Java数据结构与算法解析(一)——表

标签： java 数据结构 算法 线性表 ArrayList

2017-08-27 13:27 3005人阅读 评论(0)

分类：

数据结构与算法（1）

目录(?) [+]

本节我们讨论常见用的**数据结构**——表。

如果要通俗简单的说什么是表，那我们可以这样说：**按顺序排好的元素集合就是表。**

表的概述

抽象数据类型是带有一组操作的一些对象的结合

1、定义：

线性表是一个线性结构，它是一个含有n≥0个结点的有限序列，对于其中的结点，有且仅有一个开始结点没有前驱但有一个后继结点，有且仅有一个终端结点没有后继但有一个前驱结点，其它的结点都有且仅有一个前驱和一个后继结点。

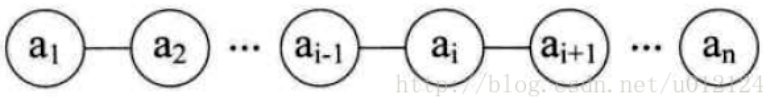
2、特征/性质

1）集合中必存在唯一的一个第一个元素

2）集合中必存在唯一的一个最后元素

3）除最后一个元素之外，均有唯一的后继

4）除第一个元素之外，均有唯一的前驱



在上图中，a1是a2的前驱，ai+1 是ai的后继，a1没有前驱，an没有后继 ，n为线性表的长度 ，若n==0时，线性表为空表 ，下图就是一个标准的线性表

数据结构与算法 (2)

文章存档

2017年09月 (1)
2017年08月 (5)
2017年07月 (6)
2017年06月 (4)
2017年05月 (14)

展开

阅读排行

Android Gradle知识梳理 (20535)
一篇博客让你了解RxJava (16805)
深入解析OkHttp3 (10738)
全面解析Notification (10104)
一篇博客理解Recyclerview的... (8881)
设计模式学习之策略模式 (7362)
浅谈安卓中的MVP模式 (7159)
Android进程保活全攻略（上） (6530)
Android性能优化系列之布局... (6363)
Android热修复学习之旅——... (5892)

评论排行

设计模式学习之策略模式 (15)
全面解析Notification (14)
一篇博客让你了解RxJava (9)
一篇博客让你了解Material D... (8)
Android性能优化系列之布局... (7)
浅谈安卓中的MVP模式 (7)
Android Studio常用技巧汇总 (7)
Android性能优化系列之内存... (7)
一篇博客理解Recyclerview的... (7)
《深入理解java虚拟机》学习... (7)

推荐文章

* CSDN日报20170828——《4个方法快速打造你的阅读清单》
* Android检查更新下载安装
* 动手打造史上最简单的 Recycleview 侧滑菜单
* TCP网络通讯如何解决分包粘包问题
* 程序员的八重境界
* 四大线程池详解

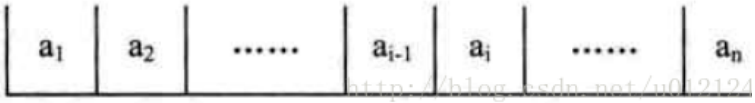
最新评论

一篇博客让你了解Material Design的使用 暗夜丿使者 ：很好，感谢分享！
一篇博客让你了解Material Design的使用 来自星星的谢广坤 ：牛BI了我的哥
一篇博客让你了解Material Design的使用 礼枝书笙 ：受用了，赞赞
一篇博客让你了解Material Design的使用 十四期-苏怡仙 ：又懂了点，感谢你的分享
一篇博客让你了解Material Design的使用 qq_39980761 ：很好很强大
一篇博客让你了解Material Design的使用 qq_37867965 ：不错
一篇博客让你了解Material Design的使用 mg52033 ：简单易懂
一篇博客让你了解Material Design的使用

学号	姓名	性别	出生年月	家庭地址
1	张三	男	1995.3	东街西巷 1 号 203 室
2	李四	女	1994.8	北路 4 弄 5 号 6 室
3	王五	女	1994.12	南大道 789 号
……	……	……	……	……

线性表分为如下几种：

顺序存储方式线性表



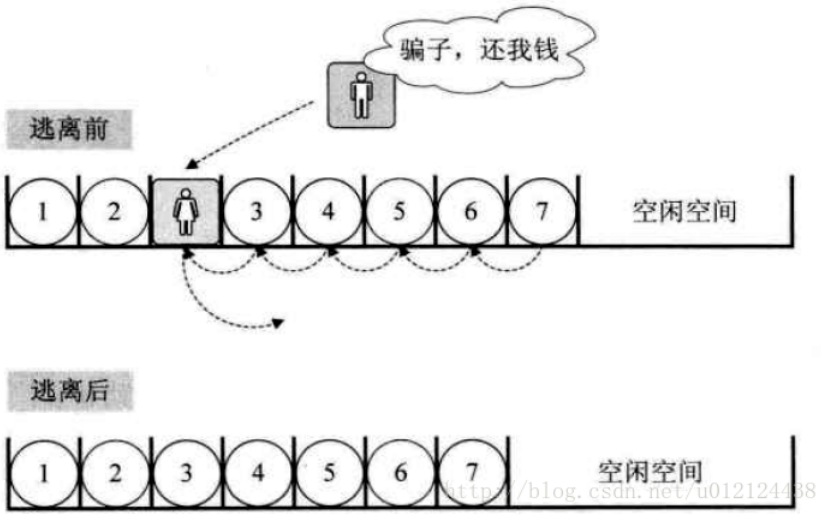
顺序存储方式线性表中，存储位置连续，可以很方便计算各个元素的地址

如每个元素占C个存储单元，那么有： Loc(An) = Loc(An-1) + C,于是有： Loc(An) = Loc(A1) + (i-1) *C;

优点：查询很快

缺点：插入和删除效率慢

下图很形象的表现了，插入和删除慢的特点



表的简单数组实现

顺序存储方式线性的典型就是数组，对于表的所有操作都可以通过使用数组来实现。虽然数组创建时就已经是固定大小，但在需要的使用可以用双倍的容量创建一个不同的数组。下面是扩容的伪代码：

```
1       int[] aar = new int[10];
2       //扩大aar
3       int[] newArr = new int[aar.length * 2];
4       for (int i = 0; i < aar.length; i++) {
5           newArr[i] = aar[i];
6       }
7       aar = newArr;
```

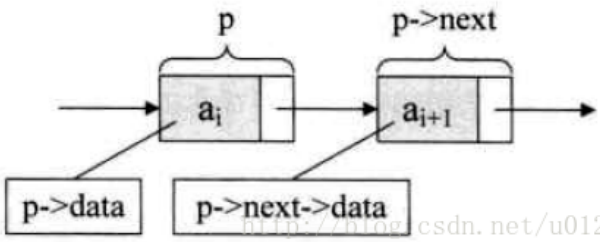
数组的实现使得printList以线性时间被执行，而findKth(返回特定位置上的元素)则花费常数的时间。

最坏的情形是，在位置0插入一个元素，需要将数组中所有元素向后移动一个位置，而删除一个元素，则需要将所有元素向前移动一个位置，两种情况复杂度都是O（n）。平均来看，两种操作都需要移动表一半的元素，因此需要线性时间，但是如果插入和删除都发生在数组的最尾，则插入和删除都只需要花费O（1）的时间。

如果频繁的插入和删除发生在表的最前端，则使用链表会更好。

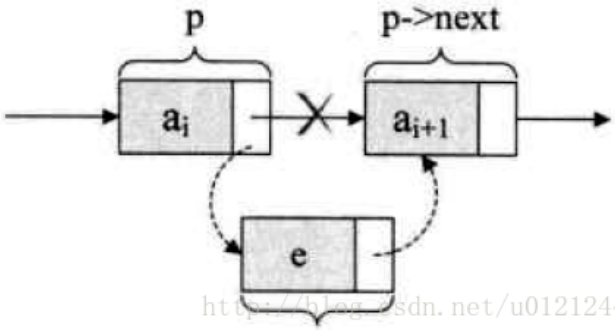
链式存储方式线性表

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素，这
是连续的，也可以是不连续的



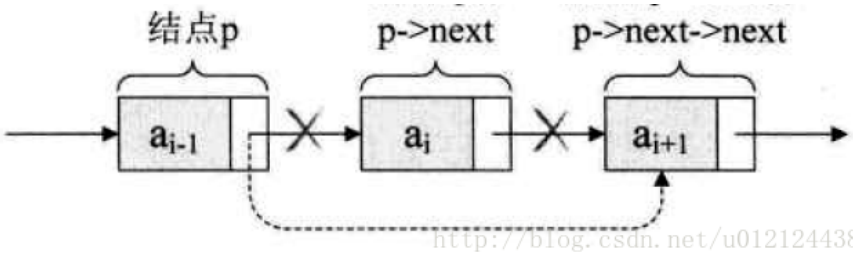
优点：相对于数组，删除还插入效率高

缺点：相对于数组，查询效率低



要执行插入操作，只需要如下的代码：

```
s->next = p->next
p->next = s ;
```

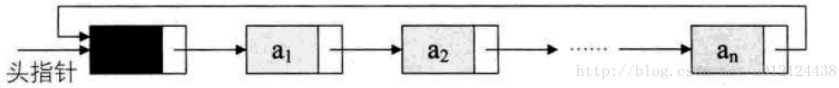


执行删除操作，只需要如下的代码：

```
p->next = p->next->next
```

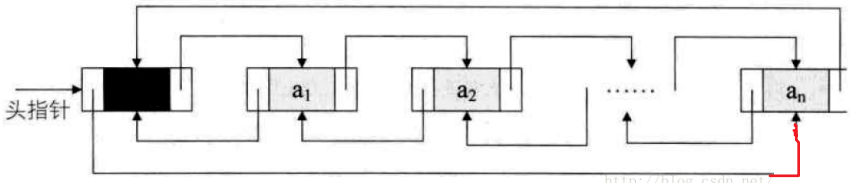
循环链表

将单链表中终端结点的指针端由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相连的单链表称为单循环链表，简称循环链表

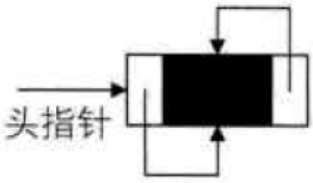


双向循环链表

双向循环链表是单向循环链表的每个结点中，再设置一个指向其前驱结点的指针域



对于空的双向循环链表



双向循环链表插入

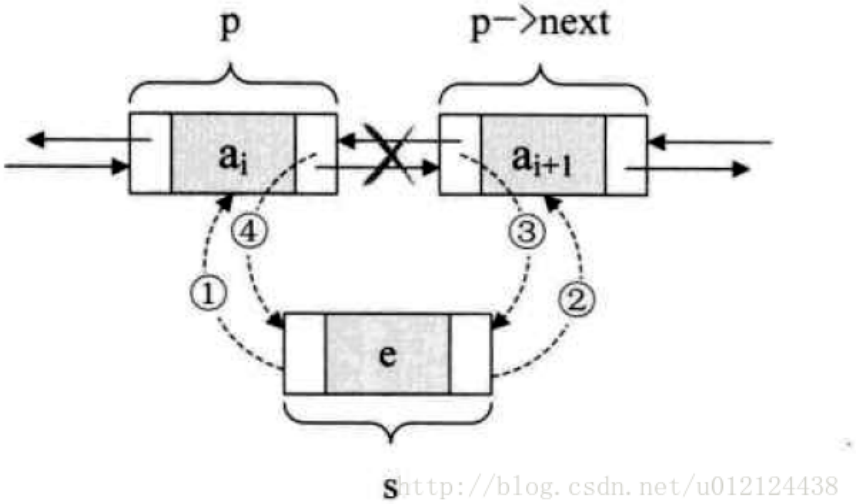


图 3-14-3

```
s - >prior = p;              /*把 p 赋值给 s 的前驱，如图中①*/
s -> next = p -> next;       /*把 p->next 赋值给 s 的后继，如图中②*/
p -> next -> prior = s;      /*把 s 赋值给 p->next 的前驱，如图中③*/
p -> next = s;               /*把 s 赋值给 p 的后继，如图中④*/
```

Java Collection Api中的表

1.Iterator

Iterator接口的思路，通过Iterator方法，每个集合均创建并返回给客户一个实现Iterator接口的对象，并将当前位置的概念在对象内部存储下来。

```
1  public interface Iterator<E> {
2
3      boolean hasNext();
4
5      E next();
6
7      default void remove() {
8          throw new UnsupportedOperationException("remove");
```

```
    }

}
```

Iterator中的方法有限，因此，很难使用Iterator做遍历Collection意外的任何工作。Iterator还包含一个remove()方法。该方法的作用是删除next()最新返回的项（此后不能再调用remove()，直到你下一次调用next()）。

如果对正在被迭代的集合进行结构上的改变（即对该集合使用add,remove或clear），那么迭代器将不再合法（并且在其后使用该迭代器将出现ConcurrentModificationException异
了避免迭代器准备给出某一项作为下一项而该项此后或者被删除，所以只有在需要
代器的时候，我们才应该获取迭代器。然而，如果迭代器调用了它自己的remove方
代器就仍然合法的。

2.List接口

```
public interface List<E> extends Collection<E> {

    int size();

    boolean isEmpty();

    Iterator<E> iterator();

    Object[] toArray();

    <T> T[] toArray(T[] a);

    boolean add(E e);

    boolean remove(Object o);

    boolean containsAll(Collection<?> c);

    boolean addAll(Collection<? extends E> c);

    boolean addAll(int index, Collection<? extends E> c);

    boolean removeAll(Collection<?> c);

    boolean retainAll(Collection<?> c);

    void clear();

    boolean equals(Object o);

    int hashCode();

    E get(int index);

    E set(int index, E element);

    void add(int index, E element);

    E remove(int index);

    int indexOf(Object o);

    int lastIndexOf(Object o);

    ListIterator<E> listIterator();

}
```

List ATD有两种流行的实现方式，ArrayList和LinkedList。

ArrayList的优点是，get和set调用花费常数时间。缺点是新项的插入和现有项的删除代价昂贵，除非变动的是ArrayList的末端。

LinkedList优点是在表的前端添加和删除都是常数时间，缺点是不容易作索引，get的调用是昂贵的，除非是接近表的端点

```
public static void makeList1(List<Integer> lst,int n){
        lst.clear();
        for (int i = 0; i < n; i++) {
            lst.add(i);
        }
    }
```

不管ArrayList还是LinkedList作为参数被传递，makeList1的运行时间都是O（N）[illegible]的每次调用都是在表的末端进行从而花费常数时间（可以忽略对ArrayList偶尔扩展）。[illegible]如果我们通过在表的前端添加一些项来构造一个List:

```
public static void makeList2(List<Integer> lst,int n){
        lst.clear();
        for (int i = 0; i < n; i++) {
            lst.add(i);
        }
    }
```

对于LinkedList它的运行时间是O（N），但是对于ArrayList其运行时间则是O（n^2），因为在ArrayList中，在前端进行添加是一个O（N）操作。

```
public static int sum(List<Integer> lst){
        int total = 0;
        for (int i = 0; i < n; i++) {
            total+=lst.get(i);

        }
        return total;
    }
```

这里，ArrayList的运行时间是O（N），但对于LinkedList来说，其运行时间则是O（n^2），因为LinkedList中，对get的调用为O（N）操作。可是，要是使用一个增强的for循环，那么它对任意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，对Collection的contains和remove方法调用均花费线性时间。

例子：remove方法对LinkedList类的使用

例子1：假设现在有6，5，1，4，2五个数，需要在方法调用之后去除所有的偶数。

思路：

1.创建一张包含所有奇数的新表，清除原表，再将奇数拷贝回去。

2.直接在原表中进行遍历，遇到偶数时直接进行移除。

ArrayList和LinkedList针对于remove都是低效的，在LinkedList中，到达i位置的代价是昂贵的。

```
public static void removEventVer1(List<Integer> lst) {
        int i = 0;
```

```
        while (i < lst.size()) {
            if (lst.get(i) % 2 == 0) {
                lst.remove(i);
            } else {
                i++;
            }
        }
    }
```

对于LinkedList来说，上面的解法运行时间则是O（n^2），使用迭代器的效率会更好，当然在使用迭代器时，我们不能直接使用List的

remove,否则会抛出异常，就像下面的写法（增强for循环底层还是用的迭代器）

```
  public static void removEventVer2(List<Integer> lst) {
        for (Integer x : lst) {
            if (x % 2 == 0) {
                lst.remove(x);
            }

        }
    }
```

为了解决上面的问题，我们可以直接使用迭代器的remove方法，这样做是合法的

```
 public static void removEventVer3(List<Integer> lst) {
        Iterator<Integer> itr = lst.iterator();
        while (itr.hasNext()){
            if (itr.next()%2==0){
                itr.remove();
            }
        }

    }
```

使用了Iterator以后，LinkedList的remove操作消耗的就是O（n）时间，因为Iterator已经位于需要被删除的节点上。

而即使使用Iterator，ArrayList的remove方法还是O（n^2），因为删除，数组的数还是需要进行移动。

ListIterator接口

ListIterator接口扩展了Iterator，hasNext和hasPrevious方法，使得既可以从前遍历也可以从尾巴进行遍历，add在当前位置插入一个新的项，set方法改变Iterator调用hasNext或hasPrevious返回的当前值。

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    boolean hasPrevious();
    void remove();
    void set(E e);
    void add(E e);
```

实现一个ArrayList

下面，我们自己手写一个ArrayList，且支持泛型，代码如下：

```
public class MyArrayList<T> implements Iterable<T> {

    private static final int DEFAULT_CAPACITY = 10;
    private T[] mArray;
```

```
5       private int mArraySize;
6
7       @Override
8       public Iterator<T> iterator() {
9           return new ArrayIterator();
10      }
11
12
13      private class ArrayIterator implements Iterator<T> {
14          private int currentPositon;
15
16          @Override
17          public boolean hasNext() {
18              return currentPositon < mArraySize;
19          }
20
21          @Override
22          public T next() {
23              if (!hasNext()) {
24                  throw new NoSuchElementException();
25              }
26
27              return mArray[currentPositon++];
28          }
29
30          @Override
31          public void remove() {
32              MyArrayList.this.remove(--currentPositon);
33          }
34      }
35
36
37      public void trimTosize() {
38          ensureCapacity(size());
39      }
40
41      public int size() {
42          return mArraySize;
43      }
44
45      public boolean isEmpty() {
46          return mArraySize == 0;
47      }
48
49
50      public MyArrayList(int size) {
51          if (size < DEFAULT_CAPACITY) {
52              mArraySize = size;
53          } else {
54              ensureCapacity(DEFAULT_CAPACITY);
55          }
56      }
57
58      private void ensureCapacity(int newCapacity) {
59          T[] newArray = (T[]) new Object[newCapacity];
60          for (int i = 0; i < mArray.length; i++) {
61              newArray[i] = mArray[i];
62          }
63          mArray = newArray;
64      }
65
66      public boolean add(T t) {
67          add(t, mArraySize);
68          return true;
69      }
70
71      public void add(T t, int position) {
72          if (mArraySize == mArray.length) {
73              ensureCapacity(mArraySize * 2 + 1);
74          }
75          for (int i = position; i < mArraySize - 1; i++) {
```

```
            mArray[i + 1] = mArray[i];
        }
        mArray[position] = t;
        ++mArraySize;
    }

    public T reomve() {
        return remove(mArraySize);
    }

    private T remove(int position) {
        T t = mArray[position];
        for (int i = position; i < mArraySize - 1; i++) {
            mArray[i] = mArray[i + 1];
        }
        --mArraySize;
        return t;
    }

    public T get(int position) {
        if (position < 0 || position > mArraySize) {
            throw new ArrayIndexOutOfBoundsException();
        }
        return mArray[position];
    }

    public T set(T t) {
        return set(t, mArraySize - 1);
    }

    public T set(T t, int position) {
        if (position < 0 || position > mArraySize) {
            throw new ArrayIndexOutOfBoundsException();
        }
        T old = mArray[position];
        mArray[position] = t;
        return old;
    }
}
```

值得一提的是，我们不能直接new T[]，而是需要通过下面的代码创建一个泛型的数组

```
T[] newArray = (T[]) new Object[newCapacity];
```

还有一点值得说明的是，在ArrayIterator中使用MyArrayList.this.remove是为了避免和迭代器自身的remove冲突

```
 @Override
        public void remove() {
            MyArrayList.this.remove(--currentPositon);
        }
```

实现LinkedList

在LinkedList中，最前端的节点叫做头节点，最末端的节点叫做尾节点。这两个额外的节点的存在，排除许多特殊情况，极大简化了编码。

例如：如果不使用头节点，那么删除第一个节点就是特殊情况，因为在删除时需要重新调整链表到第一个节点的链，还因为删除**算法**一般还要访问被删除节点前面的那个节点（如果没有头节点的话，第一个节点就会出现前面没有节点的特殊情况）。

```
public class MyLinkedList<T> implements Iterable<T> {

    private Node<T> headNote;
```

```
4       private Node<T> endNote;
5
6       private int mSize;
7       private int modCount;
8
9       public MyLinkedList() {
10          init();
11      }
12
13      private void init() {
14          headNote = new Node<>(null, null, null);
15          endNote = new Node<>(null, headNote, null);
16          headNote.mNext = endNote;
17
18          mSize = 0;
19          modCount++;
20      }
21
22      public int size() {
23          return mSize;
24      }
25
26      public boolean isEmpty() {
27          return mSize == 0;
28      }
29
30      public boolean add(T t) {
31          addBefore(t, size());
32          return true;
33      }
34
35      public T get(int index) {
36          Node<T> temp = getNode(index, 0, size());
37          return temp.mData;
38      }
39
40      public T remove(int position) {
41          Node<T> tempNode = getNode(position);
42          return remove(tempNode);
43      }
44
45      private T remove(Node<T> tempNode) {
46          tempNode.mPre.mNext = tempNode.mNext;
47          tempNode.mNext.mPre = tempNode.mPre;
48          mSize--;
49          modCount++;
50          return tempNode.mData;
51      }
52
53      public T set(int index, T t) {
54          Node<T> tempNode = getNode(index);
55          T old = tempNode.mData;
56          tempNode.mData = t;
57          return old;
58      }
59
60      private Node<T> getNode(int index) {
61          return getNode(index, 0, size() - 1);
62      }
63
64
65
66      private Node<T> getNode(int index, int lower, int upper) {
67          Node<T> tempNode;
68
69          if (lower < 0 || upper > mSize) {
70              throw new IndexOutOfBoundsException();
71          }
72
73          if (index < mSize / 2) {
74              tempNode = headNote.mNext;
```

```
75              for (int i = 0; i < index; i++) {
76                  tempNode = tempNode.mNext;
77              }
78          } else {
79              tempNode = endNote;
80              for (int i = mSize; i > index; i--) {
81                  tempNode = tempNode.mPre;
82              }
83          }
84          return tempNode;
85      }
86
87
88      private static class Node<T> {
89
90          private Node<T> mNext;
91          private T mData;
92          private Node<T> mPre;
93
94          public Node(T data, Node<T> pre, Node<T> next) {
95              mData = data;
96              mPre = pre;
97              mNext = next;
98          }
99      }
100
101
102     private class LinkedListIterator implements Iterator<T> {
103         private Node<T> currentNode = headNote.mNext;
104         private int expectedModCount = modCount;
105         private boolean okToMove;
106
107         @Override
108         public boolean hasNext() {
109             return currentNode != endNote;
110         }
111
112         @Override
113         public T next() {
114             if (modCount != expectedModCount) {
115                 throw new ConcurrentModificationException();
116             }
117             if (!hasNext()) {
118                 throw new NoSuchElementException();
119             }
120             T t = currentNode.mData;
121             currentNode = currentNode.mNext;
122             okToMove = true;
123             return t;
124         }
125
126          @Override
127         public void remove() {
128             if (modCount != expectedModCount) {
129                 throw new ConcurrentModificationException();
130             }
131             if (!okToMove) {
132                 throw new IllegalStateException();
133             }
134             MyLinkedList.this.remove(currentNode.mPre);
135             expectedModCount++;
136             okToMove = false;
137         }
138
139     @Override
140     public Iterator<T> iterator() {
141         return new LinkedListIterator();
142     }
143 }
```

1.modCount代表自从构造以来对链表所做改变的次数。每次对add或remove的调用都将更新modCount。想法在于,当一个迭代器被建立时，他将存储集合的modCount。每次个迭代器方法（next或remove）的调用都将该链表内的当前modCount检测在迭代器内存储的modCount，并且当两个计数不匹配时，抛出一个ConcurrentModificationException异常。

2.在LinkedListIterator中，currentNode表示包含由调用next所返回的项的节点。注意，当currentNode被定位在endNote，对next调用是非法的。

在LinkedListIterator的remove方法中，currentNode是保持不变的，因为curren
前面节点被删除的影响，与ArrayIterator不同，（在ArrayIterator中，项被移动，
current）

参考书籍:

《数据结构与算法分析》

- 上一篇 一篇博客让你了解Material Design的使用
- 下一篇 Java数据结构与算法解析(二)——栈