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`.

   Derive the worst case big *O* running time of the algorithm you used in your implementation. What are the dominant algorithmic operations you are counting towards the running time?

   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.

   Dominant operations are `push`, `pop`. (`isEmpty` is auxiliary, used only as a stopping condition. The 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. For a stack of size *n*, this will add up to *4n* pushes and pops, which 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. Consider a smart array that automatically expands on demand. (Like the `java.util.ArrayList`.) It starts with some given initial capacity of 100, and whenever it expands, it **adds 50 to the current capacity**. So, for example, at the 101st add, it expands to a capacity of 150.

   How many total units of work would be needed to add 1000 items to this smart 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.

   SOLUTION

   This is the sequence of adds and work:

   - First 100 adds: 100 units for writes
   - Expansion: 1 unit for allocation + 100 units to write from old to new
   - Next 50 adds: 50 units for writes
   - Expansion: 1 unit for allocation + 150 units to write from old to new
   - Next 50 adds: 50 units for writes
   - Expansion: 1 unit for allocation + 200 units to write from old to new
   - Next 50 adds: 50 units for writes

     .

     .

     .

   To add 1000 elements, 18 expansions of 50 required (100+18*50=1000)

   Units of work done: 100 + (1+100+50)+ (1+150+50)+ (1+200+50)+ ... (1+950+50)

---

4. 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.

---

5. 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 the algorithm that drives your implementation. What are the dominant algorithmic operations you are counting towards the 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;
}
```

   Dominant operations are `enqueue` and `dequeue`. Every item is enqueued twice and dequeued twice. For a queue of size *n*, this adds up to *4n* operations, which is 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;
    }
```

Dominant operations are `enqueue` and `dequeue`. For a queue of size *n*, there are *n* enqueques and dequeues each, for 2*n* operations, which gives O(*n*) time.

---

6. * 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 the algorithm that drives your algorithm. What are the dominant algorithmic operations you are counting towards the 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;
    }
```

Dominant operations are `enqueue` and `dequeue`. For a queue of size *n*, there are *n* enqueues and *n* dequeues, for a total of *2n* opertions. So *O(n)* time.