Or Aboodi

Oda2102

Data Structures Hw 2, written part

1. Java method of printLots(L,P): (must import java.util.List; beforehand)

```
public static <AnyType> void printLots(List<AnyType> L, List<Integer> P) {
        for (int idx : P) {
                if (idx >= L.size()) {
                        throw new IndexOutOfBoundsException();}
                // if any index in P is too large, an exception is thrown

                Iterator<AnyType> l = L.iterator();
                //create a new iterator each P integer

                for (int i=0; i<idx; i++){
                        l.next();
                        //go through the iterator until one before the P index
                }
                //print the item at that index
                System.out.println(l.next());
        }
}
```

2. Problem 3.4 in Weiss. The way I implemented this method is linear time because the while loop stops when the smaller list is traversed first.

```
public static List<AnyType> intersection(List<AnyType> L1, List<AnyType> L2) {

//AnyType should be the same and extend comparable!

  List <AnyType> L3 = new List<AnyType>(min(L1.size(), L2.size());

  //a new list is created with the size of the smaller of the two lists

  //while the notation is not necessarily right in java, the idea is understandable as the new list only needs the size of the smaller of the input lists.

  //this could be any list, whether an arraylist or another that implements the List interface.

  int count1 = 0; int count2 = 0; //initialize two counters for both lists

  int count3 = 0; //start a counter for adding new elements to the third list

  while (count1<L1.size() & count2<L2.size()) { //only go until one list is traversed

    if (L1.get(count1).compareTo(L2.get(count2)) == 0) {

      L3.add(count3++, L1.get(count1));

      count1++; count2++; }

      //if the elements are the same, add them to the list and increment all 3 list counters

    else { //if the elements are not the same

      if (L1.get(count1).compareTo(L2.get(count2)) == -1)

        count1++; //if the second element is greater, keep going on the first list

      else {

        count2++; }}}//if the first element is greater, keep going on the second list

  return L3; }
```

3. Weiss 3.24- the way I implemented this was having one stack start at the beginning of the array, and one stack start at the end of the array moving backwards. I have trackers for the ending of each stack, but the trackers are one above the actual last element. This way, the overflow occurs when the end tracker of the first stack is one after the end tracker of the second stack and a push tries to occur.

```
public class TwoStackArray {

  //instance variables:

  AnyType[] arr; //the array for the stacks, with AnyType of object in it
```

```
int end1;
int end2; //the last element for each stack (indexes)
//constructor method:
public TwoStackArray( int sizeOfArray){
   if (sizeOfArray < 2)
        sizeOfArray = 2; //make sure the array Is at least 2 elements long (just for ease of use)
   arr = new AnyType[] (sizeOfArray); //initialize the new array of anytype.
   end1 = 0; end2 = arr.length-1;} //the end of the stacks are the indexes of the edges of the array

public AnyType pop1() {
   if (arr[end1 – 1] == null)
      //throw a null pointer exception (something went wrong)
   if (end1 == 0)
      //also throw an exception (we have reached the end of the stack)
   return arr[--end1]; // decrement end1 and return the element at its new position
public AnyType pop2(){
   if (arr[end2+1] == null)
      //throw a null pointer exception, something went wrong
   if (end2 == arr[arr.length-1])
      //also throw an exception (we have reached the end of the stack)
  return arr[++end2]; // increment end2 and return the element at its new position
public void push1(AnyType x){
   if (end1 – 1==end2) //if the stacks caused the array to be full already: the indexes are no longer pointing to nulls, but are pointing to the first element of the other stack. This is ok up until now because the pointers end1 and end2 point to the index after the actual last element. There is no overlap yet.
      //throw new overflow exception
   arr[end1] = x; //add the new element
   end1++; //increment the end of the stack.
public void push2(AnyType x){
```

```
   if (end1 – 1==end2) //if the stacks caused the array to be full already (same as push1)
      //throw new overflow exception
   arr[end2] = x; //add the new element
   end2--; //decrement the end of the stack
//the following are not needed, just for fun
//size method that checks the amount of objects in the array
public int size() { //returns the amount of total elements in the list, not including null
   return end1 + arr.length() – 1 – end2; //returns the total of both stacks of elements
//checks if the array is empty
public boolean isEmpty() {
   return size() == 0; }
//restarts the indexes of the array
public void makeEmpty();
   end1=0; end2=arr.length -1; //reset the indexes to their beginning points, as if the array is empty
```

4. A) the following are the sequence of steps:
   4 to S1, 3 to S1, 1 to output, 8 to S2, 2 to output, 3 to output, 4 to output, 7 to S2,
   (just to keep track- we are with (4,3,2,1) in the output, and (7,8) in S2.)
   6 to S2, 9 to S3, 5 to output, 6 to output, 7 to output, 8 to output, 9 to output. Done.

   b) [1,9,7,8,6,5,4,3,2] is an example of such a train. Once 2, 3, 4 are on the holding tracks, 5 cannot go on any holding track without later messing up the pattern, and cannot go on the output already.