**Practice for Mid-Term Test**

**Using Fundamental Data Structures (Arrays, Linked Lists), Stacks, and Queues**

References: Chapter 3,4,5,6 lecture slides, lab assignments. This material provides the necessary information that you need to complete the exercises.

**Exercise 1**

C-3.25: Write a method for **concatenating two singly linked lists L and M**, into a single list **L'** that contains all the nodes of **L** followed by all the nodes of **M**.

**Hint**: Traverse list L until you reach the last element. Then, make the last element of L point to the first element of M as its “next” node.

Let’s use the SinglyLinkedList implementation from Lesson 2 slides

tail

head

TO

LA

BO

VA

Ø

tail

head

TX

CI

AT

ST

Ø

We need to traverse the list L to reach the last node. Then, we need to link the last node of L to first node of M. This is easy, knowing that *head* of M already is the required link. The code below creates the concatenate method inside SinglyLinkedList class:

**public** **static** <E> SinglyLinkedList<E> concatenate(SinglyLinkedList<E> L, SinglyLinkedList<E> M)

{

//Create a new node v

Node<E> v = **new** Node<E>(**null**, **null**);

v = L.head; //point to head of L

//walk to the end of list L

**while**(v.getNext() != **null**)

v = v.getNext();

//link to list M

v.setNext(M.head);

SinglyLinkedList<E> C = L;

**return** C;

}

//main method

**public** **static** **void** main(String[] args)

{

//create list L

SinglyLinkedList<String> L = **new** SinglyLinkedList<String>();

L.addFirst("MSP");

L.addLast("ATL");

L.addLast("BOS");

//

L.addFirst("LAX");

System.***out***.println(L);

//create list M

SinglyLinkedList<String> M = **new** SinglyLinkedList<String>();

M.addFirst("OTA");

M.addLast("TOR");

M.addLast("CAL");

M.addFirst("VAN");

System.***out***.println(M);

//concatenate L with M

System.***out***.println(*concatenate*(L,M));

}

**Exercise 2**

R-3.6: Write a method for finding the second-to-last node in a singly linked list in which the last node is indicated by a null next reference.

tail

head

TX

CI

AT

ST

Ø

Again, we need to traverse the list. However, this time we stop at the node before the last node. Note the use of getNext() method two times in the loop condition, to ensure that we reach the second-to-last node. Add this method to SinglyLinkedList class.

**private** Node<E> penultimate( ) {

**if** (size < 2)

**throw** **new** IllegalStateException("list must have 2 or more entries");

Node<E> walk = head;

//walk until the second-to-last node is reached

**while** (walk.getNext().getNext() != **null**)

walk = walk.getNext();

**return** walk;

}

//main method

**public** **static** **void** main(String[] args)

{

SinglyLinkedList<String> list = **new** SinglyLinkedList<String>();

list.addFirst("MSP");

list.addLast("ATL");

list.addLast("BOS");

//

list.addFirst("LAX");

System.***out***.println("the list:" + list);

System.***out***.println("second last in the list: "+ list.penultimate().getElement());

System.***out***.println("tail element in the list: "+list.tail.getElement());

//

}

**Exercise 3**

C-5.17: Write a short recursive Java method that takes a character string s and outputs its reverse. For example, the reverse of 'pots&pans' would be 'snap&stop'

Let ***n*** be the index of last character in a string ***s***. For example, if s="ab", then n=1. String method *charAt (int index),* returns the character given its index in a string.

The statement **s.charAt(n)** will return character ’b’. The statement **s.charAt(n-1)** will return “a”. You can see how this can become a recursive call. The **stopping point is n=0**. See the code below:

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

String s = "pots&pans";

*printReverse*(s);

}

/\*

\* input – a string s

\* output - its reverse.

\* parameters:

\* string s

\* int n - index of last character n in the string s

\*

\*/

**static** **void** printReverse(String s, **int** n) {

//for a string with two characters "ab", n=1

//this method will:

// print "b"

// call itself with n=0

// print "a"

**if** (n >= 0) { //stop condition is n=0

System.***out***.print(**s.charAt(n)**);

*printReverse***(s, n-1)**;

}

}

//

**static** **void** printReverse(String s) {

*printReverse***(s, s.length() - 1)**;

}

**Exercise 4**

R-6.5: Give a **recursive method for removing all the elements from a stack**. **Hint:** First check if the stack is already empty. If the stack is empty, then return (the stack is empty). Otherwise, **pop** the top element from the stack **and recur**.

Let’s use the linked list implementation of the stack, hence a LinkedStack. Elements are inserted (**push**) and removed (**pop**) at the front. To create a recursion from the process of repeatedly popping the elements off the stack (naturally a loop would do that), we need a stopping condition, at which point we cannot remove more elements (stack becomes empty). The method *isEmpty()* serves for that purpose. Therefore, call a **pop** operation and recur (the method calls itself).

J

I

A

K

Here is the Java code:

**static** <E>**void** removeAllStackElements(LinkedStack<E> stack)

{

**if**(stack.isEmpty()) //stop condition

**return**;

**else**

{

stack.pop();

*removeAllStackElements*(stack); //recur

}

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

LinkedStack<String> stack = **new** LinkedStack<String>();

stack.push("java");

stack.push("python");

stack.push("javascript");

System.***out***.println(stack.toString());

*removeAllStackElements*(stack);

System.***out***.println(stack.toString());

}