## 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)

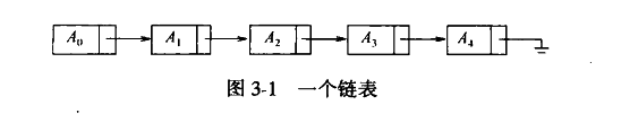
**int** [] arr = **new int**[10];  
**int** [] newArr = **new int**[arr.**length** \* 2];  
**for**(**int** i = 0; i < arr.**length** ; i ++){  
 newArr [i] = arr[i];  
}  
arr = newArr;

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

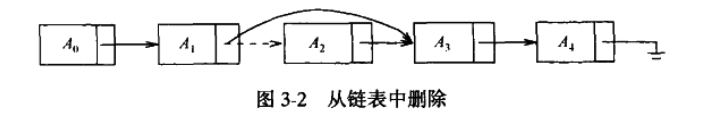
为了避免插入和删除的线性开销,我们需要保证表可以不连续存储,否则表的每个部分都可能需要整体移动.图3-1指出链表的一般想法



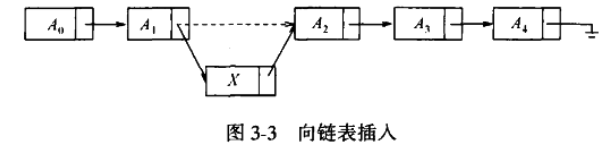
链表由一些列节点组成,这些节点不必再内存中相连.每个节点均含有表元素和到包含该元素后继几点的链(link),我们称之为next链.最后一个单元的next链引用为null.

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.

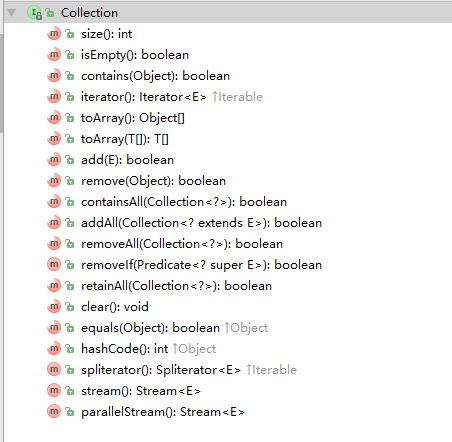


向链表中插入或从链表中删除一项的操作不需要移动很多的项,而设计常数个节点链的改变.

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



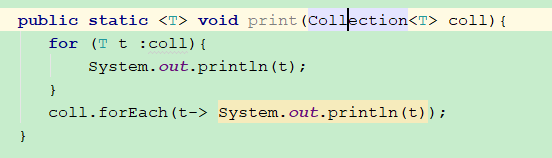
size()返回集合中的项数;

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;

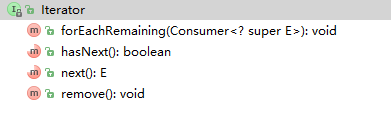
add remove 从集合中添加和删除x,如果成功则返回true,如果失败则返回false.

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

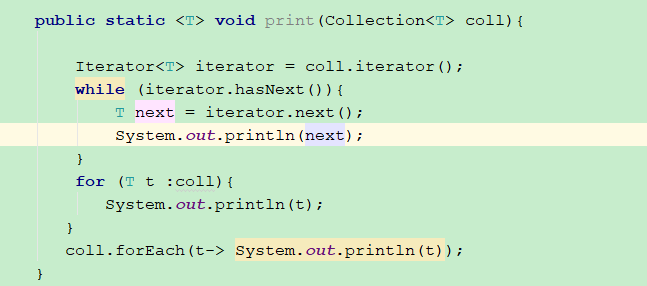
实现Iterator接口的集合必须提供一个称谓iterator()的方法,该方法返回一个Iterator类型的对象.该Iterator是一个在java.util包中定义的接口



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

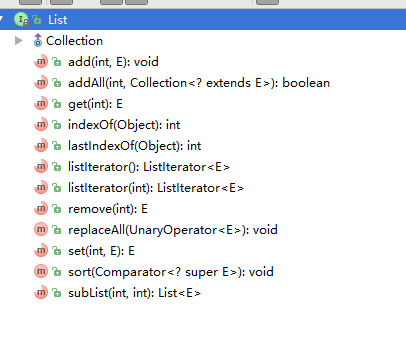


remove()方法可以删除由next()最新返回的项(此后,我们不能再调用remove,直到对next()再次调用以后).虽然Collection接口也包含一个remove方法,但是使用Iterator方法可能会有更多的优点.

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

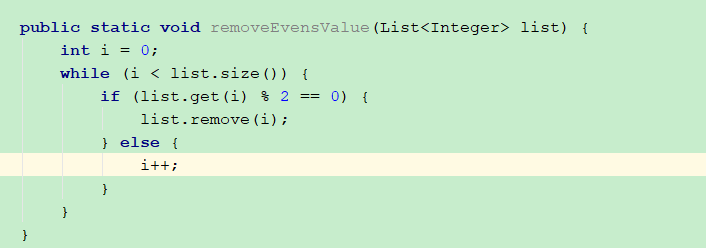
### 3.3.4 例: remove()方法对LinkedList类的使用

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