HW-7

Question 1.

Consider the Node class:

```
class Node<T> {
    T data;
    Node<T> next;

    public Node(T data){
        this.data = data;
        next = null;
    }

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

Write each of the following pure functions:

- `public static <T> Node<T> addFirst(Node<T> head, T elm)`: This function adds `elm` at the beginning of the given list whose first node is referenced by `head`, and returns the resulting list.

- `public static <T> Node<T> addLast(Node<T> head, T elm)`: This function adds `elm` at the end of the given list whose first node is referenced by `head`, and returns the resulting list.

- `public static <T> Node<T> reverse(Node<T> head)`: This function returns a reversed copy of the given linked list whose first node is referenced by `head`.

- `public static <T extends Comparable<T>> boolean sorted(Node<T> head)`: This function returns true if the given linked list is sorted in ascending order, and false otherwise.

- `public static <T extends Comparable<T>> Node<T> merge(Node<T> head1, Node<T> head2)`: This function returns a merged list of two ascendingly sorted lists such that the merged list remains ascendingly sorted.

Question 2.

Considert the `MyLinkedList` class given below. The implementations of the methods `contains(E e)`, `get(int index)`, `indexOf(E e)`, `lastIndexOf(E e)`, and `set(int index, E e)` are omitted. Implement these methods.

```
public class MyLinkedList<E> implements MyList<E> {
private Node<E> head, tail;
private int size = 0; // Number of elements in the list

/** Create an empty list */
public MyLinkedList() {
}

/** Create a list from an array of objects */
public MyLinkedList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
        add(objects[i]);
}

/** Return the head element in the list */
public E getFirst() {
    if (size == 0) {
        return null;
    }
    else {
        return head.element;
    }
}

/** Return the last element in the list */
public E getLast() {
    if (size == 0) {
        return null;
    }
    else {
        return tail.element;
    }
}

/** Add an element to the beginning of the list */
public void addFirst(E e) {
    Node<E> newNode = new Node<>(e); // Create a new node
    newNode.next = head; // link the new node with the head
    head = newNode; // head points to the new node
    size++; // Increase list size

    if (tail == null) // the new node is the only node in list
        tail = head;
```

```
}

/** Add an element to the end of the list */
public void addLast(E e) {
    Node<E> newNode = new Node<>(e); // Create a new for element e

    if (tail == null) {
        head = tail = newNode; // The new node is the only node in list
    }
    else {
        tail.next = newNode; // Link the new with the last node
        tail = newNode; // tail now points to the last node
    }

    size++; // Increase size
}

@Override /** Add a new element at the specified index
           * in this list. The index of the head element is 0 */
public void add(int index, E e) {
    if (index == 0) {
        addFirst(e);
    }
    else if (index >= size) {
        addLast(e);
    }
    else {
        Node<E> current = head;
        for (int i = 1; i < index; i++) {
            current = current.next;
        }
        Node<E> temp = current.next;
        current.next = new Node<>(e);
        (current.next).next = temp;
        size++;
    }
}

/** Remove the head node and
 *  return the object that is contained in the removed node. */
public E removeFirst() {
    if (size == 0) {
        return null;
    }
    else {
        E temp = head.element;
        head = head.next;
        size--;
        if (head == null) {
            tail = null;
        }
        return temp;
    }
}

/** Remove the last node and
 * return the object that is contained in the removed node. */
public E removeLast() {
    if (size == 0) {
        return null;
    }
    else if (size == 1) {
        E temp = head.element;
        head = tail = null;
        size = 0;
        return temp;
    }
    else {
        Node<E> current = head;

        for (int i = 0; i < size - 2; i++) {
            current = current.next;
        }

        E temp = tail.element;
        tail = current;
        tail.next = null;
        size--;
        return temp;
    }
}

@Override /** Remove the element at the specified position in this
           *  list. Return the element that was removed from the list. */
public E remove(int index) {
    if (index < 0 || index >= size) {
        return null;
    }
    else if (index == 0) {
        return removeFirst();
    }
    else if (index == size - 1) {
        return removeLast();
    }
    else {
        Node<E> previous = head;
```

```
            for (int i = 1; i < index; i++) {
                previous = previous.next;
            }

            Node<E> current = previous.next;
            previous.next = current.next;
            size--;
            return current.element;
        }
    }

    @Override /** Override toString() to return elements in the list */
    public String toString() {
        StringBuilder result = new StringBuilder("[");

        Node<E> current = head;
        for (int i = 0; i < size; i++) {
            result.append(current.element);
            current = current.next;
            if (current != null) {
                result.append(", "); // Separate two elements with a comma
            }
            else {
                result.append("]"); // Insert the closing ] in the string
            }
        }

        return result.toString();
    }

    @Override /** Clear the list */
    public void clear() {
        size = 0;
        head = tail = null;
    }

    @Override /** Return true if this list contains the element e */
    public boolean contains(Object e) {
        // Left as an exercise
        return true;
    }

    @Override /** Return the element at the specified index */
    public E get(int index) {
        // Left as an exercise
        return null;
    }
```

```
@Override /** Return the index of the head matching element in
           *  this list. Return -1 if no match. */
public int indexOf(Object e) {
    // Left as an exercise
    return 0;
}

@Override /** Return the index of the last matching element in
           *  this list. Return -1 if no match. */
public int lastIndexOf(E e) {
    // Left as an exercise
    return 0;
}

@Override /** Replace the element at the specified position
           *  in this list with the specified element. */
public E set(int index, E e) {
    // Left as an exercise
    return null;
}

@Override /** Override iterator() defined in Iterable */
public java.util.Iterator<E> iterator() {
    return new LinkedListIterator();
}

private class LinkedListIterator
    implements java.util.Iterator<E> {
    private Node<E> current = head; // Current index

    @Override
    public boolean hasNext() {
        return (current != null);
    }

    @Override
    public E next() {
        E e = current.element;
        current = current.next;
        return e;
    }

    @Override
    public void remove() {
        // Left as an exercise
    }
}
```

```
  private static class Node<E> {
      E element;
      Node<E> next;

      public Node(E element) {
          this.element = element;
      }
  }

  @Override /** Return the number of elements in this list */
  public int size() {
      return size;
  }
}
```