Stack ADT

We began our discussion of lists back in lecture 3 by discussing what a list is in an abstract sense: a set of things and a certain set of methods (operations) that are applied to these things. Stated in this general way, a *list* is an abstract data type (ADT). We will see two more ADT's in the next few lectures, namely the stack and the queue.

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. Note that a stack is a kind of list, in the sense that it is a finite set of ordered elements. However, it is restricted type of list since it has fewer operations you can apply on it.

Data structure for a stack

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 an array list, then you should push and pop to/from the end of the list (`addLast()` or `removeLast`. The reason is that if you add or remove from the front of an array list, you need to shift all the other elements each time which is inefficient.

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 (recall) is that removing i.e. popping from the back of a singly linked list would be 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 or at the back. Both would have the same efficiency.

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. Here is how the stack evolves over time:

```
                    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.

Suppose there are multiple types of left and right parentheses, for example, `(`, `)`, `{`, `}`, `[`, `]`. 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. 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.

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 checking if parentheses are balanced

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. [**In the lecture, I forgot to mention that a token can include things other than left or right parentheses, but that the algorithm ignores these other things. So, essentially this algorithm just deals with left and right parentheses.**] 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()
```

In the lecture, I did not go through this algorithm line by line (and in general in this course I will try to avoid walking you through pseudocode). I do want you to go through it yourself at home though. Pseudocode is a very helpful way to formulate ideas before getting tied down to particular language syntax e.g. Java, so you need to get used to it.

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*. If you have never looked at HTML markup 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.

Tags are of the form `<tag>` and `</tag>`. They correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface". HTML tags are *supposed to be* properly nested. 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. It does make some sense, if you think of the tags as turning on or off some state (bold, italic). But the HTML language is not supposed to allow this. And writing HTML markup this way can get you into trouble since errors such as a forgotten or extra parenthesis can be very hard to find.

ASIDE: Many HTML authors write improply nested HTML markup. Because of this, web browsers typically will allow improper nesting. The reason is that 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 markup, then users of the browser would give up and find another browser.

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

Example 4: if-then-else statements (see Assignment 2)

Consider a language that allows if-then-else statements of the form:

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

Since a `statement` can itself be an if-then-else statement, we can have statements within statements within statements, etc. To parse statements, one generally uses a stack. You will do something similar in Assignment 2.

Example 5: stacks in graphics

The next example is a simple version of how stacks are used in computer graphics. Consider a drawing program which can draw unit line segments (say 1 cm). Suppose the pen tip has a *state* (x, y, θ) that specifies its (x, y) position on the page and an angular direction θ. This is the direction in which it will draw the next line segment (see below). The pen state is initialized to be (0,0,0), where $\theta = 0$ is in the direction of the x axis.

Let's say there are five commands:

- `D` - draws a unit line segment from the current position and in the direction of θ, that it, it draws it from (x_0, y_0) to $(x_0 + \cos\theta, y_0 + \sin\theta)$. It moves the state position to the end of the line just drawn. Recall $\cos(0) = 1, \cos(90) = 0, \cos(180) = -1, \sin(0) = 0, \sin(90) = 1, \sin(180) = 0, ...$
- `L` - turns left (counter-clockwise) by 90 degrees
- `R` - turns right (clockwise) by 90 degrees
- `[` - pushes the current state onto the stack
- `]` - pops the stack, and current state $\leftarrow$ popped state

See the slides for a few examples.

Example 6: the "call stack"

We have been discussing stacks of things. One can also have a stacks of tasks. 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. While chatting, the phone rings and you answer it. You finish the phone conversation and go back to the person in your office. Then maybe there is another interruption which you take care of, return to work, etc. In each case, when you are done with a task, you ask yourself "what was I doing just before I began this task?".

A similar stack of tasks occurs when a computer program runs. The program starts with a `main` method. 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 keep track of methods and to return to the 'caller' is to use 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.

```
Class    Demo {
   void   mA( ) {
      mB( );
      mC( );
   }
   void  mB( ) {  }
   void  mC( ) {  }
   void  main( ){
       mA(  );
   }
}
```

The stack evolves as follows:

```
                        B             C
                 A      A      A      A      A
        main    main   main   main   main   main  main
----    ----    ----   ----   ----   ----   ----  ----  ----
```

Also see the slides for an example using the `SLinkedList1` code from the linked list exercises. I briefly showed how the `TestSLinkedList1` calls the `addLast()` method of the LinkedList clas, and I show the Eclipse call stack. When you use Eclipse in debugger mode, and you set breakpoints in the middle of methods, there is a panel that shows you the call stack.