```
/**
 * This is running an infix to postifx converter on the given input file and
 * printing the output I affirm that I have carried out the attached academic
 * endeavors with full academic honesty, in accordance with the Union College
 * Honor Code and the course syllabus.
 *
 * @author xavier
 *
 */
public class Client {

      /**
       * Just runs the converter
       */
      public static void main(String[] args) {

            Converter con = new Converter("src/input.txt");

            // Prints the output
            System.out.println(con.convert());

      }

}
```

```
/**
 * This is an infix- postifx converter that takes a file location with semicolon
 * deliminated equations and converts it to the postfix format.
 *
 * @author xavier
 *
 */
public class Converter {
      private FileReader r;

      /**
       * Constructs a new converter
       *
       * @param location
       *            is the location of the file to convert
       */
      public Converter(String location) {
            r = new FileReader(location);
      }

      /**
       * This runs the conversion on the given text file and returns the converted
       * string
       *
       * @return Returns the converted string
       */
      public String convert() {
            String postFix = "";
            String tmp = "";
            Stack pile = new Stack<Token>();
            tmp = r.nextToken();

            while (!tmp.equals("EOF")) { // While next is not EOF
                  postFix += analyze(tmp).handle(pile);
                  tmp = r.nextToken();
            }

            return postFix;

      }

      /**
       * Given the input string, makes and returns the correct token
       *
       * @param toConvert
       * @return
       */
      private Token analyze(String toConvert) {

            if (toConvert.equals("(")) {
                  LeftParen toHandle = new LeftParen(toConvert);
                  return toHandle;
            } else if (toConvert.equals(")")) {
                  RightParen toHandle = new RightParen(toConvert);
                  return toHandle;
            } else if (toConvert.equals("^")) {
                  Power toHandle = new Power(toConvert);
                  return toHandle;
            } else if (toConvert.equals("*")) {
                  Times toHandle = new Times(toConvert);
                  return toHandle;
            } else if (toConvert.equals("/")) {
                  Divide toHandle = new Divide(toConvert);
                  return toHandle;
            } else if (toConvert.equals("+")) {
                  Plus toHandle = new Plus(toConvert);
                  return toHandle;
            } else if (toConvert.equals("-")) {
                  Minus toHandle = new Minus(toConvert);
                  return toHandle;
            } else if (toConvert.equals(";")) {
                  Semicolon toHandle = new Semicolon(toConvert);
                  return toHandle;
            } else {
                  Operand toHandle = new Operand(toConvert);
                  return toHandle;
            }
      }

}
```

```
/**
```

```
 * A token for divide
 *
 * @author xavier
 *
 */
public class Divide implements Token {

      private final static int PRECEDENCE = 2;
      private String identity;

      /**
       * The constructor for this token
       *
       * @param assign
       *            what string this will return.
       *
       *            This may seem unnecessary but this would let you customize the
       *            format which you get the postfix (+ vs. plus), as well as
       *            offer more modularity for my sorting system.
       */
      public Divide(String assign) {
            identity = assign;
      }

      /**
       * A tostring for the token
       *
       * @return A string of the token
       */
      public String toString() {
            return identity;
      }

      /**
       * Handles the token based on what it is
       *
       * @param s
       *            Is the stack the token will handle itself with
       * @returns the next section of string
       */
      public String handle(Stack<Token> s) {
            String toReturn = "";

            while (!s.isEmpty() && !s.peek().toString().equals("(")
                        && s.peek().getPrec() >= this.getPrec()) {
                  toReturn += s.pop().toString();
            }

            s.push(this);
            return toReturn;
      }

      /**
       * @return The precedence of the token
       */
      public int getPrec() {
            return PRECEDENCE;
      }
}
```

```java
/**
 * A token for left paren
 *
 * @author xavier
 *
 */
public class LeftParen implements Token {

      private final static int PRECEDENCE = 4;
      private String identity;

      /**
       * The constructor for this token
       *
       * @param assign
       *            what string this will return.
       *
       *            This may seem unnecessary but this would let you customize the
       *            format which you get the postfix (+ vs. plus), as well as
       *            offer more modularity for my sorting system.
       */
      public LeftParen(String assign) {
            identity = assign;
      }

      /**
       * A tostring for the token
       *
       * @return A string of the token
       */
      public String toString() {
            return identity;
      }

      /**
       * Handles the token based on what it is
       *
       * @param s
       *            Is the stack the token will handle itself with
       * @returns the next section of string
       */
      public String handle(Stack<Token> s) {
            String toReturn = "";

            s.push(this);

            return toReturn;
      }

      /**
       * @return The precedence of the token
       */
      public int getPrec() {
            return PRECEDENCE;
      }
}
```

```java
/**
 * Linked List is a collection of data nodes. All methods here relate to how one
 * can manipulate those nodes.
 *
 * @author Xavier
 * @version 02.09.17
 *
 *          * I affirm that I have carried out the attached academic endeavors
 *          with full academic honesty, in accordance with the Union College
 *          Honor Code and the course syllabus.
 */
public class LinkedList<T> {
      private int length; // number of nodes
      private ListNode firstNode; // pointer to first node

      public LinkedList() {
            length = 0;
            firstNode = null;
      }

      /**
       * Inserts at specified location
       *
       * @param The index at which to insert, starting at 0
       * @param The thing to insert
       */
      public void insertAt(int place, T toInsert) {
            ListNode<T> newNode = new ListNode<T>(toInsert);

            if (getLength() == 0) {
                  firstNode = newNode; // If theres nothing in the list, insert at
                                                      // start
            }

            else {
                  ListNode<T> n;
                  n = firstNode;
                  int index = 0;
                  while (index != (this.getLength() - 1) && index < (place - 1)) {
                        n = n.next;
                        index++;
                  }

                  if (index == this.getLength()) { // adds to end if place is above
                                                                        // end
                        n.next = newNode;
                  } else if (place <= 0) { // if place is 0 or less, insert at
start
                        newNode.next = firstNode; //
                        firstNode = newNode;
                  }

                  else { // n.next is 'place', adds before place.
                        newNode.next = n.next;
                        n.next = newNode;
                  }
            }
            length++;
      }
```

```
/**
 * Removes the data from the given location and returns it
 *
 * @param The
 *            location to remove, starting at 0
 * @return The removed information
 */

public T removeAt(int place) {
      if (place < 0 || place >= this.getLength())
            return null;

      ListNode<T> n;
      n = firstNode;
      T toReturn;

      if (place == 0) { // If its the first node
            toReturn = n.data;
            firstNode = n.next;
            length--;
            return toReturn;
      }

      int index = 0;
      while (index != (this.getLength() - 1) && index < (place - 1)) {
            n = n.next;
            index++;
      }

      toReturn = (T) (n.next.data); // returns the data at the place
      n.next = n.next.next;

      length--;
      return toReturn;
}

/**
 * Turn entire chain into a string
 *
 * @return return linked list as printable string of format (string, string,
 *         string)
 */
public String toString() {
      String toReturn = "(";
      ListNode<T> n;
      n = firstNode;
      while (n != null) {
            toReturn = toReturn + n.toString();
            n = n.next;
            if (n != null) {
                  toReturn = toReturn + ", ";
            }
      }
      toReturn = toReturn + ")";
      return toReturn;
}

/**
```

```
     * getter for number of nodes in the linked list
     *
     * @return length of LL
     */
    public int getLength() {
        return length;
    }

    /**
     * Gets the information from the specified location
     *
     * @param The
     *            index of what to get, starting at 0
     * @return the information
     */
    public T getData(int place) {
        if (place < 0 || place >= this.getLength())
            return null;

        ListNode<T> n;
        n = firstNode;
        T toReturn;
        if (place == 0) {
            toReturn = n.data;
            return toReturn;
        }

        int index = 0;
        while (index != (this.getLength() - 1) && index < (place - 1)) {
            n = n.next;
            index++;
        }
        toReturn = (T) (n.next.data);
        return toReturn;
    }

    /**
     * Clears the linked list of everything
     */
    public void clear() {
        firstNode = null;
        length = 0;
    }

}
```

```java
/**
 * ListNode is a building block for a linked list of data items
 *
 * This is the only class where I'll let you use public instance variables.
 * It's so we can reference information in the nodes using cascading dot
 * notation, like
 *          N.next.data instead of
 *          N.getNext().getData()
 *
 * @author C. Fernandes and G. Marten and Xavier
 * @version 02.09.17
 */
public class ListNode<T>
{
    public T data;
    public ListNode next;   // pointer to next node

    /** Non-default constructor
     *
     * @param String a reservation you want stored in this node
     */
    public ListNode(T String)
    {
        this.data = String;
        this.next = null;
    }

    // if you say "System.out.println(N)" where N is a ListNode, the
    // compiler will call this method automatically to print the contents
    // of the node.  It's the same as saying "System.out.println(N.toString())"
    public String toString()
    {
      if(this.data!=null)
            return data.toString();
        return null;  // call the toString() method in String class
    }
}
```

```
/**
 * A token for minus
 * @author xavier
 *
 */
public class Minus implements Token
{

      private final static int PRECEDENCE=1;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Minus(String assign) {
            identity=assign;
      }

      /**
       * A tostring for the token
       * @return A string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";

      while(!s.isEmpty() && !s.peek().toString().equals("(") &&
s.peek().getPrec()>=this.getPrec()) {
            toReturn+=s.pop().toString();
      }

      s.push(this);
        return toReturn;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```java
/**
 * A token for operands
 * @author xavier
 *
 */
public class Operand implements Token
{

      private final static int PRECEDENCE=4;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Operand(String assign) {
            identity=assign;
      }

      /**
       * @return a string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {

        return identity;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```
/**
 * A token for plus
 * @author xavier
 *
 */
public class Plus implements Token
{

      private final static int PRECEDENCE=1;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Plus(String assign) {
            identity=assign;
      }

      /**
       * A tostring for the token
       * @return A string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";

      while(!s.isEmpty() && !s.peek().toString().equals("(") &&
s.peek().getPrec()>=this.getPrec()) {
            toReturn+=s.pop().toString();
      }

      s.push(this);
        return toReturn;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```
/**
 * A token for power
 * @author xavier
 *
 */
public class Power implements Token
{

      private final static int PRECEDENCE=3;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Power(String assign) {
            identity=assign;
      }

      /**
       * A tostring for the token
       * @return A string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";
      while(!s.isEmpty() && !s.peek().toString().equals("(") &&
s.peek().getPrec()>=this.getPrec()) {
            toReturn+=s.pop().toString();
      }

      s.push(this);
        return toReturn;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```
/**
 * Testing suite for InfixPostFix converter
 * 
 * @author Xavier Qunn, Chris Fernandes, and Matt Anderson *I affirm that I have
 *         carried out the attached academic endeavors with full academic
 *         honesty, in accordance with the Union College Honor Code and the
 *         course syllabus.
 */
public class ProjectTesting {

      public static final boolean VERBOSE = true;

      /*
       * Runs a bunch of tests for the BetterBag class.
       * 
       * @param args is ignored
       */
      public static void main(String[] args) {

            Testing.setVerbose(true);
            Testing.startTests();

            testInserts();

            testRemove();

            testToStringAndPush();

            testPopAndPeek();

            testSizeAndIsEmpty();

            ConverterTester();

            Testing.finishTests();

      }

      private static void testInserts() {
            Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
            LinkedList<String> list = new LinkedList<String>();

            LinkedList<String> list2 = new LinkedList<String>();

            LinkedList<Integer> intList = new LinkedList<Integer>();

            list.insertAt(0, "One");
            Testing.assertEquals("Tests addition in empty list at start", "(One)",
                        list.toString());
            Testing.assertEquals("Tests addition in empty list capacity", 1,
                        list.getLength());

            list.insertAt(5, "Two");
            Testing.assertEquals("Tests addition at location longer than length",
                        "(One, Two)", list.toString());
            Testing.assertEquals("Tests addition in empty list capacity", 2,
                        list.getLength());

            list.insertAt(1, "Three");
```

```
        Testing.assertEquals("Tests addition between nodes",
                "(One, Three, Two)", list.toString());
        Testing.assertEquals("Tests addition in empty list capacity", 3,
                list.getLength());

        list.insertAt(0, "Four");
        Testing.assertEquals("Tests addition at start",
                "(Four, One, Three, Two)", list.toString());
        Testing.assertEquals("Tests addition in empty list capacity", 4,
                list.getLength());

        list.insertAt(-6, "Five");
        Testing.assertEquals("Tests addition at negative index",
                "(Five, Four, One, Three, Two)", list.toString());

        Testing.assertEquals("Tests addition in empty list capacity", 5,
                list.getLength());
        list.insertAt(6, "Six");
        Testing.assertEquals("Tests addition at end",
                "(Five, Four, One, Three, Two, Six)", list.toString());
        Testing.assertEquals("Tests addition in empty list capacity", 6,
                list.getLength());

        list2.insertAt(0, "a");
        list2.insertAt(1, null);
        list2.insertAt(2, "b");

        Testing.assertEquals("Tests addition between nodes", "(a, null, b)",
                list2.toString());

        intList.insertAt(0, 1);
        Testing.assertEquals("Tests addition in empty list at start", "(1)",
                intList.toString());
        Testing.assertEquals("Tests addition in empty list capacity", 1,
                intList.getLength());

    }

    private static void testRemove() {
        Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
        LinkedList<String> list = new LinkedList<String>();
        list.insertAt(10, "One");
        list.insertAt(10, "Two");
        list.insertAt(10, "Three");
        list.insertAt(10, "Four");
        Testing.assertEquals("Just checking", "(One, Two, Three, Four)",
                list.toString());
        Testing.assertEquals("Tests addition in empty list capacity", 4,
                list.getLength());

        Testing.assertEquals("Test removal of last", "Four", list.removeAt(3));
        Testing.assertEquals("Test removal of last", "(One, Two, Three)",
                list.toString());
        Testing.assertEquals("Tests capacity after removal", 3,
                list.getLength());

        Testing.assertEquals("Test removal of first", "One", list.removeAt(0));
        Testing.assertEquals("Test removal of first", "(Two, Three)",
                list.toString());
```

```
            Testing.assertEquals("Tests capacity after removal", 2,
                        list.getLength());

            Testing.assertEquals("Test removal of first", null, list.removeAt(-5));
            Testing.assertEquals("Test removal of first", "(Two, Three)",
                        list.toString());
            Testing.assertEquals("Tests capacity after removal", 2,
                        list.getLength());

            Testing.assertEquals("Test removal of first", null, list.removeAt(5));
            Testing.assertEquals("Test removal of first", "(Two, Three)",
                        list.toString());
            Testing.assertEquals("Tests capacity after removal", 2,
                        list.getLength());
      }

      private static void testToStringAndPush() {
            Testing.testSection("Testing toString and push");

            Stack<String> stack = new Stack<String>();
            Testing.assertEquals(
                        "An empty stack. (> indicates the top of the stack)",
"{>}",
                        stack.toString());

            stack.push("A");
            Testing.assertEquals("A stack with one item", "{>A}",
stack.toString());

            stack.push("B");
            stack.push("C");
            Testing.assertEquals("A stack with several items", "{>C, B, A}",
                        stack.toString());
      }

      private static void testPopAndPeek() {
            Testing.testSection("Testing Pop and Peak");

            Stack<String> stack = new Stack<String>();
            Testing.assertEquals("An empty stack, poping should return null", null,
                        stack.pop());
            Testing.assertEquals("An empty stack, peeking should return null",
                        null, stack.peek());

            stack.push("A");

            Testing.assertEquals("Peek a stack with one item ", "A", stack.peek());
            Testing.assertEquals("Pop a stack with one item ", "A", stack.pop());
            Testing.assertEquals("An empty stack, poping should return null", null,
                        stack.pop());
            Testing.assertEquals("An empty stack, peeking should return null",
                        null, stack.peek());

            stack.push("A");
            stack.push("B");
            stack.push("C");

            Testing.assertEquals("Peek a stack with several items ", "C",
                        stack.peek());
```

```
          Testing.assertEquals("Peek a stack with several items ", "C",
                      stack.pop());

          Testing.assertEquals("Peek a stack with several items ", "B",
                      stack.peek());
          Testing.assertEquals("Peek a stack with several items ", "B",
                      stack.pop());

          Testing.assertEquals("Peek a stack with several items ", "A",
                      stack.peek());
          Testing.assertEquals("Peek a stack with several items ", "A",
                      stack.pop());

          Testing.assertEquals("Peek a stack with several items ", null,
                      stack.peek());
          Testing.assertEquals("Peek a stack with several items ", null,
                      stack.pop());

    }

    private static void testSizeAndIsEmpty() {
          Testing.testSection("Testing toString and push");

          Stack<String> stack = new Stack<String>();
          Testing.assertEquals("An empty stack size", 0, stack.size());
          Testing.assertEquals("An empty stack is empty", true, stack.isEmpty());

          stack.push("A");
          Testing.assertEquals("A stack with one item size", 1, stack.size());
          Testing.assertEquals("A stack with one item is empty", false,
                      stack.isEmpty());

          stack.push("B");
          Testing.assertEquals("A stack with two item size", 2, stack.size());
          Testing.assertEquals("A stack with two item is empty", false,
                      stack.isEmpty());

          stack.push("C");
          Testing.assertEquals("A stack with three item size", 3, stack.size());
          Testing.assertEquals("A stack with three item is empty", false,
                      stack.isEmpty());

          stack.pop();
          Testing.assertEquals("A stack with 2 item size after pop", 2,
                      stack.size());
          Testing.assertEquals("A stack with 2 item is empty after pop", false,
                      stack.isEmpty());

          stack.pop();
          Testing.assertEquals("A stack with one item size after pop", 1,
                      stack.size());
          Testing.assertEquals("A stack with one item is empty after pop", false,
                      stack.isEmpty());

          stack.pop();
          Testing.assertEquals("A stack with no item size after pop", 0,
                      stack.size());
          Testing.assertEquals("A stack with no item is empty after pop", true,
                      stack.isEmpty());
```

```
	}

	private static void ConverterTester() {
		/*
		 *I wasn't sure the best way to test this, so I found an online infix-
postfix converter and made a file in the format
		 *of the output and I will use the file reader to put it into a string,
then compare it to the actual output.
		 *
		 *So while its only one test, its testing a lot of different things
		 */
		FileReader correct = new FileReader("src/CorrectOutput.txt");
		Converter c = new Converter("src/input.txt");
		String correctString="";
		String tmp="";


		while(!tmp.equals("EOF")) { //While next is not EOF
			correctString += tmp;
			if(tmp.equals(";")) {
				correctString+="\n";
			}
			tmp=correct.nextToken();
		}

		Testing.assertEquals("Testing all of input file", correctString,
				c.convert());

	}

}
```

```java
/**
 * A token for right paren
 * @author xavier
 *
 */
public class RightParen implements Token
{

      private final static int PRECEDENCE=1;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public RightParen(String assign) {
            identity=assign;
      }

      /**
       * A tostring for the token
       * @return A string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";

      while(!s.isEmpty() && !s.peek().toString().equals("(")) {

            toReturn+=s.pop().toString();
      }
      s.pop(); //discard (
        return toReturn;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```
/**
 * A token for plus
 * @author xavier
 *
 */
public class Semicolon implements Token
{

      private final static int PRECEDENCE=1;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnescisary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Semicolon(String assign) {
            identity=assign;
      }

      /**
       * @return a string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";

      while(!s.isEmpty()) {
            toReturn+=s.pop().toString();
      }

      toReturn+=this.toString();
        return toReturn + "\n";
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
```

```
// Don't forget the Javadocs!
// Notice that the generic type parameter does NOT implement
// the Token interface.  Make sure you understand why it shouldn't
// (and see the StackTester class for a hint.  Or just ask me!)


/**
* A Stack ADT that holds tokens.
* The first to be inserted into the stack is the last to be removed
*
*/
public class Stack<T>
{
    private LinkedList can;


    /**
     * Default constructor, makes a stack
     * Takes no inputs
     */
    public Stack() {
      can = new LinkedList<T>(); //The LinkedList is more complex than it needs to
be
                                             //But I am reusing one that I already
made,
                                             //Which is good I think. But thats why
its
                                             //Not optimized for a stack
    }


   /**
    * Checks if the stack is empty
    * @return true if empty, false if not empy
    */
    public boolean isEmpty() {
      if(can.getLength()>0) {
            return false;
      }
      else {
            return true;
      }
    }

    /**
     * Adds given token to top of stack
     * @param toPush the token to add to the top of the stack
     */
    public void push(T toPush) {
      can.insertAt(0, toPush);
    }

    /**
     * Removes and returns the top token of the stack
     * @return The top token of the stack, if there is no top token, return null
     */
    public T pop() {
      return (T)can.removeAt(0);
    }
    /**
     * Looks at the top of the stack, but does not return it.
     * @return The top token of the stack, or null if there isnt one
     */
    public T peek() {
      return (T)can.getData(0);
    }

    /**
     * Returns the "height" of the stack
     * @return The size of the stack
     */
    public int size() {
      return can.getLength();
    }

    /**
     * Returns the entire stack in string format, read top to bottom. The top value
is indicated with a >.
     * @return A string version of the stack
     */
      public String toString() {

            String toReturn = "{>";
            int tester = 0;

            while (can.getData(tester) != null && tester < this.size()) { // While
you dont run out of nodes

                  toReturn = toReturn + can.getData(tester); // Adds the info

                  if (tester < this.size() - 1
                              && can.getData(tester + 1) != null) {
                        toReturn += ", ";
                  }
                  tester++;
            }
            toReturn += "}";
            return toReturn;

      }

}
```

```
/**
 * A token for times
 * @author xavier
 *
 */
public class Times implements Token
{

      private final static int PRECEDENCE=2;
      private String identity;


      /**
       * The constructor for this token
       *
       * @param assign what string this will return.
       *
       * This may seem unnecessary but this would let you customize the
       * format which you get the postfix (+ vs. plus), as well as offer
       * more modularity for my sorting system.
       */
      public Times(String assign) {
            identity=assign;
      }

      /**
       * A tostring for the token
       * @return A string of the token
       */
    public String toString() {
      return identity;
    }

    /**
     * Handles the token based on what it is
     * @param s Is the stack the token will handle itself with
     * @returns the next section of string
     */
    public String handle(Stack<Token> s)
    {
      String toReturn="";

      while(!s.isEmpty() && !s.peek().toString().equals("(") &&
s.peek().getPrec()>=this.getPrec()) {
            toReturn+=s.pop().toString();
      }

      s.push(this);
        return toReturn;
    }

    /**
     * @return The precedence of the token
     */
    public int getPrec() {
      return PRECEDENCE;
    }
}
/** Describes the methods that must be defined in order for an
 * object to be considered a token.  Every token must be able
 * to be processed (handle) and printable (toString).
 *
 * @author Chris Fernandes
 * @version 10/26/08
 *
 */
public interface Token
{
      /** Processes the current token.  Since every token will handle
       *  itself in its own way, handling may involve pushing or
       *  popping from the given stack and/or appending more tokens
       *  to the output string.
       *
       *  @param s the Stack the token uses, if necessary, when processing itself.
       *  @return String to be appended to the output
       */
    public String handle(Stack<Token> s);

    /** Returns the token as a printable String
     *
     *  @return the String version of the token.  For example, ")"
     *  for a right parenthesis.
     */
    public String toString();

    /**
     * Returns the precedence of the token
     * @return The precedence
     */
    public int getPrec();



}
```

The following is a text file with what the output should be:

```
;
AB+;
AB+C+;
AB+C-;
ABC*+;
AB+C*;
AB/;
AB/C*;
AB/C*D-;
AB+CD-/;
AB+CD-*E+FG+/;
ABC^/D-;
ACB+/DE-^FG*HIJ-^+/;
```