DATA STRUCTURES

Stack Data Structure-Applications

By
Zainab Malik

Content

- Application of Stack
  - Symbol Balancing
  - Polish Notations
  - Evaluation of Postfix Expression
  - Conversions of Polish Notations
    - Infix-to-Postfix Conversion
    - Infix-to-Prefix Conversion
    - Postfix-to-Infix Conversion
    - Prefix-to-Infix Conversion
    - Postfix-to-Prefix Conversion
    - Prefix-to-Postfix Conversion
  - Other Applications

Stack Application-Symbol Balancing

1. Create an empty stack of char
2. Read input text char by char till the end of input
   - 2.1. If char is an opening symbol, push it on the stack
   - 2.2. If char is a closing symbol
     - 2.2.1. If stack is empty, report an error ("Opening symbol missing")
     - 2.2.2. If stack is non-empty, pop a char from stack and match it with input char. If both characters do not match then report an error ("Symbol mismatch")
3. At the end of input, if stack is non-empty then report an error ("Closing Symbol missing")

Stack Application-Symbol Balancing

0	1	2	3	4	5
(	[	(	)	]	)

Valid String or symbols are balanced

Stack Application-Symbol Balancing

0	1	2	3	4	5
(	[	(	]	)	)

Invalid String or symbols are mismatched

Stack Application-Symbol Balancing

0	1	2	3	4
(	[	)	]	)

Invalid String or Opening symbol is missing

Stack Application-Symbol Balancing

0	1
)	]

Invalid String or Opening symbol is missing

Polish Notations

- Infix Expression (a+b)
- Prefix Expression (+ab)
- Postfix Expression (ab+)

- Computer System uses postfix notation to evaluate an expression.

Stack Application-Postfix Evaluation

1. Create an empty stack of type double
2. Read input postfix expression char by char till the end of input
   - 2.1. If char is an operand, convert it into its double equivalent and then push it on the stack
   - 2.2. If char is an operator then pop two elements from the stack, perform the operation and push the result back on the stack.
3. At the end of input, Pop the final result and return it

Infix-to-postfix manually

9 – 5 / 3 + 5 * 2

9 - 53/ + 5 * 2

953/- + 5 * 2

953/- + 52*

953/-52*+

Postfix Evaluation: 953/-52*+

Scanned Character	Stack	Operation	Rule
9	9		2.1
5	9,5		2.1
3	9,5,3		2.1
/	9	5/3=1.67	2.2
	9 ,1.67		
-		9 - 1.67=7.33	2.2
	7.33		
5	7.33, 5		2.1
2	7.33, 5, 2		2.1
*	7.33	5*2=10	2.2
	7.33,10		
+		7.33+10=17.33	2.2
	17.33		
			3
Output	**17.33**		

Stack Application-(infix-to-postfix conversion)

1.Create an empty stack of type char

2.Read input expression char by char till the end of input

- **2.1**.Operand: display it
- **2.2**.Opening Parenthesis: push
- **2.3**.Operator:
  - **2.3.1.** If stack is empty then push it
  - **2.3.2.** If stack is non-empty then pop characters from stack and display them until we find an operator of lower precedence or an opening parenthesis or stack become empty. When popping is done push the current operator on to the stack.
- **2.4.**Closing Parenthesis: Pop operators from stack and display them until we pop an opening parenthesis which will be popped but not displayed.

3.At the end of input, if stack is non-empty, pop operators from stack and display them until the stack becomes empty.

Infix-to-postfix conversion: a + b * c + (d * e + f) * g

Scanned Char	Stack	Output Expression	Rule
a		a	2.1
+	+	a	2.3.1
b	+	ab	2.1
*	+ +*	ab	2.3.2
c	+*	abc	2.1
+	+* + Emp +	Abc* Abc*+	2.3.2
(	+ +(	Abc*+	2.2
d	+(	Abc*+d	2.1
*	+(+(*	Abc*+d	2.3.2
e	+(*	Abc*+de	2.1

Infix-to-postfix conversion: a + b * c + (d * e + f) * g			
Scanned Char	Stack	Output Expression	Rule
+	+(* +(+(+	Abc*+de Abc*+de*	2.3.2
f	+(+	Abc*+de*f	2.1
)	+(+ +(+	Abc*+de*f Abc*+de*f+	2.4
*	+ +*	Abc*+de*f+	2.3.2
g	+*	Abc*+de*f+g	2.1
Input End	+* + emp	Abc*+de*f+g*, Abc*+de*f+g Abc*+de*f+g*, Abc*+de*f+g* Abc*+de*f+g*, Abc*+de*f+g*+	3

Stack Application-(infix-to-prefix conversion)

1.Create an empty stack of type char
2.Create an output string
3.**Reverse the input expression**
4.Read the reversed input expression char by char till the end of input
- **4.1.**Operand: add to a output string
- **4.2.Closing Parenthesis**: push
- **4.3.**Operator:
  - **4.3.1**.If stack is empty then push it
  - **4.3.2**.If stack is non-empty then pop characters from stack and add them to output string until we find an operator with **lower or equal precedence** or an **closing parenthesis** or stack become empty. When popping is done push the current operator on to the stack.
- **4.4.Opening** Parenthesis: Pop operators from stack and add them to output string until we pop a **closing** parenthesis which will be popped but not be added to output string.

5. At the end of input, if stack is non-empty, pop operators from stack and add them to output string until the stack becomes empty.
6. Reverse the output string and display it.

Infix-to-pretfix conversion: a + b * c + (d * e + f) * g			
Scanned Character	**Stack**	**Output Expression**	**Rule**
Reverse infix expression: g *) f + e * d (+ c * b + a			**3**
g		g	4.1
*	*	g	4.3.1
)	*)	g	4.2
f	*)	gf	4.1
+	*) *)+	gf	4.3.2
e	*)+	gfe	4.1
*	*)+ *)+*	gfe	4.3.2
d	*)+*	gfed	4.1
(	*)+* *)+ *) *	Gfed Gfed* Gfed*+	4.4
+	* Empty +	gfed*+ gfed*+*	4.3.2

Infix-to-pretfix conversion: a + b * c + (d * e + f) * g			
Scanned Character	**Stack**	**Output Expression**	**Rule**
Reverse infix expression: g *) f + e * d (+ c * b + a			**3**
c	**+**	**gfed*+*c**	**4.1**
*****	**+** **+***	**gfed*+*c**	**4.3.2**
b	**+***	**gfed*+*cb**	**4.1**
+	**+*** **+** **++**	**gfed*+*cb** **gfed*+*cb***	**4.3.2**
a	**++**	**gfed*+*cb*a**	**4.1**
Input End		**gfed*+*cb*a++**	**5**
Reverse Expression:		**++a*bc*+*defg**	**6**

Stack Application-(postfix-to-infix conversion)

1. Create an empty stack of type string
2. Read the input postfix expression char by char till the end of input
    - 2.1. If the char is an operand: push it on the stack
    - 2.2. It the char is operator: Pop two operands from stack from an infix sub-expression and push the sub-expression back on the stack.
3. At the end of the input pop the resultant infix expression from the stack and display it

Postfix-to-infix conversion: $A\ B\ C * +\ D\ F\ /\ -\ E +$			
Scanned Character	**Stack**	**Output Expression**	**Rule**
A	A		2.1
B	A B		2.1
C	A B C		2.1
*	A (B * C)		2.2
+	(A + (B * C))		2.2
D	(A + (B * C)) D		2.1
F	(A + (B * C)) D F		2.1
/	(A + (B * C)) (D/F)		2.2
-	((A + (B * C)) – (D/F))		2.2
E	((A + (B * C)) –(D/F)) E		2.1
+	(((A + (B * C)) –(D/F)) + E)		2.2
Output		(((A + (B * C)) –(D/F)) + E)	3

Stack Application-(prefix-to-infix conversion)

1. Create an empty stack of type string
2. Reverse the input expression
3. Read the reversed input prefix expression char by char till the end of input
    - 3.1. If the char is an operand: push it on the stack
    - 3.2. It the char is operator: Pop two operands from stack from an infix sub-expression)(and push the sub-expression back on the stack.
4. At the end of the input pop the resultant expression from the stack, reverse and display it.

Prefix-to-infix conversion: $+ - + A * B C / D F$E			
Scanned Character	**Stack**	**Output Expression**	**Rule**
Reverse Prefix Expression: E F D / C B * A + - +			**2**
E	E		3.1
F	E F		3.1
D	E F D		3.1
/	E)F / D (		3.2
C	E)F / D (C		3.1
B	E)F / D (C B		3.1
*	E)F / D ( )C *B(		3.2
A	E)F / D ( )C *B(A		3.1
+	E)F / D ( ))C *B(+ A(		3.2
-	E))F / D (-))C *B(+ A((		3.2
+	)E +))F / D (-))C *B(+ A(((		3.2
Output		(((A+(B *C))-(D/F))+E)	4

Stack Application-(prefix-to-postfix conversion)

1. Apply prefix-to-infix conversion
2. Apply infix-to-postfix conversion

Stack Application-(postfix-to-pretfix conversion)

1. Apply postfix-to-infix conversion
2. Apply infix-to-prefix conversion

Other Applications

- Any modern computer environment uses a stack as the primary memory management model for a running program.
  - Manages the nested features of any programming language
  - Manages the recursive calls of a recursive function etc.
- Stack data structures are used in backtracking problems.

Thank You