Assignment 4 - Queues and Lists

Q1:

ADT Priority Queue: A group of data that stores items ordered by some priority setting
Constructor

- PQueue:
  Input: None
  Precondition: None
  Process: Initialize an empty priority queue
  Postcondition: None
  Output: Return a new empty priority queue

Checker:

- PQEmpty
  Input: None
  Precondition: None
  Process: Check if the queues contain any data items.
  Postcondition: None
  Output: Return 1 or true if the queue is empty and 0 or false otherwise.

Manipulator:

- PQEnqueue:
  Input: A new data item to insert
  Precondition: None
  Process: Store the new item following the priority setting to the group
  Postcondition: The PQ contains additional item
  Output: None

- PQDequeue:
  Input: None
  Precondition: PQ is not empty
  Process: Remove and return the highest rank with the priority setting from the PQ
  Postcondition: The PQ contains one less item
  Output: Return the removed data

- PQPeek:
  Input: None
  Precondition: PQ is not empty
  Process: Retrieve the highest rank with the priority setting from the PQ
  Postcondition: The PQ doesn't change
  Output: Return the retrieved data

end ADT Priority Queue

Q2:

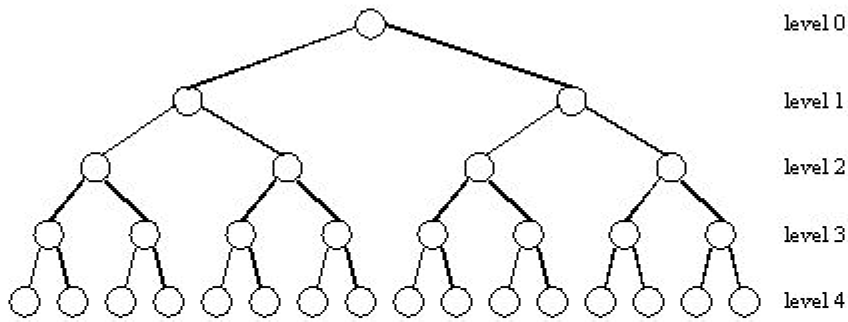
```java
// assume the singly linked list is defined with Node class
public class Node{
   datatype data;
   Node next;
}

// reverse(): reverse the direction of a singly linked list
public static Node reverse(Node head){
   if (head.next == null) return head;

   // call method recursively to get the tail of the list
   Node last = reverse(head.next);

   // setting the next node's NEXT pointer connected the self
   // and setting its next pointer as null
   // ===> reversing the direction of the pointers.
   head.next.next = head;
   head.next = null;

   // return the last node as the new header reference of the linked list
   return last;
}
```

Q3:
Unordered List: Since searching through a linked list is sequential, the minimum number of visited nodes is 1, and the maximum is N: the length of the list. so the cost would be O(N) and the average would be N/2

Ordered List: Same as an unordered list, the ordered linked list requires sequential search, which is linear. Hence the time complexity is O(N), and the average number of the visited nodes is N/2

Unordered Array: Even though an array implementation allows random access, we can't do binary search for an unordered array. So the worst case would be visiting every element, and the best case would be getting the target element at the first check. The average will be N/2

Ordered array: We can use binary search in an ordered array. For calculating average nodes visited, we can count the level of the tree from the top to the bottom.
Let's say there are 2^n - 1 item, the binary tree would be full.



For searching a element of level 0, the probability is 1/N and only 1 node visited for it.
For level 1, the probability is 2/N, and 2 read for searching them.
For level 2, 3 reads with prob 4/N
…
For the last level, the probability would be 2^(n-1) / N, and the cost would be log2(N)

So the average number can be calculated as below

$$\sum_{i=1}^{\log2(N)} \frac{2^{i-1}}{N} \cdot i = \frac{1}{N} \sum_{i=1}^{\log2(N)} 2^{i-1} \cdot i$$

$Since\ i \le \log2(N):$

$$\frac{1}{N} \sum_{i=1}^{\log2(N)} 2^{i-1} \cdot i \le \frac{1}{N} \sum_{i=1}^{\log2(N)} 2^{i-1} \cdot \log2(N) = \frac{\log2(N)}{N} \sum_{i=1}^{\log2(N)} 2^{i-1}$$

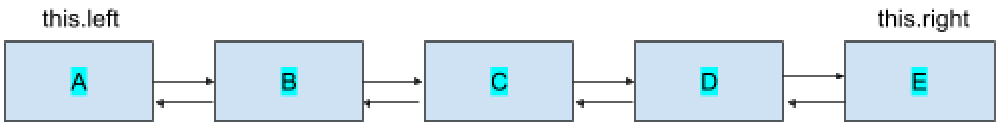
$$= \frac{\log2(N)}{N} \cdot \left(2^{\log2(N)} - 1\right) = \frac{\log2(N)}{N} \cdot (N-1) = \le \log2(N)$$

So the upper bound of the average would be log2(N) => O(log2(N))

Q4:

```
interchange(Node node, int m, int n){
   // locate Node M and N and previous nodes for each
   Node nodeM = search(node, m);
   Node nodeN = search(node, n);
   Node beforeM = search(node, m-1);
   Node beforeN = search(node, n-1);

   // changing the pointers of the previous nodes of the M and N
   beforeM.next = nodeN;
   beforeN.next = nodeM;

   // swapping the pointers of the M and N
   Node tmp = nodeM.next;
   nodeM.next = nodeN.next;
   nodeN.next = tmp;

}

// helper: search(), search a node given a location n, zero indexed
search(Node head, int n){
   int cnt = 0;
   while (cnt != n){
       head = head.next;
   }

   return head;
}
```

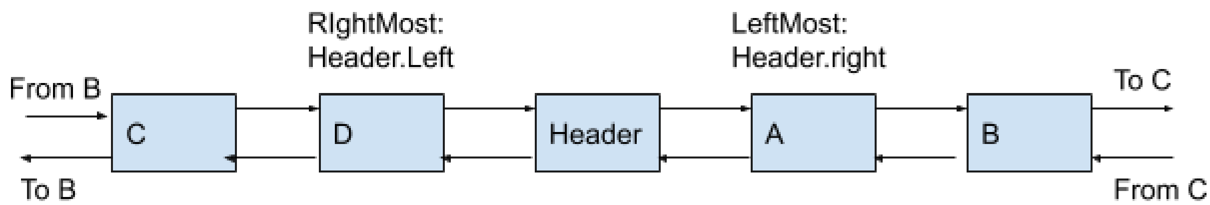
Q5:




```
// implement a doubly-linked node
public class Node {
   Datatype data;
   Node left;
   Node right;
}

// implement a deque data type
public class Deque{
   // reference to the left most node
   Node leftSide;

   // reference to the right most node
   Node rightSide;

   public void insertLeft(Node newNode){
       // setting right pointer for the newly inserted node
       newNode.right = this.leftSide;

       // setting the left pointer for the original left-most node
       this.leftSide.left = newNode;

       // update the left most node reference
       this.leftSide = newNode;
   }

   public void DeleteRight(){

       // referencing the next right-most node
       Node nextRight = this.rightSide.left;

       // clearing the current right most node's left pointer
       this.rightSide.left = null;

       // clearing the right pointer of the next right-most node
```

```
        nextRight.right = null;

        // update the right-most node reference
        this.rightSide = nextRight;

    }
}
```

Q6:




```
// implement a doubly-linked node (can be cyclic)
public class Node {
    Datatype data;
    Node left;
    Node right;
}

// implement a deque data type
public class Deque_alt{
    // reference to the header node
    // since we are using a circular doubly linked list
    // header.right node is the left most node
    // header.left node is the right most node
    Node header;


    public void insertRight(Node newNode){
        // copying the reference of the right most node before insertion
        Node curRight = this.header.left;

        // setting the left pointer of the new node towards the current right-most node
```

```java
        newNode.left = curRight;

        // setting the new node's right pointer points to header node
        newNode.right = this.header;

        // updating the left pointer of the header to newNode
        this.header.left = newNode;

        // change the right pointer of the previous right most node to the newNode
        curRight.right = newNode;

    }

    public void DeleteLeft(){
        // copying the reference of the current left-most node
        Node curLeft = this.header.right;

        // setting a new left most node by changing the pointer of the header
        this.header.right = curLeft.right;

        // changing the a new left most node's left pointer towards the header node
        curLeft.right.left = this.header;

        // clearing the previous left most node's left & right pointer
        curLeft.right = null;
        curLeft.left = null;

    }
}
```