CSE 1201
Data Structure

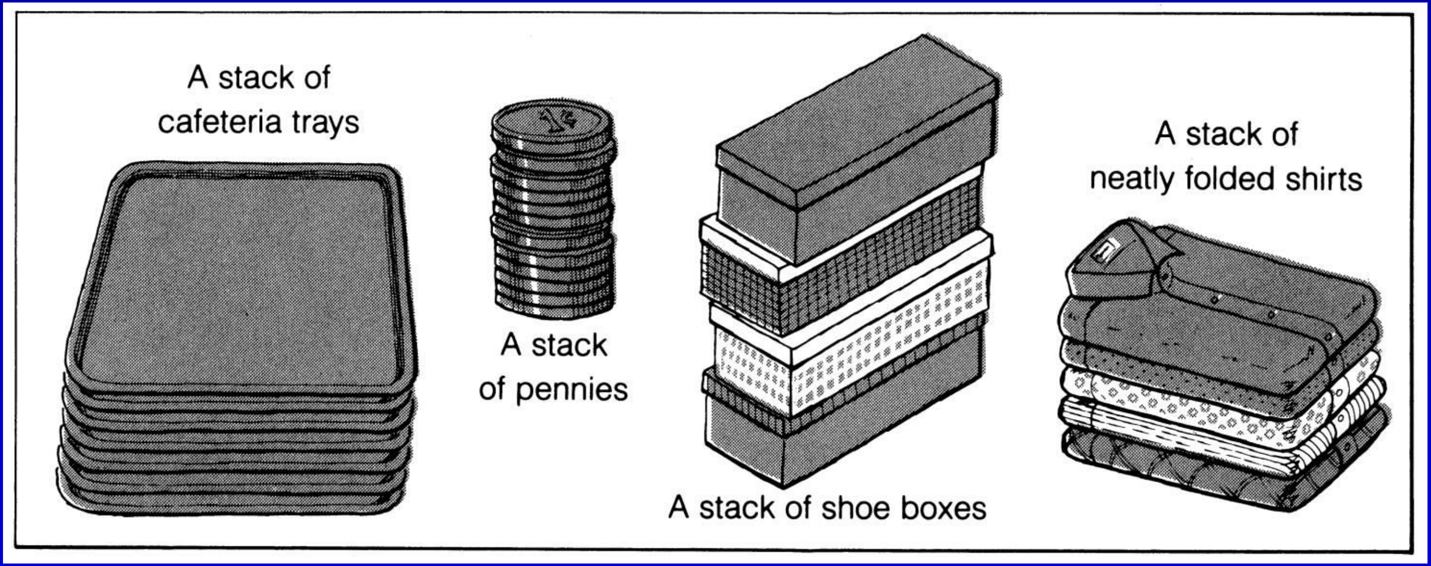
Chapter 4

Stacks

Instructor: Md. Shahid Uz Zaman
Dept. of CSE, RUET, Bangladesh

What is a stack?

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Stack Specification

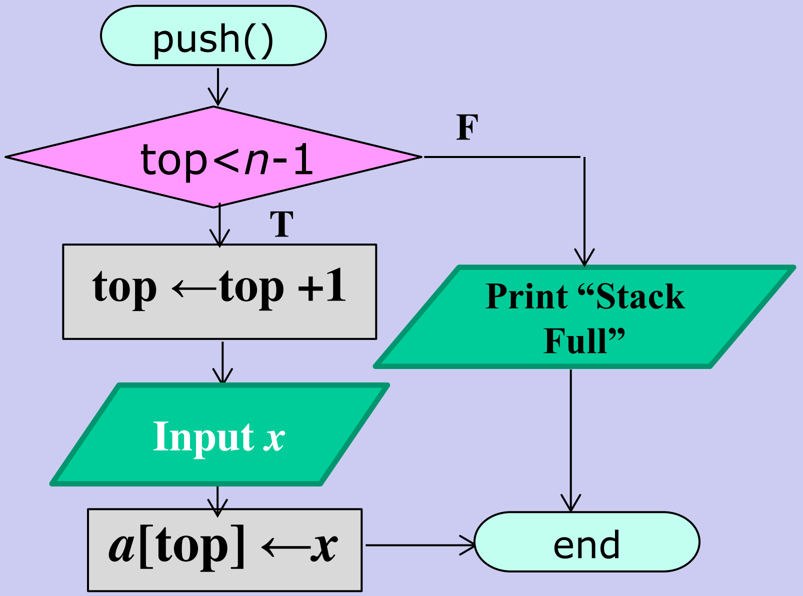
- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the stack
  - *ItemType*: Data type of the items on the stack
- Operations
  - Push (ItemType newItem)
  - Pop ()

Push (ItemType newItem)

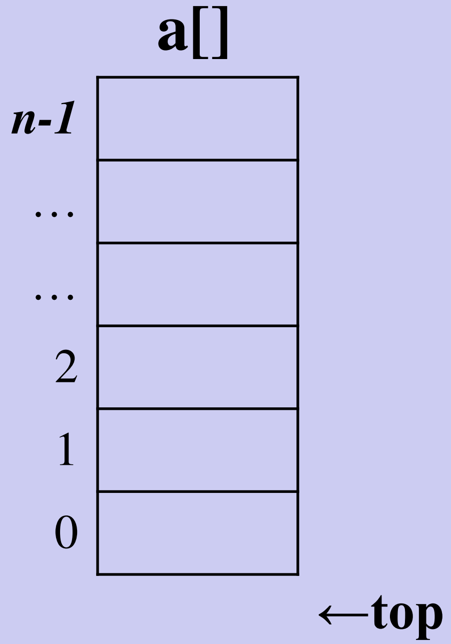
- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Stack: push() function

Topic 1: Write an Algorithm to push a new element in a stack



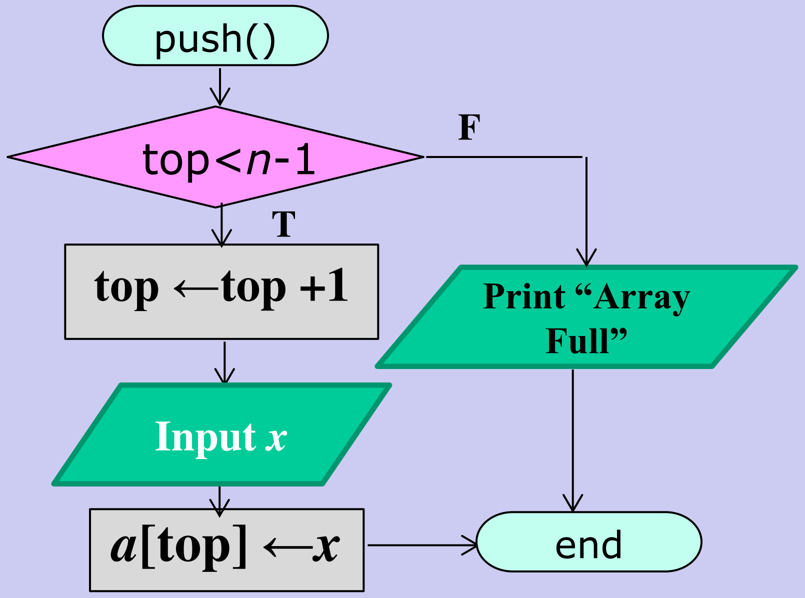
n: size of *a[ ]*
x: input variable
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



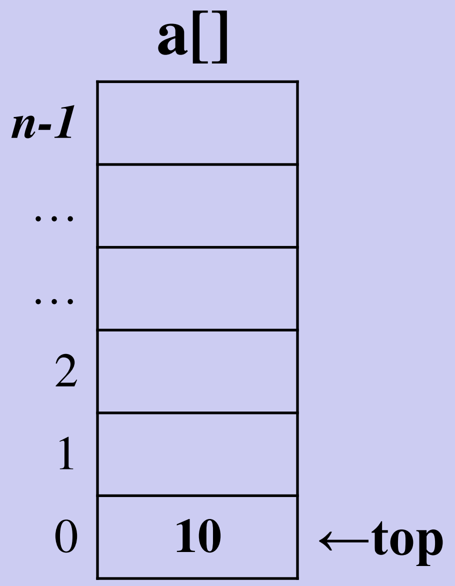
Initially top = -1

Stack: push() function

Topic 1: Write an Algorithm to push a new element in a stack



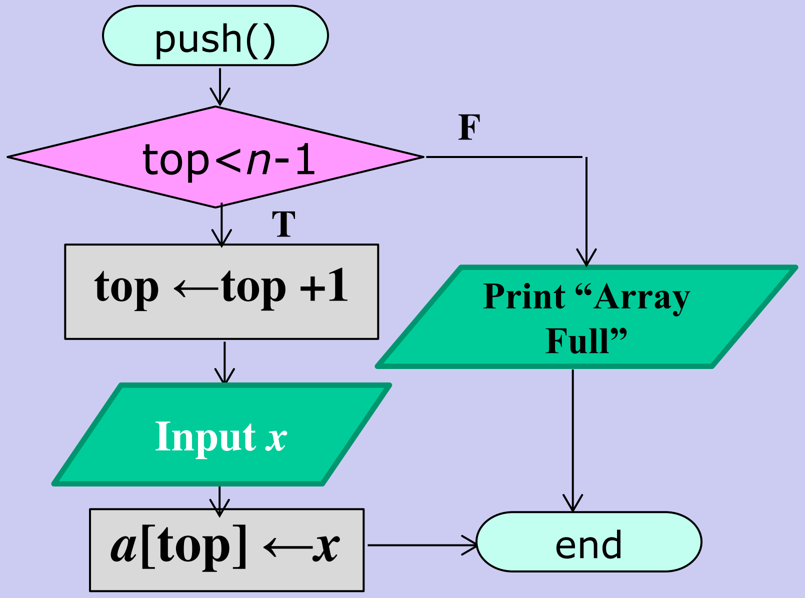
n: size of *a[ ]*
x: input variable
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



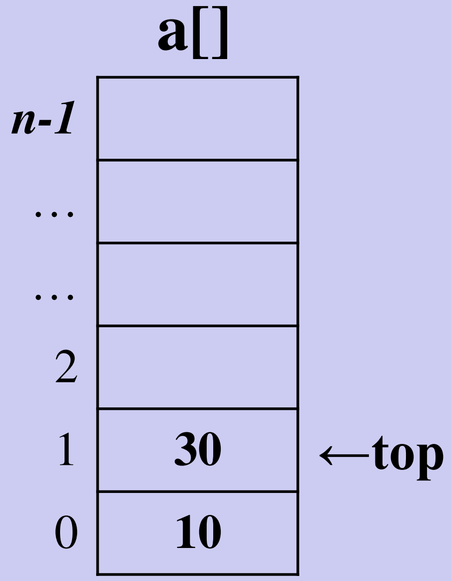
After 1st element (10) push, top = 0

Stack: push() function

Topic 1: Write an Algorithm to push a new element in a stack



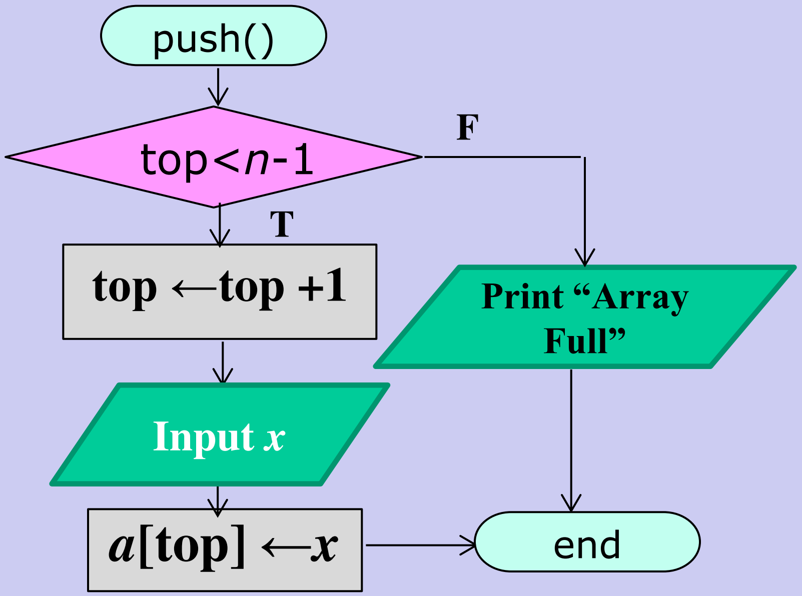
n: size of *a[ ]*
x: input variable
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



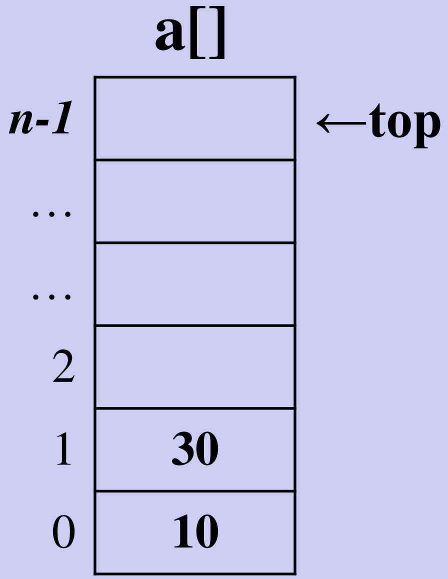
After 2nd element (30) push, top = 1

Stack: push() function

Topic 1: Write an Algorithm to push a new element in a stack



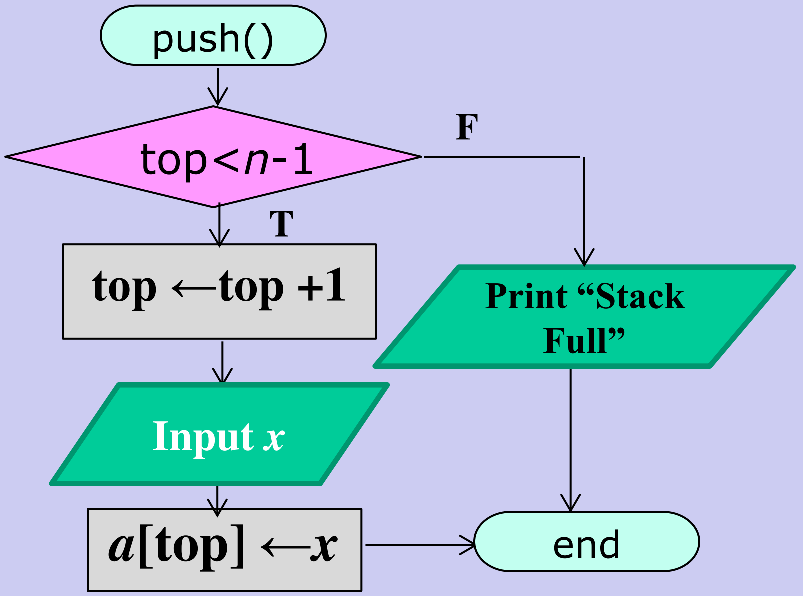
n: size of *a[ ]*
x: input variable
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



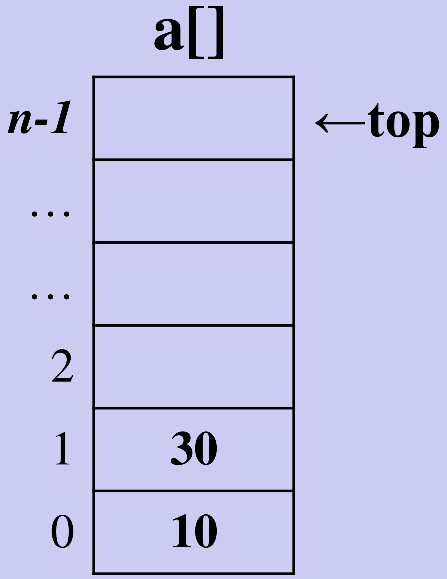
After last element (30) push, top = *n*-1

Stack: push() function

Topic 1: Write an Algorithm to push a new element in a stack



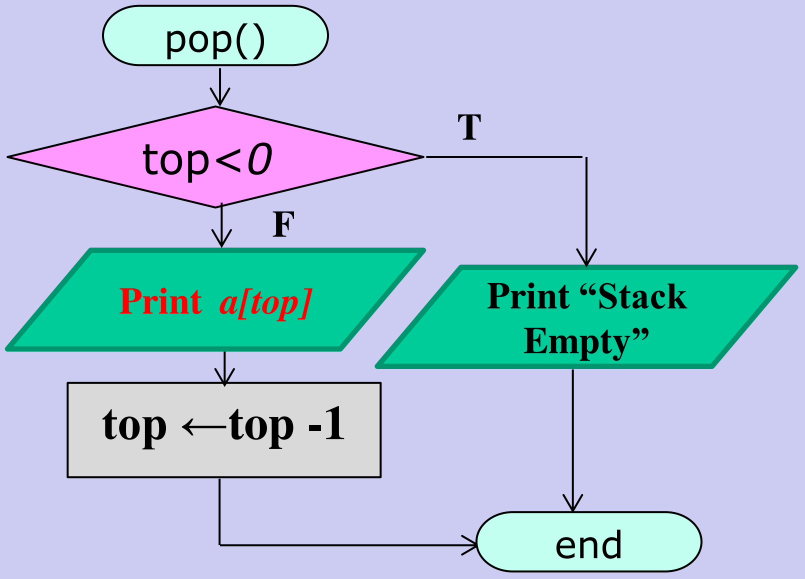
n: size of *a[ ]*
x: input variable
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



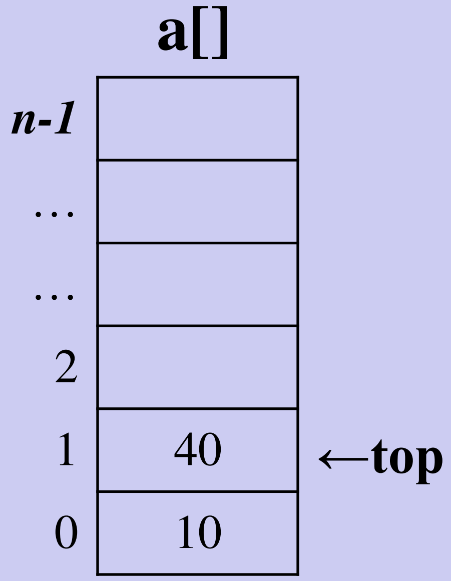
Try to push more get "Stack Full" message

Stack: pop() function

Topic 1: Write an Algorithm to pop element from a stack



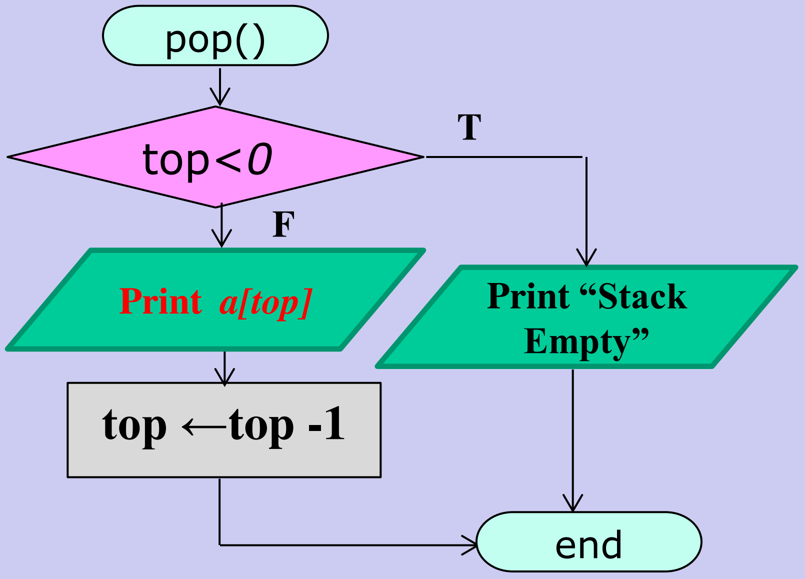
n: size of *a[ ]*
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



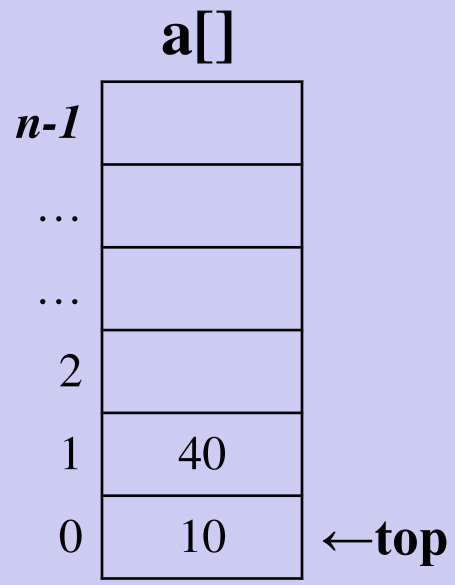
Suppose top = 1

Stack: pop() function

Topic 1: Write an Algorithm to pop element from a stack



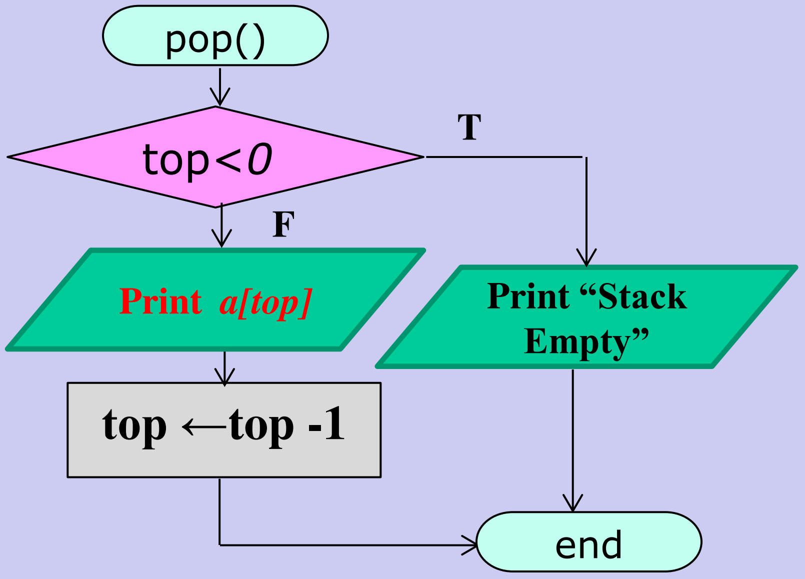
n: size of *a[ ]*
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



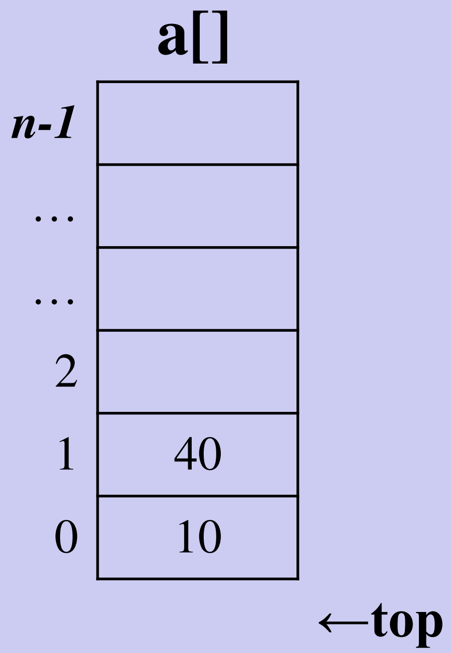
After deletion 2nd element, top=0

Stack: pop() function

Topic 1: Write an Algorithm to pop element from a stack



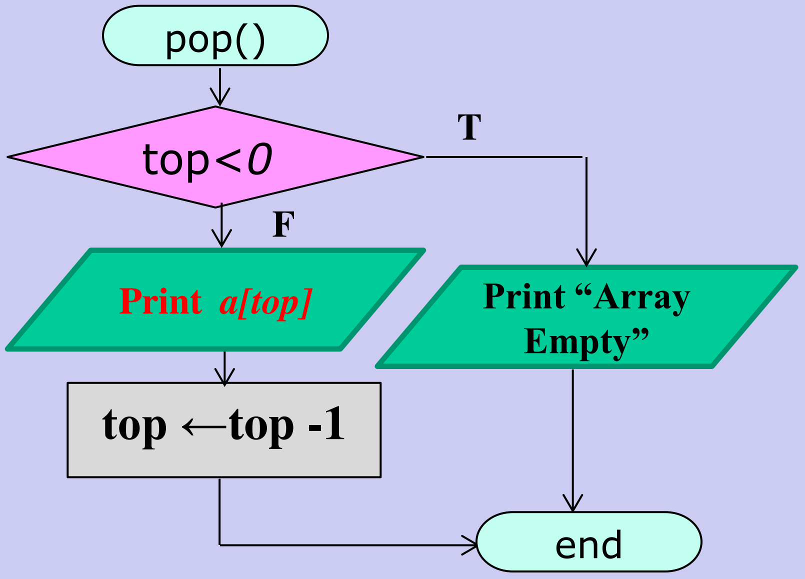
n: size of *a[ ]*
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



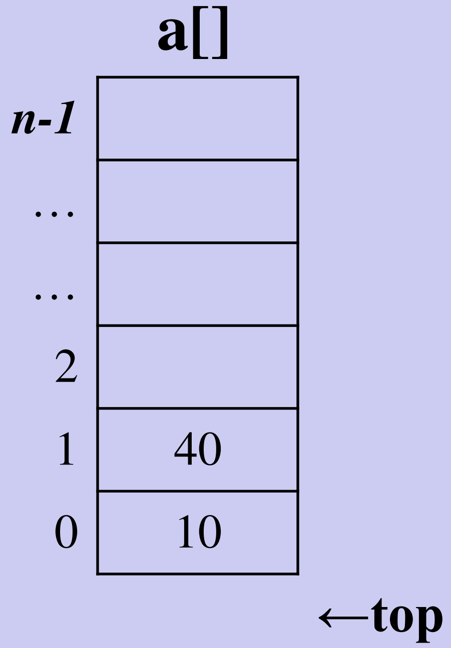
After deletion 1st element, top=-1

Stack: pop() function

Topic 1: Write an Algorithm to pop element from a stack



n: size of *a[ ]*
top: index of last input, declare it global.
Array Elements: *a*[0]…..*a*[*n*-1]



Try to delete more gives message "Stack Empty"

Polish Notation

What is Polish Notation?

Polish notation is a way of expressing arithmetic expression that avoids the use of brackets to define priorities for evaluation of operators.

Types of Polish Notation

1. **Infix** → Operator lies between two operands i.e. 2+3
2. **Prefix**→ Operator lies before two operands i.e. +23
3. **Postfix**→ Operator lies after two operands i.e. 23+

Polish Notation

Infix: 1+2x3+1/2 = ?

((1(+1((22x3)))(3+1)(1/2))/2)24=765

Postfix (Polish notation):

Let A, B, be operands, ♦ an operator.

Instead of A♦B, write AB♦

Polish Notation

Example:

Infix:

((1+2)x(3+1))/2

Postfix:

((1+2)x(3+1))2/

(1+2)(3+1)x2/

(1+2)31+x2/

12+31+x2/

Infix:

((1+2)x(3+1))/2

Prefix:

/((1+2)x(3+1))2

/x (1+2)(3+1)2

/x+12(3+1)2

/x+12+312

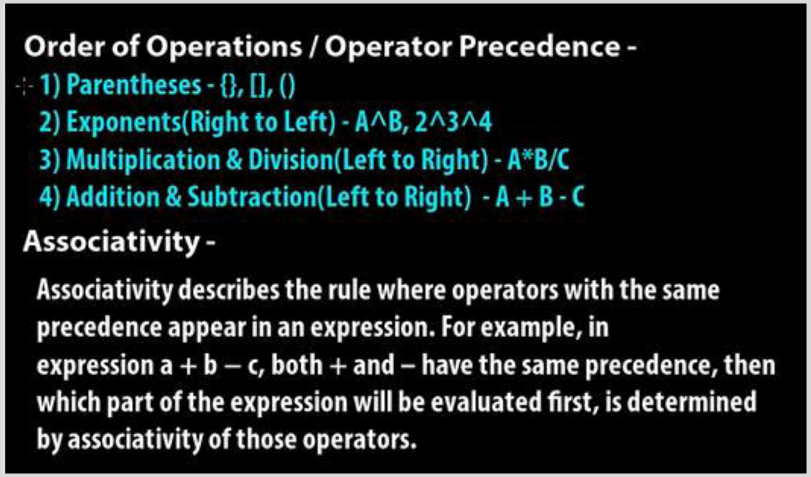
Polish Notation

Advantage of Polish Notation:

No ambiguity!

A-B-C =? (A-B)-C A-(B-C)

AB-C- ABC--



Order of Operations / Operator Precedence -

1) Parentheses - {}, [], ()
2) Exponents(Right to Left) - A^B, 2^3^4
3) Multiplication & Division(Left to Right) - A*B/C
4) Addition & Subtraction(Left to Right) - A + B - C

Associativity -

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b – c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.

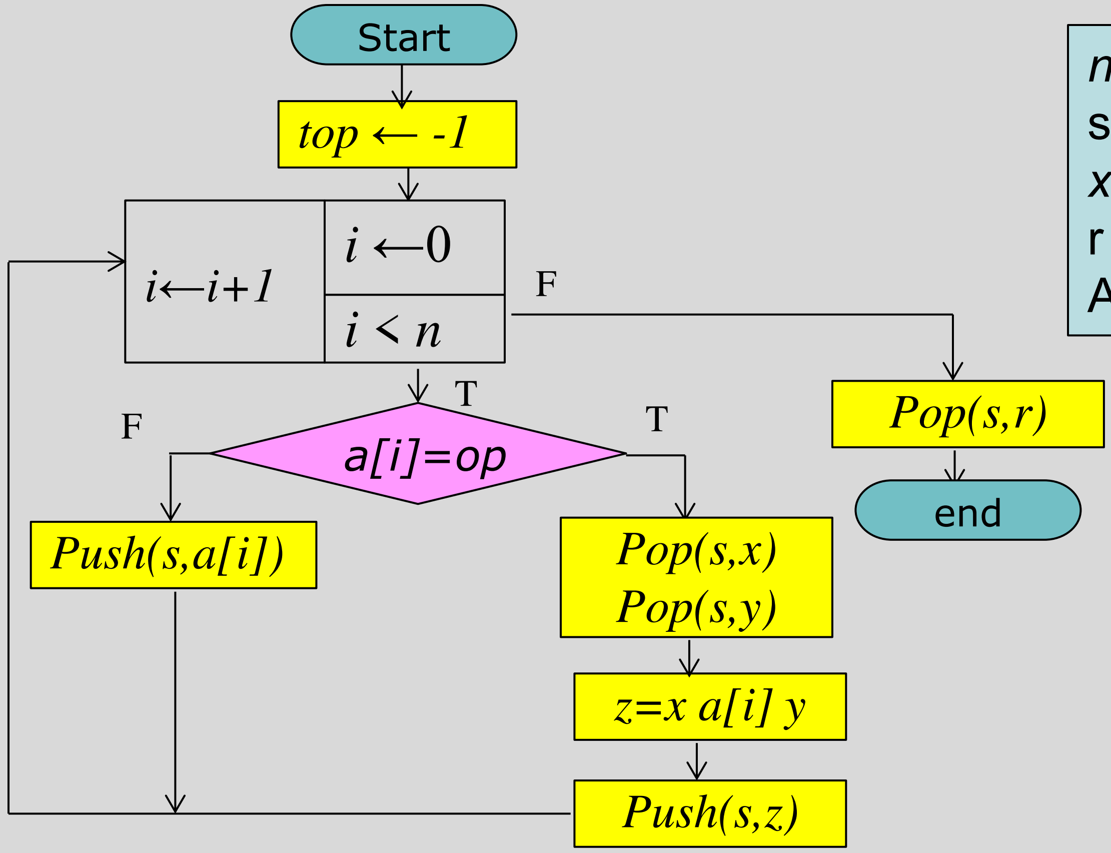
Infix to Postfix Conversion

1. Initialize the Stack.
2. Scan the operator from left to right in the infix expression.
3. If the leftmost character is an operand, set it as the current output to the Postfix string.
4. And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
5. If the scanned operator has higher precedence than the existing **precedence** operator in the Stack or if the Stack is empty, put it on the Stack.
6. If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
7. If the scanned character is a left bracket '(', push it into the Stack.
8. If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
9. Repeat all steps from 2 to 8 until the infix expression is scanned.
10. Print the Stack output.
11. Pop and output all characters, including the operator, from the Stack until it is not empty.

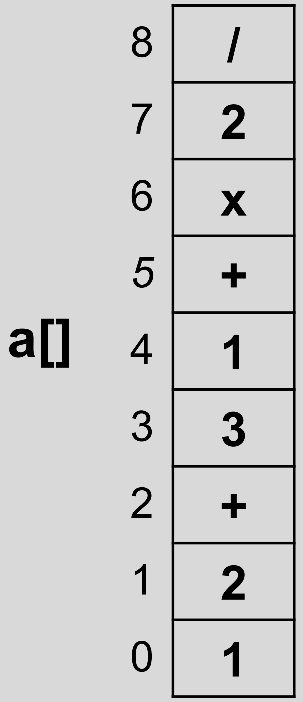
Here, we have infix expression **((A * (B + D)/E) - F * (G + H / K)))** to convert into its equivalent postfix expression:

Label No.	Symbol Scanned	Stack	Expression
1	(	(	
2	(	((	
3	A	((	A
4	*	((*	A
5	(	((*(	A
6	B	((*(	AB
7	+	((*(+	AB
8	D	((*(+	ABD
9	)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12	)	(	ABD+E/*
13	-	(-	ABD+E/*
14	(	(-(	ABD+E/*
15	F	(-(	ABD+E/*F
16	*	(-(*	ABD+E/*F
17	(	(-(*(	ABD+E/*F
18	G	(-(*(	ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH
21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23	)	(-(*	ABD+E/*FGHK/+
24	)	(-	ABD+E/*FGHK/+*
25	)		ABD+E/*FGHK/+*-

Postfix Expression



n: size of *a[]*
s: stack
x, y, z : temp variable
r : store result
Array Elements: *a*[0]…..*a*[*n*-1]

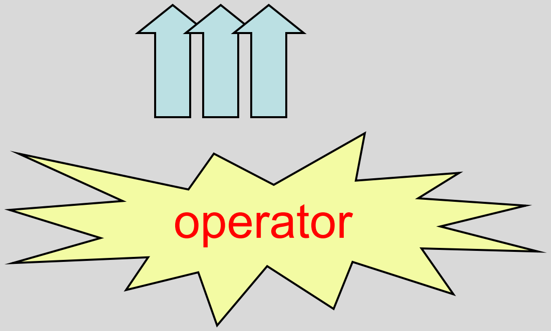


	a[]
8	/
7	2
6	x
5	+
4	1
3	3
2	+
1	2
0	1

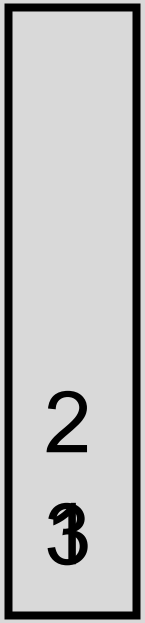
Example: 12+31+x2/ ((1+2)x(3+1))/2=6

Postfix Expression

Example: 12+31+x2/ ((1+2)x(3+1))/2=6

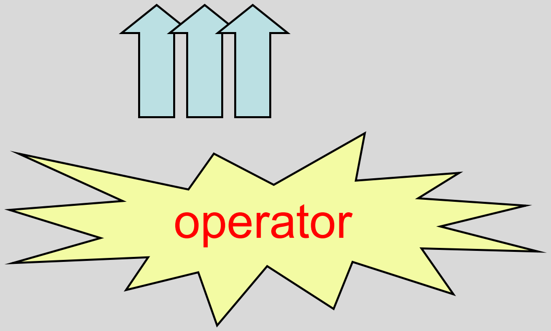


1 + 2 =3

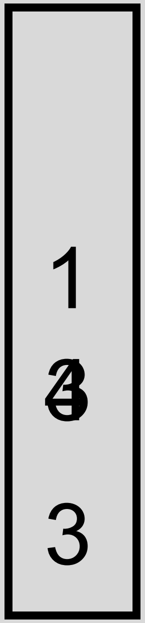


Postfix Expression

Example: 12+31+x2/ ((1+2)x(3+1))/2=6

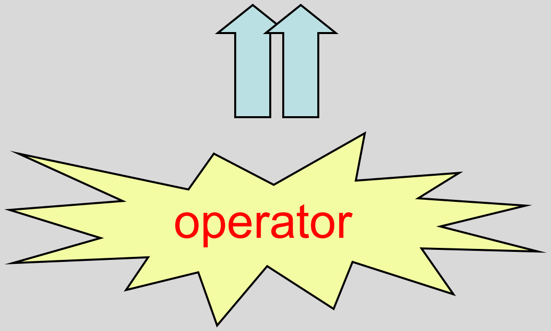


3 + 1 =4

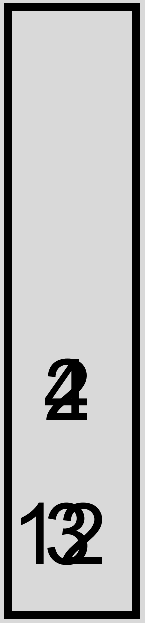


Postfix Expression

Example: 12+31+x2/ ((1+2)x(3+1))/2=6

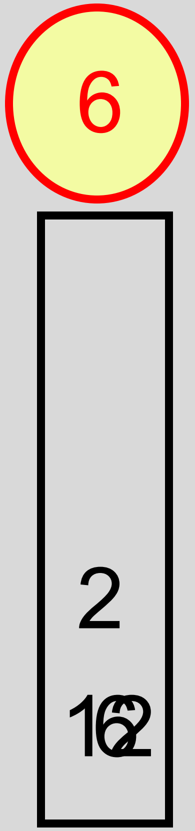


3 x 4 =12



Postfix Expression

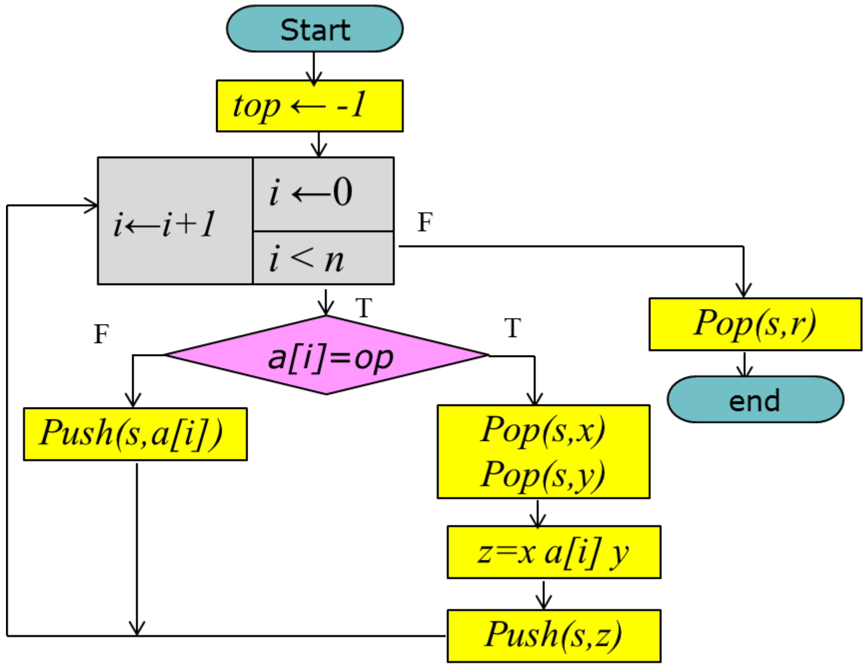
Example: 12+31+x2/ ((1+2)x(3+1))/2=6

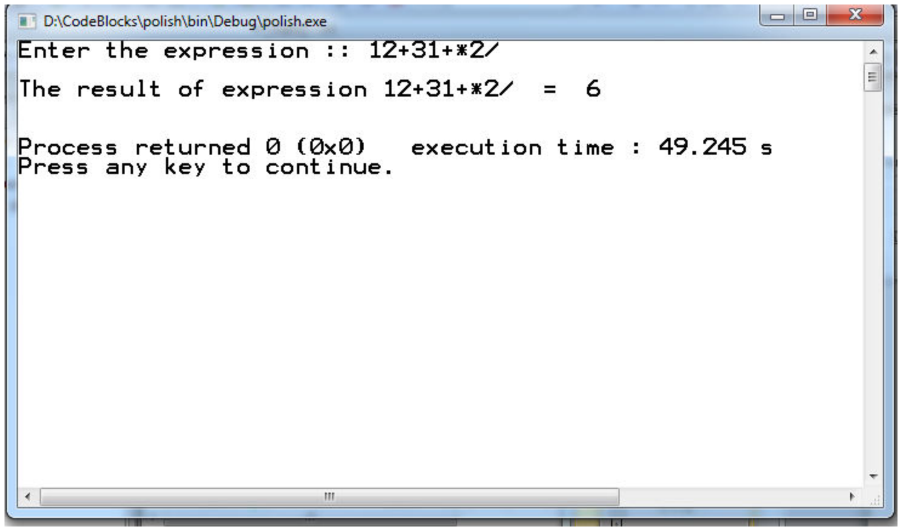


operator

12 / 2 =6

Postfix Expression





```
Enter the expression :: 12+31+*2/
The result of expression 12+31+*2/  =  6

Process returned 0 (0x0)   execution time : 49.245 s
Press any key to continue.
```

```
#include <iostream>
using namespace std;

class Polish{
  int stack[20];
  int top = -1;
 public:
  void push(int x)
  {  stack[++top] = x;   }
  int pop()
   {  return stack[top--];

   void PostFix(char *e){
     int n1,n2,n3,num;

     while(*e != '\0')
       {
            if(isdigit(*e))
            {
                num = *e - 48;
                push(num);
            }
            else
            {
                  n1 = pop();
                  n2 = pop();
                  switch(*e)
                  {
                     case '+': n3 = n1 + n2;break;
                     case '-': n3 = n2 - n1;break;
                     case '*': n3 = n1 * n2;break;
                     case '/': n3 = n2 / n1;break;
                  }
                  push(n3);
            }
            e++;
       }
       cout<<"="<<pop();
   }
};
```

```
int main()
{
      char exp[20];
      char *e;
      cout<<"Enter the expression :: ";
      cin>>exp;
      Polish p;
      cout<<"\nThe result of expression "<<exp;
      p.PostFix(exp);
      return 0;
}
```

Assignments

Prob 1: An array c[] stores characters (alphabets and digits) then write an algorithm that creates two stacks to store alphabets and digits respectively from c[].

Prob 2: Write an algorithm that converts an infix expression to its prefix equivalent