图 3-12 指出一种成功的想法：在迭代器找到一个偶数值项之后，我们可以使用该迭代器来删除这个它刚看到的值。对于一个 LinkedList，对该迭代器的 remove 方法的调用只花费常数时间，因为该迭代器位于需要被删除的节点(或在其附近)。因此，对于 LinkedList，整个程序花费线性时间，而不是二次时间。对于一个 ArrayList，即使迭代器位于需要被删除的节点上，其 remove 方法仍然是昂贵的，因为数组的项必须要移动，正如所料，对于 ArrayList，整个程序仍然花费二次时间。

```
1       public static void removeEvensVer3( List<Integer> lst )
2       {
3           Iterator<Integer> itr = lst.iterator( );
4
5           while( itr.hasNext( ) )
6               if( itr.next( ) % 2 == 0 )
7                   itr.remove( );
8       }
```

图 3-12　删除表中的偶数：对 ArrayList 是二次的，但对 LinkedList 是线性的

如果我们传递一个 LinkedList＜Integer＞运行图 3-12 中的程序，对于一个 400 000 项的 lst，花费的时间是 0.031 秒，而对于一个 800 000 项的 LinkedList 则花费 0.062 秒，显然这是线性时间例程，因为运行时间与输入大小增加相同的倍数。当我们传递一个 ArrayList＜Integer＞时，对于一个 400 000 项的 ArrayList 程序几乎花费 2.5 分钟，而对于 800 000 项的 ArrayList 程序花费大约 10 分钟；当输入增加到 2 倍时运行时间增加到 4 倍，这与二次的特征是一致的。

3.3.5　关于 ListIterator 接口

图 3-13 指出，ListIterator 扩展了 List 的 Iterator 的功能。方法 previous 和 hasPrevious 使得对表从后向前的遍历得以完成。add 方法将一个新的项以当前位置放入表中。当前项的概念通过把迭代器看做是在对 next 的调用所给出的项和对 previous 的调用所给出的项之间而抽象出来的。图 3-14 解释了这种抽象。对于 LinkedList 来说，add 是一种常数时间的操作，但对于 ArrayList 则代价昂贵。set 改变被迭代器看到的最后一个值，从而对 LinkedList 很方便。例如，它可以用来从 List 的所有的偶数中减去 1，而这对于 LinkedList 来说，不使用 ListIterator 的 set 方法是很难做到的。

```
1   public interface ListIterator<AnyType> extends Iterator<AnyType>
2   {
3       boolean hasPrevious( );
4       AnyType previous( );
5
6       void add( AnyType x );
7       void set( AnyType newVal );
8   }
```

图 3-13　java.util 包中 ListIterator 接口的子集

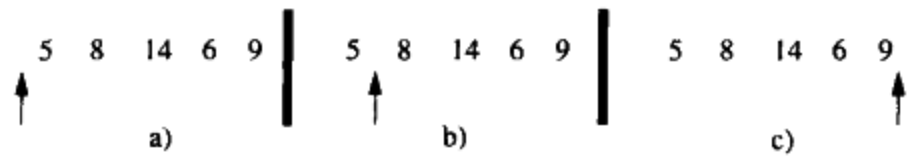

图 3-14　a) 正常起始点：next 返回 5，previous 是非法的，而 add 则把项放在 5 前；b) next 返回 8，previous 返回 5，而 add 则把项添加在 5 和 8 之间；c) next 非法，previous 返回 9，而 add 则把项置于 9 后