More About Stacks: Stack Applications

Quick Introduction

- Stacks are linear lists.
- All deletions and insertions occur at one end of the stack known as the TOP.
- Data going into the stack first, leaves out last.
- Stacks are also known as LIFO data structures (**L**ast-**I**n, **F**irst-**O**ut).

Basic Stack Operations

- push – Adds an item to the top of a stack.
- pop – Removes an item from the top of the stack and returns it to the user.
- stack top (top, peek) – Copies the top item of the stack and returns it to the user; the item is not removed, hence the stack is not altered.

Additional Notes

- Stacks structures are usually implemented using arrays or linked lists.
- For both implementations, the running time is O(n).
- We will be examining common Stack Applications.

Stack Applications

- Reversing Data: We can use stacks to reverse data.
  (example: files, strings)
  Very useful for finding palindromes.

Consider the following pseudocode:

1) read (data)
2) loop (data not EOF and stack not full)
    1) push (data)
    2) read (data)
3) Loop (while stack notEmpty)
    1) pop (data)
    2) print (data)

Stack Applications

- Converting Decimal to Binary: Consider the following pseudocode

```
1) Read (number)
2) Loop (number > 0)
      1) digit = number modulo 2
      2) print (digit)
      3) number = number / 2
// from Data Structures by Gilbert and Forouzan
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.) To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Stack Applications

- Postponement: Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a + b (what we use in grammar school)
- Postfix: a b +

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Infix to Postfix Conversion

- Rules:
  - Operands immediately go directly to output
  - Operators are pushed into the stack (including parenthesis)
    - Check to see if stack top operator is less than current operator
    - If the top operator is less than, push the current operator onto stack
    - If the top operator is greater than the current, pop top operator and push onto stack, push current operator onto stack
    - Priority 2: * /
    - Priority 1: + -
    - Priority 0: (

    If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

Infix to Postfix Example

A + B * C - D / E

Infix	Stack(bot->top)	Postfix
a) A + B * C - D / E		
b) + B * C - D / E		A
c) B * C - D / E	+	A
d) * C - D / E	+	A B
e) C - D / E	+ *	A B
f) - D / E	+ *	A B C
g) D / E	+ -	A B C *
h) / E	+ -	A B C * D
i) E	+ - /	A B C * D
j)	+ - /	A B C * D E
k)		A B C * D E / - +

Infix to Postfix Example #2

A * B - (C + D) + E

	Infix	Stack(bot->top)	Postfix
a)	A * B - (C - D) + E	empty	empty
b)	* B - (C + D) + E	empty	A
c)	B - (C + D) + E	*	A
d)	- (C + D) + E	*	A B
e)	- (C + D) + E	empty	A B *
f)	(C + D) + E	-	A B *
g)	C + D) + E	- (	A B *
h)	+ D) + E	- (	A B * C
i)	D) + E	- (+	A B * C
j)	) + E	- (+	A B * C D
k)	+ E	-	A B * C D +
l)	+ E	empty	A B * C D + -
m)	E	+	A B * C D + -
n)		+	A B * C D + - E
o)		empty	A B * C D + - E +

Postfix Evaulation

Operand: push
Operator: pop 2 operands, do the math, pop result back onto stack

1 2 3 + *

Postfix	Stack(bot -> top)
a) 1 2 3 + *	
b) 2 3 + *	1
c) 3 + *	1 2
d) + *	1 2 3
e) *	1 5 // 5 from 2 + 3
f)	5 // 5 from 1 * 5

Backtracking

- Stacks can be used to backtrack to achieve certain goals.
- Usually, we set up backtrack tokens to indicate a backtrack opportunity.