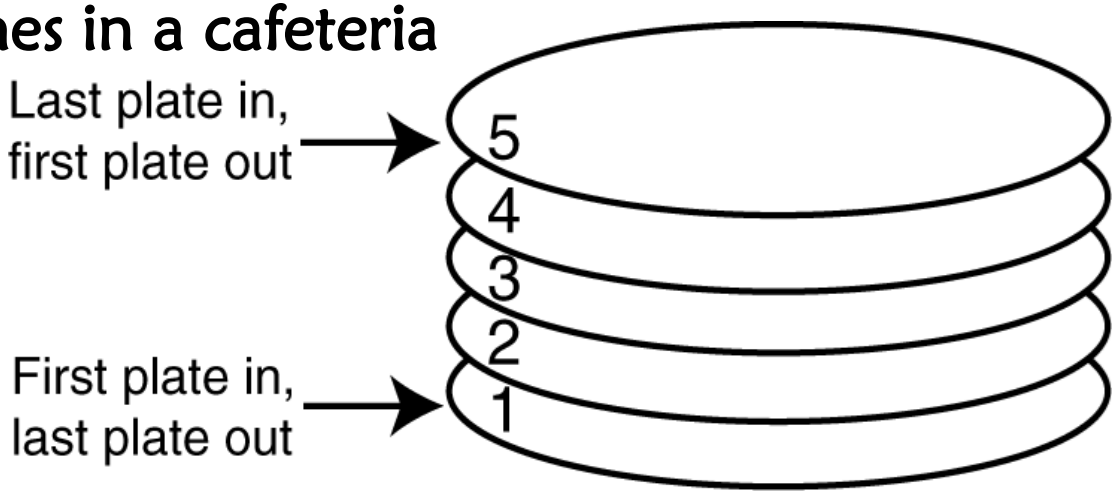
Chapter Four

Stacks & Its Applications

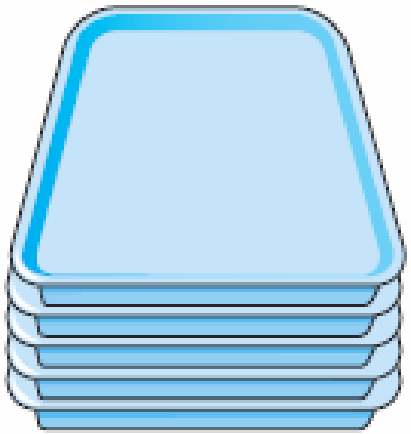
Introduction

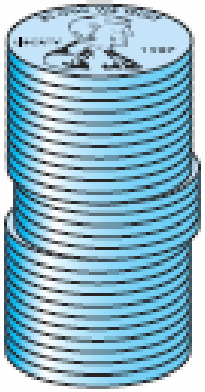
- **Stacks** are linear lists.
- All deletions and insertions occur at one end of the stack known as the **TOP**.
- Data going into the stack first, leaves out last.
- This is known as **LIFO** data structures(the last element put into the list will be the first element we take out of the list).
  - Last-In, First-Out ("LIFO")
- **Analogy**
  - **A stack of dishes in a cafeteria**



More Analogies

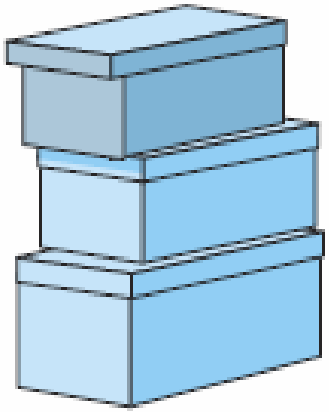
A stack of cafeteria trays

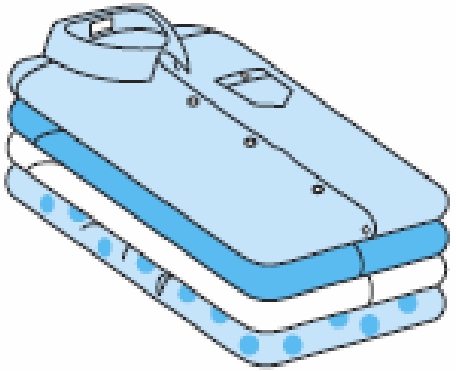




A stack of pennies

A stack of shoe boxes

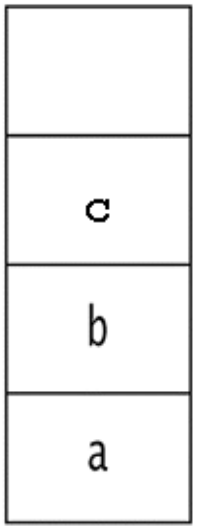




A stack of neatly folded shirts

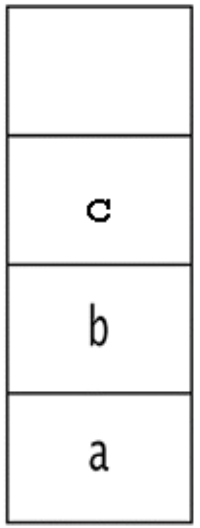
Abstract data type: Stack

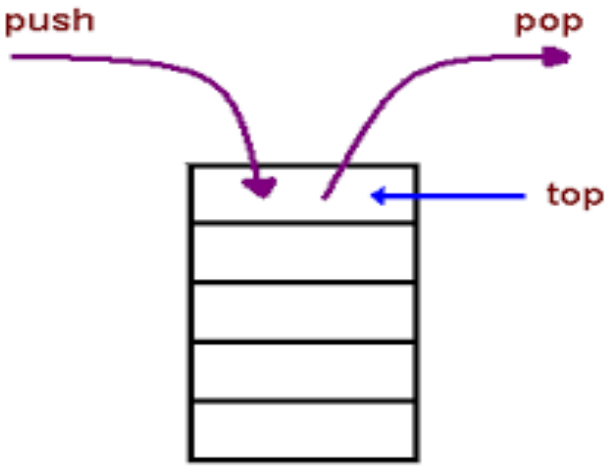
- There are many times where it is useful to use a list in a way where we always add to the end, and also always remove from the end.
- **Stack**: a more restricted List with the following constraints:
  - Elements are stored by order of insertion from "bottom" to "top".
  - Items are added to the top.
  - Only the last element added onto the stack (the top element) can be accessed or removed.



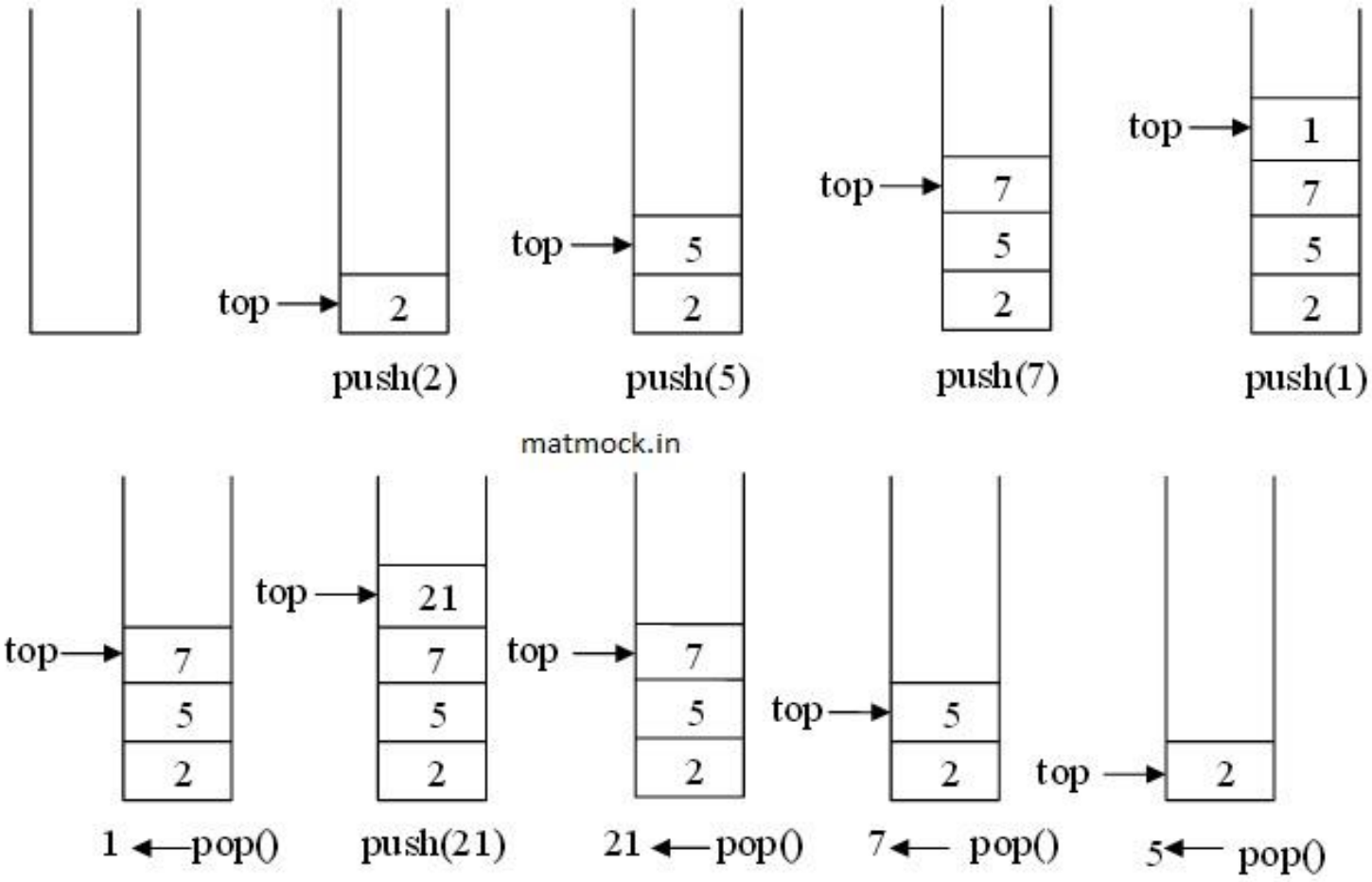
Abstract data type: Stack…

- Goal: every operation on a stack should be O(1).
  - Stacks are straightforward to implement in several different ways, and are very useful.
- **Operations on a stack**
  - **push**: add an element to the top.
  - **pop**: remove and return the element at the top.





Push & Pop Operations



peek: return (but not remove) top element; pop or peek on an empty stack causes an exception.

Other Stack Operations

- initialize():
  - Initialize the stack to be empty.
- isFull():
  - Determine if stack is full or not. A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- isEmpty():
  - Determine whether stack is empty or not. A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.
- display():
  - If the stack is not empty then retrieve all the elements starting at its top.

Stack … (continued)

Stack features:

- **ORDERING**: maintains order elements were added (new elements are added to the end by default).
- **OPERATIONS**:
  - Add element to end of stack **('push')**
  - Remove element from end of stack **('pop')**
  - Examine element at end of stack **('peek')**
  - Clear all elements from stack **('makeEmpty')**
  - Check whether stack is empty or not **('isEmpty')**
  - Find size of stack **('getSize')**
  - All of these operations are efficient! O(1).

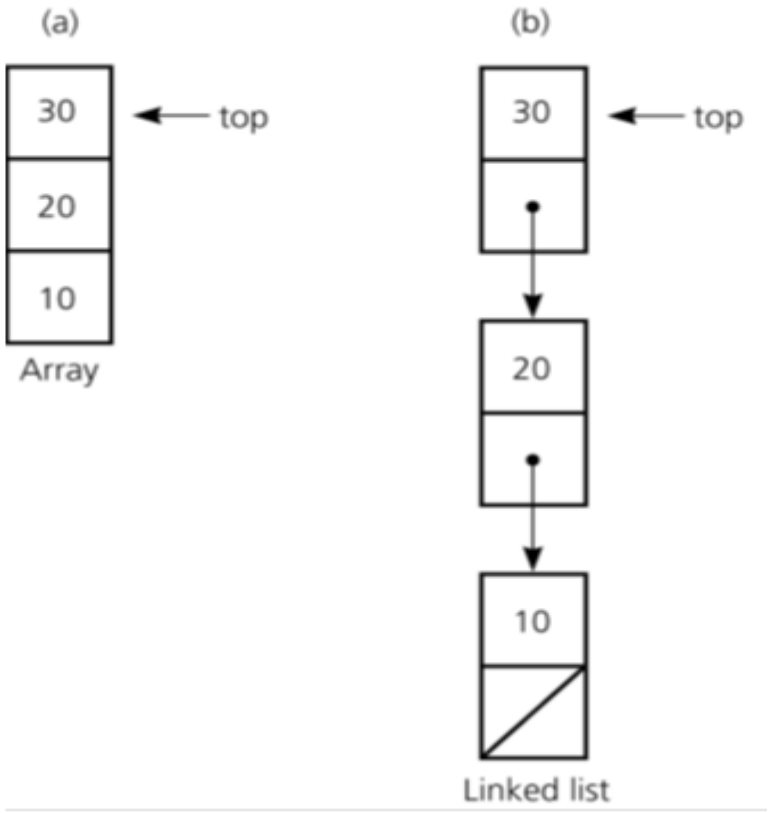
Stack … (continued)

Stacks in computer science

- The lowly stack is one of the most important data structures in all of computer science
  - Function/method calls are placed onto a stack.
  - Compilers use stacks to evaluate expressions.
  - Stacks are great for reversing things, matching up related pairs of things, and backtracking algorithms.
  - Stack programming problems:
    - ✓ Reverse letters in a string, reverse words in a line, or reverse a list of numbers.
    - ✓ Find out whether a string is a palindrome.
    - ✓ Examine a file to see if its braces { } and other operators match.

Stack … (continued)

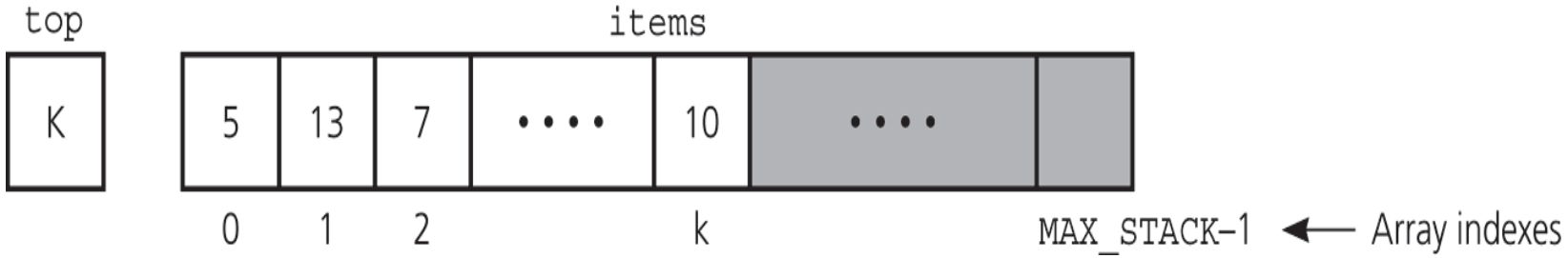
- Stacks structures are usually implemented using arrays or linked lists.



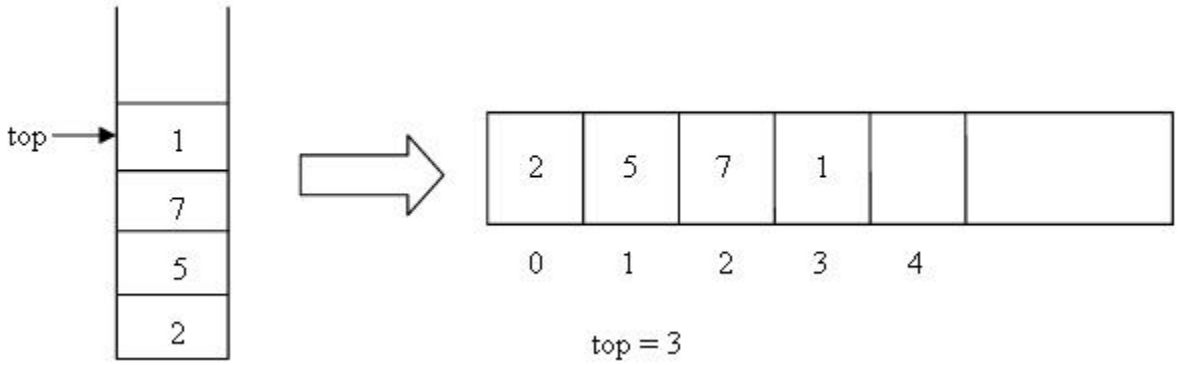
- We will be examining common Stack Applications.

Array Based Stack Implementation

- Stacks can be represented in memory using arrays.
- We can declare an array named as STACK.
- We also use TOP variable to represent top most element of stack.
- Finally a variable MAX_STACK, which gives maximum number of elements that can be stored in stack.



Array-based Stacks



- **Top** is pointing to **index** number 03, which means stack has four items.
- Sometimes when a new data is to be inserted into stack but there is no available space, this situation is called as **stack overflow.**
- Sometimes when a data is to be removed/deleted from stack but there is no available data, this situation is called as **stack underflow.**

Implementation of Push

```
bool isEmpty(){
    return (top < 0);
}

void push(data_type newItem){

    // if stack has no more room for
    // another item
    if (top >= MAX_STACK-1)
        cout<<"Stack Overflow!";

    else{
        ++top;
        items[top] = newItem;
        cout<<"Item inserted at the top";
    }
}
```

Implementation of Pop

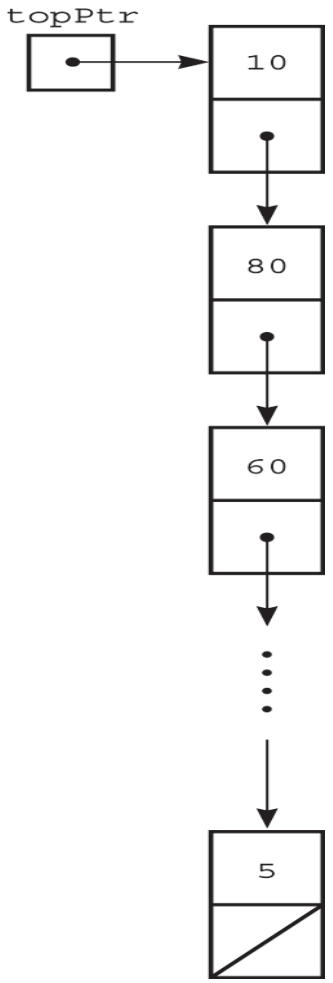
```
data_type pop(){

    if (isEmpty())
        cout<<"Stack underflow";

// stack is not empty retrieve top
    else {
        stackTop = items[top];
        --top;
        return stackTop;
    }
}
```

Linked list Implementation

- A pointer-based implementation.
  - Required when the stack needs to grow and shrink dynamically.
- Top is a reference to the head of a linked list of items.



Linked list Implementation: Push

```
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;
}
```

Linked list Implementation: Pop

```
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 implementation.
  - Does not put a limit on the size of the stack.

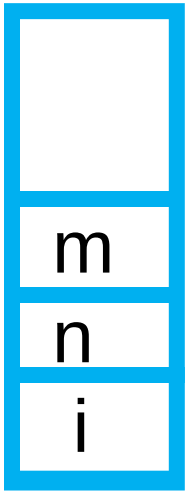
Applications of Stack

- Variable declaration
- Function Calling
- Reversing Data
- Converting Decimal to Binary
- Evaluation of Algebraic expression
- Converting Infix to Postfix

……

Variable declaration

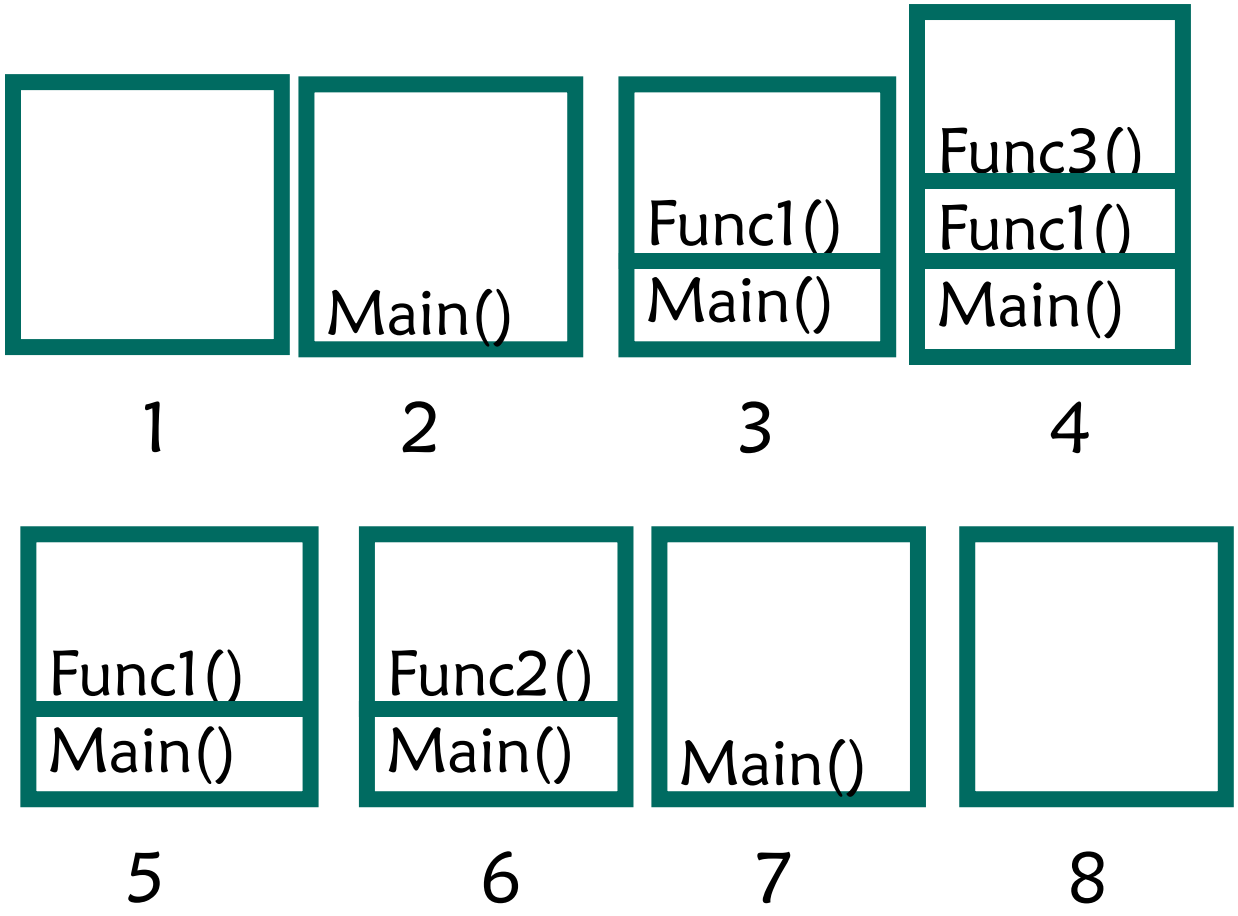
```
void main()
{
 int i, n;
 for(i=0;i<=10;i++)
    {
     int m;
     cin>>n;
     cin>>m;
     cout<<m*n;
    }
  cout<<n;
 }
```



Used to hold variables when declared

Function Calling

```
void main(){
   func1();
   func2();}
   void func1(){
   cout<<"Hello";
   func3();
   }
   void func2(){
   cout<<"Hi";
   }
   void func3(){
   cout<<"Hey";
   }
   }
```



Note : The one that is found on the top of the stack is currently executed. When empty stack remain, the program will halt.

Reversing Data

- We can use stacks to reverse data. (example: files, strings).
- Very useful for finding palindromes.
- Consider the following pseudo code:

1) read (data)
2) loop (data not EOF and stack not full)
    1) push (data)
    2) read (data)
3) Loop (while stack notEmpty)
    1) pop (data)
    2) print (data)

Converting Decimal to Binary

- Consider the following pseudo code:

  1) Read (number)
  2) Loop (number > 0)
     1) digit = number modulo 2
     2) print (digit)
     3) number = number / 2

- The problem with this code is that it will print the binary number backwards.

Example: 19 becomes 11001000 instead of 00010011

- To remedy this problem, instead of printing the digit right away, we can push it onto the stack.
- Then after the number is done being converted, we pop the digit out of the stack and print it.

Evaluation of Algebraic Expressions

- 4+5*5
  - Simple calc. → 45
  - Scientific calc. → 29

Mathematical Expression C++ Expression

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

```
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 notation **(+ A B)**
    - » C++ function: add(A, B)
  - Postfix notation **(A B +)**
    - » suitable for computers
    - » Arithmetic and Logical Unit (ALU) designed using this notation.

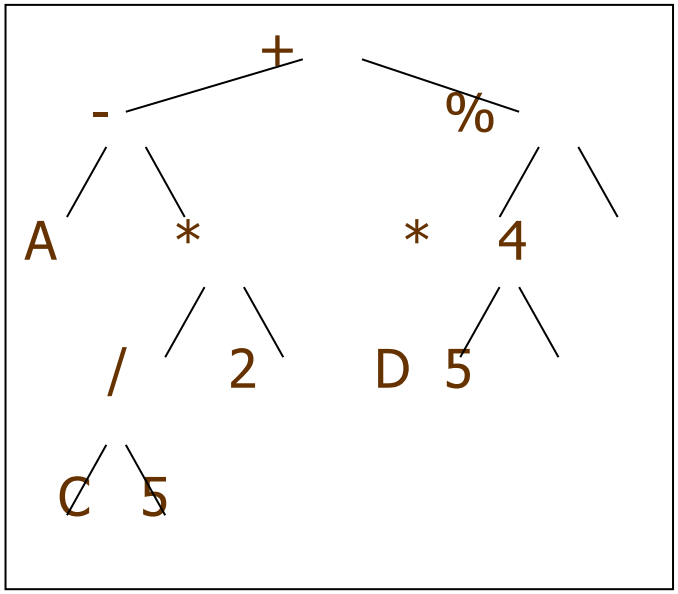
Advantages of Using Postfix Notation

- 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*** (you will study it in later classes).
- Consider the following tree: Next Slide

Post Order Traversal, Binary Tree

Infix: A – C / 5 * 2 + D * 5 % 4

Visiting principle: LRP (Left-Right-Parent) (Recursive)



Postfix: A C 5 / 2 * - D 5 * 4 % +

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

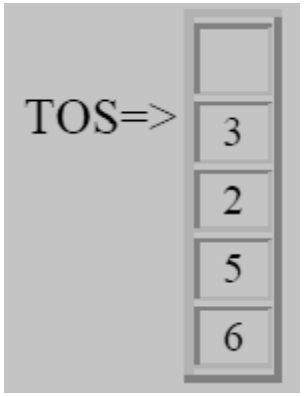
1 2 3 + *

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

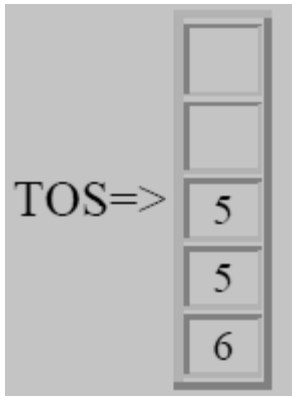
```
initialize stack to empty;
while (not end of postfix expression) {
        get next postfix item;
        if (item is value)
              push it onto the stack;
        else if (item is binary operator) {
              pop the stack to x;
              pop the stack to y;
              perform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

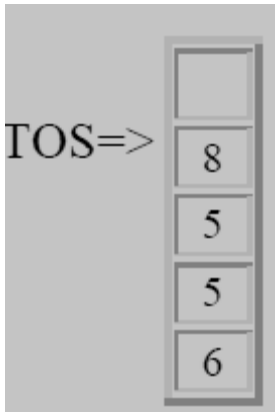
- Push items 6 through 3 **6 5 2 3** + 8 * + 3 + *

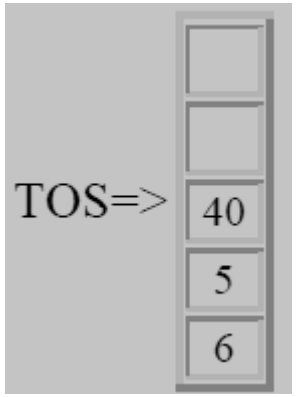


- Next + is read (binary operator), pop 3 & 2, push their sum 5 onto the stack:



- Next 8 is pushed





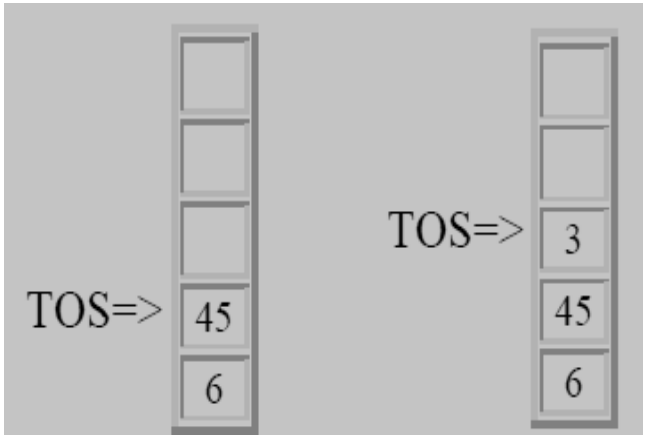
- Next item is * :

(8, 5 popped, 40 pushed)

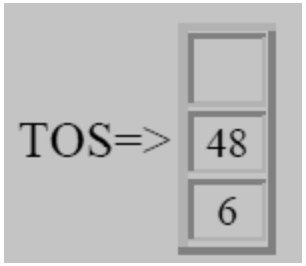
6 5 2 3 + 8 * + 3 + *

- Next the operator + followed by 3:

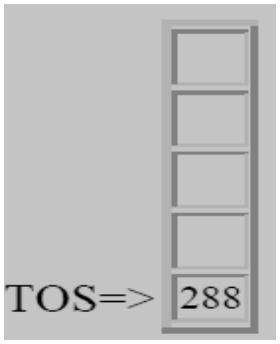
  (40, 5 popped, 45 pushed, 3 pushed)



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



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



Converting Infix to Postfix

- 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 light, 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.

Example: A + [(B + C) + (D + E) * F] / G

A + { [(BC +) + (DE +) * F] / G}

A + { [(BC +) + (DE + F *] / G}

A + { [(BC + (DE + F * +] / G}

A + [BC + DE + F *+ G /]

ABC + DE + F * + G / +

- 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 a infix notation into postfix notation, then evaluating it.

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: + -

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

	Infix	Stack(bottom->top)	Postfix
	A + B * C - D / E	empty	empty
	+ 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 / -

Example 2: A * B - (C + D) + E

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

Example 3: a + b * c + (d * e + f) * g

	Infix >top)	Postfix	Stack(bottom-
	a + b * c + (d * e + f) * g		empty
a)	+ b * c + (d * e + f) * g		empty a
b)	b * c + (d * e + f) * g	+	a
c)	* c + (d * e + f) * g	+	a b
d)	c + (d * e + f) * g	a b	+ *
e)	+ (d * e + f) * g	a b c	+ * a
f)	(d * e + f) * g	a b c *	+
g)	(d * e + f) * g	a b c * +	empty a
h)	(d * e + f) * g	a b c * +	+ a
i)	d * e + f) * g	a b c * +	+ (
j)	* e + f) * g	a b c * + d	+ (
k)	e + f) * g	+ (*	a b c *
	+ d		
l)	+ f) * g	+ (*	a

- Page-visited history in a Web browser
- To Undo and Redo in a text editor:

Pseudocode:

```
Accept the command
if(command==Undo)
   push_RS(pop_US( ))
else if(command==Redo)
   push_US(pop_RS())
 else
   push_US(command)
```