## Questions

1. Consider the following sequence of stack operations:

   `push(d), push(h), pop(), push(f), push(s), pop(), pop(), push(m).`

   (a) Assume the stack is initially empty, what is the sequence of popped values, and what is the final state of the stack? (Identify which end is the top of the stack.)

   (b) Suppose you were to replace the `push` and `pop` operations with `enqueue` and `dequeue` respectively. What would be the sequence of dequeued values, and what would be the final state of the queue? (Identify which end is the front of the queue.)

2. Use a stack to test for balanced parentheses, when scanning the following expressions. Your solution should show the state of the stack each time it is modified. The "state of the stack" must indicate which is the top element.

   Only consider the parentheses `[,],(,),{,}` . Ignore the variables and operators.

   (a) `[ a + { b / ( c - d ) + e / (f + g ) } - h ]`

   (b) `[ a { b + [ c ( d + e ) - f ] + g }`

3. Suppose you have a stack in which the values 1 through 5 must be pushed on the stack in that order, but that an item on the stack can be popped at any time. Give a sequence of push and pop operations such that the values are popped in the following order:

   (a) 2, 4, 5, 3, 1

   (b) 1, 5, 4, 2, 3

   (c) 1, 3, 5, 4, 2

   It might not be possible in each case.

4. (a) Suppose you have three stacks s1, s2, s2 with starting configuration shown on the left, and finishing condition shown on the right. Give a sequence of push and pop operations that take you from start to finish. For example, to pop the top element of `s2` and push it onto `s3`, you would write `s3.push( s2.pop())`.

```
              start                                   finish


       A                                                        A
       B                                                        B
       C                                                        D
       D                                                        C
      ---      ---      ---                       ---    ---   ---
       s1       s2       s3                        s1     s2    s3
```

   (b) What if the finish configuration on s3 was BDAC (with B on top) ?

5. Consider the following sequence of stack commands:

   `push(a), push(b), push(c), pop(), push(d), push(e), pop(), pop(), pop(), pop().`

   (a) What is the order in which the elements are popped ? (Give a list and indicate which was popped first.)

   (b) Change the position of the `pop()` commands in the above sequence so that the items are popped in the following order: `b,d,c,a,e`.

   You are *not* allowed to change the ordering of the `push` commands.

6. Assume you have a stack with operations: `push()`, `pop()`, `isEmpty()`. How would you use these stack operations to simulate a queue? In particular, how would you simulate operations `enqueue()` and `dequeue()`?

   Hint: use two stacks.

7. Assume you have a queue with operations: `enqueue()`, `dequeue()`, `isEmpty()`. How would you use the queue methods to simulate a stack ? In particular, how would you simulate `push()` and pop() ?

   Hint: use two queues.

8. [**added Sept. 26, 2012**]

   Consider a class that organizes a set of numbers. The class acts like a stack, in that it has `push` and `pop` operations, but it also has an operation `getMin()` that returns the smallest number in the set. Using two stacks, "implement" these three operations, such that each operations is done in *constant time*, i.e. independent of the number of elements in the stack.

## Answers

1. (a) Sequence of popped values: h,s,f. State of stack (from top to bottom): m, d
   (b) Sequence of dequeued values: d,h,f. State of queue (from front to back): s,m.

2. (a)
```
-       means empty stack
[
[{
[{(     TOP OF STACK IS ON THE RIGHT
[{
[{(
[{
[
-       empty stack, so brackets match
```
   (b)
```
-
[
[{
[{[        TOP OF STACK IS ON THE RIGHT
[{[(
[{[
[{
[          stack not empty, so brackets don't match
```

3.
```
24531           15423               13542

push 1          push 1              push 1
push 2          pop                 pop
pop             push 2              push 2
push 3          push 3              push 3
push 4          push 4              pop
pop             push 5              push 4
push 5          pop                 push 5
pop             pop                 pop
pop              x                  pop
pop            (not possible)       pop
```

4. (a)
```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
s3.push( s2.pop() )
```
   (b)
```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s1.push( s2.pop() )
s3.push( s2.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
```

5. (a) c (popped first), e, d, b, a
   (b) push(a), push(b), pop(), push(c), push(d), pop(), pop(), pop() push(e), pop()

6. <u>**Solution 1**</u>

The first solution is to implement `enqueue` by just pushing the new element on top of the stack. In this solution, the bottom of the stack is the front of the queue and the top of the stack is the back of the queue. How can we implement `dequeue`, that is, how do we remove the bottom element of the stack?

The idea is to use two stacks *s* and *tmpS*. We first pop all items from the original stack *s*, pushing each popped element directly onto the second stack *tmpS*. We then pop the top element of the second stack (which is the oldest element, hence it is the element to be dequeued). Finally, we refill the stack *s* by popping all elements from the second stack, pushing each back on to the first stack.

```
ALGORITHM: enqueue (using a stack operations to simulate the queue)
INPUT:  a stack s and new element e\\
OUTPUT:  the stack s with the new element e at the top \\

enqueue(e){
  s.push(e)
}

ALGORITHM: dequeue (using a stack operations to simulate the queue)
INPUT:   a stack s
OUTPUT:  the stack s with the bottom  element removed

dequeue(){
  tmpS <-  new empty stack
  while !s.isEmpty(){
    tmpS.push( s.pop() )
  }
  returnValue <- tmpS.pop()              //  the dequeued element
  while  !(tmpS.isEmpty()){
    s.push( tmpS.pop() )
  }
  return  returnValue
}
```

**Solution 2**

Here we let `dequeue` be simple and just pop the stack. For this to work, the stack needs to store the elements such that the oldest element (front of queue) is on top of the stack and the most recently added element (back of queue) is at the bottom of the stack.

```
ALGORITHM: dequeue (using a stack operations to simulate the queue)
INPUT:   a stack s with the oldest element on top
OUTPUT:  the oldest element (and the stack s should no longer contain
         that element)

dequeue(){
  return s.pop()
}
```

With this solution, `enqueue(e)` needs to do the heavy lifting: `enqueue` uses a temporary stack to invert the order of elements currently in the stack, so that the newest element is on top of this temporary stack. Then it pushes the new element on top of the temporary stack. Then it inverts the stack again i.e. recreates the original stack, but now the the newest element which was added is on the bottom.

```
ALGORITHM: enqueue (using a stack operations to simulate the queue)
INPUT:  a stack s with the oldest queue element on top, and a new
        element e to be enqueued
OUTPUT: the stack s with e inserted at the bottom

enqueue(e){
  tmp <- new empty stack
  while  ! (s.isEmpty()){
    tmpS.push( s.pop() )
  }
  tmpS.push(e)
  while ! (tmpS.isEmpty()){
    s.push( tmpS.pop())
  }
}
```

7. The concepts are similar to the previous question.

**Solution 1**

```
ALGORITHM:  push  (using queue operations enqueue, dequeue, isEmpty)
INPUT:  a queue and a new element e
OUTPUT:  the queue q with the newest element e added

push(e){
  q.enqueue(e)
}


ALGORITHM:  pop  (using queue operations enqueue, dequeue, isEmpty)
INPUT:   queue q
OUTPUT:  the newest element (and the queue q should no longer contain
         this element)

pop(){
  tmpQ <-  new empty queue
  while   ! (q.isEmpty()){
    tmpE  <-   q.dequeue()
    if   ! (q.isEmpty())
      tmpQ.enqueue( tmpE )
    else
      while   !( tmpQ.isEmpty()){
         q.enqueue( tmpQ.dequeue())
      }
    return tmpE
}
```

**Solution 2**

Here the idea is similar, but now push does the heavy lifting.

```
ALGORITHM: pop (using queue operations enqueue, dequeue, isEmpty )
INPUT:   a queue q
OUTPUT:  the newest element in q (and the queue q no longer contains
         that element)
```

```
pop(){
  q.dequeue()
}


ALGORITHM: push (using queue operations enqueue, dequeue, isEmpty )
INPUT:   a queue q and new element e to be added
OUTPUT:  the queue q with the new element added at the front of the
         queue (so that pop will return it)

push(e){
  make a new empty queue qTmp
  qTmp.enqueue(e)
  while !q.isEmpty(){
     qTmp.enqueue( q.dequeue() )
  }
  q <- qTmp
  return qTmp
}
```

8. Use two stacks. The first one `s1` is a regular stack and holds all the elements. The second one `s2` is handled such that it always has a copy of the current minimum element on top.

   Whenever we push an element E onto `s1`, we also check the top of `s2` to see if it is less than E. We push E onto S2 only if E is less or equal to the top element of `s2`. Note that the elements on `s2` are non-increasing (i.e. never increase) if one were to scane from the bottom to the top of the stack.

   Whenever we pop an element, we pop stack `s1`. If this element is the same as the one on top of `s2`, then we pop it off S2 as well.