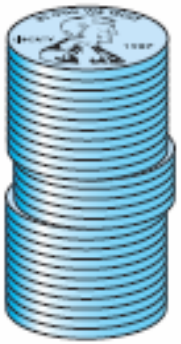
# Chapter Four

# Stacks & Its Applications

# Introduction

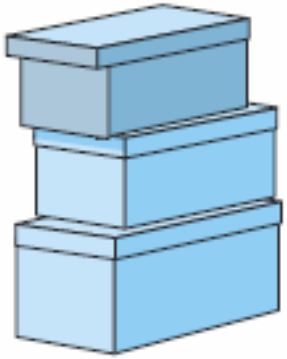
- A stack is a linear data structure that follows the principle of **Last In First Out (LIFO).**
  - This means the last element inserted inside the stack is removed first.
- All deletions and insertions occur at one end of the stack known as the **TOP**.

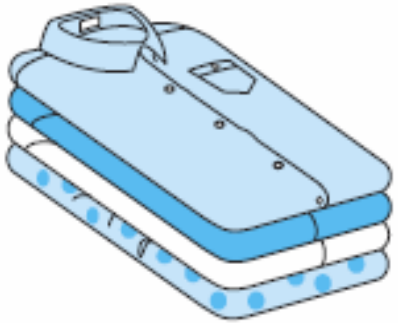
A stack of cafeteria trays



A stack of pennies

A stack of shoe boxes





A stack of neatly folded shirts

# Introduction

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.

- Stack is a more restricted List with the following constraints:
  - Elements are stored by order of insertion from "bottom" to "top".
  - Items are added to the top.
  - Only the last element added onto the stack (the top element) can be accessed or removed.

# Stack Operations

- **push():** adding or putting an element on top of the stack.
  - If the stack is **full,** then the **overflow** condition occurs.
- **pop():** removing or deleting an element from the top of the stack.
  - If the stack is **empty** means that no element exists in the stack, this state is known as an **underflow** state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.
- **peek():** It returns the top data element of the stack without removing it.

# Stack Operations

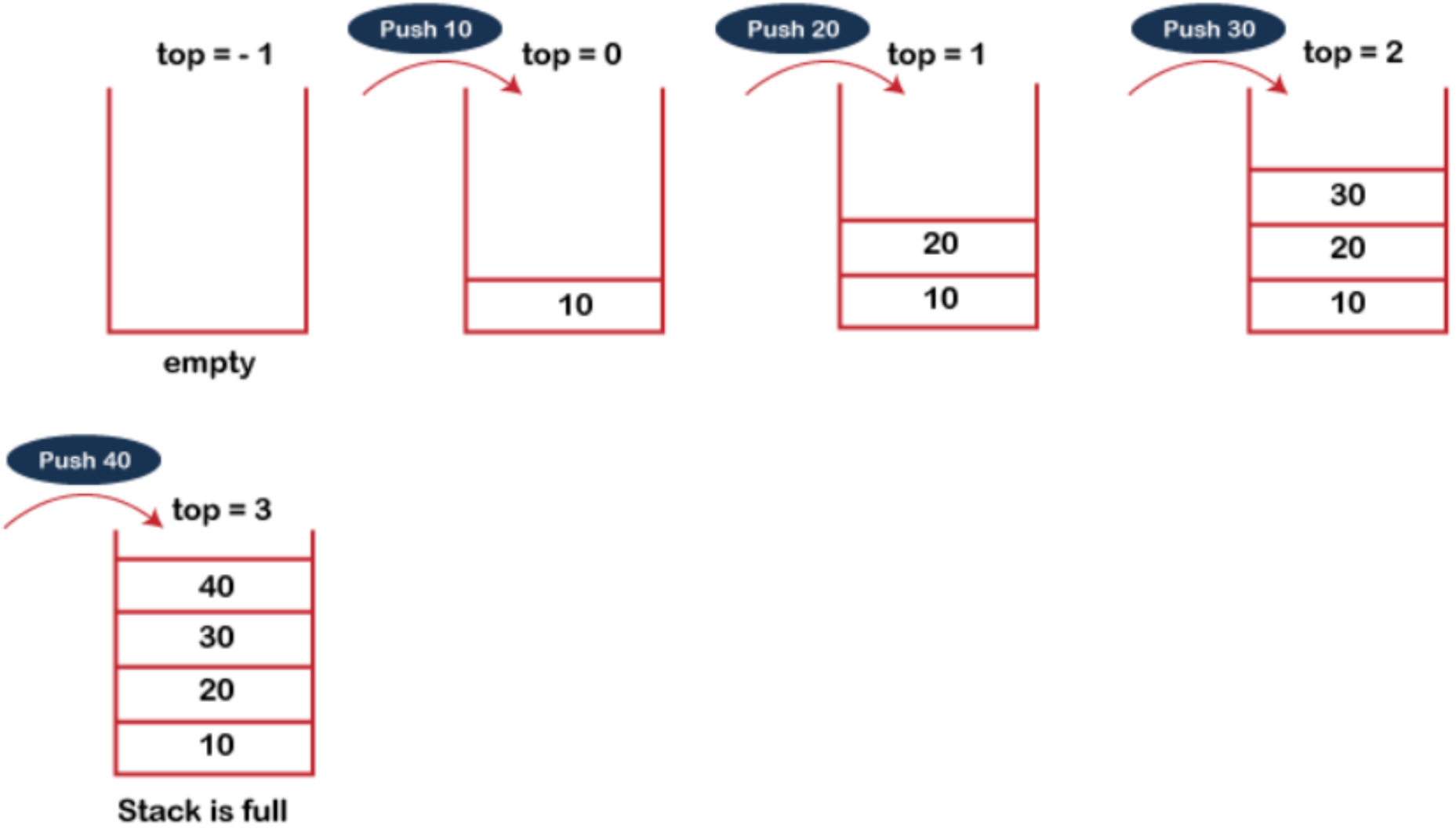
- The operations work as follows:
  - A pointer called TOP is used to keep track of the top element in the stack.
  - When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
  - On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
  - On popping an element, we return the element pointed to by TOP and reduce its value.
  - Before pushing, we check if the stack is already full
  - Before popping, we check if the stack is already empty

# Stack Operations: push()

- The process of putting a new data element onto stack is known as a Push Operation.
- Push operation involves the following steps:
  1. Checks if the stack is full or not.
  2. If the stack is full, produces an error and exit.
  3. If the stack is not full, increments top to point next empty space.
  4. Add the new data element to the stack location, where top is pointing.
  5. Returns success.

NB: If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

# Stack Operations: push()



# Stack Operations: push()

**Algorithm:**

```
begin procedure push: stack, data
   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data
end procedure
```

```
void push(int data) {
   if(!isFull())
{
      top = top + 1;
      stack[top] = data;
   }
else
{
cout<<"\nStack is full\n");
   }
}
```

# Stack Operations: pop()

- Accessing the content while removing it from the stack, is known as a Pop Operation.

- In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.

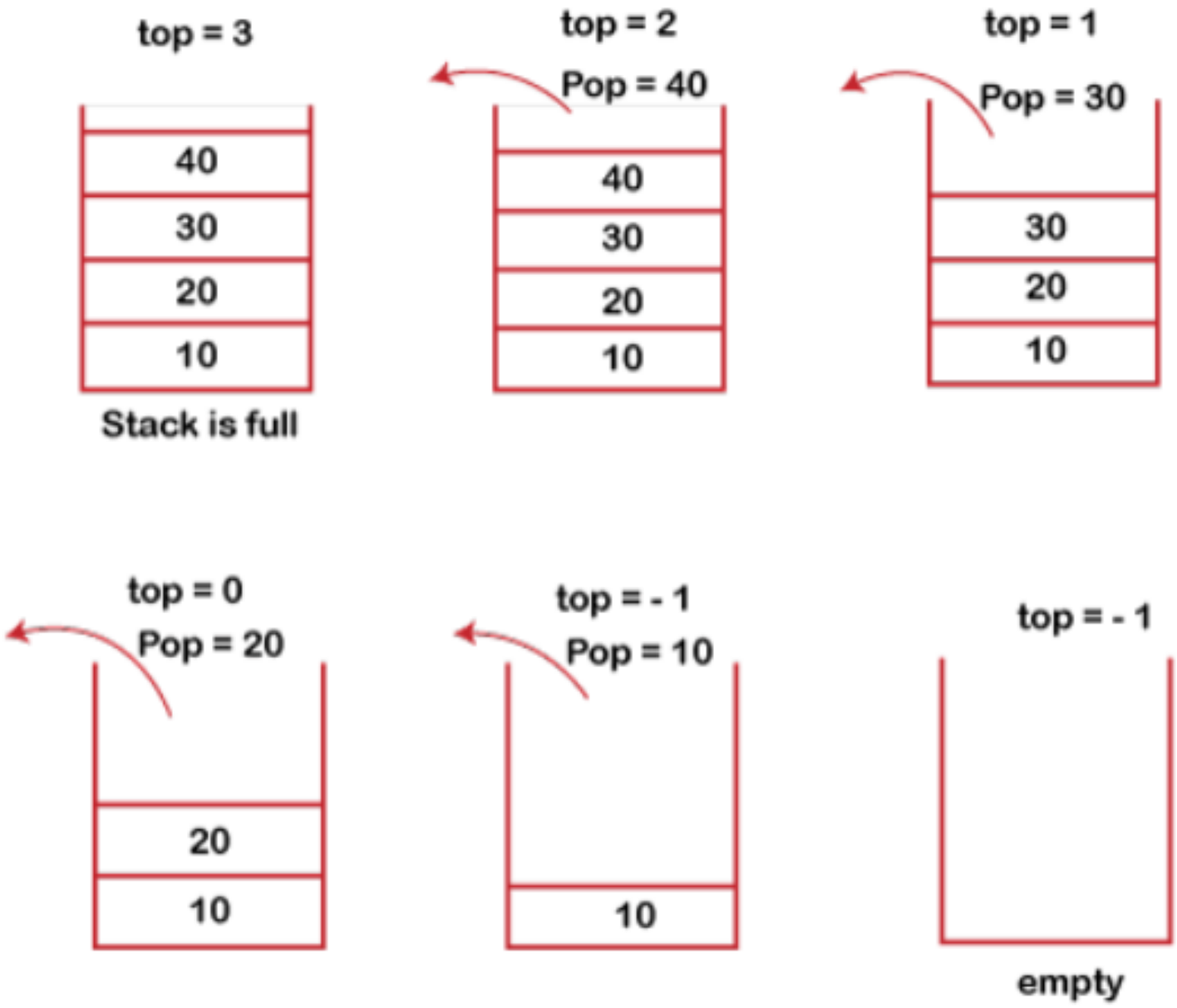
- But in linked-list implementation, pop() actually removes data element and deallocates memory space.

# Stack Operations: pop()

- Push operation involves the following steps:

1. Checks if the stack is empty or not
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

# Stack Operations: pop()



# Stack Operations: pop()

**Algorithm:**

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

```
int pop(int data) {
   if(!isempty())
{
      data = stack[top];
      top = top - 1;
      return data;
   }
else {
cout<<"\nStack is empty\n";
   }
}
```

# Stack Operations: isfull(), isempty() & peek()

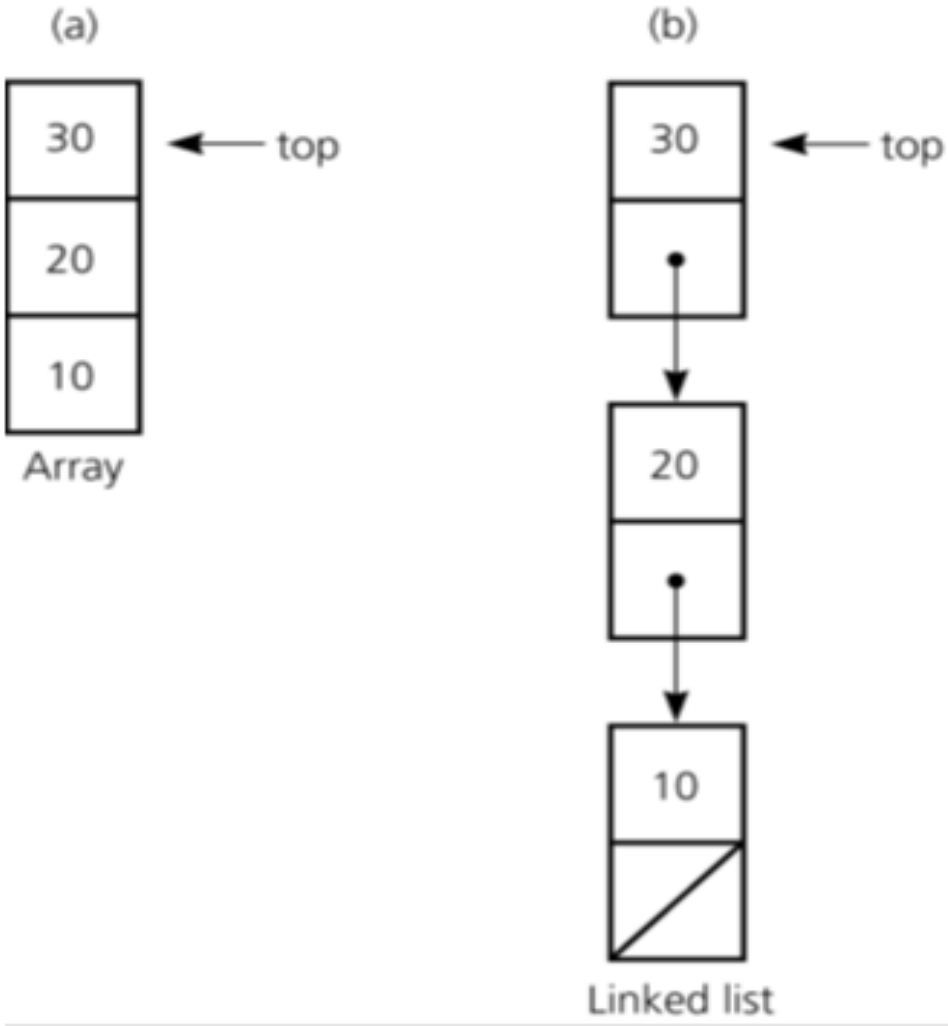
```
bool isfull() {
   if(top == MAX_SIZE-1)
      return true;
   else
      return false;
}
```

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

```
int peek()
{
   if(!isempty())
         return stack[top];
else
 cout<<"\nStack is empty\n";
}
```

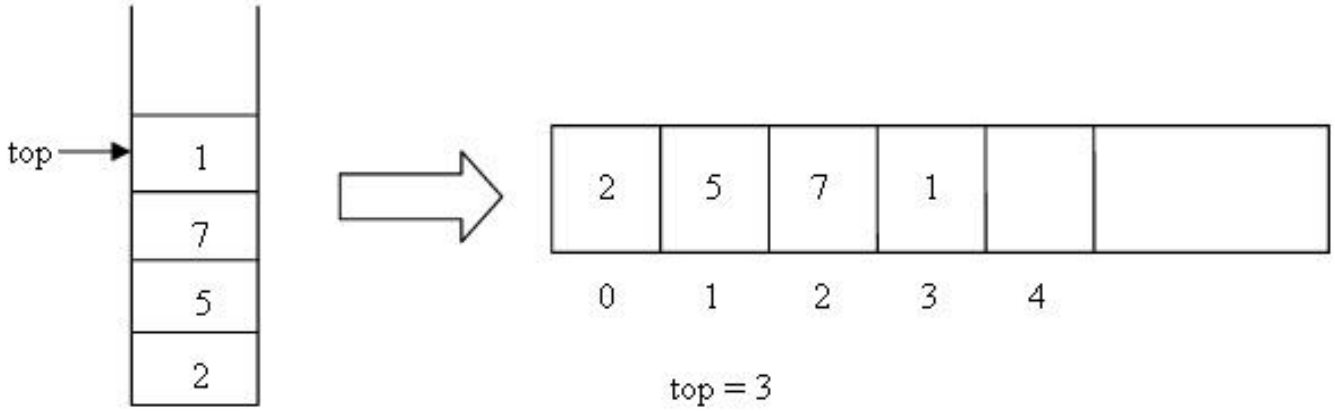
# Stack implementation

- A Stack usually implemented using array or linked list.



# Stack implementation: Array

- Stacks can be represented in memory using arrays.
- In array implementation, the stack is formed by using the array.
- All the operations regarding the stack are performed using arrays.




- Top is pointing to index number 3, which means stack has four items.

# Stack implementation: Array

## Push operation

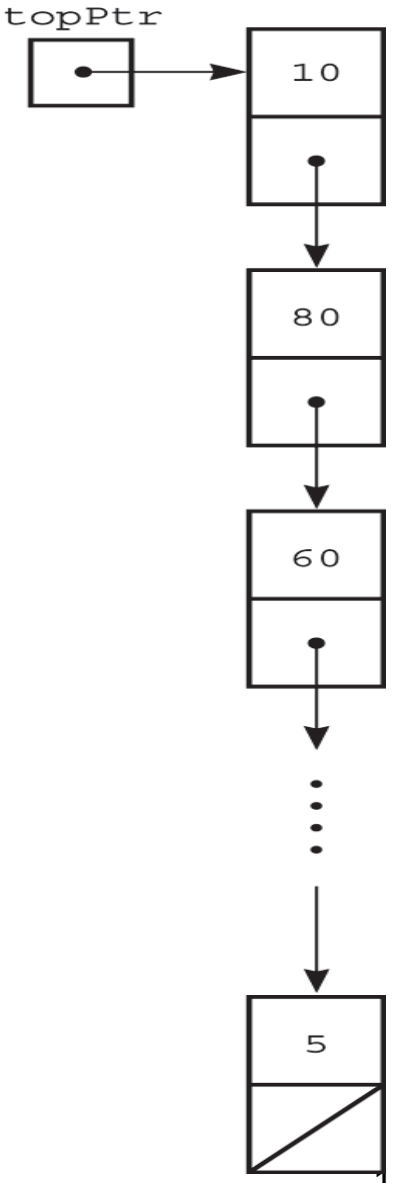
```
void push(int newdata) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = newdata;
   }
else {
cout<<"Stack is full.\n");
   }
}
```

## Pop operation

```
int pop(int data) {
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }
else {
cout<<"\nStack is empty\n";
   }
}
```

# Stack implementation: Linked list

- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Top is a reference to the head of a linked list of items.



# Stack implementation: Linked list

## Push Operation

```
bool isEmpty(){
    return (topPtr == NULL);
}

void push(dataType newItem)
{
    // create a new node
    StackNode *newPtr = new StackNode;

    // set data portion  of new node
    newPtr->item = newItem;

    // insert the new node
    newPtr->next = topPtr;
    topPtr = newPtr;
}
```

# Stack implementation: Linked list

## Pop Operation

```
dataType pop()
{
    if (isEmpty())
       cout<<"Stack is empty…!";

   // not empty; retrieve and delete top
    else{
        stackTOp = topPtr->item;
        StackNode *temp = topPtr;
        topPtr = topPtr->next;

        // return deleted node to system
        temp->next = NULL;  // safeguard
        delete temp;
        return stackTOp;
    }
}
```

# Comparing Implementations

- An array-based implementation:
  - Prevents the push operation from adding an item to the stack if the stack's size limit has been reached.

- A pointer-based (Linked list) implementation.
  - Does not put a limit on the size of the stack.

# Applications of Stack

- **String reversal:**
  - Stack is used for reversing a string.
  - Put all the letters in a stack and pop them out.
  - Because of the LIFO order of stack, you will get the letters in reverse order.

- **In browsers :**
  - The back button in a browser saves all the URLs you have visited previously in a stack.
  - Each time you visit a new page, it is added on top of the stack.
  - When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

# Applications of Stack

- **Balancing of symbols:**
  - Stack is used for balancing a symbol.
    - For example, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
- **UNDO/REDO:**
  - Stack can also be used for performing UNDO/REDO operations.
- **Expression conversion:**
  - Stack can also be used for expression conversion.
  - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.
- ……

# Applications of Stack: Expression conversion

- 4+5*5
  - Simple calculation → 45
  - Scientific calculation → 29

Mathematical Expression

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

C++ Expression

```
x = (-b+(b^2-4*a*c)^0.5)/(2*a)
```

- Naturally, we compute parenthesis first, precedence.
- Develop an algorithm to do the same?
  - Possible but complex!

# Solution: Re-expressing the Expression

- Restructure arithmetic expressions so that the order of each calculation is embedded in the expression itself.
- Types of Expressions:
  - Infix notation (A + B)
    - Used in Mathematics, suitable for humans
    - Rules: BODMAS…
  - Prefix (Polish)notation (+AB)
    - C++ function: add(A, B)
  - Postfix (Reverse-Polish) notation (AB+)
    - suitable for computers
    - Arithmetic and Logical Unit (ALU) designed using this notation.

- The following table briefly tries to show the difference in all three notations .

| Sr.No. | Infix Notation | Prefix Notation | Postfix Notation |
|---|---|---|---|
| 1 | a + b | + a b | a b + |
| 2 | (a + b) * c | * + a b c | a b + c * |
| 3 | a * (b + c) | * a + b c | a b c + * |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) * (c + d) | * + a b + c d | a b + c d + * |
| 6 | ((a + b) * c) - d | - * + a b c d | a b + c * d - |

# Advantages of Using Postfix Notation

- No need to apply operator precedence and other rules.
- Parentheses are unnecessary.
- Easy for the computer (compiler) to evaluate an arithmetic expression.
- The idea is taken from *post-order traversal* of an expression *tree.*

# Postfix Expression Evaluating Algorithm

- An algorithm exists to evaluate postfix expressions using a stack.
- The single value on the stack is the desired result.
- Binary operators: +, -, *, /, etc.,
- Unary operators: unary minus, square root, sin, cos, exp, etc.,

# Postfix Evaluation

## Psuedocode:

Operand: push

Operator: pop 2 operands for binary operator and 1 operand for unary operator, do the math, push result back onto stack

1 2 3 + *

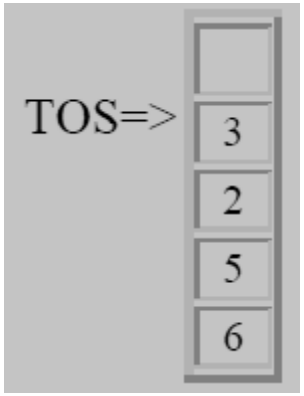
| Postfix | Stack( bot -> top ) |
|---|---|
| a) 1 2 3 + * | |
| b) 2 3 + * | 1 |
| c) 3 + * | 1 2 |
| d) + * | 1 2 3 |
| e) * | 1 5 // 5 from 2 + 3 |
| f) | 5 // 5 from 1 * 5 |

# Postfix Evaluation Algorithm

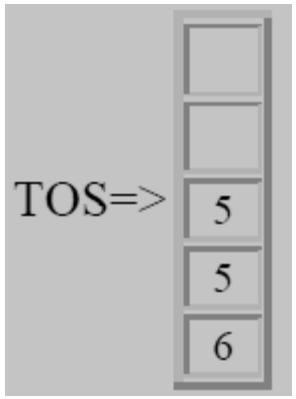
```
initialize stack to empty;
while (not end of postfix expression) {
        get next postfix item;
        if (item is value)
              push it onto the stack;
        else if (item is binary operator) {
              pop the stack to x;
              pop the stack to y;
              perform y operator x;
              push the results onto the stack;
          }
        else if (item is unary operator) {
              pop the stack to x;
              perform operator(x);
              push the results onto the stack
          }
  }
```

# Example: 6 5 2 3 + 8 * + 3 + *

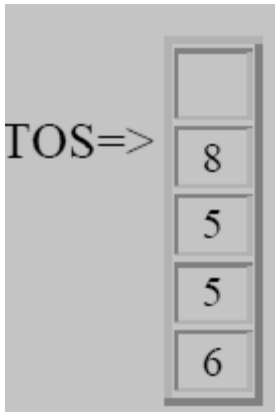
- Push items 6 through 3 **6 5 2 3** + 8 * + 3 + *



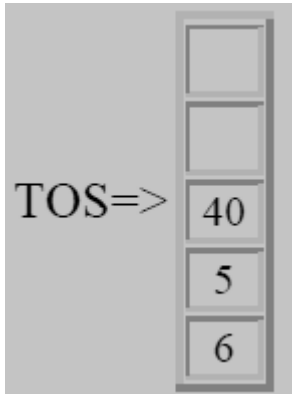

- Next + is read (binary operator), pop 3 & 2, push their sum 5 onto the stack:




- Next 8 is pushed




- Next item is * :

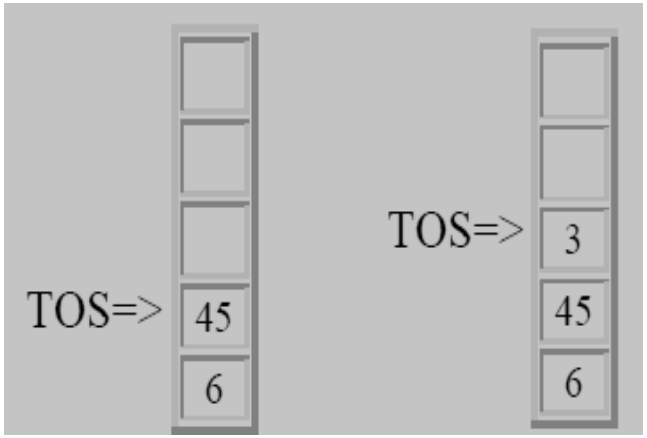



(8 & 5 popped, 40 pushed)

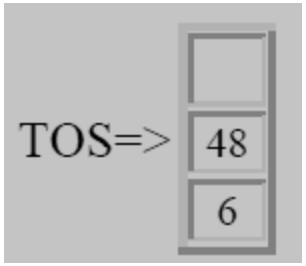
6 5 2 3 + 8 * + 3 + *

- Next the operator + followed by 3:

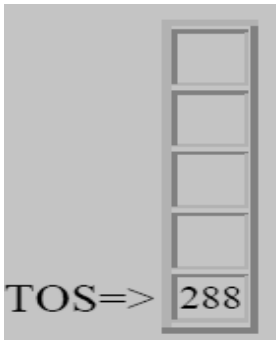
  (40 & 5 popped, 45 pushed, 3 pushed)




- Next is + , pop 3 & 45 and push 45+3=48




- Next is *, pop 48 & 6 ,and push 6*48=288



# Converting Infix to Postfix

- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.
- In high level languages, infix notation cannot be used to evaluate expressions.
- We must analyze the expression to determine the order in which we evaluate it.
- A common technique is to convert an infix notation into postfix notation, then evaluating it.

# Converting Infix to Postfix

- Convert A + B * C to postfix form:

| | |
|---|---|
| A + B * C | Infix Form |
| A + (B * C) | Parenthesized expression |
| A + (B C *) | Convert the multiplication |
| A (B C *) + | Convert the addition |
| A B C * + | Postfix form |

**Rules:**

1. Parenthesize from left to right, higher precedence operators parenthesized first.
2. The sub-expression (part of expression), which has been converted into postfix, is treated as single operand.
3. Once the expression is converted to postfix form, remove the parenthesis.

# Converting Infix to Postfix

Example (infix Expression): 3 + 2 * 4

| Infix | operator stack | postfix |
|---|---|---|
| 3 + 2 * 4 | empty | empty |
| + 2 * 4 | empty | 3 |
| 2 * 4 | + | 3 |
| * 4 | + | 32 |
| 4 | +* | 32 |
| | +* | 324 |
| | + | 324* |
| | empty | 324*+ |

# Algorithm to Convert Infix to Postfix

## Steps

1. Operands immediately go directly to output
2. Operators are pushed into the stack (including parenthesis)
   - Check to see if stack top operator is less than current operator
     - If the top operator is less than, push the current operator onto stack
   - If the top operator is greater than the current, pop top operator and append on postfix notation, push current operator onto stack.
   - If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

## Precedence Priority of operators:

- Priority 4: '(' - only popped if a matching ')' is found.
- Priority 3: All unary operators (-, sin, cosin,….)
- Priority 2: / *
- Priority 1: + -

# Algorithm to Convert Infix to Postfix

**Example 1:** A + B * C - D / E

| | Infix | Stack(bottom->top) | Postfix |
|---|---|---|---|
| | A + B * C - D / E | empty | empty |
| a) | + B * C - D / E | empty | A |
| b) | B * C - D / E | + | A |
| c) | * C - D / E | + | A B |
| d) | C - D / E | + * | A B |
| e) | - D / E | + * | A B C |
| f) | D / E | + - | A B C * + |
| g) | / E | - | A B C * + D |
| h) | E | - / | A B C * + D |
| i) | | - / | A B C * + D E |
| j) | | empty | A B C * + D E / - |

# Thank You

# Question?