CS 61B
Spring 2020

Small Group Tutoring
Section 6: Sequences Solutions

Worksheet 8

1 When am I useful Senpai?

Based on the description, choose the data structure which would best suit our purposes. Choose from:
A - arrays, B - linkedlists, C - stacks, D - queues (excluding dequeue's cause they're too OP).

1. Keeping track of which customer in a line came first.
Solution: D. Queues have a first in first out (FIFO) ordering.

2. We will expect many inserts and deletes on some dataset, but not too many searches and lookups.
Solution: B. Linked Lists aren't great for searching, but they're quite fast for insert and delete operations.

3. We gather a lot of data of a fixed length that will remain relatively unchanged overtime, but we access its contents very frequently.
Solution: A. Arrays support constant time access to an element (by index).

4. Maintaining a history of the last actions on Word in case I need to undo something.
Solution: C. Stacks support a first in last out (FILO) ordering.

2 Reverse Me

Assume that we have some MyIntQueue class, with API :

```
boolean isEmpty() //returns true if the queue is empty
void enqueue(int item) //adds item to the back of the queue
int dequeue() //removes the item at the front of the queue
int size() //returns the size of the queue
```

We have some Stack API as follows:

```
boolean isEmpty() //returns true if the stack is empty
void push(int item) //adds item to the top of the stack
int pop() //removes the item at the top of the stack
int size() //returns the size of the stack
```

Fill in the method which takes in some `MyIntQueue q`, and reverses its elements using a `Stack`.
Solution: Empty the queue into the stack and then push the elements back into the queue in reverse order by the LIFO property of the stack.

```
private static void reverse(MyIntQueue q) {
    Stack s = new Stack();
    while (!q.isEmpty()) {
        s.push(q.dequeue());
    }
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }
}
```

3 Pseudo Stack

Implement a stack's `push` and `pop` methods using Queues. We have the same `MyIntQueue` API as in the previous question.

Solution: For first solution, q1 will hold top element of the stack at its end. For alternative solution, basically reversing q1 as items are pushed.

```
public class MyIntStack {
    MyIntQueue q1 = new MyIntQueue();
    MyIntQueue q2 = new MyIntQueue();
    public boolean isEmpty() {
        //Implementation not shown
    }
    public int size() {
        //Implementation not shown
    }
    public void push(int item) {
        q1.enqueue(item);
    }
    public int pop() {
        while (q1.size() > 1) {
            q2.enqueue(q1.dequeue());
        }
        int temp = q1.dequeue();
        MyIntQueue tempQ = q1;
        q1 = q2;
        q2 = tempQ;
        return temp;
    }

    //alt solution below: slower push, but faster pop than first solution
    public void push(int item) {
        q2.enqueue(item);
        while (!q1.isEmpty()) {
            q2.enqueue(q1.dequeue());
        }
        MyIntQueue tempQ = q1;
        q1 = q2;
        q2 = tempQ;
    }
    public int pop() {
        if (!q1.isEmpty()) {
            return q1.dequeue();
        } else {
            //throw an error, which we shouldn't worry about
        }
    }
}
```

4 A Balancing Act

Given a string *str*, containing just the characters `(`, `)`, {, }, `[`, and `]`, implement a method `hasValidParens` which determines if the string is valid.

The brackets must close in the correct order so "`()`", "`()`{}", and "`[()]`" are all valid, but "`(`", "`({)`}", and "`[(`" are not.

You may refer to the Stack API from problem 2 and use the getRightParen method provided below.

Solution: The idea is to keep track of which closing parentheses we are looking for in the order that they are needed to correctly match up to their corresponding opening parentheses. Therefore, we use a stack to accommodate for the nested nature of parentheses ordering.

```
private static boolean hasValidParens(String str) {
    Stack s = new Stack();
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c == '{' || c == '(' || c == '[') {
            s.push(getRightParen(c));
        } else {
            if (s.isEmpty()) {
                return false;
            }
            if (c != s.pop()){
                return false;
            }
        }
    }
    return s.isEmpty();
}


/**
    The method getRightParen takes in the left parenthesis
    and returns the corresponding right parenthesis.
**/
private static char getRightParen(char leftParen){
    if (leftParen == '(') {
        return ')';
    } else if (leftParen == '{') {
        return '}';
    } else if (leftParen == '[') {
        return ']';
    } else {
        //not one of the valid parenthesis characters
        throw new IllegalArgumentException();
    }
}
```