

算法与数据结构

Java描述



目录

[1．算法复杂度 2](#_Toc99966576)

[1 算法的要素 2](#_Toc99966577)

[2 算法的复杂度 2](#_Toc99966578)

[3 递归 2](#_Toc99966579)

[2. 栈与队列 5](#_Toc99966580)

[1栈 LIFO 5](#_Toc99966581)

[2 队列 FIFO 8](#_Toc99966582)

[3. 向量、列表与序列 15](#_Toc99966583)

[1 向量 15](#_Toc99966584)

# 1．算法复杂度

## 1 算法的要素

输入、输出、确定性、可行性、有穷性

## 2 算法的复杂度

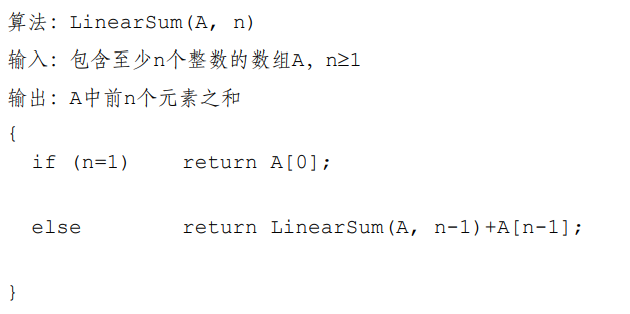
时间复杂度（主要讨论）

空间复杂度

## 3 递归

1. 线性递归

数组元素的递归求和

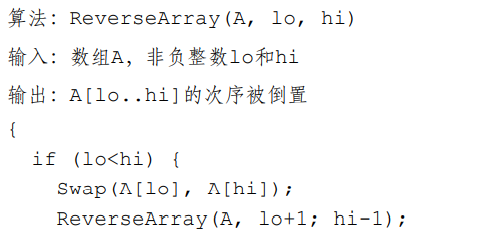


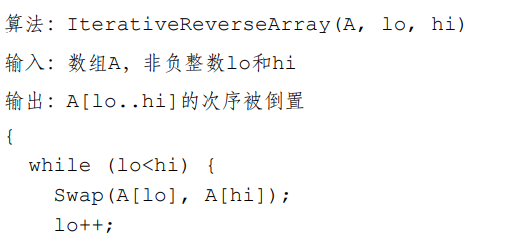
递归基

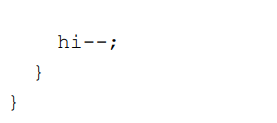
通过递归实现数组倒置 (两种情况: lo=hi / lo > hi)

只有当该方法的任何一次执行（当然，除平凡的递归基外）都以这一递归调用结束时，才属于**尾递归**

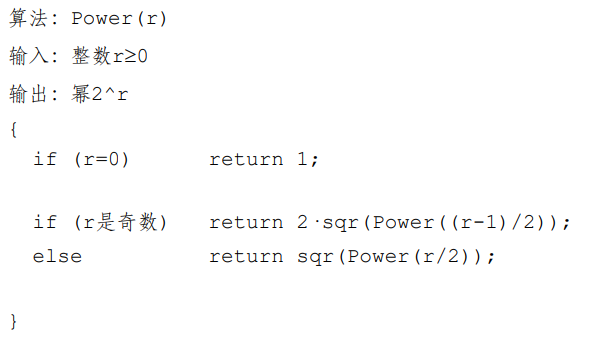
利用递归可以编写出简洁而优美的算法。然而，这些优点并非没有代价，为此 计算机通常需要使用更多的空间、需要花费额外的时间以跟踪递归调用的过程。在对运行速度要求极高、对存储空间需要精打细算时，往往需要将算法的递归版本改写成非递归的版本。





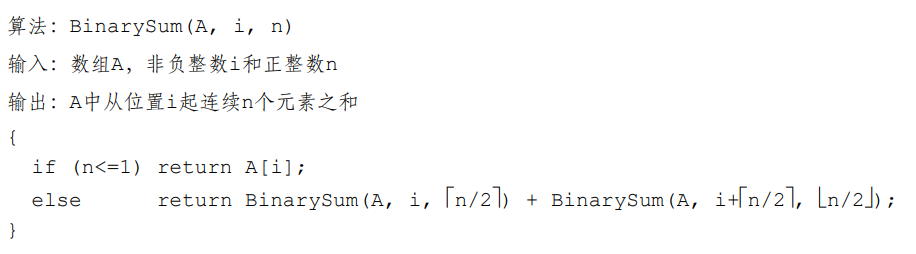


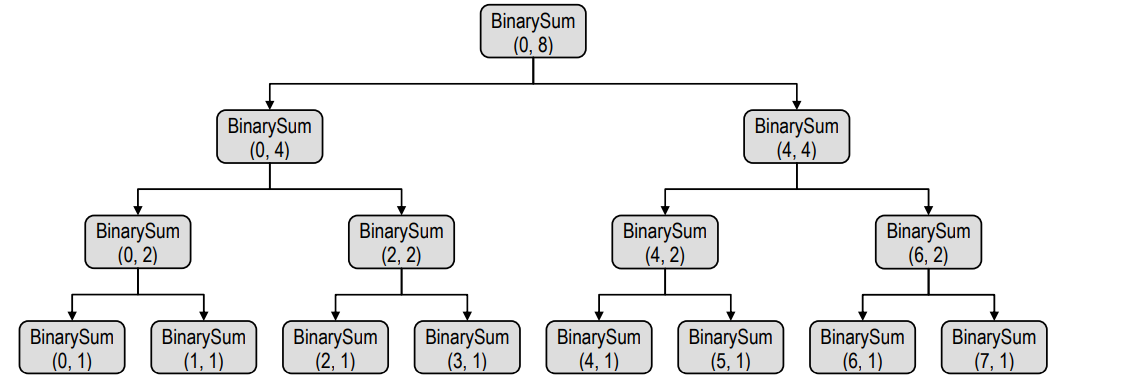
基于线性递归的幂计算



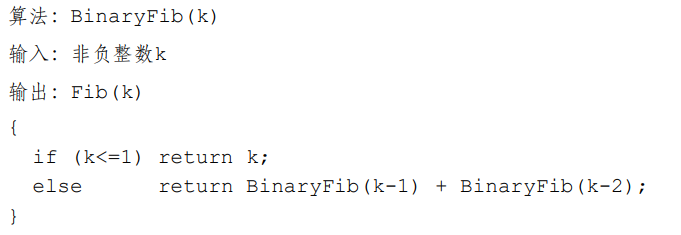
1. 二分递归

二分递归对数组元素求和



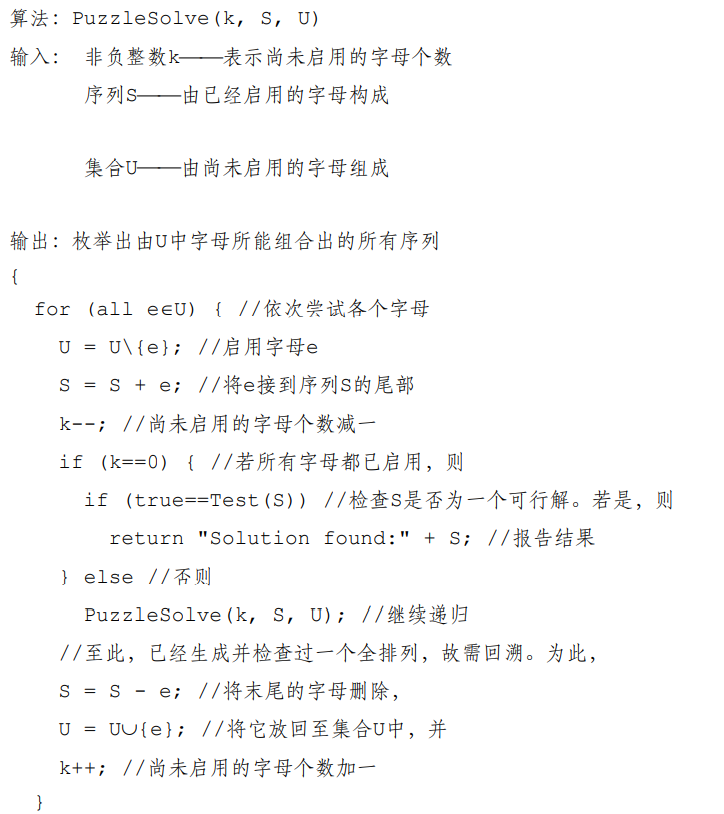


通过二分递归计算Fibonacci数(O(2^n))



通过线性递归计算Fibonacci数(O(n))

1. 多分支递归



# 2. 栈与队列

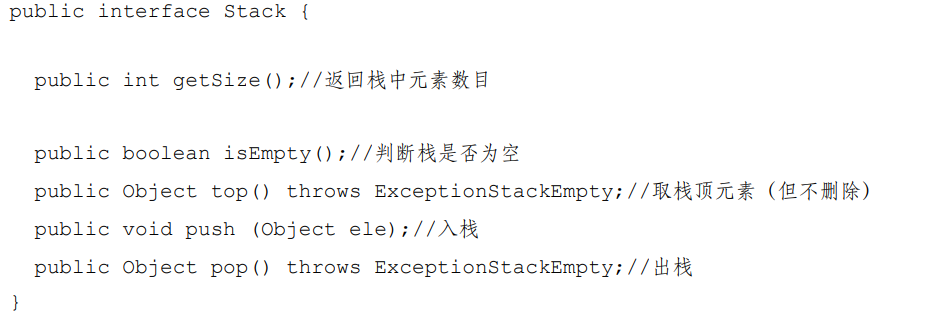
## 1栈 LIFO

java.util.Stack

push()、pop()、peek() （功能等价于 top()）、getSize()以及 empty()（功能等价于 isEmpty()）

在遇到空栈时，方法 pop() 和 peek()都会报意外错 ExceptionStackEmpty

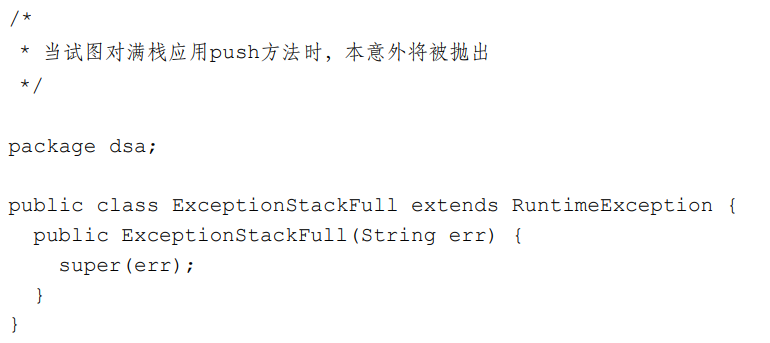
**Interface**



**Implement**

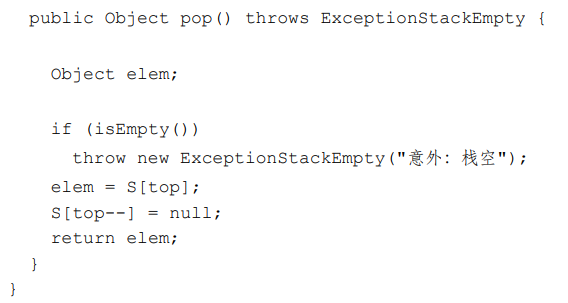
**数组实现栈 O(n)**

由于数组的容量固定，因此有可能会出现栈溢出的情况。为此，我们专门 引入一个新的意外ExceptionStackFull，这一例外并非栈ADT本身的要求，而只是针对数组实现而设置的。



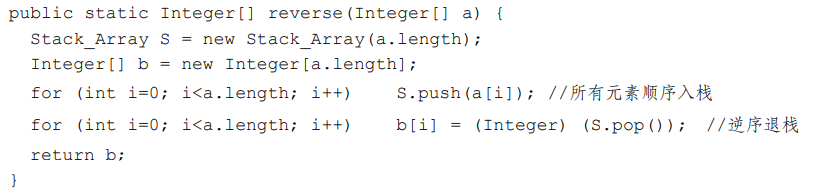




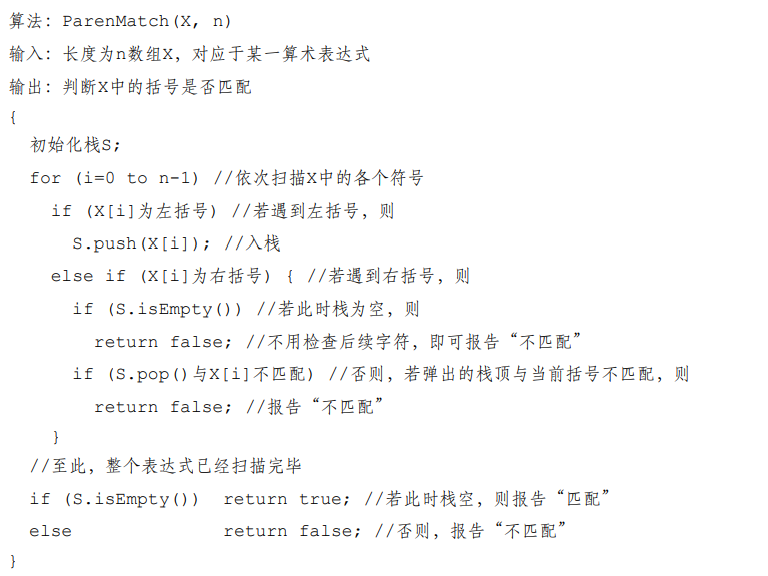


基于数组的 Stack 接口实现的缺陷 1）栈溢出 2）存储空间的浪费

* 利用栈实现对数组的倒置

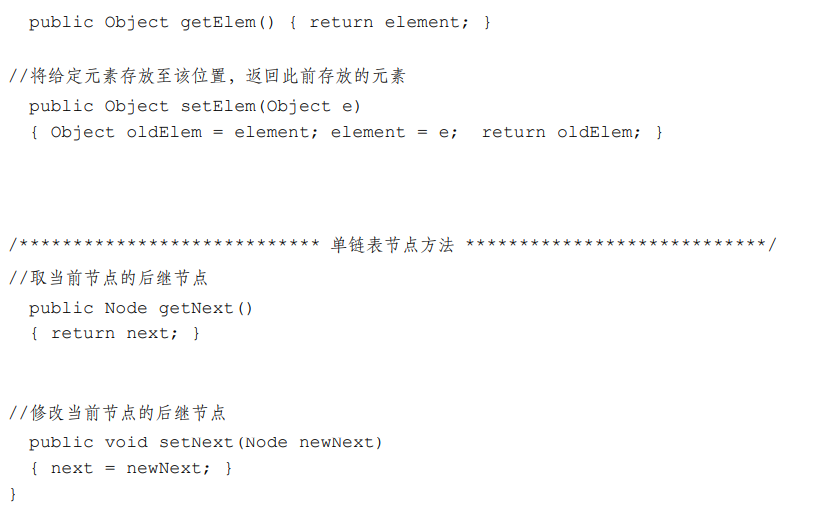


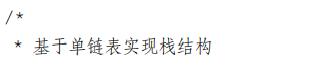
* 括号匹配算法

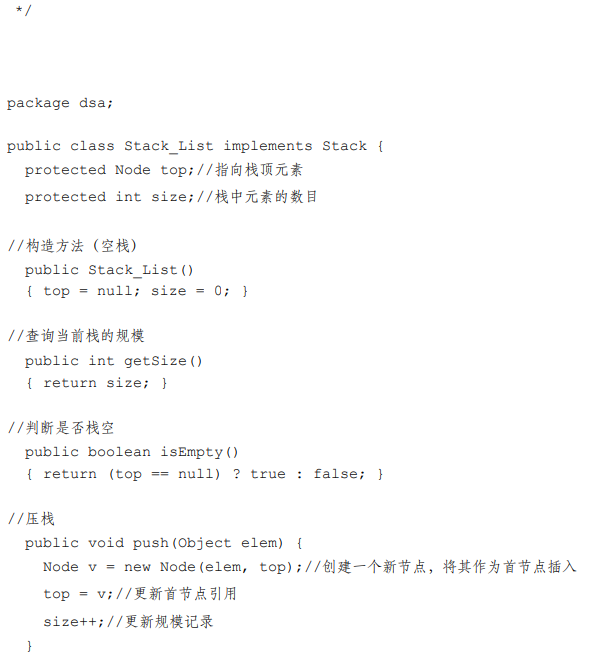


**单链表实现栈**



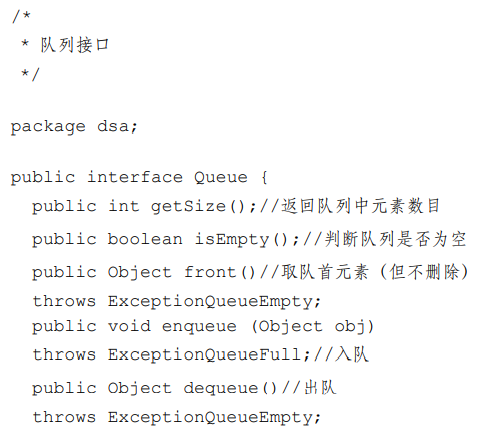
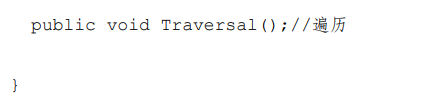






## 2 队列 FIFO

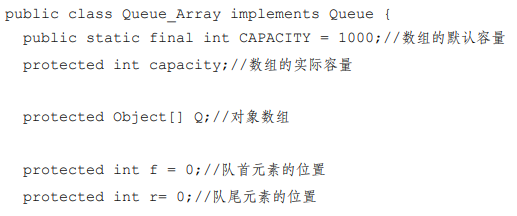
**Interface**

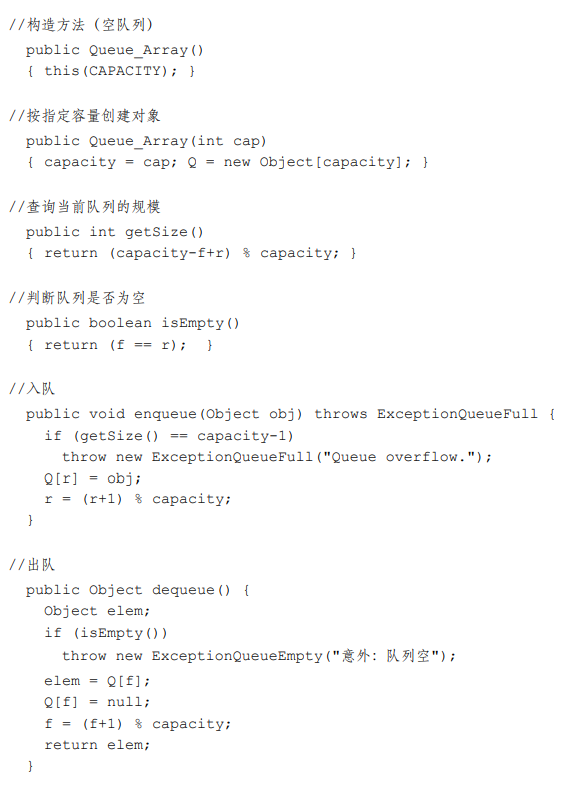
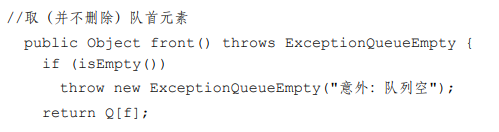


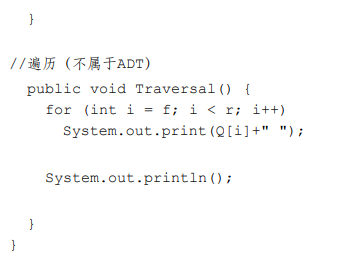
在队列的生命期内，f 和 r 始终 在单调增加。因此，若队列数组的容量为 N，则在经过 N 次入队操作后，r 所指向的单元必然超出 数组的范围；在经过 N 次出队操作后，f 所指向的单元也会出现类似的问题。

**循环数组实现队列 O(n)**

在每次 f 或 r 加1后，都要以数组的长度做取模运算



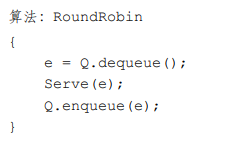




队空 f=r

队满 f=r

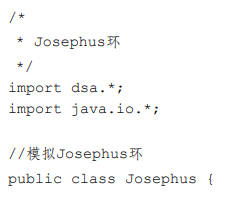
* 循环分配器

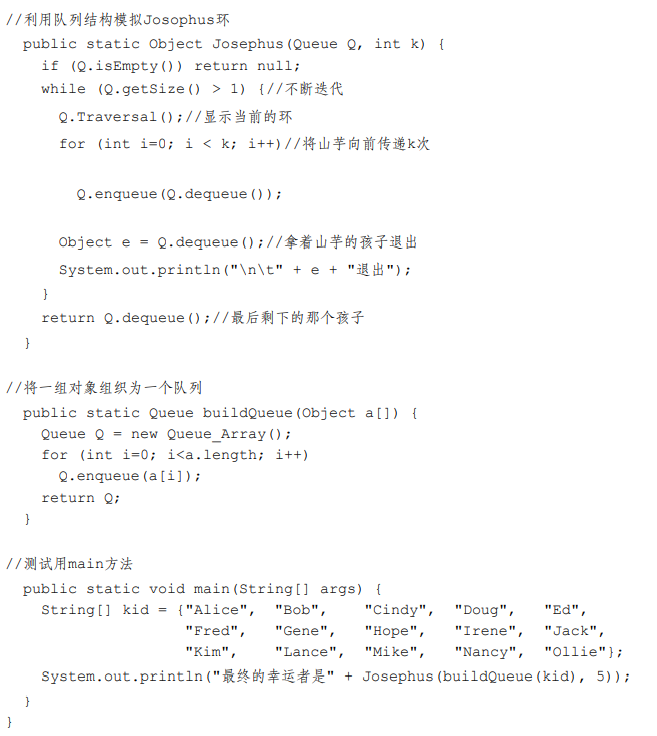


* Josephus环 O(nk)

一群小孩围成一圈，有一个刚出锅的山芋在他们之间传递。其中一个孩子负责数数，每数一次，拿着山芋的孩子就把山芋转交给右边的邻居。一旦数到某个特定的数，拿着山芋的孩子就必须退出，然后重新数数。如此不断，最后剩下的那个孩子就是幸运者。

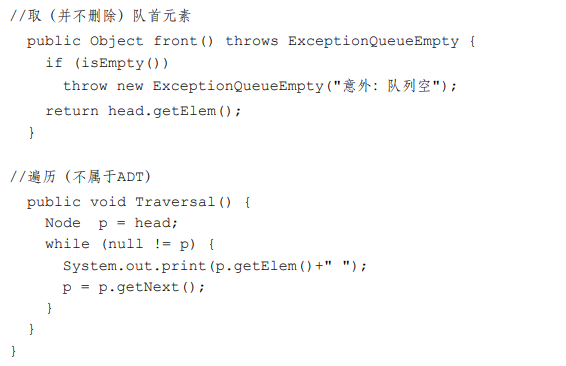
利用队列结构来表示围成一圈的n个孩子。一开始，假定对应于队列首节点的那个孩子拿着山芋。然后，按照游戏的规则，把“土豆”向后传递到第k个孩子（交替 进行k次dequeue()和k次enqueue()操作），并让她出队（dequeue()）。如此不断迭代，直到队长 （getSize()）为 1。





**单链表实现队列**

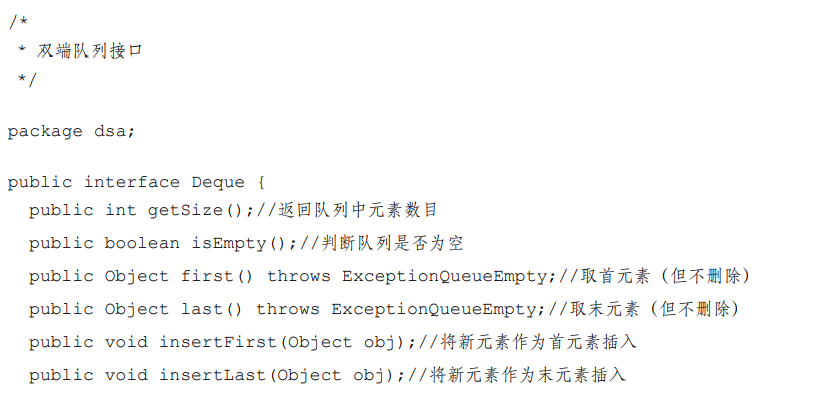


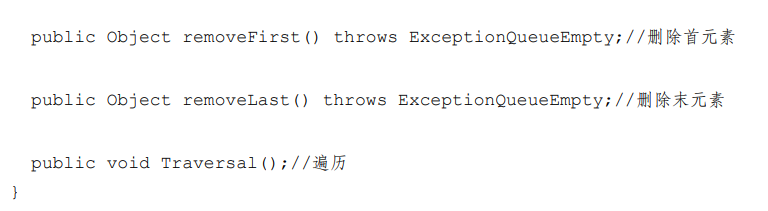


**双端队列 Deque**

前端与后端都支持插入和删除操作的队列

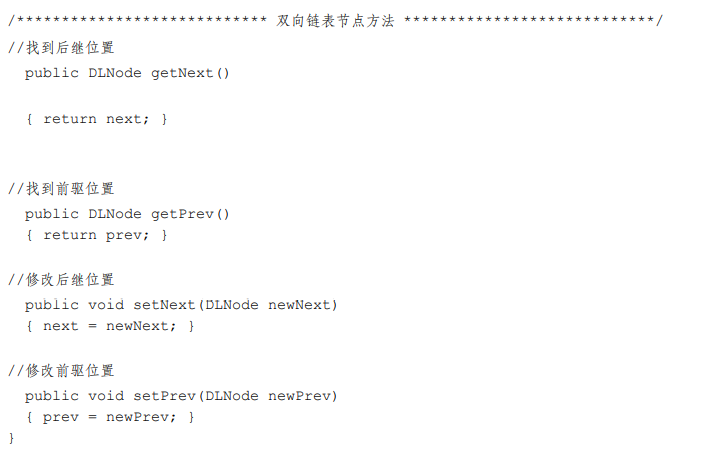
**Interface**

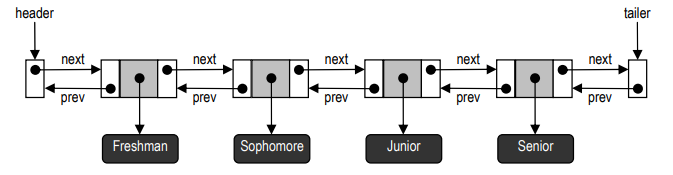




**双向链表**

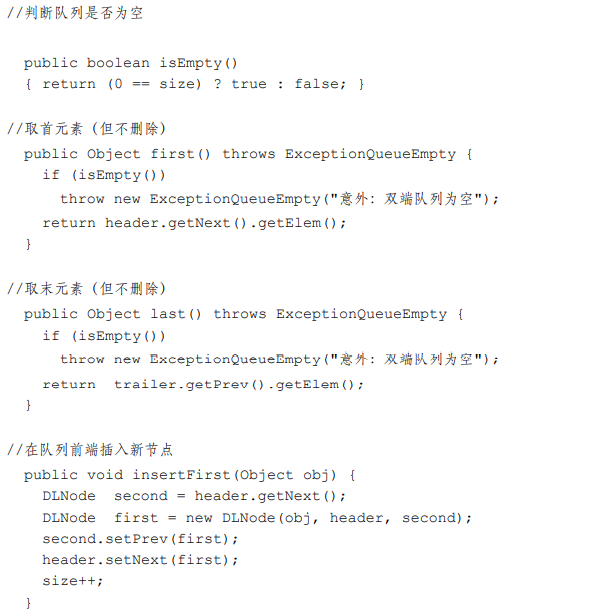


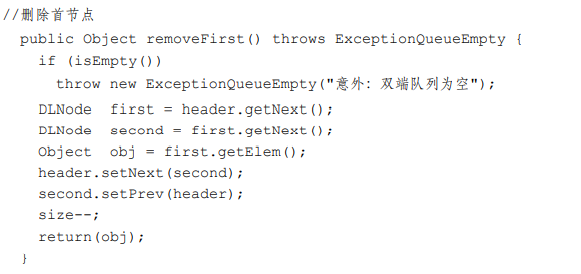




在基于NLNode类实现双向链表的时候为了使编程更加简洁，通常我们都要在最前端和最后端各设置一个哑元节点（Dummy node）。这两个节点分别称作头节点（Header node）和尾节点（Trailer node），起哨兵（Sentinel）的作用。







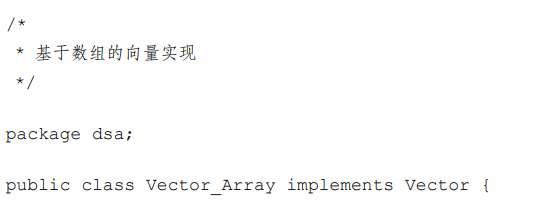


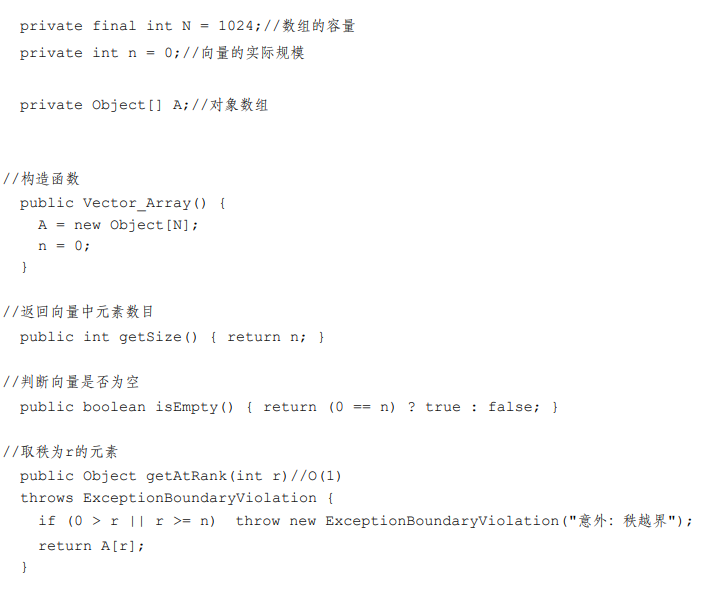
# 3. 向量、列表与序列

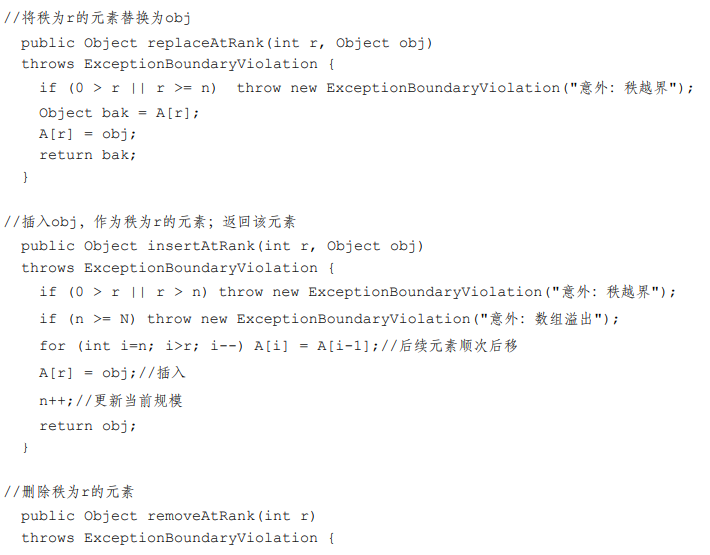
## 1 向量

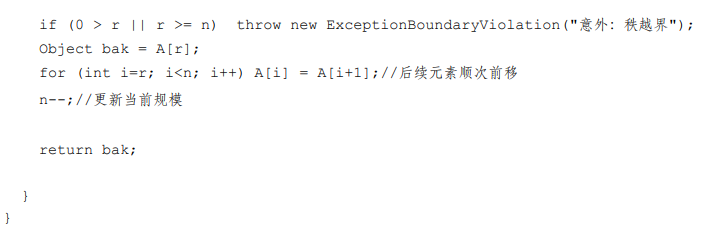
对数组结构进行抽象与扩展之后，就可以得到向量结构，因此向量也称作数组列表（Array list）。向量提供一些访问方法，使得我们可以通过下标直接访问序列中的元素，也可以将指定下标处的元素删除，或将新元素插入至指定下标。为了与通常数组结构的下标（Index）概念区分开来，我们通常将序列的下标称为秩（Rank）。

**基于数组的向量实现 O(n)**









**基于可扩充数组的实现** Extendable array

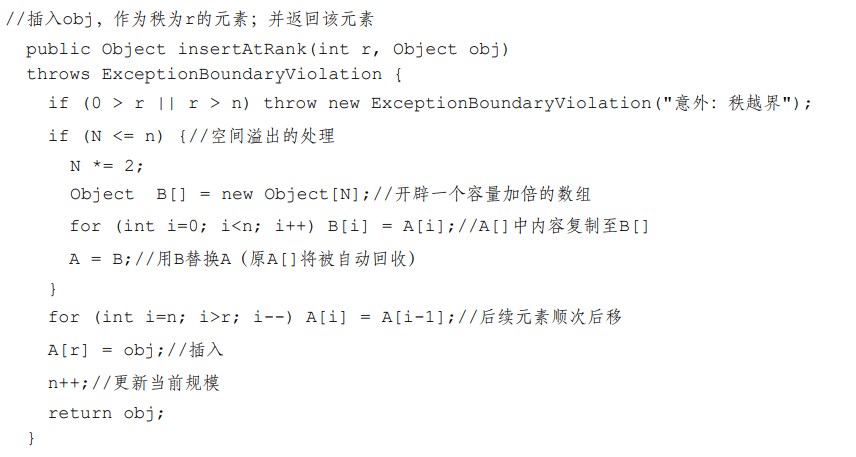
一旦数组空间溢出（即 n≥N），insertAtRank()方法就会做如下处理：

1. 开辟一个容量为2N的新数组B

2. 将A[]中各元素搬迁至B[]，即B[i]=A[i]，i=0, …, N-1

3. 将A替换为B，即令A=B

此后，insertAtRank()就可以将新元素插入至扩容后的数组



基于可扩充数组实现的向量，每次数组扩容的分摊运行时间为 O(1).

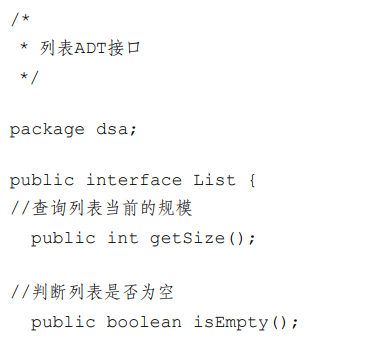
java.util.ArrayList和java.util.Vector 🡪 根据实际需要，动态扩充数组的容量

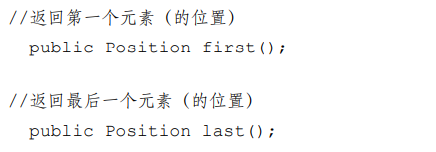
java.util.Vector类设有一个名为capacityIncrement的参数，通过这一参数，程序员可以指定每次对数组进行扩容的增量;该参数默认为0，一旦数组溢出，就将其容量加倍。

通常情况下，若将capacityIncrement设为正整数，则在连续n次操作过程中消耗于数组扩容的 时间有可能会高达Ω(n2 )⎯⎯也就是说，分摊下来每一操作平均需要O(n)时间！

## 2 列表

列表ADT则是对链表结构的抽象。列表提供的访问、更新方法，按照面向对象的规范对列表的节点对象进行了封装，称作位置（Position）。

**Interface**

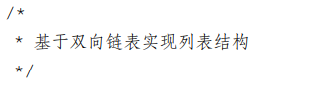


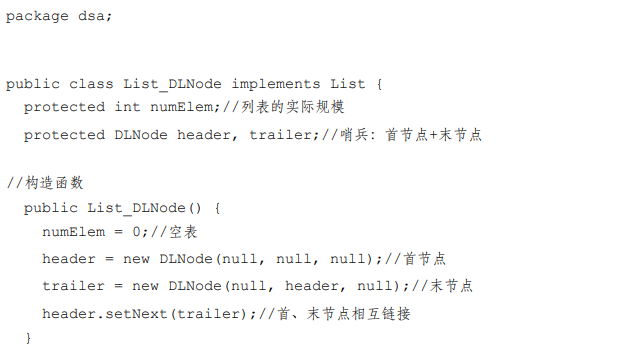


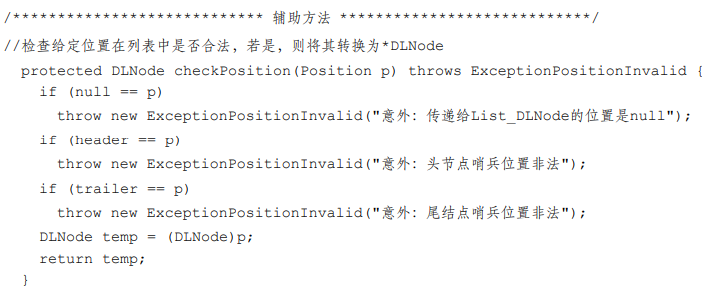


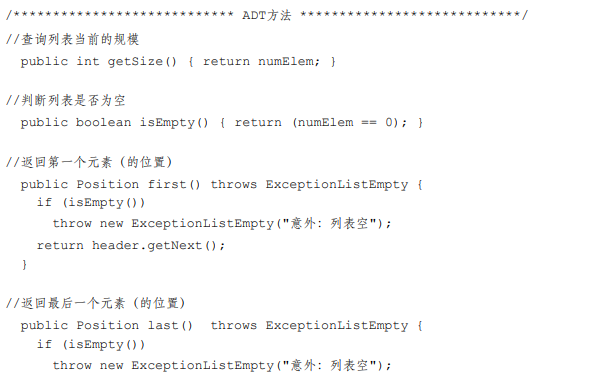


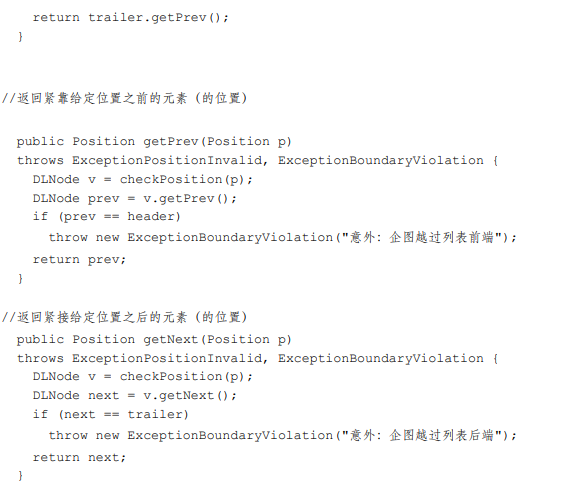
**基于双向链表实现的列表**

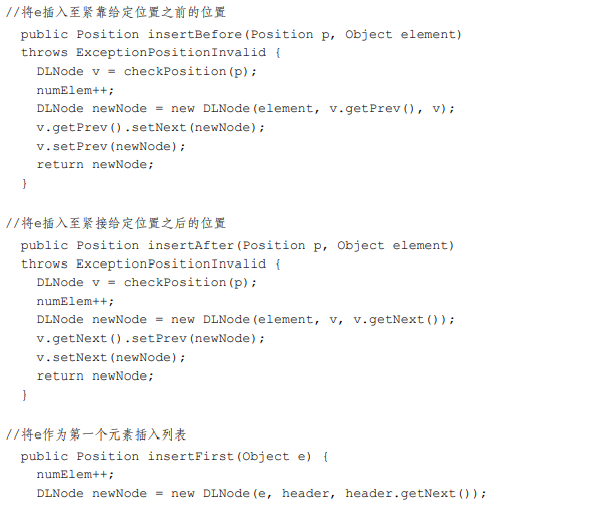


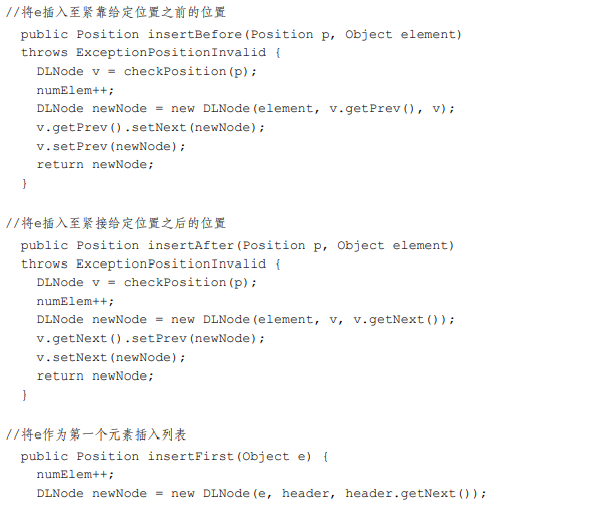


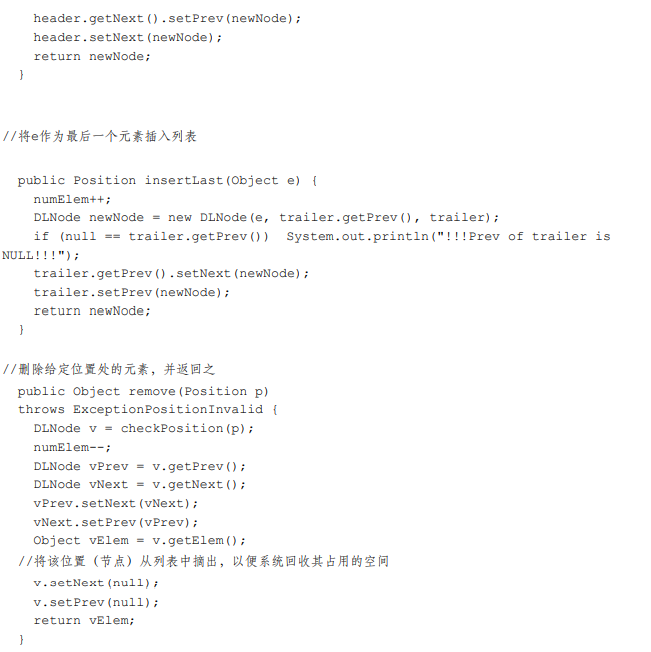


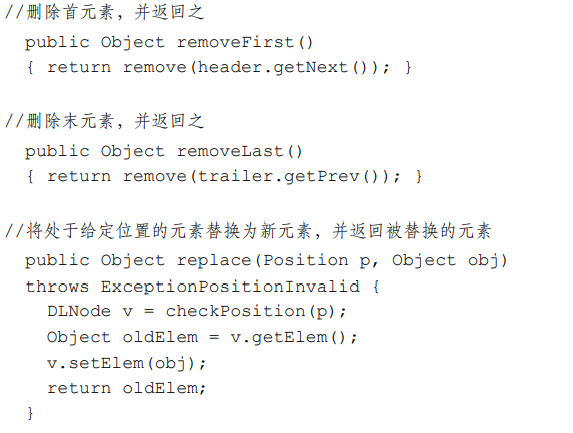


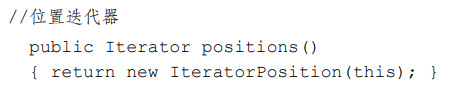


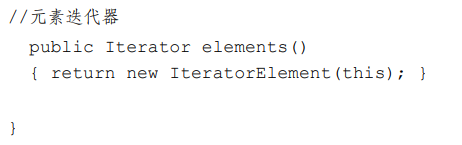








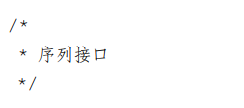


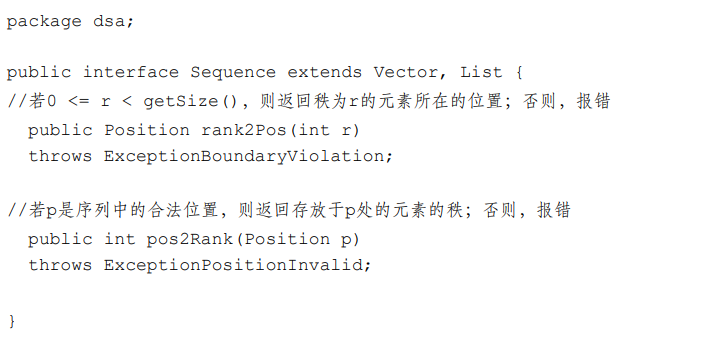


## 3 序列

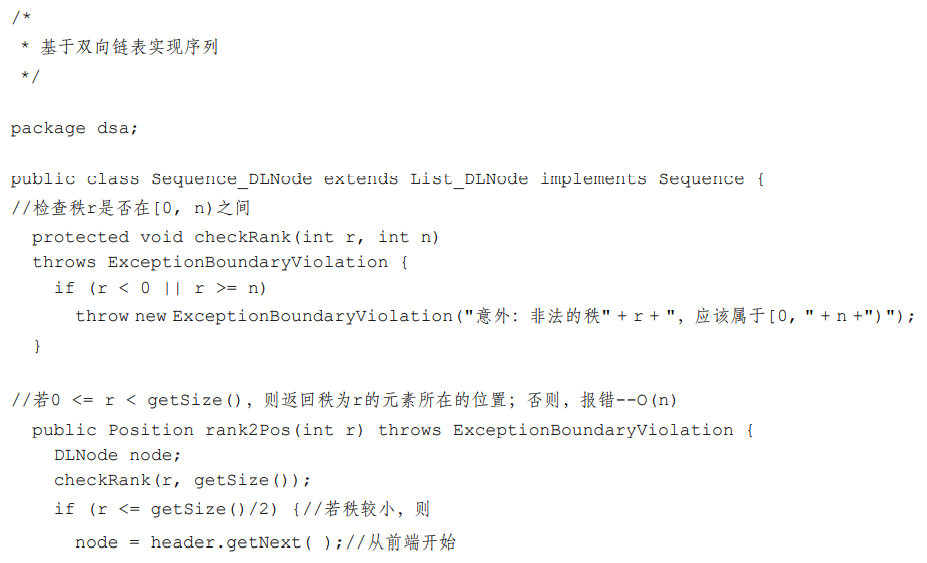
通用的序列将向量和列表的方法集成起来

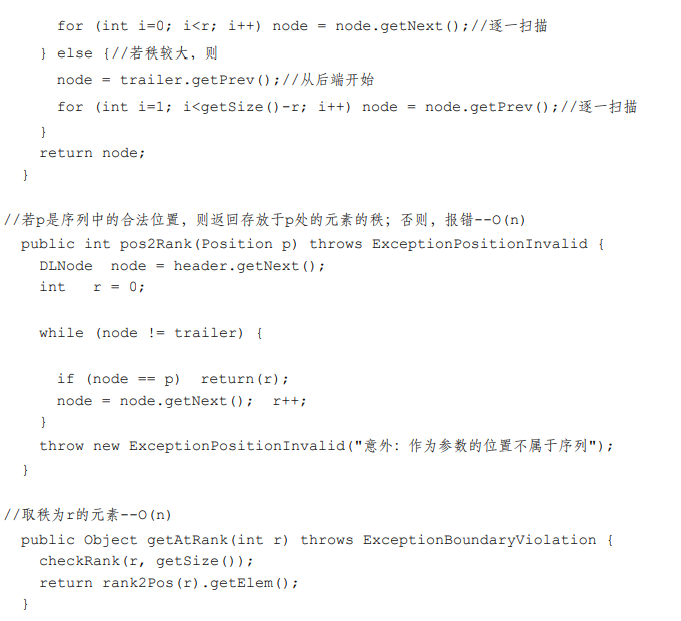
**Interface**





**基于双向链表实现序列**

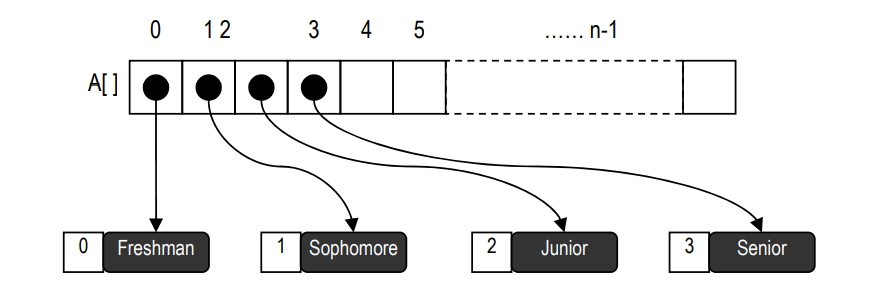






**基于数组实现序列**

将序列S中的各个元素分别存放到数组A[]对应的单元中。

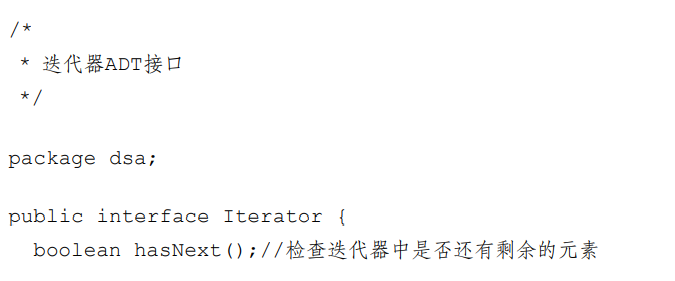


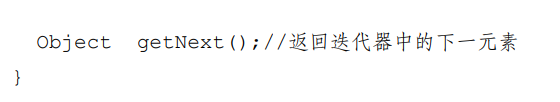
按照上述方式实现序列，一旦序列有所变动，就需要将有关的位置对象顺次移动，而且在最坏情况下需要移动O(n)个对象，因此 insertFirst()、insertBefore()、insertAfter()和 remove()方法都需要耗费O(n)时间。其它那些基于位置的操作则只需要O(1)的时间。

## 4 迭代器

迭代器本身也是一个序列S，在任何时候，迭代器中都有唯一的当前元素。迭代器还必须提供某种机制，使得我们可以不断转向S中的下一元素，并将其置为新的当前元素。

**Interface**



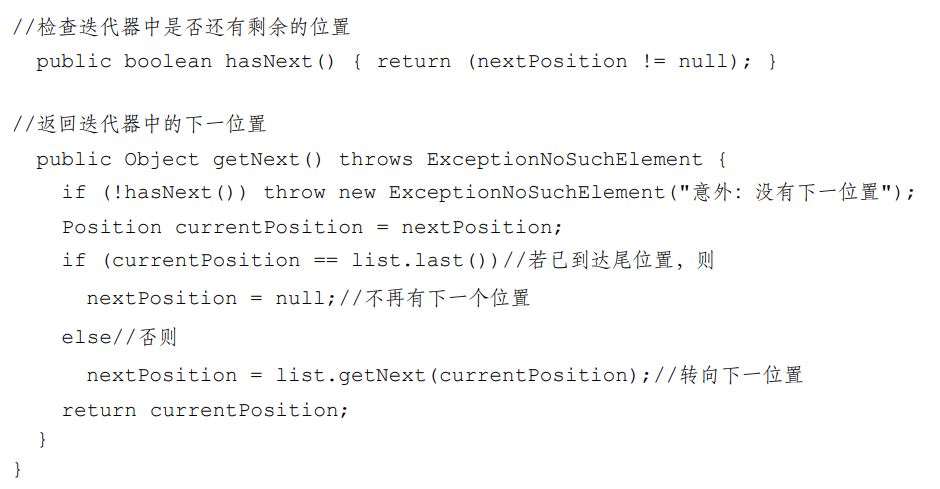


**利用快照建立迭代器 O(n)**

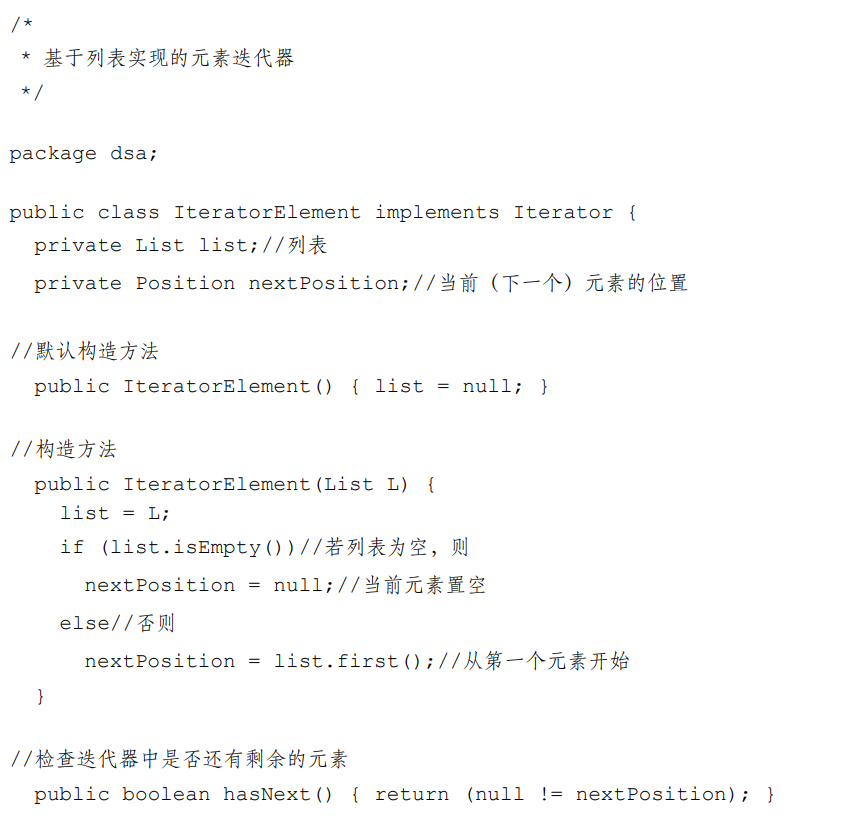
实现迭代器的一种直接办法，就是给容器中的所有元素照张“快照”，并用某一数据结构将其记录下来。比如，可以借助栈结构来存放 “快照”⎯⎯将所有元素（的引用）压入某一栈中。于是，!isEmpty()方法就等效于hasNext()方法，而pop()操作则等效于getNext()操作。

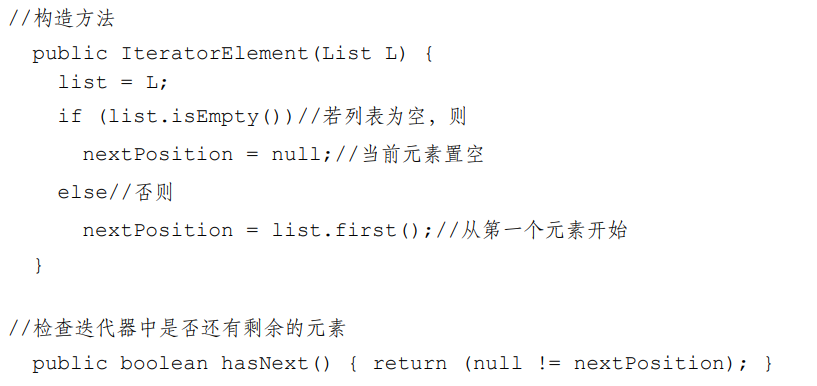
**IteratorPosition()的实现 O(1)**





**IteratorElement()的实现**





Java 对LinkedList对象的访问与更新并不是通过秩完成的，而是更倾向于另一种方式⎯⎯通过 listIterator()方法，创建一个ListIterator迭代器。java.util.ListIterator

java.util中的ArrayList类和Vector类都是基于数组实现的，而LinkedList类则是基于链表实现的。

# 4. 树

## 1 树的术语和性质

在树结构中

* 每个节点的深度都是一个非负整数；
* 深度为0的节点有且仅有一个，称作树根（Root）；
* 对于深度为 k (k≥1)的每个节点u，都有且仅有一个深度为k-1的节点v与之对应，称作u的父亲（Parent）或父节点。

若节点v是节点u的父亲，则u称作v的孩子（Child），并在二者之间建立一条树边（Edge）。

每个节点至多只有一个父亲，但却可能有多个孩子。同一节点的孩子互称“兄弟” （Sibling）。

树中所有节点的最大深度，称作树的深度或高度。

树中节点的数目，总是等于边数加一。 Num(node) = Num(edge) + 1

任一节点的孩子数目，称作它的“度”（Degree）。

至少拥有一个孩子的节点称作“内部节点”（Internal node）；没有任何孩子的节点则称作 “外部节点”（External node）或“叶子”（Leaf）。一个节点是叶子，当且仅当它的度数为零。

由树中k+1节点通过树边首尾衔接而构成的序列{ (v0, v1), (v1, v2), …, (vk-1, vk) | k ≥ 0}，称作树中长度为k的一条路径（Path）。

树中任何两个节点之间都存在唯一的一条路径。这就意味着，树既是连通的，同时又不致出现环路。特别地，在树根与每个节点之间，也存在唯一的一条路径。

若v是u的父亲，则depth(v) + 1 = depth(u)。

每个节点都是自己的“祖先”（Ancestor），也是自己的“后代”（Descendent）；

* 若v是u的父节点的祖先，则v也是u的祖先;
* 若u的父节点是v的后代，则u也是v的后代。

除节点本身以外的祖先（后代），称作真祖先（后代）。

任一节点v的深度，等于其真祖先的数目。

任一节点v的祖先，在每一深度上最多只有一个。

树T中每一节点v的所有后代也构成一棵树，称作T的“以v为根的子树（Subtree）”。

若子树v的深度（高度）为h，则称v的高度为h，记作height(v) = h。根节点的高度就是整棵树的深度（高度）。

对于叶子节点u的任何祖先v，必有depth(v) + height(v) ≥ depth(u)。

在树T中，若节点u和v都是节点a的后代，则称节点a为节点u和v的共同祖先（Common ancestor）。

每一对节点至少存在一个共同祖先。

在一对节点u和v的所有共同祖先中，深度最大者称为它们的最低共同祖先（Lowerest common ancestor），记作lca(u, v)。

每一对节点的最低共同祖先必存在且唯一。

在树T中，若在每个节点的所有孩子之间都可以定义某一线性次序，则称T为一棵“有序树（Ordered tree）”

每个内部节点均为m度的有序树，称作m叉树。

每个节点均不超过 2 度的有序树，称作二叉树（Binary tree）。

不含1度节点的二叉树，称作真二叉树（Proper binary tree），否则称作非真二叉树 （Improper binary tree）。

在二叉树中，深度为k的节点不超过2k个。

高度为h的二叉树最多包含2h+1 - 1个节点。

由n个节点构成的二叉树，高度至少为⎣log2n⎦。

在二叉树中，叶子总是比2度节点多一个。

若二叉树T中所有叶子的深度完全相同，则称之为满二叉树（Full binary tree）。

高度为 h 的二叉树是满的，当且仅当它拥有 2h匹叶子、2h+1 - 1个节点。

若在一棵满二叉树中，从最右侧起将相邻的若干匹叶子节点摘除掉，则得到的二叉树称作完全二叉树（Complete binary tree）。

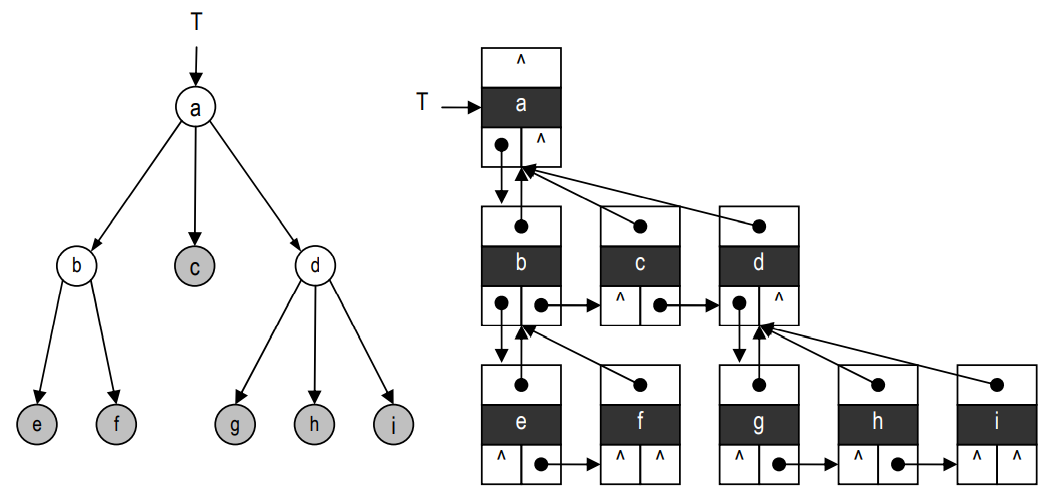
由n个节点构成的完全二叉树，高度 h = ⎣log2n⎦。

在由固定数目的节点所组成的所有二叉树中，完全二叉树的高度最低。

## 2树抽象数据类型及其实现

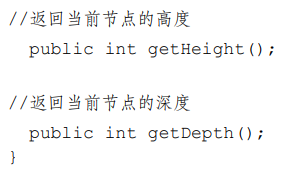
树的抽象结构



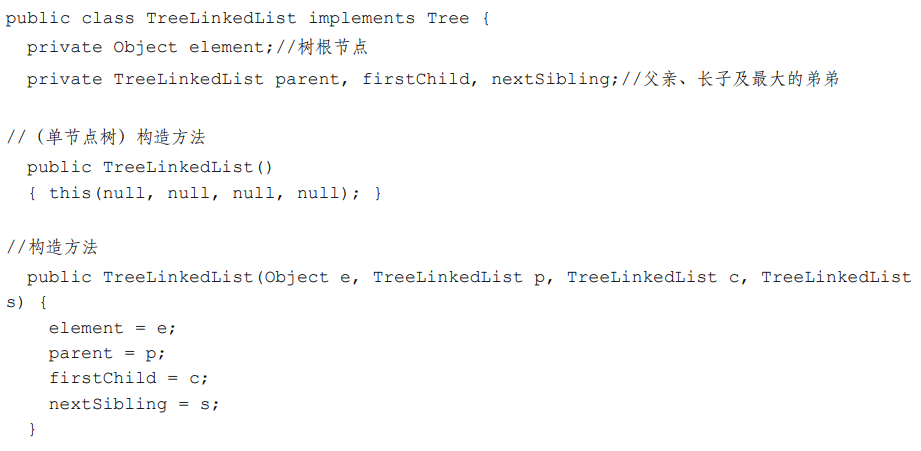


**Interface**



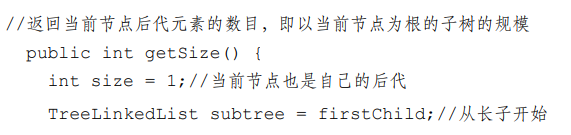


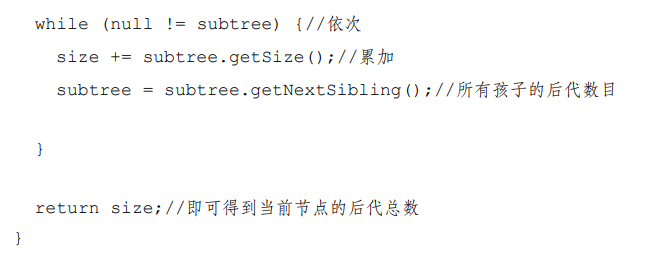
**基于链表实现树**



**getSize()**⎯⎯统计（子）树的规模 O(n)

一棵树的规模，等于根节点下所有子树规模之和再加一，也等于根节点的后代总数。

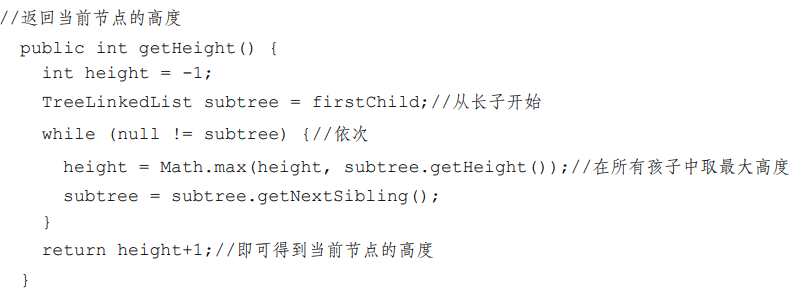




**getHeight()**⎯⎯计算节点的高度 O(n)

若u是v的孩子，则height(v) ≥ height(u) + 1；

height(v) = 1 + max height(u)。(u是v的孩子)

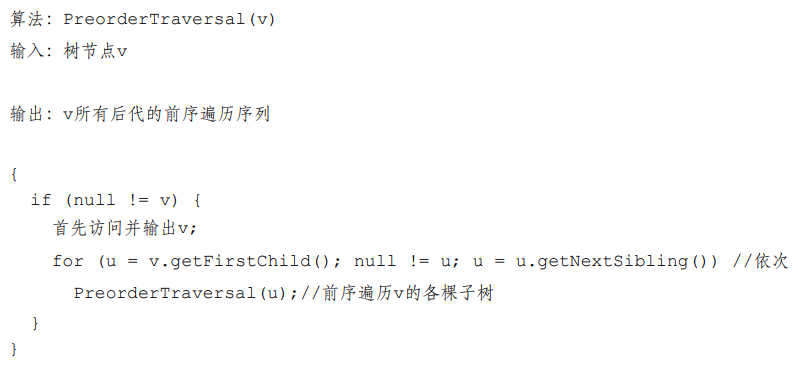


**getDepth()**⎯⎯计算节点的深度 O(n)

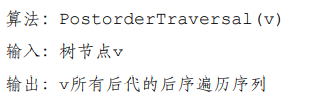
若u是v的孩子，则depth(u) = depth(v) + 1。

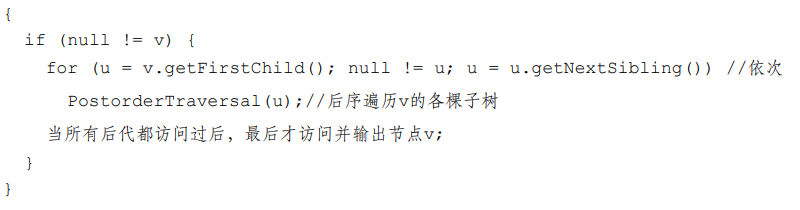


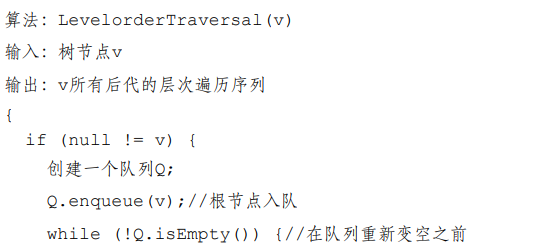
**前序遍历**

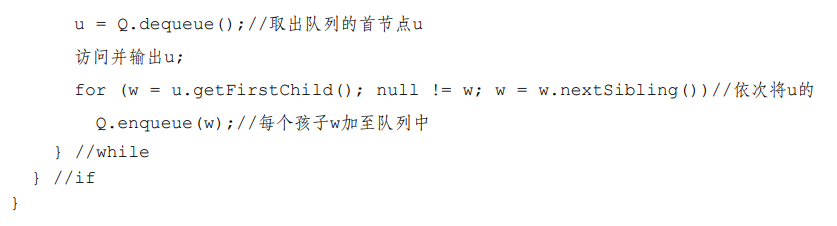


**后序遍历**

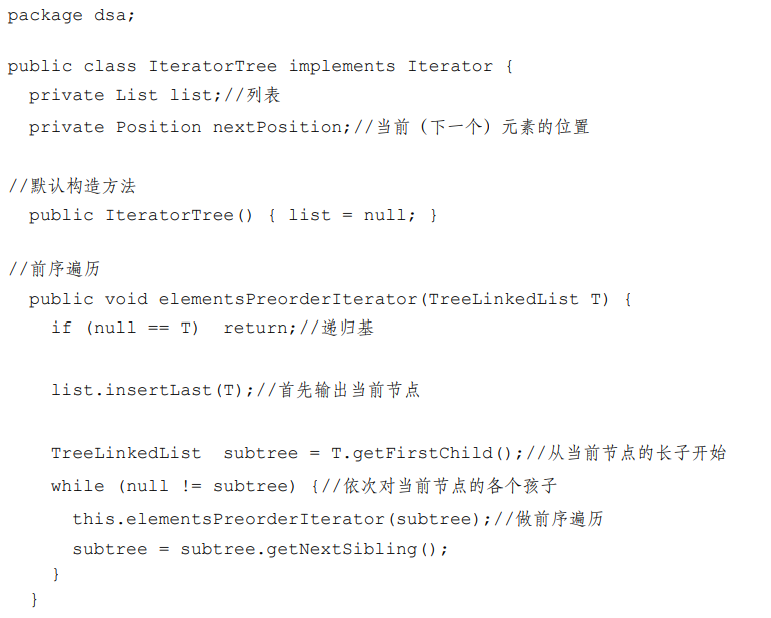


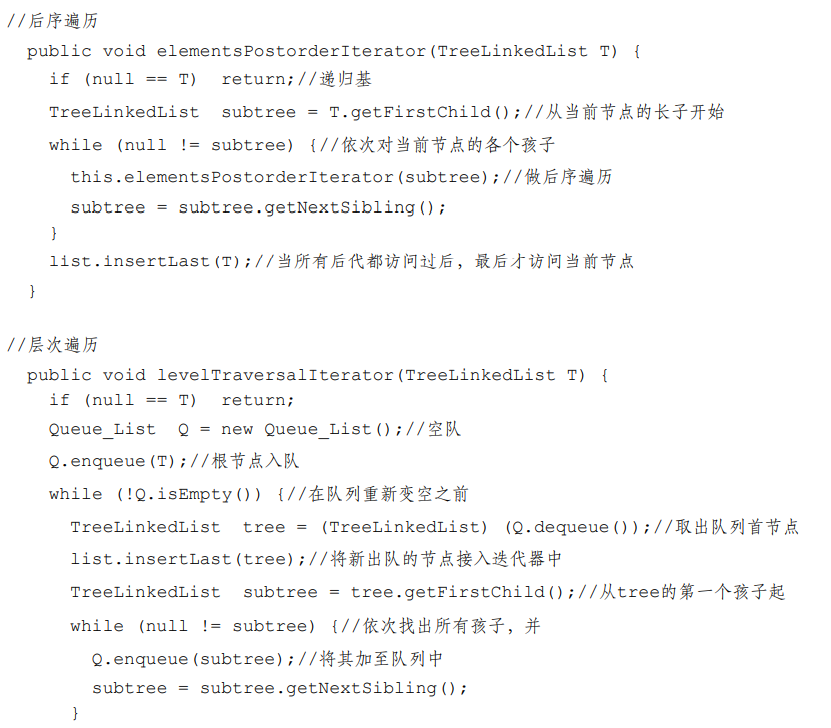


**层次遍历**



**基于列表实现的树迭代器**



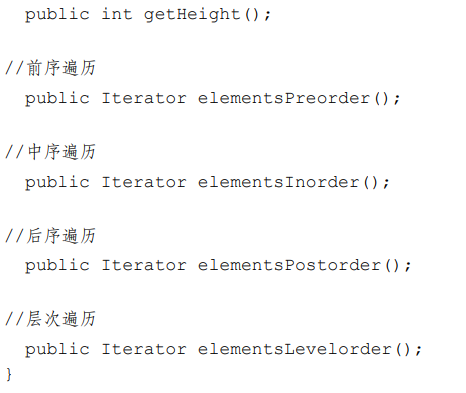


**二叉树的抽象数据类型及其实现**

**Interface**

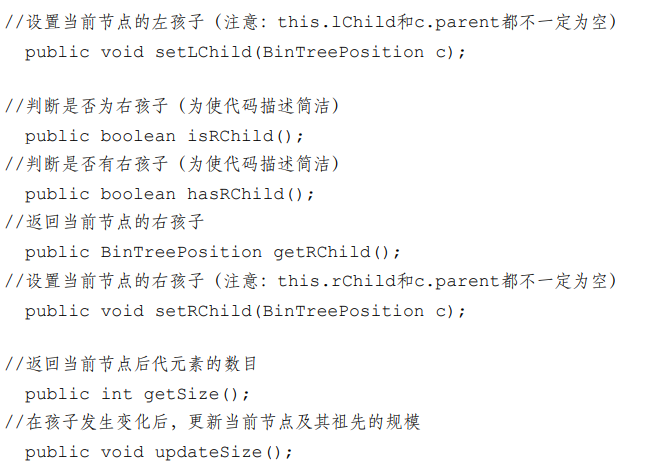
**BinTree**

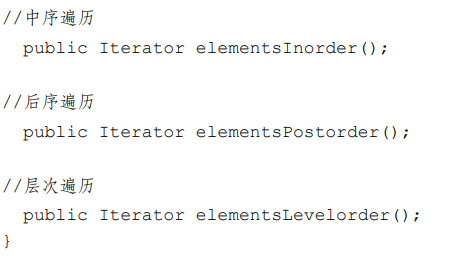




**BinTreePosition**

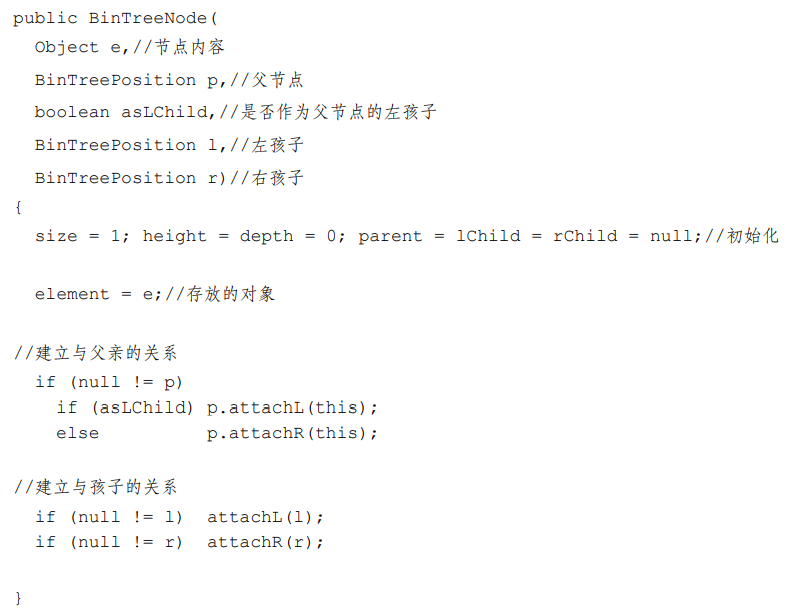




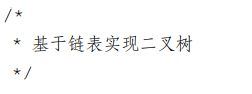


**二叉树节点类的实现**





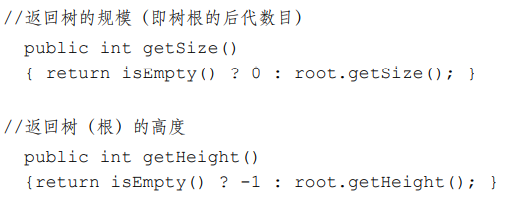
**二叉树类的实现**





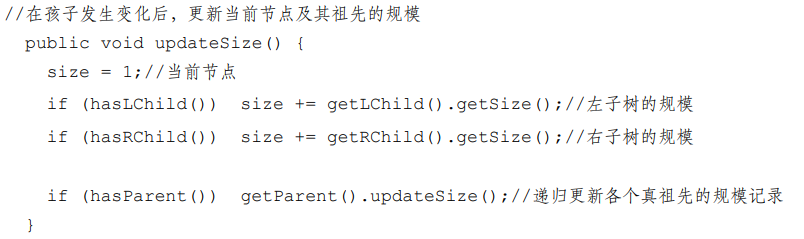
**二叉树的基本算法**

**getSize()、getHeight()和getDepth()**

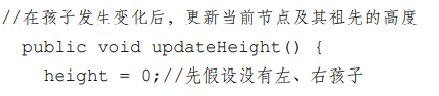
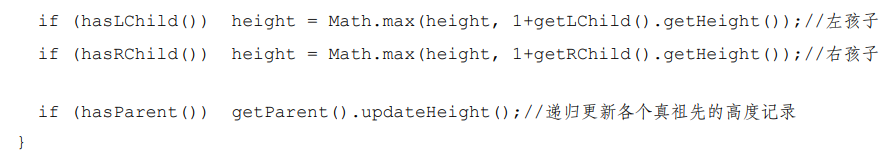


**updateSize()**

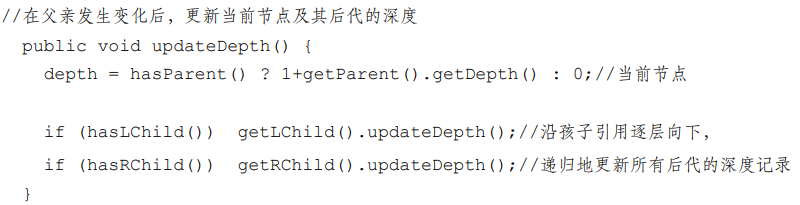
若节点v的左、右孩子分别为lc和rc，则 size(v) = 1 + size(lc) + size(rc)。



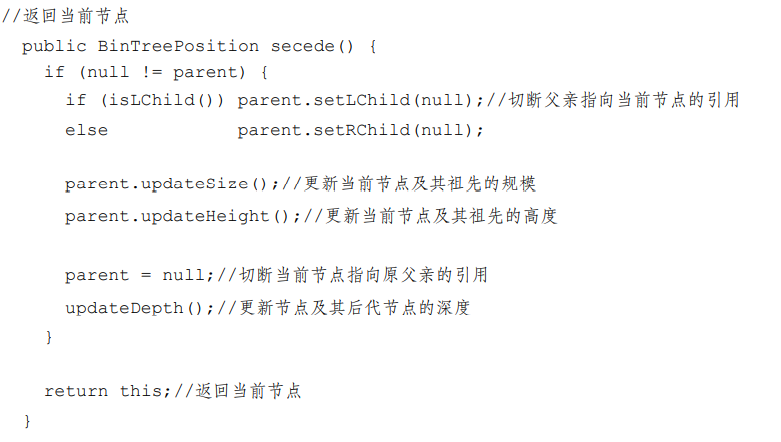
**updateHeight()**



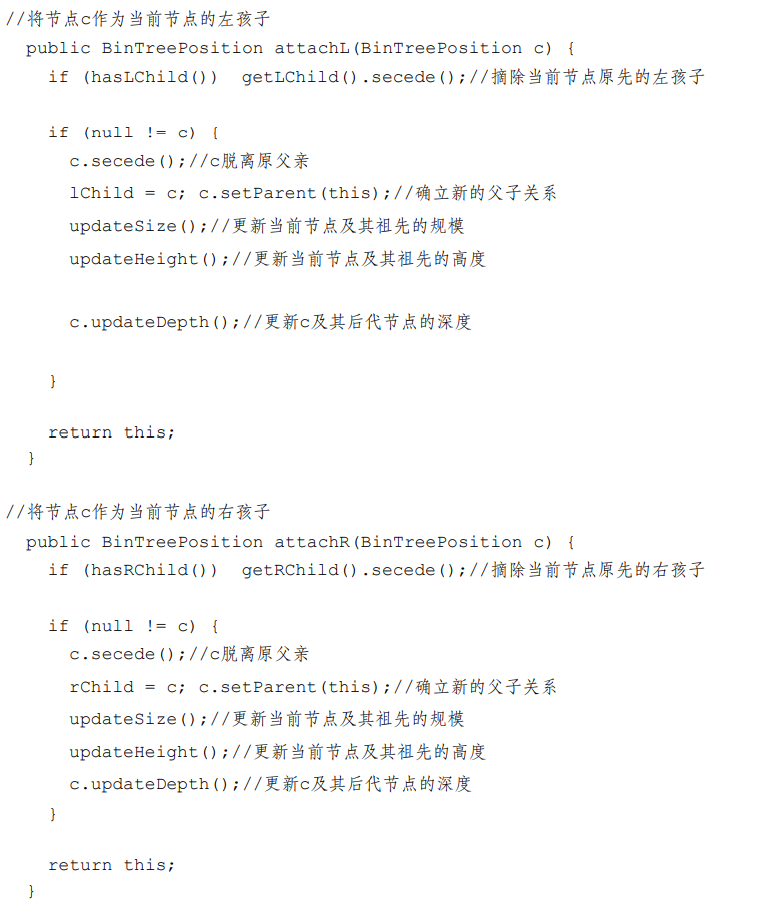
**updateDepth()**



**secede()** 将以某一节点为根的子树从母树中分离出来

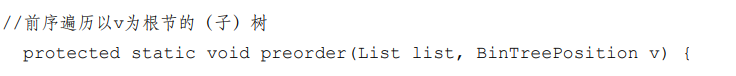


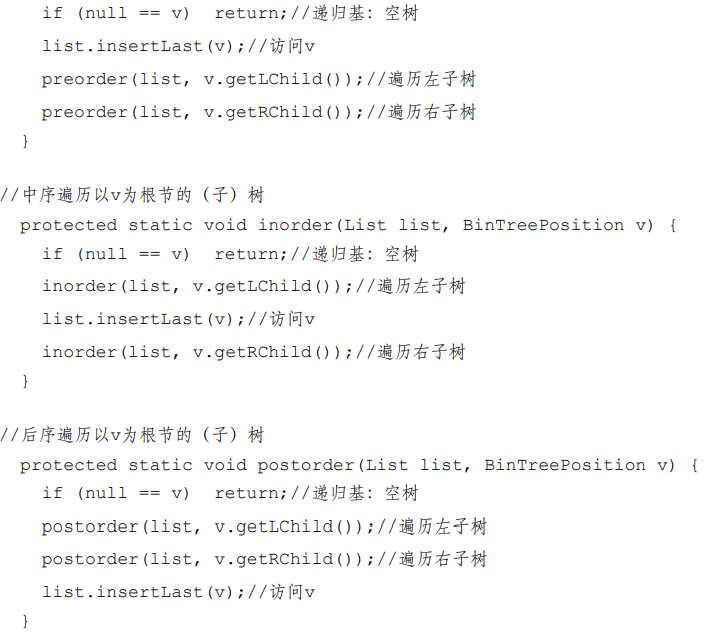
**attachL()和attachR()**

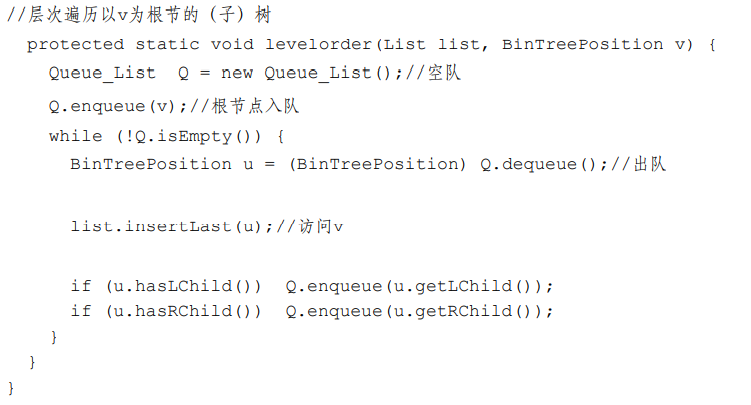


**二叉树的遍历**

**前序、中序、后序、层次**



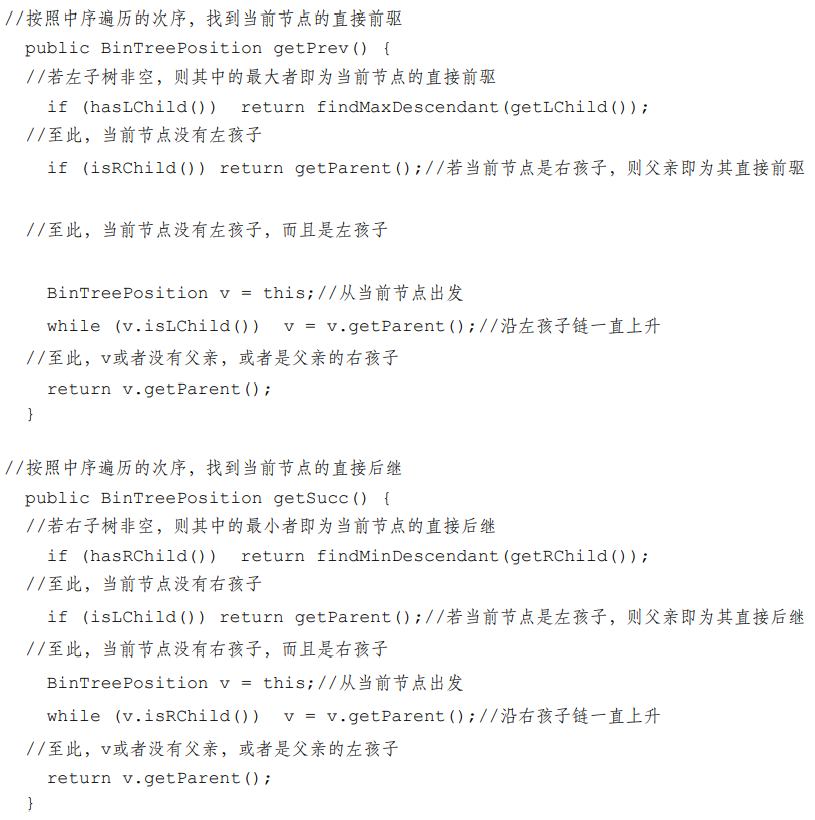




**直接前驱、直接后继的定位算法**二叉树中，除中序遍历序列中的首节点外，任一节点v的**直接前驱u**不外乎三种可能：

1. v没有左孩子，同时v是右孩子：此时，u就是v的父亲节点；
2. v没有左孩子，同时v是左孩子：此时，从v出发沿parent引用逆行向上，直到第一个是右孩子的节点w，则u就是w的父亲节点；
3. v有左孩子：此时，从v的左孩子出发，沿rChild引用不断下行，最后一个（没有右孩子的）节点就是u。

直接后继的查找算法完全是对称的。



**完全二叉树**

**Interface**

**基于向量实现完全二叉树**

若将节点v的这种编号记作i(v)，在完全二叉树中，

若节点v有左孩子，则 i(lchild(v)) = 2 × i(v) + 1；

若节点v有右孩子，则 i(rchild(v)) = 2 × i(v) + 2；

若节点v有父节点，则 i(parent(v)) = = - 1。

若基于可扩充向量来实现完全二叉树，则就分摊复杂度而言，每次 addLast()和 delLast()操作都可以在O(1)时间内完成。

**完全二叉树节点类的实现**

