DATA STRUCTURES (ITPC-203)

Stacks

Mrs. Sanga G. Chaki

Department of Information Technology

Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

- What is a stack?
- Abstract Data Type,
- Primitive Stack operations: Push & Pop,
- Array and Linked Implementation of Stack
- Application of stack
- Prefix and Postfix Expressions,
- Evaluation of postfix expression

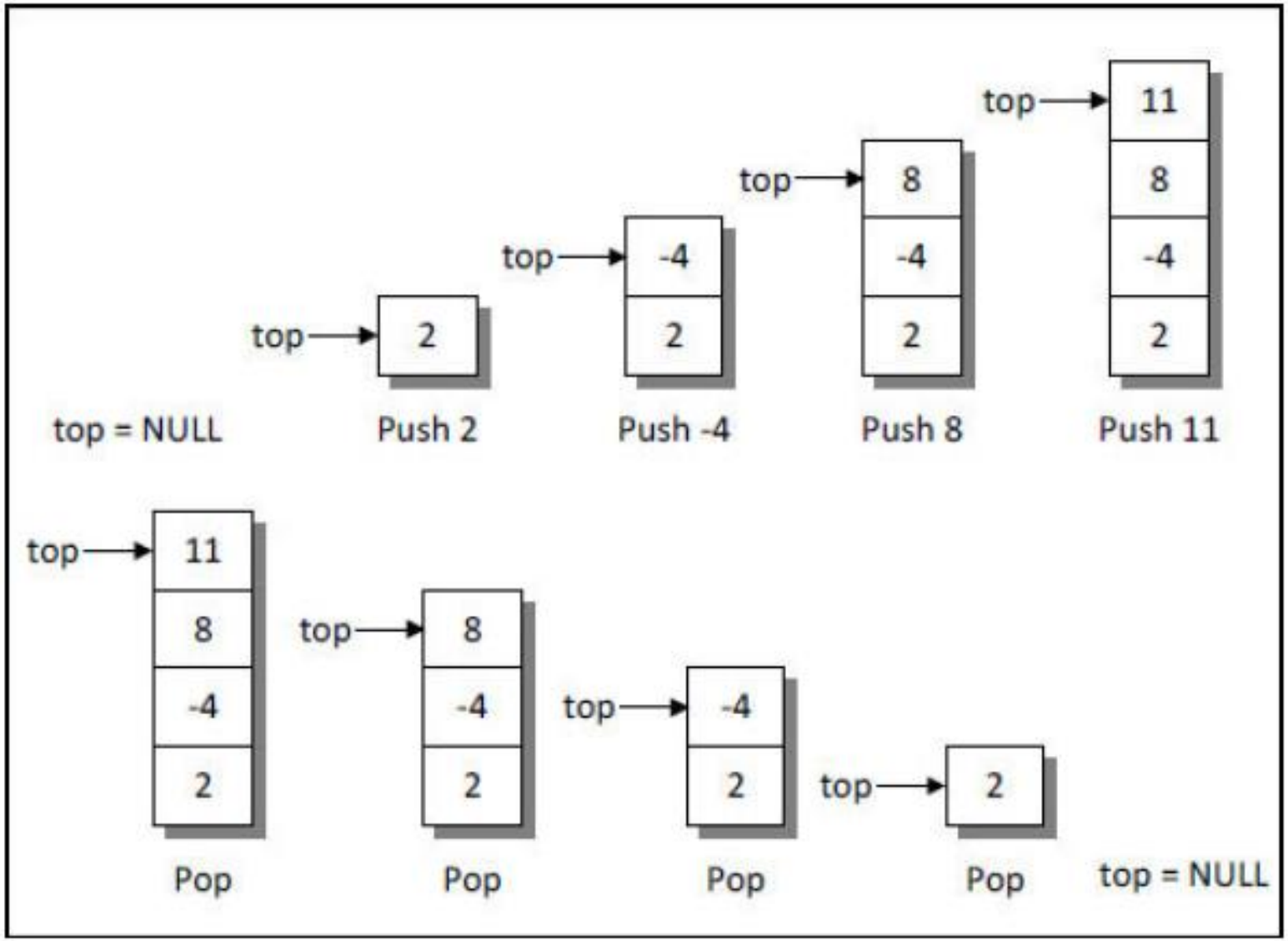
What is a Stack?

1. Data structure for storing data
2. The order in which the data arrives is important
3. Its literally like a stack of plates – one stacked on top of the others
4. When a plate a required, it is taken from the top of the pile
5. The first plate placed on the stack is the last one to be used.
6. Definition:
   - A stack is an ordered list, in which insertion and deletion are done at one end which is called as the top of the stack.
   - The last element to be inserted is the first one to be deleted.
   - So, it is called a last in first out (LIFO) list.

What is a Stack?

1. Special names given to the two changes that can be made to a stack
2. **Push**: when an element is inserted in a stack
3. **Pop**: when an element is deleted from a stack
4. Trying to pop out an empty stack is **underflow**
5. Trying to push an element in a full stack is an **overflow**
6. These are generally exceptions – causing error

Stack – Insertion and Deletion



Stack as an ADT

1. The main operations are: assuming that data is of type int
   - push (int data) – inserts data onto stack
   - int pop() – removes and returns the last inserted element from the stack.
2. Auxiliary operations:
   - int top() – returns the last inserted element without removing it
   - int size() – returns number of elements stored in stack
   - int isEmptyStack() – indicated whether any elements are stored in the stack or not
   - int isFullStack() – indicated whether the stack is full or not

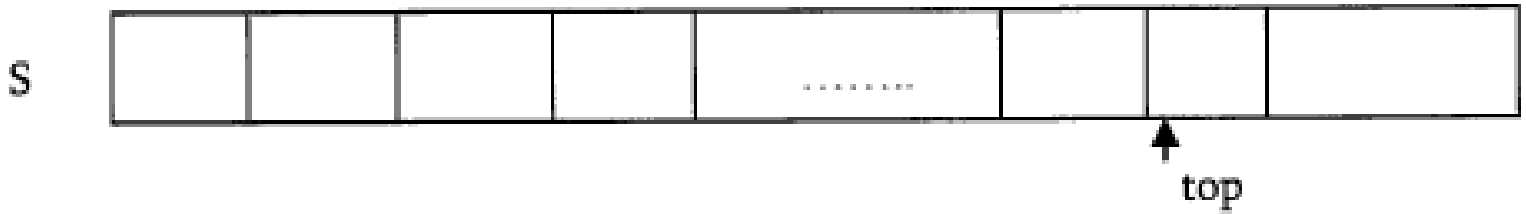
Stack Implementation

1. Simple array based
2. Linked lists based

Array Based Implementation of Stacks

Stack Implementation – Array based

1. Use an array to store the elements
2. Insert elements from let to right
3. Use a variable to keep track of the top elements



4. When the array storing the stack elements is full, a push() operation will throw a full stack exception
5. Similarly, deleting from empty will also throw a stack empty exception
6. **So, when we use array to store elements of a stack the stack can grow or shrink within the memory reserved for the array**

Stack Implementation – Array based

```
struct stack
{
   int arr[ MAX ];
   int top;
};
```

```
void initstack (struct stack *);
void push (struct stack *, int item);
int pop (struct stack *);
```

The variable top is an index into this array.

Stack Implementation – Array based - Initialization

```
struct stack
{
   int arr[ MAX ];
   int top;
};
```

```
void initstack (struct stack *s)
{
   s -> top = -1;
}
```

The variable top is an index into this array.
To indicate that the stack is empty to begin with, the variable top is set with a value -1

Stack Implementation – Array based - Push

```
/* adds an element to the stack */
void push (struct stack *s, int item)
{
    if (s -> top == MAX - 1)
    {
       printf ("Stack is full\n");
       return;
    }
    s -> top++;
    s -> arr[ s ->top ] = item;
 }
```

Stack Implementation – Array based - Pop

```
/* removes an element from the stack */
int pop (struct stack *s)
{
   int data;
   if (s -> top == -1)
   {
      printf ("Stack is empty\n");
      return NULL;
   }
   data = s -> arr[ s -> top ];
   s -> top--;
   return data;
}
```

Stack Implementation – Array based – Performance and Limitation

1. If number of elements in stack = n
2. Space complexity for n push operations = O(n)
3. Time complexity of push() and pop() = O(1)
4. Limitation: fixed size

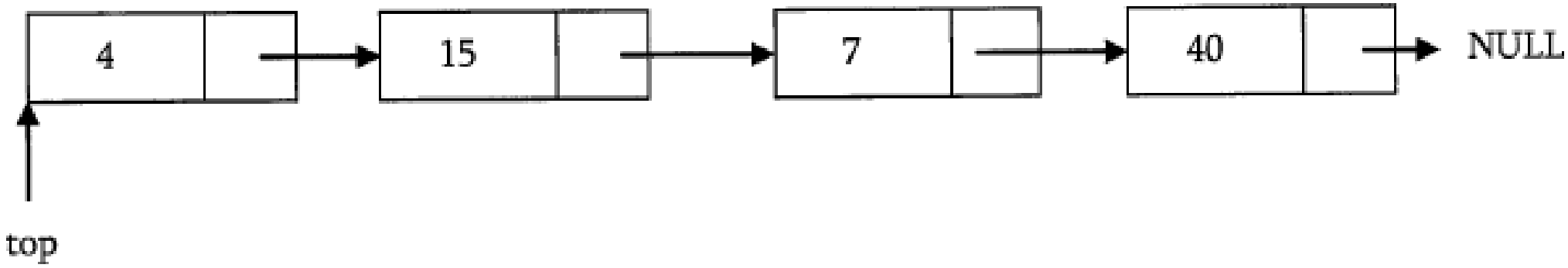
Linked List Based Implementation of Stacks

Implementation of a stack using a linked list

1. A linked list is an ordered list, and can be used to implement a stack.
2. The problem of fixed size (arrays) can be overcome if we implement a stack using a linked list.
3. A linked list has two ends: head and tail. So, where should the top be?
4. Both insertion and deletion are easy at the head.
5. Insertion is easy at the tail if we maintain a tail pointer.
6. Deletion is time-consuming even in the presence of a tail pointer.
7. So the obvious choice is:
   - **The top of the stack is at the head of the linked list**.

Stack Implementation – Linked List based

1. The pointer to the beginning of the list (head) serves the purpose of the top of the stack.
2. So push() and pop() happens at the beginning of the list.



Stack Implementation – Linked List based

```
struct node
{
   int data;
   struct node *link;
};
void push (struct node **, int);
int pop (struct node **);
```

Stack Implementation – Linked List based – push()

```
/* adds a new node at beginning of linked list */
void push (struct node **top, int item)
{
   struct node *temp;
   temp = (struct node *) malloc (sizeof (struct node));

   if (temp == NULL)
      printf ("Stack is full\n");

   temp -> data = item;
   temp -> link = *top;
   *top = temp;
}
```

Stack Implementation – Linked List based – pop()

```
/* deletes a node from beginning of linked list */
int pop (struct node **top)
{
   struct node *temp;
   int item;
   if (*top == NULL)
   {
      printf ("Stack is empty\n");
      return NULL;
   }
   temp = *top;
   item = temp -> data;
   *top = (*top) -> link;

   free (temp);
   return item;
}
```

Applications of Stack

Applications of Stacks

1. Evaluation of postfix expression
2. Conversion from infix to postfix

Infix, Prefix and Postfix

1. **Stacks are often used is in evaluation of arithmetic expression.**
2. **An arithmetic expression consists of operands and operators.**
3. **The operator is placed between two operands is called infix notation**.
4. There are some operator precedence for evaluation: (), *, /, +, -
5. Polish notation: As per this notation, an expression in infix form can be converted to either prefix or postfix form and then evaluated.
6. **In prefix notation the operator comes before the operands.**
7. **In postfix notation, the operator follows the two operands.**
8. Eg: These forms are shown below.
    - A + B - Infix form
    - + A B - Prefix form
    - A B + - Postfix form

Infix, Prefix and Postfix

1. The prefix and postfix expressions have three features:
    - The operands maintain the same order as in the equivalent infix expression
    - Parentheses are not needed to designate the expression unambiguously.
    - While evaluating the expression the priority of the operators is irrelevant.
2. In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct.

Postfix Notation

1. The infix expression: A * (B + C) / D
2. Equivalent postfix: A B C + * D /
3. The order of evaluation of operators is always left-to-right
4. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication. ((A (B C +) *) D /)
5. **Operators act on the two values immediately to the left of them**.
6. For example, the "+" above uses the "B" and "C".
7. Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".
8. Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

Prefix Notation

1. The infix expression: A * (B + C) / D
2. Equivalent postfix: / * A + B C D
3. Operators are evaluated left-to-right.
4. **Operators act on the two nearest values on the right**.
   - (/ (* A (+ B C)) D)
5. Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

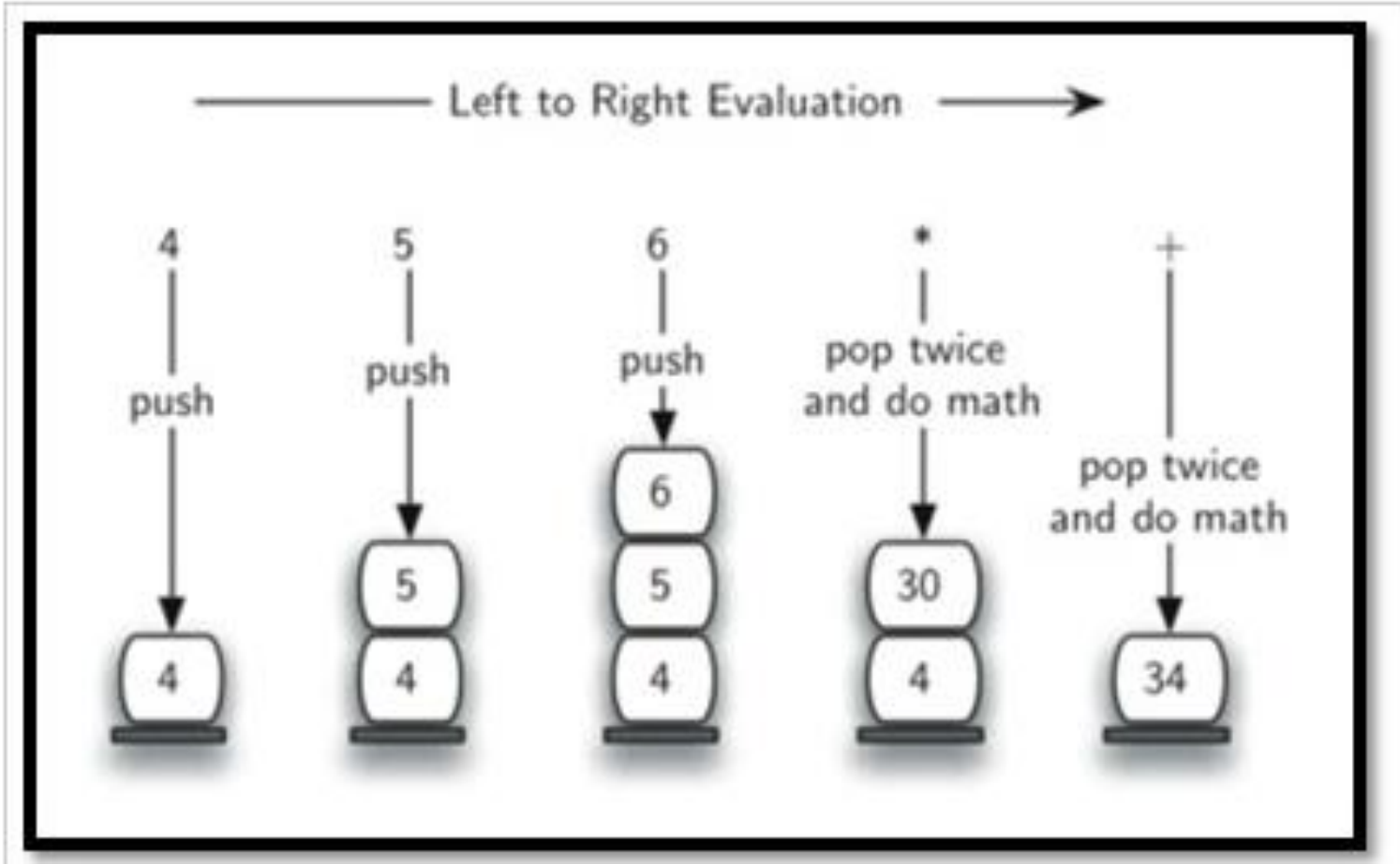
Evaluating Postfix Expression

Evaluating Postfix Expression

1. The postfix expression can be evaluated by
    - Scan expression from left to right
    - Keep on storing the operands into a stack.
    - Once an operator is received, pop the two topmost elements
    - Evaluate them and push the result in the stack again.
2. **The stack contents are the operands**

Evaluating Postfix Expression

Expression: 456*+



Evaluating Postfix Expression

1. **Example:**
2. **4 2 $ 3 * 3 - 8 4 / 1 1 + / +**
3. **Result = 46**
4. **How?**
5. **The stack contents are the operands**

Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Char. Scanned	Stack Contents
4	4
2	4, 2
$	16
3	16, 3
*	48
3	48, 3
-	45
8	45, 8
4	45, 8, 4
/	45, 2
1	45, 2, 1
1	45, 2, 1, 1
+	45, 2, 2
/	45, 1
+	46 (Result)

Evaluating Postfix Expression – Pseudocode

1. Postfix stack structure

```
struct postfix
{
    int stack[ 100 ];
    int top, nn; //store top of stack and
    int operand (temporary)
    char *s; //string to store operators
};
```

Possible functions

1. void push (struct postfix *, int);
2. int pop (struct postfix *);
3. void calculate (struct postfix *);

1. What is the array for?
   - Store the operands and intermediate results of the evaluation
2. The evaluation of the expression gets performed in the calculate() function.

Evaluating Postfix Expression – Pseudocode

```
void calculate(struct postfix *p){/* evaluates the postfix expression */
int n1, n2, n3;
while (*(p -> s)){
    if (isdigit (*(p -> s))) { /* if digit is encountered */
        p -> nn = *(p -> s) - '0'; //convert each char to the int it represents
        push (p, p -> nn); }

    else{/* if operator is encountered */
        n1 = pop (p);
        n2 = pop (p);
        switch (*(p -> s)){
            case '+' : n3 = n2 + n1; break;
            case '-' : n3 = n2 - n1; break; and so on for other operators ….}
        push (p, n3); }
    p -> s++; }
}
```

Evaluating Postfix Expression – Pseudocode

1. In the calculate() function, this expression gets scanned character by character
2. If the character scanned is an operand, then first it is converted to a digit form (from string form), and then it is pushed onto the stack.
   - The easiest way to convert a single character to the int it represents is to subtract the value of '0'. Eg. If we take '3' (ASCII 51) and subtract '0' (ASCII 48), we are left with int 3.
3. If the character scanned is an operator, then the top two elements from the stack are popped
4. The arithmetic operation is performed between them and
5. The result is then pushed back onto the stack.
6. These steps are repeated as long as the input postfix expression is not exhausted.

Evaluating Postfix Expression – Example

1. Evaluate the following postfix expressions:
2. 2 3 1 * + 9 – (-4)
3. 100 200 + 2 / 5 * 7 + (757)
4. 5 4 6 + * 4 9 3 / + *
5. a b c * + d e * f + g * + where a = 1, b= 2, c = 3, d = 4, e = 5, f = 6, g = 2
6. a b + c d / - where a = 5, b = 4, c =9, d = 3
7. What is the time complexity for this conversion?

TILL HERE

Infix to Postfix Conversion using Stack

Converting between these notations

1. The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation e.g.:

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+ (* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/ (* A (+ B C)) D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

2. You can convert directly between these bracketed forms simply by moving the operator within the brackets e.g. (X + Y) or (X Y +) or (+ X Y).
3. Repeat this for all the operators in an expression, and finally remove any superfluous brackets.
4. The stack data structure is used while carrying out the conversion of an expression given in one form to another.

Infix to Postfix Conversion using Stack - Algorithm

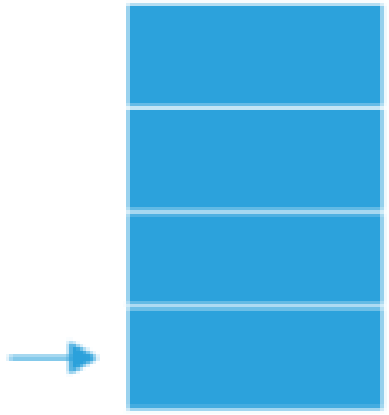
1. Scan all the symbols one by one from left to right in the given Infix Expression.
2. If symbol is an operand, then immediately append it to the Postfix Expression.
3. If the symbol is left parenthesis '(', then Push it onto the Stack.
4. If symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
5. If symbol is an operator (+, –, *, /), then Push it onto the Stack.
   - However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix.
   - If an open parenthesis is there on top of the stack then push the operator into the stack.
6. If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

Infix to Postfix Conversion using Stack - Examples

Infix expression => a * (b + c + d)

(1)

Infix	Stack	Postfix
a * (b + c + d)		a

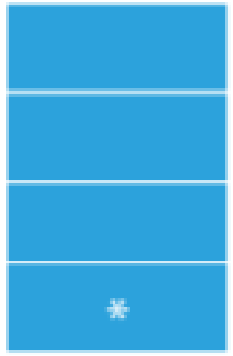


'a' is an operand append it to the post fix

Infix to Postfix Conversion using Stack - Examples

(2)

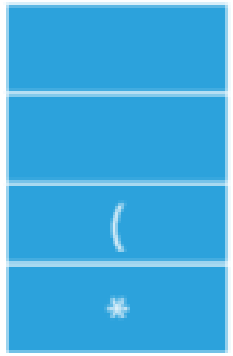
Infix	Stack	Postfix
a * (b + c + d)		a



'*' is an operator push it into the stack after removing the higher or equal precedence operator.

(3)

Infix	Stack	Postfix
a * (b + c + d)		a



'(' is left parenthesis push it into the stack

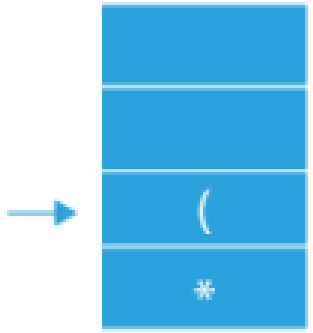
Infix to Postfix Conversion using Stack - Examples

(4) Infix: a * (b + c + d)

Stack | Postfix

ab

'b' is operand append to postfix

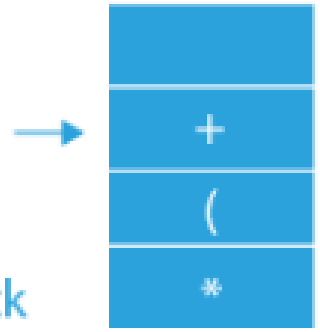


(5) Infix: a * (b + c + d)

Stack | Postfix

ab

'+' is an operator follow the step 5 of algorithm. Top of stack is open parantheses, So just push it.



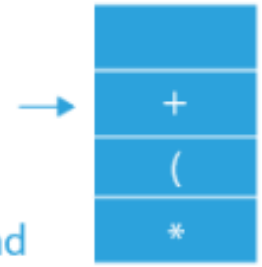
Infix to Postfix Conversion using Stack - Examples

(6) Infix: a * (b + c + d)

Stack

Postfix: abc

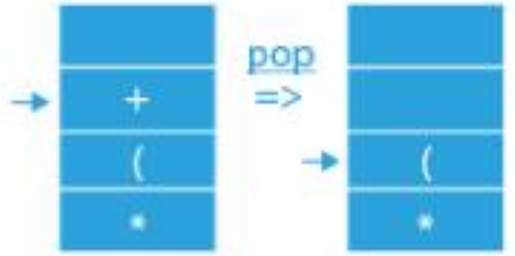
'c' is an operator follow the step 2 of algorithm and append it to the postfix.



(7) Infix: a * (b + c + d)

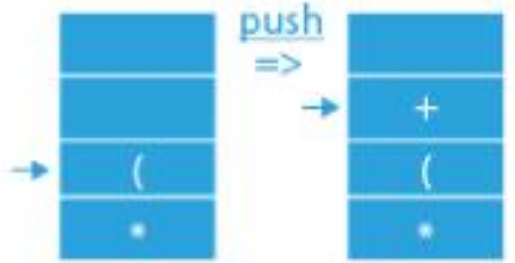
=> '+' is an operator, so push it into stack after removing higher or equal priority operators from top of stack.

=> Current top of stack is '+', it has equal precedence with with input symbol '+'. So, pop it and append to postfix.



postfix => a b c +

=> Now the top of stack is '(', so push the input operator.

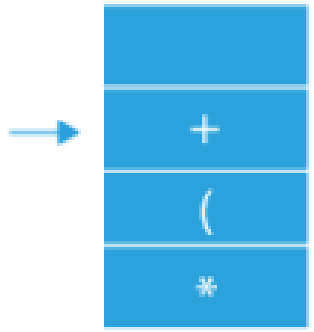


Infix to Postfix Conversion using Stack - Examples

(8)

Infix	Stack	Postfix
a * (b + c + d)		a b c + d

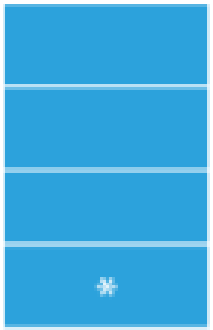
'd' is an operand.
append it to postfix.



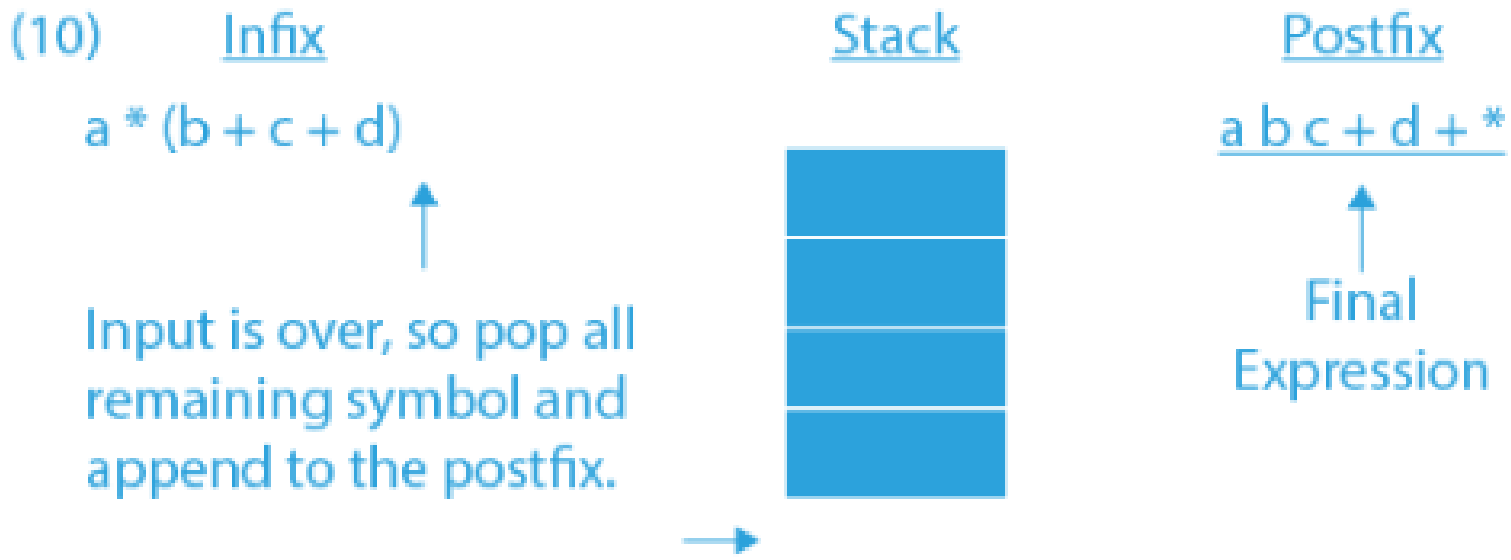
(9)

Infix	Stack	Postfix
a * (b + c + d)		a b c + d +

')' is an right paranthesis,
flow step n of algorithm
and pop all content of
stack until respective '('
is popped and append
to the output.



Infix to Postfix Conversion using Stack - Examples



Work out the examples in slide 27 with this algorithm and cross check the postfix expressions

Infix to Postfix Conversion

1. 2+3+4
2. (2+3)*(4+5)
3. a/(b+c)
4. a*b+c*d
5. (a+(b*c)/(d-e))
6. 4$2*3-3+8/4/(1+1)

Infix to Postfix Conversion

Infix Expression: 4 $ 2 * 3 - 3 + 8 / 4 / (1 + 1)		
Char Scanned	**Stack Contents**	**Postfix Expression**
4	Empty	4
$	$	4
2	$	4 2
*	*	4 2 $
3	*	4 2 $ 3
-	-	4 2 $ 3 *
3	-	4 2 $ 3 * 3
+	+	4 2 $ 3 * 3 -
8	+	4 2 $ 3 * 3 - 8
/	+ /	4 2 $ 3 * 3 - 8
4	+ /	4 2 $ 3 * 3 - 8 4
/	+ /	4 2 $ 3 * 3 - 8 4 /
(	+ / (	4 2 $ 3 * 3 - 8 4 /
1	+ / (	4 2 $ 3 * 3 - 8 4 / 1
+	+ / (+	4 2 $ 3 * 3 - 8 4 / 1
1	+ / (+	4 2 $ 3 * 3 - 8 4 / 1 1
)	+ /	4 2 $ 3 * 3 - 8 4 / 1 1 +
	Empty	4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Infix to Postfix Conversion - Pseudocode

```
struct infix
{
    char target[ MAX ];
    char stack[ MAX ];
    char *s, *t;
    int top;
};
```

Possible Functions:

1. void push (struct infix *, char);
2. char pop (struct infix *);
3. void convert (struct infix *);
4. int priority (char);

TILL HERE