Data Structures in Java

Lecture 6: Stacks.

9/28/2015

Daniel Bauer
Reminder: Recitation Session tonight

• Thursday session permanently moved to Monday.

• 7:35 - Schermerhorn 614

• This week: Homework 1 review.

Homework

• Thank you for submitting homework 1!

• Homework 2 out tonight.
The Stack ADT

- A Stack $S$ is a sequence of $N$ objects $A_0, A_1, A_2, ..., A_{N-1}$ with three operations:
  
  - `void push(x)` - append element $x$ to the end (on “top”) of $S$.
  
  - `Object top() / peek()` = returns the last element of $S$.
  
  - `Object pop()` - remove and return the last element from $S$.

- Stacks are also known as Last In First Out (LIFO) storage.
The Stack ADT

• A Stack $S$ is a sequence of $N$ objects $A_0, A_1, A_2, \ldots, A_{N-1}$ with three operations:
  • void push(x) - append element $x$ to the end (on “top”) of $S$.
  • Object top() / peek() = returns the last element of $S$.
  • Object pop() - remove and return the last element from $S$.
• Stacks are also known as **Last In First Out (LIFO)** storage.
Stack Example

Top → 5
Stack Example

push(42)
Stack Example

push(42) push(23)

Top 23
     42
     5
Stack Example

push(42)  push(23)

Top  23  top() → 23
     42
     5
Stack Example

push(42)  push(23)  push(3)

Top → 3
23
42
5

top() → 23
Stack Example

push(42)  push(23)  push(3)

Top  →  [23
        [42
        [5

pop() → 3

top() → 23
Implementing Stacks

• Think of a Stack as a specialized List:
  • push: Inserts only allowed at the end of the list.
  • pop: Remove only allowed at the end of the list.
• Can implement Stack using any List implementation.
Implementing Stacks

- Think of a Stack as a specialized List:
  - push: Inserts only allowed at the end of the list.
  - pop: Remove only allowed at the end of the list.
- Can implement Stack using any List implementation.
- push and pop run in $O(1)$ time with ArrayList or LinkedList.
A Stack Interface

```java
interface Stack<T> {
    /* Push a new item x on top of the stack */
    public void push(T x);
    /* Remove and return the top item of the stack */
    public T pop();
    /* Return the top item of the stack without removing it */
    public T top();
}
```
Using MyLinkedList to implement Stack

```java
public class LinkedListStack<T> extends MyLinkedList<T>
    implements Stack<T> {

    public void push(T x) {
        add(size(), x);
    }

    public T pop() {
        return remove(size() - 1);
    }

    public T top() {
        return get(size() - 1);
    }
}
```
Direct Implementation Using an Array

(sample code)
Application: Balancing Symbols

- Compilers need to check for syntax errors.
- Need to make sure braces, brackets, parentheses are well nested.
- What’s wrong with this code:

```java
for(int i=0; i<=topOfStack; i++) {
    sb.append(theArray[i] + " ");
    sb.append("[]");
}
```
Balancing Symbols

```java
for(int i=0;i<=topOfStack;i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
}
push("("");
```
Balancing Symbols

```java
for(int i=0;i<=topOfStack;i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
}
push("(");
pop("(");
```
Balancing Symbols

```java
for(int i=0; i<=topOfStack; i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
}
push("(");
pop("(");
push("{"");
```
Balancing Symbols

```java
for(int i=0;i<=topOfStack;i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
}
```

```
push( "("
pop( "("
push( "{"
push( "("
```
Balancing Symbols

```java
for(int i=0; i<=topOfStack; i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
push( "(" );
pop( "(" );
push( "{" );
push( "(" );
push( "[" );
```

```
[  
(  
{  
```
Balancing Symbols

```
for(int i=0;i<=topOfStack;i++) {
    sb.append(theArray[i] + " ");
    sb.append("]");
}
```

push( "(" ) pop( "(" ) push( "{" )
push( "(" ) push( "[" )
Postfix Expressions

- How would you do the following calculation using a simple calculator:

\[ 5 + 27 / (2 \times 3) \]
Postfix Expressions

• How would you do the following calculation using a simple calculator:

\[ 5 + \frac{27}{(2 \times 3)} \]

\[ 2 \times 3 = 6 \]

remember intermediate results
Postfix Expressions

• How would you do the following calculation using a simple calculator:

\[ 5 + 27 / (2 * 3) \]

\[ 2 * 3 = 6 \]

\[ 27 / 6 = 4.5 \]
Postfix Expressions

• How would you do the following calculation using a simple calculator:

\[
5 + 27 \div (2 \times 3)
\]

\[
2 \times 3 = 6
\]

\[
27 \div 6 = 4.5
\]

\[
5 + 4.5 = 9.5
\]
Postfix Expressions

• How would you do the following calculation using a **simple** calculator:

\[5 + \frac{27}{(2 \times 3)}\]

\[2 \times 3 = 6\]

\[27 \div 6 = 4.5\]

\[5 + 4.5 = 9.5\]

5   27   2   3   *   /   +
Evaluating Postfix Expressions

5 + 27 / (2 * 3)

5 27 2 3 * / +

• for c in input
  • if c is an operand, push it
  • if c is an operator x:
    • pop the top 2 operands
      a₁ and a₂
    • push $a_3 = a_2 \times a_1$
    • pop the result.
Evaluating Postfix Expressions

5 + 27 / (2 * 3)

push(5)

• for c in input
  • if c is an operand, push it
  • if c is an operator x:
    • pop the top 2 operands \( a_1 \) and \( a_2 \)
    • push \( a_3 = a_2 \times a_1 \)
  • pop the result.
Evaluating Postfix Expressions

\[
5 + 27 / (2 \times 3)
\]

\[
\begin{array}{c}
5 & 27 & 2 & 3 & \times & / & + \\
\end{array}
\]

push(27)

- for \( c \) in input
- if \( c \) is an operand, push it
- if \( c \) is an operator \( x \):
  - pop the top 2 operands \( a_1 \) and \( a_2 \)
  - push \( a_3 = a_2 \times a_1 \)
- pop the result.

<table>
<thead>
<tr>
<th></th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

\[ 5 + 27 / (2 * 3) \]

- push(2)
- for \( c \) in input
  - if \( c \) is an operand, push it
  - if \( c \) is an operator \( x \):
    - pop the top 2 operands \( a_1 \) and \( a_2 \)
    - push \( a_3 = a_2 \times a_1 \)
  - pop the result.

\[
\begin{array}{c}
  2 \\
  27 \\
  5 \\
\end{array}
\]
Evaluating Postfix Expressions

\[ 5 + 27 / (2 \times 3) \]

\[ 5 \quad 27 \quad 2 \quad 3 \quad * \quad / \quad + \]

push(3)

- for \( c \) in input
  - if \( c \) is an operand, push it
  - if \( c \) is an operator \( x \):
    - pop the top 2 operands \( a_1 \) and \( a_2 \)
    - push \( a_3 = a_2 \times a_1 \)
    - pop the result.
Evaluating Postfix Expressions

5 + 27 / (2 * 3)

• for c in input
  • if c is an operand, push it
  • if c is an operator x:
    • pop the top 2 operands \( a_1 \) and \( a_2 \)
    • push \( a_3 = a_2 \times a_1 \)
  • pop the result.

pop() -> 3
pop() -> 2
push(2*3)

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

\[ 5 + \frac{27}{(2 \times 3)} \]

- For `c` in input:
  - If `c` is an operand, push it.
  - If `c` is an operator `x`:
    - Pop the top 2 operands `a_1` and `a_2`.
    - Push `a_3 = a_2 \times a_1`.
  - Pop the result.

\[
\begin{array}{c}
5 \\
27 \\
2 \\
3 \\
* \\
/ \\
+ \\
4.5 \\
5
\end{array}
\]

- Pop() -> 6
- Pop() -> 27
- Push(27/6)
Evaluating Postfix Expressions

\[ 5 + 27 / (2 * 3) \]

- for c in input
  - if c is an operand, push it
  - if c is an operator x:
    - pop the top 2 operands \( a_1 \) and \( a_2 \)
    - push \( a_3 = a_2 \times a_1 \)
  - pop the result.

pop() -> 4.5
pop() -> 5
push(5 + 4.5)

\[ 9 \]
Converting Infix to Postfix Notation

Input: \( a + b \times c + (d \times e + f) \times g \)

Output: 

Converting Infix to Postfix Notation

Input: $a + b \cdot c + (d \cdot e + f) \cdot g$

Output: $a\ b\ c\ \cdot\ +\ d\ e\ \cdot\ f\ +\ g\ \cdot\ +$
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: $a + b \times c + d$

Output: 

Order of Precedence:
$+$ = 1
$\times$ = 2
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \(a + b * c + d\)

Output: \(a\)

Order of Precedence:
+ = 1
* = 2
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input:  a + b * c + d
Output:  a

Order of Precedence:
+  = 1
*  = 2
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \( a + b * c + d \)

Output: \( a \ b \)

Order of Precedence:
\[ + = 1 \]
\[ * = 2 \]
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \[ a + b \times c + d \]

Output: \[ a \ b \]

Order of Precedence:

\[
\begin{array}{c|c}
+ & 1 \\
* & 2 \\
\end{array}
\]

* has higher priority than +,
so we want * in the output first. Keep pushing.
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: $a + b \times c + d$

Output: $a \ b \ c$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Order of Precedence:

$+$ = 1
$\times$ = 2
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \( a + b \times c + d \)

Output: \( a \ b \ c \)

Order of Precedence:

\[
\begin{array}{|c|}
\hline
* \\
\hline
+ \\
\hline
\end{array}
\]

+ = 1
* = 2
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: $a + b \times c + d$

Output: $a \ b \ c \ + \ d$

Order of Precedence:

$+$ = 1  
$*$ = 2

$+$ has lower priority than $*$, so we need to pop $*$ and write it to the output first.
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Order of Precedence:

\[ + = 1 \]
\[ * = 2 \]

Input: \( a + b * c + d \)

Output: \( a b c * + \)

Need to pop the first + too to keep sequential order.
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \( a + b \times c + d \)

Output: \( a \ b \ c \ + \ *

Order of Precedence:
+ = 1
* = 2

Then push the new +
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \( a + b * c + d \)

Output: \( a \ b \ c \ * \ + \ d \)

Order of Precedence:

\[ + = 1 \]
\[ * = 2 \]

Then push the new +
Converting Infix to Postfix Notation

Idea: keep lower-precedence operators on the stack.

Input: \( a + b \times c + d \)

Output: \( a \ b \ c \ \times \ + \ d \ + \)

Order of Precedence:
\[ + = 1 \]
\[ \times = 2 \]

Pop remaining stack elements.
Converting Infix to Postfix
Algorithm Sketch

- for c in input
  - if c is an operand: print c
  - if c is “+”, “*”:
    - while stack is not empty and priority(stack.top()) ≥ priority(c):
      - print stack.pop()
      - push c
  - while stack is not empty:
    - print stack.pop()
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching "(".

Input: \[ a \times ( b + c ) \times d + e \]

Output: \[ a \]

Order of Precedence:
+ = 1
* = 2
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \[ a \times ( b + c ) \times d + e \]

Output: \[ a \]

Order of Precedence:

\[ + = 1 \]
\[ \times = 2 \]
Converting Infix to Postfix
Dealing with ()

Idea: Put “(“ on stack. When “)” is seen, reduce stack until matching “(“.

Input: \[ a \ast (b + c) \ast d + e \]

Output: \[ a \]

Order of Precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>\ast</td>
<td>2</td>
</tr>
</tbody>
</table>
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \( a \times ( b + c ) \times d + e \)

Output: \( a \ b \)

| (  |
| *  |

Order of Precedence:

+ = 1
* = 2
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \[ a \times ( b + c ) \times d + e \]

Output: \[ a \ b \]

Order of Precedence:
\[
\begin{array}{c|c}
+ & 1 \\
( & 2 \\
\times & 2 \\
\end{array}
\]
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching "('.

Input: \( a \times (b + c) \times d + e \)

Output: \( a \ b \ c \)

Order of Precedence:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

\[ + = 1 \]
\[ \times = 2 \]
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching "(".

Input: \[ a \times (b + c) \times d + e \]

Output: \[ a \ b \ c \]

Order of Precedence:

\[
\begin{array}{c|c}
+ & 1 \\
( & \text{highest precedence} \\
* & 2 \\
\end{array}
\]
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \[ a \times ( b + c ) \times d + e \]

Output: \[ a \ b \ c \ \times \]

Order of Precedence:
\[
\begin{array}{c|c}
+ & 1 \\
\times & 2 \\
\end{array}
\]
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \[ a \ast ( b + c ) \ast d + e \]

Output: \[ a \ b \ c \ + \]

Order of Precedence:
\[ + = 1 \]
\[ \ast = 2 \]
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching "(".

Input: \( a \times (b + c) \times d + e \)

Output: \( a \ b \ c \ + \)

Order of Precedence:

\( + = 1 \)

\( \times = 2 \)
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: $a * ( b + c ) * d + e$

Output: $a \ b \ c \ + \ *$

Order of Precedence:

+ = 1
* = 2
Converting Infix to Postfix
Dealing with ()

Idea: Put “(“ on stack. When “)“ is seen, reduce stack until matching “(“.

Input:  \( a \ast ( b + c ) \ast d + e \)

Output:  \( a \ b \ c \ + \ast \)

Order of Precedence:
\( + = 1 \)
\( \ast = 2 \)
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching "(".

Input: \[ a \times (b + c) \times d + e \]

Output: \[ a \ b \ c \ + \ * \ d \]

Order of Precedence:
\[ + = 1 \]
\[ \times = 2 \]
Converting Infix to Postfix

Dealing with ()

Idea: Put “(“ on stack. When “)“ is seen, reduce stack until matching “(“.

Input:  \[ a \ast (b + c) \ast d + e \]

Output:  \[ a \ b \ c \ + \ast \ d \]

Order of Precedence:
\[ + = 1 \]
\[ \ast = 2 \]
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When ")" is seen, reduce stack until matching ")".

Input:  a * ( b + c ) * d + e
Output:  a b c + * d *

Order of Precedence:
+ = 1
* = 2
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: a * ( b + c ) * d + e

Output: a b c + * d *

Order of Precedence:
+ = 1
* = 2
Converting Infix to Postfix

Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \( a \times ( b + c ) \times d + e \)

Output: \( a \; b \; c \; + \; \times \; d \; \times \; e \)

Order of Precedence:
+ = 1
* = 2
Converting Infix to Postfix
Dealing with ()

Idea: Put "(" on stack. When "")" is seen, reduce stack until matching "(".

Input: \( a \times ( b + c ) \times d + e \)

Output: \( a \ b \ c \ + \ times \ d \times e + \)

Order of Precedence:

\[ + = 1 \]
\[ \times = 2 \]
Stacks in Hardware

• Stack as a memory abstraction:

  • CPU implement a hardware stack (use register to point to “top” location in main memory).

  • CPU operations push, pop will write/get value and increase or decrease register with a single byte code instruction.
Stack Machines

- Most modern computers are register machines. To compute 2+3:
  - `mov eax, 2`
  - `move ebx, 3`
  - `add eax, abx` which stores the result in eax

- In a Stack Machine:
  - `push 2`
  - `push 3`
  - `add` which stores the result back on the stack.

- Hardware stack machines are rare, but most virtual machines (including JVM) are stack machines.
What’s wrong with this program?

```java
public class Factorial {
    public static int factorial(int n) {
        return factorial(n - 1) * n;
    }

    public static void main(String[] args) {
        System.out.println(factorial(10));
    }
}
```

```
$ javac Factorial.java
$ java Factorial
Exception in thread "main" java.lang.StackOverflowError
    at InfiniteRecursion.factorial(Factorial.java:4)
    at InfiniteRecursion.factorial(Factorial.java:4)
    at InfiniteRecursion.factorial(Factorial.java:4)
    ...
Method Call Stacks

• Every function keeps an activation record on the method call stack.

• Represent current state of execution of this function.

• Includes instruction pointer, value of variables, parameters, intermediate results.

```java
public static int factorial(int n) {
    return factorial(n - 1) * n;
}
```

```java
public static void main(String[] args) {
    System.out.println(factorial(10));
}
```
Method Call Stacks (2)

- When a function is called
  - Execution of the current function is suspended.
  - A new activation record is pushed to the stack.
  - The new function is run.

```java
public static void main(String[] args) {
    System.out.println(factorial(10));
}
```

```java
public static int factorial(int n) {
    return factorial(n-1) * n;
}
```

Instruction pointer

```java
public static void main(String[] args) {
    String[] args = {};
    System.out.println(factorial(10));
}
```

Instruction pointer
Runaway Recursion

- Recursion will quickly grow the method call stack.
- Execution of the current function is suspended.
Fixing Runaway Recursion

• We forgot to add the base case:

```java
public static int factorial(int n) {
    if (n == 1)
        return 1;
    return factorial(n-1) * n;
}
```

• Still can get stack overflows for large $n$. 
Rewriting Recursion

• This is a stupid use for recursion.

```java
g public static int factorial(int n) {
    if (i == 1)
        return 1;
    return factorial(n-1) * n;
}
```  

• In general, any recursion can be removed, but this will often lead to unreadable code.

• But recursion is often more readable.

```java
g public static int factorial(int n) {
    int result = 1;
    for (i = 1; i<=n; i++)
        result = result * i;
}
```
Tail Recursion

- Compilers can detect and remove some types of recursion.

- A method is tail recursive if the last thing it does is call itself. Compilers can turn this into a loop.

```java
public static long factorial(long n) {
    return facRec(n, 1);
}

public static long facRec(long n, long result) {
    if (n==1)
        return result;
    else
        return facRec(n-1, result * n);
}
```