置插入和删除某个元素；而 findKth 则返回(作为参数而被指定的)某个位置上的元素。如果 34，12,52,16,12 是一个表，则 find(52)会返回 2;insert(x,2)可把表变成 34,12,x,52,16,12(如果我们插入到给定位置上的话)；而 remove(52)则又将该表变为 34,12,x,16,12。

当然，一个方法的功能怎样才算恰当，完全要由程序设计者来确定，就像对特殊情况的处理那样(例如，上述 find(1)返回什么?)。我们还可以添加一些操作，比如 next 和 previous，它们会取一个位置作为参数并分别返回其后继元和前驱元的位置。

3.2.1 表的简单数组实现

对表的所有这些操作都可以通过使用数组来实现。虽然数组是由固定容量创建的，但在需要的时候可以用双倍的容量创建一个不同的数组。它解决由于使用数组而产生的最严重的问题，即从历史上看为了使用一个数组，需要对表的大小进行估计。而这种估计在 Java 或任何现代编程语言中都是不需要的。下列程序段解释一个数组 arr 在必要的时候如何被扩展(其初始长度为 10)：

```
int [ ] arr = new int[ 10 ];
    ...
// 下面我们决定需要扩大 arr.
int [ ] newArr = new int[ arr.length * 2 ];
for( int i = 0; i < arr.length; i++ )
    newArr[ i ] = arr[ i ];
arr = newArr;
```

数组的实现可以使得 printList 以线性时间被执行，而 findKth 操作则花费常数时间，这正是我们所能够预期的。不过，插入和删除的花费却潜藏着昂贵的开销，这要看插入和删除发生在什么地方。最坏的情形下，在位置 0 的插入(即在表的前端插入)首先需要将整个数组后移一个位置以空出空间来，而删除第一个元素则需要将表中的所有元素前移一个位置，因此这两种操作的最坏情况为 $O(N)$。平均来看，这两种操作都需要移动表的一半的元素，因此仍然需要线性时间。另一方面，如果所有的操作都发生在表的高端，那就没有元素需要移动，而添加和删除则只花费 $O(1)$时间。

存在许多情形，在这些情形下的表是通过在高端进行插入操作建成的，其后只发生对数组的访问(即只有 findKth 操作)。在这种情况下，数组是表的一种恰当的实现。然而，如果发生对表的一些插入和删除操作，特别是对表的前端进行，那么数组就不是一种好的选择。下一节处理另一种数据结构：链表(linked list)。

3.2.2 简单链表

为了避免插入和删除的线性开销，我们需要保证表可以不连续存储，否则表的每个部分都可能需要整体移动。图 3-1 指出链表的一般想法。

图 3-1 一个链表

链表由一系列节点组成，这些节点不必在内存中相连。每一个节点均含有表元素和到包含该元素后继元的节点的链(link)。我们称之为 next 链。最后一个单元的 next 链引用 null。

为了执行 printList 或 find(x)，只要从表的第一个节点开始然后用一些后继的 next 链遍历该表即可。这种操作显然是线性时间的，和在数组实现时一样，不过其中的常数可能会比用数组