```
class Link {
    public Object element;
    public Link next;

    public Link() { element = null; next = null; }
}

class List {
    private Link first;
    private int noOfElements;

    public List() { first = null; noOfElements = 0; }

    public Object getFirst() { return(first.element); }
    public int getNoOfElements() { return(noOfElements); }

    public void insertFront(Object newElement) { ... }
    public void insertRear(Object newElement) { ... }
    public void delete(Object existingElement) { ... }

    public ListIter newIter() { return(new ListIter(first)); }
}

class ListIter {
    private Link current;

    public ListIter(Link first) { current = first; }

    public Object getObject() {
        return(current == null ? null : current.element);
    }

    public void moveToNext() {
        if (current != null)
            current = current.next;
    }

    public boolean atEnd() { return(current == null); }
}
```

Q: Reusing the class(es) given above as needed, write a class named `Stack` to implement a stack.

```java
class Stack {
    private List elements;

    public Stack() { elements = new List(); }

    public Object getTop() { return(elements.getFirst()); }

    public void push(Object newElement) {
        elements.insertFront(newElement);
    }

    public Object pop() {
        Object first = getTop();

        if (first != null)
            elements.delete(first);

        return(first);
    }
}
```

Q: Reusing the class(es) given above as needed, write a class named `StringStack` to hold a stack of strings (stack elements are of type `String`, which is a predefined class in the Java API) where the interface of the class specifically refers to strings as opposed to just objects (i.e., instances of `String` as opposed to `Object`).

```java
class StringStack {
    private Stack elements;

    public StringStack() { elements = new Stack(); }

    public String getTop() {
        return((String) (elements.getTop()));
    }

    public void push(String newString) {
        elements.push(newString);
    }

    public String pop() {
        return((String) (elements.pop()));
    }
}
```

Q: Reusing the class(es) given above as needed, write a class named `IncStringStack` to implement a string stack in which the strings in the stack are always in a lexicographically increasing order (e.g. “abcd” is lexicographically less than “abce”). That is, the string at the top of the stack is lexicographically greater than the one right below the top, and so on. Such order should be checked against during insertion (i.e., push) and the operation should be allowed *only when* it does not violate the order.

Hint: Class `String` has a public method `int compareTo(String str)` that compares the current object to `str` and returns zero when they are equal, a negative integer when the current string is lexicographically less than the argument `str`; a positive integer, otherwise.

```
class IncStringStack extends StringStack {
    public void push(String newString) {
        String top = getTop();

        if (top == null || newString.compareTo(top) > 0)
            super.push(newString);
    }
}
```