In this lecture and the next, we are going to look at stacks and queues. Each is an ordered collection of objects, where the ordering determined by *when* each object was inserted into the collection. With a stack, one accesses the newest element (most recently inserted). With a queue, one accesses the oldest element (least recently inserted). Today we look at stacks.

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. We also saw a problem (Tower of Hanoi) which used stacks (three of them), though we didn't call them stacks.

The basic *stack* has an abstract data type with two operations: `push` and `pop`. You either push something onto the top of the stack or you pop something off 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`) or to ask how many elements there are (`size`). But these operations not necessary for us to call something a stack.

Example

Here we manipulate a stack of strings. We assume the stack is empty initially.

```
push(``3'')
push(``fred'')
push(``blue'')
push(``hello'')
pop()
push(``green'')
pop()
pop()
```

The elements that are popped will be `` `hello'' ``, `` `green'' ``, `` `blue'' `` in that order, and afterwards the stack will have two elements in it (`` `fred'' `` and `` `3'' ``, with `` `fred'' `` being on top).

Applications of Stacks

Before we examine common ways of implementing a stack, let's look at a few problems that involve stacks.

Balancing parentheses

It often occurs that you have a string of symbols which include left and right parentheses that must be properly nested and balanced. For example, consider the opening and closing parentheses "(" and ")" and the expression:

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

Or consider html's opening and closing tags (sometimes called begin and end tags) which are of the form <tag> and </tag>. These correspond to left and right parentheses, respectively. These tags can demarcate regions of italic, specific list items in a list, etc. Here is an example:

```
<li> <i>A Little Book on Java</i> by F. Aahmad and P. Panangade n</li> <li> The course
textbook is <a href="http://ww0.java4.datastructures.net/"> <i> Data Structures and
Algorithms in Java </i></a> (Fourth Edition) by M. T. Goodrich and R. Tamassia </li>
```

And here are few examples in which the parentheses are *not* balanced:

- 3 + (4 - x)* 7) + (y))) - ((2 * (2 + x))
- 3 + (4 - x]* 7 + [y - 2 * (2 + x))
- <i>A Little Book on Java by F. Aahmad and P. Panangaden The course textbook is <i> Data Structures and Algorithms in Java </i> (Fourth Edition) by M. T. Goodrich and R. Tamassia

To verify that a sequence has balanced parenthesis, one uses a stack. (If the order of the parenthesese is not important, then you don't need a stack. You can just use a counter. But typically order *is* important.

What is the intuition here? You can have as many left parentheses as you like. But as soon as you have a right parenthesis, you need to check if this right parenthesis matches the previous parenthesis, namely the previous parenthesis needs to be a left parenthesis of the same kind. Locally examining the sequence, you can tell that ...( )... matches, and that ...( ]... doesn't match.

The idea is to push each left parenthesis onto the stack, and then when you meet a right parenthesis, you pop the stack (which contains only left parentheses) and checked that this popped left parenthesis matches the current right parenthesis you are considering.

The basic algorithm is shown on the next page. We assume the input has been already partioned ("parsed") into disjoint *tokens*. A token can be one of the following:

- a left parenthesis
- a right parenthesies
- a string not containing a left or right parenthesis

Example: 3 + (4 - x)* 7 + (y - 2 * (2 + x))

The sequence of stack states is:

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

Algorithm: check for balanced left and right parentheses
Input: sequence of tokens (see above)
Output: true or false

```
while (not end of expression) do
   t ← next token
   if t is a left parenthesis then
      push(t)
   else
      if t is a right parenthesis then
         if stack is empty then
            return false
         else
            left ← pop()
            if !(match(left, t)) then
               return false
            end if
         end if
      end if
   end if
end while
if stack is empty then
   return true
else
   return false
end if
```

Example: html

Here is the html snippet with irrelevant characters removed, and below it is shown the sequence of stack states.

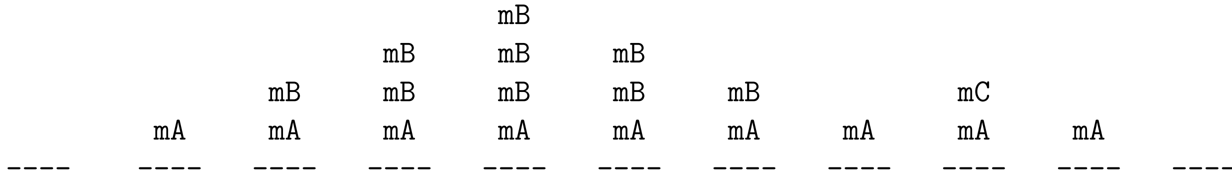
```
<li> <i>  <b> </b>   <b> </li> <li> </b> <a> <i> </b> </i> </a> </b> </li>
```

```
            b           b
     i      i     i     i
li   li     li   li    li
---  ---   ---  ---   ---
```

At that point we pop the left parenthesis `<b>`, but the match to `</li>` fails and so we know the sequence is not balanced.

Call stack

In lecture 10, I briefly discussed the "call stack" which is used at run time to keep track of methods that call each other. Suppose method `mA` calls methods `mB` and then `mC`. Moreover, suppose method `mB` is a recursive method, which (in the case we consider here) calls itself twice before reaching the base case. Then the call stack has the following sequence of states:



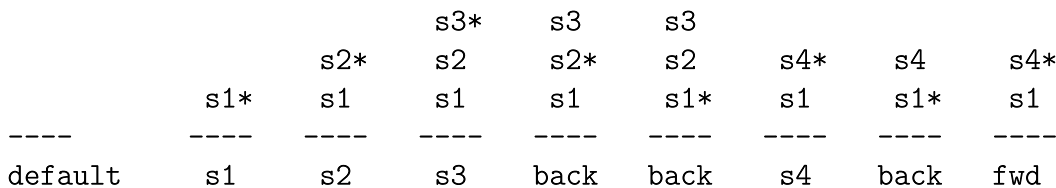
[ASIDE: Each stack frame holds data including the return address (method and line), local variables, arguments passed. You will learn how stack frames work in more detail in later courses.]

An example that is *not* a stack

Web browsers sometimes have "stack-like" behavior for keeping track of the sequences of websites you visit. You first open the browser and it defaults to some web page. Then you start navigating and build up a sequence of pages visited. At some point you may wish to use the BACK button to go back to previously visited pages. This operation is stack-like (pop).

The data structure used is not a stack, however. Rather, it is some sort of list. For example, the `back` and `forward` buttons allow you to move a "current" reference (the `*` in the example below) along the list by following the next and prev links (as well as jumping to an arbitrary point in the list).

In the example below, note what happens when you visit site `s4`, while you are currently at site `s1`. The sites `s2` and `s3` that were in the list are chopped off. (That is how Firefox and Explorer do it. Check it out for yourself.) I mention this example because it is often happens in computer science that one refers to stack like properties (push and pop) for data structures that are not, strictly speaking, stacks. You should keep a "heads up" for this.



Implementing a Stack (not discussed in class)

The textbook (chapter 5) has an implementation of stacks, which use an interface `Stack`. Source code is available from: `http://ww0.java4.datastructures.net/source/`
You are not responsible for knowing it, but I recommend you have a look. In particular, you will see how generics have been added to the earlier representation of lists.

The authors provide two classes that implement their `Stack` interface. The first uses a singly linked list. To `push` an element onto the stack, they add it to the front of the list. To `pop`,

they remove an item from the front of the list (modifying the current `head`). This might seem counterintuitive – you might think you should push an item by adding it the back of the list (modifying the current `tail`). However, this would be inefficient. Removing an item from the back (tail) of singly linked list is $\Theta(n)$ since it requires that we traverse the whole list.

array

They also provide implementation of a stack using an array (more specifically, an ArrayList see p. 194). Let's suppose that the top item is at array index `top`. The basic idea is that stack is initialized to be empty and `top` is initialized to -1. Whenever they push an element into the top of the stack, they increment `top`, and whenever they pop an element from the top of the stack they decrement `top`.

Java class `Stack`

Java doesn't have an interface `Stack`. Rather, Java has a class `Stack`, namely is `java.util.Stack<E>`. Interestingly, it is really not a stack. It has methods `push` and `pop`. But this class extends a class `Vector` which is very similar to `ArrayList`. The class `Vector` has methods `add(int i)` and `remove(int i)` which allow you to add or remove an item from an *arbitrary position* in the "stack." Real stacks don't work that way.

The basic issue here is this: every stack is a list, but not every list is a stack. Purists say that Java's `Stack` class is therefore a bad example of the use of inheritance.