CS-218 DATA STRUCTURES AND ALGORITHMS

LECTURE 14

BY UROOJ AINUDDIN

STACKS

BOOK 1 CHAPTER 7

IN THE LAST LECTURE…

- We were introduced to stacks.
- We implemented stacks using the Python list and the singly linked list.
- We listed several problems that use stacks for solution.
- We learned about different formats for mathematical expressions.
- We learned how to convert between formats.
- We learned how to evaluate postfix expressions.

THE PROBLEM WITH INFIX EXPRESSIONS

- We work with infix expressions on a regular basis and find them rather easy to evaluate. But the task is more difficult for a computer program.
- A mathematical expression is represented as a string in the computer.
- Consider the expression $\overrightarrow{A \times B + C / D}$. The operator is placed between the two operands. This kind of expression is called an **infix** expression.
- When evaluating this expression stored as a string and scanning one character at a time from left to right, how does the computer know the addition has to wait until after the division?
- Since the computer does a single left-to-right scan of the expression string, it requires a representation in which the operators appear in the order they should be performed.

WHY MOVE TO POSTFIX NOTATION?

In infix notation in which the same expression can bear multiple meanings, with the use of parentheses. $3 + 4 \times 5 \neq (3 + 4) \times 5$

In the postfix and prefix notations, each expression is unique.

Postfix notation for $3 + 4 \times 5$ is $3,4,5,\times,+$, while that for $(3 + 4) \times 5$ is $3,4,+,5,\times$.

The postfix and prefix notations do not use parentheses to override the order of precedence.

In the postfix notation, the scanning program gathers the operands before the operation, which is the norm for all procedures.

ALGORITHM FOR EVALUATION OF POSTFIX EXPRESSION

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is an operand, push it onto the stack. Otherwise, it is an operator. Do the following:
      i. Pop from stack and call this B.
      ii. Pop from stack and call this A.
      iii. Apply the operator to these two values. Result=A token B.
      iv. Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack. It must be the only value in the stack.

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
    a. Get the next token (character) of the expression.
    b. If the token is an operand, push it onto the stack. Otherwise, it is an operator. Do the following:
        i. Pop from stack and call this B.
        ii. Pop from stack and call this A.
        iii. Apply the operator to these two values. Result=A token B.
        iv. Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack. It must be the only value in the stack.

EVALUATE:

1,2,-,4,5,^,3,×,6,×,2,2,2,^,^,/,-

Token	Stack [→	Actions
1	1	
2	1, 2	
−	-1	B=2, A=1, R=1−2
4	-1, 4	
5	-1, 4, 5	
^	-1, 1024	B=5, A=4, R=4^5
3	-1, 1024, 3	
×	-1, 3072	B=3, A=1024, R=1024×3
6	-1, 3072, 6	
×	-1, 18432	B=6, A=3072, R=3072×6
2	-1, 18432, 2	
2	-1, 18432, 2, 2	

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
    a. Get the next token (character) of the expression.
    b. If the token is an operand, push it onto the stack. Otherwise, it is an operator. Do the following:
        i. Pop from stack and call this B.
        ii. Pop from stack and call this A.
        iii. Apply the operator to these two values. Result=A token B.
        iv. Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack. It must be the only value in the stack.

EVALUATE:

1,2,-,4,5,^,3,×,6,×,2,2,2,^,^,/,-

Token	Stack [→	Actions
2	-1, 18432, 2, 2, 2	
^	-1, 18432, 2, 4	B=2, A=2, R=2^2
^	-1, 18432, 16	B=4, A=2, R=2^4
/	-1, 1152	B=16, A=18432, R=18432/16
−	-1153	B=1152, A=-1, R=-1−1152

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is an operand, push it onto the stack. Otherwise, it is an operator. Do the following:
      i. Pop from stack and call this B.
      ii. Pop from stack and call this A.
      iii. Apply the operator to these two values. Result=A token B.
      iv. Push the resulting value back onto the stack.
3. When the end of the expression is encountered, its value is on top of the stack. It must be the only value in the stack.

- In step 1, the __init__ function is called for the class that implements the stack. It takes $O(1)$ time.
- Step 2 has a loop that runs as many times as the number of tokens.
- Let the number of tokens be n.
- If the token is an operand, we push it in $O(1)$ time.
- If the token is an operator, we pop twice, compute and push. This takes $O(1)$ time.
- The body of the loop in step 2 takes $O(1)$ time in all cases.
- Step 2 takes $O(n)$ time.
- Step 3 encompasses a pop operation. It takes $O(1)$ time.

$O(n)$

SPACE COMPLEXITY OF AN ALGORITHM

- Space complexity is asymptotic analysis of the space used by an algorithm to solve a problem.
- We consider how an algorithm’s memory requirements increase as the input size increases.
- We investigate the additional variables and data structures that an algorithm uses to solve a problem.
- The space occupied by the input to the algorithm is NOT considered in space complexity analysis.

SPACE COMPLEXITY OF THE ALGORITHM FOR EVALUATION OF POSTFIX EXPRESSION

- In any postfix expression composed of binary operators, there are x operators and $x + 1$ operands.
- Let the length of the postfix expression be n.

$$n = x + (x + 1)$$

$$x = (n - 1)/2$$

- We keep operands on the stack in the algorithm for postfix evaluation.
- The maximum size of the stack is $x + 1$.

$$x + 1 = (n + 1)/2$$

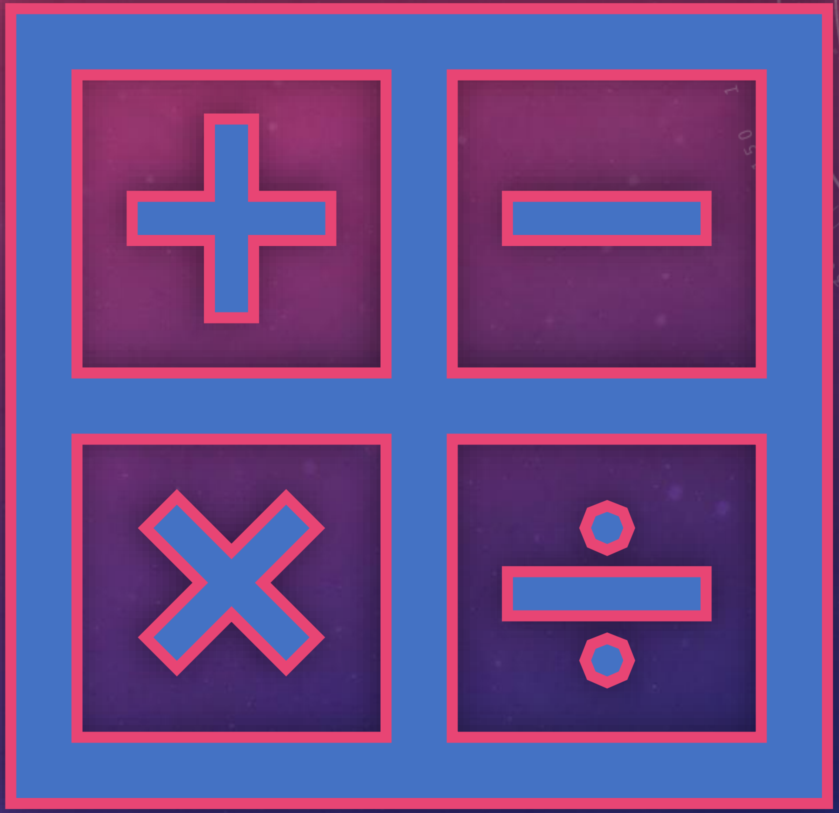
$O(n)$

ALGORITHM FOR CONVERSION FROM INFIX TO POSTFIX EXPRESSION

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is:
      i. a left parenthesis: Push it onto the stack.
      ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
      iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
      iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

PRECEDENCE LEVELS OF OPERATORS

- Highest: ^
- ×, /
- Lowest: +, −



1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is:
      i. a left parenthesis: Push it onto the stack.
      ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
      iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
      iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

CONVERT:

2,^,3,×,7,+,(,5,-,4,),×,15,/,3,-,8

Token	Stack [→	Output
2	Empty	2
^	^	
3	^	2, 3
×	×	2, 3, ^
7		2, 3, ^, 7
+	+	2, 3, ^, 7, ×
(	+, (	
5		2, 3, ^, 7, ×, 5
–	+, (, –	
4		2, 3, ^, 7, ×, 5, 4
)	+	2, 3, ^, 7, ×, 5, 4, –
×	+, ×	

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is:
      i. a left parenthesis: Push it onto the stack.
      ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
      iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
      iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

CONVERT:

2,^,3,×,7,+,(,5,-,4,),×,15,/,3,-,8

Token	Stack [→	Output
15	+, ×	2, 3, ^, 7, ×, 5, 4, −, 15
/	+, /	2, 3, ^, 7, ×, 5, 4, −, 15, ×
3		2, 3, ^, 7, ×, 5, 4, −, 15, ×, 3
−	−	2, 3, ^, 7, ×, 5, 4, −, 15, ×, 3, /, +
8		2, 3, ^, 7, ×, 5, 4, −, 15, ×, 3, /, +, 8
	Empty	2, 3, ^, 7, ×, 5, 4, −, 15, ×, 3, /, +, 8, −

Instead of a character-by-character display, we can append each character that we want to display to a Python list, and then print the list out at the very end.

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
    a. Get the next token (character) of the expression.
    b. If the token is:
        i. a left parenthesis: Push it onto the stack.
        ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
        iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
        iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

- In step 1, the __init__ function is called for the class that implements the stack. It takes $O(1)$ time.
- Step 2 has a loop that runs as many times as the number of tokens.
- Let the number of tokens be n.
- If the token is a left parenthesis, we push it in $O(1)$ time.
- If the token is a right parenthesis, we pop several times, say x. We need to know what the maximum value of x is, to determine worst case complexity.
- If the token is an operator, we compare it with the top stack element several times, say y, before we push the token onto the stack.

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
    a. Get the next token (character) of the expression.
    b. If the token is:
        i. a left parenthesis: Push it onto the stack.
        ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
        iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
        iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

- We can only push a higher priority token over a lower priority token.
- There are three levels of precedence for operators. This means we can have three operators of different priority levels in the stack CONSECUTIVELY. An example is $-, /, \wedge$.
- We can have more than three operators in the stack, like $+, \times, (, -, /, \wedge$.
- If we can push 3 operators in the stack consecutively, the maximum number of comparisons, y, is 4, 3 comparisons with operators and the last with (.
- The maximum number of pops when we encounter a right parenthesis, x, is 4, as we will pop 3 operators and (.

1. Initialize an empty stack.
2. Repeat the following until the expression ends:
   a. Get the next token (character) of the expression.
   b. If the token is:
      i. a left parenthesis: Push it onto the stack.
      ii. a right parenthesis: Pop and display stack elements until a left parenthesis is encountered, but do not display the left parenthesis.
      iii. an operator: If the stack is empty or token has a higher priority than the top stack element, push token onto the stack. (A left parenthesis in the stack has lower priority than all operators.) Otherwise pop and display the top stack element; then repeat the comparison of token with the new top stack item.
      iv. an operand: Display it.
3. When the end of the expression is encountered, pop and display stack items until the stack is empty.

- If the token is an operand, it will be displayed in $O(1)$ time.
- In the worst case, the token will be compared four times before being pushed, no matter what the value of n is.
- In the worst case, four elements will have to be popped when a right parenthesis is seen, no matter what the value of n is.
- The body of the loop in step 2 takes $O(1)$ time in all cases.
- Step 2 takes $O(n)$ time.
- Step 3 encompasses four pop operations in the worst case, no matter what the value of n. It takes $O(1)$ time.

$O(n)$

SPACE COMPLEXITY OF THE ALGORITHM FOR CONVERSION FROM INFIX TO POSTFIX EXPRESSION

- In any intix expression composed of binary operators, there are x operators, $x + 1$ operands, b left parentheses and b right parentheses.
- Let the length of the postfix expression be n.

$$n = x + (x + 1) + b + b$$

$$x + b = (n - 1)/2$$

$O(n)$

- We keep operators and left parentheses on the stack in the algorithm for conversion from infix to postfix expression.
- The maximum size of the stack is $x + b$.

HOMEWORK

1. Convert the following infix expression to postfix using a stack:

$$8 + 9 - (5 - 2) \times ((1 - 7) + 4)/2$$

2. Evaluate the given postfix expression using a stack:

$$4, 12, 11, 9, \times, 9, -, 6, +, 12, /, +, -$$

HOMEWORK

3. Implement the algorithm for postfix evaluation in Python.
4. Implement the algorithm for conversion from infix to postfix expressions in Python.

SO WHAT DID WE LEARN TODAY?

We discussed the reasons behind preference of postfix expressions for computing machines.

We traced through the algorithm for postfix evaluation.

We traced through the algorithm for conversion from infix to postfix expression.

We investigated both time and space complexities of the algorithms discussed.

THINGS TO DO

Read the book!

Note your questions and put them up in the relevant online session.

Email suggestions on content or quality of this lecture at uroojain@neduet.edu.pk