```
//merge two sorted linked lists
//non-recursive implementation
public static Node merge(Node L1, Node L2) {
  if (L1 == null){ return L2;}
  if (L2 == null){ return L1;}

  Node front;
  if (L1.data < L2.data) {
    front = L1;
  } else {
    front = L2;
    L2 = L1;
    L1 = front;
  }
  while(L1.next != null && L2 != null) {
    if (L1.next.data <= L2.data) {
      L1 = L1.next;
    } else {
      Node temp = L1.next;
      L1.next = L2;
      L2 = temp;
    }
  }
  if (L1.next == null){ L1.next = L2;}
  return front;
}

//find common elements between two linked lists
//in this case type Integer
public IntNode commonElements(IntNode frontL1, IntNode frontL2){
 IntNode first = null, last = null;
 while (frontL1 != null && frontL2 != null) {
       if (frontL1.data < frontL2.data) {
          frontL1 = frontL1.next
       } else if (frontL1.data > frontL2.data) {
           frontL2 = frontL2.next;
       } else {
          IntNode ptr = new IntNode(frontL1.data, null);
          if (last != null) {
             last.next = ptr;
          } else {
             first = ptr;
          }
          last = ptr;
          frontL1 = frontL1.next;
          frontL2 = frontL2.next;
       }
  }
    return first;
}
//Circular Linked List
public boolean addAfter(String newItem, String afterItem)
throws NoSuchElementException {
         if (rear == null) { // empty
            return false;
         }
         Node ptr = rear;
         do {
             if (afterItem.equals(ptr.data)) {
                Node temp = new Node(newItem,ptr.next);
                ptr.next = temp;
                if (ptr == rear) { // new node becomes last
                    rear = temp;
                }
                return true;
             }
             ptr = ptr.next;
         } while (ptr != rear);
         return false; // afterItem not in list
}

//Circular Linked List
public boolean delete(String target) {
   if (rear == null) { // list is empty
       return false;
   }

   if (rear == rear.next) { // list has only one node
      if (target.equals(rear.data)) { // found, delete, leaves empty list
         rear = null;
         return true;
      } else {   // not found
         return false;
      }
   }

   Node prev = rear, curr = rear.next;
   do {
       if (target.equals(curr.data)) {
             prev.next = curr.next;
             if (curr == rear) { // if curr is last node, prev → new last node
        rear == prev;
       }
       return true;
       }
       // skip to next node
       prev = curr; curr = curr.next;
   } while (prev != rear); return false; // not found  }

  //Doubly Linked List
  public static DLLNode moveToFront(DLLNode front, DLLNode target) {
          if (target == null || front == null || target == front) {
             return;
          }

          // delink the target from the list
          target.prev.next = target.next;

          // make sure there is something after target before setting its prev
          if (target.next != null) {
             target.next.prev = target.prev;
          }
          target.next = front;
          target.prev = null;
          front.prev = target;
          return target;
 }
```

```
int orderedInsert (int arr[], int first, int last, int target){
// insert target into arr such that arr[first..last] is sorted,
//   given that arr[first..last-1] is already sorted.
//   Return the position where inserted.
  int i = last;
  while ((i > first) && (target < arr[i-1]))
    {
      arr[i] = arr[i-1];
      i = i - 1;
    }
  arr[i] = target;
  return i;
}
```

To find common elements in two ***unsorted*** lists, one of length m and the other of length n:

1) sort using merge sort
2) traverse through arrays to find intersection (common elements)
3) BIG-Oh: O(mlogm) + O(nlogn) ~~+ O(m+n~~)]- worst case time
4) Best case: O(mlogm) + O(nlogn) ~~+ min{m,n~~}]- best case time