HW2 Written

Christophe Rimann: cjr2185

October 7th, 2016

Q.1

(Working code on next page)

```java
//Takes in params l (list to be searched on) and p (list of
//indices to search for), and print out each value at each
//index.
public static void printLots(List<Integer> l, List<Integer> p) {
                //Create the iterators
                Iterator<Integer> lIterator = l.iterator();
                Iterator<Integer> pIterator = p.iterator();

                //The iterator for l must be set to the first value, or
                //else the for loop later will not work as it is a <
                lIterator.next();

                //Creating our variables
                int lastPosition = 0;
                int val = 0;

                //while the pIterator has next, iterate to that value of
                //the lIterator, and print that value out.
                while (pIterator.hasNext()) {
                        int iterateTo = pIterator.next();
                        for (int i = 0; i < (iterateTo - lastPosition); i++) {
                                val = lIterator.next();
                        }
                        System.out.println(val);
                        lastPosition = iterateTo;

                }
        }
```

Q.2

I wrote the following code to show the logic; it is not actual java code but pseudocode (even if it looks like java code). The basic approach was as follows: start at the beginning of each list, if one list is larger than the other, iterate the one with the smaller value until it is greater than or equal to the first. If its equal to, add it to a new list and iterate both. If its greater than, iterate the other list until its greater than or equal to. (Pseudocode on following page)

```
import java.util.ArrayList;
import java.util.Iterator;

public class Written2<AnyType> {

        public ArrayList<AnyType> intersection(ArrayList<AnyType> L1,
                        ArrayList<AnyType> L2) {
                ArrayList<AnyType> matching = new ArrayList<AnyType>();

                // Create our iterators
                Iterator<AnyType> L1Iterator = L1.iterator();
                Iterator<AnyType> L2Iterator = L2.iterator();

                // Start our iterators on
                AnyType L1val = L1Iterator.next();
                AnyType L2val = L2Iterator.next();

                // As long as neither of the iterators has
                // reached the end, keep testing
                while (L1Iterator.hasNext() & L2Iterator.hasNext()) {
                        // Test to see if the values are equal;
                        // if so add to the matching list
                        if (L1val.equals(L2val)) {
                                matching.add(L1val);
                                L1val = L1Iterator.next();
                                L2val = L2Iterator.next();
                        }
                        // If L1 is bigger than L2, only iterate L2
                        // (which makes it bigger)
                        else if (L1val.compareTo(L2) > 0) {
                                L2val = L2Iterator.next();
                        }
                        // If L2 is bigger than L1, only iterate L1
                        // (which makes it bigger)
                        else if (L2val.compareTo(L1) > 0) {
                                L1val = L1Iterator.next();
                        }
                }
                // Because the loop quits when the iterator is at
                // the end, it does not test the last 2 values.
                // So we need to test it one last time.
                if (L1val.equals(L2val)) {
                        matching.add(L1val);
                }
                return matching;
        }

}
```

Q.3

The basic approach was as follows: each stack starts at one end of the array and moves towards the middle by incrementing the value posit1 and decrementing posit2. As such, one stack is moving right from the left and one stack is moving left from the right. When popped, each stack moves towards its side, returning the rightmost value if coming from the left and the leftmost value if coming from the right. isEmpty just checks to see if the posit1 is ==0 (in which case the left array holds nothing) and posit2 == length-1 (in which case the right array holds nothing). (Working code for this implementation is available on the next page).

```
public class TwoStack<AnyType>{
        private int stack1length, stack2length;
        private AnyType[] mainArr;
        private int posit1, posit2;

        //Constructor: create a new array to hold the 2 stacks
        //and initialize the position of each stack at either side of
        //the array.
        public TwoStack(int length){
                mainArr = (AnyType[]) new Object[length];
                posit1=0;
                posit2=length-1;
        }

        //push takes 2 parameters: list (which stack it should be
        //added to) and addition, which is what it should be added
        //to. If list is 1, add it to the next available spot on the
        //left side of the array; if 2, add it to the next available
        //spot on the right.
        public void push(AnyType addition, int list) throws StackOverflowError
                //If the two are equal, it could either mean that it's
                //overflowing or at the end. Check for both cases; if its
                //at the end, add it, if its overflowing, throw an error.
                if (posit1 == posit2) {
                         if (isEmpty(1) && list == 2) {
                                mainArr[posit2--] = addition;
                        } else if (isEmpty(2) && list == 1) {
                                mainArr[posit1++] = addition;
                        } else
                                throw new StackOverflowError();
                }

                //If one stack makes up the whole array, and the array is full,
                //Throw an error
                else if(posit1 > posit2) {
                        throw new StackOverflowError();
                }
                //Otherwise add it normally.
                else {
                        if (list == 1) {
```

```
                        mainArr[posit1++] = addition;
                } else {
                        mainArr[posit2--] = addition;
                }
        }
}

//pop takes parameter stack, which signifies which
//stack should be popped. If stack is 1, return and remove
//the value of the first full position on the left; if 2,
//return and remove the value of the first full position
//on the right
public AnyType pop(int stack){
        if(stack==1){
                if (isEmpty(1)){
                        return null;
                }
                else return mainArr[--posit1];
        }
        else{
                if (isEmpty(2)){
                        return null;
                }
                else return mainArr[++posit2];
        }
}

//peek takes parameter stack, which signifies which
//stack should be popped. If stack is 1, return
//the value of the first full position on the left; if 2,
//return the value of the first full position
//on the right
public AnyType peek(int stack){
        if(stack==1){
                if (isEmpty(1)){
                        return null;
                }
                else return mainArr[posit1-1];
        }
        else{
                if (isEmpty(2)){
                    return null;
                }
                else return mainArr[posit2+1];
            }
        }

        //isEmpty takes parameter stack which signifies which
        //stack should be popped. If stack is 1, check value of
        //posit 1; if 0, that stack is empty. Else, check value of
        //posit 2 vs length of array - 1; if they're equal stack is
        //empty.
        public boolean isEmpty(int stack){
            if (stack==1){
                return posit1==0;
            }
            else{
                return ((mainArr.length-1)==posit2);
            }
        }
}
```

Q.4

A

We can utilize the following moves on [5, 9, 6, 7, 2, 8, 1, 3, 4] to produce the output [9, 8, 7, 6, 5, 4, 3, 2, 1]:

$$4 \rightarrow S1 \tag{1}$$
$$3 \rightarrow S1 \tag{2}$$
$$1 \rightarrow OtherSide \tag{3}$$
$$8 \rightarrow S2 \tag{4}$$
$$2 \rightarrow OtherSide \tag{5}$$
$$7 \rightarrow S2 \tag{6}$$
$$6 \rightarrow S2 \tag{7}$$
$$9 \rightarrow S3 \tag{8}$$
$$POP3 \rightarrow OtherSide \tag{9}$$
$$POP4 \rightarrow OtherSide \tag{10}$$
$$5 \rightarrow OtherSide \tag{11}$$
$$POP6 \rightarrow OtherSide \tag{12}$$
$$POP7 \rightarrow OtherSide \tag{13}$$
$$POP8 \rightarrow OtherSide \tag{14}$$
$$POP9 \rightarrow OtherSide \tag{15}$$

B

We just have to swap the order of 6 and 7 and it will not be accepted anymore: [5, 9, 7, 6, 2, 8, 1, 3, 4].