Problem Set 4 - Solution

Stack, Array List

---

1. Suppose that the `Stack` class consisted only of the three methods `push`, `pop`, and `isEmpty`:

```
public class Stack<T> {
   ...
   public Stack() { ... }
   public void push(T item) { ... }
   public T pop() throws NoSuchElementException { ... }
   public boolean isEmpty() { ... }
}
```

   Implement the following "client" method (i.e. *not* in the `Stack` class, but in the program that uses a stack):

```
    public static <T> int size(Stack<T> S) {
        // COMPLETE THIS METHOD
    }
```

   to return the number of items in a given stack `S`.

   Analyze this method for worst case big *O* running time, following these steps:

   - Identify the basic unit-time operations that contribute to the running time. You may assume that the constructor, as well as the `push`, `pop`, and `isEmpty` methods are all worst case *O(1)* running time.
   - Count the number of times each of these basic operations are executed in the worst case, and compute the total
   - Convert the total number of basic operations to big *O*

   SOLUTION

   Create another, temporary stack. Pop all items from input to temp stack, count items as you go. When all done, push items back from temp stack to input, and return count.

```
    public static <T> int size(Stack<T> S) {

        // COMPLETE THIS METHOD
        Stack<T> temp = new Stack<T>();
        int count=0;
        while (!S.isEmpty()) {
           temp.push(S.pop());
           count++;
        }
        while (!temp.isEmpty()) {
           S.push(temp.pop());
        }
        return count;
    }
```

   Note: There's no `try-catch` around the `S.pop()` and `temp.pop()` calls because we know there won't be an exception, since we only popping when the stack is not empty.

   Basic operators are `push`, `pop`, and `isEmpty`. (Constructor is only used once, so can be ignored since it is independent of the input stack size.) Each item in the stack is popped and pushed two times. With each push or pop, there is also a check for empty, and an additional check to terminate the loop. If the length of the input stack is *n*, the total units of time taken will be *2n* (pushes) + *2n* (pops), + *2n+2* (empty checks) = *6n+2*. This gives a big O time of O(*n*).

---

2. A postfix expression is an arithmetic expression in which the operator comes *after* the values (operands) on which it is applied. Here are some examples of expressions in their regular (infix) form, and their postfix equivalents:

```
Infix              Postfix
-------------------------
2                  2
2 + 3              2 3 +
2 * (3 + 4)        2 3 4 + *
2 * (3 - 4) / 5    2 3 4 - * 5 /
```

   Note that the postfix form does not ever need parentheses.

   Implement a method to evaluate a postfix expression. The expression is a string which contains either single-digit numbers (0--9), or the operators +, -, *, and /, and nothing else. There is exactly one space between every two characters. The string has no leading spaces and no trailing spaces. You may assume that the input expression is not empty, and is correctly formatted as above.

   You may find the following `Stack` class to be useful - assume the constuctor and methods are already implemented.

```
public class Stack<T> {
    public Stack() { ... }
    public push(T item) { ... }
    public T pop() throws NoSuchElementException { ... }
    public T peek() throws NoSuchElementException { ... }
    public boolean isEmpty() { ... }
    public void clear(T item) { ... }
    public int size (T item) { ... }
}
```

   }

   You may use the `Character.digit(char,10)` method to convert a character to the integer value it represents. For example, `Character('2',10)` returns the integer 2. (The parameter 10 stands for the "radix" or base of the decimal number system.)

   You may write helper methods (with full implementation) as necessary. You may not call any method that you have not implemented yourself.

```
    public static float postfixEvaluate(String expr) {
        /** COMPLETE THIS METHOD **/
    }
```

   SOLUTION:

```
public static float postfixEvaluate(String expr) {
   Stack<Float> stk = new Stack<Float>();
   for (int i=0; i < expr.length(); i++) {
       char ch = expr.charAt(i);
       if (ch == ' ') { continue; }
       if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
          float second = stk.pop();
          float first = stk.pop();
          switch (ch) {
          case '+': stk.push(first + second);
          case '-': stk.push(first - second);
          case '*': stk.push(first * second);
          case '/': stk.push(first / second);
          }
          continue;
       }
       stk.push(Character.digit(ch,10));
   }
   return stk.pop();
}
```

---

3. This question compares the space usage for two versions of a stack, one using a linked list in which each node holds a reference to an object and a pointer to the next node, and the other using the Java `ArrayList` (array cells holding references to objects). Suppose the stack holds 1000 objects at its peak usage. How many bytes of space are used (a) by the linked list implementation, and (b) by the `ArrayList` implementation, at peak usage? Use the following data:
   - A reference/pointer to an object uses 4 bytes of space.
   - The `ArrayList` starts with an initial capacity of 10, and doubles each time it is resized.
   - The linked list implementation keeps a "front" reference/pointer to the first node
   - Both implementations keep an integer "size" field (4 bytes)
   - The `ArrayList` implementation keeps an integer capacity field (4 bytes)

SOLUTION

- Linked list implementation: 1000 nodes, each with two fields/8 bytes of space for 8000 bytes. Plus a front reference, 4 bytes, and a size field, 4 bytes, so 8008 bytes in all.
- Array list implementation: 1280 references at 4 bytes apiece, plus a size field and a capacity field, for 5128 bytes.

---

4. Consider a smart array that automatically expands on demand. (Like the `java.util.ArrayList`.) It starts with an initial capacity of 50, and whenever it expands, it **adds 30 to the current capacity**. So, for example, at the 51st add, it expands to a capacity of 80.

   How many total units of work would be needed to add 300 items to this smart array? (Add appends to the end of the array.) Assume it takes one unit of work to write an item into an array location, and one unit of work to allocate a new array of any length, when expanding. You don't need to count anything else.

   SOLUTION

   This is the sequence of expansions, happening just before the listed add:

   ```
   51st add, 81st add, 111th add, ..., 291st add
           0         1              7
   ```

   (We will use the numbers under the expansions starting with 81st to come up with a series to sum.)

   The work can be broken down into three components:

   - The actual adds: there are 300 items to add, so 300 units in all
   - The expansions: there are 9 expansions, 1 unit of work per to allocate new array, for a total of 9 units
   - The copies: Every time a new array is allocated, all items in current array have to be copied, i.e. written into, the new array. We can build a series like this for these copies:

     ```
     Expansion seequence:
       51st add, 81st add, 111th add, ..., 291st add
                 0         1               7

     Copies:
       50      + 80      + 110        ...+ 290
     = 50      + 80      + (80+1*30) + (80+2*30) + ... + (80+7*30)
     = 50 + 80 + 80*7 + (1+2+3..+7)*30
     = 50 + 80 + 80*7 + (7*8/2)*30
     = 50 + 80 + 560 + 28*30
     = 1530
     ```

   - Grand total units of work: 300+9+1530=1839

---

5. Suppose you set up a smart array with an initial capacity of 5, with a *DOUBLING* of capacity every time there is a resize. What would be the **average** number of units of work per add, in the course of performing **100** adds? Assume the same work units as the previous exercise.

   SOLUTION

   We will expand the array 5 times: to 10, 20, 40, 80 and 160, one unit per expansion for a total of 5 units.

   Each time we expand, we have to copy the current length over. So cost for that will be 5+10+20+40+80 = 155.

   We will need to write 100 elements, which will cost 100 units.

   So total = 5 + 155 + 100 = 260. The average is 260/100 = 2.6 for each add.