内部类：内部类是定义在另一个类内部的类，作为其外部类的一部分。它提供了一种将逻辑相关的类组织在一起的方式，同时也能访问其外部类的成员，包括私有成员。

链表：链表是一种常用的数据结构，由一系列节点（Node）构成，每个节点包含数据域和指针域（用于指向下一个节点）。

基于课堂上对SimpleLList的学习，我们可以进一步思考链表的使用。

请基于SimpleLList已有的增删改查操作，完成以下任务： 1、判断链表中是否有重复的元素 2、交换某两个节点 3、将当前的链表进行反转

课后作业要求： 实现三个方法来完成上述任务

提示：充分理解并应用链表增删改查的逻辑和思路。

```
// CMPINF 0401 Summer 2024
// Simple primitive Linked List class of String.  See also SimpleAList.java
// We will see later how to make this class generic so that it can be a list
// of any Java reference type.
// See also main program ALvsLL.java

public class SimpleLList
{
    // Instance variables are a reference to the front of the list and an int.
    private Node front;
    private int size;

    // Self-referential inner class for the nodes.  Note that each Node has a
    // reference of type Node within it -- this allows a Node to connect to the
    // next Node in the chain.  This class is a private inner class -- this means
    // that it is not accessible outside of SimpleLList.  This is exactly what we
    // want here since the whole implementation of this list should be abstracted
    // out of the user's view.  A user of this class does not need to know or
    // rely on the fact that this class is implemented with linked nodes.
    private class Node
    {
        // The data in this class is private.  However, it is STILL accessible
        // to the SimpleLList class.  This is a characteristic of data declared
        // within inner classes.  Note that the data and next variables are
        // directly accessed in the methods below.
        private String data;
        private Node next;

        // Create a new Node with the specified data and null next field.
        public Node(String val)
        {
            data = new String(val);
            next = null;
        }

        // Create a new Node with the specified data and the specified next
        // field.
```

```
        public Node(String val, Node nextNode)
        {
            data = new String(val);
            next = nextNode;
        }
    }

    // Note: Constructor with init capacity is removed since
    // the idea of "capacity" is not defined for a linked list

    // Initialize a new empty SimpleLList
    public SimpleLList()
    {
        front = null;
        size = 0;
    }

    // Add new item at the end of the list.  Note that for this
    // method the add is a LOT more work than for the array, since
    // we must traverse to the end of the list to add the item.  There
    // are ways to make this more efficient.  A detailed comparison
    // of the array list vs. the linked list will be covered in the
    // CS 0445 course.  Note also that there is no "resize" here
    // since the size is always exactly correct.  In other words, in
    // a linked list, the logical size and physical size are always
    // equal.
    public boolean add(String val)
    {
        Node curr = front;
        if (curr == null)   // Special case for empty list.  Note that
                            // when adding to an empty list the front
            // instance variable must change.  However, adding anywhere
            // else will only change nodes in the middle of the list.
            // Special cases are always an issue with data structures
            // such as linked lists and trees.
            front = new Node(val);
        else
        {
            // Move down the list until we get to the last node.  The
            // new node is then linked after the last node.  How could we
            // change our SimpleLList object to avoid this loop?  Would
            // it be worthwhile?
            while (curr.next != null)
            {
                curr = curr.next;
            }
            curr.next = new Node(val);
        }
        size++;
        return true;
    }

    // Add item into arbitrary index.  We must traverse to that
    // location, but note that we do not have to shift.
```

```
    public boolean add(int loc, String val)
    {
        if (loc >= 0 && loc <= size)  // make sure loc is valid
        {
            Node newNode = new Node(val);  // make new Node
            // We again have a special case if we happen to be adding
            // at the front of the list.  Note that the constructor will
            // link the new node to the old front -- keeping the list
            // connected.
            if (loc == 0)
            {
                newNode.next = front;  // link newNode to OLD front
                front = newNode;       // make newNode new front
            }
            else
            {
                // Move down to the node BEFORE the location where we want
                // the new node to be. This is necessary because we must
                // link the old node to the new node.  We must also link the
                // new node to its successor.  Trace this code so that you
                // understand what is going on here.
                Node curr = front;
                for (int i = 1; i < loc; i++)
                    curr = curr.next;

                newNode.next = curr.next;      // link to successor
                curr.next = newNode;           // link previous to it
                // Note the order of the two lines above.  This order is
                // necessary -- we must get the old value of curr.next to
                // assign to the new node and then we must change curr.next
                // to point to the new node.
            }
            size++;
            return true;
        }
        return false;
    }

    // Remove item at stated index.  We again must traverse to the location
    // but again we do not have to shift.  This code is demonstrating a
    // common approach to this method -- keeping two references, one to the
    // current node and one to the node just before it.  This allows us to
    // have a reference before the node we want to delete.
    public String remove(int loc)
    {
        if (loc >= 0 && loc < size)
        {
            // Start out with the curr in the front and prev as null.
            // Move both down the list until we get to the desired location.
            Node prev = null;
            Node curr = front;
            for (int i = 0; i < loc; i++)
            {
                prev = curr;
```

```
            curr = curr.next;
        }
        // Get the data (this will be returned).
        String old = curr.data;

        if (prev == null) // If prev is still null then we did not move
                    // at all.  This is the special case of deleting the
                    // front node.  In this case we must update the front
                    // reference.
        {
            front = front.next;
            size--;
        }
        else  // Deleting a node in the middle of the list.  Note that
              // with the two references this is a simple assignment.
        {
            prev.next = curr.next;
            size--;
        }
        return old;
    }
    return null;
}

// The next two methods require a simple traversal of the list to get to
// the node so that we can "get" or "set" it.  We could put this traversal
// into both methods but another approach is to write a private "helper"
// method to traverse to the required location.  We then either "get" it
// or "set" it as required.

// Return item at stated index
public String get(int loc)
{
    Node curr = getNodeAt(loc);
    if (curr != null)
        return curr.data;
    else
        return null;
}

// Assign new value to stated location in list, returning old value
public String set(int loc, String val)
{
    Node curr = getNodeAt(loc);
    if (curr != null)
    {
        String old = curr.data;
        curr.data = val;
        return old;
    }
    return null;
}

// Private method to go to the specific location in the list.  If the
```

```java
    // index is invalid return null.
    private Node getNodeAt(int loc)
    {
        if (loc >= 0 && loc < size)
        {
            Node curr = front;
            for (int i = 0; i < loc; i++)
                curr = curr.next;
            return curr;
        }
        return null;
    }

    // Return logical size of list
    public int size()
    {
        return size;
    }


    public boolean hasDuplicate(){
        // to do

    }

    public void swap(int loc1,int loc2){
        // to do
    }

    public void reverse(){
        // to do
    }



}
```