CS 228: Introduction to Data Structures
Lecture 22
Monday, March 9, 2015

The `PureStack` Interface

On Friday, we saw two simple implementations of stacks, one based on arrays, the other on linked lists. We can use either of them to implement the following simple stack interface, posted on Blackboard.

```
public interface PureStack<E>
{
  void push(E item);

  E pop();

  E peek();

  boolean isEmpty();

  int size();
}
```

Code for the array and linked list implementations is posted on Blackboard — see `ArrayBasedStack` and

`LinkedStack`, respectively. All operations take O(1) time (amortized, in the case of `push` for `ArrayBasedStack`).

Java Implementations of Stacks

It is easy to implement a stack as a Java `List`:

Stack Method	**`List` method**
`push()`	`add()`
`peek()`	`get(size()-1)`
`pop()`	`remove(size()-1)`
`isEmpty()`	`isEmpty()`
`size()`	`size()`

We can then use one of Java's implementations of the `List` interface. Two convenient implementations are the following.

- `ArrayList` implements `List` as a resizable array, so all the stack methods run in O(1) time. (To be precise, `add()` runs in O(1) amortized time.)

- `LinkedList` implements `List` as a doubly-linked list, so all stack methods take O(1) time.

Java has a legacy `Stack` class that implements all the required methods. However, Oracle recommends using the more modern `Deque` (for "doubly-ended queue) interface instead, as it provides "a more complete and consistent set of LIFO stack operations".

Stack Method	**Deque Method**
`push()`	`addFirst()`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

`Deque` has many other methods. We will revisit this interface when we study queues. `ArrayDeque` and `LinkedList` implement `Deque`.

Note. Although the Java implementations of stacks we have seen are fine for many applications, they do come loaded with features — e.g., `indexOf()` and `listIterator()` — that may be unnecessary for your specific application. In such cases, a simple implementation, like our PureStack and, e.g., `LinkedStack` might be more suitable.

Verifying Matched Parentheses

As a first application of stacks, consider the problem of verifying whether an string of parentheses is well-formed. For instance,

“`{[(){[]}]()}`”

is well-formed, but

“`{[]}[]]()}`”

is not.

More precisely, a string γ of parentheses is well-formed if either γ is empty or γ has the form

$$(\alpha)\beta \qquad [\alpha]\beta \quad \text{or} \qquad \{\alpha\}\beta$$

where α and β are themselves well-formed strings of parentheses. This kind of recursive definition lends itself naturally to a stack-based algorithm.

To check a `String` like `"{[(){[]}]()}"`, scan it character by character.

- When you encounter a left paren — `'{'`, `'['`, or `'('` — push it onto the stack.

- When you encounter a right paren, pop its counterpart from atop the stack, and check that they match.

If there is a mismatch or exception, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.

Detailed code is given in `ParenExample.java` on BlackBoard. It uses the `PureStack` interface, but it could have easily been done using `List` or `Deque`.

Arithmetic Expressions

Infix notation is the notation for arithmetic and logical formula to which you are accustomed. In it, operators are written between the operands they act on; e.g., "`2 + 2`". Parentheses surrounding groups of operands and operators are used to indicate the intended order in which operations are to be performed. In the absence of parentheses, ***precedence rules*** determine the order of operations.

Example. The infix expression "`3−4*5`" is evaluated as "`3−(4*5)`", not as "`(3−4)*5`", because "`−`" has lower precedence than "`*`". If you want it to evaluate the second way, you need to parenthesize.

In ***postfix notation***, also known as ***reverse Polish notation***[1] (RPN), the operators follow their operands. For instance, to add three and four, one would write “`3 4 +`” rather than “`3 + 4`”. If there are multiple operations, the operator is given immediately after its second operand; so the expression written “`3 − 4 + 5`” in infix notation would be written “`3 4 − 5 +`” in RPN: first subtract 4 from 3, then add 5 to that.

RPN obviates the need for the parentheses that are required by infix. While “`3 − 4 * 5`” can also be written “`3 − (4 * 5)`”, that means something quite different from “`(3 − 4) * 5`”. In postfix, the former is written “`3 4 5 * −`”, which unambiguously means “`3 (4 5 *) −`”, and which reduces to “`3 20 −`”; the latter is written “`3 4 − 5 *`”, which unambiguously means “`(3 4 −) 5 *`”. Postfix notation is easier to parse by computer than infix notation, but many programming languages use infix due to its familiarity.

Evaluating Postfix Expressions

Stacks and postfix expressions go hand-in-hand. To evaluate a postfix expression, start with an empty stack and scan the expression from left to right.

[1] Called “Polish” in reference to the nationality of logician Jan Łukasiewicz (1878–1956), who invented *prefix* notation.

- If the next item is a number, push it onto the stack.
- If the next item is an operator, *Op*, do the following:
    - Pop two items `right` and `left` off the stack.
    - Evaluate (`left Op right`) and push the value back onto the stack.

If there are fewer than two operands on the stack when scanning an operator, the expression must be invalid: there are more operators than operands. Now, suppose we have reached the end of the expression. If all went well (that is, if the expression is well-formed), then there is a single value in the stack: this must be the value of the expression. Otherwise, more than one operand remains on the stack. This means that the expression was invalid: it had too many operands.

Exercise. Try the above algorithm out on

$$7\ 11\ -\ 2\ *\ 3\ +$$

Note that, in infix, this expression would be written as

$$(7 - 11) * 2 + 3.$$

Comment. As someone pointed out in class today, the latter infix expression is equivalent to the infix expression

```
3 + 2 * (7 - 11)
```

which in postfix is

```
3 2 7 11 - * +
```

Since the work is O(1) per “token” (i.e., operand or operator) in the expression, the total time complexity is O(n), where n is the number of items.

A simple implementation of the preceding algorithm is posted on Blackboard (`PosfixExample.java` and `ExpressionFormatException.java`). It assumes that expressions are space-delimited.