Q1

This method will return the given singly linked list in the reverse order. It acts by rearranging pointers. Head and tail pointers are correctly set.

```
/* Java/pseudocode to reverse the elements of a singly-linked list */
reverseSLL(list)
   prevNode = null
   currNode = list.head
   nextNode = null

   while(currNode != null) // list traversal
      nextNode = currNode.next
      currNode.next = prevNode
      prevNode = currNode
      currNode = nextNode

   list.tail = list.head
   list.head = prevNode

   return list
```

Q2

For both unordered lists and arrays, one must iterate through each element in the list/array until the value is found. It is assumed that the order of the unordered list/array is completely random, in which case the average number of nodes accessed in a search will be n/2.

For an ordered array (random access), a binary search can be employed to search for a particular node. The time complexity of binary search is O(log2 n) or simply O(log n). The best case is (1); however, the average performance of binary search is (log n). Thus, the average number of nodes accessed will be log n.

For an ordered list, average number of nodes accessed would depend on the implementation namely whether list is random or sequential access. From the Java API on binarySearch (https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html)

> This method runs in log(n) time for a "random access" list (which provides near-constant-time positional access). If the specified list does not implement the RandomAccess interface and is large, this method will do an iterator-based binary search that performs O(n) link traversals and O(log n) element comparisons.

Thus, the average number of nodes for a sequential access list will be n/2, and for a random access list log n.

Q3

This question is deceptively tricky because there are several different cases to evaluate.

```
/* Java/pseudocode to interchange the mth and nth elements of a singly-linked list */
swap(list,m,n)
   curr = list.head
   prevM = null
   currM = null
   nextM = null
   prevN = null
   currN = null
   nextN = null

   if(curr == M) // case head --> m
      currM = curr
      nextM = curr.next

   if(curr == N) // case head --> n
      currN = curr
      nextN = curr.next

   while(curr != null) // list traversal
      if(curr.next == M)
         prevM = curr
         currM = curr.next
         nextM = curr.next.next
      if(curr.next == N)
         prevN = curr
         currN = curr.next
         nextN = curr.next.next
      curr = curr.next

   if(prevM == null) // case head --> m
      list.head = currN
      currN.next = nextM

      prevN.next = currM
      currM.next = nextN

   else if(prevM == null) // case head --> n
      list.head = currM
      currM.next = nextN

      prevM.next = currN
      currN.next = nextM

   else
      prevM.next = currN // m-1 --> n
      currN.next = nextM // n --> m+1

      prevN.next = currM // n-1 --> m
      currM.next = nextN // m --> n+1
```

Q4

```
/* Java/pseudocode to implement the deque as a doubly-linked list (not circular, no
    header) with InsertLeft and DeleteRight methods */
public class Deque
   private Node head = null // left
   private Node tail = null // right

   private class Node
      Node left
      Node right
      Item data

      Node(Item data, Node left, Node right)
         this.data = data
         this.left = left
         this.right = right

   boolean isEmpty()
      if(head == null AND tail == null)
         return true
      else
         return false

   void insertLeft(Item data)
      Node node = new Node(data, null, head) // data, left (head), right (tail)
      head = node

   Item deleteRight()
      if(isEmpty())
         throw exception
      else
         Node temp = tail
         tail.left.right = tail.right // set node n-1 next to null
         tail = tail.left // set tail to node n-1
         return temp
```

Q5

```
/* Java/pseudocode to implement the deque as a doubly-linked circular list with a header
    and InsertRight and DeleteLeft methods. */
public class Deque

   private class Node
      Node left // head
      Node right // tail
      Item data

      Node(Item data, Node left, Node right)
         this.data = data
         this.left = left // head
         this.right = right // tail

   Node header = new Node(null,null,null)
```

```
boolean isEmpty()
    if(head == null AND tail == null)
        return true
    else
        return false

void insertRight(Item data)
    if(isEmpty())
        Node node = new Node(data, node, node) // points to itself
    else
        Node node = new Node(data, header.left, header.right)
        header.right.right = node
        node.right = header.left
        header.right = node

Item deleteRight()
    if(isEmpty())
        throw exception
    else
        Node temp = header.left // temp = first (head) node
        header.right.right = header.left.right // set trail node next to header.next
        header.left = header.left.right // update where header head points to
        return temp
```

Q6

My answer to this question does require additional working out, but these are the basics.

```
/* Java/pseudocode for hybrid implementation of list capable of handling multiple
    stacks/queues */
public class HybridArray{

    private int size;
    Node[] data = new Node[size];
    Stack free = new Stack(size);

    public HybridArray(int size){
        this.size = size;
        for(int i = 0; i < size; i++){
            free.push(i);
        }
    }

    // Implement stack as an array
    private class Stack{
        private int arr[];
        private int size;
        private int top = 0;

        private Stack(int size){
            this.size = size;
```

```
            arr = new int[size];
        }

        private void push(int element){
            if(top == size) {
                System.out.println("Invalid.");
            }
            else {
                arr[top] = element;
                top++;
            }
        }

        private int pop(){
            if (isEmpty()){
                System.out.println("Invalid.");
            }
            return arr[top-1];
        }

        private boolean isEmpty(){
            if (top == 0) {
                return true;
            }
            return false;
        }
    }

    // Implement node to hold data and pointer
    private class Node{
        int data;
        int next;

        private Node(int data, int next){
            this.data = data;
            this.next = next;
        }
    }


    // Methods to push and pop to/from the free index stack
    public int freeIndex(){
        if(free.isEmpty()){
            System.out.println("Invalid.");
        }
        return free.pop();
    }

    public void returnFree(int index){
        free.push(index);
    }

    // Stack methods
    public void push(int data, int next){
```

```
        int freeIndex = free.pop(); // obtain an index from the free index stack
        Node node = new Node(data, 0)
        data[freeIndex] = node;
    }

    public Node pop(){
        if(data.isEmpty()){
            throw exception;
        }
        node = startNode
        while(node.next != 0){ // traverse array to find last node in stack
            prevNode = node;
            node = data[node.next];
        }
        data[prevNode].next = 0; // set pointer of new last item to 0
        returnFree(node); // return the index to the free index stack
        return data[node];
    }

    // Queue methods
    public void add(int data, int next){
        int freeIndex = free.pop(); // obtain an index from the free index stack
        Node node = new Node(data, 0)
        data[freeIndex] = node;
    }

    public void remove(){
        temp = startNode;
        startNode = startNode.next; // set new startNode to the next element in queue
        returnFree(temp); // return the index to the free index stack
        return data[temp];
    }

}
```