Com S 228
Spring 2014
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 (Chris, Caleb B)
2. R 2:10 pm (Bryan, Ben)
3. R 1:10 pm (Jesse, Monica)
4. R 4:10 pm (Caleb V, Brad)
5. R 3:10 pm (Kyle, Nick)
6. T 9:00 am (Brady, Ade)
7. T 2:10pm (Kyle, Shana)

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

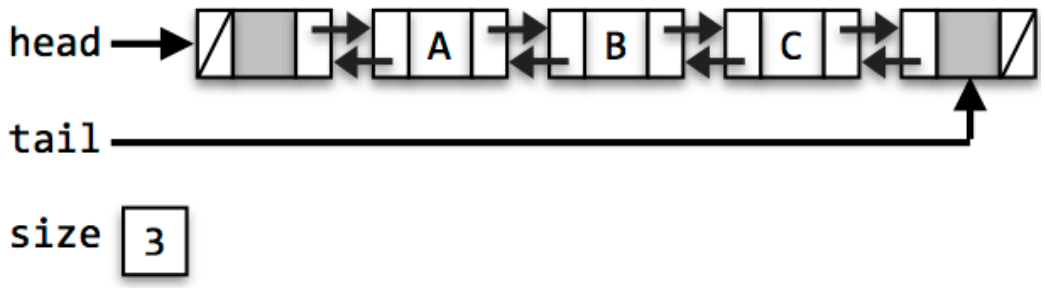
- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

It helps to *peel off* pages 11-18 from your exam sheets for code lookup convenience and scratch purpose.

If you have questions, please ask!

Question	Points	Your Score
1	24	
2	28	
3	48	
Total	100	

1. (24 pts) The next questions refer to the **doubly-linked list** implementation of the List interface, seen in class, called **`DoublyLinkedList`**. Recall that objects of type **`DoublyLinkedList`** have a **head** node and a **`tail`** node, as well as a **`size`** field. An example of such a list appears below.



Recall also that **`DoublyLinkedList`** implements all methods of the **`ListIterator`** interface.

In the next questions, unless stated otherwise, assume that the list has n elements.

a) (3 pts) Give the big-*O* time complexity of the following **`List`** API operation.

```
public int size()
```

b) (4 pts) Give the big-O time complexity of the following **`List`** API operation.

```
public boolean contains(Object obj)
```

c) (4 pts) Give the big-O time complexity of the following **`ListIterator`** API operation.

```
public void set(E item)
```

d) (6 pts) Give the big-O time complexity of the following code snippet.

```
List<String> c = new DoublyLinkedList();

for (int i = 0; i < n; i++)
      c.add("Y");

ListIterator<String> iter = c.listIterator();
while (iter.hasNext())
{
      String s = iter.next();
      System.out.println(s);
      iter.set("Z");
}
```

e) (7 pts) Give the big-O time complexity of the following code snippet.

```
List<String> c = new DoublyLinkedList();

for (int i = 0; i < n; i++)
      c.add("Y");

for (int i = 0; i < n; i++)
{
      String s = c.get(i);
      System.out.println(s);
}
```

2. (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
     // ...
}

public static void print(List<String> aList, int cursorIndex)
{
      if (aList == null) return;

      int counter = 0;
      ListIterator<String> iter = aList.listIterator();
      while (iter.hasNext())
      {
            if (counter == cursorIndex)
               System.out.print("| ");

            System.out.print(iter.next() + " ");
            counter++;
      }

      if (counter == cursorIndex)
           System.out.print("| ");

      System.out.println();
}
```

Note that each snippet is *separate* and executed *independently* right after the initialization. For each snippet, show what **the output** is, if any. 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. The output of the first code snippet has been given.

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
`iter = aList.listIterator();` `print(aList, iter.nextIndex());`	| A B C D E

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

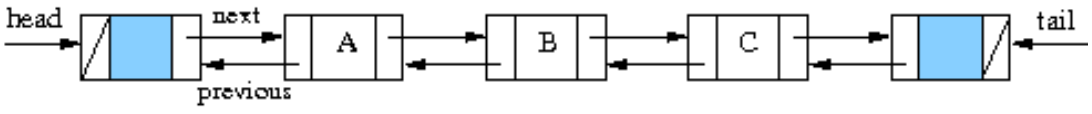
`// 5 pts`
`iter = aList.listIterator();`
`iter.remove();`
`print(aList, iter.nextIndex());`	

`// 6 pts`
`iter = aList.listIterator();`
`while (iter.hasNext())`
`{`
`  iter.set(iter.next() + iter.previous());`
`  print(aList, iter.nextIndex());`
`  iter.next();`
`}`	

`// 6 pts`
`iter = aList.listIterator();`
`while (iter.hasNext())`
`{`
`  iter.add(iter.next());`
`  print(aList, iter.nextIndex());`
`}`	

`// 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);`
 	
`  print(aList, iter.nextIndex());`
`  print(aList, iter2.nextIndex());`
`}`	

3. (48 pts) Refer to the **DoublyLinkedList<E>** class in Appendix B that was discussed in class. Note that only partial implementation is given in the appendix. As illustrated in the example below, the class keeps two dummy nodes: **head** and **tail**.



To answer each part of this question, you should **not** use any of the methods from the original class whose implementation is omitted in the appendix.

a) (4 pts) Implement the private helper method **unlink** that detaches (i.e., removes) a node **current** from the list.

```
private void unlink(Node current)
{
      // TODO









}
```

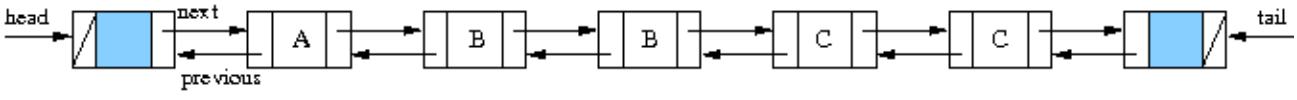
b) (16 pts) Implement a private method **findNodeAhead** that takes a node **target** and a non-negative integer **offset**, and returns the node which is ahead of **target** by **offset** nodes. The two dummy nodes **head** and **tail** are not counted; and once **tail** is reached, the next node will be **head.next**.

For example, in the linked list shown above, assume that the variables **nodeA**, **nodeB**, **nodeC** reference the nodes storing **"A"**, **"B"**, **"C"**, respectively. Then

```
      findNodeAhead(nodeA, 0)  returns  nodeA
      findNodeAhead(nodeA, 2)  returns  nodeC
      findNodeAhead(nodeB, 2)  returns  nodeA
      findNodeAhead(nodeC, 101)  returns  nodeB



private Node findNodeAhead(Node target, int offset)
{
      if (offset < 0)
            throw new IllegalArgumentException("Offset must be non-negative.");


      // TODO









}
```

c) (28 pts) Within the **DoublyLinkedList<E>** class, add a method **duplicateGreaterElements**, which takes as input a value **value**, and duplicates every node in the list that stores a value *greater than* **value**. Every duplicate node must be next to the original node in the list. The method also takes a supplied **Comparator** object **comp** defined in the class **E** or some superclass of **E**.

For example, the same list from b), after a call of **duplicateGreaterElements** with **value == "A"** and a **Comparator** object that does the default string comparison, grows to the list below.



Please make note of the following:

- A comparator comp is supplied.
- **Fill a wildcard type** in the blank preceding the comp.
- The list may be **empty**.
- For you convenience, the implementation breaks down into steps.
- The comments before each step outline what the step does. It is helpful to follow them.
- **Do not use iterators.**

To be helpful, a template for the method is given below. You are **encouraged to follow the template** by simply filling in the blank spaces.

```
public boolean duplicateGreaterElements(E value,
                  _____________________________________ comp)   // (5 pts)
{
      // TODO

      // initialization before a traversal (2 pts)
      Node current = _____________________________________;

      // list traversal
      while (_____________________________________)    // 2 pts
      {
            // fetch the data value stored at the visited node.   // 2 pts
```

```
            // duplicate the node if its data value is greater.

            if (_____________________________________)  // 3 pts
            {

                  // create a copy node (2 pts)









                  // insert the node into the list by updating links.
                  // do not use any helper method.  (6 pts)

                  // other related updates (4 pts)

            }

            // advance to the next node in the original list (2 pts)

      }

      return true;
}
```

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.

Appendix B: Partial sample code for the DoublyLinkedList class

```
public class DoublyLinkedList<E> extends AbstractSequentialList<E>
{

	/**
	* Reference to dummy node at the head.
	*/
	private Node head;


	/**
	* Reference to dummy node at the tail.
	*/
	 private Node tail;


	/**
	* Number of elements in the list.
	*/
	private int size;


	/**
	* Constructs an empty list.
	*/
	public DoublyLinkedList()
	{
		head = new Node(null);
		tail = new Node(null);
		head.next = tail;
		tail.previous = head;
		size = 0;
	}


	/**
	 *   Public methods
	 */
	//...


	/**
	 *   Override iterator() and listIterator() methods
	 */
	// ...



	/**
	 *   Helper methods
	 */
```

```
	// ...


	/**
	 * Doubly-linked node type for this class.
	 */
	private class Node
	{
		public E data;
		public Node next;
		public Node previous;

		public Node(E data)
		{
			this.data = data;
		}
	}


	/**
	 * Implementation of ListIterator for this class
	 */

	private class DoublyLinkedIterator implements ListIterator<E>
	{
	   // ...
	}
}
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)