线性表实现及应用

线性表提供了组织数据的一种方法，用它可以组织管理待办事项清单、礼品单、地址列表、甚至清单的清单等。这些清单为人们有条理的安排生活提供了一定的保障。所以说，线性表是一个基础性很强的数据结构。在很多的高级语言中，一般都会提供对这类数据组织方式的支持。

任务 1： 为指定的 List ADT 实现各种数据结构

在本次实验中，主要完成的任务是：

1、为指定的 List ADT（该 ADT 的具体要求请参见文件 List.java）实现三种数据结构：①使用顺序数组做为存储表示； ②使用单向链表做为存储表示； ③使用双向链表做为存储表示。不论哪种存储表示下实现的数据结构，实验中需要处理的数据类型都是 Character 类型。

2、用给定的输入数据文件验证所实现的数据结构是否正确。

3、使用表格列举每种数据结构下所实现的所有方法的时间复杂度。

为了方便进行测试验证，对 List 的各种操作指定了相应的命令符号，具体的符号含义如下：

+	insert
-	remove
=	replace
#	gotoBeginning
*	gotoEnd
>	gotoNext
<	gotoPrev
~	clear

假如对一个初始化空间大小为 16 的空表按顺序依次执行每一行的命令列表，并且假设该线性表中插入的元素为 Character 类型，那么表格中第二列即为执行相对应的第一列中的命令列表后调用 showStructure 方法的运行结果：

命令行内容	执行结果（调用 List 的 showStructure 方法）[1]
+a +b +c +d	a b c d {capacity = 16, length = 4, cursor = 3}
# > >	a b c d {capacity = 16, length = 4, cursor = 2}
*** < <**	a b c d {capacity = 16, length = 4, cursor = 1}
-	a c d {capacity = 16, length = 3, cursor = 1}
+e +f +f	a c e f f d {capacity = 16, length = 6, cursor = 4}
*** -**	a c e f f {capacity = 16, length = 5, cursor = 0}
-	c e f f {capacity = 16, length = 4, cursor = 0}
=g	g e f f {capacity = 16, length = 4, cursor = 0}
~	Empty list {capacity = 16, length = 0, cursor = -1}

实验完成之后，必须通过实验中提供的测试用例。借助测试用例的运行结果，用来检查所撰写的代码功能是否正确。测试用例中的每一行的内容都类似于上表中的每一行“命令行内容”列中

[1] 如果 List 实现的存储方式为链式结构，那么 capacity 的值即为真实的链表中的结点个数。实验文件中仅提供了基于数组实现的测试用例的运行结果。

所指示的内容。要求每执行一行，就调用 List 接口中的 showStructure 行为，用以验证该命令行的执行是否正确。每行“命令行内容”都不是独立的，是针对同一个 List 类型的对象实例运行的结果。实验包里包括了个文件，一个是“list_testcase.txt”，其内包含了测试用例；另一个是“list_result.txt”，其内包含了对应测试用例的运行结果。

任务 2： 创建一个可自动调整空间大小的 List 数据结构

观察任务 1 中基于数组实现的线性表的测试用例的运行结果，发现大部分时候空间的使用率是不高的（length 和 capacity 的比值反映了这一事实），而且还存在有空间不够用的例外发生。当然，基于链式存储实现的线性表则不存在此类问题。为了解决空间利用率以及空间不够用的问题，任务 2 将使用动态调整的方式改善数组空间的大小，方案可以有很多种，但在本次实验中将采用如下的调整方案，具体步骤如下：

① 使用 capacity 表示当前线性表的最大容量（即最多能够存储的线性表元素个数）；

② 初始情况下，capacity=1；

③当插入元素时线性表满，那么就重新生成一个容量为 2*capacity 的数组，将原数组中的 capacity 个元素拷贝到新数组中，让新数组成为当前线性表的存储表示；

④ 当删除元素之后，如果当前线性表中的元素个数 length 是 capacity 的四分之一时，则重新生成一个容量为 capacity/2 的数组，将原数组中的 length 个元素拷贝到新数组中，让新数组成为当前线性表的存储表示。

如果基于数组存储表示的线性表按照如上的方式完成空间的动态调整，那么构造方法中就不需要再指定初始空间的大小了。假如继续使用任务 1 中的示例数据，则运行结果如下表所示：

命令行内容	执行结果（调用 L 的 showStructure 方法）
+a +b +c +d	a b c d {capacity = 4, length = 4, cursor = 3}
# > >	a b c d {capacity = 4, length = 4, cursor = 2}
* < <	a b c d {capacity = 4, length = 4, cursor = 1}
-	a c d {capacity = 4, length = 3, cursor = 1}
+e +f +f	a c e f f d {capacity = 8, length = 6, cursor = 4}
* -	a c e f f {capacity = 8, length = 5, cursor = 0}
-	c e f f {capacity = 8, length = 4, cursor = 0}
=g	g e f f {capacity = 8, length = 4, cursor = 0}
~	Empty list {capacity = 1, length = 0, cursor = -1}

该任务中需要完成的工作如下：

① 按照任务 1 中的 List 接口定义，实现一个 ResizingAList 线性表，数组空间的调整方案如该任务中所描述的；

② 继续使用“list_testcase.txt”进行测试，并将结果中的每行运行结果中当前线性表的空间使用率和任务 1 中的空间使用率用图的方式展示其变化过程。

任务 3： 栈

众所周知，栈虽然是一个操作受限的线性表，但是其用途却很广泛，栈的数据结构实现非常简单，因此我们只从应用层面熟悉栈。请完成下面三个子任务。

①递归是一种解决很多复杂问题最简单的思想方法，而任何编程语言对递归程序的支持都是通过栈实现的。请利用课堂上讲解的“Hanoi 塔”问题的非递归转化方法完成对递归快速排序的非

递归转化。

②算术混合运算表达式的计算。表达式不仅能处理整数，还需要处理小数。表达式中涉及的运算符包括+、-、*、/、^(指数)。表达式可以包含括号（只包含圆括号）嵌套，因此要处理括号匹配失败的情形。

③当我们在使用很多软件时都有类似“undo”功能，比如 Web 浏览器的回退功能、文本编辑器的撤销编辑功能。这些功能都可以使用 Stack 简单实现，但是在现实中浏览器的回退功能也好，编辑器的撤销功能也好，都有一定的数量限制。因此我们需要的不是一个普通的 Stack 数据结构，而是一个空间有限制的 Stack，虽然空间有限，但这样的 Stack 在入栈时从不会溢出，因为它会采用将最久远的记录丢掉的方式让新元素入栈，也就是说总是按照规定的数量要求保持最近的历史操作。比如栈的空间是 5，当 a\b\c\d\e 入栈之后，如果继续让元素 f 入栈，那么栈中的元素将是 b\c\d\e\f。请设计一个满足上面要求的 LeakyStack 数据结构，要求该数据结构的每一个操作的时间复杂度在最坏情形下都必须满足 O(1)。

任务 4： 基数排序

使用自定义的队列数据结构实现对某一个数据序列的排序(采用基数排序)，其中对待排序数据有如下的要求：

①当数据序列是整数类型的数据的时候，数据序列中每个数据的位数不要求等宽，比如： 1、21、12、322、44、123、2312、765、56

②当数据序列是字符串类型的数据的时候，数据序列中每个字符串都是等宽的，比如："abc","bde","fad","abd","bef","fdd","abe"

注：radixsort1.txt 和 radixsort2.txt 是为上面两个数据序列提供的测试数据。