Chapter 6-1, Lecture notes

The Abstract Data Type Stack

1 We are now in part two of the text where we will concentrate on other data types that can be useful in problem solving.

1.1 A well-known ADT is the stack – an important one, by the way.

1.1.1 Consider what happens when you type on a keyboard and make a mistake.
1.1.2 Typically you would use the backspace key to correct the boo-boo and keep on typing.
1.1.3 For instance, if you enter the following (with ← meaning backspace):

abcc←ddde←←←ef←fg

1.1.4 The corrected input would be:

abcdefg

1.1.5 How would you implement entering in the line?
1.1.5.1 We can define the operations and delay the details in true ADT fashion.
1.1.5.2 The first draft of the pseudo code would look like:

```
// read the line, correcting mistakes along the way
while (not end of line)
{       Read new character ch
        if (ch is not a '←')
                Add ch to the ADT
        else
                remove from the ADT the item that was added most recently
} // while
```

1.1.6 Two operations the ADT must have are:

- Add a new item to the ADT
- Remove from the ADT the item that was added most recently

1.1.7 Now what happens if you press the '←' and the ADT is empty?
1.1.7.1 You could either generate an error and terminate or you could just ignore the '←'.
1.1.7.2 We'll go for the ignoring. So the pseudo code looks like:

```
// read the line, correcting mistakes along the way
while (not end of line)
{       Read new character ch
        if (ch is not a '←')
                Add ch to the ADT
        else if (the ADT is not empty)
                remove from the ADT the item that was added most recently
        else
                Ignore the '←'
} // while
```

1.1.8 So we need to add another operation to the ADT:
- Determine whether the ADT is empty

1.2 Now how do we display the line? We could do a hunk of pseudo code that does:

```
Remove from the ADT the item most recently added
Write that … Uh-oh!
```

1.2.1 The problem is when you remove the item it is GONE!
1.2.2 Also the most recently added item is at the BACK of the line not the front where we want to start printing.
1.2.3 For now, we'll deal with the first problem and delay writing the line forward (we'll write it backwards for now).
1.2.4 So we need a Retrieve last item operation that puts a copy of the last item in another variable, THEN we can remove the last item!
1.2.5 Therefore:

```
// write the line in reversed order
while (the ADT is not empty)
{       Retrieve from the ADT the item was added most recently and put it in ch
        Write ch
        Remove from the ADT the item that was added most recently
} // end while
```

1.2.6 So the fourth required operation for the ADT is:
- Retrieve from the ADT the item that was added most recently

1.3 The operations thus far would apply to the ADT that is usually called a "stack".
1.3.1 The ADT stack operations would be:
- Create an empty stack
- Destroy an empty stack
- Determine whether a stack is empty
- Add a new item to the stack
- Remove from the stack the item that was added most recently
- Retrieve from the stack the item that was added most recently

2 The term “stack” is frequently associated with a stack of dishes in a cafeteria (you can also have a stack of books or a stack of “to-dos”).

2.1 A “Stack of” can be also called a “pile of”.

2.1.1 However, pardoning the pun, with computer scientists, a stack is not just “any old pile”.

2.1.2 A stack would have a definite property of “Last in, first out” or “LIFO”.

2.2 The stack of dishes at the cafeteria is an excellent example of this.

2.2.1 Take a dish off the stack, “top of the stack”, and the other dishes move up.

2.2.2 Put new dishes onto the stack and the older dishes move down.

2.2.3 The first dish added to the stack would not come off until all the other dishes had been used … First dish on would be last dish off!

2.3 Now if we tried to apply this to a line to a movie the first poor soul to show up may be the last to see the movie!

2.3.1 For lines to the movie or the grocery check out (and to be fair), we use the First in, First out property or “FIFO”.

2.3.2 A line in Britain would be called a “queue” and that would be the term we would use for FIFO (and, in real life, we would have movie queues and NOT movie stacks).

2.3.3 But in many problems in computer science a stack is exactly what we would need!

3 But with the implementation of the stack, it would not necessarily be like a stack of dishes in a cafeteria.

3.1 Removing the top item off the stack does NOT imply moving all the other items.
3.2 Also, you can only add to the top, access the top, and remove to the top of the stack – necessary restrictions.
3.3 Though one refinement we could have is to not have the remove the top and retrieve the top be separate operations.
3.4 We could combine them so that we could inspect AND remove the top item!
3.5 That is not that unusual of an operation.

4 So our Operation Contract for the ADT Stack would be:

StackItemType is the type of items stored in the stack
isEmpty():boolean {query} Determines whether this stack is empty
push(in newItem:StackItemType) throw StackException Adds newItem to the top of this stack. Throw StackException if insertion fails
pop() throw StackException Removes the top of this stack; that is, removes the item that was added most recently. Throws StackException if the deletion is not successful
pop(out stackTop:StackItemType) throw StackException Retrieves into stackTop and then removes the top of this stack. That is, retrieves and removes the item that was added most recently. Throws StackException if delete fails.
getTop(out stackTop:StackItemType) {query} throw StackException Retrieves into StackTop the top of this stack (i.e. the item most recently added). Does not change the stack. Throws StackException if retrieval fails

5 Let's try the above in the read a line problem like so:

```
+readAndCorrect(out aStack:Stack)
// Reads the input line. For each character read, either enters it into stack or, if it is
// '←', corrects the contents of aStack.

        aStack.createStack()
        Read newChar
        While (newChar is not the end-of-line symbol)
        {       if (newChar is not '←')
                        aStack.push(newChar)
                else if (!aStack.isEmpty())
                        aStack.pop()

                Read newChar
        } // end while

+displayBackward(in aStack:Stack)
// Displays the input line in reversed order by writing the contents of stack aStack.

        while (!aStack.isEmpty())
        {       aStack.pop(newChar)
                Write newChar
        } // end while

        Advance to new line
```

Chapter 6-2, Simple Applications of the ADT Stack

1 Let's try one simple example that the LIFO property of a stack is ideal for.
  1.1 Let's check for balanced curly braces, "{}", like with C++.
    1.1.1 You need an open brace and a close brace like so:

      abc{defg{ijk}{l{mn}}op}qr

    1.1.2 Those are balanced but:

      abc{def}}{ghij{kl}m

    1.1.3 are not balanced.

  1.2 There are certain rules you must follow with matching up the curly braces.
    1.2.1 The string of "}a{" is NOT balanced.
    1.2.2 The rules are:
      - Each time you encounter a "}", it matches an already encountered "{"
      - When you reach the end of the string, you have matched each "{"

  1.3 You need to keep track of each unmatched "{" and discard one each time you encounter a "}".
    1.3.1 One way to do this is to push a "{" onto the stack and pop one off each time you hit a "}".
    1.3.2 Like so:

```
while (not at the end of the string)
{       if (the next character is a '{')
            aStack.push('{')
        else if (the character is a '}')
            aStack.pop()
} // end while
```

    1.3.3 The problem with the above code is that it does not check the two rules for the curly braces.
    1.3.4 We need to check if the stack is empty before popping off a "[".

```
aStack.createStack()
balancedSoFar = true
i = 0

while (balancedSoFar and I < length of aString)
{       ch = character at position i in aString
        ++i

        // push an open brace
```

```
        if (ch is '{')
                aStach.push('{')

        // close brace
        else if (ch is '}')
                if (!aStack.isEmpty())
                        aStack.pop()    // pop a matching open brace

                else                    // no matching open brace
                        balancedSoFar = false

        // ignore al characters other than braces
} // end while

if (balancedSoFar and aStack.isEmpty())
        aString has balanced braces
else
        aString does not have balanced braces
```

Figure 6-3 on page 294 shows what happens with a few examples:

Input string	Stack as Algorithm executes 1	2	3	4	
{a{b}c}	{ {	{ {	{ {		1. push "{" 2. push "{" 3. pop 4. pop Stack empty ⇒ balanced
{a{bc}	{ {	{ {	{ {		1. push "{" 2. push "{" 3. pop Stack not empty ⇒ not balanced
{ab}c}	}				1. push "{" 2. pop 3. pop Stack empty on last "}" ⇒ not balanced

1.4 One note, you can have a failure with the push operation if the stack is fixed sized and is full. You would need to have a try/catch combination to catch the error.

2 It is possible to have a solution without the stack: you could use a counter to accomplish the same task (i.e. increment counter with a "{" and decrement with a "}".

2.1 If counter goes below zero, it is not balanced or, when done, if the counter is not zero, it is not balanced).

2.2 However, using the stack for the balance curly braces is good practice.

3 Now let's try recognizing if a string is in a particular language.
3.1 For instance:

L = {w$w' : w may be an empty string of characters other than $, w' is reverse (w)}

3.2 Strings of AA, ABCCBA, and $ and in L but AB$AB and ABC$CB are not. T
3.3 his language is similar to the language of palindromes but with a $ in the middle.
3.4 Let's try a stack with this (we'll assume that there is only one "$"):

```
aStack.createStack()

// push the characters before $, that is, the characters in w, onto the stack
i = 0
ch = character at position i in aString
while (ch is not '$')
{       aStack.push(ch)
        ++i
        ch = character at position i in string
} // end while

// skip the $
++i

// match the reverse of w
inLanguage = true // assume string is in the language
while (inLanguage and I < length of aString)
{       try
        {       aStack.pop(stackTop)
                ch = character at position i in aString
                if (stackTop equals ch)
                        ++i  // characters match
                else
                        // top of stack is not ch (character do not match)
                        inLanguage = false // reject string
        } // end try
        catch StackException
        {       // pop failed, stack is empty (first half of string is shorter than second half)
                inLanguage = false
        } // end catch
} // end while
if (inLanguage and aStack.isEmpty())
        aString is in the language
else
        aString is not in the language
```

3.5 Note that both examples presented in this section depend on the operations of the stack and not on how they are implemented.