CIS 351 – Activity wk10a

Q1. Stack-Based Expression Evaluation

- Convert the the following infix expression to postfix form:
    - $7\times2+3*2-10 7\times2+3*2-10$

```
72*32*+1072*- 32*+10-
```

- Show the steps of using a stack to evaluate the postfix expression. Show the contents of the stack before and after each operation is performed.

	Symbol	Stack	Postfix string
1.	7		7
2.	*	*	7
3.	2	*	72
4.	+	+	72*
5.	3	+	72*3
6.	*	+*	72*3
7.	2		72*32*+
8.	-	-	72*32*+
9.	107	-	72*32*+107
10.		*	-* 72*32*+107
11.		2	- 72*32*+1072*
12.		+	+ 72*32*+1072*-
13.		3	+ 72*32*+1072*-3
14.		*	+* 72*32*+1072*-3
15.		2	+ 72*32*+1072*-32*+
16.		-	- 72*32*+1072*-32*+
17.		10	72*32*+1072*-32*+10-

Q2. Matching Parentheses

Write pseudocode for an algorithm that checks a string to see if it contains correctly matched pairs of brackets and parentheses. Test your algorithm on the following inputs: **"[()]", "[]()", "[(])", "((", "))".**

```
  ALGORITHM: CheckMatches
      INPUT:  I - A string containing only the characters '[', ']', '(' and ')'.
      OUTPUT: True if the string contains only correctly matched brackets
              and parentheses, False otherwise.
String char;
For(i=0, i<inputSize, i++){
        (if the character is equaled to the [ , ], (, or )
                Return true;
        }
        Else{
                Return false;
        }
}
```

Q3. Queue Using Stacks

Implement the Queue ADT using only stacks to store information. Analyze the running time of each operation.

```
// Declare instance variables...

StackQueue() {
```

```
Front = 0;
Size = 0;

}

void enqueue(E item) {
    queueArray[size] = item;
    size++;

}

E dequeue() {
if(size == 0) {
      return null;
    }
    else {
      E newFront = queueArray[front - 1];
      size--;
      front++;
      return newFront;

}
```

Q4. Stack Using Queues

Implement the Stack ADT using only queues to store information. Analyze the running time of each operation.

```
// Declare instance variables...

QueueStack() {
Front = 0;
Size = 0;


}
void push(E item) {
stackArray[size] = item;
size++;


}

E pop() {
if(item == 0) {
      return null;
    }
    else {
      E newFront = stackArray[front - 1];
      size--;
      front++;
      return newFront;


}
```

Q5. List Using Stacks

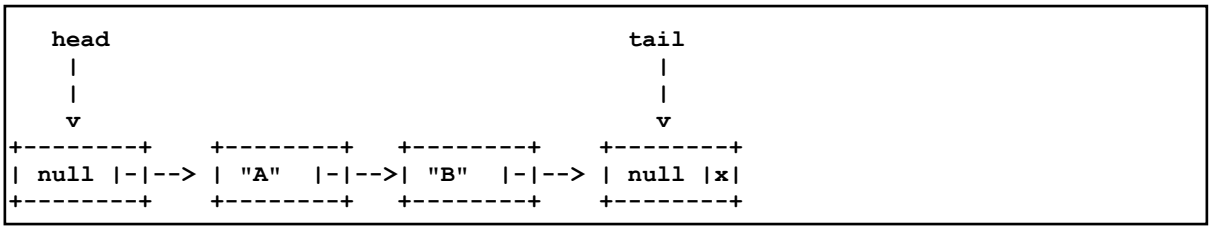
Implement the following subset of the List ADT using only Stacks to store information. Analyze the running time of each operation.

```
// Declare instance variables...
Stack<String> stack = new Stack<String>();
StackList() {
Front = 0;
Size = 0;

}

void insert(E item) {
item.add()



}

E remove() {
For(I = 0; I < stacksize; i++)
E(i).remove


}
void moveToPos(int pos) {
if(pos > 0 && pos<listSize)
        stack.next


}
```

Q6. Tracing Linked List Code: Carefully read the **Link** Class:

```
class Link<E> {          // Singly linked list node class
    private E e;           // Value for this node
    private Link<E> n;     // Point to next node in list
    // Constructors
    Link(E it, Link<E> inn) { e = it; n = inn; }
    Link(Link<E> inn) { e = null; n = inn; }

    E element() { return e; }                          // Return the value
    E setElement(E it) { return e = it; }              // Set element value
    Link<E> next() { return n; }                       // Return next link
    Link<E> setNext(Link<E> inn) { return n = inn; } // Set next link
}
```

The **`insertAtPos`** method below is *broken*. Assuming that the figure below represents the state of memory, draw the state of memory after **`list.insertAtPos(1, "C")`** is called. How should the method be fixed? **It != null;**



```
public boolean insertAtPos(int pos, E it) {
// Return false if the position is invalid
if ((pos < 0) || (pos > listSize)) {
return false;
}
// Find the insertion node
Link<E> current = head.next();
for (int i = 0; i < pos; i++) {
current = current.next();
}
// Insert
Link<E> newLink = new Link<E>(it, current);
current.setNext(newLink);
if (tail == current) {
tail = current.next(); // New tail
}

listSize++;
return true;
}
```

Q7. Analyzing List Algorithms: Analyze each of the methods below according to the instructions in the Javadoc comments.

a)

```
/* What is the big-Theta running time of this method, assuming that list is
 *
 * an ArrayList: (n)/2
 *
 *
 * a LinkedList: (n+1)/2
 *
 */
public static int sumByIndex(List<Integer> list) {

    int sum = 0;
    for (int i = 0; i < list.size(); i++) {
        sum += list.get(i);
    }
    return sum;
}
```

b)

```
/* What is the big-Theta running time of this method, assuming that list is
 *
 * an ArrayList: n
 *
 * a LinkedList: n/2
 *
 */
public static int sumWithIterator(List<Integer> list) {

    int sum = 0;
    for (int curValue : list) {
        sum += curValue;
    }
```

```
    return sum;

}
```

c)

```
/* What is the big-Theta running time of this method, assuming that toList
 * is initially empty, and that both lists are of type
 *
 * ArrayList: n/2
 *
 * LinkedList: logn
 *
 */
public static void copy(List<Integer> fromList,
                        List<Integer> toList) {

    for (int item : fromList) {
        toList.add(item);
    }
}
```

d)

```
/* What is the big-Theta running time of this method, assuming that toList
 * is initially empty, and that both lists are of type
 *
 * ArrayList: n/3
 *
 * LinkedList: nlogn
 *
 */
public static void copyReverseA(List<Integer> fromList,
                                List<Integer> toList) {

    for (int item : fromList) {
        toList.add(0, item);
    }

}
```