Com S 228
Spring 2015
Exam 2

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

ISU NetID (username): _____

Recitation section **(please circle one):**

1. R 10:00 am (Monica, Chen-Yeou)
2. R 2:10 pm (Caleb B, Blake)
3. R 1:10 pm (Caleb V, Ryan)
4. R 4:10 pm (Yuxiang, Anthony)
5. R 3:10 pm (Jacob, Andrew)
6. T 9:00 am (Chen-Yeou, Brian)
7. T 2:10pm (Chris, Austin)

Closed book/notes, no electronic devices, no headphones. Time limit **60 minutes**. Partial credit may be given for partially correct solutions.

- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

If you have questions, please ask!

Question	Points	Your Score
1	28	
2	20	
3	16	
4	36	
Total	100	

1. (28 pts) The `main()` method below executes a code snippet after the initialization of a `List` object. On the next page, you will see several snippets of code, each to be executed **within a separate call** of the `main()` method.

```
public static void main(String[] args) throws NoSuchElementException,
                                               IllegalStateException
{
      List<String> aList;
      ListIterator<String> iter, iter2;

      // initialization
      aList = new ArrayList<String>();
      aList.add("A");
      aList.add("B");
      aList.add("C");
      aList.add("D");
      aList.add("E");

      // code snippet
      // ...
}
```

Note that each snippet is *separate* and executed *independently* right after the initialization.

For each snippet,

a) show what **the output** from the `println` statement is, if any, and

b) draw **the state of** `aList` **and the iterator** after the code executes, and

c) do **not** display any other list that may appear in the code.

However, if the code throws an exception, do **not** draw the list but instead write down the exception that is thrown. In this case, **also show the output**, if any.

Use a bar (**|**) symbol to indicate the iterator's logical cursor position. For example, right after the statement

```
iter = aList.listIterator();
```

the list would be drawn as follows.

|A B C D E

(the first one has been done for you as an example). If two iterators appear in the code, **label** (or draw or color) **them** differently.

Suggestion: For **partial credit**, you may also want to draw the intermediate states of the list and iterator after executing every one or few lines of code in a snippet.

Code snippet	Output	List and iterator state, or exception thrown
`iter = aList.listIterator();`	(none)	|A B C D E

`// 3 pts`
`iter = aList.listIterator();`
`iter.next();`
`System.out.println(iter.previousIndex());`		

`// 5 pts`
`iter = aList.listIterator();`
`while (iter.hasNext())`
`  System.out.println(iter.next());`
`System.out.println(iter.next());`		

`// 4 pts`
`aList.add("O");`
`iter = aList.listIterator(aList.size()/2);`
`while (iter.hasPrevious())`
`{`
`  iter.set("X");`
`  iter.remove();`
`  iter.previous();`
`}`		

`// 8 pts`
`iter = aList.listIterator(aList.size());`
`List<String> bList = new ArrayList<String>();`
`while (iter.hasPrevious())`
`{`
`  bList.add(iter.previous());`
`  iter.remove();`
`}`
`iter = bList.listIterator();`
`while (iter.hasNext())`
`  aList.add(iter.next());`		

`// 8 pts`
`iter = aList.listIterator();`
`iter2 = aList.listIterator(aList.size());`
`while (iter.nextIndex() < iter2.previousIndex())`
`{`
`  String s = iter.next();`
`  String t = iter2.previous();`
`  iter.set(t);`
`  iter2.set(s);`
`}`
`iter.next();`
`iter.remove();`		

2. (20 pts)

a) Give the big-*O* time complexity of the following two implementations of the set operation from the Java `List` interface. Suppose that the underlying list contains n elements.

i) (6 pts)

```
DoublyLinkedList.set(int index, E element)
```

ii) (6 pts)

```
ArrayList.set(int index, E element)
```

b) (8 pts) Suppose that `arr` is an array consisting of `n` integers and `s` is a stack of integers, which is initially empty, and is implemented using a linked list. What is the worst-case time complexity of the following pseudocode?

```
for (int i = 0; i < n; i++)
    if (arr[i] > 0)
        s.push(arr[i]);
    else
        while (!s.empty())
            s.pop();
```

3. (16 pts) Infix and postfix expressions.

a) (8 pts) Convert the following infix expression into postfix.

```
2 ^ a ^ b * c - ((d + e) * 2 + f ^ (g - h)) - 5 / (i + j)
```

Postfix:

b) (8 pts) Convert the following postfix expression into infix.

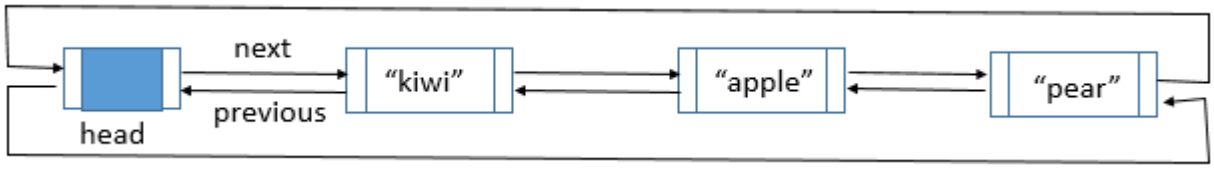
```
a b * 2 c d ^ ^ / 3 e f - g h - * 5 + * -
```

Infix:

4. (36 pts) Complete one condition and five methods in the `DLinkedList` class below. The class represents a circular doubly-linked list (CDLL) with **a dummy** `head` **node**. If the list is empty, then the following conditions must hold:

```
head.next == head && head.previous == head
```

Show below is a CDLL list that stores three `String` objects.



Here are some facts about the data stored in a CDLL:

a) They are **distinct**, where equality is determined by some available `compareTo()` method.
b) They are **not ordered**.

You are asked to complete the implementation of the class `CDLinkedList`. Please pay attention to the following:

a) Do **not** use `equals()` for comparison since `compareTo()` is available.
b) In the implementation of the `rearrange()` method, two stack objects will be used. You will not need other Stack methods than `push()` and `pop()`.

```
import java.util.Stack;

// (2 pts) fill in the condition on E below so that compareTo() is available
// to E objects.
public class CDLinkedList <_____________________________________________>
{
      private class Node
      {
            public E data;
            public Node next;
            public Node previous;
      }

      private Node head;  // dummy node
      private int size;

      public CDLinkedList()
      {
            head = new Node();
            head.next = head;
            head.previous = head;
            size = 0;
      }
```

```
/**
 * (2 pts) Unlink a node from the list without updating size.
 * Precondition: node != head
 */
private void unlink(Node node)
{
      // insert code below (2 pts)




}

/**
 * (4 pts) Insert newNode into the list after cur without
 * updating size.
 *
 * Precondition: cur != null && newNode != null
 */
private void link(Node cur, Node newNode)
{
      // insert code below (4 pts)









}


/**
 * (8 pts) Add to the beginning of the linked list.  Do not add
 * if an equal value is already in the list, as determined by
 * compareTo().
 *
 * @param  val
 * @return true   if an insertion takes place
 *         false  otherwise
 */
public boolean add(E val)
{
      // search the linked list for val.
      Node node = head.next;
      while (node != head)
      {
```

```
            // insert the condition in the if statement below (2 pts)

            if (_____________________________________________________)
                  return false;

            // insert one line of code below (2 pts)



      }

      // if val is not found, then create a node and add it right after the
      // dummy node.

      // insert code below (4 pts)









      return true;
}


/**
 * (6 pts) Search for the node storing a given value, and remove it
 * if found.
 *
 * @param  val
 * @return true   if val is found
 *         false  otherwise
 */
public boolean remove(E val)
{
      // search the linked list for val.
      Node cur = head.next;

      // insert condition in the while statement below (2 pts)

      while (_____________________________________________________________)

            cur = cur.next;

      // insert condition in the if statement below (2 pts)

      if (_____________________________________________________________)
```

```
            return false;  // val not found
      else
      {
            // insert code below (2 pts)




            return true;
      }
}


/**
 * (14 pts) Rearrange the list.  The parameter val splits the nodes
 * into two groups.  The first group includes those nodes that store
 * data less than or equal to val, and the second group includes
 * those nodes that store data greater than val.  After the
 * rearrangement, the following two conditions must be satisfied:
 *
 *     a) The nodes from the first group must precede those from the
 *        second group.
 *     b) Within each group, the original order of the nodes is
 *        reversed.
 *
 * You are asked to use two provided stacks for the task.
 *
 */
public void rearrange(E val)
{
      // declare two stacks
      Stack<E> stk1 = new Stack<E>();
      Stack<E> stk2 = new Stack<E>();

      // split the values stored in the linked list using the stacks.
      // remove these values from the list in the same time.
      Node cur = head.next;
      while(cur != head)
      {
            // insert condition in the if statement below (2 pts)

            if (_____________________________________________________)

                  // insert a line of code below (2 pts)



            else
                  // insert a line of code below (2 pts)
```

```
            // insert code below (2 pts)




        }

        // pop the values out of the two stacks to reconstruct the
        // linked list.
        while (!stk1.empty())
        {
            // insert code below (3 pts)









        }

        while (!stk2.empty())
        {
            // insert code below (3 pts)









        }
    }


}
```

Appendix A: Excerpt from List documentation, for reference.

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list. Returns `true` if the element is added.
void	**add**(int index, E element) Inserts the element at the specified position in this list. Throws `IndexOutOfBoundsException` if index is less than zero or greater than `size()`.
void	**clear**() Removes all of the elements from this list.
E	**get**(int index) Returns the element at the specified position in this list. Throws `IndexOutOfBoundsException` if the index is less than zero or is greater than or equal to the size of the list.
int	**indexOf**(Object obj) Returns index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position . Throws `IndexOutOfBoundsException` if the index is less than zero or is greater than size.
E	**remove**(int index) Removes and returns the element at the specified position in this list. Throws `IndexOutOfBoundsException` if index is less than zero or is greater than or equal to size of the list.
boolean	**remove**(Object obj) Removes the first occurrence of the specified element from this list, if it is present. Returns `true` if the list is modified.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element. Throws `IndexOutOfBoundsException` if index is less than zero or is greater than or equal to the `size()`
int	**size**() Returns the number of elements in this list.

Excerpt from Iterator documentation, for reference.

Method Summary

boolean	**hasNext**() Returns `true` if the iteration has more elements.
E	**next**() Returns the next element in the iteration. Throws `NoSuchElementException` if there are no more elements in the collection.
void	**remove**() Removes from the underlying collection the last element returned by `next()`. Throws `IllegalStateException` if the operation cannot be performed.

Excerpt from Collection documentation, for reference.

Method Summary

boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object obj) Returns true if this collection contains the specified element.
boolean	**isEmpty**() Returns true if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present
int	**size**() Returns the number of elements in this collection.

Excerpt from ListIterator documentation, for reference.

Method Summary

void	**add**(E e) Inserts the specified element into the list.
boolean	**hasNext**() Returns `true` if this list iterator has more elements in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements in the reverse direction.
E	**next**() Returns the next element in the list. Throws `NoSuchElementException` if there are no more elements in the forward direction.
int	**nextIndex**() Returns the index of the element that would be returned by `next()`.
E	**previous**() Returns the previous element in the list. Throws `NoSuchElementException` if there are no more elements in the reverse direction.
int	**previousIndex**() Returns the index of the element that would be returned by `previous()`.
void	**remove**() Removes from the list the last element that was returned by `next` or `previous`. Throws `IllegalStateException` if the operation cannot be performed.
void	**set**(E e) Replaces the last element returned by `next` or `previous` with the specified element. Throws `IllegalStateException` if the operation cannot be performed.

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)