pdf：目录P9

P156

**回溯法**

辗转相除法

朴素贝叶斯分类算法

**动态规划算法**

**Dijkstra算法**

**BFS(广度优先搜索)**

**DFS（深度优先搜索）**

BFPRT(线性查找算法)

**二分查找算法**

枚举

**字符串匹配 KMP算法。**

**HashMap**

分支限界

随机算法

**桶排序**

Bellman-Ford

**拓扑排序**

# 数学基础

约定：本笔记出现的所有的形如lgn都指的是log2n

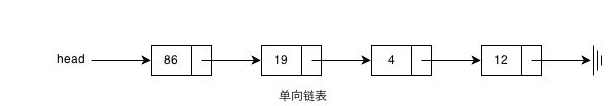
等差数列求和公式：formula 或Sn=n(a1+an)/2

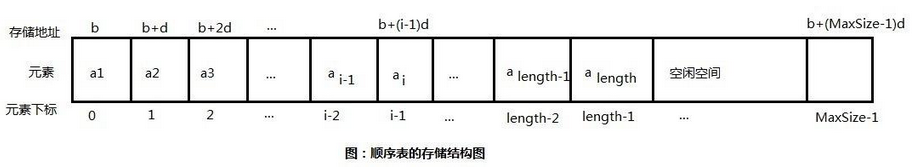
等比数列求和公式： formula或Sn= (a1-an\*q)/(1-q)

1. 数据结构
   1. 表

表分为两种，一种表是**链表**，一种表是**顺序表**，**顺序表实际上就是数组**，链表在逻辑上连续，物理地址不连续，数组逻辑上连续，物理地址也连续

链表的每个元素具有next指针，用于指向下一个元素节点，如下表节点86的next指针指向节点19





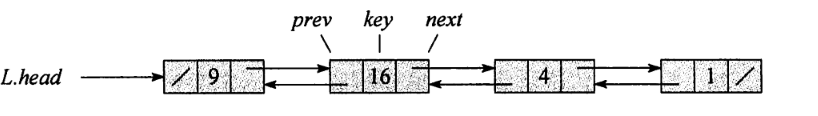
如果链表带头结点，则L.head->next=null时链表为空，若链表不带头结点，则L.head=NIL时链表为空(NIL和NULL的区别是：NIL是空对象，NULL是空，前者是后者的真子集)

**优缺点：**

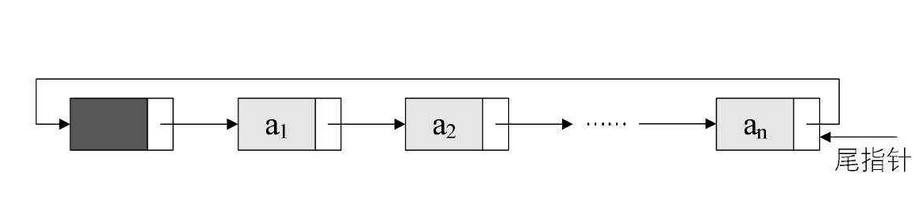
顺序表在内存物理地址上连续，因此访问节点速度快，但增删节点速度慢

链表因为在内存物理地址上不连续，因此访问节点速度慢，但增删节点速度快

**双向链表**是链表的一个变种，它的每个元素节点比链表多了一个prev指针，用来指向前一个节点

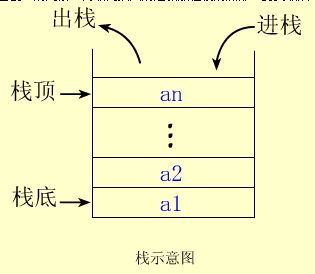


**循环链表**也是链表的一个变种，与普通的单链表不同，它的最后一个节点的next不再指向null，而是指向头结点



* 1. 栈

栈是一种**先进后出**的数据结构，只能在栈顶进行删除和插入操作

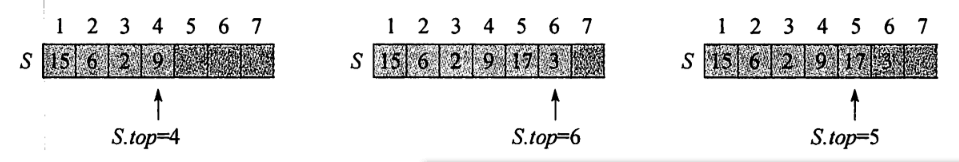


操作：

push(栈S,数据data) 入栈

pop(栈S) 出栈

stackEmpty(栈S) 检测是否栈空



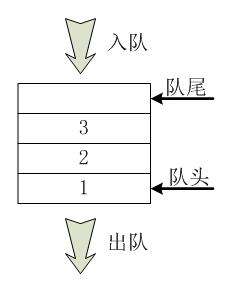
S[1] 栈底

S[S.top] 栈顶

当S.top=0时，栈空；S.top=1时，栈中只有一个元素

* 1. 队列

队列是一种**先进先出**的数据结构，只能在队尾插入，队头删除

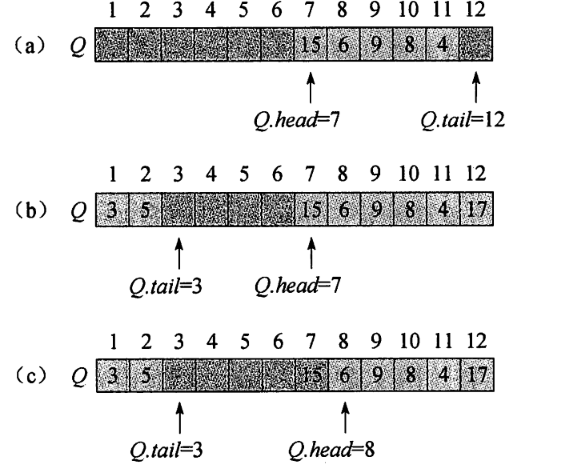


操作：

enQueue(栈S,数据data) 入队

deQueue(栈S) 出队

stackEmpty(栈S) 检测是否栈空



Q[Q.head] 队头

Q[Q.tail] 队尾

当Q.head=Q.tail时，队空

上图(a)，进行enQueue(Q,17), enQueue(Q,3), enQueue(Q,5)三个操作后得到(b);

调用deQ ueue(Q)后得到(c)

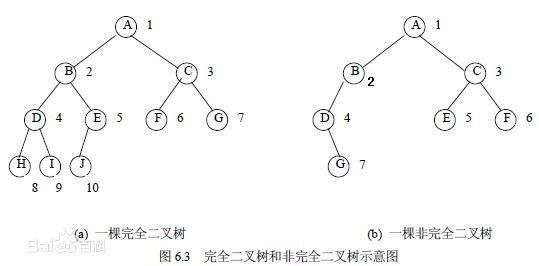
* 1. 树

树是一种链式的，有层次结构的数据结构，它由一个根节点和诸多子孙节点组成

* + 1. 二叉树的性质

二叉树是树结构中最重要的一种类型，它的每个父节点最多只能有两个子节点。

若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。



**(a)有4层，高度为⌊log210⌋+1=4，D节点的深度为3,C节点的度为2，E节点的度为1，J节点的度为0**

**二叉树的性质：**

1. 在**二叉树**的第i层上至多有2i-1个结点
2. 高度为h的**二叉树**，最多有2h-1个节点

证明：

20+21+22+23+.…2h-1=2h-1

1. 具有n个结点的**完全二叉树**的高度h为⌊log2n⌋+1
2. 对于一棵**二叉树**，设n0，n1，n2分别代表度为0,1,2的节点，则n0=n2+1

证明：

对于任意二叉树，都有n= n0+n1+n2

二叉树除了根节点外的每个节点都有一根**’绳子’**吊着，有多少根**’绳子’**意味着二叉树的总**’度’**数有多少有多少，即

0\* n0+1\*n1+2\*n2=n-1

联立得：n0=n2+1

1. 在**二叉树**的第i层上至多有2i-1个结点
   * 1. 遍历二叉树

对于一棵二叉树，我们一般有三种遍历方式：**先序遍历**，**中序遍历**，**后序遍历**

**(1)先序遍历**

访问根结点

先序遍历左子树

先序遍历右子树

**(2)中序遍历**

中序遍历左子树

访问根结点

中序遍历右子树

**(3)后序遍历**

后序遍历左子树

后序遍历右子树

访问根结点

* + - 1. 递归实现

递归实现比较简单，这里只举一个中序遍历的例子

//中序遍历——递归实现  
private List<Node> midIteratorByRecurison(Node node) {  
 if (null == node) {  
 return null;  
 }  
 List<Node> result = new ArrayList<>();  
 List<Node> leftNodeArr = null;  
  
  
 if (null != node.left) {  
 //递归遍历左子树  
 leftNodeArr = midIteratorByRecurison(node.left);  
 }  
 if (!CollectionUtils.*isEmpty*(leftNodeArr)) {  
 result.addAll(leftNodeArr);  
 }  
 //递归遍历根节点  
 result.add(node);  
 List<Node> rightNodeArr = null;  
 if (null != node.right) {  
 //递归遍历右子树  
 rightNodeArr = midIteratorByRecurison(node.right);  
 }  
 if (!CollectionUtils.*isEmpty*(rightNodeArr)) {  
 result.addAll(rightNodeArr);  
 }  
 return result;  
}

* + - 1. 非递归实现

非递归实现主要是借助栈来帮助我们实现，经过观察可以发现，无论是哪种遍历，都是优先访问左子树，后访问右子树的规则，这跟栈很相似，因此可以借助栈保存左、右子树

* + - * 1. 先序遍历

先序遍历分为以下几个步骤：

1. 将currentNode压入stack中
2. currentNode节点出栈，访问currentNode节点
3. 如果currentNode的右子树不空，则右节点入栈
4. 如果currcurrentNodeent的左子树不空，则左节点入栈

重复上面的循环，每一次循环的currentNode节点都是从stack中取出的Node

//前序遍历——非递前序归遍历实现  
private List<Node> preIterator(Node root) {  
 List<Node> result = new ArrayList<>();  
 Stack<Node> stack = new Stack<>();  
 //栈中压入根节点  
 stack.push(root);  
  
 while (!stack.isEmpty()) {  
 //栈不空就出栈  
 Node currentNode = stack.pop();  
 result.add(currentNode);  
 if (null != currentNode.right) {  
 //如果右子树不空，先入栈保存，访问右子树的顺序遵循先进后出原则  
 stack.push(currentNode.right);  
 }  
 if (null != currentNode.left) {  
 //左子树不空，也入栈，这样出栈的时候是先左子树，再右子树  
 stack.push(currentNode.left);  
 }  
 }  
 return result;  
}

* + - * 1. 中序遍历

1. 每次循环都将currentNode节点的所有左子树都按顺序压入stack中，直到左孩子为空
2. Stack出栈一个节点，判断其是否有右子树
3. 如果有右节点，则重复上述操作，将右节点当成currentNode，重复1、2步骤
4. 如果没有右节点，则继续出栈

//中序遍历——非递中序归遍历实现  
private List<Node> midIterator(Node root) {  
 List<Node> result = new ArrayList<>();  
 Stack<Node> stack = new Stack<>();  
 Node currentNode = root;  
 while (currentNode != null || !stack.isEmpty()) {  
 //每次循环都将当前节点currentNode的所有左子树入栈（包括currentNode）  
 while (null != currentNode) {  
 //左子树不空，入栈  
 stack.push(currentNode);  
 currentNode = currentNode.left;  
 }  
  
 currentNode = stack.pop();  
 result.add(currentNode);  
 if (null != currentNode.right) {  
 //右子树不空，进入右子树  
 currentNode = currentNode.right;  
 }else{  
 //右子树为空，置空currentNode，不要进入左子树入栈的循环  
 currentNode = null;  
 }  
 }  
 return result;  
}

* + - * 1. 后序遍历

后序遍历只需要再中序遍历上进行修改，我们需要加一个指针，用于记录被完全访问完毕的节点

//后序遍历——非递后序归遍历实现  
private List<Node> afterIterator(Node root) {  
 List<Node> result = new ArrayList<>();  
 Stack<Node> stack = new Stack<>();  
 //记录上一次被访问完的节点(访问完意味着左子树遍历完，右子树遍历完，当前节点也访遍历了)  
 Node prev = null;  
 Node currentNode = root;  
 while (currentNode != null || !stack.isEmpty()) {  
 //每次循环都将当前节点currentNode的所有左孩子入栈（包括currentNode）  
 while (null != currentNode) {  
 //左子树不空，入栈  
 stack.push(currentNode);  
 currentNode = currentNode.left;  
 }  
  
 currentNode = stack.peek();  
 //如果右节点已经被访问完了，或者右节点为空  
 if (currentNode.right == prev || null == currentNode.right) {  
 //右子树为null，代表左子树访问完了，又没有右子树，所以出栈，并且将当前节点加到结果集里  
 currentNode = stack.pop();  
 result.add(currentNode);  
 //记录上一次被访问完的节点(访问完意味着左子树遍历完，右子树遍历完，当前节点也访遍历了)  
 prev = currentNode;  
 //右子树为空，置空currentNode，不要进入上面的左子树入栈的循环  
 currentNode = null;  
 } else {  
 //右子树不空，进入右子树  
 currentNode = currentNode.right;  
 }  
 }  
 return result;  
}

* + 1. 二叉搜索(查找)树
       1. 定义

二叉搜索树是以二叉树的形式来组织的，它的每个节点包含**left，right和p三个指针，分别指向左右孩子节点和双亲节点**，根节点是二叉搜索树中唯一一个p指针指向NIL的节点，其时间复杂度为**log2N~N**。

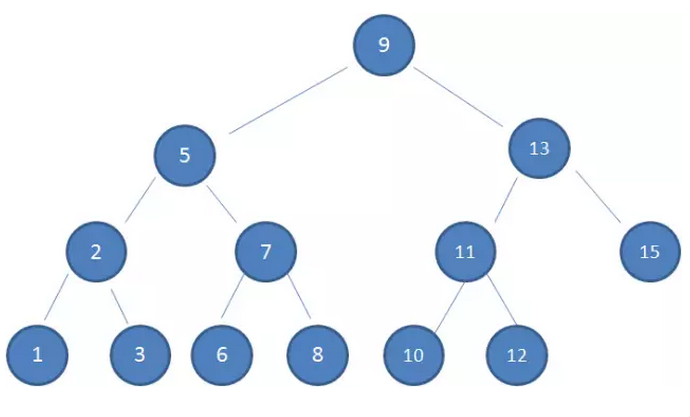
二叉搜索树的定义如下：

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉查找树

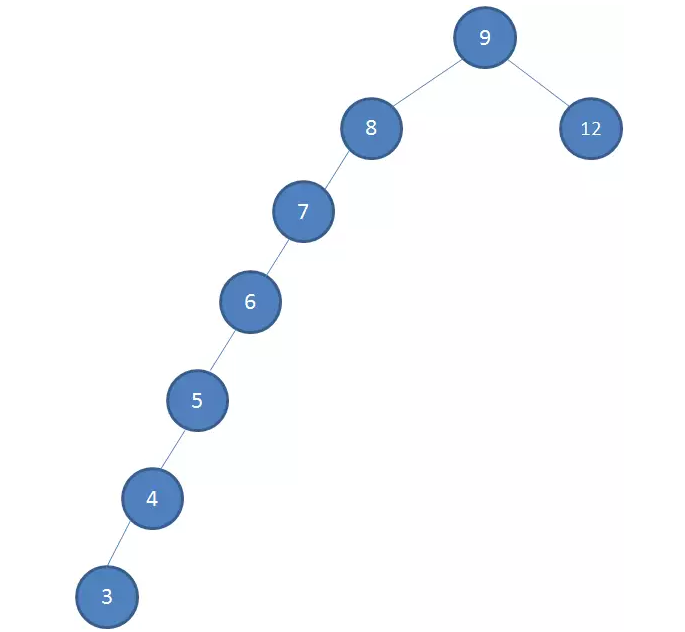
如下图：

 这样的结构可以帮助我们快速查找到自己想要的节点，比如我们希望查找值为12的节点，首先9<12，所以看右子树，13>12，所以看左子树，11<12所以看右子树，成功找到值为12的节点。二叉查找树，其最坏情况下的查找次数等于树的高度，这极大提高了节点的查询效率。当然了，在插入节点时，也是类似的道理。

* + - 1. 遍历

树的遍历根据根节点出现的位置顺序一般分为三种：**先序遍历、中序遍历、后序遍历，**由于二叉搜索树的特殊性质，我们在选择遍历方式时，一般选择中序遍历，这样输出的遍历结果就是有序的

但是，在一些极端情况下，二叉查找树可能退化到接近链表的程度，比如下面这种情况



在这种情况下，二叉查找树的查找效率将大打折扣，为了解决这种极端情况下性能损耗严重的问题，二叉平衡树应运而生

* + 1. 平衡二叉树
       1. 定义

二叉平衡树是一种特殊的二叉查找树，它具有以下特征：

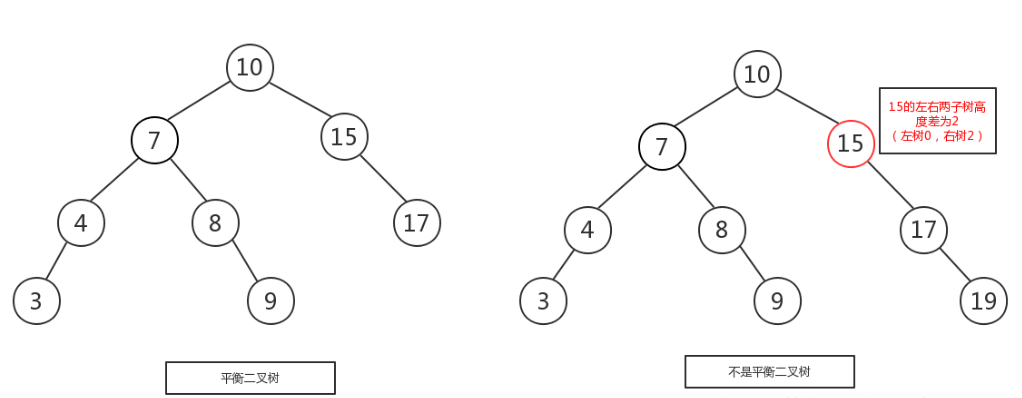
1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3. 树的左右两边的层级数相差不会大于1

4. 左、右子树也分别为二叉平衡树

相比于二叉查找树，二叉平衡树(AVL)在左右子树的高度上做了限制，保证二者高度不会相差过大，这会使得二叉平衡树(AVL)的插入和删除操作变得更加复杂(新插入、删除节点都需要重新平衡二叉树)，但会将最好情况和最差情况的时间复杂度都维持在log2(n)

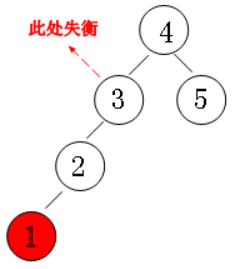


* + - 1. 再平衡

接下来我们来看一看插入结点时，如何平衡二叉平衡树。

平衡二叉树不平衡的情形可以分为以下4种：

1. 左左情况(LL)**左子树的左节点失衡**



在向树种添加1节点时，左子树的左边节点失衡。

解决方法：

1. **右右情况(RR)**
2. **左右情况(LR)**
3. **右左情况(RL)**
   * 1. 红黑树
        1. 定义

首先，红黑树是一棵二叉查找树，它在每一个节点上增加了一个存储域用于表示当前节点的颜色。它满足所有二叉查找树的条件，除此之外，他还要满足以下条件：

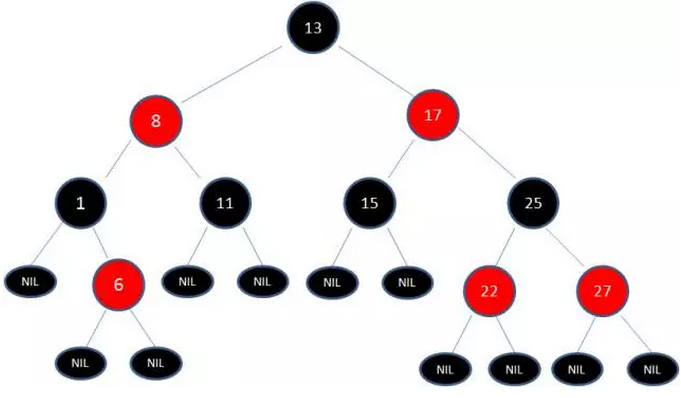
1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

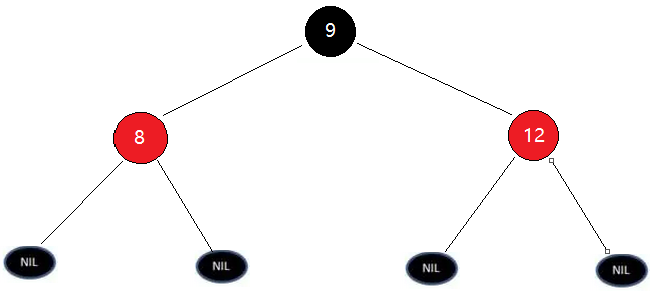
5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



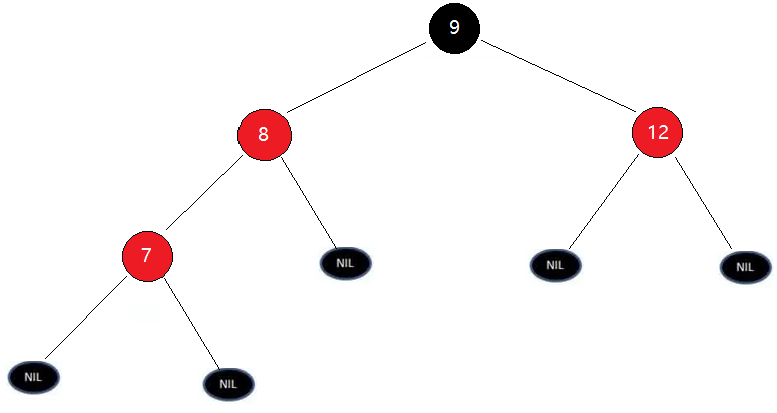
通过第四、五点的约束，我们可以知道，从根节点到叶子节点的所有路径中，最长路径是红黑交替的路径，最短路径是全黑的路径，因此**从根到叶子的所有路径中，没有哪条路径会比别的路径长出2倍。所以我们可以说红黑树是近似平衡的**

再回到我们上面的二叉搜索树的最差情况，

首先是8、9、12，然后依次向其中加入7，6，5，4，3

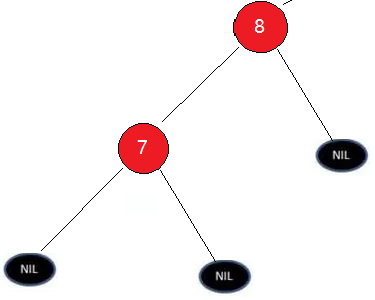
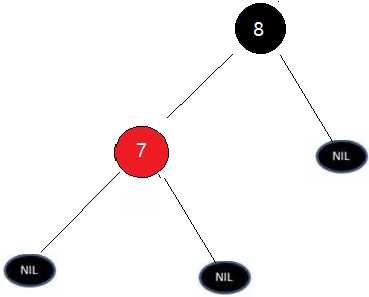


7应当在8的左边，为了保证，9经过8，7到叶子节点只有两个黑色节点(第5条约束)，因此新加入了7应当是红色的，但这又与第4条约束向矛盾(不能有两个相连的红色节点)，因此我们必须对红黑树进行调整。调整的方式有很多，主要分为变色和旋转，而旋转又可以分为左旋转和右旋转(ps:一般来说都是直接把插入的节点染成红色，这样只要满足4即可)

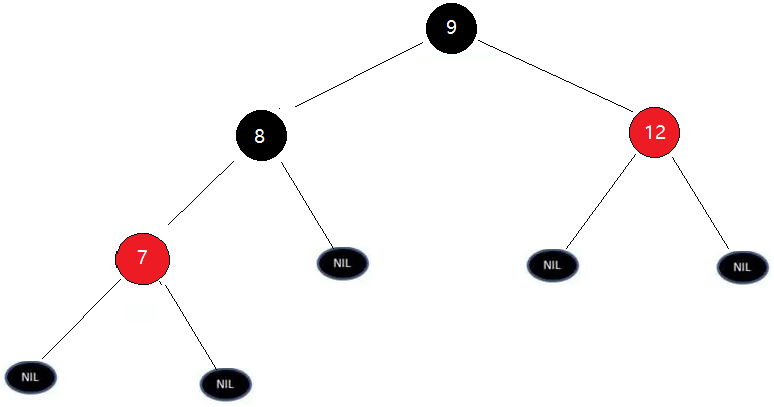
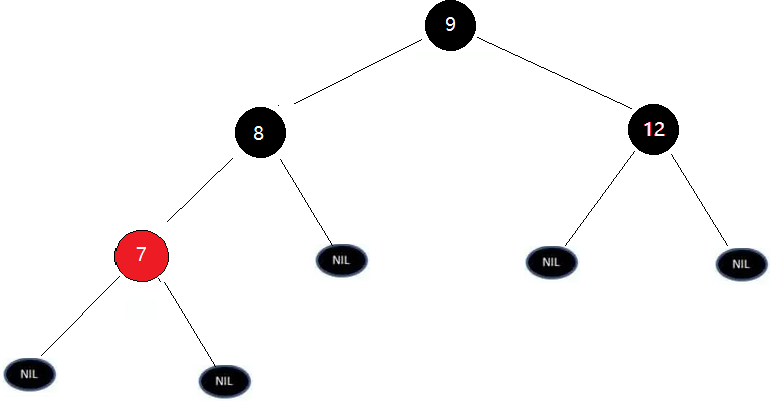


* + - 1. 变色

变色是为了维护红黑树的结构，尝试将节点颜色进行转变的一种操作

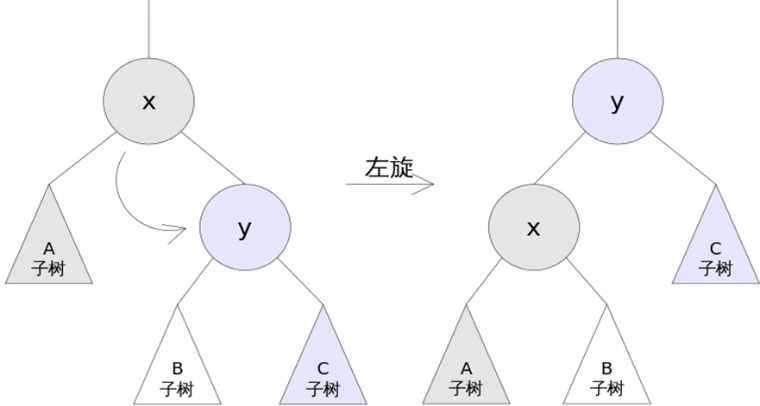
 转变为 

但是这样一来，又违背了第5条约束，从根节点到叶子节点出现了3个黑色节点，出现了连锁变化，因此还需要修改树节点的其他地方

转变为

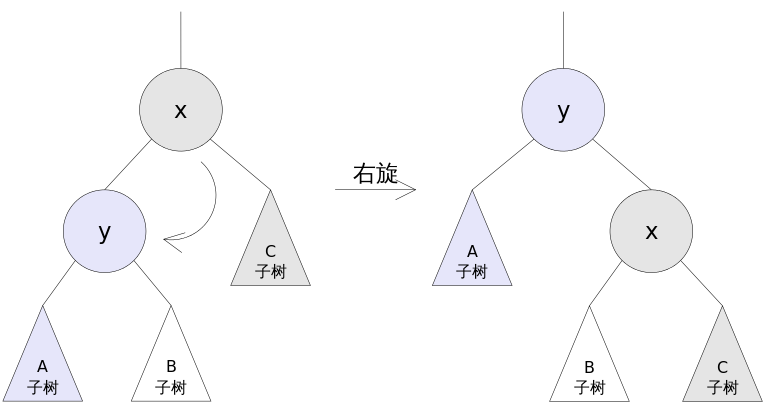
* + - 1. 左旋转

左旋转：逆时针旋转红黑树的两个节点——父节点x和右子节点y，x被y取代，y成为父节点，x成为y的左子节点，同时，y原来的左子节点将变为x的右子节点



* + - 1. 右旋转

右旋转：顺时针旋转红黑树的两个节点——父节点x和左子节点y，x被y取代，y成为父节点，x成为y的右子节点，同时，y原来的右子节点将变为x的左子节点



* 1. 散列表

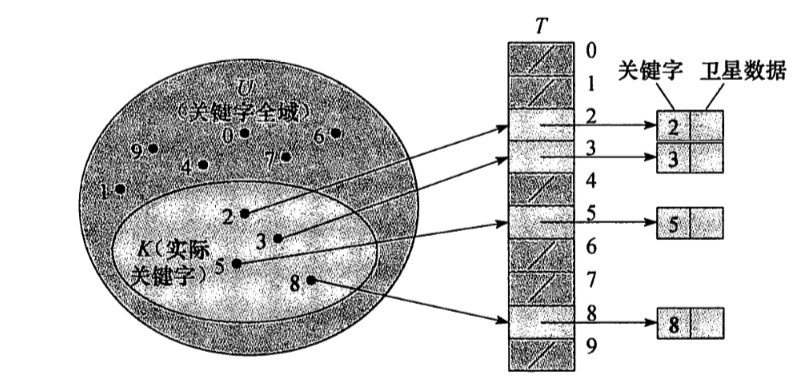
散列表又叫哈希表，是一种重要的数据结构，它根据关键码值(Key value)而直接进行访问

**定义：给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希(Hash）表，函数f(key)为哈希(Hash) 函数。**

举个例子，如编译原理的符号表，有一个变量标识key和变量值value，把Key通过一个固定的哈希函数处理，先转换成一个整型数字，然后将该数字用数组长度进行取余，取余结果(hash值)就当作数组的下标，将value存储在以该数字为下标的数组空间里。这个数组就是**hash表**，hash表的每个位置称为**槽**，这个函数就是**hash函数**，hash函数的输出结果就是**hash值**

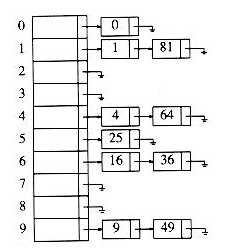
这样做可能导致散列表的大小小于hash函数输入的数量，不同的函数输入可能会散列成相同的下标输出，也即是两个关键字key的hash值f(key)可能相等，这种情形被称为**冲突**，后面将会介绍如何通过***链接法***和***开放寻址法***解决冲突

在关键字key的全集U较小时，使用直接寻址来读取和存储数据较为简单，如一个规模较小的数组，但当key的全集U较大时，这样做显然浪费空间甚至难以实现，如U大于计算机的内存时

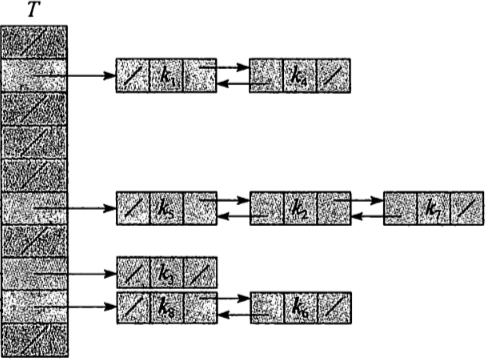


* + 1. 链接法

链接法把散列到同一个槽上的所有元素放到一个链表中，散列表的每一个槽都只存储指针，如果该槽上有元素，则槽中存储的指针就指向存储该槽上元素的链表，若该槽上没有元素，则槽中指针为空，如下图，关键字集合为(0,1,4,9,16,25,36,49,64,81)

****

**为了使散列表能够更快地删除某个键值，最好将其上的链表设为如下图的双向链表**

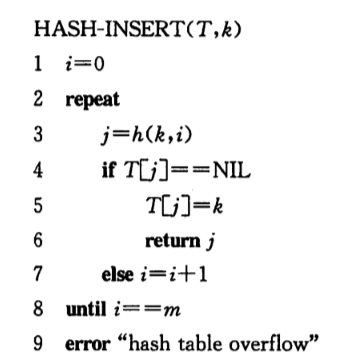


**这样查找前驱元素较快，可以更快地修改前驱元素的next指针**

* + 1. 开放寻址法

在开放寻址法中，所有的元素都直接存储在散列表上，也就是说散列表数组的每一个表项要么存储key，要么为空，当查找某个元素时，需要检索整张表，直到查到对应元素，或最终查明元素不在散列表中

开放寻址法通过**连续检索**hash表直到为产生冲突的关键字找到一个空槽，这个过程称为**探查**，在探查的过程中可能产生多个非空的地址序列Hi(i=1,2…m)，若Hi-1槽非空，则再求下一个地址Hi，直到查找到一个空槽



i是探查的次数，j=h(k,i)是关键字k第i次探查的hash值(槽的地址)，m是散列表的长度

但是开放寻址法**删除操作较为麻烦**，不能简单地将槽中的key置空来删除数据，试想一下，我们在插入关键字k时，发现hash值H1的槽被占用，这时会继续寻找H2，H3….Hm,如果我们将H2的值置空了，就无法检索Hm了，解决方法是给散列表中的每个槽添加一个deleted位来标记该槽是否被删除。

**因此，在具有删除操作的hash表中，更多采用链接法来避免冲突**

* + 1. 散列函数

一个好的散列函数可以尽可能地避免冲突产生，多数散列函数都假设关键字key的全集是自然数N={0,1,2….}，如果key不是自然数，散列函数的第一步就应当选择一种方法先将其转化为自然数，下面假设所给的key都是自然数

* + - 1. 除法散列法

除法散列法通过取key除以某个数p**(p<=散列表长度m)**的余数，将关键字key散列到散列表的m个槽上，即：

f(key)=key mod p

需要注意的是，p应当尽量避免为2的整数幂，即p≠2e，因为如果这样做，余数就等于key二进制的最低e位数字，**最好选择一个不太靠近2的整数幂的素数**，如下图：

设p=2^3=8



* + - 1. 乘法散列法

1. 算法基础——排序
   1. 插入排序
      1. 基本思想

插入排序的基本思想是将一串数组从第一个数开始，依次取未排序区的第一位，然后与已排序区的数组的部分数进行比较，找到其位置，然后将该位置后的已排序区的数依次后移一位，并将该数填到对应的空缺位置。

8 2 9 3 4

2 8 9 3 4 8后移一位，将2放到第一位上

2 8 9 3 4

2 3 8 9 4 8和9后移一位，将3放到第二位上

2 3 4 8 9 8和9后移一位，将4放到第三位上

**代码如下：**

**public** **static** **int**[] insertSort(**int**[] array, **int** length) {

**int** seqLen = 0;// 有序区域的长度

**int** temp = 0;

**int** j = 0;

**boolean** founded = **false**;

**for** (**int** i = 1; i < length; i++, seqLen++) {// 从第二个数开始取值

founded = **false**;

**for** (j = 0; j < seqLen + 1; j++) {// 被选中的数与有序区进行比较

**if** (array[seqLen + 1] < array[j]) {

temp = array[seqLen + 1];

founded = **true**;// 防止出现位置不需调换的数出现

**break**;

}

}

**for** (**int** n = seqLen; n >= j; n--) {

array[n + 1] = array[n];

}

**if** (founded) {

array[j] = temp;

}

}

**return** array;

}

* + 1. 循环不变式

循环不变式通常可以帮助我们理解算法的正确性，它是指在一个正确的算法中始终存在的一个维持不变的特性，此特性一直保持到循环结束甚至算法结束，用以保证算法的正确性。

比如插入排序，其循环不变式为：当我们要插入下标为i的数前，A[0,……,i-1]是已排好序的；比如选择排序，在选择排序中，在选择第i个数进行排序时，A[0,……,i-2]是A中 小(大)的i-1个数，并且是有序的

**如何证明循环不变式**

我们必须证明三条性质。

初始化：循环的第一次迭代之前，循环不变式为真

保持：若循环的某次迭代前为真，那么下次循环迭代前循环不变式也为真

终止：当循环不变式终止时，不变式为我们提供了一个有用的性质，该性质有助于证明算法是正确的。

前两步类似于数学归纳法，但第三步不同，数学归纳法中，归纳步被无限使用，但第三步是循环终止，应停止”归纳”。

**对于插入排序的循环不变式证明**

初始化：当i=1时，已排好的区域只有一个元素，即A[0]，它是有序的，循环不变式为真

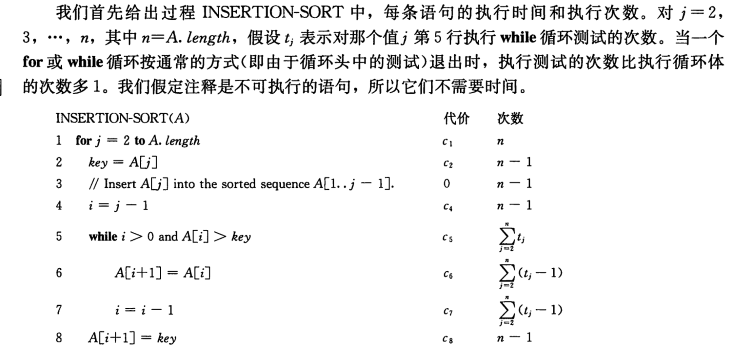
保持：需证明每次循环不变式迭代都能保持循环不变式为真，循环体每次循环都会将A[i-1],A[i-2]……向后移动一个位置变为A[i],A[i-1]……，直到找到A[i]的适当位置，再将A[i]插入，此时A[0……i]由原本的A[0……i-1]和A[i]共同组成，但已排好序，因此每次迭代都能保证循环不变式为真

终止：终止条件为i>A.length-1=n-1,因此结束时i=n-1，此时，即将插入下标为n的元素(不存在)，然后执行i++，i=n>A.length-1,循环终止，此时A[0……n-1]是有序的，因此整个数组有序，算法正确

* + 1. 算法分析

**影响插排运行时间的变量有：**

1. 输入数组的规模影响插排的运行时间，排序3个数永远快于排序3000个数；
2. 数组已被排序程度影响插排运行时间，若需被排列的数越少，插排查找位置和移动元素越少，排序速度越快

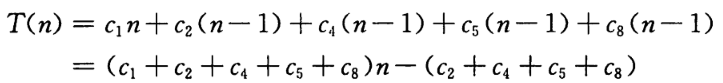


所以运行时间

T（n）= 

**最佳情况(所有的数按相应顺序排列)：**

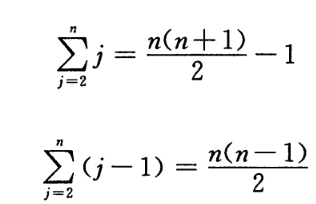
当最佳情况发生时，C5判断n-1次，C6和C7不消耗时间，则



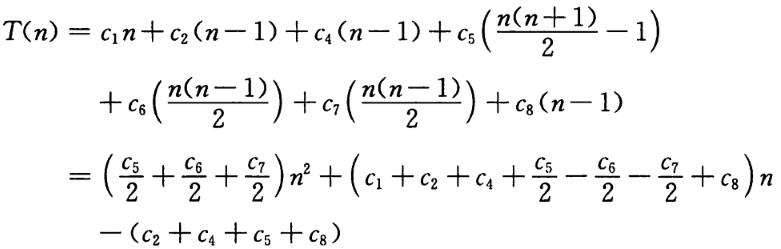
我们可以将其表示为T(n)=an+b;a和b是常数，取决于各条语句执行时间，因此，我们可以说，插排在最佳情况下是线性的；

**最坏情况(所有数按逆序排列)：**

当最坏情况发生时，C5判断了i+1=j次，C6和C7执行了j-1，又因为



则



我们可以将其表示为T(n)=an²+bn+c，因此，我们可以说，插排在最坏情况下的时间复杂度是一个二次函数

**一个算法的最坏情况给出了算法运行时间的上界**

**通常，我们使用增长量级O（）来表示运行时间的增长率（即随着输入规模扩大，运行时间增加的速度），我们通常忽略低阶项和常系数。**

如：插排最佳情况的增长量级为O(n)，最坏情况的增长量级为O(n²).，平均时间复杂度也为O(n²)

**在实际应用中，插排适用于输入规模小于50的数目的数组排序，插排是稳定排序，也是原址排序**

* + 1. 算法优化

在插入排序中我们可以使用二分法进行优化，为了排序A[0….n]，我们递归地排序A[0….n-1],再将A[n]插入已排序数组A[0….n-1]，我们在查找A[n]的位置时，可以使用二分查找，即每次用A[0…..n-1]的中点A[mid]与A[n]进行比较，每次将查找序列规模减半，相应的二分查找代码为：

//该函数返回插入位置的右边一位

**public** **static** **int** findLocation(**int**[]array,**int** begin,**int** end,**int** num){

**if**(end-begin>0){//递归终止条件

**int** mid=(end+begin)/2;

**if**(num<array[mid])

**return** *findLocation*(array,begin,mid,num);

**else**

**return** *findLocation*(array,mid+1,end,num);

}

**return** begin;

}

注意：该方法能有效防止最坏情况地出现，它将插入排序最坏情况地O(n²) 改进为O(nlgn)

* 1. 归并排序(采用了分治法)
     1. 主要思想

首先，当数组中只有一个数时，该数组是有序的。

将数组平均分为两部分，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。若二组组内的数据无序，那么就继续将其分为两部分，直到数组中的数都只有一个时，依次返回归并数组。

归并数组时，只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数组中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

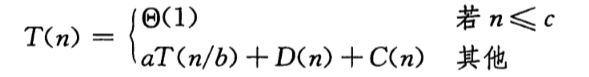
* + 1. 代码

public static int[] mergeSort(int[] array, int length) {  
 int newLen = length / 2;  
 int[] array\_2 = null;  
 int[] array\_1 = null;  
 if (length > 1) {  
 array\_1 = new int[newLen];  
 for (int i = 0; i < newLen; i++) {  
 array\_1[i] = array[i];  
 }  
 array\_2 = new int[length - newLen];  
 for (int i = 0; i < length - newLen; i++) {  
 array\_2[i] = array[i + newLen];  
 }  
 *mergeSort*(array\_1, newLen);  
 *mergeSort*(array\_2, length - newLen);  
 }  
 int firstIndex = 0;  
 int secondIndex = 0;  
 int index = 0;  
 while (firstIndex < newLen && secondIndex < length - newLen) {  
 if (array\_1[firstIndex] < array\_2[secondIndex]) {  
 array[index++] = array\_1[firstIndex++];  
 } else {  
 array[index++] = array\_2[secondIndex++];  
 }  
 }  
 while (firstIndex < newLen&&array\_1!=null) {  
 array[index++] = array\_1[firstIndex++];  
 }  
 while (secondIndex < length - newLen&&array\_2!=null){  
 array[index++] = array\_2[secondIndex++];  
 }  
 return array;  
}

* + 1. 算法分析

归并排序在空间复杂度上表现得十分不理想，因为它不是原址排序，它占用了大量非原址的空间，将一个整数组复制到很多非本数组的空间上去，但它的时间复杂度在数万数量级层次的数的排序上优于快速排序

分治法首先将问题分为a个子问题，每个问题的规模为原问题的1/b(归并排序中a=b=2)，因此我们可以得出分治法的时间复杂度递推式



其中，D(n)为分解问题需要的时间，C(n)为合并问题需要的时间

对于一个归并排序，有三个步骤：

分解：只需计算子数组的中间位置，需要常量时间，因此D(n)=O(1);

解决：递归求解2个规模为n/2的子问题，需2T(n/2)的运行时间

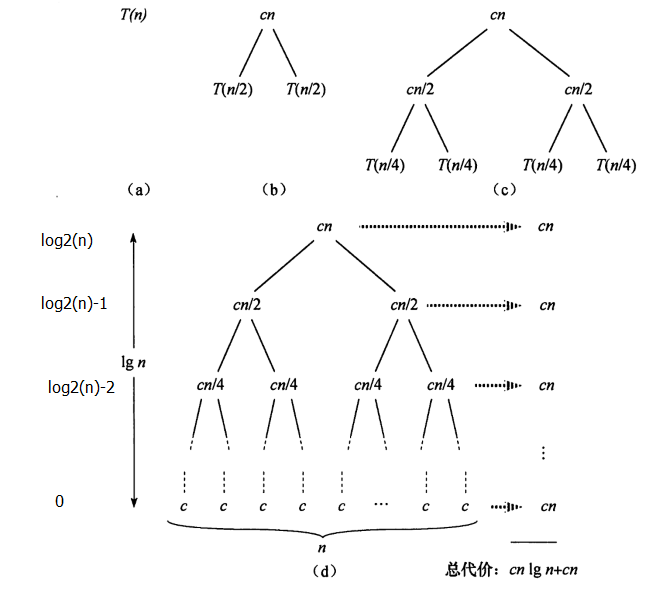
合并：MERGE算法需要O(n)的时间，因此C(n)=O(n);

D(n)和C(n)相加，和是O(n)，因此T(n)可简化为

T(n)=2T(n/2)+O(n) n>1

T(n)=2T(n/2)+cn n>1

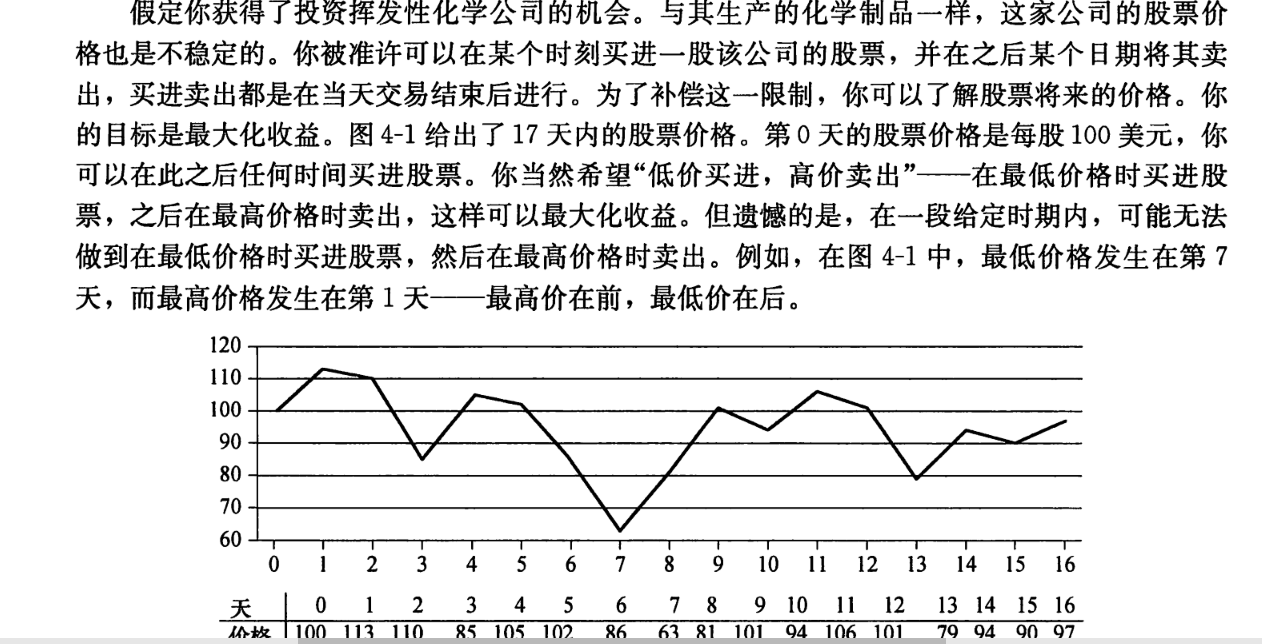
将T(n)分解为递归树，不断扩展，直到问题规模下降到1



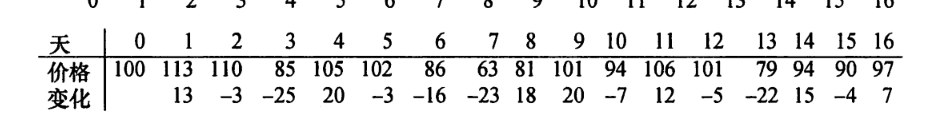
因此耗时T(n)=cn\*lgn+cn，因此O(n)=nlgn

* + 1. 求最大子数组问题(分治法,也称递归)

问题背景：



我们可以通过观察价格的变化来解决问题



这样问题就变成了求数组{13,-3,-25,20,-3,-16,-23,18,20,-7,12,-5,-22,15,-4,7}的最大连续子数组和问题，这时再通过暴力破解问题就变得简单得多，比如求所有以第一天为开始的连续数组和时可以依次往后加上当天的变化值，如：第1天~第2天为10，第1天~第3天为10-25=-15。

那么我们能不能通过分治法进行相关的简化呢？当然可以，首先，我们先来思考一下，如何将问题划分为更小规模的问题，如果我们取数组A[low…….high]的中点为mid，那么所有的最大连续数组和出现的情况可以分为三种：

1. 处于A[low…..mid]之间
2. 处于A[mid……high]之间
3. 横跨两个区间，即取得最大连续子数组和的数组为A[i…..mid……j]（low≤i<mid，mid≤high）

这样，问题就被我们划分为三种情况，前两种都可以划分为更小的问题，就可以进行递归取值，最后再将结果与第三种情况的结果进行比较，整个问题的最终结果必为三种情况中的最大者

第三种情况又可以分解，第三种情况的子数组可以分为两部分，一部分是A[i….mid],另一部分是A[mid…..j],当两部分的和都为最大时，A[i….j]的和才取到最大。

这样，整个问题就迎刃而解

代码如下：

**暴力求解：**

**public** **static** **int** MaxSubSum1(**int**[] arr, **int** len) {

**int** i, j;

**int** MaxSum = 0;

// 每次开始累加的起始位置的循环

**for** (i = 0; i < len; i++) {

**int** CurSum = 0;

// 向后累加的循环

**for** (j = i; j < len; j++) {

CurSum += arr[j];

**if** (CurSum > MaxSum)

MaxSum = CurSum;

}

}

**return** MaxSum;

}

**分治法：**

**public** **static** **int**[] getMaxChildArray(**int**[] array, **int** low, **int** high) {

**if** (high - low == 0) {

**return** **new** **int**[] { low, high, array[low] };

} **else** {

**int**[] array\_1 = **new** **int**[3];// 储存位于low~mid之间的子数组，第一个数是起始位置，第二个数是结束位置，第三个数是子数组和

**int**[] array\_2 = **new** **int**[3];

**int**[] array\_3 = **new** **int**[3];

**int** mid = (high + low) / 2;

array\_1 = *getMaxChildArray*(array, low, mid);

array\_2 = *getMaxChildArray*(array, mid + 1, high);

//求跨越中点的最大连续子数组

**if** (high - low >= 1) {

**int** sum=array[mid];

**int** sum\_left=sum;

**int** left=mid;

**int** right=mid+1;

//求左半边最大值，必须与mid紧连

**for**(**int** i=mid-1;i>=low;i--){

sum+=array[i];

**if**(sum>sum\_left){

sum\_left=sum;

left=i;

}

}

sum\_left+=array[mid+1];

sum=sum\_left;

**for**(**int** i=mid+2;i<=high;i++){

sum+=array[i];

**if**(sum>sum\_left){

sum\_left=sum;

right=i;

}

}

array\_3[0]=left;

array\_3[1]=right;

array\_3[2]=sum\_left;

}

// 第一种情况，若最大连续子数组在前半段

**if** (array\_1[2] > array\_2[2] && array\_1[2] > array\_3[2]) {

**return** array\_1;

}

// 第二种情况，若最大连续子数组在后半段

**else** **if** (array\_2[2] > array\_1[2] && array\_2[2] > array\_3[2])

**return** array\_2;

// 第三种情况，若最大连续子数组跨越中点

**else**

**return** array\_3;

}

}

由此，我们得到一个T(n)=2T(n/2)+θ(n)即T(n)= θ(nlogn)的归并算法

* 1. 堆(完全二叉树)排序
     1. 完全二叉树的性质

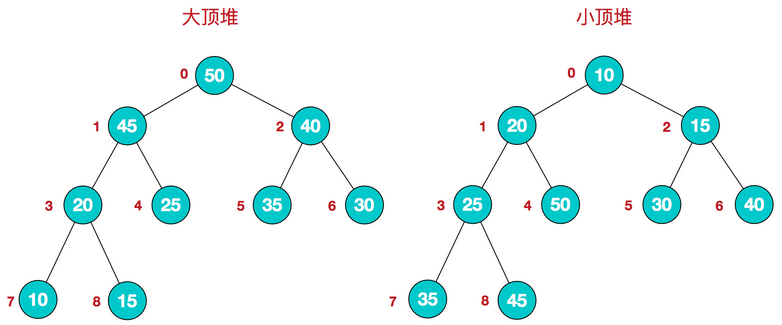
1.对于一个共有n个结点的树，其高度为**⌊**logn**⌋**+1( 4.9调用用向下取整函数得到的是 4)；

2.对于根结点索引为1的树，其i结点的左孩子是2\*i，右孩子是2\*i+1；

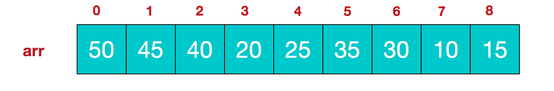
3.对于一个共有n个结点的树，其索引为**⌊**n/2**⌋+1**的结点都为叶子结点

4.对于任何一个二叉树，设度为0的结点(叶子结点)数为n0，度为2的结点数为n2，则n0=n2+1，因此，叶子结点从**⌊**n/2**⌋+1**开始;

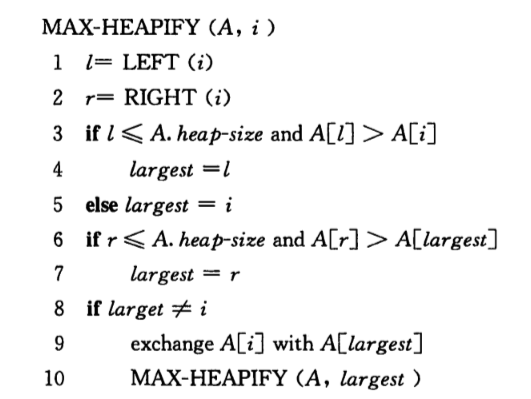
**堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆。**



同时，我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子



* + 1. 维护堆的性质

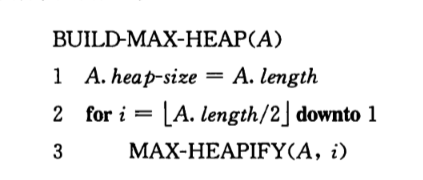


第10步的递归调用是防止置换下来的节点(largest)会导致以该节点为根节点的子树违反最大堆的性质

**源代码：**

public static void keepMaxHeap(int[] array, int i) {  
 int left = 2 \* i;  
 int right = 2 \* i + 1;  
 int largest = i;  
 int temp;  
 if (left <= heap.heapSize && array[largest - 1] < array[left - 1]) {  
 largest = left;  
 }  
 if (right <= heap.heapSize && array[largest - 1] < array[right - 1]) {  
 largest = right;  
 }  
 if (largest != i) {  
 temp = array[largest - 1];  
 array[largest - 1] = array[i - 1];  
 array[i - 1] = temp;  
 *keepMaxHeap*(array, largest);  
 }  
}

* + 1. 建堆



**源代码：**

**public** **static** **void** buildMaxHeap(**int**[] array){

*heap*.heapSize=array.length;

**for**(**int** i=array.length/2;i>0;i--){

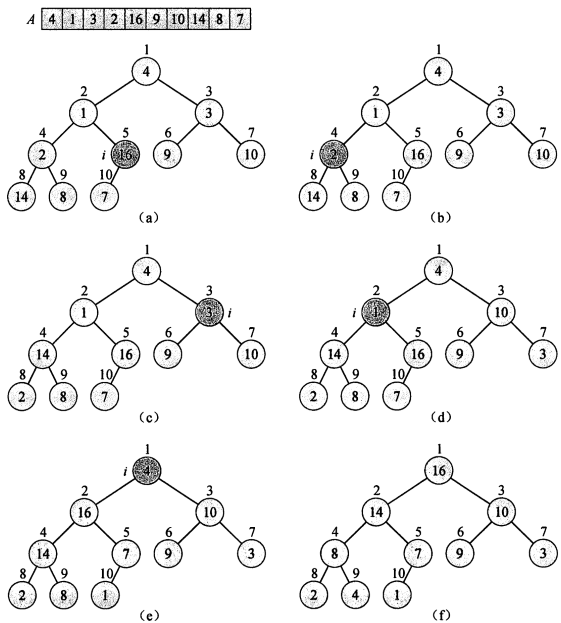
*keepMaxHeap*(array,i);

}

}

此处一定要从下往上（从[A.length/2]到1），因为维护堆性质的函数MAX-HEAPIFY每次运行只能保持本次运行以i为根节点的子树符合最大堆性质，若从下至上，首先每个子树符合最大堆性质后，再执行MAX-HEAPIFY使两棵子树合并的更大的子树符合性质，如果是从上至下，则无法保证其实现

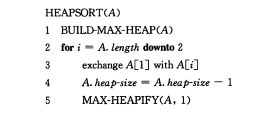
此方法可以在O(n)的线性时间里把一个无序的数组构建成一个最大堆



* + 1. 排序

每次都将最大堆的根与树的最后一个节点交换，然后将堆的大小减一，重复操作即可实现堆排序。

通过最大堆进行堆排序可由小到大进行排序，反之可由大到小进行排序



**源代码：**

**class** Heap{

**public** **int** heapSize;

**int**[] array;

}

**public** **static** **void** heapSort(**int**[] array){

**int** temp;

**for**(**int** i=*heap*.heapSize;i>1;i--){

temp=array[0];

array[0]=array[i-1];

array[i-1]=temp;

*heap*.heapSize--;

*keepMaxHeap*(array, 1);

}

}

堆排序的时间复杂度为O (nlgn)

* 1. 快速排序

快速排序一般是实际排序中最好的选择，它是一种原址排序，平均性能较好

* + 1. 主要思想

**分解：**将数组A[p…r]划分为两个可能为空的数组**A[p…q-1]<A[q]<A[q+1…r]**，q可以任取

**解决**：通过递归，对 A[p…q-1]和A[q+1…r]也进行分解排序

**合并：**因为子数组是原址的，不需要进行合并，所有的递归触底后，A[p…r]就是有序的

取一个数作为基准数(q)，比基准数大的放右边，小的放左边

其时间复杂度为O (nlgn)

* + 1. 代码

**public** **static** **void** quickSort(**int**[] arr, **int** begin, **int** end) {

**if** (end - begin > 0) {

**int** holeIndex = begin;

**int** temp = arr[begin];

**int** bIndex = begin + 1;

**int** eIndex = end;

**while** (bIndex < eIndex) {

**while** (bIndex < eIndex && temp <= arr[eIndex]) {

eIndex--;

}

**if** (bIndex < eIndex) {

arr[holeIndex] = arr[eIndex];

holeIndex = eIndex;

}

**while** (bIndex < eIndex && temp > arr[bIndex]) {

bIndex++;

}

**if** (bIndex < eIndex) {

arr[holeIndex] = arr[bIndex];

holeIndex = bIndex;

}

}

arr[holeIndex] = temp;

*quickSort*(arr, begin, holeIndex - 1);

*quickSort*(arr, holeIndex + 1, end);

}

}

* + 1. 代码优化

在分解阶段，我们可以通过“挖坑填数”的方法对算法进行优化

以一个数组作为示例，取区间第一个数为基准数。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 72 | 6 | 57 | 88 | 60 | 42 | 83 | 73 | 48 | 85 |

初始时，bIndex = 1;  eIndex = 9;   temp= 72; holeIndex=0;

由于已经将a[0]中的数保存到temp中，可以理解成在数组a[0]上挖了个坑，可以将其它数据填充到这来。

从eIndex开始向前找一个比temp小或等于的数。当eIndex=8，符合条件，将a[8]挖出再填到上一个坑a[0]中。a[0]=a[8];  这样一个坑a[0]就被搞定了，但又形成了一个新坑a[8]，接着再找数字来填a[8]这个坑。这次从bIndex开始向后找一个大于temp的数，当bIndex=3，符合条件，将a[3]挖出再填到上一个坑中a[8]=a[3];

数组变为：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 48 | 6 | 57 | 88 | 60 | 42 | 83 | 73 | 88 | 85 |

bIndex = 3;   eIndex = 8;  temp =72; holeIndex=3;

再重复上面的步骤，先从后向前找，再从前向后找。直到bIndex>=eIndex 为止

数组变为：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 48 | 6 | 57 | 42 | 60 | 72 | 83 | 73 | 88 | 85 |

4.2的代码已经过优化处理

* 1. 顺序统计量和中位数问题

第i个顺序统计量是指将一组数由小到大排序后的第i个数，一般情况下查找顺序统计量采用先排序再查找的方式，但本节将会有一些更快的算法

* + 1. 最大值最小值问题

在一般的算法中，查找n个元素的最大值最小值都需要n-1次比较，只需依次将n-1个数和最大(小)的数比较即可，这样看来，同时查找n个元素的最大值最小值似乎最少需要2n-2次比较，但事实上，我们有渐进更优的算法来解决这个问题，我们只需要3n/2次比较就可以完成操作

若n为奇数，先取n个元素中的第一个数作为最大值和最小值，然后每次取两个数，先比较这两个数大小，然后大的与最大值比较，小的与最小值比较，直至比较结束，一共比较3n/2次，若n为偶数，则直接取两个数比较，一个作为最大值，一个作为最小值，然后按上述步骤继续执行，一共比较(3n/2)-2次，相比较来说，该算法是渐进最优的

* + 1. 求顺序统计量问题

我们需要借助快速排序中的分割思想进行分割，可以在快速排序的”挖坑填数”思想上进行改进，先通过”挖坑填数”将n个元素分为**arr[0…mid-1] <arr[mid]<arr[mid +1…n]**，然后查看分割点前面有几个数，即最后一次挖的坑**mid**前面有几个数，如果**mid**前面的数的个数等于i，则返回arr[i],伪代码如下

**getOrderStatistics(int[] arr,int begin, int end, int i ){**

**if begin==end**

**return arr[begin];**

**mid=divide(arr,begin,end);//将数组用”挖坑填数”法划分为arr[0…mid-1] <arr[mid]<arr[mid +1…n]**

**preLength=mid-begin+1;//前半段长度**

**if preLength ==i**

**return arr[mid];**

**else if preLength>i**

**getOrderStatistics(arr,begin,mid-1,i)**

**else**

**getOrderStatistics(arr,mid+1,end,i-preLength);**

**}**

代码如下：

**public** **static** **int** divide(**int**[] arr, **int** begin, **int** end) {

**int** holeIndex = begin;

**int** temp = arr[begin];

**int** bIndex = begin + 1;

**int** eIndex = end;

**while** (bIndex < eIndex) {

**while** (bIndex < eIndex && temp <= arr[eIndex]) {

eIndex--;

}

**if** (bIndex < eIndex) {

arr[holeIndex] = arr[eIndex];

holeIndex = eIndex;

}

**while** (bIndex < eIndex && temp > arr[bIndex]) {

bIndex++;

}

**if** (bIndex < eIndex) {

arr[holeIndex] = arr[bIndex];

holeIndex = bIndex;

}

}

arr[holeIndex] = temp;

**return** holeIndex;

}

**public** **static** **int** getOrderStatistics(**int**[] arr, **int** begin, **int** end,**int** i) {

**if**(end==begin){

**return** arr[begin];

}

**int** mid=*divide*(arr,begin,end);

**int** preLength=mid-begin+1;

**if**(preLength==i){

**return** arr[mid];

}**else** **if**(preLength>i){

**return** *getOrderStatistics*(arr,begin,mid-1,i);

}**else**

**return** *getOrderStatistics*(arr,mid+1,end,i-preLength);

}

1. 常见算法
   1. 动态规划算法

动态规划算法和分治法很类似，他们都是通过组合子问题的解来求解原问题。但是不同的是，分治法将原问题划分为若干个互不相交的子问题，递归求解再组合。而动态规划则应用于子问题重叠的情况，即不同的子问题具有公共的子子问题，如果这种情形下使用分治法，只会造成不必要的重复计算

* + 1. 主要思想

* 1. 贪心算法

贪心算法又称贪婪算法