Com S 228
Fall 2016
Exam 2

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: _____

ISU NetID (username): _____@iastate.edu

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

1. M 11:00 am (Anthony, Chandler)
2. M 10:00 am (Robert, Yuxiang)
3. M 4:10 pm (Andrew, Ryan K)
4. T 11:00 am (Kevin, John)
5. T 10:00 am (Blake, Kelsey)
6. T 4:10 pm (Lige, Ryan Y)
7. W 4:10 pm (Alex, Susan)
8. W 11:00am (Mike, Jake)

Closed book/notes, no electronic devices, no headphones. Time limit **75 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	25	
2	20	
3	18	
4	37	
Total	100	

1. (25 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

`// 2 pts`
`aList.add("X");`
`aList.remove("F");`		

`// 4 pts`
`iter = aList.listIterator(2);`
`iter.previous();`
`iter.remove();`
`System.out.println(iter.previousIndex());`		

`// 5 pts`
`iter = aList.listIterator(aList.size());`
`while (iter.hasPrevious())`
`{`
`  iter.previous();`
`  if (iter.hasNext())`
`  System.out.println(iter.next());`
`  iter.previous();`
`}`		

`// 4 pts`
`aList.remove("A");`
`iter = aList.listIterator();`
`iter.next();`
`iter.set("X");`
`iter.remove();`
`iter.set("Y");`		

`// 6 pts`
`iter = aList.listIterator();`
`iter2 = aList.listIterator(1);`
`while (iter2.hasNext())`
`{`
`  iter.next();`
`  iter.set(iter2.next());`
`  System.out.println(iter.previous());`
`}`		

`// 4 pts`
`iter = aList.listIterator(aList.size() - 2);`
`while (iter.hasNext())`
`{`
`  iter.previous();`
`  iter.remove();`
`}`		

2. (20 pts)

a) Give the big-*O* worst-case time complexity of the following two implementations of the `remove()` method from the Java `List` interface. Suppose that the underlying list contains *n* elements.

i) (6 pts)

`DoublyLinkedList.remove(Object obj)`

ii) (6 pts)

`ArrayList.remove(Object obj)`

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++)
{
    while (!s.isEmpty() && arr[i] > s.peek())
        s.pop();
    s.push(arr[i]);
}
```

3. (18 pts) Infix and postfix expressions.

a) (9 pts) Convert the following infix expression into postfix by filling the row of boxes below **from left to right**, with each filled box containing **exactly one** operand, operator, or parenthesis. (You may end up with some unfilled boxes on the right).

```
3 * (a + b + c) - (e ^ 2 + f * g) / h + i ^ 3
```

Postfix:

b) (9 pts) Convert the following postfix expression into infix by filling the row of boxes below in the same way as specified for part a). (Again, you may end up with some unfilled boxes.)

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

Infix:

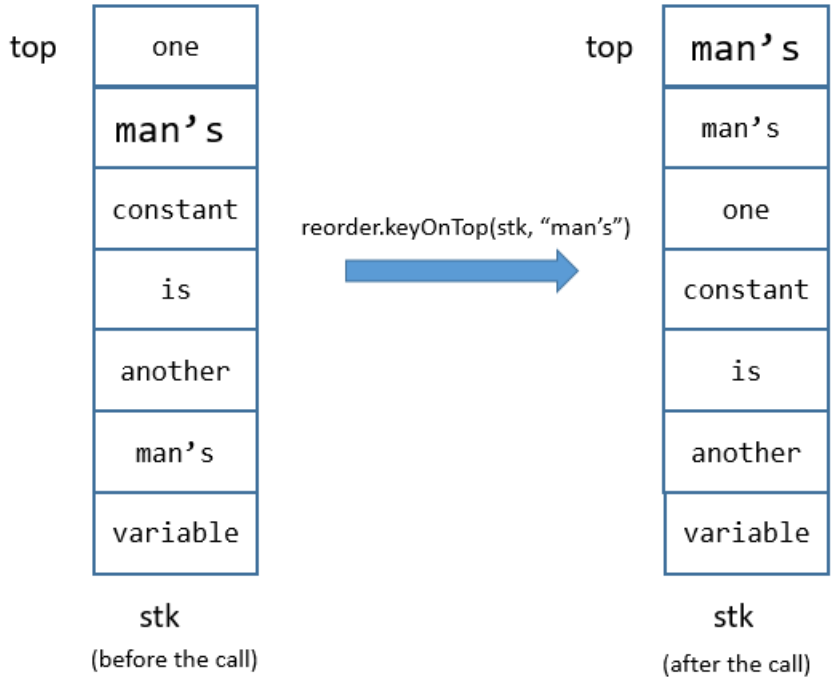
4. (37 pts) Implement a generic class `StackReorder` to reorder the contents of an existing stack. The class provides a method `keyOnTop()`, which takes as input a stack `stk` and a value `key`. The value `key` and the elements on the stack `stk` are objects of the class `E`, whose superclass (possibly itself) implements the `Comparable` interface. After execution, the stack `stk` will have been reorganized to satisfy the following conditions:

a) all the elements on `stk` that are equal to `key` (as determined according to the implementation of `Comparable`) will appear above those elements that are not equal to key;

b) all the elements equal to `key` will preserve their relative order on the stack before the method call;

c) all the elements not equal to `key` will also preserve their relative order before the call.

On the left of the figure below shows that the input stack `stk` stores seven String objects. Two of the objects "`man's`" are equal, but displayed in different font sizes for distinguishing purpose. An object named `reorder` of the class `StackReorder` invokes the call `keyOnTop(stk, "man's")`. The post-call contents of the stack are displayed on the right below, with the two "man's" on the top (and the one displayed in a larger font still above the other in a smaller font) and the remaining String objects below them with their original order unaltered.



If the value of `key` does not appear on the stack, then the stack is not changed after the call. For the same stack `stk` on the left of the above figure, a call `reorder.keyOnTop(stk, "procedure")` would result in no change to `stk`.

You are asked to implement the class `StackReorder`, which makes use of the classes `PureStack` and `ArrayBasedStack` discussed in class and implemented in Appendix B. The method `keyOnTop()` will make use of a stack and a singly linked list, which is an object of the private class `SimpleList` that you also need to implement. Please carefully read the javadocs before each method and between the lines, and follow their instructions in your implementation.

```
import java.util.NoSuchElementException;

// This class is parametrized with the type E, whose superclass must have implemented
// Comparable. Fill a wildcard type in the blank below to make comparison possible
// between two objects of type E.

public class StackReorder<_____________________________________________> // 5 pts
{

      /**
       * Search for all the elements on a stack that are equal to key.  Reorder
       * the stack such that these elements are on the top while preserving their
       * original order.  The remaining elements not equal to key must preserve
       * their original order as well.
       *
       * @param stk
       * @param key
       */
      public void keyOnTop(PureStack<E> stk, E key)
      {
            // initialize
            //    a) a stack as an object of the ArrayBasedStack class;
            //    b) a list as an object of the inner SimpleList class.
            //
            // fill in the blanks on the right hand sides of the next two assignments.

            PureStack<E> tempStk = _____________________________________________; // (1 pt)

            SimpleList tempList = _____________________________________________; // (1 pt)

            // perform a loop to store
            //    a) elements from the stack stk equal to key on the stack tempStk,
            //    b) other elements from the stack in the list tempList.
            //
            // insert code below (6 pts)
```

```
        // merge elements from tempStk and tempList onto the stack stk.
        //
        // insert code below (6 pts)









}


// singly-linked list for temporary storage
private class SimpleList
{
        private Node head;
        private int size;

        /**
         * default constructor
         */
        public SimpleList()
        {
                // insert code below (3 pts)









        }


        /**
         * Create a new node to contain a provide item. Insert the node to the
         * front of the list.
         *
         * @param item  to be added
         */
```

```
void add(E item)
{
      // insert code below (4 pts)









}


/**
 * Remove the first node.
 *
 * @return data stored in the removed node if the list is not empty
 * @throws IllegalStateException if the list is empty
 */
E remove() throws IllegalStateException
{
      // check if the list is empty.
      //  fill in the blank below  (1 pt)

      if (_____________________________)

      {
          // insert code below (2 pts)




      }

      // size != 0
      // insert code below (6 pts)
```

```
        }

        /**
         *
         * @return true  if the list is empty
         */
        boolean isEmpty()
        {
              // insert code below (2 pts)




        }


        // fully implemented class
        public class Node
        {
            public E data;
            public Node next;

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

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: Additional Code for Problem 4.

```
public interface PureStack<E>
{
    /**
     * Adds an element to the top of the stack
     */
    void push(E item);

    /**
     * Removes and returns the top element of the stack. Throws
     * NoSuchElementException if the stack is empty
     */
    E pop();

    /**
     * Returns the top element of the stack without removing it. Throws
     * NoSuchElementException if the stack is empty
     */
    E peek();

    /**
     * Return true if the stack is empty, false otherwise
     */
    boolean isEmpty();

    /**
     * Returns the number of elements in the stack.
     */
    int size();
}
```

```
import java.util.Arrays;
import java.util.NoSuchElementException;

/**
 * Sample implementation of a stack using an expandable array as the backing
 * data structure.  Elements are added and removed at the end of the array.
 */
public class ArrayBasedStack<E> implements PureStack<E>
{
    private static final int DEFAULT_SIZE = 10;

    /**
     * Index of next available cell in array.
     */
    private int top;
```

```
/**
 * The data store.
 */
private E[] data;


/**
 * Constructs an empty stack.
 */
public ArrayBasedStack()
{
    // Unchecked warning is unavoidable
    data = (E[]) new Object[DEFAULT_SIZE];
}


@Override
public boolean isEmpty()
{
    return top == 0;
}


@Override
public E peek()
{
    if (top == 0) throw new NoSuchElementException();
    return data[top - 1];
}


@Override
public E pop()
{
    if (top == 0) throw new NoSuchElementException();
    E ret = data[--top];
    data[top] = null;
    return ret;
}


@Override
public void push(E item)
{
    checkCapacity();
    data[top++] = item;
}
```

```
    @Override
    public int size()
    {
        return top;
    }


    /**
     * Ensures that the backing array has space to store at least
     * one additional element.
     */
    private void checkCapacity()
    {
        if (top == data.length)
        {
            // create a copy of the data array with double the capacity
            data = Arrays.copyOf(data, data.length * 2);
        }
    }

}
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)