

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 7 – Stacks and Queues

(Week 9, starting 14 March 2016)

1. Java API Stack and Queue

In the following program, we create instances of the API **Stack/LinkedList** classes and use some of their behavior. **Draw diagrams** to represent the contents of s1, s2 and q at each step.

```
public static void main(String[] args) {
    Queue<Integer> q = new LinkedList<Integer>();
    Stack<Integer> s1 = new Stack<Integer>();
    Stack<Integer> s2 = new Stack<Integer>();

    // Draw contents after these 3 statements
    s1.push(new Integer(3));
    s1.push(new Integer(2));
    s1.push(new Integer(1));

    // Draw contents after each iteration
    while (!s1.empty()) {
        s2.push(s1.pop());
        if (!s1.isEmpty()) s2.push(s1.peek());
        q.offer(s2.peak());
    }

    // Draw contents after this statement
    s1.push(q.remove());

    // Print out the contents of the stacks and queue
    String output = "";
    while (s1.size() > 0)
        output = s1.pop() + " " + output;
    System.out.println("S1 : " + output + "(top)");

    output = "";
    while (s2.size() > 0)
        output = s2.pop() + " " + output;
    System.out.println("S2 : " + output + "(top)");

    output = "";
    while (q.size() > 0)
        output = output + " " + q.remove();
    System.out.println("Q : (head)" + output);
}
```

Find out for yourself, why does Queue have:

poll() and remove()

peek(), peekFirst() and peekLast()

<https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>

2. Waiting Queue

In our day-to-day life, it is common to wait in a queue/line, be it buying a hamburger at McDonald's, or waiting to pay for accommodation at a residence. People join the queue sequentially, and are served in a first-come-first-served manner. However, using pure queue operations is not enough, as people in the queue might grow impatient and leave.

In this exercise, you want to implement a **WaitingQueue** which contains the names of the people in the queue. In addition to the standard queue behavior, it allows people to leave at any time. An **array** is used as the **underlying data structure**. You may assume that the names of people in the queue are unique.

- (a) How does this data structure differ from the **Queue** learnt in lectures?
- (b) If the **WaitingQueue** class is fully and correctly implemented, what is the output of the main() method shown below?

```
public static void main(String[] args) {
    WaitingQueue q = new WaitingQueue();

    q.addAPerson("Person 1");
    q.addAPerson("Person 2");
    q.addAPerson("Person 3");
    q.addAPerson("Person 4");
    q.addAPerson("Person 5");
    q.addAPerson("Person 6");
    q.addAPerson("Person 7");
    q.addAPerson("Person 8");

    System.out.println(q.serveNextPerson());
    System.out.println(q.serveNextPerson());

    boolean b1 = q.leave("Person 2");
    boolean b2 = q.leave("Person 3");
    boolean b3 = q.leave("Person 4");

    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);

    while (!q.isEmpty())
        System.out.println(q.serveNextPerson());
}
```

- (c) Complete the implementation of **WaitingQueue** using the code snippet below. There will be at most 9 people in the queue. Nobody will be able to join the queue when it is full.

```
public class WaitingQueue {
    private String[] waitingHere;
    private int front; // "Leave a gap" when array is full
    private int back;  // back is the index AFTER last element

    private static final int ARR_LENGTH = 10;

    public WaitingQueue() {
        waitingHere = new String[ARR_LENGTH];
    }

    public boolean isEmpty() {
        return false; // TODO: Implement isEmpty method
    }

    // Returns true if Person is successfully added
    public boolean addAPerson(String newPerson) {
        return false; // TODO: Offer to back of queue
    }

    public String serveNextPerson() {
        return null; // TODO: Remove from front of queue
    }

    // Returns true if someone is removed from the queue
    public boolean leave(String personName) {
        return false; // TODO: Implement leaving
    }
}
```

- (d) Think of at least two other ways to implement the `leave()` method, and comment on whether these ways are efficient. You do NOT need to implement them.

3. Expression Evaluation

In the **Lisp** programming language, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expressions are enclosed in parentheses. There is only one operator in a pair of parentheses. The operators behave as follows:

- `( + a b c )` returns the sum of all the operands, and `( + )` returns 0.
- `( - a b c )` returns $a - b - c - \dots$ and `( - a )` returns $0 - a$.
The minus operator must have at least one operand.
- `( * a b c )` returns the product of all the operands, and `( * )` returns 1.
- `( / a b c )` returns $a/b/c/\dots$ and `( / a )` returns $1/a$.
The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:

`( + ( - 6 ) ( * 2 3 4 ) )`

The expression is evaluated successively as follows:

```
( + -6 ( * 2 3 4 ) )  
( + -6 24 )  
18.0
```

Design and implement a program that uses up to 2 stacks to evaluate a legal Lisp expression composed of the four basic operators, integer operands, and parentheses. The expression is well-formed (i.e. no syntax error), there will always be a space between 2 tokens, and we will not divide by zero.

=p

Trace through an algorithm
Why does this algorithm work?
What data structure is needed?
How does this data structure help?