01286222 / 05506006 Lab 5 Name………………………………………………… id …………………………

Objective(s):

a. To practice array-based data structure.

b. To be familiar with Reverse Polish Notation creation process.

c. To be able to create customed LinkedList data structure

d. Students are able to demonstrate their understanding on implementing Shunting Yard Algorithm.

Task 1. Create sub package of solutions named code5_Postfix. Implement your stack using array called **MyStackA.java** using lecture slides. (Do not expand from MyArray.java previously created).

MyStackA has the following methods :

- void push(double d)
- double pop()
- double top()
- boolean isFull()
- boolean isEmpty()

```
public class MyStackA {
    private int MAX_SIZE = 100;
    private double [] stack = new double[MAX_SIZE];
    /* your code */
    @Override
    public String toString() {
        StringBuffer sb = new StringBuffer();
        sb.append("top->");
        for(int i=top-1; i>=0; i--) {
            sb.append("[");
            sb.append(stack[i]);
            sb.append("]->");
        }
        sb.append("bottom");
        return new String(sb);
    }
}
```

```
static void task1() {
    MyStackA stack = new MyStackA();
    stack.push(1.0);
    stack.push(2.2);
    stack.push(4.4);
    stack.push(3.3);
    System.out.println(stack);
}
```

Task 2 Implement MyRPN.java which contains static double computeRPN(String postfixString)

Notice

- how to process each token of the StringTokenizer st.

- how regular expression of Pattern pattern is used.

- you may supply your customed postfix string when calling L5_RPN_Main i.e., java L5_RPN_Main "3 1 – 4 5 + *". Else the default postfix string would be "8 5 - 4 2 + 3 / *"

```
public class MyRPN {
    private static Pattern pattern =
                         Pattern.compile("-?\\d+(\\.\\d+)?");
    public static boolean isNumeric(String strNum) {
        if (strNum == null)
            return false;
        return pattern.matcher(strNum).matches();
    }
    public static double computeRPN(String rpn) {
.       /* your code */
        return 0.0;
    }
}
```

```
public class Lab5_RPN_Main {
    static void task2(String postfixString) {
        System.out.println(postfixString);
        System.out.println("= " + MyRPN.computeRPN(postfixString));
    }
    public static void main(String[] args) {
        // task1();
        // 3 1 - 4 5 + *
        String postfixString = "8 5 - 4 2 + 3 / *";
        task2(postfixString);
    }
}
```

(For the sake of simplifying the lab's technical difficulty, let's use a new Node.java class working with String (instead of modifying the previous lab int type attribute of nested Node class).

Task 3: Complete **MyStackL.java**. The structure of this class employs Node tail attribute so that accessing the last node can be achieved easily.

```
// figure 1 for MyStackL
public class Node {
  String value;
  Node next;

  public Node(String d) {
    value = d;
    next = null;
```

```
public class MyStackL {
    private Node top;
    public MyStackL() {
        top = null;
    }
    public void push(String d) {
      /* your code */
    }

    public String pop() {
        /* your code */     }

    // Top()
    public String peek() {
      /* your code */
    }
```

```
  public boolean isFull() {
      return false;
  }

  public boolean isEmpty() {
      return top == null;
  }


  @Override
  public String toString() {
    StringBuilder sb = new
                          StringBuilder();
    sb.append("Top->");
    Node temp = top;
    while (temp != null) {
        sb.append(temp.value).append("->");
        temp = temp.next;
    }
    sb.append("Bottom");
    return sb.toString();
  }
}
```

Notice that instead of extending MyLinkedList, we could customize the add() and remove() method to push() and pop() without any hassle.

Task 4: To implement a queue, we could

a) Create a MyQueueL from scratch just like MyStackL but we lose the java's collections features such as iterable.

b) Create a queue structure directly by allocating a LinkedList object and use it to behave just like a natural queue. The disadvantage is there is no method such as enqueue() or dequeue() to abstract the queue capability. OOP allows you to **extends** LinkedList class. However the callers must be coupled with this list structure.

```
// figure 2 queue via extends LinkedList

public class MyQueueExtendsLinkedList<T>
                extends LinkedList<T> {

    public void enqueue(T d) {
        this.add(d);            }
    public T dequeue() {
        return this.poll();     }
    public T top() {
        return this.peek();     }
   …
}
```

```
// figure 3 before making a class
iterable

public class Type1 <T> {

    private ArrayList<T> items =
                    new ArrayList<>();
    private int count;

    public void add(T item) {
        items.add(item);
        count++;
    }
    public T get(int index) {
        return items.get(index);
    }
}
```

```
public class Type2 <T> implements
Iterable<T> {
  private ArrayList<T> items = new
                        ArrayList<>();
  private int count;

  public void add(T item) {
    items.add(item);
    count++;
  }
  public T get(int index) {
    return items.get(index);
  }

  @Override
  public Iterator<T> iterator() {
    return new AnyItemsIterator(this);
  }

  private class AnyItemsIterator
              implements Iterator <T> {
    private Type2<T> lis;
    private int idx;
    public AnyItemsIterator(Type2<T> arg) {
      this.lis = arg;
    }
    @Override
    public boolean hasNext() {
        return (idx < lis.count);
    }
    @Override
    public T next() {
        return lis.items.get(idx++);
    }
  }
}
```

c) Make our class iterable. This allows callee to adopt any structure yet allowing the caller traverse the callee's structure. Hence, in this task, we will construct a **MyQueueListWrap.java**, by wrapping a LinkedList, which implements Iterable to this class. Study the figure3.

Type1 and Type2 difference is that

Type2's caller cannot iterate for each element though ArrayList is a java collection (,while type1 cannot.). This can be done simply by mapping it to an object of Iterator (lis) and implement required Iterator interface, the hasNext() and next(). This way, your choice of data collection is decouple from the caller. i.e. you can you LinkedList<T> items and the caller would have no trouble with your choice.

Also notice how our created collection can use generic type. No need to concern the type of data in Node.

Task 5: use the Shunting Yard pseudo code to complete **MyShuntingYard.java**

```java
public class MyShunting {
    private static int order(String c) {
        return switch (c) {
            case "+", "-" -> 1;
            case "*", "/" -> 2;
            default -> 0;
        };
    }
    public static String infixToPostfix(String infixString) {
        MyQueueListWrap<String> queue = new MyQueueL();
        MyStackL stack = new MyStackL();
        String resultPostfixString = "";
        StringTokenizer st = new StringTokenizer(infixString);
        while (st.hasMoreTokens()) {
            String t = st.nextToken();
            if (MyRPN.isNumeric(t))
                queue.enqueue(t);
            else if (t.equals("(")) {
                stack.push(t);
            } else if (t.equals(")")) {
                while (!stack.peek().equals("(")){
                    queue.enqueue(stack.pop());
                }
                stack.pop(); //discard "("

            } else {
                if(!stack.isEmpty()) { /
                    while (!stack.isEmpty() && order(stack.peek()) >= order(t)) {
                        queue.enqueue(stack.pop());
                    }
                }
                 /* your code */
            }
            System.out.println("current q = " + queue.dumpToString());
        }
        while (!stack.isEmpty()) {
            queue.enqueue(stack.pop());
        }
        resultPostfixString = queue.dumpToString();
        return resultPostfixString; //"happy coding";
    }
}
```

Notice that our infix calculator handles only + - * and /, all of which has left-associativity property.

Test the String infixToPostfix(String infix) with the below code.

```
    // computeInfix("( 4 + 2 ) / 3 * ( 8 - 5 )"); // in main

    public static void computeInfix(String infixString) {
        String postfixString = MyShunting.infixToPostfix(infixString);
        double ans = MyRPN.computeRPN(postfixString);
        System.out.println(ans);
    }
```

Submission:

MyStackA_XXYYYY.java and MyRPN_XXYYYY.java

MyQueueListWrap _XXYYYY.java, MyStackL.java and MyShuntingYard_XXYYYY.java

Due date: TBA