Stack ADT

You are familiar with stacks in your everyday life. You can have a stack of books on a table. You can have a stack of plates on a shelf. In computer science, a *stack* is an abstract data type (ADT) with two operations: `push` and `pop`. You either push something onto the top of the stack or you pop the element that is on the top of the stack. A more elaborate ADT for the stack might allow you to check if the stack has any items in it (`isEmpty`) or to examine the top element without popping it (`top`, also known as `peek`). But these operations not necessary for us to call something a stack.

[ASIDE: Note that a stack is a kind of list, in the sense that it is a finite set of ordered elements. However, with stacks, you typically only are allowed to use `push, pop, isEmpty,` and `top` operations. In Java, you can declare a variable to be of type `Stack` which this allows you to use the methods of a `List` too. Traditional computer scientists frown this, since it blurs the notion of what is a stack. A traditional stack does not allow you to say `get(i)`.]

Example 1

Here we make a stack of numbers. We assume the stack is empty initially, and then we have a sequence of pushes and pops.

```
push(3)
push(6)
push(4)
push(1)
pop()
push(5)
pop()
pop()
```

The elements that are popped will be `1`, `5`, `4` in that order, and afterwards the stack will have two elements in it, with `6` at the top and `3` below it.

```
                  1           5
            4     4     4     4     4
      6     6     6     6     6     6     6
3     3     3     3     3     3     3     3
--    --    --    --    --    --    --    --
```

Example 2: Balancing parentheses

It often occurs that you have a string of symbols which include left and right parentheses that must be properly nested or balanced. (In this discussion, I will use the term "nested" and "balanced" interchangeably.) One checks for proper nesting using a stack.

Example 2a

Consider the opening and closing parentheses "`(`" and "`)`" and an expression:

```
3 + (4 - x)* 7 + (y - 2 * (2 + x)).
```

in which the parentheses *are* balanced. But how do we determine that it is balanced?

Let's scan through this expression and ignore the variables (x,y), numbers, and operators *, +, -. Whenever we see a left parenthesis, we push it on the stack, and when we see a right parenthesis, we pop the (matching) left parenthesis from the stack. The sequence of stack states is:

```
                    (
      (         (   (   (
---  ---  ---  ---  ---  ---  ---
```

If were to try to pop an empty stack (i.e. we reach a right parenthesis and the stack is empty), or we were to finish scanning the string and the stack were non-empty, then we would have an error – the parentheses would not be balanced.

For our problem here, you don't really need a stack. You just need a counter. You start the counter at 0 and increment the counter when you see a left parenthesis and decrement the counter when you see a right parenthesis. The parentheses are balanced if the counter never becomes negative, and it is 0 when you are finished scanning.

Example 2b

A more interesting example of balancing parentheses in which you *do* need a stack is if there are multiple types of left and right parentheses, for example, (,), {, }, [,], and you require that the parentheses are properly *nested*. Consider the string:

```
( ( [ ] ) ) [ ] { [ ] }
```

You can check for balanced parentheses using a stack. You scan the string left to right. When you read a left parenthesis you push it onto the stack. When you read a right parenthesis, you pop the stack (which contains only left parentheses) and check if the popped left parenthesis matches the right parenthesis that you just read. For the above example, the sequence of stack states would be as follows.

```
          [
      (   (   (                       [
   (  (   (   (   (       [       {   {   {
-  -  -   -   -   -   -   -   -   -   -   -   -
```

and the algorithm terminates with an empty stack. So the parentheses are properly balanced.

Example 2c

Here is an example where each type of parenthesis on its own is balanced, but overall the parentheses are not balanced.

```
( ( [ ) ] ) [ [ ] ] { [ } ]
```

```
             [
         (   (
     (   (   (
--- --- --- ---  X  since next symbol is ")" which doesn't match top
```

Algorithm for balancing parentheses

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. A token can be one of the following:

- a left parenthesis (there may be various kinds)
- a right parenthesis (there may be various kinds)
- a string not containing a left or right parenthesis (operators, variables, numbers, etc)

Any token other than a left or right parenthesis is ignored by the algorithm.

```
ALGORITHM:  CHECK FOR BALANCED LEFT AND RIGHT PARENTHESES
INPUT:  SEQUENCE OF TOKENS
OUTPUT: TRUE OR FALSE (I.E. BALANCED OR NOT)

WHILE (not at end of token sequence){
   token <-  get next token
   if token is a left parenthesis
      push(token)
   else if token is a right parenthesis {
           if (stack is empty)
               return false
           else{
               left  <-  pop()
               if  !( left.matches(token) )
                   return false
           }
         }
}
return (stack.isEmpty())
```

Example 3: HTML tags

The above problem of balancing different types of parentheses might seem a bit contrived. But in fact, this arises in many real situations. An example is HTML *tags*.[6] They are of the form `<tag>` and `</tag>` and these correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface".

[6] If you have never looked at HTML source code before, then open a web browser right NOW and look at "view → page source" and check out the tags. They are the things with the angular brackets.

HTML tags are *supposed to be* properly nested[7] For example, consider

```
<b> I am boldface, <i> I am boldface and italic, </i>
</b><i> I am just italic </i>.
```

The tag sequence is `<b><i></i></b><i></i>` and the "parenthesis" are indeed balanced, i.e. properly nested. Compare that too

```
<b> I am boldface, <i> I am boldface and italic </b>
  I am just italic </i>
```

whose tags sequence is `<b><i></b></i>` which is not properly balanced. The latter is the kind of thing that novice HTML programmers write – and it's logical. They are treating the tags and turning an imaginary attribute state (bold, italic) on or off. But the HTML language is not supposed to allows this. And writing HTML code this way can get you into trouble since errors such as a forgotten or extra parenthesis can be very hard to find.

See `http://www.w3schools.com` for basic HTML tutorials (and other useful simple tutorials).

Example 4: if-then-else statements

Consider a language that allows two types of if-then-else statements:

```
if  boolean  then statement  else statement
if  boolean  then statement
```

Notice that statements can be nested, i.e. you can have a statement within a statement (within a statement within a statement etc). For such a language, you often need brackets to disambiguate a given statement. For example, there are two interpretations possible for the statement:

```
if  (i > 0) then  if (a > 0) then b=4 else b = 5
```

These two interpretations are

```
if  (i > 0) then { if (a > 0) then b=4   else b = 5 }    //  FIRST
if  (i > 0) then { if (a > 0) then b=4 } else b = 5      //  SECOND
```

Let `F,T` be false and true, respectively. The table below shows the actions taken in the four combinations of the conditions. Note how easy it is to make mistakes with this, even with brackets !

i > 0	a > 0	FIRST	SECOND
F	F	do nothing	b = 5
F	T	do nothing	b = 5
T	F	b = 5	do nothing
T	T	b = 4	b = 4

[7]Many HTML authors write improply nested HTML code. Because of this, web browsers typically will allow improper nesting. Why? Because web browser programmers (e.g. google employers who work on Chrome) want people to use their browser and if the browser displayed junk when trying to interpret improper HTML code, then users of the browser would give up and find another browser.

Interestingly if the language only allows one type of if-then-else statement:

```
if  boolean  then statement  else statement
```

then you don't need brackets, no matter how deep the nesting. For example,

```
if (i > 0) then if (a > 0) then b=4 else if (i < 5) then b = 6 else b = 7
```

is uniquely parsed as

```
if (i>0) then {if (a>0) then b=4 else {if (i < 5) then b = 6 else b = 7}}
```

Example 5: Evaluation of arithmetic expressions

Consider an arithmetic expression such as:

```
3 + ( 4 - 1 ) * 7 + ( 6 - 2 * ( 2 + 3 ) ).
```

Let's scan through this expression from left to right. The following algorithm tell us exactly how using two stacks, an Operator stack to store operators `*`, `+`, `-` as well as opening parentheses, and an Argument stack to store the numbers involved in arithmetic operations, we determine the value of such an expression.

```
 ALGORITHM:  EVALUATE ARITHMETIC EXPRESSION
 INPUT:  SEQUENCE OF TOKENS
 OUTPUT: VALUE OF EXPRESSION

WHILE (not at end of token sequence){
    token <- get next token
    if type(token) == number { pushA(token) }
    if type(token) == operator {
       if prio(token) <= prio(topO()) {
          op <- popO()
          arg2 <- popA()
          arg1 <- popA()
          pushA(exec(arg1,op,arg2)) }
       pushO(token) }
    if token == "(" { pushO(token) }
    if token == ")" {
       op <- popO()
       WHILE NOT op == "(" {
          arg2 <- popA()
          arg1 <- popA()
          pushA(exec(arg1,op,arg2))
          op <- popO() } } }
WHILE NOT isemptyO(){
  op <- popO()
  arg2 <- popA()
  arg1 <- popA()
  pushA(exec(arg1,op,arg2)) }
return(popA())
```

Example 6: the "call stack"

We have been discussing stacks of things. One can also have a stacks of tasks.[8] Imagine you are sitting at your desk getting some work done (main task). Someone knocks on your door and you let them in and chat (task A). While chatting, the phone rings and you answer it (task B). You finish the phone conversation and go back to the person in your office (A). Then maybe there is another interruption (C) which you take care of, return to A and return to main.

A similar stack of tasks occurs when a computer program runs. Say we have a `main` method where the program starts. The main method typically has instructions that cause other methods to be called. The program "jumps" to these methods, executes them and returns to the main method. Sometimes these methods themselves call other methods, and so the program jumps to these other methods, executes them, returns to the calling method, which finishes, and then returns to main.

A natural way to think of jumping from method to method is in terms of a stack. Suppose `main` calls method `mA` which calls method `mB`, and then when `mB` returns, `mA` calls `mC`, which eventually returns to `mA`, which eventually returns to `main` which then finishes. (See slides for this lecture.)

```
                        mB          mC
                mA      mA      mA      mA      mA
        main    main    main    main    main    main    main
----    ----    ----    ----    ----    ----    ----    ----    ----
```

Why do we need to use a stack? Can't you just jump from method to method as the program runs? No, that won't work. The problem is that when a method finishes and returns, it needs to remember where to return. This information (the "return address") is part of the bundle of information (called a stack frame) that is thrown on the stack.[9]

Data structure for a stack

Finally, ... what is a good data structure for a stack? A stack is a list, so its natural to use one of the list data structures we have considered.

If you use a singly linked list to implement a stack, then you should push and pop to/from the front of the list, not to/from the back. The reason is that removing (popping) from the back of a singly linked list is inefficient i.e. you need to walk through the entire list to find the node that points to the last element (which you are popping). For a doubly linked list, it doesn't matter whether you push/pop at the front (head) or at the back (tail) as long as you do one or the other.

If you use an array, then you should push and pop to/from the end of the list (indicated by index top or size-1). The reason is that if you add/remove from the front of an array that you need to shift all the other elements if you want to maintain the property that the top is at index 0.

[8] though typically we don't call them "stacks", that is what they are.

[9] How *exactly* the call stack and the "return address" work is a much more advanced topic which you will learn about properly in COMP 273 or in ECSE 221.