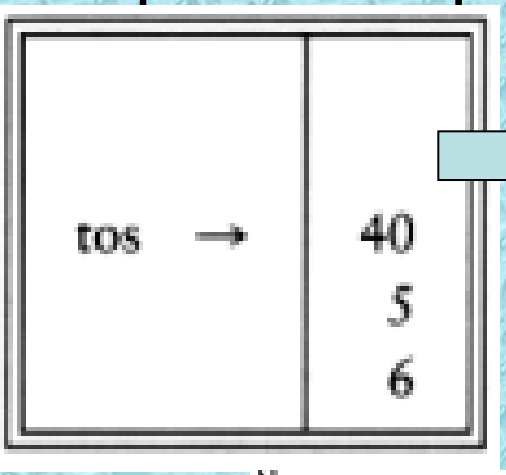


Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed.

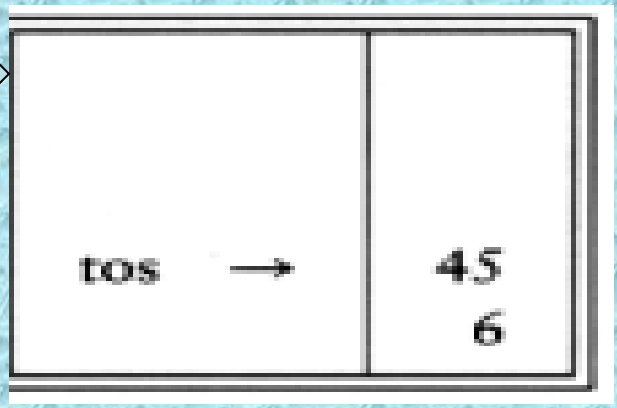


Cont. Another Example

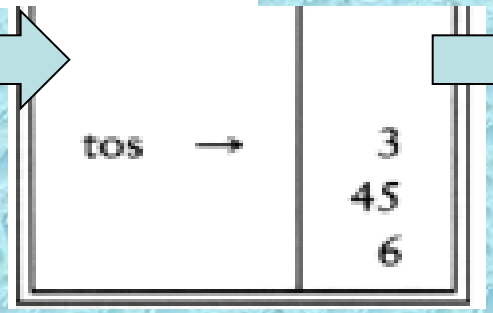
the postfix expression 6 5 2 3 + 8 * + 3 + *



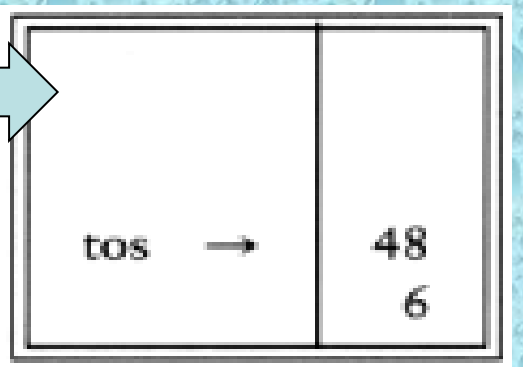
Next a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed.



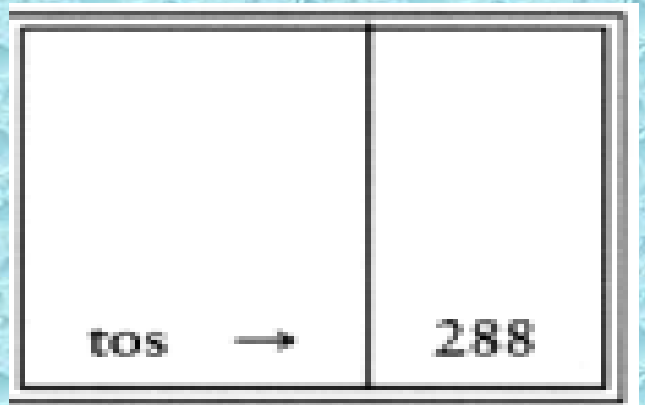
Now, 3 is pushe.



Next '+' pops 3 and 45 and pushes 45 + 3 = 48.



Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed



Checking Prefix and Postfix of Stack in C.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void check(char[]);
main()    {char s[100];
cin>>s;
check(s);
getch();   }
void check(char s[])
{char c;
int i, x, y; x=y=0;
for (i=0;(c=s[i])!='\0';i++){
if(c=='('||c=='[')   x++;
```

Cont. // Checking Prefix and Postfix of Stack in C.

- else
- if(c==')'||c==']') y++;
- if(y>x){ cout<<"ERROR\n"; exit(1);
- }
- }
- if(y>x||x>y)
- {
- cout<<"ERROR\n";
- exit(1); }
- cout<<"ACCEPT \n"; }

Question?

Write a source code of program for Checking Prefix and Postfix of Stack in C using structure?

Write the infix of the postfix expression 6 5 2 3 + 8 * + 3 + *