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.peek());
    }

    // 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:

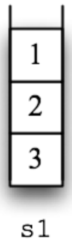
`poll()` and `remove()`

`peek()`, `peekFirst()` and `peekLast()`

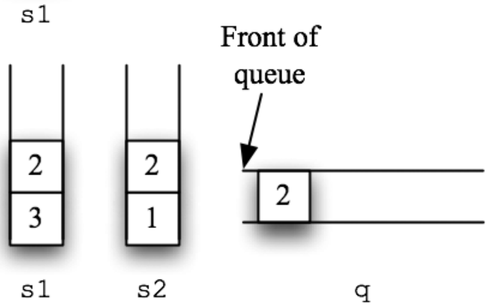
https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

Answer

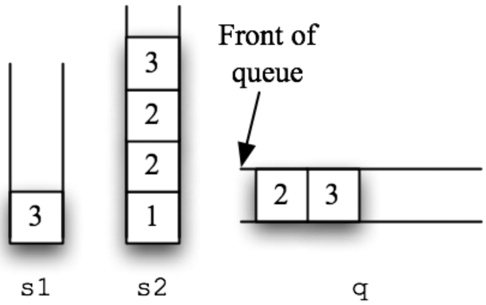
Step 1:



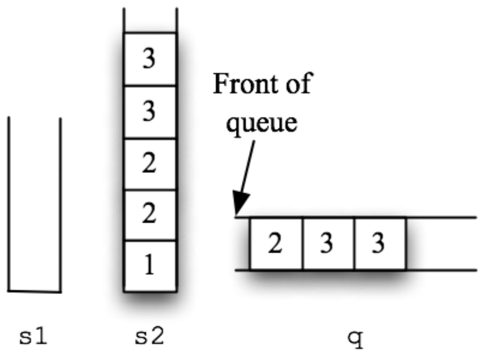
Step 2:



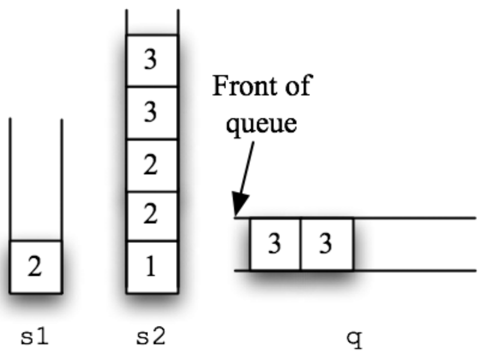
Step 3:



Step 4:



Step 5:



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.

```java
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.

Answer

(a)

WaitingQueue is not a pure Queue data structure, because leave() treats the people in the queue as a Collection. Of course, addAPerson() and serveNextPerson() preserve the FIFO property that we would expect a queue to have.

(b)

```
Person 1
Person 2
false
true
true
Person 5
Person 6
Person 7
Person 8
```

(c)

```java
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 front == back;
    }

    // Returns true if Person is successfully added
    public boolean addAPerson(String newPerson) {
        if ((back + 1) % ARR_LENGTH == front) // array full
            return false;
        waitingHere[back] = newPerson;
        back = (back + 1) % ARR_LENGTH; // circular array behavior
        return true;
    }

    public String serveNextPerson() {
        if (isEmpty()) // empty queue
            return null;
        String firstPerson = waitingHere[front];
        waitingHere[front] = null; // optional
        front = (front + 1) % ARR_LENGTH;
        return firstPerson;
    }

    // Returns true if someone is removed from the queue
```

```
    public boolean leave(String personName) {

        // find first matching person
        boolean found = false;
        int position = front;
        while (position != back) { // pos may NOT be < back !!!
            if (waitingHere[position].equals(personName)) {
                waitingHere[position] = null; // optional
                found = true;
                break;
            }
            position = (position + 1) % ARR_LENGTH;
        }

        if (!found)
            return false;

        // left shift elements
        position = (position + 1) % ARR_LENGTH;
        while (position != back) {
            if (position != 0)
                waitingHere[position-1] = waitingHere[position];
            else
                waitingHere[ARR_LENGTH - 1] = waitingHere[0];
            position = (position + 1) % ARR_LENGTH;
        }

        // decrement back
        back = (back + ARR_LENGTH - 1) % ARR_LENGTH;
        return true;
    }
}
```

(d)

Assume there are N people in the queue.

Current implementation - Left-shift remaining elements

All N elements in the queue are always accessed. This causes **leave()** to be **inefficient**, while allowing the **two queue operations to remain efficient**. Only one element is accessed in addAPerson() and serveNextPerson().

Lazy deletion

Each element in the queue has a boolean flag indicating whether a person has left the queue, or not. If we create a Person class, let the flag be one of its attributes, then we can efficiently indicate that a person has left. **If we already have a reference** to the matching Person object, we only need to access that **one element**.

However, **serveNextPerson() will suffer**, as we now have to access more than one element in order to clear the deleted objects at the front of the queue.

Maintain separate collection of people who want to leave

We can store the names of the people who want to leave the queue in a separate data structure. When a person is served, the collection is searched to find a matching person. We will learn how to implement a collection that allows elements to be added and searched efficiently later in the semester.

The efficiency of leave() is improved as compared to the current implementation, but the method **requires more space**. **serveNextPerson()** will be less efficient too, as we may have to remove more than one element before we find someone who has not already left the queue. Meanwhile, the person already left still takes up one position in this queue before it served, which reduces the valid length of the queue.

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 - … 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/… 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.

Answer

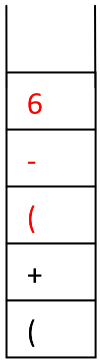
One algorithm uses two stacks. The first is used to store the tokens read from the expression one by one until the operator ")". The second stack is used to perform a simple operation on the operands in the innermost expression already in the first stack. The tokens are pushed into the second stack in reverse order. Therefore, tokens from the second stack are popped in the order of input. The calculated result is then pushed back into the first stack.

An example is given in the next few pages:

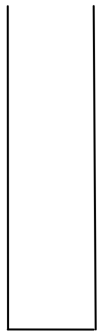
Answer

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

1. The main stack pushes the tokens one by one until it reads “)”.

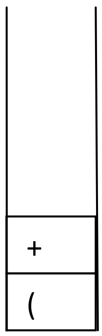


The main stack

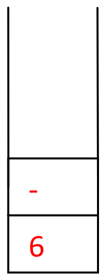


The temporary stack for simple calculation

2. The main stack pops out the tokens and push them one by one to the temporary stack.

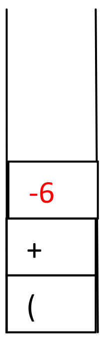


The main stack

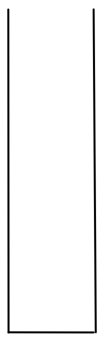


The temporary stack for simple calculation

3. The temporary stack pushes back the result after calculation.



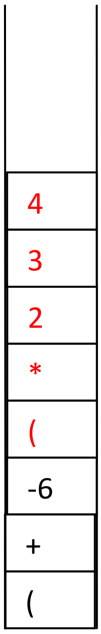
The main stack



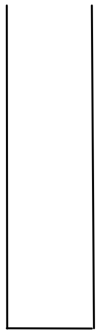
The temporary stack for simple calculation

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

4. The main stack continues to push in the tokens from the expression until it reads “)”.

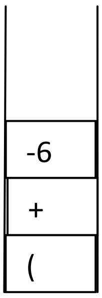


The main stack

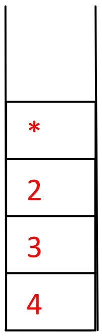


The temporary stack for simple calculation

5. The main stack pops out the tokens and push them one by one to the temporary stack.

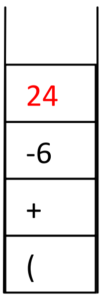


The main stack

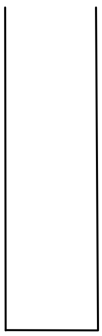


The temporary stack for simple calculation

6. The temporary stack pushes back the result after calculation.



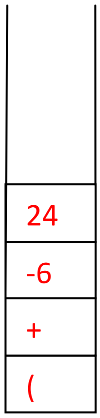
The main stack



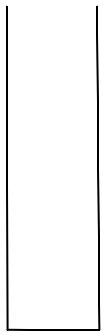
The temporary stack for simple calculation

Expression (+ -6 24)

7. The main stack continues to push in the tokens from the expression until it reads “)”.

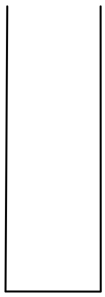


The main stack

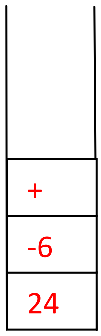


The temporary stack for simple calculation

8. The main stack pops out the tokens and push them one by one to the temporary stack.

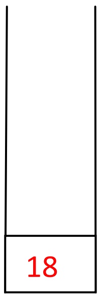


The main stack

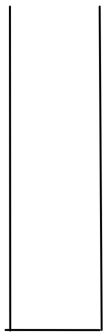


The temporary stack for simple calculation

9. The temporary stack pushes back the result after calculation. When it is at the end of expression, the final result is stored in the main stack.



The main stack



The temporary stack for simple calculation

```java
// Evaluate entire well-formed Lisp expression
public double evaluateLispExpr(String input) {
     Stack<String> tokens = new Stack<String>(); // outer stack
     Scanner sc = new Scanner(input);

     while (sc.hasNext()){
          String currentToken = sc.next();
          if (currentToken.equals(")")){
               Stack<String> expr = new Stack<String>(); // inner
               while (!tokens.peek().equals("("))
                    expr.push(tokens.pop());
               tokens.pop(); // remove "("
               tokens.push("" + performOperation(expr));
          } else {
               tokens.push(currentToken);
          }
     }
     return Double.parseDouble(tokens.peek());
}

// Evaluate simple expression of form: operator operand1 operand2...
private double performOperation(Stack<String> s){
   double result = 0;
   char operator = s.pop().charAt(0); // pop() here returns String
   switch (operator){
      case '+':
         result = 0;
         while (!s.empty())
            result = result + Double.parseDouble(s.pop());
         return result;
      case '-':
         if (s.size() == 1)
            return 0 - Double.parseDouble(s.pop());
         result = Double.parseDouble(s.pop());
         while (!s.empty())
            result = result - Double.parseDouble(s.pop());
         return result;
      case '*':
         result = 1;
         while(!s.empty())
            result = result * Double.parseDouble(s.pop());
         return result;
      case '/':
         if (s.size() == 1)
            return 1 / Double.parseDouble(s.pop());
         result = Double.parseDouble(s.pop());
         while (!s.empty())
            result = result / Double.parseDouble(s.pop());
         return result;
   } // switch-case: don't forget to "break;" otherwise
   return result; // should not happen =X
}
```

We have just used Stack<String> to simplify the algorithm within evaluateLispExpr(). It is a better design for performOperation() to have parameters (char operator, Stack<Double> operands) instead, since the first token in an operation has to be an operator, and the rest of the operands have to be real numbers.

=þ

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