

Stacks and Queues

The data type Stack is pervasive in computing. It is fundamental in parsing and in evaluation, it is the natural way to maintain successive calls to methods or functions. It is an obvious way to model recursion.

A stack is a linear structure that is characterised by the special way that the operations of insert and delete take place. They both always operate at one and the same end of the structure (called the **top**). A stack serves as a kind of temporary memory of the order in which its elements were added. Traditionally the insert operation on a stack is called **push** and the removal, or deletion, of the top element is called **pop**. So the last element that has been pushed onto the stack must be the first element to be removed when we apply pop. Hence the structure is said to have a last-in first-out (**LIFO**) behaviour.

A stack can be considered abstractly as a list ordered by the time the elements were added. The only difference between a stack and a general list is the restriction on the operations of push and pop: they operate at one end of the list only.

Specification of the ADT Stack

Domain: A bounded linear sequence of data elements ordered by ‘time of arrival’.

Operations:

Push (Stack S, Object e)

pre: not Full(S)

post: post-S is pre-S with element e added on the top

Pop (Stack: S, Object e)

pre: not Empty(S)

post: post-S is pre-S with its top element removed, e stores the top element of pre-S.

Empty (Stack S) : boolean

pre: none

post: Empty is true iff S is empty

Full(Stack S) : boolean

pre: None

post: Full is true iff S is full

Create (Stack S, created boolean)

pre: none

post if S can be created then created is true and S is an empty stack, otherwise created is false.

Destroy (Stack S)

pre : none

post: the stack S is destroyed

We can separate the operations associated with pop into:

- (a) inspecting and copying the top element (peek);
- (b) deleting the top element

A parenthesis checker...

A common procedure is to check an expression for whether the parentheses occurring in it are well-balanced. For example:

- (i) $((a + b) + c) * d$ extra left parenthesis
- (ii) $(a + b) + c) * d$ missing left parenthesis
- (iii) $((a + b) + c) * d$ matching parentheses

The algorithm to be followed is:

```
read the characters of a string from left to right
while there are characters to read do
    begin
        if character read is '('
            push a marking element on to a stack
        else
            if character read is ')'
                pop the stack
            else
                go to next character
        end;
```

If the pop operation fails, or the stack has any elements on it at the end, then the parentheses are mismatched, otherwise they are matched.

It is left as an exercise here to write an implementation of the above algorithm in Java.

Stacks and queues are examples of linear data structures. A **linear** structure is one which has unique first and last elements and each element except the last has a unique successor, each except the first has a unique predecessor. (Note, as in lecture, an **ordered** structure may not be linear.) An interface for Stack is as follows. Note that the operation names are traditional and used in most books (*except W&B!*).

```
public interface Stack {

    public boolean isEmpty();
    //post: true iff stack is empty

    public void clear();
    //post: stack is empty

    public void push (Object item);
    //post: item is added to the top of
    //stack

    public Object pop ();
    //pre: stack is not empty
    //post: most recently pushed item is
    //removed and returned

    public Object peek();
    //pre: stack is not empty
    //post: top value is returned (but not
    //removed)

}
```

This interface, together with other features of Java, does satisfy the specification given above. Operations Full() and Create() are dealt with by StackOverflowError and the class constructors respectively.

There is a Stack class provided in java.util.Stack which is implemented using Vectors. So the key operations will look like:

```
protected Vector data;

public StackVector(int size) {

//post: creates an empty stack with
//initial capacity of size

    data = new Vector(size);
}

.....

public void push(Object item) {
//post: item is added to top of stack

    data.addElement(item);
}

public Object pop() {
//pre: stack is not empty
//post: most recently pushed item removed
//and returned

Object result = data.elementAt(size()-1);
data.removeElementAt(size()-1);
```

```
return result;  
}
```

Note that this is correct only because `addElement()` is defined in the `Vector` class to append the item onto the end of the vector (which here corresponds to the top of the stack at position `size() - 1`).

It is equally simple to use `addToHead()` and `removeFromHead()` in the `SinglyLinkedList` provided in the `structure` package to provide a linked-list implementation of `Stack`. For details and comparison see Bailey (pp.134-6). See also W&B Ch.6.

A classical problem in compiling is to parse arithmetic expressions into well-formed parts ready for evaluation. A convenient way to do this is first to use a stack to convert infix expressions into postfix expressions. Suppose Q is an infix expression which may contain left and right parentheses and operators for power ($^$), multiplication ($*$), division ($/$), addition ($+$) and subtraction ($-$) with usual rules of precedence. Then an algorithm to convert Q into an equivalent postfix expression is:

Polish (Q,P)

1. Push '(' onto a stack, and add ')' to the end of Q .
2. Scan Q from left to right and repeat steps 3 – 6 for each element of Q until the stack is empty.
 3. If an operand is read, add it to P .
 4. If a left parenthesis is read, push it on stack.
 5. If an operator OP is read then

- (a) repeatedly pop from the stack and add to P each operator (on the top of the stack) which has the same precedence as, or higher precedence than, OP;
- (b) add OP to the stack.

6. If a right parenthesis is read then:

- (a) repeatedly pop from the stack and add to P each operator (on the top of the stack) until a left parenthesis is read;
- (b) remove left parenthesis (do not add to P).

Use convention that symbols on stack are listed from bottom-to-top in left-to-right order as a string.

Consider conversion of the infix expression
 $a + (b * c - (d / e ^ f) * g) * h$
 into postfix.

Line	Q symbol	stack state	P
1	a	(a
2	+	(+	a
3	((+(a
4	b	(+(ab
5	*	(+(*	ab
6	c	(+(*	abc
7	-	(+(-	abc*
8	((+(-(abc*
9	d	(+(-(abc*d
10	/	(+(-(/	abc*d

11	e	(+(-(/	abc*de
12	^	(+(-(/ ^	abc*de
13	f	(+(-(/ ^	abc*def
14)	(+(-	abc*def^/
15	*	(+(- *	abc*def^/
16	g	(+(- *	abc*def^/g
17)	(+	abc*def^/g*-
18	*	(+ *	abc*def^/g*-
19	h	(+ *	abc*def^/g*-h
20)		abc*def^/g*-h*+

The subtraction (-) operator in line 7 sends * from the stack to P before it (-) is pushed onto the stack. The right parenthesis in line 14 sends ^ and then / from the stack to P, and then removes the left parenthesis from top of stack. The right parenthesis in line 20 sends * and then + from the stack to P, then removes the left parenthesis from the top of the stack.

Exercises:

- (i) Follow through each step of the above example of the Polish(Q,P) algorithm noting how precedence and parentheses in the original expression Q are dealt with appropriately.
- (ii) Use the algorithm on the infix expression $((a + b) * d)^{(e - f)}$ to show that the equivalent postfix expression is $P = a b + d * e f - ^$