Ben Durao

CS253

Homework 3

Code

```
/* This is the master class that runs the tests of the program. It enqueues a test expression (infix) to the Infix
*Queue and runs  that translate the expression to postfix form (postfix queue) and then evaluates the
*expression on the value stack.
 * @author (ben durao)
 * @version (10/30/16) */
public class main
{
   public static void main(String[] args)
   {
      InfixQueue iq = new InfixQueue();
      PostfixQueue pq = new PostfixQueue();
      OperatorStack os = new OperatorStack();
      String infix = "5*3+(6-8/2)#";
      String postfix = "";
      String match = "53*682/-+";
      String token = "";

      // Queue the expression
      for(int j=0; j < infix.length(); j++)
      {
         iq.enqueue( Character.toString( infix.charAt(j) ) );
         System.out.println("Enqueued to IQ: " + Character.toString( infix.charAt(j) ));
      }
      // init the operator stack
      os.push("#");
      System.out.println("Pushed " + os.ontop() + " to stack.");

      token = iq.front();
      // dequeue tokens until it is #
      while(!(token.equals("#")))
      {
         token = iq.dequeue();
         System.out.println("\nDequeued " + token + " from IQ");
         // is token an operand? [+ enqueue the operand on postfix queue]
         if(!token.equals("*") && !token.equals("/") && !token.equals("+") && !token.equals("-") &&
!token.equals("(") && !token.equals(")") && !token.equals("#"))
         {
            pq.enqueue(token);
            System.out.println("Enqueued " + token + " to PQ");
         }
         else if(token.equals("#"))
         {
            while(!os.ontop().equals("#"))
            {
               if(os.ontop().equals("("))
               {
```

```
            os.pop();
          }
          else {
            System.out.println("Enqueued " + os.ontop() + " on PQ");
            pq.enqueue( os.pop() );
          }
        }
      }
      else
      {
        if( token.equals(")") )
        {
          String operator = os.ontop();
          // pop entries from operator stack and enqueue them until matching left parenthesis
          while( !operator.equals("(") )
          {
            // disregard the parenthesis when queueing to postfix
            operator = os.pop();
            System.out.println("Popped " + operator + " off the stack.");
            if(!operator.equals( ")") && !operator.equals("(") )
            {
              pq.enqueue(operator);
              System.out.println("Enqueued " + operator + " to the PQ");
            }
            if(!os.empty())
              operator = os.ontop();
          }
        }
        else
        {
          // another operator
          if(os.size() > 1)
          {
            String operator = os.ontop();
            if(os.stackPriority(operator) >= iq.infixPriority(token) && !operator.equals("("))
            {
              operator = os.pop();
              System.out.println("Popped " + operator + " off the stack.");
              pq.enqueue(operator);
              System.out.println("Enqueued " + operator + " to PQ");
            }
          }
          os.push(token);
          System.out.println("Pushed " + token + " to stack.");
        }
      }
    }
    System.out.println();

    // Evaluate PostFix Expression
    token =" ";
    OperatorStack value = new OperatorStack();
    value.push("#");
```

```
      while(!pq.empty())
      {
         token = pq.dequeue();
         postfix += token;
         if(token.equals("*"))
         {
            value.push( Integer.toString( ( Integer.parseInt(value.pop()) * Integer.parseInt(value.pop()) ) ) );
            System.out.println("Pushed on value: " + value.ontop());
         }
         else if(token.equals("/"))
         {
            int denom = Integer.parseInt(value.pop());
            int numerator = Integer.parseInt(value.pop());
            value.push( Integer.toString( numerator / denom ) );
            System.out.println("Pushed on value: " + value.ontop());
         }
         else if(token.equals("+"))
         {
            value.push( Integer.toString( ( Integer.parseInt(value.pop()) + Integer.parseInt(value.pop()) ) ) );
            System.out.println("Pushed on value: " + value.ontop());
         }
         else if(token.equals("-"))
         {
            int subtractor = Integer.parseInt(value.pop());
            int prime = Integer.parseInt(value.pop());
            value.push( Integer.toString( prime - subtractor ) );
            System.out.println("Pushed on value: " + value.ontop());
         }
         else
         {
            value.push(token);
            System.out.println("Pushed on value: " + value.ontop());
         }
      }
      System.out.println("\nPostfix Expression = " + value.pop());
      System.out.println("Postfix = " + postfix);
      System.out.println("It should look like this = " + match);

   }
}
```

```
public interface LLStack {
  public void push (String c);
  public String pop();
  public int size();
  public boolean empty();
  public String ontop();
}
```

```
public interface LLQueue {
  public void enqueue (String item);
  public String dequeue();
  public int size();
  public boolean empty();
  public String front();
}
```

```
/**
 * Node is a piece of data that is arranged in collections.
 *
 * @author (ben durao)
 * @version (10/30/16)
 */
public class Node
{
   private String data;
   private Node next;

   public Node () { this("", null);   }

   public Node (String d) {  data = d;   }

   public Node (String d, Node n) {
      data = d;
      next = n;
   }

   public void setData (String newData){   data = newData;     }

   public void setNext (Node newNext)      next = newNext;   }

   public String getData ()  {    return data;   }

   public Node getNext () {     return next;   }

   public void displayNode () {  System.out.print(data);   }
}
```

```
/**
 * Operator stack is a restricted list with all additions (push) and removals (pop) happen at the top of the stack
 * It is used in main first to hold the operators during the transformation from infix to postfix.
* Then it is used to evaluate the postfix expression
 * @author (ben durao)
 * @version (10/30/16)
 */
public class OperatorStack implements LLStack
{
   private Node top;
   private int size;

   public OperatorStack() {
      top = null;
      size = 0;
   }
   public boolean empty () { return (top == null);  }
   public int size () {   return size;  }
   public void push (String c) {
      Node newNode = new Node ();
      newNode.setData(c);
      newNode.setNext(top);
      top = newNode;
      size++;
   }
   public String pop () {
      String c;
      c = top.getData();
      top = top.getNext();
      size--;
      return c;
   }
   public String ontop () {
      String c = pop();
      push(c);
      return c;
   }
   public int stackPriority(String c)
   {
      int value = 0;
      if(c.equals("*"))
         value = 2;
      else if(c.equals("/"))
         value = 2;
      else if(c.equals("+"))
         value = 1;
      else if(c.equals("-"))
         value = 1;
      else if(c.equals("("))
         value = 3;
      else if(c.equals("#"))
         value = 0;
      return value; }    }
```

```
/**
 * InfixQueue is a restricted list with removals at the front(dequeue) and additions at the rear (enqueue)
 * It is used in main to hold the infix expression.
 * @author (ben durao)
 * @version (10/30/16)
 */
public class InfixQueue implements LLQueue
{
   private int size;
   private Node front;
   private Node rear;

   public InfixQueue() {
      size = 0;
      front = null;
      rear = null;
   }
   public boolean empty () {  return (size == 0); }
   public int size() {  return size; }
   public void enqueue (String c) {
      Node newNode = new Node ();
      newNode.setData(c);
      newNode.setNext(null);
      if (this.empty())
         front = newNode;
      else
         rear.setNext(newNode);
      rear = newNode;
      size++;
   }
   public String dequeue() {
      String c;
      c = front.getData();
      front = front.getNext();
      size--;
      if (this.empty())
         rear = null;
      return c;
   }
   public String front () {  return front.getData();  }
   public int infixPriority(String c)
   {
      int value = 0;
      if(c.equals("*"))
         value = 2;
      else if(c.equals("/"))
         value = 2;
      else if(c.equals("+"))
         value = 1;
      else if(c.equals("-"))
         value = 1;
      else if(c.equals("("))
         value = 3;
```

```
        else if(c.equals(")"))
            value = 0;
        else if(c.equals("#"))
            value = 0;
        return value;
    }
}
```

```
/**
 * Postfix Queue is a restricted list with removals at the front (dequeue) and additions to the rear (enqueue)
 * It is used in main to hold the postfix expression.
 * @author (ben durao)
 * @version (10/30/16)
 */
public class PostfixQueue implements LLQueue
{
   private int size;
   private Node front;
   private Node rear;

   public PostfixQueue() {
      size = 0;
      front = null;
      rear = null;
   }

   public boolean empty () {
      return (size == 0);
   }

   public int size () {
      return size;
   }

   public void enqueue (String c) {
      Node newNode = new Node ();
      newNode.setData(c);
      newNode.setNext(null);
      if (this.empty())
         front = newNode;
      else
         rear.setNext(newNode);
      rear = newNode;
      size++;
   }

   public String dequeue () {
      String c;
      c = front.getData();
      front = front.getNext();
      size--;
      if (this.empty())
         rear = null;
      return c;
   }

   public String front () {
      return front.getData();
   }
}
```

Program Run

```
Enqueued to IQ: 5
Enqueued to IQ: *
Enqueued to IQ: 3
Enqueued to IQ: +
Enqueued to IQ: (
Enqueued to IQ: 6
Enqueued to IQ: -
Enqueued to IQ: 8
Enqueued to IQ: /
Enqueued to IQ: 2
Enqueued to IQ: )
Enqueued to IQ: #
Pushed # to stack.

Dequeued 5 from IQ
Enqueued 5 to PQ

Dequeued * from IQ
Pushed * to stack.

Dequeued 3 from IQ
Enqueued 3 to PQ

Dequeued + from IQ
Popped * off the stack.
Enqueued * to PQ
Pushed + to stack.

Dequeued ( from IQ
Pushed ( to stack.

Dequeued 6 from IQ
Enqueued 6 to PQ

Dequeued - from IQ
Pushed - to stack.

Dequeued 8 from IQ
Enqueued 8 to PQ
```

```
Dequeued / from IQ
Pushed / to stack.

Dequeued 2 from IQ
Enqueued 2 to PQ

Dequeued ) from IQ
Popped / off the stack.
Enqueued / to the PQ
Popped - off the stack.
Enqueued - to the PQ

Dequeued # from IQ
Enqueued + on PQ

Pushed on value: 5
Pushed on value: 3
Pushed on value: 15
Pushed on value: 6
Pushed on value: 8
Pushed on value: 2
Pushed on value: 4
Pushed on value: 2
Pushed on value: 17

Postfix Expression = 17
Postfix = 53*682/-+
It should look like this = 53*682/-+
```

Data Structures:

Postfix Queue is a restricted list with removals at the front (dequeue) and additions to the rear (enqueue). It is used in main to hold the postfix expression.

InfixQueue is a restricted list with removals at the front(dequeue) and additions at the rear (enqueue). It is used in main to hold the infix expression.

Operator stack is a restricted list with all additions (push) and removals (pop) happen at the top of the stack. It is used in main first to hold the operators during the transformation from infix to postfix. Then it is used to evaluate the postfix expression.

Node is a piece of data that is arranged in collections with pointers to its neighbor.

## Problem # 2

Postorder – Recursively checks down is left child subtree until it reaches a spot where there is no left node. It then collects data from all right nodes until there are no more right children. It then climbs back up the tree by collecting the parent root node.

Inorder – Collects the leftmost subtree then its parent node before grabbing the right children. Which is why E is collected before D in the example. It then recursively works back up the tree to the root before collect the nodes in the right subtree.