Postfix Expressions

We normally write arithmetic expressions using **infix** notation: the operator (such as +) goes *in between* the operands (the two numbers being added).

Another way to write arithmetic expressions is to use **postfix** notation: the two operands come first, and the operator comes *after*.

For example,

- 3 4 + is same as 3 + 4
- 1 2 − 5 − 6 5 / + is same as
  \[(1 - 2) - 5 + (6 / 5)\]

One advantage of postfix is that *you don’t need parentheses to indicate the order of evaluation*.

For instance,

- (1 + 2) * 3 becomes 1 2 + 3 *
- 1 + (2 * 3) becomes 1 2 3 * +
Using a Stack to Evaluate Postfix Expressions

Pseudocode:

while input has more tokens do
  if the token is an operand then
    push it on the stack
  if the token is an operator then
    pop operand x off the stack
    pop operand y off the stack
    apply the operator to x and y
    push the result back on the stack
end while

return the value left on the stack

This works when all operators are binary (take two operands).
StringTokenizer Class

Java’s StringTokenizer class is very helpful to break up the input string into operators and operands — called “parsing”.

- Create a StringTokenizer object out of the input string. It converts the string into a sequence of tokens which are separated by specified delimiters.
- Use instance method hasMoreTokens to test whether there are more tokens.
- Use instance method nextToken to get the next token from the input string.
- Second argument to constructor indicates that, in addition to the whitespace characters (blank, newline, tab, and carriage return), the following are also used as delimiters: +, -, *, and /.
- Third argument to constructor indicates that all delimiters are to be returned as tokens.
Java Method to Evaluate Postfix Expressions

public static double evalPostFix(String postfix)
    throws EmptyStackException {
    Stack S = new Stack();
    StringTokenizer parser = new StringTokenizer(
            postfix, " \n\t\r+-/*", true);
    while (parser.hasMoreTokens()) {
        String token = parser.nextToken();
        char c = token.charAt(0);
        if (isOperator(c)) {
            double y = ((Double)S.pop()).doubleValue();
            double x = ((Double)S.pop()).doubleValue();
            switch (c) {
                case '+':
                    S.push(new Double(x+y)); break;
                case '-':
                    S.push(new Double(x-y)); break;
                case '*':
                    S.push(new Double(x*y)); break;
                case '/':
                    S.push(new Double(x/y)); break;
            } // end switch
        } // end if
        else if (!isWhiteSpace(c)) // token is operand
            S.push(Double.valueOf(token));
    } // end while
    return ((Double)S.pop()).doubleValue();
}
public static boolean isOperator(char c) {
    return ( (c == '+') || (c == '-') ||
            (c == '*') || (c == '/'));
}

public static boolean isWhiteSpace(char c) {
    return ( (c == ' ') || (c == '
') ||
            (c == '	') || (c == ''));
}

Does not handle negative numbers in the input: it interprets \(-3\) as the binary minus operator, followed by 3, instead of the unary minus applied to 3.

Does no error checking to see if operands are well-formed, or if the postfix expression itself is well-formed.
Implementing a Stack with an Array

Since Java supplies a Stack class, why bother? Basic understanding; other languages.

Idea: As elements are pushed, they are stored sequentially in an array, keeping track of the last element entered. To pop, return the element at the end of the active part of the array.

Issues for Java implementation:

- elements in the array are to be of type Object
- throw exception if try to pop an empty stack
- dynamically increase the size of the array to avoid overflow

To handle the last point, we’ll do the following:

- initially, the size of the array is, say, 16.
- if array is full and a push occurs, use new to create an array twice the size of current array, and copy everything in old array to new array.
Implementing a Stack with an Array in Java

class Stack {
    private Object[] A;
    private int next;

    public Stack () {
        A = new Object[16];
        next = 0;
    }

    public void push(Object obj) {
        if (next == A.length) {
            // array is full, double its size
            Object[] newA = new Object[2*A.length];
            for (int i = 0; i < next; i++)  // copy
                newA[i] = A[i];
            A = newA;  // old A can now be garbage collected
        }
        A[next] = obj;
        next++;
    }

    public Object pop() throws EmptyStackException {
        if (next == 0)
            throw new EmptyStackException();
        else {
            next--;
            return A[next];
        }
    }
}
Implementing a Stack with an Array in Java (cont’d)

```java
public boolean empty() {
    return (next == 0);
}

public Object peek() throws EmptyStackException {
    if (next == 0)
        throw new EmptyStackException();
    else
        return A[next-1];
}
} // end Stack class

class EmptyStackException extends Exception {

    public EmptyStackException() {
        super();
    }

}
```
Time Performance of Array Implementation

- push: $O(1)$ UNLESS array is full; then it is $O(n)$ plus time for system to allocate space (more later)
- pop: $O(1)$
- empty: $O(1)$
- peek: $O(1)$
Implementing a Stack with a Linked List in Java

Idea: a push causes a new node to be inserted at the beginning of the list, and a pop causes the first node of the list to be removed and returned.

class StackNode {
    Object item;
    StackNode link;
}

class Stack {
    private StackNode top; // first node in list, the top

    public Stack () {
        top = null;
    }

    public void push(Object obj) {
        StackNode node = new StackNode();
        node.item = obj;
        node.link = top;
        top = node;
    }
}
Implementing a Stack with a Linked List in Java (cont’d)

public Object pop() throws EmptyStackException {
    if (top == null)
        throw new EmptyStackException();
    else {
        StackNode temp = top;
        top = top.link;
        return temp.item;
    }
}

public boolean empty() {
    return (top == null);
}

public Object peek() throws EmptyStackException {
    if (top == null)
        throw new EmptyStackException();
    else
        return top.item;
}

Time Performance of Linked List Implementation

- push: $O(1)$ plus time for system to allocate space (more later)
- pop: $O(1)$
- empty: $O(1)$
- peek: $O(1)$
Interchangeability of Implementations

If you have done things right, you can:

- write a program using the built-in Stack class
- compile and run that program
- then make available your own Stack class, using the array implementation (e.g., put Stack.class in the same directory

- **WITHOUT CHANGING OR RECOMPILING YOUR PROGRAM**, run your program — it will use the local Stack implementation and will still be correct!

- then replace the array-based Stack.class file with your own linked-list-based Stack.class file

- again, **WITHOUT CHANGING OR RECOMPILING YOUR PROGRAM**, run your program — it will use the local Stack implementation and will still be correct!