MIDTERM II REVIEW SOLUTIONS

1.1

Assuming the ArrayList implementation in lab 8, following is the binary search that finds whether an element is present in a sorted list:

```
    boolean binarySearch(T element) {
        int low = 0;
        int high = numberOfEntries;

        while (low <= high) {
            int mid = low + (high - low)/2;
            if (list[mid].equals(element))
                return true;
            else if (list[mid].compareTo(element) < 0)
                low = mid + 1;
            else
                high = mid - 1;
        }

        return false;
    }
```

Analysis of Time Complexity:
The question to answer is how many times do we divide the list in half until
we have only one element in the list. This can be mathematically represented as : $1 = N/2^x$ where N is the size of the list and x is the number of times the list is divided.

On multiplying the above equation by 2^x we get:

$$2^x = N$$

Now take a log_2 on both sides:

$$log_2 2^x = log_2 N$$
$$x log_2 2 = log_2 N$$
$$x = log_2 N$$

This means that you can divide the list $log_2 N$ times to tell whether an element exists in a list.

1.2

```
public void intersect(AList<T> other) {
    int currListLen = numberOfEntries;
    int otherListLen = other.numberOfEntries;
    T[] intersectedList = (T[]) new Comparable[Math.min(currListLen,otherListLen)];
    int intersectIndex = 0;

    for (int i = 0; i < currListLen; i++) {
        for (int j = 0; j < otherListLen; j++) {
            if (this.get(i).compareTo(other.get(j)) == 0)
                intersectedList[intersectIndex++] = other.get(j);
        }
    }

    list = intersectedList;
```

```
    this.numberOfEntries = intersectIndex;
}
```

2. Linked Lists

2. 1

```
public void removeEvery(int n) {
    if (n == 0 || n > size)
        return;

    //Every node to be removed
    if (n == 1) {
        first = last = null;
    }

    Node<T> currentNode = first;
    Node<T> prevNode = null;
    int count = 0;

    while (currentNode != null) {
        count++;

        if (count % n == 0) {
            prevNode.setNext(currentNode.getNext());
        }

        prevNode = currentNode;
        currentNode = currentNode.getNext();
    }
}
```

2.2

```
public List<LinkedList<T>> extractGroupsOf(int n) {
    if (n == 0 || n > size) {
        List<LinkedList<T>> lst = new java.util.LinkedList<LinkedList<T>>();
        return lst;
    }

    Node<T> currentNode = first;
    List<LinkedList<T>> extractedList = new java.util.LinkedList<LinkedList<T>>();
    int count = 0;

    while (currentNode != null) {
        count = 1;
        LinkedList<T> groupedList = new LinkedList<>();

        //Make individual linked lists
        while (currentNode != null && count <= n) {
            groupedList.add((currentNode.getData()));
            count++;
            currentNode = currentNode.getNext();
        }

        //Adding the individual linkedlist to the extracted groups
        extractedList.add(groupedList);
    }

    return extractedList;
}
```

3. **Queues**

```
class MyQueue<T extends Comparable<T>> {

    Stack<T> tempstack = new Stack<T>();
    Stack<T> dequeuestack = new Stack<T>();

    // Adds an element to the end of the queue
    public void enqueue(T x) {
        if(dequeuestack.isEmpty()){
            dequeuestack.push(x);
        }else{
            while(!dequeuestack.isEmpty()){
                tempstack.push(dequeuestack.pop());
            }

            dequeuestack.push(x);

            while(!tempstack.isEmpty()){
                dequeuestack.push(tempstack.pop());
            }
        }
    }

    // Removes the element at the front of queue.
    public void dequeue() {
        if (dequeuestack.isEmpty()) {
            throw new NoSuchElementException();
        }

        dequeuestack.pop();
    }

    // Get the front element.
    public T peek() {
        if (dequeuestack.isEmpty()) {
            throw new NoSuchElementException();
        }

        return dequeuestack.peek();
    }

    // Return whether the queue is empty.
    public boolean isEmpty() {
        return dequeuestack.isEmpty();
    }
}
```

3.2 Biased Queue

We will use a LinkedList to implement this BiasedQueue datastructure. (Note: Using doubly linked list to implement this Biased Queue would result in an efficient solution in terms of time complexity)
The Node class for the LinkedList is as follows:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;
```

```
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public Node(T data) {
        this(data, null);
    }

    public Node() {
        this(null, null);
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

    public Node<T> getNext() {
        return next;
    }

    public void setNext(Node<T> next) {
        this.next = next;
    }
}
```

The BiasedQueue class using a LinkedList is as follows:

```
public class BiasedQueue {
    private Node<String> first, last;
    private int size;

    public BiasedQueue() {
        first = null;
        last = null;
        size = 0;
    }

    public void enqueue(String person) {
        if(person == null)
            return;

        Node<String> newNode = new Node<>(person);

        if(size == 0) {
            first = newNode;
            last = newNode;
        }
        else if (person.equals("Budger")) {
            Node<String> currNode = first;
            Node<String> prevNode = null;

            //Iterate till the penultimate element
            while(currNode.getNext() != null) {
                prevNode = currNode;
                currNode = currNode.getNext();
            }

            //Let the budger budge his way in
            if (prevNode == null) {
                first = newNode;
```

```
                newNode.setNext(currNode);
            }
            else {
                prevNode.setNext(newNode);
                newNode.setNext(currNode);
            }
        }
        else {//New person is not a Budger
            last.setNext(newNode);
            last = newNode;
        }

        size++;
    }

    public void dequeue() {

        if (first == null)
            throw new NoSuchElementException();

        Node<String> firstElem = first;
        Node<String> prevElem = null;

        //Iterate till end of the list or till a non-Budger is found
        while (firstElem != null && firstElem.getData().equals("Budger")) {
            prevElem = firstElem;
            firstElem = firstElem.getNext();
        }

        //Only Budgers were found in the list, so remove the first Budger
        if (firstElem == null) {
            first = first.getNext();
        }
        else {//Non-Budger found
            if (prevElem == null) {
                first = first.getNext();
            }
            else {//Some Budgers were found before the Non-Budgers
                prevElem.setNext(firstElem.getNext());
            }
        }

        size--;
    }
}
```

4. **Stacks**

```
public class Postfix {
    public double evaluate(String[] expression) {
        Stack<Double> stack = new Stack<Double>();

        for (int i = 0; i < expression.length; i++) {
            switch (expression[i]) {
                case "+":
                    stack.push(stack.pop() + stack.pop());
                    break;

                case "-":
                    stack.push(-stack.pop() + stack.pop());
                    break;

                case "*":
                    stack.push(stack.pop() * stack.pop());
                    break;

                case "/":
                    Double n1 = stack.pop(), n2 = stack.pop();
                    stack.push(n2 / n1);
                    break;

                default:
                    stack.push(Double.parseDouble(expression[i]));
            }
        }

        return stack.pop();
    }
}
```

5. **Dictionaries**

```
public class Anagram {
    public boolean isAnagram(String first, String second) {
        Map<Character, Integer> charCount = new HashMap<>();

        //Unequal length strings can't be anagrams
        if (first.length() != second.length())
            return false;

        //Storing the count of each character of the first string in a HashMap
        for (int i = 0; i < first.length(); i++) {
            if (charCount.containsKey(first.charAt(i))) {
                charCount.put(first.charAt(i), charCount.get(first.charAt(i)) + 1);
            }
            else {
                charCount.put(first.charAt(i), 1);
            }

        }

        //Iterating the second string to verify if its an anagram
        for (int i = 0; i < second.length(); i++) {
            // For every character encountered in second string,
            // reduce its count if it is present in the first string
            if (charCount.containsKey(second.charAt(i)) &&
                        (charCount.get(second.charAt(i)) > 0)) {
                charCount.put(second.charAt(i), charCount.get(second.charAt(i)) -
1);
            }
            else {//Character of second string not found in first string or
                // the count of the corresponding character is zero
                return false;
            }
        }
        return true;
    }
}
```

6. **Complexity Analysis**

6. 1

Runtime Complexity	Best-Case	Average-Case	Worst-Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

6.2

The complexity calculation can be broken down as follows:
Assume the length of array is ‘n’
The outer for loop executes $(n - k + 1)$ times
The inner for loop executes $(i + k) - (i + 1)$ times
Both combined it becomes, $(n - k + 1)(k - 1) = nk - n - k^2 + 2k - 1$
Writing in Big O we get $O(nk)$ as the worst case complexity of the algorithm.

6.3

The complexity of for add can be broken down as follows:
Let the size(or number of elements) of list be denoted by 'n'
In case the list is not full, it only takes a constant time to add to this list
In case the list is full, it takes n operations to add an element to this list because the old list elements have to be recopied into the larger array. Since this is the worst case for an add operation, the Big O complexity is $O(n)$.

6.4

The ArrayList is implemented as dynamically re-sizing array whereas the LinkedList is implemented as a singly linked list.

The LinkedList allows for constant time insertion or removal of elements using iterators. However, it only provides a sequential access to the elements i.e the list can either be traversed in forward or backward direction. In case the list is going to large, the LinkedList occupies more memory compared to the ArrayList due to every node containing both data element and reference to next node.

The ArrayList on the other hand, provides fast random read access performance. However, adding or removing elements from anywhere but the end requires the shifting of all subsequent elements to either make an opening or fill a gap.

So in general if you are sure there are going to be a lot of inserts and deletions to the list then LinkedList is a better choice. If superior read performance is desired, then ArrayList performs better due to superior random access capability.