Data Structures

Lecture Five

Second Stage

First Course - 2024-2025

Second Stage

First Course - 2024-2025

ALI MUWAFAQ SHABAN

Master of Computer Engineering

Data Structures

Stack's Application

Following are some of the important applications of a Stack data structure:

1. Stacks can be used for expression evaluation.

2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.
4. Stacks can be used for Memory Management.
5. Stack data structures are used in backtracking problems.

Implementation of stacks

1- Check for balancing of parenthesis.

2- Convert infix expression to postfix.

3- Evaluate postfix expression.

4- Use of Stack in Function calls.

5- Check for palindrome strings

Expression Evaluation

Two standard examples that illustrate the utility of the stack expression involve the evaluation of an arithmetic expression. Normally we are used to writing arithmetic expressions in what is termed infix form. Here a binary operator is written between two arguments, as in 2 + 3 * 7. Precedence rules are used to determine which operations should be performed first, for example multiplication typically takes precedence over addition. Associativity rules apply when two operations of the same precedence occur one right after the other, as in 6 – 3 – 2. For addition, we normally perform the left most operation first, yielding in this case 3, and then the second operation, which yields the final result 1. If instead the associativity rule specified right to left evaluation we would have first performed the calculation 3 – 2, yielding 1, and then subtracted this from 6, yielding the final value 5. Parenthesis can be used to override either precedence or associativity rules when desired. For example, we could explicitly have written 6 – (3 – 2).

The evaluation of infix expressions is not always easy, and so an alternative notion, termed postfix notation, is sometimes employed. In postfix notation the operator is written after the operands. The following are some examples:

Infix	2 + 3	2 + 3 * 4	(2 + 3) * 4	2 + 3 + 4	2 - (3 – 4)
Postfix	2 3 +	2 3 4 * +	2 3 + 4 *	2 3 + 4 +	2 3 4 - -

The following table briefly tries to show the difference in all three notations:

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Concert Infix notation to Postfix Notation algorithm using two stacks:

1. Initialize two empty stacks, (st1) to store operands, (st2) to store the operators of the mathematic operations.
   - Read the tokens from the infix string one at a time from left to right
   - If the token is operands then push it in st1.
   - If the token is lift parentheses then push it in st2.
   - If the token is right parentheses then pop all the operators from st2 and push it in st1 .
   - Repeat this step until lift parentheses is encountered, then remove and ignore the lift parentheses and right parentheses.
   - If the token is operator then pop all operators (if you found) which have priority higher than or equal the priority of the current operator from st2 and push it in st1, then push the current operators in st2.
   - Else
   - Push the current operator in st2
2. After all the characters are read and the st2 is not empty then Pop the stack and add the tokens to the st1
3. Return the Postfix st1.

Example: Convert the following infix expression to postfix using one stack:
A + (B /C) #

Ch	Opstack	Postfix (output string)	Commentary
A		A	Add ch to postfix
+	+	A	Push ch to opstack
(	+(	A	Push ch to opstack
B	+(	AB	Add ch to postfix

/	+(/	AB	Push ch to opstack
C	+(/	ABC	Add ch to postfix
)	+	ABC/	Pop and add to postfix until (is reached.
# or end of expression		ABC/+	Pop and add to postfix until the opstack is empty.

Example: Convert the following infix expression to postfix using one stack;
((A – (B + C)) * D) / (E +F) #

Ch	Opstack	Postfix(Output string)	Commentary
(	(		Push ch to opstack
(	((		Push ch to opstack
A	((	A	Add ch to postfix
_	((-	A	Push ch to opstack
(	((- (	A	Push ch to opstack
B	((- (	AB	Add ch to postfix
+	((- (+	AB	Push ch to opstack
C	((- (+	ABC	Add ch to postfix
)	((-	ABC+	Pop and add to postfix until (is reached
)	(	ABC+-	Pop and add to postfix until (is reached
*	(*	ABC+-	Push ch to opstack
D	(*	A B C + -D	Add ch to postfix
)	(*	ABC+ - D*	Pop and add to postfix until (is reached
/	/	A B C +-D *	Push ch to opstack
(	/(	A B C + -D *	Push ch to opstack
E	/ (	A B C+-D*E	Add ch to postfix
+	/ (+	ABC+-D*E	Push ch to opstack
F	/ (+	ABC+-D*E F	Add ch to postfix
)	/	A B C + - D * E F +	Pop and add to postfix until (is reached
# or end of expression		A B C + - D * E F +/	Pop and add to postfix until the opstack is empty.

Example: Convert the following infix expression to postfix using two stack: a-b*(c+d)/(e-f) #

Ch	Stack2	Stack1	Commentary
a		a	Push ch to stack1
-	-	a	Push ch to stack2
b	-	ab	Push ch to stack1
*	-*	ab	Push ch to stack2
(	-*(	ab	Push ch to stack2

c	-*(	abc	Push ch to stack1
+	-*(+	abc	Push ch to stack2
d	-*(+	Abcd	Push ch to stack1
)	-*	abcd+	Pop ch from stack2 and push them into stack1 until (is reached.
/	-/	abcd+*	Push ch to stack2
(	-/(	abcd+*	Push ch to stack2
e	-/(	abcd+*e	Push ch to stack1
-	-/(-	abcd+*e	Push ch to stack2
f	-/(-	abcd+* ef	Push ch to stack1
)	-/	abcd+*ef-	Pop ch from stack2 and push them into stack1 until (is reached.
# or end of expression		abcd+*ef-/-	Pop ch from stack2 and push them into stack1 until the stack2 is empty.

Homework: Let us illustrate the procedure InfixToPostfix with the following arithmetic expression: Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

Check for balancing of parenthesis.

A stack is useful here because we know that when a closing symbol such as) is seen , it matches the most recently seen unclosed (. therefore , by placing an opening symbol on a stack, we can easily determine whether a closing symbol makes sense. Specifically , we have the following algorithm .

1- Make an empty stack.

2- Read symbols until the end of the file .

a- If the symbol is an opening symbol , push it onto the stack.

b- If it is a closing symbol, do the following:

i- If the stack is empty , report an error

ii- Otherwise ,pop the stack

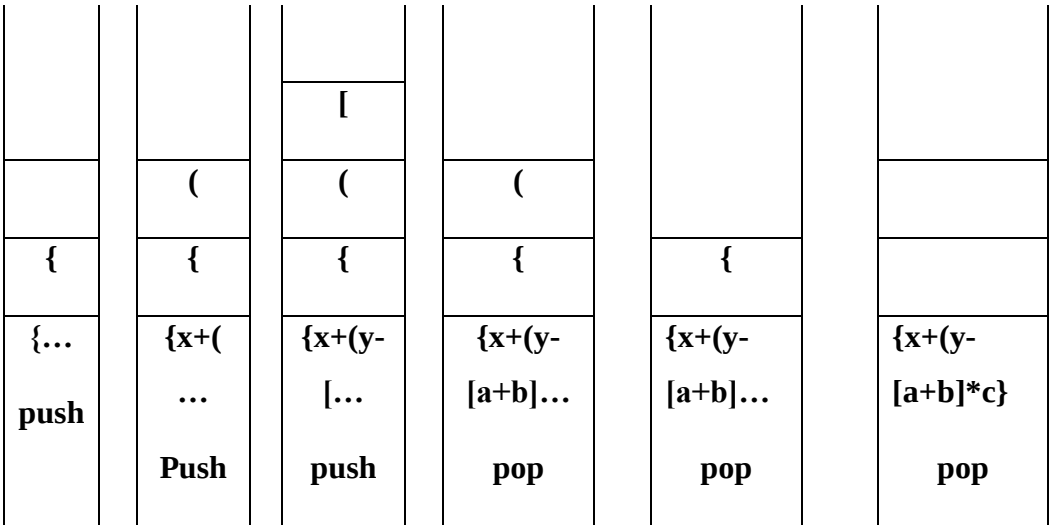
- if the symbol popped is not the corresponding opening symbol, report an error

- else we continue.

3- At the end of the file , if the stack is not empty , report an error.

figure below , shows the state of the stack after reading in parts of the string :

{X+(Y-[a+b])*c}



Evaluate postfix expression(complier)

We consider evaluating postfix expression using one stack . we will use an algorithm that **using one stack** The description of the algorithm is as follows:

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation:

Step 1: scan the expression from left to right

Step 2: if it is an operand push it to stack

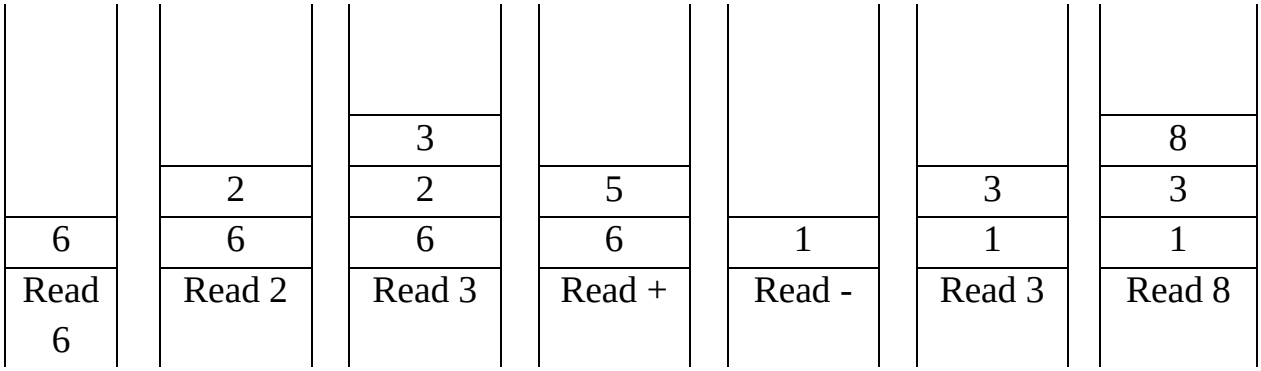
Step 3: if it is an operator pull operand from stack and perform operation

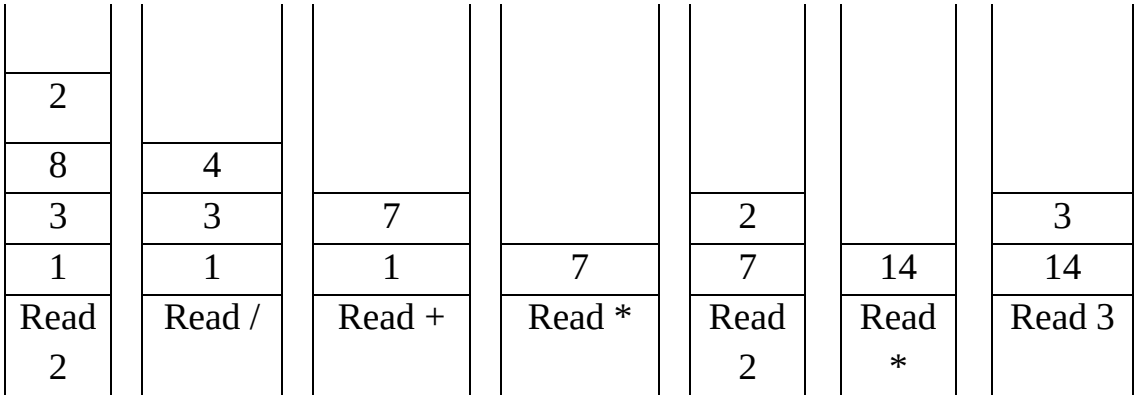
Step 4: store the output of step 3, back to stack

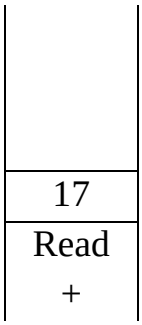
Step 5: scan the expression until all operands are consumed

Step 6: pop the stack and perform operation

an example , Execute the following postfix notation using one stack:
6 2 3 + - 3 8 2 / + * 2 * 3 +







Example: Execute the following infix notation using the two stack :
3 + 7 * 2 - 6

Ch	Stack1(operation)	Stack2(operand)	Commentary

3		3	Push ch to stack2
+	+	3	Push ch to stack1
7	+	7 3	push ch to stack2
*	* +	7 3	Push ch to stack1
2	* +	2 7 3	push ch to stack2
-	+	14 3	Pop stack1 (*) Pop stack2(2) Pop stack2 (7) 7*2= 14 push 14 to stack2
	-	17	Pop stack1 (+) Pop stack2 (14) Pop stack2 (3) 3+14 = 17 push 17 into stack2 Push ch to stack1 (-)
6	-	6 17	Push ch to stack2
# or end of expression	-	11	Pop stack1 (-) Pop stack2 (6) Pop stack2 (17) 17-6= 11 push 11 to stack2

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed. To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Memory Management

The assignment of memory takes place in contiguous memory blocks. We call this stack memory allocation because the assignment takes place in the function call stack. The size of the memory to be allocated is known to the compiler.

When a function is called, its variables get memory allocated on the stack. When the function call is completed, the memory for the variables is released. All this happens with the help of some predefined routines in the compiler. The user does not have to worry about memory allocation and release of stack variables.

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.