Problem Set 4 - Solution

Stack, Array List, Queue

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 2*n* (pushes) + 2*n* (pops), + 2*n*+2 (empty checks) = 6*n*+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. **WORK OUT THE SOLUTION TO THIS PROBLEM ON PAPER, AND TURN IT IN AT RECITATION**

   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: To hold 1000 items, the ArrayList capacity grows like this: 10 -> 20 -> 40 -> 80 -> 160 -> 320 -> 640 -> 1280. So there are 1280 references at 4 bytes apiece (of which 280 will be null), so 1280*4 bytes, plus a size field and a capacity field, for 5128 bytes.

---

4. You are given the following `Queue` class:

```
public class Queue<T> {
   public Queue() { ... }
   public void enqueue(T item) { ... }
   public T dequeue() throws NoSuchElementException { ... }
   public boolean isEmpty() { ... }
   public int size() { ... }
}
```

   Complete the following *client* method (*not* a `Queue` class method) to implement the `peek` feature, using only the methods defined in the `Queue` class:

```
// returns the item at the front of the given queue, without
// removing it from the queue
public static <T> T peek(Queue<T> q)
throws NoSuchElementException {
   /** COMPLETE THIS METHOD **/
}
```

   Derive the worst case big *O* running time of your implementation. You may assume that the constructors and methods of the `Queue` class all have a worst case *O(1)* running time.

   SOLUTION

   1. Version 1, using temporary queue and `isEmpty()` method

```
// returns the item at the front of the given queue, without
// removing it from the queue
public static <T> T peek(Queue<T> q)
throws NoSuchElementException {
   /** COMPLETE THIS METHOD **/
   if (q.isEmpty()) {
       throw new NoSuchElementException("Queue Empty");
   }
   T result = q.dequeue();

   Queue<T> temp = new Queue<T>();
   temp.enqueue(result);

   while(!q.isEmpty()) {
      temp.enqueue(q.dequeue());
   }

   while(!temp.isEmpty()) {
      q.enqueue(temp.dequeue());
   }
   return result;
}
```

      Basic unit time operations are `enqueue`, `dequeue` and `isEmpty`. If the input queue is of size *n*, the total time is *2n* (enqueues) + *2n* (dequeues) + *2n+2* (empty

checks) = *6n+2*, which results in O(*n*) time.

2. Version 2, using `size()` method, no scratch queue needed

```
// returns the item at the front of the given queue, without
// removing it from the queue
public static <T> T peek(Queue<T> q)
throws NoSuchElementException {
   /** COMPLETE THIS METHOD **/
   if (q.isEmpty()) {
       throw new NoSuchElementException("Queue Empty");
   }
   T result = q.dequeue();
   q.enqueue(result);

   // dequeue an element and enqueue it again for (size-1) elements
   // if there was only 1 element, this loop will not execute
   for (int i=0; i < q.size()-1; i++) {
      q.enqueue(q.dequeue());
   }
   return result;
}
```

Basic unit time operations are `enqueue`, `dequeue`, `isEmpty()` and `size`. If the input queue is of size *n*, the total time is *1* (isEmpty) + *n* (enqueues) + *n* (dequeues) + *1* (size) = *2n+2*, which results in O(*n*) time.

---

5. * Suppose there is a long line of people at a check-out counter in a store. A new counter is opened, and people in the even positions (second, fourth, sixth, etc.) in the original line are directed to the new line. If a check-out counter line is modeled using a `Queue` class, we can implement this "even split" operation in this class.

   Assume that a `Queue` class is implemented using a CLL, with a `rear` field that refers to the last node in the queue CLL, and that the `Queue` class already contains the following constructors and methods:

```
public class Queue<T> {
   public Queue() { rear = null; }
   public void enqueue(T obj) { ... }
   public T dequeue() throws NoSuchElementException { ... }
   public boolean isEmpty() { ... }
   public int size() { ... }
}
```

   Implement an additional method in this class that would perform the even split:

```
// extract the even position items from this queue into
// the result queue, and delete them from this queue
public Queue<T> evenSplit() {
   /** COMPLETE THIS METHOD **/
}
```

   Derive the worst case big *O* running time of your implementationYou may assume that the constructors and existing methods of the `Queue` class all have a worst case *O(1)* running time.

   SOLUTION

```
// extract the even position items from this queue into
// the result queue, and delete them from this queue
public Queue<T> evenSplit() {
   /** COMPLETE THIS METHOD **/

   // Front of queue is at position 1, so we will extract 2nd, 4th, 6th, ...
   Queue<T> evenQueue = new Queue<T>();
   int originalSize = size();// size of this original queue

   // iterate once over each pair of queue elements
   for (int pos=2; pos <= originalSize; pos += 2) {
       // the first in a pair is recycled
       enqueue(dequeue());

       // the second in a pair goes to result queue
       evenQueue.enqueue(dequeue());
   }

   // if original size was an odd number, we need to
   // recycle one more time
   if ((originalSize % 2) == 1) {
       enqueue(dequeue());
   }

   return evenQueue;
}
```

   The basic operations are `enqueue` and `dequeue`. The `size()` method is only called once, at the beginning, and can be ignored. If the size of the queue is *n*, there are exactly *n* enqueues and *n* dequeues, for a total time of *2n*. Which then works out to *O(n)*