**Conor Sweeney – cjs2201**

**COMS W3134 Spring 2016 (Sections 1 and 2)**

**Homework 1**

**Due: 4:00pm on Friday, February 19th**

**Written (40 pts)**

For the written section of this assignment, type up your answers and submit a computer based document to us. You can submit MS Word doc files, pdf files, or txt files.

1. Problem 1 (10 pts):   
   Describe how you would swap two adjacent elements by adjusting only the links (and not the data) in a
   * (a) singly linked lists.

/\*My method accepts only one Node value and swaps with the next value. This eliminates the need to check that the values are adjacent. Therefore, it simply checks to make sure the given Node is not the tail and has a forward pointer. Singly list nodes cannot swap values with the previous nodes since they only have pointers forward.\*/

swapAdjacentNodes (Node givenNode){

if (givenNode.next !=null){

Node tempNode = givenNode.next;

givenNode.next = tempNode.next;

tempNode.next = givenNode;

givenNode = givenNode.next;

}

}

* + (b) doubly linked lists.

/\*This time I will swap the previous node to show that the logic is still the same. To swap a doubly linked it follows the same logic as swapping singly linked lists except it needs to be done twice to swap the pointers to the forward and previous nodes.\*/

swapAdjacentNodes (Node givenNode){

if (givenNode.previous !=null){

Node previousNode = givenNode.previous;

previousNode.next = givenNode.next;

previousNode.previous.next = node;

givenNode.next.previous= previousNode;

givenNode.next = previousNode;

previousNode.previous = givenNode;

}

}

Note: For the singly linked list you are given the reference to the head node. For the doubly linked list, you are given the reference to the head and tail node. In both case you are given the index number that you are looking for, so you will have to include the search step for finding the appropriate node reference.

1. Problem 2 (10 pts): Weiss 3.8 parts a, b, and c

The following routine removes the first half of the list passed as a parameter:

public static void removeFirstHalf( List<?>1st)

{

int theSize = 1st.size() / 2;

for( int i = 0; I < theSize; i++)

1st.remove(0);

}

1. Why is theSize saved prior to entering the for loop?
   * So that the correct size of the list is saved and can be used to limit the for loop to only remove the first half of the list. If it was within the for loop, then the size would change every time that the loop is run.
2. What is the running time of removeFirstHalf if 1st is an ArrayList?
   * O(N)
   * The loop is simply removing the first element of the array each time. It is not increasing or decreasing in speed. The size does not matter since it is always removing the element in the beginning.
3. What is the running time of removeFirstHalf if 1st is a LinkedList?
   * O(N)
   * See above.
4. Problem 3 (10 pts): Weiss 3.24 - you only need to provide pseudocode for this problem

Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.

* An array with n values can have two stacks implemented at both ends. The first stack will begin at array [0] and increase (array[1], array[2],… etc), while the second stack will begin at array[n] and decrease (array [n-1], array [n-2],… etc).

public class TwoStacksUsingArray {

private int size;

private int firstStackTop;

private int secondStackTop;

private int array[];

public TwoStacksUsingArray(int size){

if(size<2){

System.out.println("Error: Invalid Array Size");

}

this.size = size;

array = new int[size];

firstStackTop = 0;

secondStackTop = size – 1;

}

public void push (int stackId, int data){

if (secondArrayTop == firstTop+1){

System.out.println(“Array is full”);

}

if (stackId == 1){

array[++firstStackTop]= data;

}

else if (stackId == 2){

array[--secondStackTop]= data;

}

else{

return;

}

}

public int pop (int stackId, int data){

if (stackId == 1){

if(firstStackTop == 0){

System.out.println(“Stack 1 is empty”);

}

int popData = array[firstStackTop--];

return popData;

}

else if (stackId == 2){

if(secondStackTop == this.size-1){

System.out.println(“Stack 2 is empty”);

}

int popData = array[secondStackTop++];

return popData

}

else{

return -1;

}

}

public int peek (int stackId){

if (stackId ==1){

if (firstStackTop == 0){

System.out.println(“Stack 1 is empty”);

}

int popData = array[firstStackTop];

return popData;

}

else if(stackId == 2){

if (secondStackTop == this.size -1){

System.out.println(“Stack 2 is empty”)

}

int popData = array[secondStackTop];

return popData;

}

else{

return -1;

}

}

public boolean isEmpty(int stackId){

if(stackId==1){

return (firstStackTop == 0);

}

else if (stackId == 2){

return(secondStackTop == this.size-1);

}

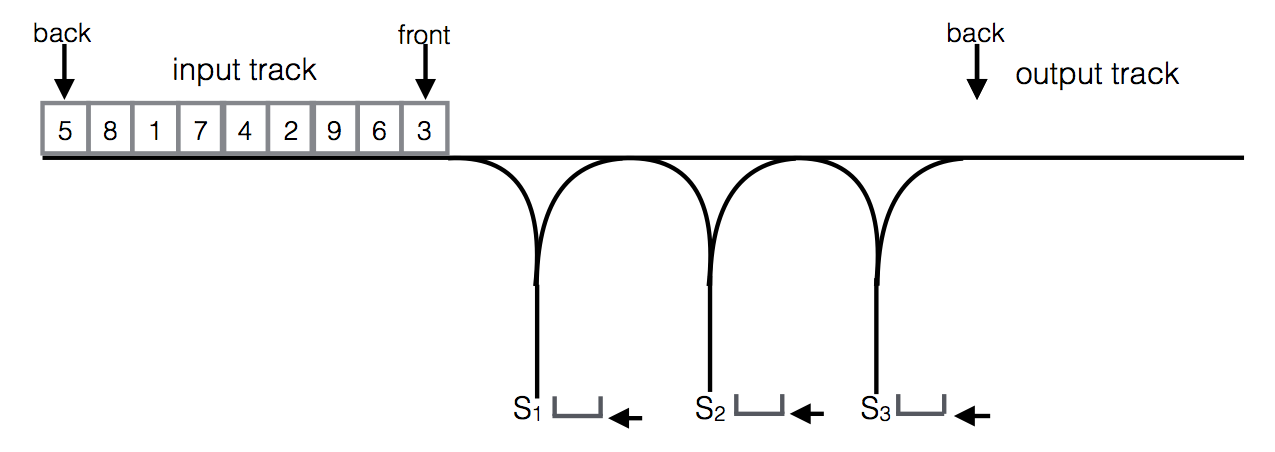
else{

return true;

}

}

}

1. Problem 4 (10 pts):  
   In the MTA Subway system, occasionally cars on a train need to be re-arranged. For instance, assume we label the cars of a train with the number [5,8,1,7,4,2,9,6,3] (the right end of the list is the front of the train - in this case 3), and we would like to arrange the cars like this: [9,8,7,6,5,4,3,2,1].  
   For this purpose the MTA has special *shunting* yards. A shunting yard consists of an *input track*, an *output track*, and *holding tracks*. The input and output track operate like queues (in first in first out fashion). The holding tracks have to be used like stacks (in a last in first out fashion): the last train car entering the holding track is the first car to leave the track.  In this case there are 3 holding tacks called: *S₁, S₂, and S₃.*  
     
   To reorganize a train, the train dispatcher can use only the following operations:
   * Move the car at the front of the input track to the back of of the output track.
   * Move the car at the front of the input track to the top of one of the holding tracks.
   * Move the car at the top of one of the holding tracks to the back of the output track.

Cars cannot be moved between the holding tracks. Once a car is on a holding track it cannot move back to the input track (since the input track is a queue). The goal is always to organize the train cars in increasing order (front to back on the output track, as in the example).  
Answer the following questions:

* + (a) Provide a solution for this specific input train and 3 holding tracks as a sequence of steps.
  + [5,8,1,7,4,2,9,6,3] 🡪[9,8,7,6,5,4,3,2,1
    1. Move car 3 from input track to S3,
    2. Move car 6 from input track to S2,
    3. Move car 9 from input track to S1,
    4. Move car 2 from input track to S3,
    5. Move car 4 from input track to S2,
    6. Move car 7 from input track to S1,
    7. Move car 1 from input track to output track,
    8. Move car 2 from S3 to output track,
    9. Move car 3 from S3 to output track,
    10. Move car 4 from S2 to output track,
    11. Move car 8 from input track to S3,
    12. Move car 5 from input track to output track,
    13. Move car 6 from S2 to output track,
    14. Move car 7 from S1 to output track,
    15. Move car 8 from S3 to output track,
    16. Move car 9 from S1 to output track,
  + (b) Show an example for a train of length 9 that cannot be rearranged in increasing order using 3 holding tracks.

Whenever the car labels in a holding track are not in increasing order from top to bottom then the car rearrangement cannot be completed. This means that any beginning order arranged in such a way that it is impossible for the cars to avoid this will not work.

[1,9,8,7,6,5,4,3,2] is a perfect example. The logical arrangement begins with moving 2 to S3, 3 to S2, 4 to S1, however 5 is larger than all three of those values. Which means that it would have to be moved on top of 2, 3, or 4 which makes it impossible to rearrange the cars following these rules.