## 1.3 递归简论

当一个函数用它自己定义时，就称为时递归(recursive)的

基准情况(base case)，即此时函数的值可以直接算出而不用求助递归。

Java的递归方法若无基准情况也是毫无意义的。

虽然我们定义一个方法用的是这个方法本身，但是我们并没有用方法本身定义该方法的一个特定实例。

所以，递归并不是循环推理(circular logic)

递归调用将反复进行直到基准情况出现

递归有两个基本法则：

1、基准情况 base case必须总要有些基准的情形，他们不用递归就能求解

2、不断推进 making progress 对于那些要递归求解的情形，递归调用必须总能够朝着一个基准情形推进

3、设计法则，假设所有的递归调用都能运行

4、合成效益法则，在求解一个问题的同一个实例时，切勿在不同的递归调用中作重复性的工作

## 1.4 实现泛型特性构建

1.4.1 使用Object表示泛型

1.4.2 基本类型的包装

1.4.3 使用接口类型表示泛型

1.4.4 数组类型的兼容性

## 1.5 利用Java5泛性实现反省特性成分

1.5.1 简单的泛性类和接口

1.5.2 自动装箱/拆箱

1.5.3 带有限制的通配符

# 算法分析

算法（algotithm）是为求解一个问题所需要遵循的、被清楚指定的简单指定的集合。

这一章，我们将讨论

* 如何估计一个程序所需要的时间
* 如何将一个程序的运行时间从天或年降低到秒甚至更少
* 粗心使用递归的结果
* 将一个数自乘得到其幂，以及计算两个数的最大公因数的非常有效的算法

## 2.1 数学基础

定义2.1 如果存在正常数和使得当N≥时，T(N)≤cf(N),则记为T(N)=O(f(n))。

定义2.2 如果存在正常数c和n0，使得当N≥cg(N)时，T(N)≥cg(N)，则记为T(N) =Ω(g(N))。

定义2.3 T(N) = Ɵ（h（N））当且仅当T(N)=O(h(N))和T(N)=Ω(h(N))。

定义2.4 如果对每一个正常数c，都存在常数n0，使得当N>n0时，T(N)<cp(N),则T(N) = o(p(N))。

有时也可以说，如果T(N)=O(p(N))且T(N)≠Ɵ（p（N））,则T(N) = o(p(N))

这些定义的目的是要在函数间建立一种相对的级别。给定两个函数，通常存在一些点，在这些点上一个函数的值小于另一个函数的值，因此一般宣城，比如f(N)<g(N)，时没有什么意义的。于是我们比较他们的相对增长率(relative rate of growth)。当将相对增长率从应用到算法分析时，我们将会明白为什么它是重要的度量。

如果用传统的不等式来计算增长率，那么

第一个定义是说T(N)的增长率小于或等于f(N)的增长率，

第二个定义T（N）=Ω（g（N））是说T（N）的增长率大于或等于g（N）的增长率。

第三个定义T（N）= Ɵ（h（N））说的是T(N)的增长率等于h(N)的增长率。

最后一个定义T（N）=o（p（N））说的则是T（N）的增长率小于p（N）的增长率。

法则1

如果T1(N) = O(f(N))且T2(N)=O(g(N)),那么

1. T1(N) + T2(N)=O(f(N)+g(N))(直观地和非正式地可以写成max(O(f(N),O(g(N))))
2. T1(N) \* T2(N)=O(f(N)\*g(N))

法则2

如果T(N)是一个k次多项式,则T(N)=Ɵ(Nk).

法则 3

对于任意常数k,logkN=O(N),他告诉我们,对数增长得非常缓慢

有几点要注意:

首先,将常数或者低阶项放进O是非常坏的习惯,不要写成T(N)=O(2N2)或者T(N) = O(N2 + N).

在大O分析中,各种简化都是可能发生的.低阶项一般可以忽略,而常数也可以弃掉.此时要求的精度是非常粗糙的.

第二,我们总能够通过计算极限来确认两个函数的的相对增长率,必要的时候可以使用洛必达法则.该极限可能有四种值:

* 极限是0 这意味着f(n) = o(g(n))
* 极限是c ≠ 0,这意味着f(N) = Ɵ(g(N))
* 极限是∞ 这意味着f(n) = Ω(g(N)),即g(N) = o(f(N))
* 极限摆动,二者无关

|  |  |
| --- | --- |
| 函数 | 名称 |
|  | 常数 |
|  | 对数 |
| log2N | 对数平方 |
| N | 线性的 |
| N |  |
| N2 | 二次的 |
| N3 | 三次的 |
| 2N | 指数的 |

典型的增长率

## 2.2 模型

为了在正式的架构中分析算法,我们需要一个计算模型.

我们的模型基本上是一台标准的计算机,在机器中被顺序地执行.

我们假设模型做任何一件简单的工作都恰好花费一个时间单位

模型机有无限的内存

## 2.3 要分析的问题

主要因素:

1使用的算法

2对该算法的输入

典型的情况是,输入的大小是主要的考虑方面

我们定义两个函数Tavg(N)和Tworst(N),分别为算法对于输入量N所花费的平均时间和运行时间.显然,Tavg≤Tworst(N).

平均情形常常反应典型的行为

最坏情形的性能则代表对任何可能输入的性能的一种保证

程序是算法以一种特殊编程语言的实现,程序设计语言的细节几乎总是不影响大O的答案.

## 2.4 运行时间的计算

为了简化分析,我们将采纳一下的约定:

不存在特定的时间单位

因此,我们抛弃一些前导的常数

并抛弃低阶项

从而要做的就是计算大O的运行时间.

### 2.4.1 一个简单的例子

计算的一个简单程序片段

**public static int** sum(**int** n){  
 **int** partialSum = 0;  
 **for** (**int** i = 0;i <= n;i++){  
 partialSum += i^3;  
 }  
 **return** partialSum;  
}

### 2.4.2 一般法则

* **法则1 for循环**

1个for循环的运行时间至多是该循环内部那些语句的运行时间乘以迭代次数

* **法则2 嵌套的for循环**

从里向外分析这些循环.在一组嵌套循环内部的一句语句总的运行时间为该语句的运行时间诚意改组所有的for循环的大小的乘积

例如:

**public static int** TestON2(**int** n) {  
 **int** k = 0;  
 **for** (**int** i = 0; i < n; i++) {  
 **for** (**int** j = 0; j < n; j++) {  
 k++;  
 }  
 }  
 **return** n;  
}

* **法则3 顺序语句**

将各个语句的运行时间求和即可,这意味着,其中的最大值就是所得的运行时间

**public static int**[] testOrder(**int** n) {  
 **int**[] a = **new int**[n];  
 **for** (**int** i = 0; i < n; i++ ){  
 a[i] = 0;  
 }  
 **for** (**int** i = 0; i< n ;i++) {  
 **for** (**int** j =0;j<n;j++) {  
 a[i] += a[j] +i +j;  
 }  
 }  
 **return** a;  
}

上面的程序片段先是花费O(N),然后是O(N2),因此总量也是O(N2)

* **法则4 if/else语句**

if(condition){

$1

}else{

$2;

}

# 第三章 表、栈和队列

## 3.1 抽象数据类型（abstract data type）ADT

抽象数据类型是带有一组操作的一些对象的集合.

## 3.2 表ADT

对于除空表外的任何表,我们说Ai后继Ai-1(或继Ai-1之后,i<N),并称Ai-1前驱Ai(i>0).

### 3.2.1 表的简单数组实现

对于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(其初始容量为10)

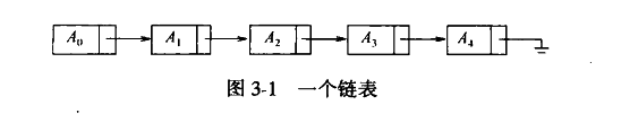
**int** [] arr = **new int**[10];  
**int** [] newArr = **new int**[arr.**length** \* 2];  
**for**(**int** i = 0; i < arr.**length** ; i ++){  
 newArr [i] = arr[i];  
}  
arr = newArr;

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都是O(N).平均来看,这两种操作都需要移动表的一般的元素,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最前端,那没有元素要移动,而添加和删除则只花费O(1)时间.

存在许多情形,在这些情形下得表示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).在这种情形下,数组是表的一种恰当的实现.然后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

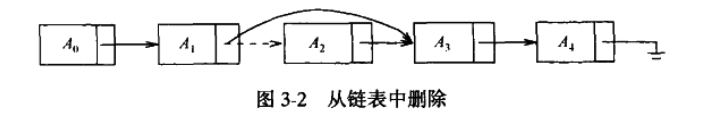
为了避免插入和删除的线性开销,我们需要保证表可以不连续存储,否则表的每个部分都可能需要整体移动.图3-1指出链表的一般想法



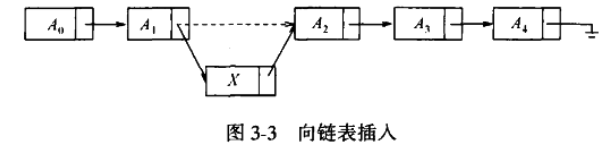
链表由一些列节点组成,这些节点不必再内存中相连.每个节点均含有表元素和到包含该元素后继几点的链(link),我们称之为next链.最后一个单元的next链引用为null.

为了执行pintList或者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.这些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实现时要大.findKth操作不如数组实现时的效率高;findKth(i)花费O(i)的时间并以这种明显的方式遍历链表而完成.在实践中这个界是保守的,因为调用findKth常常是以(按i)排序后的方式进行.例如findKth(2),findKth(3),findKth(4),以及finKth(6)可通过对表的一次扫描同时实现.

remove方法可以通过修改一个next引用来实现.图3-2给出在原表中删除第三个元素的结果.



insert方法需要使用new操作符从系统中取得一个新的节点,伺候执行两次引用的调节.其一般想法在图3-3中给出,其中虚线表示原来的next引用.

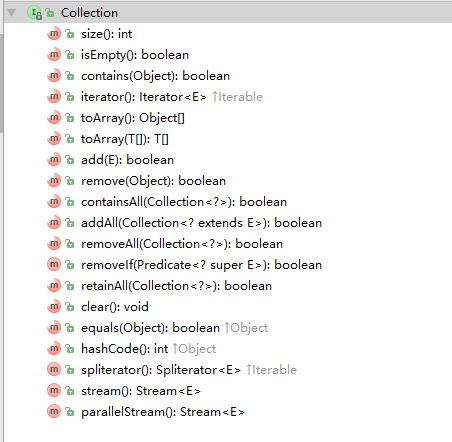


向链表中插入或从链表中删除一项的操作不需要移动很多的项,而设计常数个节点链的改变.

让每一个节点持有一个指向它在表中的前驱节点的链,如图3-4所示,我们称之为双向链表(double linked list).

## 3.3 Java Collections API

### 3.3.1 Collections接口



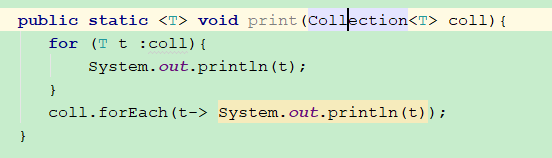
size()返回集合中的项数;

isEmpty()放回true当且仅当集合的大小为0;

contains()返回true 当x在集合中 注意这个接口并不规定集合如何决定x是否属于该集合--这要由实现该接口的具体类来确定;

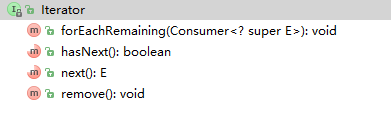
add remove 从集合中添加和删除x,如果成功则返回true,如果失败则返回false.

Collection接口扩展了Iterable接口.实现Iterable接口的那些类都可以拥有增强for循环,该循环用于这些类智商以观察它们的所有项,并可以使用forEach函数接口lamda表达式.



### 3.3.2 Iterator接口

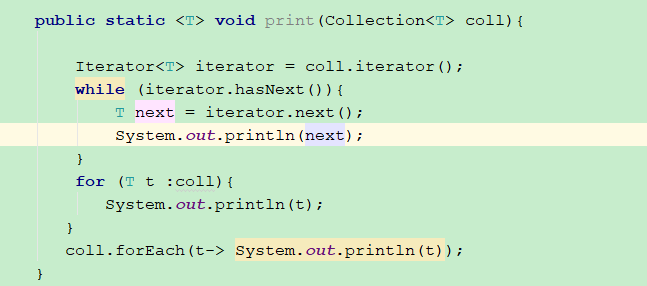
实现Iterator接口的集合必须提供一个称谓iterator()的方法,该方法返回一个Iterator类型的对象.该Iterator是一个在java.util包中定义的接口



Iterator接口的思路是,通过iterator方法,每个集合均可创建并返回一个实现Iterator接口的对象,并将当前位置的概念在对象内部存储下来.

next()给集合的下一项(尚未见到的)

hasNext()用来判断是否存在下一项

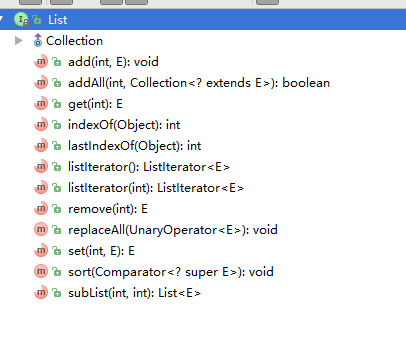


remove()方法可以删除由next()最新返回的项(此后,我们不能再调用remove,直到对next()再次调用以后).虽然Collection接口也包含一个remove方法,但是使用Iterator方法可能会有更多的优点.

1 Collection的remove()方法必须先找出要被删除的项,如果直到要删除的项的准确位置,那么删除它的开销可能要小很多,潜藏着更高的效率

2当直接使用Iterator时,重要的是要记住一个基准法则:如果对正在被迭代的集合进行结构上的改变,(即对该集合进行add remove clear操作),那么迭代器就不再合法(并且其后使用该迭代器将会有ConcurrentModificationException异常抛出).避免也许一个新的项正好被插入该项的前面这样的讨厌情形,有必要记住次法则.然后如果迭代器调用了自己的remove()方法,那么这个迭代器仍是合法的.

### 3.3.3 List接口 ArrayList类和LinkedList类



get和set使得用户可以访问或改变通过由位置索引idx给定的表中指定位置上的项

索引0位于表的前端,索引size()-1代表表中的最后一项

索引size()则表示新添加的项可以被放置的位置

add()使得在位置idx处植入一个新的项,并把其后的项向后推移一个位置

remove(int)可以删除指定位置上的项

listIterator() 它将产生比通常认为的还要复杂的迭代器

List ADT有两种流行的实现方式:

ArrayList类提供了List ADT的一种可增长数组的实现

优点在于对get和set的调用花费常数时间,时间复杂度为O(1)

缺点是新项的插入和现有项的删除代价昂贵,除非变动实在ArraList的末端进行

LinkedList类则提供了List ADT的双链表实现

优点在于,新项的插入和现有项的删除均开销很小,这里假设变动项是一致的这意味着在表的前端进行删除和添加都是常数时间的操作,由此LinkedList更提供了addFirst和removeFirst addLast、 removeLast、 getFirst、 getLast等有效添加 删除和访问两端的项

缺点是它不容易做索引,因此对get的调用是昂贵的,除非调用非常接近表的端点

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

makeList1()时间复杂度为O(n),忽略ArrayList偶尔进行的扩展

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

LinkedList运行makeList2的时间复杂度为O(n),

ArrayList运行的时间复杂度则为O(N2)

**public static int** sum(List<Integer> list,**int** n){  
 **int** total = 0;  
 **for** (**int** i = 0; i < n;i++){  
 total += list.get(i);  
 }  
 **return** total;  
}

这里ArrayList的时间复杂度是O(N),但对于LinkedList来说,其运行的时间则是O(N2),因为在LinkedList中,对get的调用为O(N)操作.可是,要是使用增强的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都是低效的.对于remove(T)和contains(T)的调用均花费线性时间(LinkedList效率稍高,因为remove特定项为O(1),而ArrayList特动项为O(N));

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,ArraList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的扩展.再有,trimToSize可以在所有的ArrayList添加操作以后使用以避免浪费空间

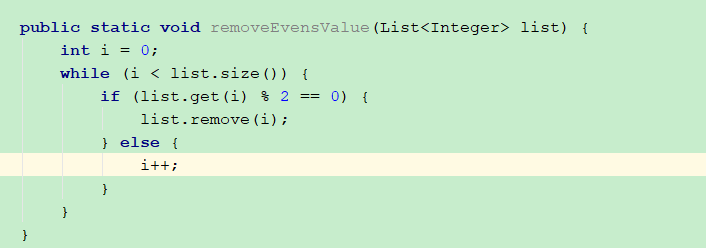
### 3.3.4 例: remove()方法对LinkedList类的使用

问题:将一个表中所有具有偶数值得项删除

方法1:构造一个包含所有的奇数的新表

方法2:避免拷贝,直接删除偶数

对于ArrayList这几乎是一个失败的策略.因为ArrayList几乎是对任何地方进行删除都是昂贵的操作.不过LinkedList中却存在某种希望.



上图显示了第一种想法,在ArrayList上,我们知道remove()的时间复杂度为O(N),所以该程序的时间复杂度是O(N2),

LinkedList暴露了两个问题get时间复杂度为O(N),且remove(index)时间复杂度也为O(N),

所以该程序的时间复杂度也为O(N2);

图3-11显示矫正该问题的一种思路.

**public static void** removeEvensValue2(List<Integer> list){  
 list.forEach(x->{  
 **if** (x %2 == 0){  
 list.remove(x);  
 }  
 });  
}

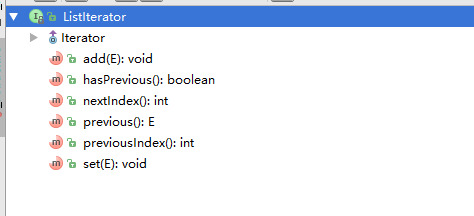
这个程序使用了迭代器遍历,这样可以便面get的地效率,但是依然没有避免remove()地效率的问题,所以他的时间复杂度仍然是O(N2)级别;更糟糕的是,因为其中一项被删除,迭代器已经非法,会抛出ConcurrentModificationException异常.

下图指出了一种成功的想法:

**public static void** removeEvensValue3(List<Integer> list) {  
 Iterator<Integer> iterator = list.iterator();  
 **while** (iterator.hasNext()) {  
 **if** (iterator.next() % 2 == 0) {  
 iterator.remove();  
 }  
 }  
}

迭代器的remove()方法首先维护了当前的迭代器合法,避免了异常,且迭代器的remove()方法用LinkedList实现的时间复杂度是O(1)的,即常数复杂度.所以整个程序的时间复杂度为O(N),而ArrayList迭代器的remove()的时间复杂度仍然是O(N).

3.3.5 关于ListIterator接口



ListIterator扩展了List的Iterator的功能.

1 方法previous()和hasPrevious()使得表从后向前遍历得以完成.

2 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作是在对next()和previous()所给出的项的之间而抽象出来的.

3 add对于ArrayList仍然是一项O(N)的昂贵操作,但是对于LinkedList则是一种常数时间操作

4 set改变被迭代器看到的最后一个值,对于LinkedList很方便,实现了O(1)的操作

## 3.4 ArrayList类的实现

1 MyArrayList将保持基础数组,数组的容量,以及存储在MyArrayList中的当前项数

2 MyArrayList将提供一种机制以改变数组的容量.通过获得一个新数组,将老数组拷贝到新数组中来改变数组的容量,允许虚拟机回收老数组

3 MyArrayList将提供set和get实现

4 MyArrayList将提供基本的程序,如size() isEmpty() clear(),他们是典型的单行程序;还提供remove(),以及两种不同版本的add().如果数组大小和容量相同,那么这两个add方法将增加容量

5 MyArrayList将提供一个实现Iterator()接口的类.这个类将存储迭代序列中的下一项的下标,并提供next() hasNext() 和remove()等方法的实现,MyArrayList的迭代器直接返回Iterator接口的该类的新构造的实例

### 3.4.1 基本类

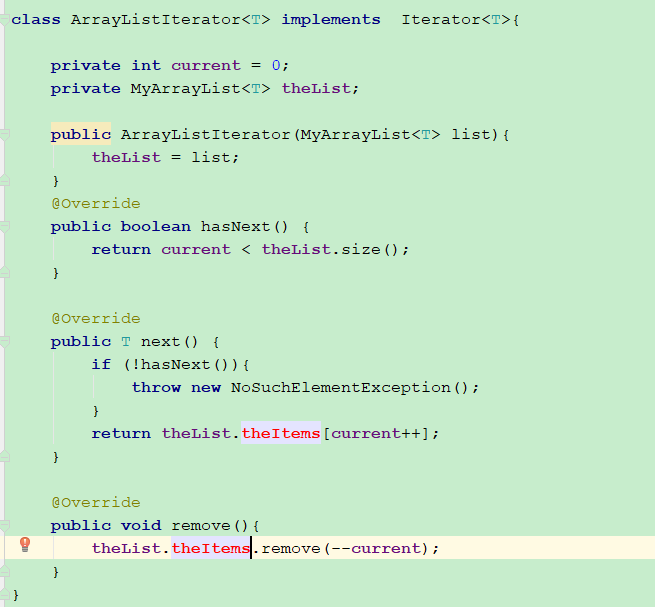
**public class** MyArrayList<T> **implements** Iterable<T>{  
  
 **private static final int *DEFAULT\_CAPACITY*** = 10;  
 **private int theSize**;  
 **private** T [] **theItems**;  
  
 **public** MyArrayList(){  
 clear();  
 }  
  
 **public void** clear(){  
 **theSize** = 0;  
 ensureCapacity(***DEFAULT\_CAPACITY***);  
 }  
  
 **public int** size(){  
 **return theSize**;  
 }  
  
 **public** T get(**int** index){  
 **if** (index < 0 || index >= size()){  
 **throw new** ArrayIndexOutOfBoundsException();  
 }**else** {  
 **return theItems**[index];  
 }  
 }  
  
 **public** T set(**int** index,T newValue){  
 **if** (index < 0 || index >= size()){  
 **throw new** ArrayIndexOutOfBoundsException();  
 }  
 T old = **theItems**[index];  
 **theItems**[index]=newValue;  
 **return** old;  
 }  
  
 **public void** ensureCapacity(**int** newCapacity){  
 **if** (newCapacity < **theSize**){  
 **return**;  
 }  
 T [] old = **theItems**;  
 **theItems** = (T[])**new** Object[newCapacity];  
 **for** (**int** i = 0; i <size();i++){  
 **theItems**[i] = old[i];  
 }  
 }  
  
  
 **public boolean** isEmpty(){  
 **return** size() == 0;  
 }  
  
 **public void** trimToSize(){  
 ensureCapacity(size());  
 }  
  
 **public boolean** add(T t){  
 add(size(),t);  
 **return true**;  
 }  
 **public void** add(**int** index,T t){  
 **if** (**theItems**.**length** == size()){  
 ensureCapacity(size() \* 2 + 1);  
 }  
 **for** (**int** i = **theSize**;i>index;i--){  
 **theItems**[i] = **theItems** [i-1];  
 }  
 **theItems** [index] = t;  
 **theSize** ++;  
 }  
  
 **public** T remove(**int** index){  
 T removedItem = **theItems**[index];  
 **for** (**int** i = index;i < size() - 1;i++){  
 **theItems**[i] = **theItems** [i+1];  
 }  
 **theSize** -- ;  
 **return** removedItem;  
 }  
  
 @Override  
 **public** Iterator<T> iterator(){  
 **return new** ArrayListIterator();  
 }  
  
 **private class** ArrayListIterator **implements** Iterator<T>{  
  
 **private int current** = 0;  
 @Override  
 **public boolean** hasNext() {  
 **return current** < size();  
 }  
  
 @Override  
 **public** T next() {  
 **if** (!hasNext()){  
 **throw new** NoSuchElementException();  
 }  
 **return theItems**[**current**++];  
 }  
  
 @Override  
 **public void** remove(){  
 MyArrayList.**this**.remove(--**current**);  
 }  
 }  
}

### 3.4.2 迭代器 Java嵌套类和内部类

内部类 inner class



迭代器1号版本(但不能使用):迭代器是一个顶级类并存储当前位置.它不能使用size()方法和theItems,因为它不是ArrayListIterator类的一部分.所以此程序根本没有意义



迭代器2,几乎能用了,它存储了当前位置以及一个连接到MyArrayList的引用,.它不能使用是因为theItems是私有变量



迭代器3版本可以使用:迭代器是一个嵌套类并存储当前位置和一个连接到MyArrayList的引用.它能够使用是因为该嵌套类被认为是MyArrayList类的一部分.

嵌套类(nested class) 外部类(outer class),我们使用static表示它是嵌套的;若无static,将得到一个内部类(inner class).

嵌套类的问题在于,在我们的原始设计中,当编写theItems而不引用其所在的MyArrayList的时候,代码看起来还可以,但确实无效的,因为编译器不可能计算出去哪个MyArrayList在被引用.

当声明一个内部类时,编译器则添加对外部类对象的而一个隐式引用,该对象为运用内部类对象的构造.如果外部类的名字是Outer,则隐式引用就是Outer.this.因此,theList可以被省略



迭代器4能够使用,迭代器是一个内部类并存储当前位置和一个连接到MyArrayList的隐式引用

## 3.5 LinkedList类的实现

1 MyLinkedList类本身,它包含两端的链 表的大小以及一些方法

2 Node类,它可能是一个私有的嵌套类.一个节点包含数据以及到前一个节点的链和到下一个节点的链

3 LinkedListIterator类,该类抽象了位置的概念,是一个私有类,并实现了接口Iterator,它提供了next() hasNext() remove()的实现

标记节点

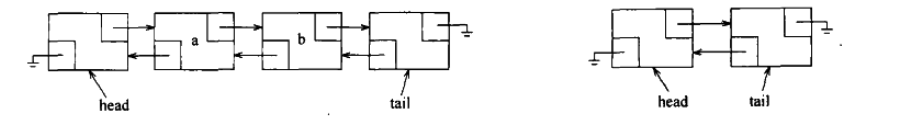
前端的节点叫做头节点 header node

末端的几点叫做尾节点 tail node

优势在于通过排除许多特殊情形极大优化了编码

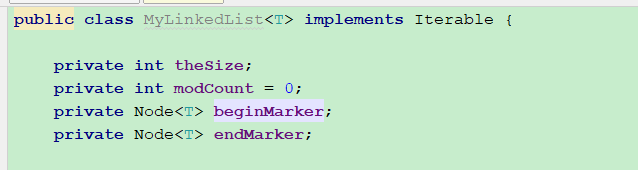
1不必通过轮询找到第一个节点的链

2 避免删除算法访问前面的节点



Node类

Node类的所有成员变量都是public修饰,而Node类是私有的,因此在Node类中的那些数据成员的可见性都是公用的;那些MyLinkedList的方法都尅见到Node的所有数据成员,而MyLinkedList外面的类则见不到Node类.



MyLinkedList的成员变量,

1头节点和尾节点beginMarker和endMarker

2 数据成员theSize的大小,从而size方法可以以常数时间实现

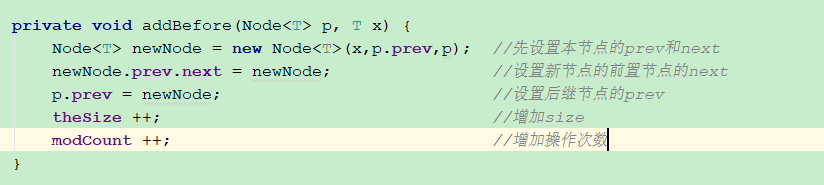
3 modCount是一个附加的数据域,用来帮助迭代器检测集合中的变化



modCount表示自从构造以来对链表所做的改变的次数,每次add和remove的调用都会更新modCount,每次对一个迭代器方法next()或者remove()的用用都会用该链表内的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

clear()方法由构造方法调用.它创建并连接头节点和尾节点,然后设置大小为0.

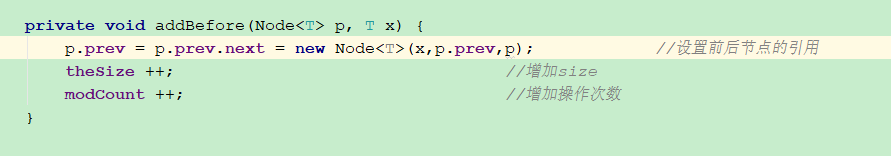
addBefore()方法解释了一个包含x的新节点是如何被拼接到由p引用的一个节点和其前置节点之间的

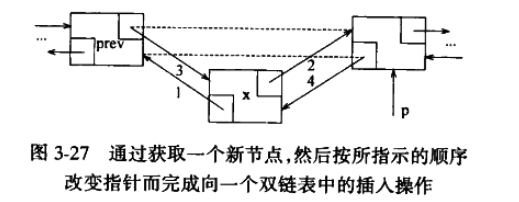


也可以将第2步和第三步合并

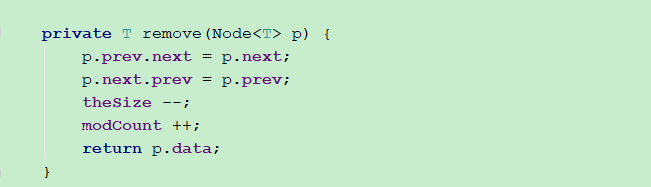
p.**prev** = p.**prev**.**next** = newNode; *//设置前后节点的引用*

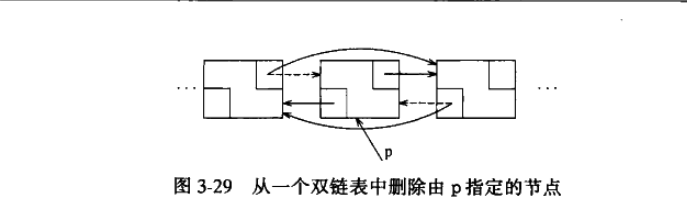
也可以将1 2 3都合并





删除一个节点的逻辑过程,如果p引用正在被删除的节点,那么该节点被断开连接和可以被虚拟机回收之前只有两个链的改动





MyLinkedListIterator具有类似于MyArrayList的逻辑

@Override  
**public boolean** hasNext() {  
 **return current** != **endMarker**;  
}  
  
@Override  
**public** T next() {  
 **if** (**modCount** != **expectedModcount**){  
 **throw new** ConcurrentModificationException();  
 }  
 **if** (!hasNext()){  
 **throw new** NoSuchElementException();  
 }  
 T data = **current**.**data**;  
 **current** = **current**.**next**;  
 **okToRemove** = **true**;  
 **return** data;  
}

1合并了重要的错误检测.

2该迭代去保留了一个当前位置cuurent表示包含由调用next所返回的项的节点

3注意current被定于endMarker时,对next的调用是非法的

4 为了检测在迭代器件集合被修改的情况,迭代器被构造时的链表的modCount存储在expectedModCount中,如果next已经被执行而没有其后的remove,则布尔值okToremove为true

5 next()方法在获得将要返回的节点的值后,向后推进current

@Override  
**public void** remove(){  
  
 **if** (**modCount** != **expectedModcount**){  
 **throw new** ConcurrentModificationException();  
 }  
 **if** (!**okToRemove**){  
 **throw new** IllegalStateException();  
 }  
 MyLinkedList.**this**.remove(**current**.**prev**);  
 **okToRemove** = **false**;  
 **expectedModcount** ++;  
}

6 remove()方法的主要逻辑是错误检测,current保持不变.因为current正在观察的节点不受前面节点被删除的影响(而MyArraListIterator中,项被移动,要求更新current)

## 3.6 栈ADT

### 3.6.1 栈模型

**栈**(Stack)是限制插入和删除只能在一个位置上进行的表,该位置是表的末端,叫做栈的顶(top).

对栈的基本操作有

push(进栈),相当于插入

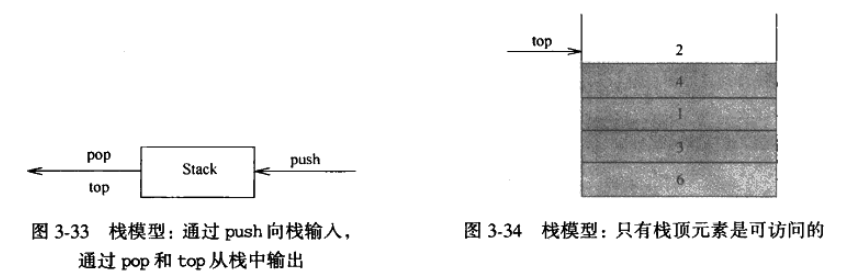
pop(出栈),删除最后插入的元素

最后插入的元素可以通过top方法在执行pop之前进行考察

对空栈进行的pop或者top一般被认为是栈ADT中的一个错误

当运行push时空间用尽是一个实现限制,但不是ADT错误

栈有时也叫做LIFO(last input first out)后进先出表



3-33表示的模型,push是输入操作,而pop和top是输出操作,普通的清空栈和判断是否是空栈的测试都是栈的操作指令系统的一部分,但是,我们对栈所能做的,基本上也就是push和pop操作

3-34表示在进行若干操作以后的一个抽象的栈,一般的模型是,存在某个元素位于栈顶,而该元素也是唯一可见的元素

### 3.6.2 栈的实现

由于栈是一个表,因此任何实现表的方法都能实现栈.

显然ArrayList和LinkedList都支持栈操作;99%的时间他们都是最合理的选择.

偶尔设计一种特殊目的的实现可能会更快(例如被放入栈顶的项属于基本类型)

因为栈操作是常数时间操作,所以除非在非常独特的环境下,这是不可能产生任何明显的改进的.

对于这些特殊的实际,我们将给出两个流行的实现方法,一种方法是使用链式结构,而另一种方法则使用数组,二者均简化了在ArrayList和LinkedList中的逻辑,因此我们不提供代码

**栈的链表实现**

单链表,,通过在表的顶端插入来实现push,通过删除表顶端元素来实现pop,top操作只是考察表顶端元素并返回它的值.有时候pop操作和top操作合二为一

**栈的数组实现**

由于模仿ArrayList的add操作,因此数组实现方法非常简单.与每个栈相关联的操作都是theArray和topOfStack,对于空栈它是-1(这就是空栈初始化的做法).

### 3.6.3 应用

**平衡符号**

编译器检查程序的语法错误

做一个空栈.读入字符直到文件结尾.如果字符是一个开放符号,则将其推入栈顶.如果字符是一个封闭符号且空栈,则报错.否则,将栈元素弹出.如果弹出的符号不是对应的开放符号,则报错.在文件结尾,如果栈非空则报错.

**后缀表达式**

计算一个后缀表达式花费的时间是**O**(N),因为每个元素的处理都是由一些栈操作组成从而花费常数时间.当一个表达式以后缀记号给出时,没必要知道任何优先的规则,这是一个明显的优点

**中缀到后缀的转换**

a+b\*c+(d\*e+f)\*g

转换成后缀表达式

abc\*+de\*f+g\*+

1 当读到一个操作数的时候,立即把它放到输出中.

2 操作符不立即输出,从而必须先存在某个地方,正确的做法是将已经见到过但尚未输出中的操作符推入栈中

3 当遇到左圆括号时我们也要将其推入栈中

4 当见到一个右括号,那么就将栈元素弹出,将弹出的符号写出直至遇到一个(对应的)左括号,但是这个左括号只弹出并不输出

5 如果我们见到任何其他的符号(,+,\*,(),那么我们从栈中弹出栈元素直到发现优先级更低的的元素为止.有一个例外:除非是在处理一个)的时候,否则我们绝不从栈中移走(.对于这种操作,+的优先级最低,而(的优先级最高.当从栈弹出元素的工作完成后,我们再将操作符压入栈中.

6 最后,如果读到输入的末尾,我们将栈元素弹出直到该栈变成空栈,将符号写到输出中

## 3.7 队列ADT

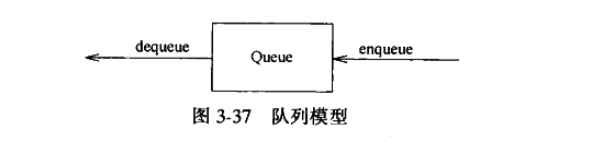
像栈一样,队列也是表,它是在一段进行插入而删除则在另一端进行

### 3.7.1 队列模型

队列的基本操作

enqueue:在表的末端(队尾)插入一个元素

dequeue:删除并返回在表的开头(表头)的元素



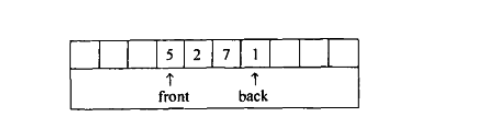
### 3.7.2 队列的数组实现

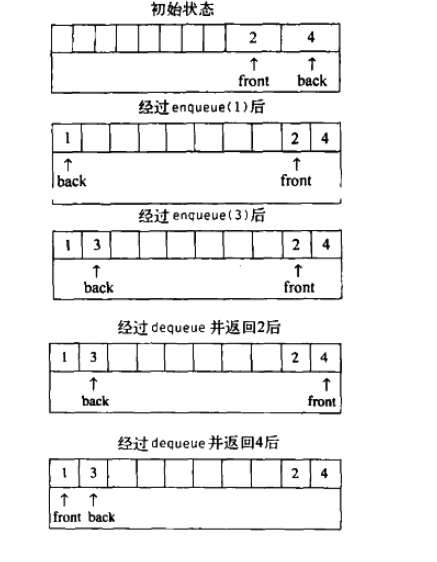
链表和数组的实现都给出快速的0(1)运行时间,队列的链表实现是简单直接的

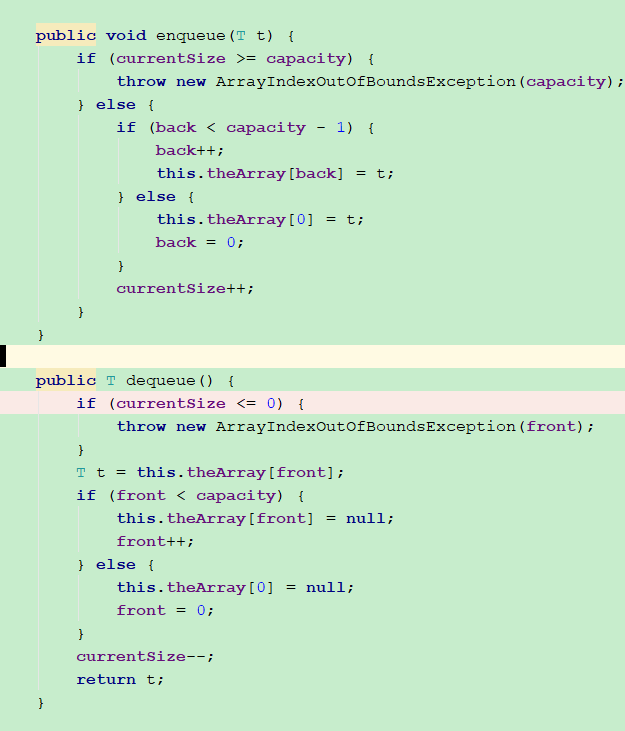
**数组实现:**

对于每一个队列数据结构,我们保留一个数组theArray以及位置front和back,他们代表队列的两端,我们还要记录实际存在于队列中的元素的个数currentSize.

下面是处于一个中间状态的队列







# 树

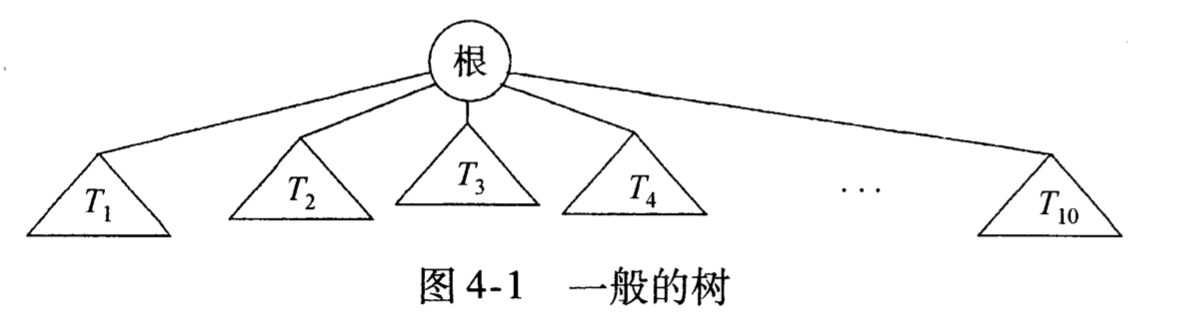
对于大量的输入数据,链表的线性访问时间太慢,不宜使用.本章讨论一种简单的数据结构,其大部分操作的运行时间为O(logN).

二叉查找树(binary search tree).二叉查找树是两种库集合类TreeSet和TreeMap实现的基础.

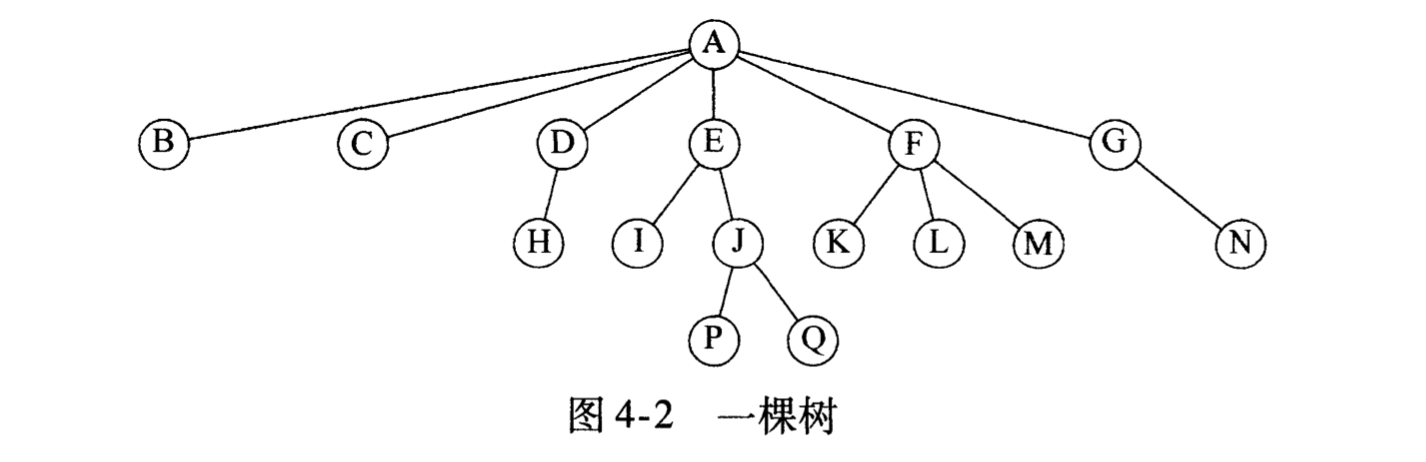
## 4.1 预备知识

一棵树(tree)是一些节点的集合.这个集合可以是空集;若不是空集,则树由称作根(root)的节点r及0个或多个非空的子树T1,T2,...,Tk组成,这些子树中每一颗的根都被来自根r的一条有向的边(edge)所连接

每一棵子树的根叫做根 r 的**儿子**(child)，而 r 是每一棵子树的根的**父亲**(parent)



一棵树是 N 个节点和 N-1条边的集合，其中的一个节点叫做根。



没有儿子的节点称为**树叶**（leaf）

具有相同父亲的节点为**兄弟**（siblings）

**祖父**(grandparent)

**孙子**(grandchild)

从节点n1到nk的**路径**(path)定义为节点n1,n2,...nk的一个序列,使得对于1≤i≤k的节点,ni是ni+1的父亲.

这条路径的**长度**(length)为该路径上的边的条数,即k-1,注意,从一棵树的根到每个节点,恰好存在一条路径

**深度**(depth)为从跟到ni的唯一路径的长

**高**(height)是从ni到一片树叶的最长路径的长

因此所有树叶的高度都是0,一棵树的高等于它的根的高

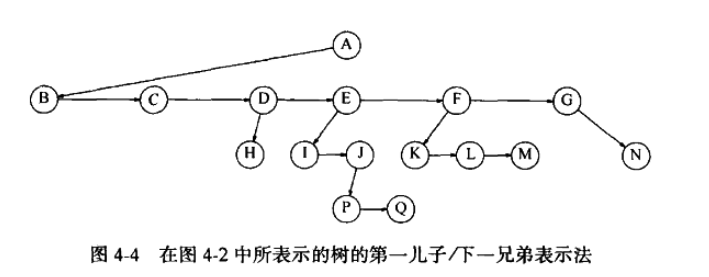
如果存在从n1到n2的一条路径,那么n1是n2的以为**祖先**(ancestor)而n2是n1的一个**后裔**(descendant).如果n1≠n2,那么n1是n2的**真祖先**(proper ancestor),

而n2是n1的**真后裔**(proper descendant)

### 4.1.1 树的实现

实现树的一种方法可以是在每一个节点除数据外还要有一些链,使得该节点的每一个儿子都有一个链指向它.然而,由于每一个节点的儿子数量可以变化很大并且实现不知道,因此在数据结构中建立到各儿子节点直接的链接是不可行的,因为这样会产生太多浪费的空间.

实际上解决方法很简单:将每个节点的所有儿子都放在树节点的链表中.



class TreeNode{

Object element;

TreeNode firstChild;

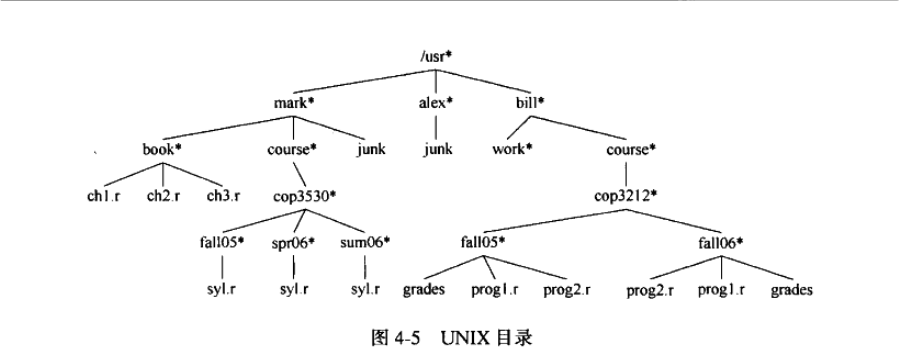
TreeNode nextSibings;

}

途中向下的箭头是指向firstChild的链

水平箭头是指向nextSibling的链

### 4.1.2 树的遍历及应用



**全路径名**(pathname)

private void listAll(int depth){

printName(depth); //Print the name of the object

if(isDirectory()){

for each file c in this directory(for each child)

c.lishAll(depth+1);

}

}

public void listAll(){

listAll(0);

}

算法逻辑:

文件对象的名字和适当的跳隔次数一起打印出来.

如果文件是一个目录,那么以递归的方式一个一个处理它所有的儿子.

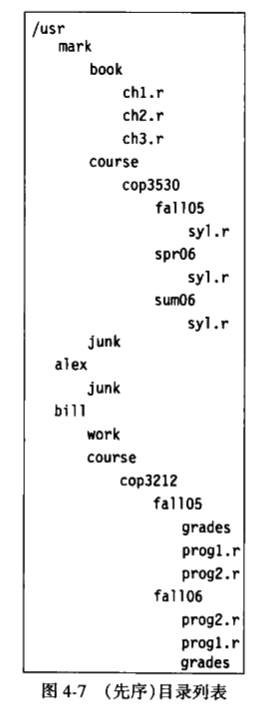
这些儿子均处在下一层的深度上,因此需要缩进一个附加空间

这种遍历策略叫做**先序遍历**(pre order traversal ).在先序遍历中,对节点的处理工作室在它的诸儿子节点被处理之前进行的.

该程序运行时,第一行对每个节点恰好执行一次,因为每个名字只输出一次.由于第一个行对每个节点最多执行一次,因此第二行也必然对每个节点执行一次.

不仅如此,对于每个节点的每一个子节点,第四行最多只能被执行一次.但是,儿子节点的个数恰好比节点的个数少1.

最后,第四行每执行一次,,for循环就迭代一次,每当循环结束时再加上一次.因此在每个节点上的总的工作量是常数,如果有N个文件名需要输出,则运行的时间就是O(N)



**后续遍历**(post order traversal).

public int size(){

int totalSize = sizeOfThisFile();

if(isDirectory()){

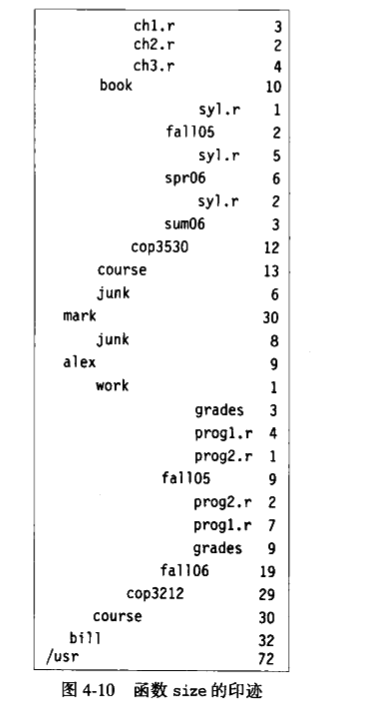
for each file c in this directory(for each child)

totalSize += c.size();

}

return totalSize;

}

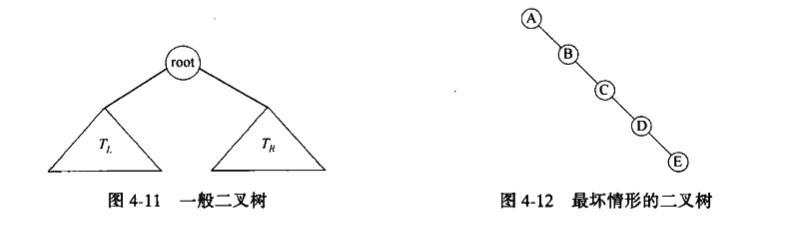


## 4.2 二叉树

**二叉树**(binary tree)是一颗树,其中每个节点都不能有多于两个的儿子

二叉树的一个性质是一颗平均二叉树的深度要比节点个数小得多,这个性质有时很重要

分析表明,其平均深度为O(),而对于特殊的二叉树,即二叉查找树(binary search tree),其深度的平均值是O(log N).不幸的是,真如图4-12中的例子所示,这个深度是可以大到N-1的



### 4.2.1 实现