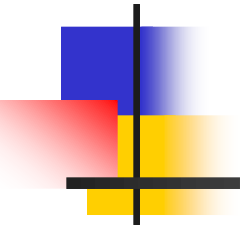
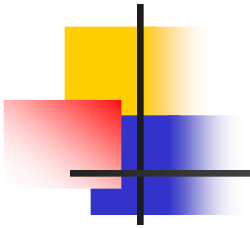


# Chapter 3

STACKS



# Objective

- To introduce:
  - Stack concepts
  - Stack operations
  - Stack applications

**CONTENT**

2.1 Introduction

2.2 Stack using Array-Based Implementation

2.3 Stack Application: Postponing Data Usage

- Infix To Postfix Transformation
- Evaluating Postfix Expressions
- Infix to Prefix Transformation

# 2.1 Introduction

- Stack is a linear list in which data items can only be accessed at one end, called the **top** of the stack.
- Last-In–First-Out (LIFO) concept.

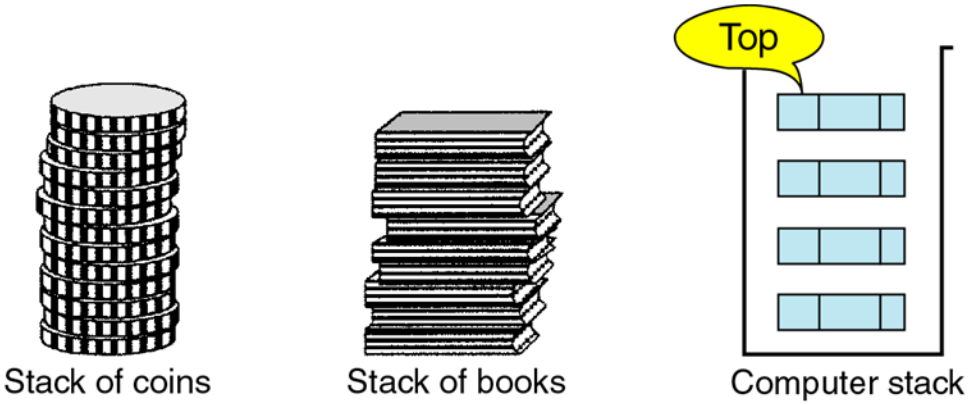



Fig. 1: Stack

# 2.1 Introduction

- Data item: Simple data types.
- Stack applications classified into 4 broad categories:
  - ✓ Reversing data – e.g. reverse a list & convert decimal to binary.

    Eg. 26 = 110102
  - ✓ Parsing data – e.g. translate a source program to machine language.
  - ✓ Postponing data usage – e.g. evaluation, transformation.
  - ✓ Backtracking – e.g. computer gaming, decision analysis, expert systems.

# 2.1 Introduction

- Basic operations:
  - ✓ Peek – if the stack is not empty, return its top element.
  - ✓ Pop – if the stack is not empty, delete and return its top element.
  - ✓ Push – add a given element to the top of the stack.

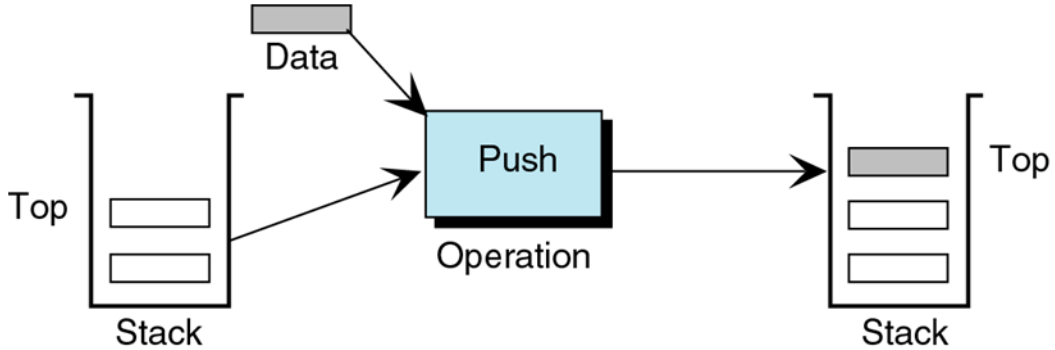



Fig. 2: Push stack operation

# 2.1 Introduction

- Size – return the number of elements in the stack.
- Pop – remove the top element of the stack and return it to the user.

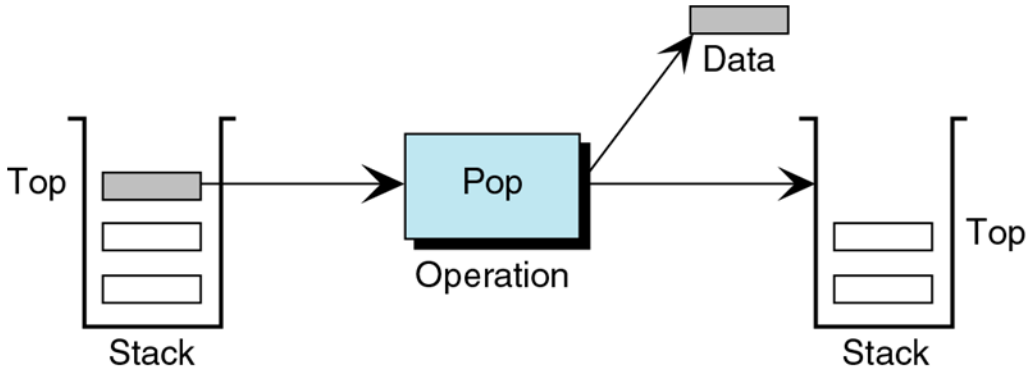



Fig. 3: Pop stack operation

# 2.1 Introduction

- Peek – copies the element at the top of the stack; return the data in the top element to the user but does not delete it.

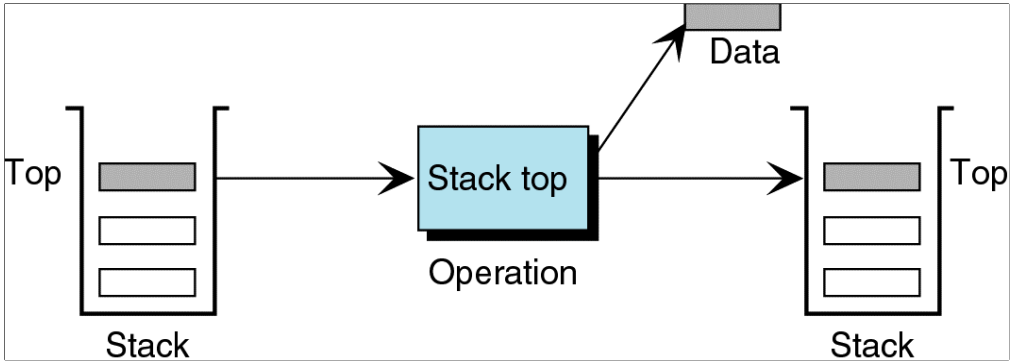



Fig. 3: Peek stack operation

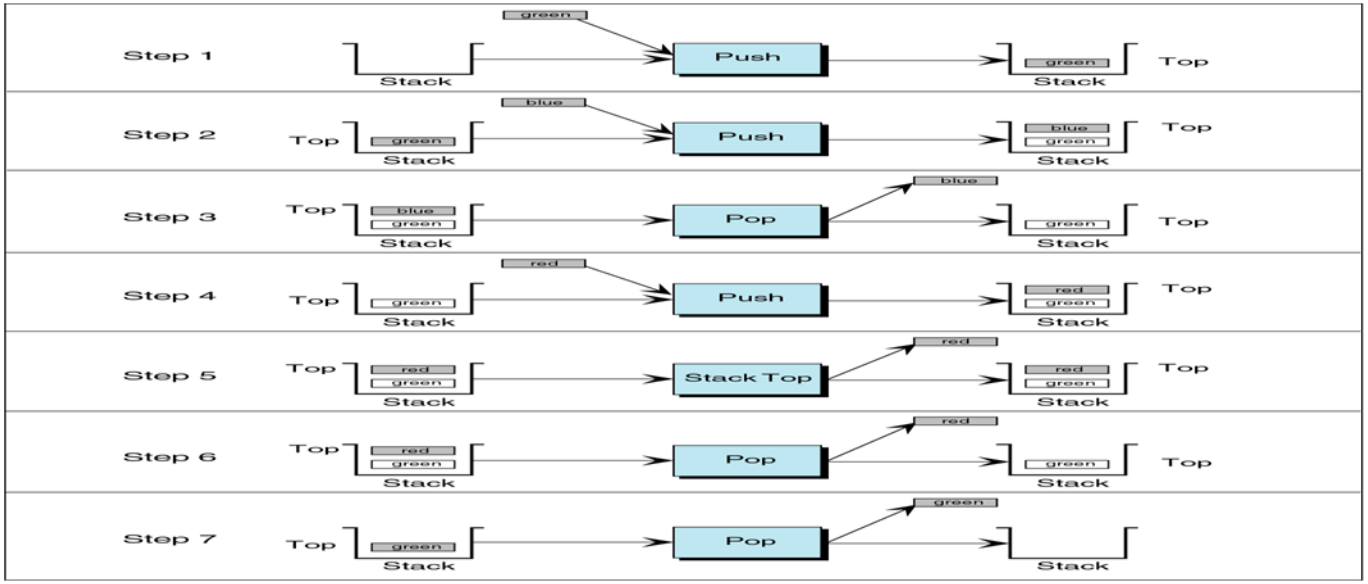
# 2.1 Introduction

- Example:

  Given stack S with list of operations as below. Illustrate the stack operations step by step:

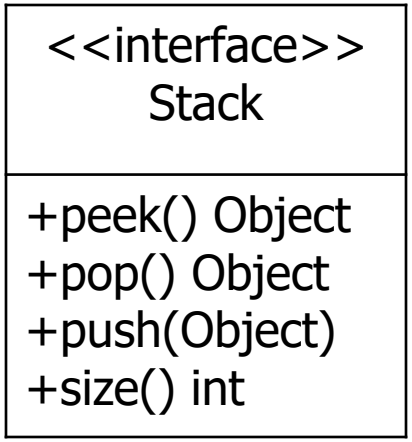
  - S.push (green), S.push (blue), S.pop( ), S.push(red),

    topItem = S.peek( ), S.pop( ), S.pop( )



# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

- Declarations of data structure and methods prototype for Stack operations in a Header File.



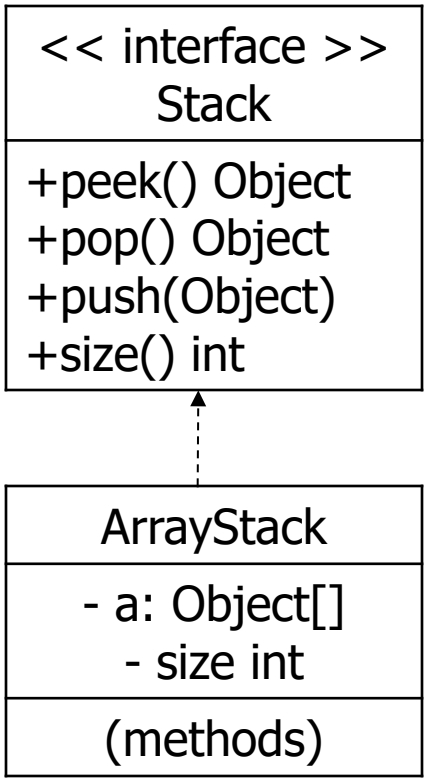
# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

- A Stack Interface:

```
1 /**
2 * The <code>Stack</code> interface specifies the basic operations
3 * of a last-in-first-out (LIFO) containers.
4 */
5 public interface Stack {
6
7    /**
8    * Returns a reference to the top element on this stack, leaving
9    * the stack unchanged.
10   *
11   * @return the element at the top of this stack.
12   * @throws IllegalStateException if this stack is empty.
13   */
14   public Object peek();
15
16   /**
17   * Removes and returns the element at the top of this stack.
18   *
19   * @return the element at the top of this stack.
20   * @throws IllegalStateException if this stack is empty.
21   */
22   public Object pop();
23
24   /**
25   * Adds the specified element to the top of this stack.
26   *
27   * @param object the element to be pushed onto this stack.
28   */
29   public void push(Object object);
30
31   /**
32   * Returns the number of elements in this stack.
33   *
34   * @return the number of elements in this queue.
35   */
36   public int size();
37 }
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

- The implementation file begins as follows:



# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

**An ArrayStack Class**

```
public class ArrayStack implements Stack {
        private Object[] a;
       private int size;
}
```

- The implementations of the class's member functions are included at this point in the implementation file.

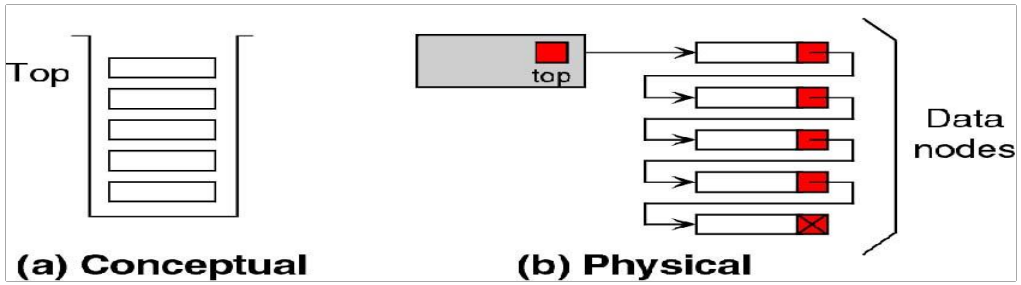



Fig. 6: Conceptual & physical stack implementation

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

- Create Stack
  - ✓ Create stack initialize the stack pointer (i.e. top) to NULL, this indicates top as the new empty stack.
  - ✓ Function definition:

```
public ArrayStack(int capacity) {
  a = new Object[capacity];
}
```

- Empty Stack
  - ✓ Check if stack is empty.
  - ✓ Function definition:

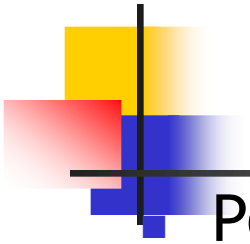
```
public boolean isEmpty() {
  return (size == 0);
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

- Push
  - ✓ Adds an item at the top of the stack.
  - ✓ If there is not enough space to insert - overflow state.

```
public void push(Object object) {
        if (size == a.length) resize();
                a[size++] = object;
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION



Pop

```
// retrieves and removes the top of a stack


public Object pop() {
    if (size == 0) throw new IllegalStateException("stack is
      empty");
   Object object = a[--size];
   a[size] = null;
   return object;
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

## Peek

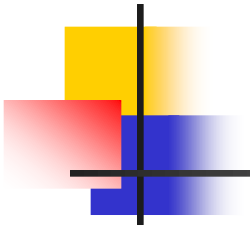
- Function definition:

```
public Object peek() {
    if (size == 0) throw new IllegalStateException("stack
    is empty");
    return a[size-1];
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

Resize an array

```
private void resize()  {
        Object[] aa = a;
        a = new Object[2*aa.length];
        System.arraycopy(aa, 0, a, 0, size);
        }
```



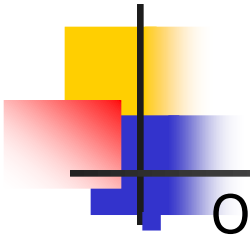
```
public int size() {
    return size;
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION

## Implementation:

```
public class Test ArrayStack {
   public static void main (String[] args) {
      Stack crates = new ArrayStack(4);
      crates.push(“CARROTS”);
      crates.push(“ORANGES”);
      crates.push(“RAISINS”);
      crates.push(“PICKLES”);
      crates.push(“BANANAS”);
      system.out.println(“crates.size(): “ + crates.size() +
            “\tcrates.peek(): “ + crates.peek());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.size(): “ + crates.size() +
            “\tcrates.peek(): “ + crates.peek());
      crates.push(“WALNUTS”);
      crates.push(“OYSTERS”);
      system.out.println(“crates.size(): “ + crates.size() +
            “\tcrates.peek(): “ + crates.peek());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
      system.out.println(“crates.pop(): “ + crates.pop());
   }
}
```

# 2.2 STACK USING ARRAY-BASED IMPLEMENTATION



Output:

```
crates.size(): 5     crates.peek(): BANANAS
crates.pop(): BANANAS
crates.pop(): PICKLES
crates.pop(): RAISINS
crates.size(): 2     crates.peek(): ORANGES
crates.size(): 4     crates.peek(): OYSTERS
crates.pop(): OYSTERS
crates.pop(): WALNUTS
crates.pop(): ORANGES
crates.pop(): CARROTS
java.lang.IllegalStateException: stack is empty
Exception in thread main
```

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- **ARITHMETIC STATEMENT**
  - ✓ Example:

    A * B + C (How computers generate it???)
  - ✓ Arithmetic expression written in INFIX as above example.
  - ✓ However compiler change to POSTFIX/PREFIX for calculating purposes.
  - ✓ Three different formats:

    Infix: **A + B** the operator comes between two operands.

    Prefix: **+AB** the operator comes before the two operands.

    Postfix: **AB+** the operator comes after its two operands.

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- How to convert INFIX expression to POSTFIX and PREFIX?

Example:

Infix form: 2 + 5 * 3 - 1

Convert to postfix: 2 5 3 * + 1 –

| INFIX | PREFIX | POSTFIX |
|---|---|---|
| A+B*C | | |
| (A+B)/(C-D) | | |
| (A+B)*C | | |
| A/B-C/D | | |
| A+B-C*D | | |

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

**Evaluate postfix expression using Stack**

- To calculate INFIX expression requires priority like ()*/ etc whereas POSTFIX and PREFIX are not.

1. Initialize an empty stack.
2. Repeat the following until the end of the expression is encountered:

2.1 **Get next token (**/constant, variable, arithmetic operator) in the postfix expression.

2.2 **If token is an operand, push it onto the stack.**

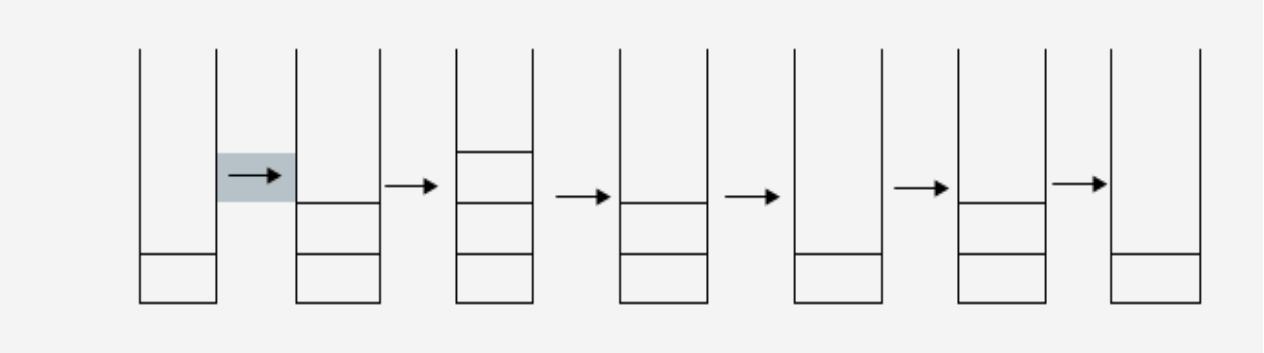
**If it is an operator, then**

i. Pop top two values from the stack.
   If stack does not contain two items, error due to a malformed postfix.
   Evaluation terminated.
ii. Apply the operator to these two values.
iii. Push the resulting value back onto the stack.

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

3. When the end of expression encountered, its value is on top of the stack (and, in fact, must be the only value in the stack).

Try this: 2 5 3 * + 1 – ?



**Code:**

Using Stack Interface defined above.

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

```
public static void main(String[] args)
{
      char ch;
      Stack s; // create stack
      double operan1, operan2, value;
      System.out.println( "Type postfix expression: ");
      while ( ch = System.in.read() && ch != '\n') {
                if ( Character.isDigit(ch) ) s.push(ch – 48);
                else {
                        operan2 = s.pop();
                        operan1 = s.pop();
                        switch (ch) {
                             case '+' :        s.push(operan1+operan2);
                             break;
                             case '-' :        s.push(operan1-operan2);
                             break;
                             case '*' :        s.push(operan1*operan2);
                             break;
                             case '/' :        s.push(operan1+operan2);
                             break;
                        } // end switch
                } // end else
      } //end while
      value = s.pop();
     System.out.println(" = " + value);
}
```

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- Below is the output:

345*+ = 23.00
345+* = 27.00
34*5+ = 17.00
4321*++ = 9.00
9324+5-*/7+ = 10.00

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

**CONVERT INFIX TO POSTFIX EXPRESSION**

Example:

7 + 2 * 3 ?

1. Start scan from left to right.

   Copy operand 7 to output expression.

   Push operand + into stack.

Output

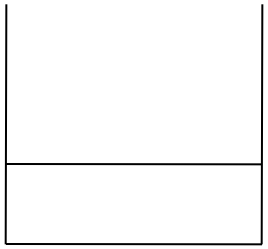
Stack

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

2. Copy operand 2 to output expression.
   Ensure that 2 is the right operand for the previous operator or left operand for the next operator.
   Compare operators' priority.
   Because the priority of * is higher than + then push operator * into stack.

Output

Stack

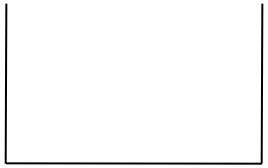


# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

3. Copy operand 3 to output expression.

Output

Stack



4. End Scan. Copy all the operator from stack to output expression.

Output

7 2 3 * +

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- Algorithm: Convert infix to postfix

1. Read infix expression as input.
2. If input is operand, output the operand.
3. If input is an operator +, -, *, /, then
   pop and output all operators of >= precedence. Push operator.
4. If input is (, then push.
5. If input is ), then
   pop and output all operators until you see a ( on the stack. Pop the ( without output.
6. If no more input then pop and output all operators on stack.

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- Hierarchy of operator priority:

| Operator | Stack | Input |
|---|---|---|
| ) | | 0 |
| ( | 0 | 5 |
| + - | 2 | 1 |
| * / | 4 | 3 |

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- Example: 2 * 3 + (4 - 2) convert to Postfix

| Output | Stack | Description |
|---|---|---|

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

- Converting Expression from Infix to Prefix using STACK

  It is a bit trickier algorithm, in this algorithm we first reverse the input expression so that a+b*c will become c*b+a and then we do the conversion and then again the output string is reversed. Doing this has an advantage that except for some minor modifications the algorithm for Infix->Prefix remains almost same as the one for Infix->Postfix.

- Algorithm

  1. Reverse the input string.
  2. Examine the next element in the input.
  3. If it is operand, add it to output string.
  4. If it is Closing parenthesis, push it on stack

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

5. If it is an operator, then
   i. If stack is empty, push operator on stack.
   ii. If the top of stack is closing parenthesis, push operator on stack.
   iii. If it has same or higher priority than the top of stack, push operator on stack.
   iv. Else pop the operator from the stack and add it to output string, repeat step 5.

6. If it is a opening parenthesis, pop operators from stack and add them to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.

7. If there is more input go to step 2

8. If there is no more input, pop the remaining operators and add them to output string.

9. Reverse the output string.

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

Example

Suppose we want to convert 2*3/(2-1)+5*(4-1) into Prefix form:
Reversed Expression: )1-4(*5+)1-2(/3*2

| Char Scanned | Stack Contents (Top on right) | Prefix Expression (right to left) |
|---|---|---|
| ) | ) | |
| 1 | ) | 1 |
| - | )- | 1 |
| 4 | )- | 14 |
| ( | Empty | 14- |
| * | * | 14- |
| 5 | * | 14-5 |
| + | + | 14-5* |
| ) | +) | 14-5* |
| 1 | +) | 14-5*1 |
| - | +)- | 14-5*1 |
| 2 | +)- | 14-5*12 |
| ( | + | 14-5*12- |
| / | +/ | 14-5*12- |
| 3 | +/ | 14-5*12-3 |
| * | +/* | 14-5*12-3 |
| 2 | +/* | 14-5*12-32 |
| eof | Empty | 14-5*12-32*/+ |

Reverse the output string : +/*23-21*5-41
So, the final Prefix Expression is +/*23-21*5-41

# 2.3 STACK APPLICATION: POSTPONING DATA USAGE

**Exercises**

1. Convert the following infix expressions to postfix expressions using the algorithmic method (a stack diagram):

   a) A + B * ( C – D ) / ( E – F )

   b) (A – B + C * ( D + E ) ) / F

   c) A * ( B + C / ( D – E ) )

2. Using the algorithmic method (a stack diagram) evaluate the following postfix expressions when the variables have the following values A is 2, B is 3, C is 4, and D is 5.

   a) A B * C – D +

   b) A B C + * D –