LAST NAME (PRINT): ____________________ FIRST NAME (PRINT): ____________________

CS 367: Introduction to Data Structures
Midterm Exam 1 – SOLUTION –

Friday, July 14th 2017.

100 points (26% of final grade)

Instructor: Meena Syamkumar

1. **Fill in these fields and their bubbles on the scantron form (use #2 pencil).**
   (a) LAST NAME - fill in your last (family) name starting at left column.
   (b) FIRST NAME - fill in first five letters of your first (given) name.
   (c) IDENTIFICATION NUMBER is your UW Student ID number.
2. **DOUBLE-CHECK THAT YOU HAVE FILLED IN THE BUBBLES FOR EACH COLUMN OF ABOVE FIELDS ON SCANTRON.**
3. **Read, agree to, and sign this ACADEMIC CONDUCT STATEMENT.**

   I will keep my answers covered so that they may not be viewed by another student during the exam or prior to completion of their exam. I will not view or in any way use another's work or any unauthorized devices. I understand that I may not make any type of copy of any portion of this exam. I understand that being caught doing any of these or other actions that permit me or another student to submit work that is not our own will result in automatic failure of the course. All such penalties are reported to the Deans Office for all involved.

Signature:

Parts	Number of Questions	Question Format	Possible Points
I	15	Simple Choice	15
II	15	Multiple Choice	45
III	10	Written	40
	40	Total	100

Turn off and put away all electronic devices and wait for the proctor to signal the start of the exam.

Part I: Simple Choice (1 point each)

Select the word or phrase that makes the statement **true**.

If there is no _______________, then mark **A** if the statement is true or **B** if the statement is false. Be sure to mark the corresponding letter on the answer sheet (scantron). Unless otherwise specified, assume the ADTs, data structures, interfaces, and algorithms mentioned are those discussed in lecture and in the readings.

1. _______________ is a data structure. A. *StackADT* B. *A chain of linked nodes*
2. Given the following *interface* and *class* definition, which code fragment compiles?

   ```
   public interface A { ... }
   public class B implements A { ... }
   ```

   A. *A a = new B();*

   B. *B b = new A();*

   C. *Choices A and B both compile.*
3. Without modifying *BagADT interface*, it _______________ possible to have an indirect access iterator over the items in a *SimpleArrayBag*. Assume *SimpleArrayBag* implements *BagADT*. Recall that *BagADT* is:

   ```
   interface BagADT {
           void add(Object item);
           Object remove();
           boolean isEmpty();
   }
   ```

   A. *is* B. *is not*
4. The *<E>* in a Java *interface* or class header is a type parameter that will be replaced with a known element type. A. *True* B. *False*
5. _______________ exceptions must be caught or the method must have a throws clause declaration. A. *Checked* B. *Unchecked*
6. A *chain of nodes* _______________ insert and remove without shifting the other items. A. *can* B. *cannot*
7. Locking N doors on a building with a *one-time use single master passcode* is _______________ time complexity for the security guard. A. *O(1)* B. *O(N)*
8. Iterating through the data in a *LinkedList* using a *direct access iterator* is _______________ time complexity. A. *linear* B. *quadratic*
9. When using a *singly linked list* to implement a *Stack*, the top of the stack must be kept at _______________ to ensure the most efficient push and pop operations?
   A. *last node* B. *first node*
10. It is better to use a _______________ for a problem that uses the chronological timing of events to ensure that the earliest event will be handled next. A. *Stack* B. *Queue*
11. A *header node* that is added to a linked list implementation of ListADT reduces the Big-O time complexity of the add operation. A. *True* B. *False*
12. _______________ is a *value-oriented* data type. A. *PriorityQueueADT* B. *StackADT*

13. *Re-heapify* restores the ______________ constraint of a heap A. *shape* B. *order*

14. In recursion, each recursive case must make progress towards ______________ case. A. *a base* B. *a recursive*

15. ______________ legal for methodA to call methodB and then for methodB to call methodA. A. ***It is*** B. *It is not*

Part II: Multiple Choice (3 points each) Choose the best answer of the available choices.

16. Which choice is a valid replacement for **BODY** in **DoorADT** ?

```
public interface DoorADT {
        BODY
}
```

A.
```
public class Door implements DoorADT {
        private boolean doorIsOpen = false;
        public void open() { doorIsOpen = true; }
        ...
}
```

B. `void open() { this.doorIsOpen = true; }`

C. `void open() { ... }`

D. `void open();`

E. None of above.

17. Which of the following methods must be defined by a class that **implements Iterator<E>** ?

i `public Iterator<E> iterator() { ... }`

ii `public boolean hasNext() { ... }`

iii `public E next() { ... }`

A. i only
B. i and ii only
C. ii and iii only
D. i, ii, and iii
E. None of the above

18. Which choice correctly gets an **Iterator** from an instance of *EmployeeList* named *employeeList*? Assume that *EmployeeList* is *Iterable*.

   A. Iterator<EmployeeList> itr = employeeList.hasNext();

   B. Iterator<EmployeeList> itr = employeeList.iterator();

   C. Iterator<EmployeeList> itr = employeeList.getIterator();

   D. Iterator<EmployeeList> itr = new EmployeeList(employeeList);

   E. Iterator<EmployeeList> itr = new EmployeeList(employeeList.get(0));

19. What is the **result** of executing this program? Assume all named exceptions are **unchecked** exceptions.

```java
public static void main(String[] args) {
        try {
                methodA();
                System.out.print("main,");
                methodB();
        } catch ( YellowException e ) {
                System.out.print("yellow,");
        } catch ( Exception e ) {
                System.out.print("caught,");
        } finally {
                System.out.print("main-finally,");
        }
}
private static void methodA() {
        try {
                System.out.print("methodA,");
        } catch (RedException e) {
                System.out.print("caught red,");
        } finally {
                System.out.print("A-finally,");
        }
}
private static void methodB() {
        try {
                System.out.print("methodB,");
                throw new YellowException();
        } catch ( GreenException e ) {
                System.out.print("caught green,");
        }
}
```

   A. methodA,A-finally,main,methodB,main-finally,

   B. methodA,A-finally,main,methodB,yellow,main-finally,

   C. methodA,caught red,A-finally,main,methodB,main-finally,

   D. methodA,A-finally,main,methodB,caught green,main-finally,

   E. program crashes

20. Which choice correctly links **newnode** into a **doubly-linked** chain of nodes? Assume *newnode* is being linked between two non-null doubly-linked nodes and that *newnode* must be inserted after *curr*.

A.
```
newnode.setPrev(curr);
newnode.setNext(curr.getNext());
newnode.getNext().setPrev(newnode);
curr.setNext(newnode);
```

B.
```
curr.setNext(newnode);
newnode.setPrev(curr.getPrev());
newnode.setNext(curr.getNext());
newnode.getNext().setPrev(newnode);
```

C.
```
curr.setNext(newnode);
newnode.setPrev(curr);
newnode.setNext(curr.getNext());
newnode.getNext().setPrev(curr);
```

21. What is the **time complexity** of the following code? Assume: *list* is implemented as a **circular doubly-linked chain** of nodes with a head reference to a header node. Recall that *mod* operation happens in constant time.

```
for ( int i = 0 ; i < N ; i++ )
        for ( int j = i ; j > 0; j-- )
                if ( (i + j) % 2 == 0 )
                        list.add(i + j);
```

A. O(N) B. $O(N^2)$ C. $O(N^3)$

22. Given that: $T(N)$ is $O(F(N))$ if $T(N) <= C * F(N)$ for $N >= n_0$.
Which values for C and n_0 can be used to show that $T(N) = 4N^3 + 25N + 10$ is $O(N^3)$?

A. $C = 4, n_0 = 2$
B. $C = 4, n_0 = 5$
C. $C = 5, n_0 = 10$
D. $C = 10, n_0 = 2$
E. None of the above can be used to show that $T(N) = 4N^3 + 25N + 10$ is $O(N^3)$

23. What is the **output** of this code fragment?

```
QueueADT queue = new LinkedQueue();
StackADT stack = new ArrayStack();
for ( int i = 0 ; i < 5 ; i += 2 ) {
        queue.enqueue(i);
        stack.push(i + 1);
}
for ( int i = 0; i < 3; i++ ) {
        System.out.print(stack.pop() + ",");
        System.out.print(queue.dequeue() + ",");
}
System.out.println();
```

A. 4,1,2,3,0,5,

B. 5,0,3,2,1,4,

C. 0,2,4,1,3,5,

D. 5,4,3,2,1,0,

E. 0,5,2,3,4,1,

24. Suppose myList is a **List ADT** that initially contains Strings in this order:
["A", "B", "C", "D", "E", "F"]

Consider the following code fragment:

```
for (int i = 0; i < myList.size(); i++)
        myList.add(myList.remove(i));
```

Which one below correctly shows the resulting contents of myList after the code fragment executes?

A. ["A", "B", "C", "D", "E", "F"]

B. ["B", "D", "F", "A", "C", "E"]

C. ["B", "D", "F", "C", "A", "E"]

D. ["B", "D", "F", "C", "E", "A"]

E. ["F", "E", "D", "C", "B", "A"]

25. What is the **Big-O** time complexity of the **removeMax** operation of a **PriorityQueue** implemented using a **sorted array**? Assume: The number of items is stored and the values are in the array in **ascending order**.

A. $O(N log_2 N)$ B. $O(N^2)$ C. $O(N)$ D. $O(log_2 N)$ E. $O(1)$

26. Which expression computes the index of the **right child** of the node storing the current data item in an **array-based heap**? Assume the heap's root node is at index 1.

A. $parent * 2$

B. $parent * 2 + 1$

C. $parent * 2 - 1$

D. $parent/2$

E. $(parent - 1)/2$

27. Which **CODE completion** reverses the characters in a char array?

```
static void reverse(char [] a) { reverse(0,a); }
static void reverse(int i, char[]a) {
        if ( i < a.length/2 ) {
                CODE ;
                char t = a[i];
                a[i] = a[a.length-1-i];
                a[a.length-1-i] = t;
        }
}
```

A. reverse(i-1, a)

[✓ B. reverse(i+1, a)]

C. reverse(i, a)

28. Given that you have implemented a **Queue** with a **circular array** and that the Strings **A,B,C,D** have been enqueued in that order (See figure).
Note: A circular array is one where index access wraps around the end of the array.

A	B	C	D	

What is the contents of the array after these operations:
dequeue(), enqueue("E"), dequeue(), enqueue("F"), dequeue(), enqueue("G"), enqueue("H")

A.

F	G	C	D	E

B.

D	E	F	G	H

C.

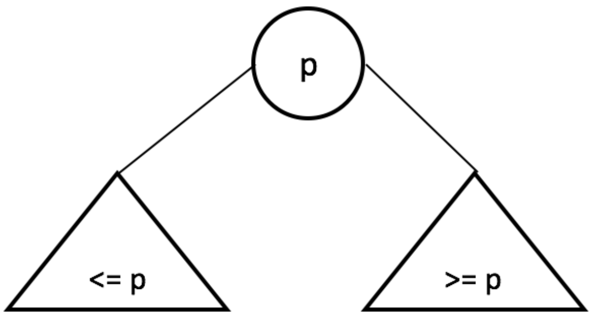
E	E	G	D	H

[✓ D.]

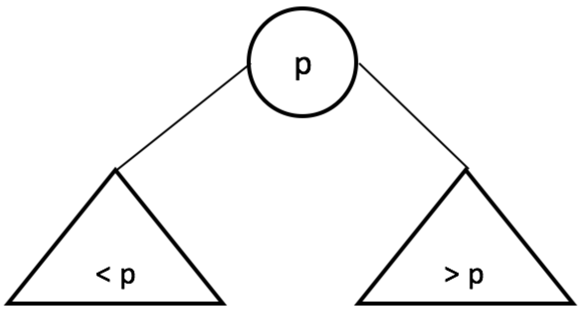
F	G	H	D	E

29. Which diagram shows the **ordering constraint** for a **max heap**?

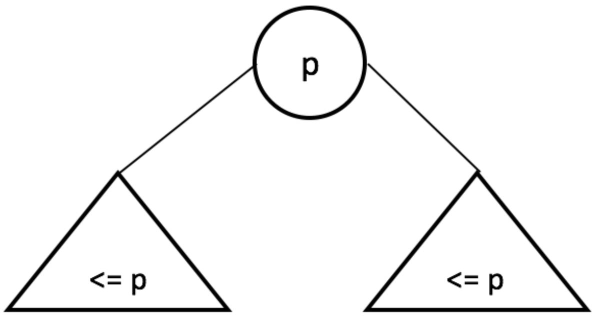
A.



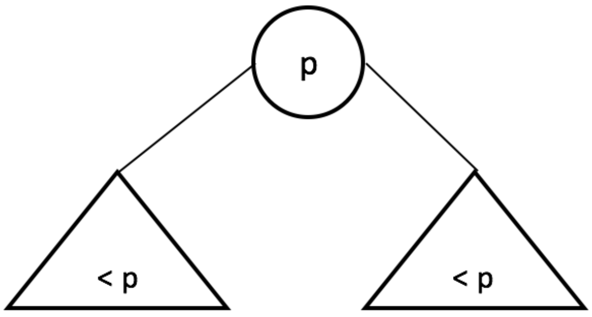
B.



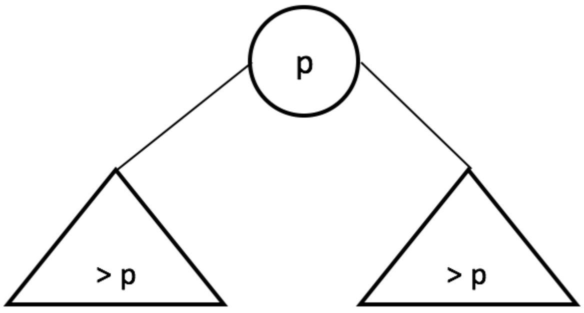
[✓ C.]



D.



E.



30. Suppose you have **N** ID cards, which have been printed in a consecutive numeric order. What is the **Big-O** time complexity for N people to **efficiently** search through and find their own ID card?

    A. $O(N)$
    B. $O(N^2)$
    C. $O(Nlog_2N)$
    D. $O(log_2N)$
    E. $O(1)$

Part III: Written (4 points each)
Write your answer within the provided space as briefly as possible.

31. Suppose list1 and list2 are two **ArrayList** that contain arbitrary number of items. Describe briefly what the following code does:

```
while (!list1.isEmpty()) {
        list2.add(list2.size(), list1.remove(0));
}
```

Moves items from list1 in order into list2 or otherwise, merges list1 into list2, while removing items from list1 in order.

32. Consider the incomplete **StringIterator** class below that represents an indirect access iterator used to step through the characters in a string. For this question, assume Java's Iterator interface is non-generic as shown below. Recall the **charAt(int index)** function in **String** class.

```
public class StringIterator implements Iterator {

        private String str = null;
        private int count = 0;

        public StringIterator( ... ) { ... }
        public boolean hasNext() { ... }
        public Character next() { ... }
        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

Complete the above class in the 4 areas indicated with "..." so that the methods behave as specified.

```
public class StringIterator implements Iterator {

        private String str = null;
        private int count = 0;

        public StringIterator( String str ) {
                this.str = str;
        }

        public boolean hasNext() {
                if(count < str.length())
                        return true;
                else
                        return false;
        }

        public Character next() {
                Character curr = str.charAt(count);
                count++;
                return curr;
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

33. Consider **singly linked list** and **Listnode** representation for one node of the singly linked list. Complete the following **reverse** function which iteratively reverses the linked list and returns the updated **head** reference.

    Note: You may use **only** methods in the *Listnode* class to complete the method.

```
Listnode<E> reverse(Listnode<E> head) {
        Listnode<E> prev = null;
        Listnode<E> curr = head;
        Listnode<E> next = null;

        while (curr != null) {
            next = curr.getNext();
            curr.setNext() = prev;
            prev = curr;
            current = next;
        }
        head = prev;

        return head;
    }
```

34. Consider the **doubly linked list** implementation which has a **tail** reference apart from the **head** reference. Give the worst-case complexity (**Big-O notation**) for each of the following **ListADT** functions, along with **one or two words reasoning** for the *non-constant* complexity:

    (a) constructor
    $O(1)$, because of initialization

    (b) add(E) // add at end
    $O(1)$, because of having access to last node with the tail reference

    (c) add(int, E) // add at pos
    $O(N)$, due to traversal

    (d) contains(E)
    $O(N)$, because of linear search

    (e) size()
    $O(1)$, because of using *numItems*

    (f) isEmpty()
    $O(1)$, because of checking if *numItems* is 0

    (g) get(int)
    $O(N)$, due to traversal

    (h) remove(int)
    $O(N)$, due to traversal

35. Assume you are comparing three different algorithms to solve some problem and have determined the worst-case time equations for each:

    (a) Algorithm 1: $T(N) = Nlog_2N + 5 + N^3$
    (b) Algorithm 2: $T(N) = 4N + 2N^2 + 17$
    (c) Algorithm 3: $T(N) = 22Nlog_2N + 147$

    A. What is the **worst-case** time complexity for each algorithm?
        (a) Algorithm 1: $O(N^3)$
        (b) Algorithm 2: $O(N^2)$
        (c) Algorithm 3: $O(Nlog_2N)$

    B. Which is the algorithm that has the **lowest worst-case** time complexity?
    Algorithm 3

    C. Should the algorithm with the lowest time complexity always be used? Briefly explain why/why not?
    No, many considerations go into choosing the most appropriate algorithm. For example, memory usage and development time are not captured by Big-O notation, but can be equally (or sometimes even more) important as running time.

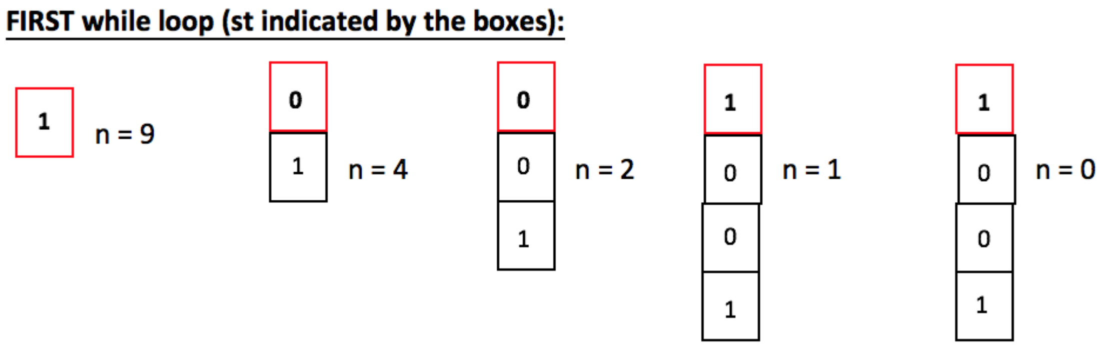
    Furthermore, Big-O notation hides large constants. In this example, Algorithm 3 has a very large constant term; if we know that our problem instances will be small (say N = 1), we might prefer Algorithm 1 or 2.

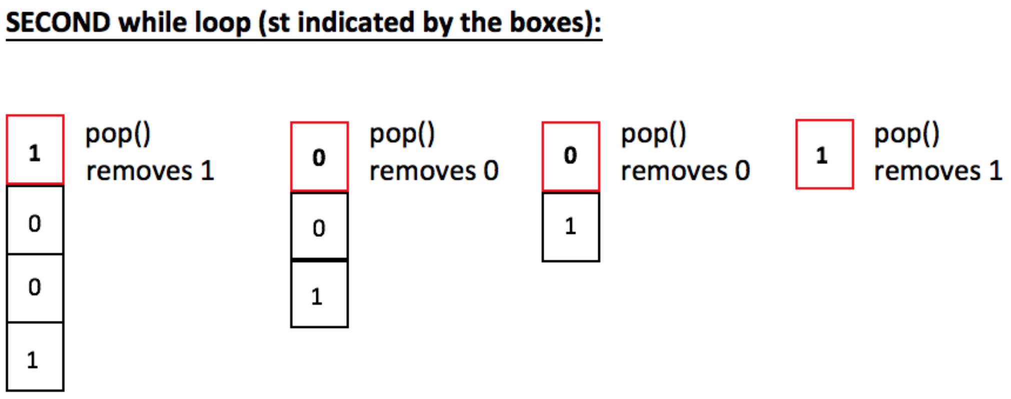
36. Read the below **secret** method thoroughly.

```
void secret(int n, Stack<Integer> st) {
        while (n > 0)
        {
                st.push(n % 2);
                n = n / 2;
        }

        while (!st.isEmpty())
                System.out.print(st.pop());
}
```

(a) **Trace** the code for this call: **secret(9, new Stack<Integer>())**.
In each iteration, show the value of **n** and the corresponding **st**; finally show the output of the function.



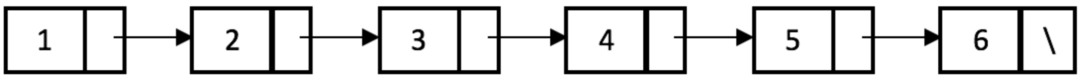


(b) What does the **secret** function do? **Hint**: Try generating the output for the call with a different value other than 9.
The **secret** function converts the given **decimal** number to its **binary** representation.
Exercise: Try tracing the method for few other values like 7, 8.

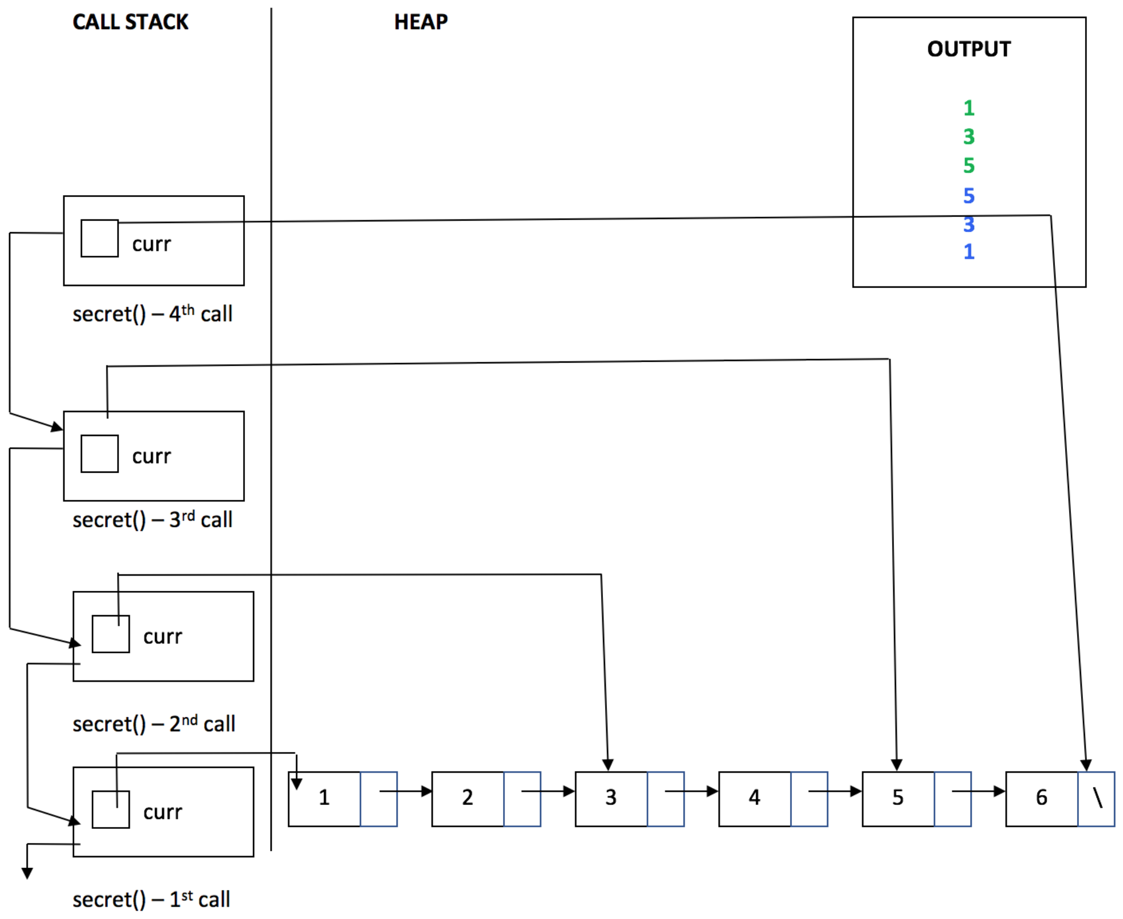
37. Read the below *recursive* **secret** method thoroughly. Show the **Call Stack**, **Heap** tracing and **Output** for the **secret** method given the following linked list.

```
void secret(Listnode<Integer> curr) {
        if(curr == null)
                return;
        System.out.println(curr.getData());

        if(curr.getNext() != null)
                secret(curr.getNext().getNext());
        System.out.println(curr.getData());
}
```



The output shown below in green color is the output from the first print statement. Observe the presence of the second print statement after the recursive call. The output of the second print statement is shown in blue color.



38. Given that below are the major steps of the **removeMax** operation on **Max Heap**, complete the code for the **removeMax** function. Assume an **array** implementation for Max Heap with root at index 1. The private *instance variables* are also mentioned below.

(a) Remove and return root

(b) Replace root with last item

(c) Re-heapify

```
private Comparable[] items;
private int numItems;

Comparable removeMax() throws EmptyPriorityQueueException {
        if (numItems <= 0) throw new
           EmptyPriorityQueueException();

        Comparable max = items[1]; // remove root and save for
           returning
        items[1] = items[numItems]; // replace root with last
           item

        numItems--; // decrement number of items

        //Re-heapify
        boolean done = false;
        int parent = 1;
        int leftChild, rightChild;
        int biggerChild;
        while(!done) {
                //Find children indices
                leftChild = 2 * parent;
                rightChild = (2 * parent) + 1;

                //First check if the children exist
                if(leftChild > numItems) {
                        //Left child itself doesn't exist
                        done = true;
                }
                else if(rightChild > numItems) {
                        //Right child alone doesn't exist
                        // Check with left child
                        if
                           (items[parent].compareTo(items[leftChild])
                           < 0) {
                                swap(items, parent, leftChild);
                                done = true;
                        }
                        else
                                done = true;
                }
                else {
                        //Find largest child
                        if
                           (items[leftChild].compareTo(items[rightChild]
                           <= 0)
                                biggerChild = rightChild;
                        else
                                biggerChild = leftChild;

                        //Swap with the largest child
                        if
                           (items[parent].compareTo(items[biggerChild])
                           < 0) {
                                swap(items, parent,
                                   biggerChild);
                                parent = biggerChild;
                        }
                        else
                                done = true;
                }
        }
        return max;
}

private void swap(items, sourceIndex, destIndex) {
        Comparable temp = items[sourceIndex];
        items[sourceIndex] = items[destIndex];
        items[destIndex] = temp;
}
```

39. Complete the following implementation of **printReverse** function that uses **recursion** to print the items in a **singly linked list** in the **reverse** order. You may assume that the function will be called from **main** with argument as the **head** reference to the linked list.

```
void printReverse(Listnode<Integer> curr) {
        if (curr == null) return;
        printReverse(curr.getNext());
        System.out.println(curr.getData());
}
```

40. Given the following **recursive** function **mult3**, work out the time complexity of the algorithm in detail. You may assume $n >= 1$.

```
int mult3(int n) {
        if n == 1
                return 3
        else
                return mult3(n-1) + 3
}
```

(a) Equations
Base case: $T(1) = 1$
Recursive case: $T(N) = 1 + T(N-1)$
Note: Although you add 3 here, you are still performing a constant O(1) addition operation. So the growth rate function for base case is O(1).

(b) Table

N	$T(N) = 1 + T(N-1)$
1	$T(1) = 1$
2	$T(2) = 1 + T(1) = 1 + 1 = 2$
3	$T(2) = 1 + T(2) = 1 + 2 = 3$
...	...
k	$T(k) = k$ GUESS

(c) Verify
$T(N) = 1 + T(N-1)$
$N = 1 + (N-1) = N$ CORRECT

(d) Complexity
O(N)