Al-Mustaqbal University

College of Healthcare and Medical Techniques

Intelligent Medical System Department

كلية التقنيات الطبية والصحية

قسم الانظمة الطبية الذكية

Subject: Data Structure

Class: Second

Lecturer: Asst. Prof. Mehdi Ebady Manaa

Lecture: (4)

Stacks II

Application of the stack:

1. Simple Balanced Parentheses**.**
2. Converting Decimal Numbers to Binary Numbers.
3. Infix, Prefix and Postfix Expressions.

1- Simple Balanced Parentheses:

Using a Stack to Process Algebraic Expressions

•Use of parentheses - must be balanced

□Positive Examples:

•A { b [c (d + e)/2 –f] + 1 }

{ [( )] } •

□Negative Examples:

{ 5 + (4 [3 + 2) * 1] }•

{ ([ )] }•

•Use stacks to evaluate parentheses usage

□Scan expression

□Push symbols

□Pop symbols

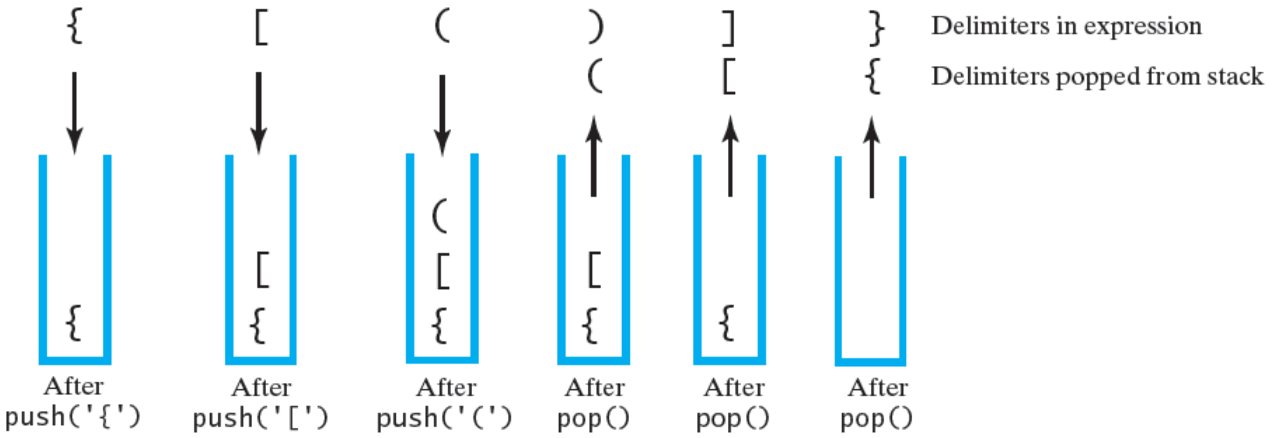
• Test the code with

□{ ([ )] }

□[( )]

□{ [( )]

Figure 5-3 The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }



```
 class OurStack:
  def __init__(self):
    self.items[] =
  def is_empty(self):
    return len(self.items) == 0
  def push(self, item):
    self.items.append(item)
  def pop(self):
    if not self.is_empty:()
      return self.items.pop()
    return None
  def peek(self):
    if not self.is_empty:()
      return self.items[1-]
    return None
  def size(self):
    return len(self.items)
def check_balance(expression):
  open_delimiter_stack = OurStack()
  is_balanced = True
  index = 0
  while is_balanced and index < len(expression):
    next_character = expression[index]
    if next_character in:']})'
      open_delimiter_stack.push(next_character)
    elif next_character in:'[{('
      if open_delimiter_stack.is_empty:()
        is_balanced = False
      else:
        open_delimiter = open_delimiter_stack.pop()
        is_balanced = is_paired(open_delimiter, next_character)
    index += 1
  if not open_delimiter_stack.is_empty:()
    is_balanced = False
  return is_balanced
def is_paired(open_delimiter, close_delimiter):
  return (open_delimiter == '(' and close_delimiter == ')') or\
      (open_delimiter == '[' and close_delimiter == ']') or\
)      open_delimiter == '{' and close_delimiter == '}('
```

```
   #Test cases
  test_cases["{([)]}","[()]}","[()]","{[()]}"]=
  for expression in test_cases:
    result = check_balance(expression)
print(f”Expression ’{expression}’ is {’balanced’ if result else ’not
balanced’}.”)
```

Figure 5-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( ])] }

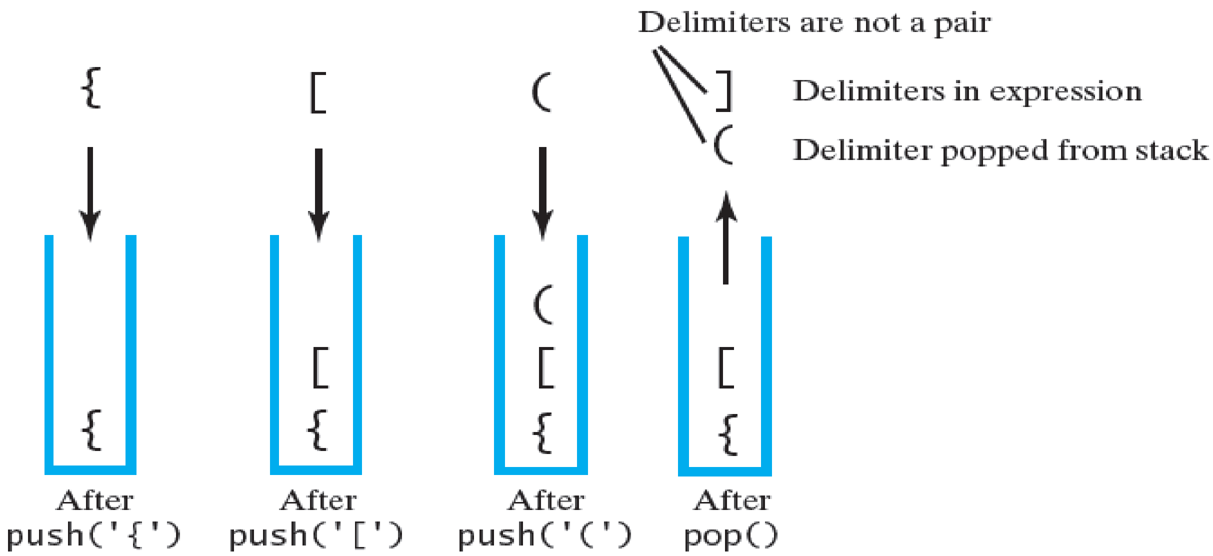


Figure 5-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters [( )] }

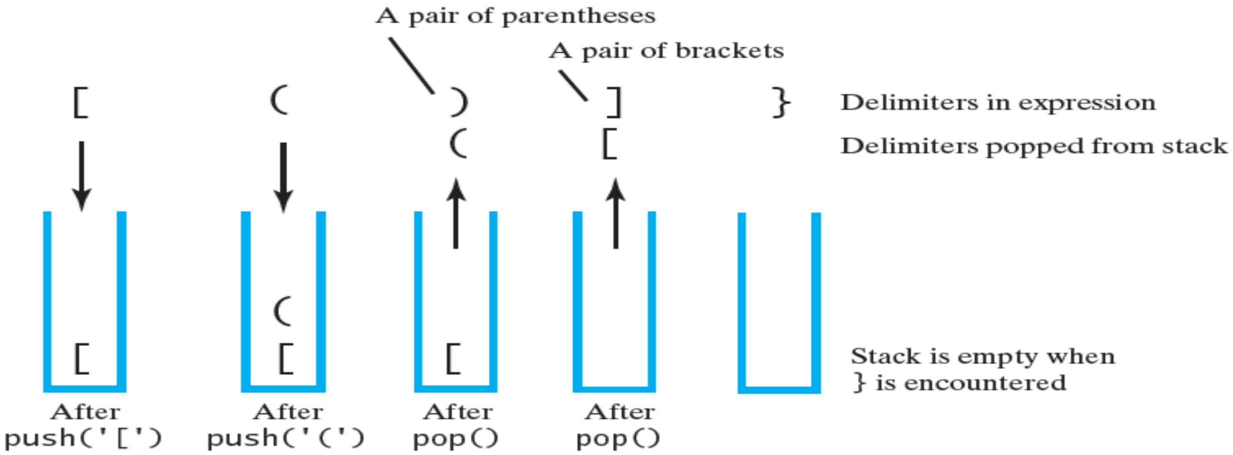
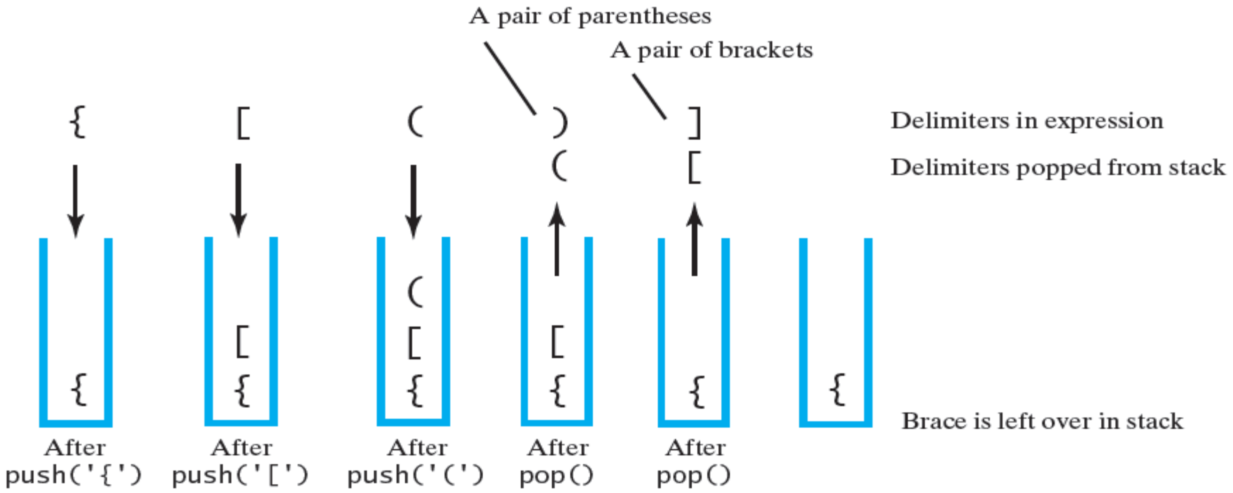
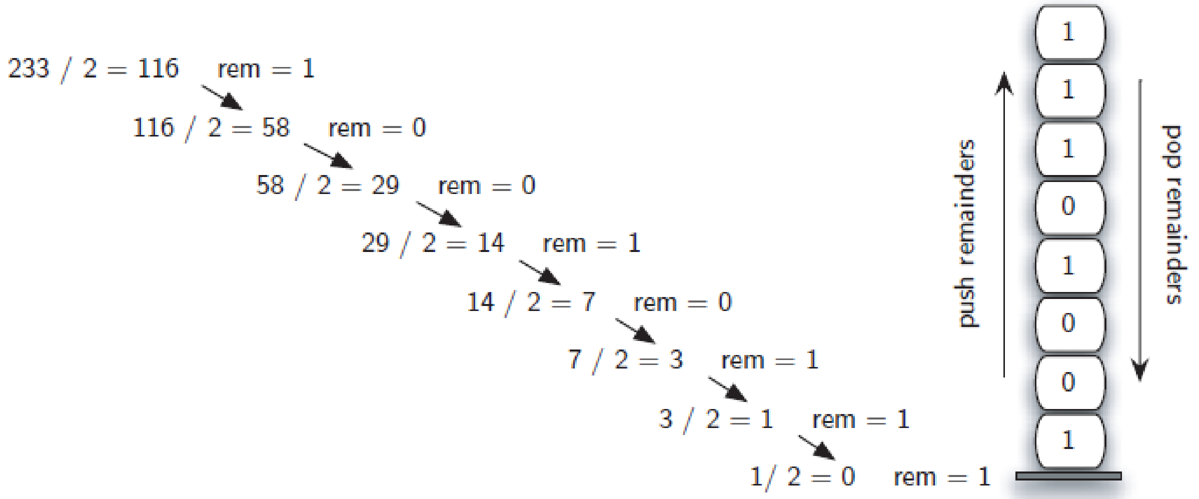


Figure 5-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters { [( )]





```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def push(self, item):
        self.items.append(item)
    def pop(self):
```

```
               if not self.is_empty():
            return self.items.pop()
        return None
def divide_by_2(dec_number):
    rem_stack = Stack()
    while dec_number > 0:
        rem = dec_number % 2
        rem_stack.push(rem)
        dec_number = dec_number // 2
    bin_string = ″″
    while not rem_stack.is_empty():
        bin_string += str(rem_stack.pop())
    return bin_string
def base_converter(dec_number, base):
    digits = ″0123456789ABCDEF″
    rem_stack = Stack()
    while dec_number > 0:
        rem = dec_number % base
        rem_stack.push(rem)
        dec_number = dec_number // base
    new_string = ″″
    while not rem_stack.is_empty():
        new_string += digits[rem_stack.pop()]
    return new_string
# Example usage:
decimal_number = 42
binary_representation = divide_by_2(decimal_number)
print(f″Binary         representation         of         {decimal_number}:
{binary_representation}″)
decimal_number = 255
base = 16
hex_representation = base_converter(decimal_number, base)
print(f″Hexadecimal       representation       of       {decimal_number}:
{hex_representation}″)
```

The Infix, Prefix, Postfix Notation:

Arithmetic expression: An expression is defined as a number of operands or data items combined using several operators. There are basically three types of notations for an expression;

1) Infix notation
2) Prefix notation
3) Postfix notation

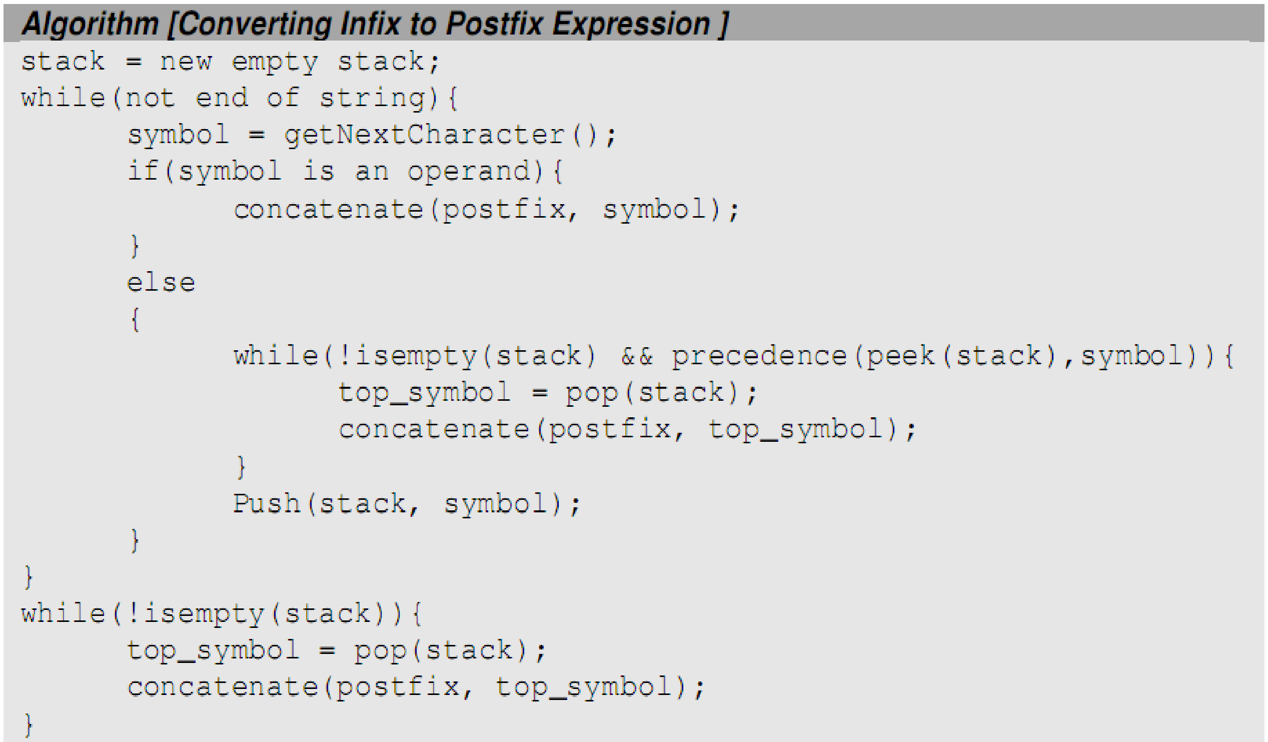
Infix notation: It is most common notation in which, the operator is written or placed in-between the two operands. The expression to add two numbers A and B is written in infix notation as, A+ B In this example, the operator is placed in-between the operands A and B.

Prefix Notation: It is also called Polish notation, refers to the notation in which the operator is placed before the operand as, +AB As the operator '+' is placed before the operands A and B, this notation is called prefix (pre means before).

Postfix Notation: In the postfix notation the operators are written after the operands, so it is called the postfix notation (post means after), it is also known as suffix notation or reverse polish notation. The above postfix if written in postfix notation looks like follows; AB+

Algorithm for Converting Infix into Postfix Expression

The following algorithm converts the infix expression into postfix expression. Java Example



Algorithm [Converting Infix to Postfix Expression]

```
stack = new empty stack;
while(not end of string){
      symbol = getNextCharacter();
      if(symbol is an operand){
            concatenate(postfix, symbol);
      }
      else
      {
            while(!isempty(stack) && precedence(peek(stack),symbol)){
                  top_symbol = pop(stack);
                  concatenate(postfix, top_symbol);
            }
            Push(stack, symbol);
      }
}
while(!isempty(stack)){
      top_symbol = pop(stack);
      concatenate(postfix, top_symbol);
}
```

Example: Suppose we want to convert 2*3/(2-1)+5*(4-1) into postfix expression.

symbol	stack	Postfix
2		2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/5
(	+*(	23*21-/5
4	+*(	23*21-/54
-	+*(-	23*21-/54
1	+*(-	23*21-/541
)	+*	23*21-/541-
		23*21-/541-*+

Pyhton Implementation

```
def precedence(operator):
    if operator == '+' or operator:'-' ==
        return 1
    if operator == '*' or operator:'/' ==
        return 2
    return 0
def infix_to_postfix(infix_expression):
    stack[] =
    postfix_expression[] =

    for char in infix_expression:
        if char.isalnum():  # If the character is an operand, add it to the postfix expression
            postfix_expression.append(char)
        elif char == '(':  # If the character is an opening parenthesis, push it onto the stack
            stack.append(char)
        elif char == ')':  # If the character is a closing parenthesis, pop operators from the
stack and add to postfix until an opening parenthesis is encountered
            while stack and stack:')' =! [1-]
                postfix_expression.append(stack.pop())
            stack.pop()  # Pop the opening parenthesis from the stack
        else:  # If the character is an operator
            while stack and stack[-1] != '(' and precedence(stack[-1]) >= precedence(char:(
                postfix_expression.append(stack.pop())
            stack.append(char)

 #    Pop any remaining operators from the stack and add to postfix expression
    while stack:
        postfix_expression.append(stack.pop())

    return ''.join(postfix_expression)

 #Example usage
infix_expression = "a+b*c-(d/e+f)*g"
postfix_expression = infix_to_postfix(infix_expression)
print("Postfix Expression:", postfix_expression)
```

Algorithm for Evaluating Postfix Expression

The following algorithm evaluates the postfix expression. Java Example

Algorithm [Evaluating a Postfix Expression]

```
stack = new empty stack;
/* scan the input string reading one element at a time into symbol */
while(not end of string){
        symbol = getNextCharacter();
        If(symbol is an operand){
                push(stack, symbol)
        }else{ // symbol is an operator
                operand2 = pop(stack);
                operand1 = pop(stack);
                value = calculate(operand1, symbol, operand2);
                push(stack, value);
        }
}
return (pop(stack));
```

Example : Let us now consider an example. Suppose that we are asked to evaluate the following postfix expression 6 2 + 5 9 * +. ➔(6+2)+(5*9)

symbol	operand2	operand1	value	stack
6				6
2				6, 2
+	2	6	8	8
5				8, 5
9				8, 5, 9
*	9	5	45	8, 45
+	45	8	53	53

```
class Stack:
  def __init__(self):
    self.items[] =
  def is_empty(self):
    return len(self.items) == 0
  def push(self, item):
    self.items.append(item)
  def pop(self):
    if not self.is_empty:()
      return self.items.pop()
    return None
class BalanceChecker:
@   staticmethod
  def check_balance(expression):
    open_delimiter_stack = Stack()
```

```
                        is_balanced = True
        index = 0
        while is_balanced and index < len(expression):
            next_character = expression[index]
            if next_character in:']})'
                open_delimiter_stack.push(next_character)
            elif next_character in:'[{('
                if open_delimiter_stack.is_empty:()
                    is_balanced = False
                else:
                    open_delimiter = open_delimiter_stack.pop()
                    is_balanced = BalanceChecker.is_paired(open_delimiter, next_character)
            index += 1
        if not open_delimiter_stack.is_empty:()
            is_balanced = False
        return is_balanced
@    staticmethod
    def is_paired(open_delimiter, close_delimiter):
        return (open_delimiter == '(' and close_delimiter == ')') or\
               (open_delimiter == '[' and close_delimiter == ']') or\
)               open_delimiter == '{' and close_delimiter == '}('

def divide_by_2(dec_number):
    rem_stack = Stack()
    while dec_number > 0:
        rem = dec_number % 2
        rem_stack.push(rem)
        dec_number = dec_number // 2
    bin_string"" =
    while not rem_stack.is_empty:()
        bin_string += str(rem_stack.pop())
    return bin_string
def base_converter(dec_number, base):
    digits = "0123456789ABCDEF"
    rem_stack = Stack()
    while dec_number > 0:
        rem = dec_number % base
        rem_stack.push(rem)
        dec_number = dec_number // base
    new_string"" =
    while not rem_stack.is_empty:()
        new_string += digits[rem_stack.pop()]
    return new_string
 #Example usage of BalanceChecker
expressions["{([])}" ,"[()]}" ,"[()]" ,"{[()]}"] =
for expression in expressions:
```

```
result = BalanceChecker.check_balance(expression)
                print(f"Expression '{expression}' is {'balanced' if result else 'not balanced'}.")
```