```java
package proj4;

/**
 * The token represents the operator minus "*"
 * This is an operator which implements the interface Operator and interface
Token
 * This operator has precedence 1 when it comes to the order of operation
 * The higher the order of operation, the higher the precedence
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Plus implements Token, Operator {
    private final int PRECEDENCE = 1;

    /**
     * Get the precedence of the operator
     * @return the precendence of divide, which is 1
     */
    public int getPrecedent(){
        return PRECEDENCE;
    }
    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */
    public int compareTo(Operator other){
        if(this.getPrecedent() > other.getPrecedent()){
            return 1;
        }
        else if(this.getPrecedent() < other.getPrecedent()){
            return -1;
        }
        else{
            return 0;
        }
    }

    /**
     * Print out the toString of the operator
     * @return the String of Divide, which is "+"
     */
    public String toString() {
        return "+";
    }
    /**
     * Take a stack, pop and append to the postfix string every operator on
the
```

```
     * stack until one of the following conditions occurs:
     * 1. the stack is empty
     * 2. the top of the stack is a left parenthesis (which stays on the
stack)
     * 3. the operator on top of the stack has a lower precedence than the
current operator
     * Then push the current operator onto the stack.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with the popped operators, "+"
     */
    public String handle(Stack<Token> s)
    {   String result = "";
        while(!s.isEmpty() && !s.peek().toString().equals("(") && this.
compareTo((Operator) s.peek()) <= 0){
            result += s.pop().toString();
        }
        s.push(this);
        return result;
    }
}
```

```java
package proj4;

/**
 * The token represents the operator minus "-"
 * This is an operator which implements the interface Operator and interface
Token
 * This operator has precedence 1 when it comes to the order of operation
 * The higher the order of operation, the higher the precedence
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Minus implements Token, Operator {
    private final int PRECEDENCE = 1;

    /**
     * Get the precedence of the operator
     * @return the precedence of divide, which is 1
     */
    public int getPrecedent(){
        return PRECEDENCE;
    }
    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */
    public int compareTo(Operator other){
        if(this.getPrecedent() > other.getPrecedent()){
            return 1;
        }
        else if(this.getPrecedent() < other.getPrecedent()){
            return -1;
        }
        else{
            return 0;
        }
    }

    /**
     * Print out the toString of the operator
     * @return the String of Divide, which is "-"
     */
    public String toString() {
        return "-";
    }
    /**
     * Take a stack, pop and append to the postfix string every operator on
the
```

```
     * stack until one of the following conditions occurs:
     * 1. the stack is empty
     * 2. the top of the stack is a left parenthesis (which stays on the
stack)
     * 3. the operator on top of the stack has a lower precedence than the
current operator
     * Then push the current operator onto the stack.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with the popped operators, "/"
     */

    public String handle(Stack<Token> s)
    {   String result = "";
        while(!s.isEmpty() && !s.peek().toString().equals("(") && this.
compareTo((Operator) s.peek()) <= 0){
            result += s.pop().toString();
        }
        s.push(this);
        return result;
    }


}
```

```java
package proj4;

/**
 * @author: Emma Vu - CSC151 project 4
 * @version: 5/25/2020
 * The Stack class gives you access to the top of a Stack
 * through the first element of the linkedList
 *
 * The stack is defined by LinkedList data, which is a generic type
 * The number of element of the stack is stored in an instance variable
called manyItems.
 * For an empty stack, we set manyItems = 0 and default capacity = 10
 * The capacity of the stack is stored in an instance variable called
capacity
 * Remove the top of the stack by removing the first element of the
linkedList
 * Add to the stack by adding to the linkedList another element at Head.
 *
 * This is an implementation of a stack of generic type
 */
public class Stack<T>
{
    private LinkedList<T> data;
    private int manyItems;
    private int capacity;
    private final int INITIAL_CAPACITY = 10;

    /**
     * Create an empty stack with default constructor.
     */
    public Stack() {
        data = new LinkedList<T>();
        manyItems = 0;
        capacity = INITIAL_CAPACITY;

    }

    /**
     * Creates a new non-default stack.
     * Have to check for valid capacity. Else, set default capacity = 10
     *
     * @param initialCapacity the initial capacity of the stack. Have to be
positive integer
     */
    public Stack(int initialCapacity){
        if(initialCapacity < 0){

            initialCapacity = INITIAL_CAPACITY;
        }

        capacity = initialCapacity;

        manyItems = 0;
```

```
        data = new LinkedList<T>();

    }

    /**
     * Check whether the stack is empty or not
     * @return True if the stack has 0 elements. Else, false
     */
    public boolean isEmpty() {
        return size() == 0;
    }

    /**
     * add a new item at the top of the stack by inserting at the head of a
LinkedList.
     * After adding, the number of items in the stack will increment by 1
     * @param toPush the value to add to the stack
     */
    public void push(T toPush) {
        data.insertAtHead(toPush);
        manyItems ++;

    }

    /**
     * pop the top item of the stack by removing the head of a LinkedList.
     * After removing, the number of items will decrement by 1
     * @return the popped item of the stack. If the stack is empty, return
null
     */

    public T pop() {
        if(!isEmpty()){
            T poppedItem = data.removeHead();
            manyItems--;
            return poppedItem;

        }
        else{
            return null;
        }
    }

    /**
     * See the top of the stack by getting the head of a LinkedList.
     * POST CONDITION: after peeking, the stack remains the same
     * @return the top item of the stack
     */
    public T peek() {
        return data.getHead();
    }

    /**
```

```
     * Get the size of the stack
     * @return the number of elements in the stack
     */

    public int size() {
        return manyItems;
    }
    /**
     * return the String represent the Stack. If the stack is empty, the
string representation is "{>}"
     * Else, the string representation will be "{>A,B,C}", etc. where each
item of the stack is separated by a comma
     * and ">A" indicates the top of the stack
     * @return String represent Stack and the top of the stack
     */

    public String toString() {

        String printToString = "{>}";
        if(!isEmpty()){
            printToString = toStringOfNonEmpty();
        }
        return printToString;

}

    /**
     * private method to return the string of a non-empty stack. Each item of
 the stack will be separated by a comma
     * ">" indicates the item at the top of the stack
     * @return the string representation of a non-empty stack
     */

    private String toStringOfNonEmpty(){

        String printStr = "{>";
        for (int i = 0; i < size(); i++){

            printStr += data.getItemAt(i);
            if(i < size()-1){
                printStr += ",";
            }
        }
        printStr += "}";
        return printStr;
    }

    /**
     * get the capacity of the stack
     * @return the capacity (the maximum items) the stack can hold
     */
    public int getCapacity(){
        return capacity;
```

```
    }

    /**
     * Increase the stack's capacity to be
     * at least minCapacity.  Does nothing
     * if current capacity is already >= minCapacity.
     *
     * @param minCapacity the minimum capacity that the stack
     * should now have.
     */
    public void ensureCapacity(int minCapacity)
    {
        if (getCapacity() < minCapacity) {
            capacity = minCapacity;
        }
    }

    /**
     * Reduce the current capacity to its actual size, so that it has
     * capacity to store only the elements currently stored.
     */
    public void trimToSize()
    {
        if (getCapacity() > size()){
            capacity = manyItems;
        }
    }




}
```

```java
package proj4;

/**
 * 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();


}
```

```java
package proj4;
/**
 * The class to run the program by creating a converter and converting infix
to postfix from the file path
 * Print out the result from the console
 *
 * @author Emma Vu
 * @version 5/23/2020
 */

public class Client
{
    public static void main(String[] args)
    {
        Converter inFixToPostFix = new Converter("src/proj4/proj4_input.txt"
);
        inFixToPostFix.convert();

    }
}
```

```java
package proj4;

/**
 * The token represents the operator divide "/"
 * This is an operator which implements the interface Operator and interface
Token
 * This operator has precedence 2 when it comes to the order of operation
 * The higher the order of operation, the higher the precedence
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Divide implements Token, Operator {
    private final int PRECEDENCE = 2;


    /**
     * Get the precedence of the operator
     * @return the precendence of divide, which is 2
     */
    public int getPrecedent(){
        return PRECEDENCE;
    }

    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */

    public int compareTo(Operator other){
        if(this.getPrecedent() > other.getPrecedent()){
            return 1;
        }
        else if(this.getPrecedent() < other.getPrecedent()){
            return -1;
        }
        else{
            return 0;
        }
    }

    /**
     * Print out the toString of the operator
     * @return the String of Divide, which is "/"
     */
    public String toString() {
        return "/";
    }
```

```
    /**
     * Take a stack, pop and append to the postfix string every operator on
the
     * stack until one of the following conditions occurs:
     * 1. the stack is empty
     * 2. the top of the stack is a left parenthesis (which stays on the
stack)
     * 3. the operator on top of the stack has a lower precedence than the
current operator
     * Then push the current operator onto the stack.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with the popped operators, "/"
     */

    public String handle(Stack<Token> s) {
        String result = "";

        while(!s.isEmpty() && !s.peek().toString().equals("(") && this.
compareTo((Operator)s.peek()) <= 0){
            result += s.pop().toString();
        }
        s.push(this);

        return result;
    }



}
```

```java
package proj4;
/**
 *
 * This class implements Token and represents operands (capital letters) like
 "A", "B", "C", etc.
 * Anything that is not an operator, left parenthesis, right parenthesis, and
 semicolon is considered to be an operand.
 * The input of this operand is a String.
 *
 * @author Emma Vu
 * @version 5/25/2020
 */
public class Operand implements Token {
    private String inputStr;

    /**
     * default constructor where the operand is represented by an empty
string
     */
    public Operand(){
        inputStr = "";
    }

    /**
     * non-default constructor where it is represented by a string
     * @param initialStr
     */
    public Operand(String initialStr){
        inputStr = initialStr;
    }

    /**
     * Return the String representation of this operand
     * @return The String of operand.
     */

    public String toString(){
        return inputStr;
    }

    /**
     * An operand (represented by a capital letter), immediately append it to
 the postfix string.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string of this operand so that can add it to the postfix
later
     */
    public String handle(Stack<Token> s){
        return toString();
    }
}
```

```java
package proj4;

/**
 * The token represents the operator exponent "^"
 * This is an operator which implements the interface Operator and interface
Token
 * This operator has precedence 3 when it comes to the order of operation
 * The higher the order of operation, the higher the precedence
 *
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Exponent implements Token, Operator{
    private final int PRECEDENCE = 3;

    /**
     * Get the precedence of the operator
     * @return the precendence of divide, which is 3
     */
    public int getPrecedent(){
        return PRECEDENCE;
    }

    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */

    public int compareTo(Operator other){
        if(this.getPrecedent() > other.getPrecedent()){
            return 1;
        }
        else if(this.getPrecedent() < other.getPrecedent()){
            return -1;
        }
        else{
            return 0;
        }
    }

    /**
     * Print out the toString of the operator
     * @return the String of Exponent, which is "^"
     */

    public String toString() {
        return "^";
```

```
    }
    /**
     * Take a stack, pop and append to the postfix string every operator on
the
     * stack until one of the following conditions occurs:
     * 1. the stack is empty
     * 2. the top of the stack is a left parenthesis (which stays on the
stack)
     * 3. the operator on top of the stack has a lower precedence than the
current operator
     * Then push the current operator onto the stack.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with the popped operators, "^"
     */

    public String handle(Stack<Token> s) {
        String result = "";

        while(!s.isEmpty() && !s.peek().toString().equals("(") && this.
compareTo((Operator)s.peek()) <= 0){
            result += s.pop().toString();
        }
        s.push(this);

        return result;
    }




}
```

```java
package proj4;

/**
 * The ListNode class is more data-specific than the LinkedList class.  It
 * details what a single node looks like.  This node has one data field,
 * holding a pointer to a String object.
 * This is an implementation of generic type list node
 *
 * This is the only class where I'll let you use public instance variables.
 * @author Emma Vu
 * @version 5/29/2020
 */
public class ListNode<T>
{
    public T data;
    public ListNode next;

    /**
     * non-default constructor creating a single generic type node
     * @param new_data the initial generic type data to assign for the node
     */
    public ListNode(T new_data)
    {
        data = new_data;
        next = null;
    }

    /**
     * print out the representation of a single node in toString()
     * @return the representation of that node's data in string
     */

    public String toString(){
        return data.toString();
    }

}
```

```java
package proj4;

/**
 * The token represents the operator multiply "*"
 * This is an operator which implements the interface Operator and interface
Token
 * This operator has precedence 2 when it comes to the order of operation
 * The higher the order of operation, the higher the precedence
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Multiply implements Token, Operator {
    private final int PRECEDENCE = 2;

    /**
     * Get the precedence of the operator
     * @return the precedence of divide, which is 1
     */
    public int getPrecedent(){
        return PRECEDENCE;
    }
    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */
    public int compareTo(Operator other){
        if(this.getPrecedent() > other.getPrecedent()){
            return 1;
        }
        else if(this.getPrecedent() < other.getPrecedent()){
            return -1;
        }
        else{
            return 0;
        }
    }

    /**
     * Print out the toString of the operator
     * @return the String of Divide, which is "*"
     */
    public String toString() {
        return "*";
    }
    /**
     * Take a stack, pop and append to the postfix string every operator on
the
```

```
     * stack until one of the following conditions occurs:
     * 1. the stack is empty
     * 2. the top of the stack is a left parenthesis (which stays on the
stack)
     * 3. the operator on top of the stack has a lower precedence than the
current operator
     * Then push the current operator onto the stack.
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with the popped operators, "/"
     */
    public String handle(Stack<Token> s)
    {   String result = "";
        while(!s.isEmpty() && !s.peek().toString().equals("(") && this.
compareTo((Operator) s.peek()) <= 0){
            result += s.pop().toString();
        }
        s.push(this);
        return result;
    }

}
```

```java
package proj4;
/**
 * Describes the methods that must be defined in order for an
 * object to be considered an operator.  Not all Tokens are operators. Every
Operator must have
 * precedent and must be able to get it, also, must be able to compare its
 * precedent to other operator, in this case they are "+", "-", "*", "/", "^"
 *
 * @author Emma Vu
 * @version 5/25/2020
 *
 */
public interface Operator {
    /**
     * Get the precedent of the operator
     * @return the precedent
     */
    public int getPrecedent();
    /**
     * Compare this operator to other operator, if this operator has higher
precedent
     * return 1, if this operator has lower precedent return -1, if this
operator has
     * equal precedent return 0
     * @param other other Operator to compare to
     * @return 1 if higher, -1 if lower, 0 if equal precedent
     */
    public int compareTo(Operator other);
}
```

```java
package proj4;

/**
 * CLASS INVARIANTS
 * This class is used to convert infix to postfix
 * the fileReader is an instance variable to store the input path
 * The operator and operand is represented by the Token with the same name.
 * For example, "+" is stored in Token Plus, "-" is stored in Token Minus
, "*" is stored in Token Multiply
 * "/" is stored in Token Divide, "^" is stored in Token Exponent,
 * "(" is stored in Token LeftParen, ")" is stored in Token RightParen
 * ";" is stored in Token Semicolon. The operand like "A", "B", "C", etc. is
stored in Token Operand
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class Converter{

    /**
     * non-default constructor; Gradescope needs this to run tests
     * @param infile path to the input file
     */
    private FileReader fileReader;
    public Converter(String infile)
    {
        fileReader = new FileReader(infile);
    }

    /**
     * Read the next token, identify and make the new Token correct type,
handle it correspondingly by adding the
     * right type to the returned string when not reaching ";".
     * After reaching ";", print out the console infix --> postfix and then
empty both infix and postfix.
     * Read the next token until reaching "EOF", meaning reading all the
input file
     */

    public void convert(){
        Stack<Token> tokenStack = new Stack<Token>();
        String toConvert = fileReader.nextToken();
        String postFix = "";
        String inFix = "";

        while(!toConvert.equals("EOF")){
            Token tokenNext = identifyTokenType(toConvert);
            postFix += tokenNext.handle(tokenStack);


            if(!toConvert.equals(";")){
                inFix += toConvert;
            }
```

```
            else {
                System.out.println(inFix + " --> " + postFix);
                postFix = "";
                inFix = "";
            }
            toConvert = fileReader.nextToken();

        }


    }

    /**
     * Identify the correct type of Token from the input string
     * For example, "+" is stored in Token Plus, "-" is stored in Token Minus
, "*" is stored in Token Multiply
     * "/" is stored in Token Divide, "^" is stored in Token Exponent,
     * "(" is stored in Token LeftParen, ")" is stored in Token RightParen
     * ";" is stored in Token Semicolon.
     * Else, the operand like "A", "B", "C", etc. is stored in Token Operand
("A"), Operand("B"), Operand("C"), etc.
     * @param toIdentify the input string to identify the Token
     * @return the correct type of Token corresponds to the input string
     */

    private Token identifyTokenType(String toIdentify){
        if(toIdentify.equals("+")){
            return new Plus();
        }
        else if(toIdentify.equals("-")){
            return new Minus();
        }
        else if(toIdentify.equals("*")){
            return new Multiply();
        }
        else if(toIdentify.equals("/")){
            return new Divide();
        }
        else if(toIdentify.equals("^")){
            return new Exponent();

        }
        else if(toIdentify.equals("(")){
            return new LeftParen();
        }
        else if(toIdentify.equals(")")){
            return new RightParen();
        }
        else if(toIdentify.equals(";")){
            return new Semicolon();
        }
        else{
```

```
            return new Operand(toIdentify);
        }
    }

}
```

```java
package proj4;

/**
 * This class implements the Token interface and represents LeftParen as "("
 *
 * @author Emma Vu
 * @version 5/23/2020
 */
public class LeftParen implements Token {
    /**
     * Push this Token onto the stack because it's a left parenthesis
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return an empty string
     */
    public String handle(Stack<Token> s){
        s.push(this);
        return "";
    }

    /**
     * Print out the representation of LeftParen
     * @return the representation, in this case it's "("
     */
    public String toString(){
        return "(";
    }

}
```

```java
package proj4;

/**
 * This class implements Token interface and represents Semicolon ";"
 *
 * @author Emma Vu
 * @version 5/25/2020
 */
public class Semicolon implements Token {
    /**
     * The Semicolon indicates that the infix expression has been completely
scanned.
     * However, the stack may still contain some operators. All remaining
operators
     * should be popped and appended to the postfix string
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return the string with all the leftover popped operators
     */
    public String handle(Stack<Token> s){
        String result = "";
        while(!s.isEmpty()){
            result += s.pop();
        }
        return result;
    }

    /**
     * print out the representation of semicolon in toString
     * @return the string representation, which is ";"
     */
    public String toString(){
        return ";";
    }

}
```

```java
package proj4;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * The test for generic type stack class
 *
 * @author Emma Vu
 * @version 5/21/2020
 *
 */
public class StackTest {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Stack<String> stack;

    @Before
    public void setUp() throws Exception {
        stack = new Stack<String>();
    }

    @After
    public void tearDown() throws Exception {
        stack = null;
    }

    @Test
    public void testStackConstructor_toString () {
        assertEquals ("An empty stack. (> indicates the top of the stack)",
"{>}", stack.toString());
    }

    @Test
    public void testToStringOneItem(){
        stack.push("A");
        assertEquals("{>A}",stack.toString());
    }

    @Test
    public void testConstructorSize(){
        assertEquals(0,stack.size());
    }

    @Test
    public void testToStringMultipleItems(){
```

```
        stack.push("A");
        stack.push("B");
        stack.push("C");
        assertEquals("{>C,B,A}",stack.toString());
    }

    @Test
    public void testToStringSimilar(){
        stack.push("A");
        stack.push("A");
        stack.push("A");
        assertEquals("{>A,A,A}",stack.toString());
    }

    @Test
    public void testStackPushOneOntoEmptyStack () {
        stack.push("A");
        assertEquals(1,stack.size());
        assertEquals ("Pushing A onto an empty stack.", "{>A}", stack.
toString().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPushTwoOntoEmptyStack () {
        stack.push("A");
        stack.push("B");
        assertEquals(2,stack.size());
        assertEquals ("Pushing first A and then B onto an empty stack.", "{>B
,A}", stack.toString().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPushThreeOntoEmptyStack () {
        stack.push("A");
        stack.push("B");
        stack.push("C");
        assertEquals(3,stack.size());
        assertEquals ("Pushing first A, then B, then C onto an empty stack."
, "{>C,B,A}", stack.toString().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPushSimilar(){
        stack.push("A");
        stack.push("A");

        assertEquals(2,stack.size());
        assertEquals("Push identical strings on an empty stack","{>A,A}",
stack.toString().replaceAll("[ ]+",""));
    }

    @Test
    public void testIsEmptyTrue(){
```

```
    assertTrue(stack.isEmpty());
}

@Test
public void testIsEmptyFalse(){
    stack.push("A");
    assertFalse(stack.isEmpty());
}

@Test
public void testSizeEmpty(){
    assertEquals(0,stack.size());
}

@Test
public void testSizeNonEmpty(){
    stack.push("A");
    stack.push("B");
    assertEquals(2,stack.size());
}

@Test
public void testPeekEmpty(){
    assertNull(stack.peek());
}

@Test
public void testPeekOneItem(){
    stack.push("A");
    assertEquals("A",stack.peek());
}

@Test
public void testPeekMultiple(){
    stack.push("A");
    stack.push("B");
    assertEquals("B",stack.peek());
}

@Test
public void testPopEmpty(){
    assertNull(stack.pop());
}

@Test

public void testPopOneItem(){
    stack.push("A");
    stack.pop();
    assertTrue(stack.isEmpty());
}

@Test
```

```
public void testPopMultipleItems(){
    stack.push("A");
    stack.push("B");
    stack.pop();
    assertEquals(1,stack.size());
    assertEquals("A",stack.peek());
    assertEquals("{>A}", stack.toString().replaceAll("[ ]+",""));
}

@Test
public void testPopMultipleTimes(){
    stack.push("A");
    stack.push("B");
    stack.push("C");
    stack.push("D");
    stack.pop();
    stack.pop();
    assertEquals(2,stack.size());
    assertEquals("B",stack.peek());
    assertEquals("{>B,A}", stack.toString().replaceAll("[ ]+",""));

}
@Test
public void testPopSimilar(){
    stack.push("A");
    stack.push("A");
    stack.push("A");
    stack.pop();
    assertEquals(2,stack.size());
    assertEquals("A",stack.peek());
    assertEquals("{>A,A}", stack.toString().replaceAll("[ ]+",""));









}
@Test
public void testGetCapacityDefault(){
    assertEquals(10,stack.getCapacity());
}

@Test
public void testGetCapacityNonDefaultNegative(){
    stack = new Stack<String>(-1);
    assertEquals(10,stack.getCapacity());
}

@Test
public void testGetCapacityNonDefaultValid(){
    stack = new Stack<String>(3);
    assertEquals(3,stack.getCapacity());
```

```
}
@Test

public void test_EnsureCapacity_Smaller(){
    stack.push("A");
    stack.push("B");
    stack.push("C");
    stack.ensureCapacity(1000);
    assertEquals(1000,stack.getCapacity());
    assertEquals(3,stack.size());
}

@Test

public void test_EnsureCapacity_Greater(){
    stack = new Stack<String>(1000);
    stack.push("A");
    stack.push("B");
    stack.ensureCapacity(10);
    assertEquals(1000,stack.getCapacity());
    assertEquals(2,stack.size());


}

@Test

public void test_EnsureCapacity_Equal(){
    stack = new Stack<String>(20);

    stack.ensureCapacity(20);
    assertEquals(20,stack.getCapacity());

}

@Test

public void test_TrimToSize_EmptySequence(){

    stack.trimToSize();
    assertEquals(0,stack.getCapacity());

}

@Test


public void test_TrimToSize_NonEmpty(){

    stack.push("A");
    stack.push("B");
    stack.trimToSize();
    assertEquals(2,stack.getCapacity());
```

```
        }



}
```

```java
package proj4;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * The test for generic type stack class
 *
 * @author Emma Vu
 * @version 5/21/2020
 *
 */
public class TokenTest {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception {
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception {
        stack = null;
    }

    @Test
    public void testToStringRightParen(){
        RightParen rightParen = new RightParen();
        assertEquals(")",rightParen.toString());
        stack.push(rightParen);
        assertEquals("{>)}",stack.toString());
    }

    @Test
    public void testToStringLeftParen(){
        LeftParen leftParen = new LeftParen();
        assertEquals("(",leftParen.toString());
        stack.push(leftParen);
        assertEquals("{>(}",stack.toString());
    }

    @Test
    public void testToStringPlus(){
        Plus plus = new Plus();
```

```
        assertEquals("+",plus.toString());
        stack.push(plus);
        assertEquals("{>+}",stack.toString());
    }

    @Test
    public void testToStringMinus(){
        Minus minus = new Minus();
        assertEquals("-",minus.toString());
        stack.push(minus);
        assertEquals("{>-}",stack.toString());
    }

    @Test
    public void testToStringMultiply(){
        assertEquals("*",new Multiply().toString());
        stack.push(new Multiply());
        assertEquals("{>*}",stack.toString());
    }

    @Test
    public void testToStringDivide(){
        Divide divide = new Divide();
        assertEquals("/",divide.toString());
        stack.push(divide);
        assertEquals("{>/}",stack.toString());
    }

    @Test
    public void testToStringExponent(){
        Exponent exponent = new Exponent();
        assertEquals("^",exponent.toString());
        stack.push(exponent);
        assertEquals("{>^}",stack.toString());
    }

    @Test
    public void testToStringSemiColon(){
        Semicolon semiColon = new Semicolon();
        assertEquals(";",semiColon.toString());
        stack.push(semiColon);
        assertEquals("{>;}",stack.toString());
    }

    @Test
    public void testConstructor(){
        assertEquals ("An empty stack. (> indicates the top of the stack)",
"{>}", stack.toString());
    }

    @Test
    public void testPlusEmpty(){
```

```
    Plus plus = new Plus();
    String returnStr = plus.handle(stack);
    assertEquals(stack.toString(),"{>+}");
    assertEquals("",returnStr);
}
@Test
public void testPushPlus(){
    Plus plus = new Plus();
    stack.push(plus);
    assertEquals("{>+}",stack.toString());


}


@Test
public void testPlusEqualPrecedent(){
    Plus plus = new Plus();

    stack.push(new Minus());
    stack.push(new Minus());
    stack.push(plus);

    String returnStr = plus.handle(stack);
    assertEquals(stack.toString(),"{>+}");
    assertEquals("+--",returnStr);
}

@Test
public void testPlusHigherPrecedent(){
    stack.push(new Multiply());
    stack.push(new Divide());
    stack.push(new Exponent());
    stack.push(new Multiply());

    String returnStr = new Plus().handle(stack);
    assertEquals(stack.toString(),"{>+}");
    assertEquals("*^/*",returnStr);
}

@Test
public void testPlusRandomPrecedent(){
    stack.push(new Multiply());
    stack.push(new Plus());
    stack.push(new Divide());
    stack.push(new Minus());

    String returnStr = new Plus().handle(stack);
    assertEquals(stack.toString(),"{>+}");
    assertEquals("-/+*",returnStr);
}
```

```
@Test
public void testMinusEmpty(){


    String returnStr = new Minus().handle(stack);
    assertEquals(stack.toString(),"{>-}");
    assertEquals("",returnStr);
}
@Test
public void testPushMinus(){

    stack.push(new Minus());
    assertEquals("{>-}",stack.toString());


}


@Test
public void testMinusEqualPrecedent(){

    Minus minus = new Minus();
    stack.push(new Plus());
    stack.push(new Plus());
    stack.push(new Minus());

    String returnStr = minus.handle(stack);
    assertEquals(stack.toString(),"{>-}");
    assertEquals("-++",returnStr);
}

@Test
public void testMinusHigherPrecedent(){
    stack.push(new Multiply());
    stack.push(new Divide());
    stack.push(new Exponent());
    stack.push(new Multiply());

    String returnStr = new Minus().handle(stack);
    assertEquals(stack.toString(),"{>-}");
    assertEquals("*^/*",returnStr);
}

@Test
public void testMinusRandomPrecedent(){
    stack.push(new Multiply());
    stack.push(new Plus());
    stack.push(new Divide());
    stack.push(new Minus());

    String returnStr = new Minus().handle(stack);
    assertEquals(stack.toString(),"{>-}");
    assertEquals("-/+*",returnStr);
```

```
}

@Test
public void testMinusLeftParen(){
    stack.push(new Multiply());
    stack.push(new Plus());
    stack.push(new LeftParen());
    stack.push(new Divide());
    stack.push(new Minus());

    String returnStr = new Minus().handle(stack);
    assertEquals(returnStr, "-/");

}

@Test
public void testMultiplyEmpty(){
    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*}");
    assertEquals("",returnStr);
}
@Test
public void testMultiplyLowerPrecedent(){
    stack.push(new Plus());
    stack.push(new Minus());
    stack.push(new Minus());
    stack.push(new Minus());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*,-,-,-,+}");
    assertEquals("",returnStr);
}

@Test
public void testMultiplyEqualPrecedent(){
    stack.push(new Multiply());
    stack.push(new Divide());
    stack.push(new Multiply());
    stack.push(new Divide());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*}");
    assertEquals("/*/*",returnStr);
}
@Test
public void testMultiplyHigherPrecedent(){
    stack.push(new Exponent());
    stack.push(new Exponent());
    stack.push(new Exponent());
    stack.push(new Exponent());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*}");
```

```
    assertEquals("^^^^",returnStr);
}
@Test
public void testMultiplyLowerThenHigherPrecedent(){
    stack.push(new Plus());
    stack.push(new Plus());
    stack.push(new Exponent());
    stack.push(new Exponent());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*,+,+}");
    assertEquals("^^",returnStr);
}
@Test
public void testMultiplyHigherThenLowerPrecedent(){
    stack.push(new Exponent());
    stack.push(new Exponent());
    stack.push(new Plus());
    stack.push(new Plus());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*,+,+,^,^}");
    assertEquals("",returnStr);

}

@Test
public void testMultiplyRandomPrecedent(){
    stack.push(new Exponent());
    stack.push(new Plus());
    stack.push(new Divide());
    stack.push(new Multiply());

    String returnStr = new Multiply().handle(stack);
    assertEquals(stack.toString(),"{>*,+,^}");
    assertEquals("*/",returnStr);
}

@Test
public void testExponentEmpty(){
    String returnStr = new Exponent().handle(stack);
    assertEquals(stack.toString(),"{>^}");
    assertEquals("",returnStr);
}

@Test
public void testExponentLowerPrecedent(){
    stack.push(new Plus());
    stack.push(new Minus());
    stack.push(new Multiply());
    stack.push(new Divide());

    String returnStr = new Exponent().handle(stack);
```

```
    assertEquals(stack.toString(),"{>^,/,*,-,+}");
    assertEquals("",returnStr);
}
@Test
public void testExponentEqualPrecedent(){
    stack.push(new Exponent());
    stack.push(new Exponent());
    stack.push(new Exponent());
    stack.push(new Exponent());

    String returnStr = new Exponent().handle(stack);
    assertEquals(stack.toString(),"{>^}");
    assertEquals("^^^^",returnStr);

}

@Test
public void testSemiColonEmpty(){
    String returnStr = new Semicolon().handle(stack);
    assertEquals(stack.toString(),"{>}");
    assertEquals("",returnStr);
}

@Test
public void testSemiColon(){
    stack.push(new Plus());
    stack.push(new Minus());
    stack.push(new Multiply());
    stack.push(new Divide());

    String returnStr = new Semicolon().handle(stack);
    assertEquals(stack.toString(),"{>}");
    assertEquals("/*-+",returnStr);
}

@Test
public void testLeftParenEmpty(){
    String returnStr = new LeftParen().handle(stack);
    assertEquals(stack.toString(),"{>(}");
    assertEquals("",returnStr);

}

@Test
public void testLeftParenNonEmpty(){
    stack.push(new Plus());
    stack.push(new Minus());
    stack.push(new Multiply());
    stack.push(new Divide());

    String returnStr = new LeftParen().handle(stack);
    assertEquals(stack.toString(),"{>(,/,*,-,+}");
    assertEquals("",returnStr);
```

```
    }

    @Test
    public void testRightParenNoneBetween(){
        stack.push(new LeftParen());

        String returnStr = new RightParen().handle(stack);
        assertEquals(stack.toString(),"{>}");
        assertEquals("",returnStr);

    }

    @Test
    public void testRightParenBetween(){
        stack.push(new Plus());
        stack.push(new Minus());
        stack.push(new LeftParen());
        stack.push(new Multiply());
        stack.push(new Divide());

        String returnStr = new RightParen().handle(stack);
        assertEquals(stack.toString(),"{>-,+}");
        assertEquals("/*",returnStr);
    }




}
```

```java
package proj4;
import java.io.*;
/**
 * A FileReader object will read tokens from an input file.  The name of
 * the input file is given when the constructor is run. The lone method,
 * nextToken(), will return the next token in the file as a String.
 *
 * @author Chris Fernandes
 * @version 2/20/17
 */
public class FileReader
{
    private StreamTokenizer st;     //file descriptor

    /** non-default constructor.
     *
     *  @param fileName path to input file. Can either be a full
     *  path like "C:/project4/proj4_input.txt" or a relative one like
     *  "src/proj4_input.txt" where file searching begins inside
     *  the project folder.
     *  Be sure to use forward slashes to specify folders.  And
     *  don't forget the quotes (it's a String!)
     */
    public FileReader(String fileName)
    {
        try {
        st = new StreamTokenizer(
                 new BufferedReader(
                     new InputStreamReader(
                         new FileInputStream(fileName))));
        } catch (IOException e) {}
        st.resetSyntax();                          // remove default rules

        st.ordinaryChars(0, Character.MAX_VALUE); // turn on all chars
        st.whitespaceChars(0, 39);                  // ignore unprintables &
        st.whitespaceChars(95, Character.MAX_VALUE); // everything after ^
    }

    /** Returns the next valid token from the input file.
     *  Possible tokens are "+", "-", "*", "/", "^", "(", ")",
     *  ";", operands (capital letters), and the special
     *  token "EOF" which is returned when the
     *  end of the input file is reached.
     *
     *  @return next valid token or "EOF"
     */
    public String nextToken()
    {
        try
        {
        while (st.nextToken() != StreamTokenizer.TT_EOF) {
            if (st.ttype < 0)
            {
```

```
                return st.sval;
            }
            else
            {
                return String.valueOf((char)st.ttype);
            }
        }
        return "EOF";
        } catch (IOException e) {}
        return "error on token read";
    }
}
```

```java
package proj4;

/**
 *  Emma Vu - Project 4
 * @version 5/23/2020
 * LinkedList is a collection of data nodes. All methods here relate to how one can manipulate those nodes
 * The LinkedList class gives the access to the beginning of a LinkedList through instance variable called firstNode
 * The length of the LinkedList (number of the items) is stored in an instance variable called length
 * The last Node of the list has variable next = null
 * The LinkedList is defined by the firstNode.
 * The second node is referred by firstNode.next, the third node is referred by firstNode.next.next, etc.
 * When we reach the last node of the LinkedList, the last node.next will be null.
 * For an empty LinkedList to start with, the default is length = 0 and firstNode = null
 * This is an implementation of generic type Linked List
 */
public class LinkedList<T>
{
    //Instance variables
    private int length;
    private ListNode<T> firstNode;

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

    /**
     * get the length of the LinkedList
     * @return length
     */
    public int getLength()
    {
        return length;
    }

    /** insert new String at linked list's head
     *
     * @param data the String to be inserted
     *
     */

    public void insertAtHead(T data)
    {
        ListNode<T> newNode = new ListNode(data);
        if (isEmpty())
```

```
        {
            firstNode=newNode;
        }
        else
        {
            newNode.next=firstNode;
            firstNode=newNode;
        }
        length++;
    }


    /**
     * return the String that denotes the elements in the list with correct
order
     * @return The list of element in correct order
     */
    public String toString(){
        String toReturn = "(";
        ListNode<T> runner = firstNode;
        while(runner != null){
            toReturn = toReturn + runner;
            runner = runner.next;
            if(runner != null){
                toReturn = toReturn + ", ";
            }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }

    /**
     * Check if the List if empty or not
     * @return true if length is > 0, false otherwise
     */
    public boolean isEmpty(){
        return getLength() == 0;
    }

    /** insert data at end of list
     *
     * @param newData new String to be inserted
     */
    public void insertAtTail(T newData)
    {
        ListNode<T> insertNode = new ListNode(newData);
        if(isEmpty()){
            firstNode = insertNode;
        }
        else{

            ListNode<T> runner = firstNode;
```

```
            while(runner.next!=null){
                runner = runner.next;
            }
            runner.next = insertNode;

        }
        length++;
    }

    /**
     * search for first occurrence of value and return index where found
     *
     * @param value string to search for
     * @return index where string occurs (first node is index 0).  Return -1
if value not found.
     */
    public int indexOf(T value)
    {   int index = 0;

        ListNode<T> runner = firstNode;
        while(runner!= null){
            if (runner.data.equals(value)){
                return index;
            }
            else{
                runner = runner.next;
                index ++;
            }
        }
        return -1;

    }

    /**
     * Remove element at specific index. If the linkedlist is empty, do
nothing
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, remove the head of
 the linkedlist
     * If the index is at the end of the linkedlist, remove the tail of the
linkedlist
     * After removing, the length will decrement by 1
     * @param index the index of the element want to remove
     */

    public void removeAtIndex(int index){
        if(!isEmpty()){
            removeAtIndexNonEmpty(index);
        }

    }

    /**
```

```
     * remove the element at the end of the linkedlist
     * if the linkedlist is empty, do nothing.
     * If the linkedlist has one element, then after removing the linkedlist
becomes empty
     * after removing, the length will decrement by 1
     */

    public void removeTail(){
        if(isEmpty()){
            return;
        }
        if(firstNode.next == null){
            firstNode = null;
            length = 0;
            return;
        }
        ListNode<T> runner = firstNode;
        while(runner.next.next!=null){
            runner = runner.next;
        }
        runner.next = null;
        length --;


    }

    /** remove and return data at the head of the list
     *
     *  @return the String the deleted node contains.  Returns null if list
empty.
     */
    public T removeHead()
    {
        if(isEmpty()){
            return null;
        }
        else{
            T remove = firstNode.data;
            firstNode = firstNode.next;
            length--;
            return remove;
        }

    }

    /**
     * Get the Data of the Node at a specified position
     * @param index the position to get the data
     * @return null for invalid index, and the data otherwise
     */


    public T getItemAt(int index){
```

```
        if(!isEmpty() && index >= 0 && index < getLength()){

            return getNodeAt(index).data;
        }
        else{
            return null;
        }
    }

    /**
     * Get the ith Node in the LinkedList
     * @param index the position of the node to get
     * @return the Node at position index
     */
    private ListNode<T> getNodeAt(int index){
        ListNode<T> nodeToGet = firstNode;
        int count = 0;
        while(nodeToGet != null && count < index){
            nodeToGet = nodeToGet.next;
            count++;
        }
        return nodeToGet;



    }

    /**
     * A private helper method to remove the element at specific index of a
non-empty linkedlist
     * Removing by running the pointer to the wanted index and then storing
pointer to the next of node to be deleted
     * After that, unlink the deleted node from linkedlist. After removing,
the length will decrement by 1
     *
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, remove the head of
 the linkedlist
     * If the index is at the end of the linkedlist, remove the tail of the
linkedlist
     *
     * @param index the index of the element want to remove
     */


    private void removeAtIndexNonEmpty(int index){
        if (index == 0) {
            removeHead();
        }
        else if (index == getLength()) {
            removeTail();
        }
        else if (index > getLength() || index < 0){
```

```
            return;
        }
        else {
            ListNode<T> runner = firstNode;
            int counter = 0;
            while(runner!=null && counter < index-1){
                runner = runner.next;
                counter++;

            }

            ListNode<T> temp = runner.next.next;


            runner.next = temp;
            length--;
        }
    }

    /**
     * Insert a new Node at a specified position of the list, with the
specified Data
     * After inserting, the length will increment by 1
     * Inserting by having current node in the wanted index moving right to
one position
     * and having the node with the wanted value be inserted in the wanted
index
     *
     * If the index is invalid (< 0 or > linkedlist.length), then do nothing
     * If the index is at the beginning of the linkedlist, insert at the head
 of the linkedlist
     * If the index is at the end of the linkedlist, insert at the tail of
the linkedlist
     *
     * @param value The data of the new Node
     * @param index The position to insert that new Node
     */
     public void insertAtIndex(T value, int index) {
        ListNode<T> nodeToInsert = new ListNode(value);
        if(isEmpty()){
            firstNode = nodeToInsert;
        }

         if (index == 0) {
             insertAtHead(value);
         }
         else if(index == getLength()) {
             insertAtTail(value);
         }
         else if(index < 0 || index > getLength()){
             return;
         }
         else{
```

```
            ListNode<T> temp = getNodeAt(index - 1).next;
            getNodeAt(index - 1).next = new ListNode(value);
            getNodeAt(index - 1).next.next = temp;
            length ++;
        }
    }

    /**
     * get the data of the firstNode of a LinkedList. If the LinkedList is
empty, return null
     * @return the data of firstNode in generic type
     */
    public T getHead(){
        if(!isEmpty()){
            return firstNode.data;
        }
        else{
            return null;
        }
    }



}
```

```
A+B;
A+B+C;
A+B-C;
A+(B+C);
A+B*C;
(A+B)*C;
A*B+C;
A*B+C*D;
A*B-C;
A*(B+C);
(A+B)/(C-D);
((A+B)*(C-D)+E)/(F+G);
A/B^C-D;
A^(B+C);
A-(B^C);
A+(B-C)^(D+E)/(F+G);
A^B^C;
A*B-C/D^E;
```

```java
package proj4;

/**
 * This class implements Token interface and represents Right Parenthesis ")"
 *
 * @author Emma Vu
 * @version 5/25/2020
 */
public class RightParen implements Token {
    /**
     * PRECONDITION: THERE IS AT LEAST ONE LEFT PAREN IN THE STACK
     * For a non-empty stack,
     * check if there is at least one left parenthesis in the stack by seeing
 the top of the stack.
     * If the top is a left parenthesis, pop it and discard both pair of
parenthesis.
     * Else, pop all The Token out until the top of the stack is a Left
Parenthesis, and add
     * those pop value to returned string.
     * After that, pop that left Parenthesis to discard both Left and Right
Parenthesis and return the string
     * @param s the Stack the token uses, if necessary, when processing
itself.
     * @return  the String resulted by the above operation.
     */
    public String handle(Stack<Token> s){
        String result = "";
        while(!s.peek().toString().equals("(") && !s.isEmpty()){
            result += s.pop();
        }
        s.pop();

        return result;
    }

    /**
     * Print out the representation of Right parenthesis in toString()
     * @return the string representation of Right parenthesis, which is ")"
     */

    public String toString(){
        return ")";
    }


}
```

```java
package proj4;
/**
 * @author: Emma Vu
 * @version: 5/29/2020
 * JUnit test class.  Use these tests as models for your own.
 */
import org.junit.*;


import org.junit.rules.Timeout;

import static org.junit.Assert.*;

import proj4.LinkedList;

public class LinkedListTester {
    @Rule
    // a test will fail if it takes longer than 1/10 of a second to run
    public Timeout timeout = Timeout.millis(100);

    private LinkedList<String> ll;
    @Before
    public void setUp() throws Exception {
       ll = new LinkedList<String>();
    }

    @After
    public void tearDown() throws Exception {
        ll = null;
    }

    @Test
    //Remove empty LinkedList.
    //Return null, should have 0 nodes, the content doesn't change
    public void testRemoveHeadEmpty(){

        assertNull(ll.removeHead());
        assertEquals(0,ll.getLength());
        assertEquals("()",ll.toString());
    }
    @Test
    //Remove LinkedList with one element. Return the only element after
removing
    //Should have 0 nodes, the content will change
    public void testRemoveHeadOne(){

        ll.insertAtHead("A");

        String expected = (String) ll.removeHead();
        assertEquals(0,ll.getLength());
        assertEquals("A",expected);
        assertEquals("()",ll.toString());
```

```
    }
    @Test
    //Remove LinkedList with more than one element
    //The length should decrement by 1. Return the removed element.
    // Content will change
    public void testRemoveHeadNonEmpty(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        String expected = (String) ll.removeHead();
        assertEquals(3,ll.getLength());
        assertEquals("A",expected);
        assertEquals("(B, C, D)",ll.toString());




    }

    @Test
    //Remove the first element in a multiple element LinkedList even though
it is identical to the others.
    //Should remove the exact element in the exact position. Length will
decrement by 1.
    //Should not alter the values besides removing from the LinkedList
    public void removeHeadSameMultiple(){

        ll.insertAtHead("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        ll.insertAtTail("A");
        assertEquals("A",ll.removeHead());
        assertEquals(4,ll.getLength());
        assertEquals("(A, A, A, A)",ll.toString());

    }

    @Test
    //Remove head of a LinkedList that has two identical elements and after
removing there will only be one left
    //The length should decrement by 1. Content will change
    public void testRemoveHeadSameTwice(){

        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals("A",ll.removeHead());
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());
    }
    @Test
```

```
    //Insert an empty LinkedList. Then it became the first element of
LinkedList
    //The length should increment by 1, content will change
    public void testInsertTailEmpty(){

        ll.insertAtTail("A");
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());

    }
    @Test
    //Insert LinkedList with more than one elements
    //The length should increment by 1
    // content will change as the inserted element will be added at the end
    public void testInsertTailNonEmpty(){

        ll.insertAtHead("C");
        ll.insertAtHead("B");
        ll.insertAtHead("A");
        ll.insertAtTail("Z");
        assertEquals(4,ll.getLength());
        assertEquals("(A, B, C, Z)",ll.toString());


    }
    @Test
    //Insert to one element LinkedList
    //The length should increment by 1
    //content will change as the inserted element will be added at the end
    public void testInsertTailOne(){

        ll.insertAtHead("A");
        ll.insertAtTail("Z");
        assertEquals(2,ll.getLength());
        assertEquals("(A, Z)",ll.toString());

    }

    @Test
    //Insert an identical element to a one-element LinkedList.
    // The LinkedList will have identical elements instead of unable to
insert.
    //Length will increment by 1

    public void testInsertTailSameOne(){

        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals(2,ll.getLength());
        assertEquals("(A, A)",ll.toString());


    }
```

```
    @Test
    //Insert tail of a multiple identical element LinkedList. Length and
content should change
    public void testInsertTailSameMultiple(){

        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtHead("F");
        ll.insertAtTail("F");
        assertEquals(6,ll.getLength());
        assertEquals("(F, F, F, F, F, F)",ll.toString());
    }
    @Test
    //IndexOf empty LinkedList. Return -1. Should not change LinkedList
content nor the length
    public void testIndexOfEmpty(){

        assertEquals(-1,ll.indexOf("A"));
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());

    }
    @Test
    //IndexOf invalid data of one element LinkedList. Return -1
    //Should not change LinkedList content nor the length
    public void testIndexOfInvalidOne(){

        ll.insertAtHead("A");
        assertEquals(-1,ll.indexOf("B"));
        assertEquals(1,ll.getLength());
        assertEquals("(A)",ll.toString());



    }

    @Test
    //IndexOf invalid data of more than one element LinkedList. Return -1
    //Should not change LinkedList content nor the length
    public void testIndexOfInvalidMultiple(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        assertEquals(-1,ll.indexOf("E"));
        assertEquals(4,ll.getLength());
        assertEquals("(A, B, C, D)",ll.toString());
```

```
}
@Test
//IndexOf one element LinkedList. Return the index of that data (0)
//Should not change LinkedList content nor the length
public void testIndexOfOne(){

    ll.insertAtHead("A");
    assertEquals(0,ll.indexOf("A"));
    assertEquals(1,ll.getLength());
    assertEquals("(A)",ll.toString());



}
@Test
//IndexOf more than one element LinkedList
//Should return the corresponding indexes when given valid data
//Should not change LinkedList content nor the length
public void testIndexOfMultiple(){

    ll.insertAtHead("A");
    ll.insertAtTail("B");
    ll.insertAtTail("C");
    ll.insertAtTail("D");
    assertEquals(0,ll.indexOf("A"));
    assertEquals(1,ll.indexOf("B"));
    assertEquals(2,ll.indexOf("C"));
    assertEquals(3,ll.indexOf("D"));
    assertEquals(4,ll.getLength());
    assertEquals("(A, B, C, D)",ll.toString());
}
@Test
//IndexOf identical data that appear in the LinkedList more than once.
//Return the first occurrence of that data
//Should not change LinkedList content nor the length
public void testIndexOfSameMultiple(){

    ll.insertAtHead("C");
    ll.insertAtTail("C");
    ll.insertAtTail("C");
    ll.insertAtTail("C");
    ll.insertAtTail("C");
    assertEquals(0,ll.indexOf("C"));
    assertEquals(5,ll.getLength());
    assertEquals("(C, C, C, C, C)",ll.toString());



}

@Test
//Get index of a LinkedList that has two identical elements. Should get
```

```
the correct index (0).
    //Should not change content nor length
    public void testIndexOfSameTwice(){

        ll.insertAtHead("A");
        ll.insertAtTail("A");
        assertEquals(0,ll.indexOf("A"));
        assertEquals(2,ll.getLength());
        assertEquals("(A, A)",ll.toString());
    }

    @Test
    //Test insertAtHead to an empty LinkedList. The inserted element will be
the first element in the LinkedList
    //Content will change, the length will increment by 1
    public void testInsertHeadEmpty(){

        ll.insertAtHead("A");
        assertEquals("(A)",ll.toString());
        assertEquals(1,ll.getLength());

    }
    @Test
    //Test insertAtHead to a LinkedList with one element. The length will
increment by 1. Content will change
    public void testInsertHeadOne(){

        ll.insertAtTail("A");
        ll.insertAtHead("D");
        assertEquals(2,ll.getLength());
        assertEquals("(D, A)",ll.toString());

    }

    @Test
    //Test insertAtHead to a LinkedList with more than one elements. The
content and length will change
    public void testInsertHeadMultiple(){

        ll.insertAtTail("S");
        ll.insertAtTail("D");
        ll.insertAtTail("A");
        ll.insertAtHead("B");
        assertEquals(4,ll.getLength());
        assertEquals("(B, S, D, A)",ll.toString());

    }

    @Test
    //Test insertAtHead to a one element LinkedList by inserting the
identical element
    //The content and length will change.
    //Should insert at the beginning instead of unable to insert
```

```
    public void testInsertHeadSameOne(){

        ll.insertAtTail("D");
        ll.insertAtHead("D");
        assertEquals(2,ll.getLength());
        assertEquals("(D, D)",ll.toString());

    }

    @Test
    //Test insertAtHead of a LinkedList with multiple identical elements.
Content and length should change
    public void testInsertHeadSameMultiple(){

        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtTail("D");
        ll.insertAtHead("D");
        assertEquals(6,ll.getLength());
        assertEquals("(D, D, D, D, D, D)",ll.toString());


    }

    @Test
    //Test isEmpty of an empty LinkedList. Return true
    public void testIsEmptyTrue(){

        assertTrue(ll.isEmpty());

    }

    @Test
    //Test isEmpty of a non-empty LinkedList. Should return false
    public void testIsEmptyFalse(){

        ll.insertAtHead("F");
        ll.insertAtTail("D");
        assertFalse(ll.isEmpty());

    }

    @Test
    //Test getLength of an empty LinkedList. Return 0
    public void testGetLengthEmpty(){

        assertEquals(0,ll.getLength());
    }

    @Test
    //Test getLength of a non-empty LinkedList. Return the length
```

```
    public void testGetLengthNonEmpty(){

        ll.insertAtHead("A");
        ll.insertAtTail("R");
        assertEquals(2,ll.getLength());
    }

    @Test
    //Test toString of empty LinkedList.
    public void testToStringEmpty(){

        assertEquals("()",ll.toString());

    }

    @Test
    //Test toString of multiple elements LinkedList
    public void testToStringMultiple(){

        ll.insertAtTail("A");
        ll.insertAtTail("D");
        ll.insertAtHead("F");
        assertEquals("(F, A, D)",ll.toString());
    }

    @Test
    //Test toString of one element LinkedList
    public void testToStringOne(){


        ll.insertAtHead("F");
        assertEquals("(F)",ll.toString());
    }


    @Test
    //test getItemAt valid position. Return the data corresponding to the
wanted index
    public void testGetItemAtNonEmptyValid(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertEquals("F",ll.getItemAt(5));
        assertEquals("B",ll.getItemAt(1));
        assertEquals("A",ll.getItemAt(0));
        assertEquals("C",ll.getItemAt(2));
        assertEquals("D",ll.getItemAt(3));
        assertEquals("E",ll.getItemAt(4));
```

```
    }

    @Test
    //test getItemAt negative position. Return null
    public void testGetItemAtNegative(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertNull(ll.getItemAt(-1));

    }


    @Test
    //test getItemAt position > the length of LinkedList. Return null
    public void testGetItemAtGreaterThanLength(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        assertNull(ll.getItemAt(10));
    }

    @Test
    //test SearchItemAt an empty LinkedList. Return null no matter the wanted
 position
    public void testGetItemAtEmpty(){

        assertNull(ll.getItemAt(0));

    }
    @Test
    //Test removeTail of a non empty linkedlist. After removing the length
will decrement by 1
    public void testRemoveTailNonEmpty(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeTail();
        assertEquals("(A, B, C)",ll.toString());
        assertEquals(3,ll.getLength());

    }
```

```
    @Test
    //Test removeTail of an empty linkedlist. Should remain the same after
removing
    public void testRemoveTailEmpty(){

        ll.removeTail();
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }

    @Test
    //Test removeTail One Element. After removing the linkedlist becomes
empty
    public void testRemoveTailOne(){

        ll.insertAtHead("A");
        ll.removeTail();
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }

    @Test
    //Test remove at index of a non-empty linkedlist with valid index. After
removing, the length will decrement by 1
    public void testRemoveAtIndexValidNonEmpty(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(1);
        assertEquals(3,ll.getLength());
        assertEquals("C",ll.getItemAt(1));
        assertEquals("(A, C, D)",ll.toString());
        ll.removeAtIndex(2);
        assertEquals("(A, C)",ll.toString());
        ll.removeAtIndex(0);
        assertEquals("(C)",ll.toString());
        ll.removeAtIndex(0);
        assertTrue(ll.isEmpty());
    }

    @Test
    //Test remove at index of an empty linkedlist no matter the index is.
Should remain the same

    public void testRemoveAtIndexEmpty(){

        ll.removeAtIndex(10);
        assertEquals("()",ll.toString());
        assertEquals(0,ll.getLength());
    }
```

```
    @Test

    //Remove at index of one-element linkedlist. After removing, the
linkedlist becomes empty
    public void testRemoveAtIndexOne(){

        ll.insertAtHead("A");
        ll.removeAtIndex(0);
        assertTrue(ll.isEmpty());
    }

    @Test
    //Remove at index with invalid index (negative integer). Should remain
the same

    public void testRemoveAtIndexNegative(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(-1);
        assertEquals("(A, B, C, D)",ll.toString());
    }

    @Test
    //Remove at index with index > the linkedlist.length. Should remain the
same

    public void testRemoveAtIndexGreaterThan(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.removeAtIndex(10);
        assertEquals("(A, B, C, D)",ll.toString());
    }

    @Test
    //Remove at index with index == 0. Remove the head of the linkedlist
    public void testRemoveAtIndexHead(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.removeAtIndex(0);
        assertEquals("(B)",ll.toString());
    }

    @Test
    //Remove at index with index at the end of the linkedlist. Remove the
tail of the linkedlist
    public void testRemoveAtIndexTail(){
```

```
        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.removeAtIndex(1);
        assertEquals("(A)",ll.toString());
    }

    @Test
    //Remove at index with index in the middle of the linkedlist. Remove the
middle element of the linkedlist
    public void testRemoveAtIndexMiddle(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.removeAtIndex(2);
        assertEquals("(A, B, D, E)",ll.toString());
    }









    @Test
    //Insert at index of a non empty linkedlist with valid index. The element
 will be inserted at the wanted index
    public void insertAtIndexValidNonEmpty(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtIndex("E",1);
        assertEquals("(A, E, B, C, D)",ll.toString());
    }

    @Test
    //Insert at index of an empty linkedlist at no matter the index is. The
linkedlist will have one element
    public void insertAtIndexEmpty(){

        ll.insertAtIndex("A",-1);
        assertEquals("(A)",ll.toString());
    }

    @Test
    //Insert at index with negative index of a nonempty linkedlist. Should
remain the same

    public void insertAtIndexNegative(){
```

```
        ll.insertAtHead("Hello");
        ll.insertAtTail("Goodbye");
        ll.insertAtIndex("Hi",-1);
        assertEquals("(Hello, Goodbye)",ll.toString());
    }

    @Test
    //insert at index greater than linkedlist.length. Should remain the same

    public void insertAtIndexGreater(){

        ll.insertAtHead("Hello");
        ll.insertAtTail("Goodbye");
        ll.insertAtIndex("Hi",10);
        assertEquals("(Hello, Goodbye)",ll.toString());
    }



    @Test
    //Insert at index with index == 0. Insert at the head of the linkedlist
    public void testInsertAtIndexHead(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtIndex("C",0);
        assertEquals("(C, A, B)",ll.toString());
    }

    @Test
    //Insert at index with index at the end of the linkedlist. Insert at the
tail of the linkedlist
    public void testInsertAtIndexTail(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtIndex("C",2);
        assertEquals("(A, B, C)",ll.toString());
    }

    @Test
    //Remove at index with index in the middle of the linkedlist. Remove the
middle element of the linkedlist
    public void testInsertAtIndexMiddle(){

        ll.insertAtHead("A");
        ll.insertAtTail("B");
        ll.insertAtTail("C");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtIndex("F",2);
        assertEquals("(A, B, F, C, D, E)",ll.toString());
```

```
    }

    @Test
    //remove tail of a non-empty linkedlist with same elements multiple times
. Length and contents will change
    public void testRemoveTailSameMultiple(){

        ll.insertAtHead("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.insertAtTail("B");
        ll.removeTail();
        ll.removeTail();
        ll.removeTail();
        assertEquals("(B, B, B, B)",ll.toString());
        assertEquals(4,ll.getLength());
    }

    @Test
    //remove tail of 2 same element linkedlist. Length and contents will
change
    public void testRemoveTailSameTwice(){

        ll.insertAtHead("A");
        ll.insertAtTail("A");
        ll.removeTail();

        assertEquals("(A)",ll.toString());
        assertEquals(1,ll.getLength());
    }

    @Test
    //remove tail multiple times of a non-empty linkedlist. Contents and
length will change
    public void testRemoveTailMultiple(){

        ll.insertAtHead("B");
        ll.insertAtTail("D");
        ll.insertAtTail("E");
        ll.insertAtTail("F");
        ll.insertAtTail("G");
        ll.insertAtTail("H");
        ll.insertAtTail("I");
        ll.removeTail();
        ll.removeTail();
        ll.removeTail();
        assertEquals("(B, D, E, F)",ll.toString());
        assertEquals(4,ll.getLength());

    }
```

```
}
```