任意 List 的运行时间都是 $O(N)$，因为迭代器将有效地从一项到下一项推进。

对搜索而言，ArrayList 和 LinkedList 都是低效的，对 Collection 的 contains 和 remove 两个方法(它们都以 AnyType 为参数)的调用均花费线性时间。

在 ArrayList 中有一个容量的概念，它表示基础数组的大小。在需要的时候，ArrayList 将自动增加其容量以保证它至少具有表的大小。如果该大小的早期估计存在，那么 ensureCapacity 可以设置容量为一个足够大的量以避免数组容量以后的扩展。再有，trimToSize 可以在所有的 ArrayList 添加操作完成之后使用以避免浪费空间。

3.3.4 例：remove 方法对 LinkedList 类的使用

作为一个例子，我们提供一个例程，将一个表中所有具有偶数值的项删除。于是，如果表包含 6,5,1,4,2，则在该方法调用之后，表中仅有元素 5,1。

当遇到表中的项时将其从表中删除的算法有几种可能的想法：当然，一种想法是构造一个包含所有的奇数的新表，然后清除原表，并将这些奇数拷贝回原表。不过，我们更有兴趣的是写一个干净的避免拷贝的表，并在遇到那些偶数值的项时将它们从表中删除。

对于 ArrayList 这几乎就是一个失败策略。因为从一个 ArrayList 的几乎是任意的地方进行删除都是昂贵的操作。不过，在 LinkedList 中却存在某种希望，因为我们知道，从已知位置的删除操作都可以通过重新安排某些链而被有效地完成。

图 3-10 显示第一种想法。在一个 ArrayList 上，我们知道，remove 的效率不是很高的，因此该程序花费的是二次时间。LinkedList 暴露两个问题。首先，对 get 调用的效率不高，因此例程花费二次时间。而且，对 remove 的调用同样地低效，因为达到位置 i 的代价是昂贵的。

```
1       public static void removeEvensVer1( List<Integer> lst )
2       {
3           int i = 0;
4           while( i < lst.size( ) )
5               if( lst.get( i ) % 2 == 0 )
6                   lst.remove( i );
7               else
8                   i++;
9       }
```

图 3-10 删除表中的偶数：算法对所有类型的表都是二次的

图 3-11 显示矫正该问题的一种思路。我们不是用 get，而是使用一个迭代器一步步遍历该表。这是高效率的。但是我们使用 Collection 的 remove 方法来删除一个偶数值的项。这不是高效的操作，因为 remove 方法必须再次搜索该项，它花费线性时间。但是我们运行这个程序会发现情况更糟：该程序产生一个异常，因为当一项被删除时，由增强的 for 循环所使用的基础迭代器是非法的。(图 3-10 中的代码解释为什么这样的原因：我们不能期待增强的 for 循环懂得只有当一项不被删除时它才必须向前推进。)

```
1       public static void removeEvensVer2( List<Integer> lst )
2       {
3           for( Integer x : lst )
4               if( x % 2 == 0 )
5                   lst.remove( x );
6       }
```

图 3-11 删除表中的偶数：由于 ConcurrentModificationException 异常而无法运行