Com S 228
Fall 2014
Exam 2

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: ______________________________

ISU NetID (username): ______________________________

Recitation section (please circle one):

1. M 11:00 am (Caleb V and Jesse)
2. M 10:00 am (Bryan and Jacob)
3. M 4:10 pm (Anthony and Ben)
4. T 11:00 am (Nick and Arko)
5. T 10:00 am (Monica and Susan)
6. T 4:10 pm (Alex and Caleb B)

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	15	
2	17	
3	6	
4	34	
5	12	
6	16	
Total	100	

1. (15 pts) On the next page, you will see several snippets of code. Assume the code executes within a class with two instance variables and a method `init()` as shown:

```
public class Exam2
{
   List<String> aList;
   ListIterator<String> iter;
   void init()
   {
      aList = new ArrayList<String>();
      aList.add("A");
      aList.add("B");
      aList.add("C");
      aList.add("D");
   }
  // ...
}
```

For each snippet,

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

(b) draw the state of the list and iterator after the code executes.

However, if the code throws an exception, print any output that occurs before the exception, but ***do not*** draw the list; instead write down the exception that is thrown.

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

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

the list would be drawn as follows.

```
|A B C D
```

(the first one has been done for you as an example).

Code snippet	Output	`List` iterator and state or exception thrown

`init();`
`iter = aList.listIterator();`
	(none)	`|A B C D`

`init();`
`iter = aList.listIterator();`
`iter.next();`
`iter.add("X");`
`iter.next();`
`iter.add("Y");`
`iter.previous();`
`iter.remove();`
`System.out.println(iter.nextIndex());`		

`init();`
`iter = aList.listIterator(aList.size());`
`iter.previous();`
`iter.next();`
`System.out.println(iter.nextIndex());`
`iter.next();`		

`init();`
`iter = aList.listIterator(aList.size());`
`System.out.println(iter.previous());`
`System.out.println(iter.next());`
`System.out.println(iter.previous());`
`iter.remove();`		

`init();`
`iter = aList.listIterator(4);`
`System.out.println(iter.nextIndex());`
`iter.add("X");`
`iter.remove();`		

2. (17 pts) Complete the implementation of the static method `isSorted()` below. The method should return true if the List `l` is in sorted order (non-decreasing) according to the Comparator `comp`, otherwise return false. Throw a `NullPointerException` if `l` is `null`.

```
public static <T> boolean isSorted(List<T> l,
                                   Comparator<? super T> comp)
{
  //TODO








}
```

3. (6 pts) Give the big-O time complexity of the following API operations from the Java Collections framework. If either operation has different amortized and worst case performance characteristics, be sure to list both and clearly denote them.

   (a) (3 pts)

   ```
   LinkedList.add(E e)
   ```

   You may assume that Java's `LinkedList` implementation uses a tail pointer.

   (b) (3 pts)

   ```
   ArrayList.add(E e)
   ```

4. (34 pts) We define a simple, doubly-linked list of `Integer` with the data and methods given in the following code.

```
class IntegerList
{
  private int size;
  private Node head; // Note that this is not a dummy node!
  private Node tail; // Note that this is not a dummy node!

  public IntegerList()
  {
    size = 0;
    head = null;
    tail = null;
  }
  private class Node
  {
    public Integer value;
    public Node prev;
    public Node next;
    public Node(Integer value) { this.value = value; }
  }

  public Integer removeHead()
  {
    /* Assume that this is implemented and works.  Returns *
     * the head.value from the input list (as opposed to   *
     * after the remove operation).                        */
  }

  public static boolean isPrime(Integer n)
  {
    /* Assume that this is implemented and works.  Returns *
     * true if n is prime; otherwise returns false.        */
  }
  // ...
}
```

Using no other methods or instance variables (notice that there are no iterators!), complete the methods specified on the following pages.

(a) (12 pts) This method will insert a new `Node` containing the value `value` at the tail of the list. `null` values are not permitted in the list. Throw a `NullPointerException` on any attempt to insert a `null` value. You may not use any instance variables or methods that are not explicitly defined in the class definition. Note that there are no iterators!

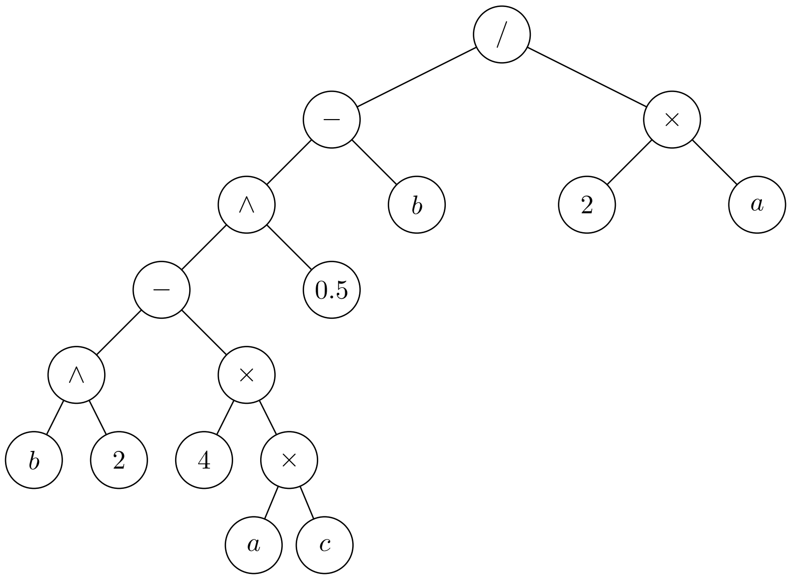
```
public void insertTail(Integer value)
{
  //TODO
















}
```

(b) (22 pts) This method removes all data from `this` (leaving an empty list) and places each value into either `prime` or `composite`, respectively, depending on whether the value is prime or composite. After the method completes, between them `prime` and `composite` will contain *all* of the data originally in `this` and *only* the data originally in `this`, *in the original order*, but split into separate lists of prime and composite values. You may assume that `prime` and `composite` are initialized and empty.

To implement this method, you may assume that your `insertTail()` method works as defined. You will not be penalized here for errors in part (a). Otherwise, you may not use any instance variables or methods that are not explicitly defined in the class definition. Note that there are no iterators!

```
public void primeSplit(IntegerList prime,
                       IntegerList composite)
{
  //TODO








}
```

5. (12 points) Given the following binary expression tree:



(a) (4 points) Give the corresponding prefix expression.

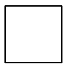
(b) (4 points) Give the corresponding infix expression with parentheses to maintain correct precedence.

(c) (4 points) Give the corresponding postfix expression.

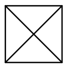
6. (16 pts) In this exercise, you will illustrate the physical representation of lists. In order to ensure uniformity in your list representation, we present the following guidelines that must be followed, but note here that ***this is essentially a semi-formal specification of the way we have been drawing lists in class.***

   Class definition and code snippets follow which build data structures from these definitions. ***Where not explicitly defined, methods have exactly the semantics as those defined in the Java*** `List` ***interface.*** When drawing your diagrams, use notational elements that follow.

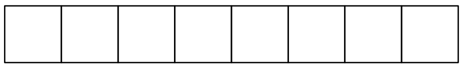
   - A reference or primitive value is represented by a (possibly named) rectangle.

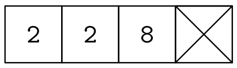
     Represent a value by writing it directly inside or referencing it with an arrow from inside (see the list example below), or show that it is null by marking it with an X:

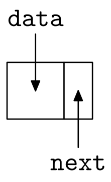
   - An array is represented by a contiguous block of references or primitives. Those in the following example of an array of length 8 do not have values set; they should either have a value or be null:

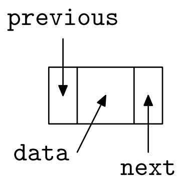
     And this array of length 4 contains the integer values 2, 2, and 8, with a null value in the final slot:

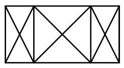
   - A singly-linked list node needs a value and a pointer:

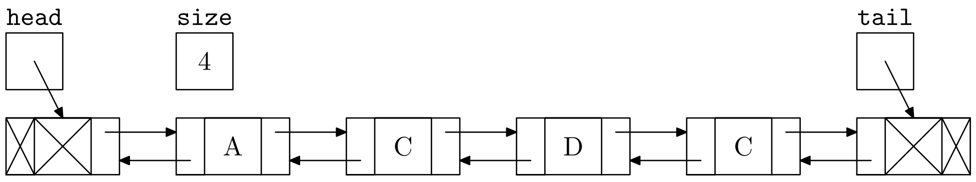
   - A doubly-linked list node needs a second pointer:

     And here's one with every field null:

- For all parts of this exercise, all defined instance variables must be fully described in your diagram. Take the definition of `Exam2DoublyLinkedList`, for example, but add dummy nodes. Here is the answer that you should produce after a call to the constructor followed by four adds, with the values `A`, `C`, `D`, and `C`, respectively.



- Label everything that has a name (like `size`, `head`, and `tail`, above). Anything that does not have a name must be referenced, either directly or indirectly, by something that does. In the example above, the head dummy node is directly referenced by `head` and indirectly referenced by `head.next.previous`.

(a) (8 pts) Exam2SinglyLinkedList

```
public class Exam2SinglyLinkedList<E> {
  private class Node<E> {
    Node next;
    E data;

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

  int size;  // The number of nodes in the list
  Node head; // Reference to the head dummy node
  Node tail; // Reference to the tail dummy node

  public Exam2SinglyLinkedList()
  {
    size = 0;
    head = new Node(null);
    tail = new Node(null);

    head.next = tail;
  }
  // ...
}
```

Draw the data structure(s) that exist after the execution of the following code.

```
Exam2SinglyLinkedList<String> L = new Exam2SinglyLinkedList();
L.add("Isabella");
L.add("Phineas");
L.add("Candace");
L.set(2, "Buford");
L.set(1, "Baljeet");
L.add("Ferb");
L.set(3, "Vanessa");
```

(b) (8 pts) Exam2ArrayList

```
public class Exam2ArrayList<E> {
  private E [] array;      // The array.  Array length doubles every
                           // time an attempt is made to write to a
                           // full array.  It is never reduced.
  private int arrayLength; // The size of the physical array
  private int size;        // The number of elements in the array

  public Exam2ArrayList()
  {
    arrayLength = 1;
    size = 0;
    array = (E[]) new Object[arrayLength]();
  }

  // ...
}
```

Draw the data structure(s) that exist after the execution of the following code.

```
Exam2ArrayList<String> L = new Exam2ArrayList();
L.add("Agent␣P");
L.add("Carl");
L.add("Dr.␣D");
L.add("Major␣Monogram");
L.remove(1);
```

Method Summaries (For Reference)

`List<E>`

`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.

`ListIterator<E>`

`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.

`Collection<E>`

`boolean add(E e)`
: Ensures that this collection contains the specified element. Returns `true` if the element is added. (Optional operation).

`void clear()`
: Removes all of the elements from this list (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 obj)`
: Removes a single instance of the specified element from this collection, if it is present. Returns `true` if the collection is modified.

`int size()`
: Returns the number of elements in this collection.

`Iterator<E>`

`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.

Scratch paper

Scratch paper

Scratch paper

Scratch paper