节选自清华计算机考研912真题（2016-2020年）数据结构部分，主要按照邓书对知识点的分类形式对这部分考题做了简单的分类，并配有本人相应解答，仅供参考，如有疑问，可以在文章下方留言。

**一、大O 渐进时间复杂度**

**(logn)^n= θ(n^logn)** (2020年)

* 正确
* 错误

错误 ❌

等式两边同时取log：左边=n×log(logn)，右边=logn×logn；

从渐进复杂度的层次来看：，即有左边>右边。



**n^(logloglogn)=O(⌊logn⌋!)** (2019年)

* 正确
* 错误

正确 ✔

对 n^(logloglogn) 和 ⌊logn⌋! 两边取对数

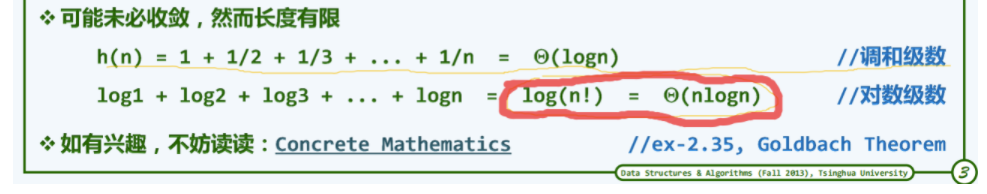
左边：logn × (logloglogn) ， 右边 ： log(⌊logn⌋!)

左边 < logn × (loglogn) ， 右边 ： log(⌊logn⌋!) = Θ(logn ×loglogn)

故 左边 < 右边

故 n^(logloglogn)=O(⌊logn⌋!)





**T(n)=a>0，无论常数a多大，时间复杂度为T(n)=T(n/2)+O(1)的解总是O(logn) 。** (2018年)

* 正确
* 错误

正确 ✔

T(n)=T(n/2)+O(1) 就是二分查找的递推式，二分查找的复杂度就是O(logn)。

猜测原题应该是T(0)=a。

另外 T(n)=2×T(n/2)+O(n) = O(nlogn) ，典型的如归并排序。

**若f(n)=时间复杂度O(g(n)),也不一定有f(n)=O(g(n-1))。** (2017年)

* 正确
* 错误

正确 ✔

反例：

设 f(n)= (n-1)^n，g(n) = n^n 则 g(n-1) = (n-1)^(n-1)

f(n) **<=** g(n) ，f(n) = O(g(n)) 符合题意

但是 g(n-1)= (n-1)^(n-1) **<=** f(n)= (n-1)^n ,即 g(n-1) = O(f(n))

所以不一定都是 f(n) <= g(n-1)，f(n)=O(g(n-1)) ，如这个反例

**二、排序**

**冒泡排序**

**在进行冒泡排序时，有可能出现某些元素在排序过程中一直远离它的最终位置。** (2016年)

* 正确
* 错误

正确 ✔

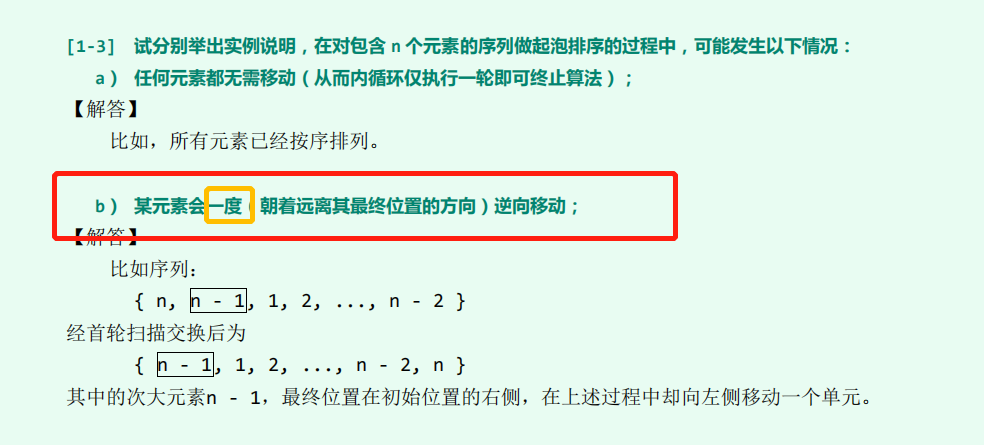
这里的“一直”表达的意思应该是排序过程中持续远离最终位置的情况，但是随着排序的推进，这些元素肯定要逐渐接近最终位置。

比如序列 {7 6 5 4 3 2 1 0}

第一趟冒泡后变为 { 6 5 4 3 2 1 0 7}

第二趟冒泡后变为 { 5 4 3 2 1 0 6 7}

元素5从位置2变为位置1，然后变为位置0，往左远离了它的最终位置。



**插入排序**

**插入排序每次插入数据，即使不增加循环节，也不至减少。**(2019年)

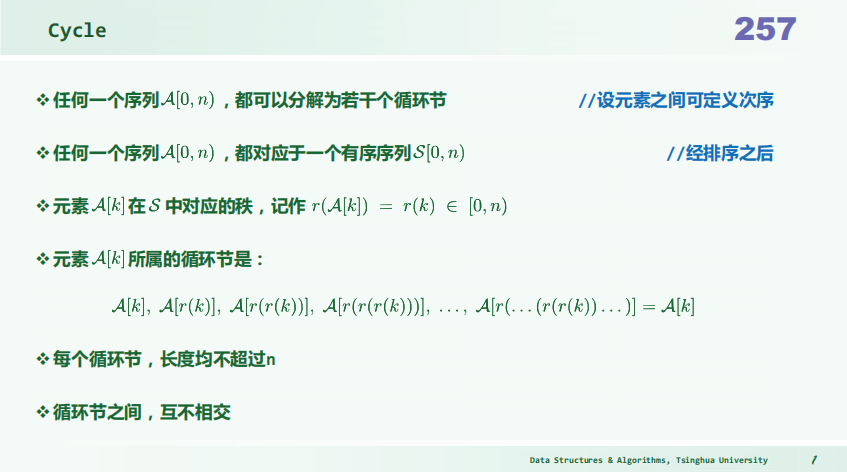
* 正确
* 错误

错误 ❌

题目表述的意思是插入排序每次操作后循环节数不变。

但是实际上，插入排序过程中循环节个数可能减少，也可能增加，反例见下👇

首先循环节定义如下：



现按照定义计算序列 A[6] = {6，5，4，3，2，1} 在插入排序过程中的循环节个数变化

初始时循环节{6，1}，{5，2}，{4，3}，共3个

第一趟插入排序：第一个元素6就位，循环节数目不变。

第二趟插入排序，元素5就位

第二趟循环节{5，2，6，1}和{4，3}，循环节数目共2个，相对于上一趟循环节数目减少。

..............中间省略

最后一趟排序完成

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rank | 0 | 1 | 2 | 3 | 4 | 5 |
| A[ ]：原始序列 | 6 | 5 | 4 | 3 | 2 | 1 |
| S[ ]：最终有序序列 | 1 | 2 | 3 | 4 | 5 | 6 |
| r[ ]：原序列元素在S中的秩 | 5 | 4 | 3 | 2 | 1 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rank | 0 | 1 | 2 | 3 | 4 | 5 |
| A[ ]第二趟 | 5 | 6 | 4 | 3 | 2 | 1 |
| S[ ]：最终有序序列 | 1 | 2 | 3 | 4 | 5 | 6 |
| r[ ]：原序列元素在S中的秩 | 4 | 5 | 3 | 2 | 1 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| rank | 0 | 1 | 2 | 3 | 4 | 5 |
| A[ ]最后一趟 | 1 | 2 | 3 | 4 | 5 | 6 |
| S[ ]：最终有序序列 | 1 | 2 | 3 | 4 | 5 | 6 |
| r[ ]：原序列元素在S中的秩 | 0 | 1 | 2 | 3 | 4 | 5 |

最后一趟循环节 {1}，{2}，{3}，{4}，{5}，{6}，循环节数目共6个，循环节数目增加。

可以看到在上述的插入排序过程中，循环节数目有不变，有减少，也有增加。

**交换某个逆序对中的两个元素，必然会减少总逆序对数。** （2019年）

* 正确
* 错误

正确 ✔

交换某个逆序对中的两个元素，就是减少了一对逆序对。随着逆序对的减少直至等于0，乱序序列也最终有序。

**简答：相比选择排序，插入排序的优势是什么，说明两点。**(2019年)

答：

1.插入排序对输入敏感，选择排序不具有此特点

2.插入排序属于在线算法

3.插入排序最好时间复杂度为O(n)，而选择排序为O(n^2)

**序列（64，63，...，2，1）进行直接插入排序比较次数最接近于（ ）** (2017年)

A.2800   B.2600  C.2400   D.2200   E.2000

选 E

根据【习题解析 3-12】，设有一个哨兵 -∞，所以第一个元素的比较次数从1开始

元素 64比较1次

元素 63比较2次

元素 62比较3次

.....

元素 1 比较64次

1+2 + 3 +......+ 63 = 64×65/2 = 2080

**折半插入算法在寻找插入的位置时，采用的是二分查找，因此整个折半插入算法的时间复杂度为O(nlogn)。** (2016年)

* 正确
* 错误

错误 ❌

寻找插入的位置是O(logn)，但是找到位置后要将此位置及其后的元素往后挪动一个位置，再将元素插入到该位置需要O(n)，所以即使折半插入算法的时间复杂度也是O(n^2)。

**基数排序**

**若底层排序算法不稳定，采用基数排序算法后（ ）**(2020年)

A.未必正确并且不稳定

B.正确并且不稳定

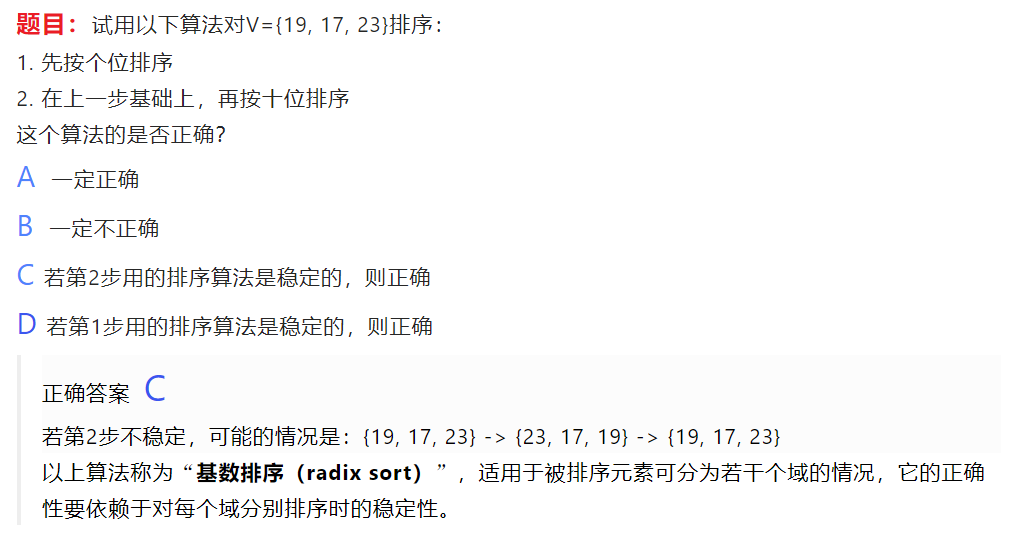
C.未必正确并且稳定

D.正确并且稳定

选 A

基数排序的正确性依赖于对每个域分别排序时的稳定性，只有每个域的排序算法都是稳定的，才能保证最后的基数排序是正确且稳定的，否则就不一定正确，更何其谈稳定性呢。

MOOC上的一道题。



**如果基数排序底层采用不稳定的算法，那么得到的结果可能是不正确的。**(2019年)

* 正确
* 错误

正确 ✔

如上题所说

基数排序的正确性依赖于对每个域分别排序时的稳定性，只有每个域的排序算法都是稳定的，才能保证最后的基数排序是正确且稳定的，否则就不一定正确。

**基数排序的底层排序算法一定是稳定的。** (2018年)

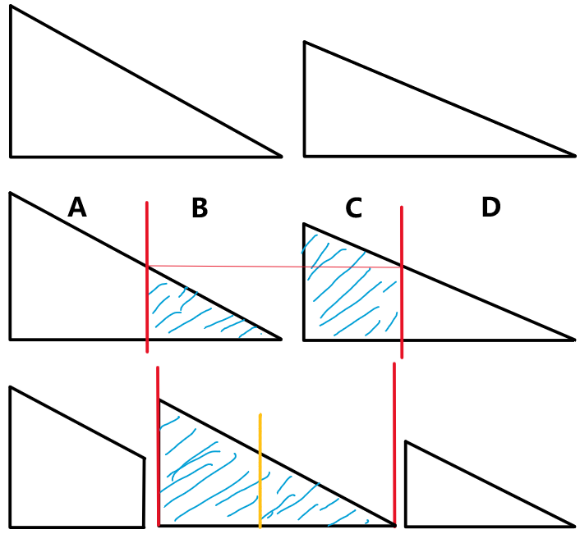
* 正确
* 错误

正确 ✔

如果底层算法不是稳定的，就不一定能正确排序了，就不会叫基数排序了。

**归并排序**

如图所示。假设已有两个有序的子序列。(2017年)



改进的归并策略为：

* 将两个子序列分别进行分割为四个子序列，使得两个序列在割点处的值相等。（似乎也可能是确定割点为1/2或1/3处）
* 直接拼接中间的两个序列（即交换它们的位置），从而得到3个有序子序列。
* 对这3个子序列，先归并1、2，用得到的结果再归并3，从而完成整个策略。

1）填空 merge

2）对 ABCDE 处的注释补充

3）rotate（）

4）说明这种算法的优缺点

**快速排序**

**快速排序的时间复杂度在平均情况下为O(nlogn)，最好情况亦如此。** （2020年）

* 正确
* 错误

正确 ✔

快速排序时间复杂度最好和平均都是O(nlogn)，最差是O(n^2)。

另外空间复杂度最好是O(logn)，最差是O(n)。

**基于比较式的算法可以在O(n)时间内在任意n个无序整数中找出前10%**（2020年）

* 正确
* 错误

正确 ✔

根据邓书12.2.6的【k-选取算法】可以在O(n)时间内完成k-选取。本题中k = 10%。

**二分查找**

**任何情况下折半查找都比顺序查找快** (2018年)

* 正确
* 错误

错误 ❌

如要查找的元素就是数组的首元素，顺序查找只需O(1)，而折半要O(logn)。

**算法题**

**单峰向量(13')** (2018年)

已知A[0,n ), A[0~k)严格单调递增，A[k~n)严格单调递减，设计一个O(logn)算法找出k

1)伪代码描述算法

2)说明算法正确性

3)证明最坏情况下时间复杂度也是O(logn)

1）二分查找

#include <iostream>

using namespace std;

int searchK(int A[],int lo,int hi) {

int left = lo,right = hi;

while(lo < hi ){

int mid = (lo + hi) / 2 ;

if(mid == left) return lo;

if(mid + 1 == right) return mid;

if( A[mid-1] < A[mid] && A[mid] > A[mid + 1])

return mid;

else if(A[mid-1] < A[mid] && A[mid] < A[mid + 1]) //A[mid]和左右按升序

lo = mid;

else if(A[mid+1] < A[mid] && A[mid] < A[mid - 1]) //A[mid]和左右按降序

hi = mid;

}

}

int main(){

int a1[] = {1,2,3,4,5,6,7};

int a2[] = {7,6,5,4,3,2,1};

int a3[] = {1,2,3,4,7,6,5};

int k1 = searchK(a1, 0, 7);

cout<<"k1="<<k1<<endl;

int k2 = searchK(a2, 0, 7);

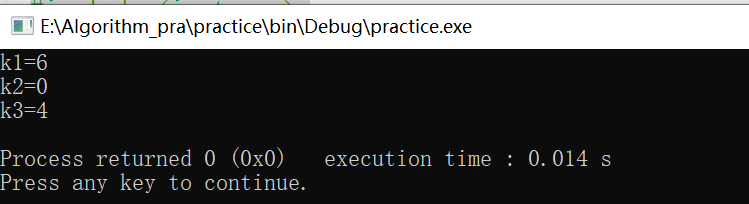
cout<<"k2="<<k2<<endl;

int k3 = searchK(a3, 0, 7);

cout<<"k3="<<k3<<endl;

return 0;

}



2) 算法的主要思想是二分查找

每次用A[mid]和左右两边比较，如果A[mid]是三者中最大，那么mid就是我们要找的k

如果A[mid]和左右按升序，那么查找范围就缩小到原序列的右边，去掉了左半边，缩减一半

如果A[mid]和左右按降，那么查找范围就缩小到原序列的左边，去掉了右半边，缩减一半

3）每次查找都会降查找范围缩小为原来的一半，故最坏不过O(logn)

**其他**

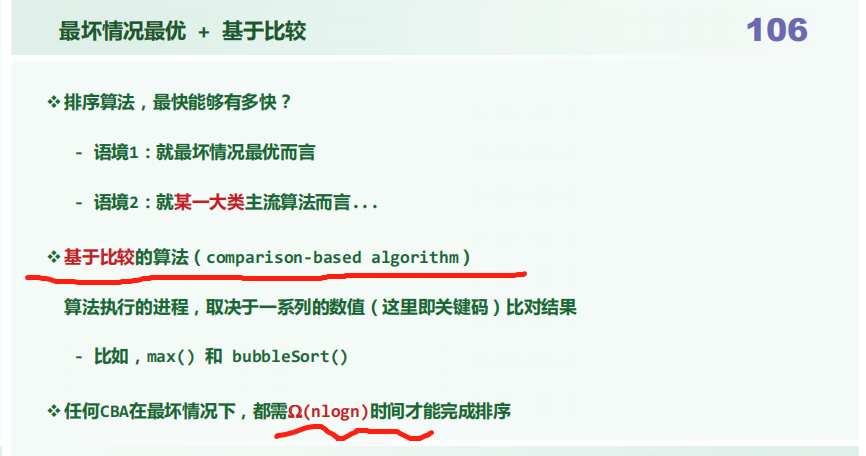
**基于CBA的算法对所有大小为n的数组时间复杂度是Ω(nlogn)** (2018年)

* 正确
* 错误

错误 ❌

题目里没有说最坏情况下，任何CBA在最坏情况下，都需要Ω(nlogn)时间才能完成排序。

但是如果序列原本就是有序的，冒泡、插入排序对原本就有序的序列排序，时间是 O(N)。



**就地算法的空间复杂度是** (2018年)

 A.O(1)  B.  C.  D.

选A ，就地算法的空间复杂度就是O(1)。

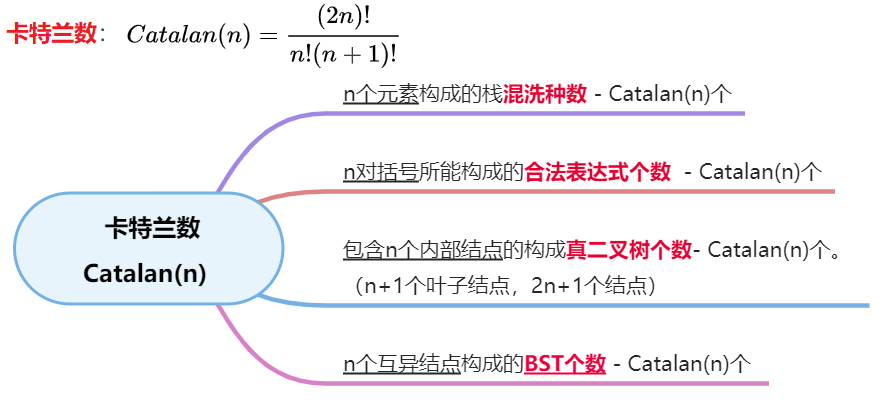
**三、树**

**卡特兰数**

**\_个无差别节点构成的真二叉树，与由2019对括号构成的合法表达式一样多（ ）**(2020年)

A.2018     B.2019     C.4038     D.4039

选D



故本题：

2019对括号构成的合法表达式个数为Catalan(2019)。

包含2019个内部结点，2020个叶子结点，也就是4039个结点的真二叉树的个数为Catalan(2019)。

**叶节点为2019的真二叉树数量小于2018对括号所组成的合法表达式数量** (2019年)

* 正确
* 错误

错误，不是小于，而是等于。

叶节点数为 2019，内部结点数为2018个，对应的真二叉树个数为Catalan(2018)。

2018 对括号所组成的合法表达式个数为Catalan(2018) 。所以两者相等。

**五个互异节点构造的二叉搜索树有多少种？** (2017年)

A.  B.  C.  D.

n个互异结点构成的BST有 Catalan(n)，

故本题：Catalan(5) = 42。

**哈夫曼编码**

**9个字符出现频率为0,1,1,2,3,5,8,13,21，其哈夫曼编码（ ）**(2020年)

A.最大长度为6

B.最大长度为7

C.最大长度为8

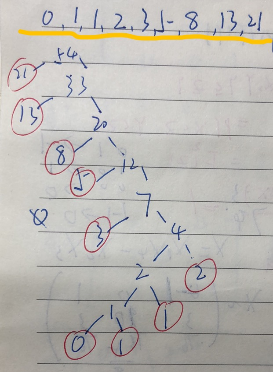
D.最大长度为9

选C，长度指的就是叶结点的深度，最大长度指的就是叶结点中的深度的最大值。

先构造出哈夫曼编码树，然后数一数最深的是哪个就知道了

可以记住规律结论：

* 字符频率数 = 叶子节点数
* 哈夫曼编码最大深度 = 叶子结点总数 - 1



**交换哈夫曼树不同深度的节点，平均编码长度必然改变** (2019年)

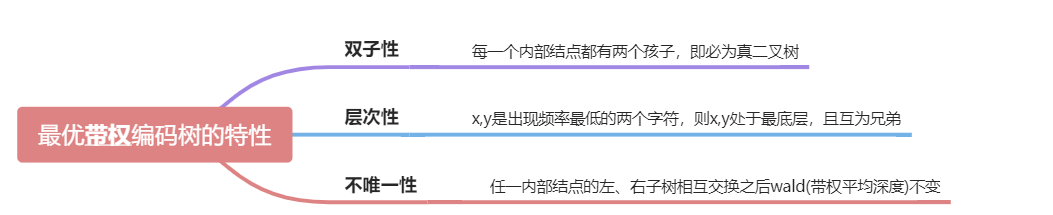
* 正确
* 错误

错误。结点的深度不同，但是如果结点的权值或者说频率相同，那么平均编码长度是不变的。

**PFC(最优前缀编码)互换不同深度节点位置一定会破坏其性质** (2018年)

* 正确
* 错误

错误。



互换频率相同的节点，不会破坏其性质。

**哈夫曼树距离根节点深度更小的节点的权值可能小于深度更大的节点的权值。**(2017年)

* 正确
* 错误

错误。这里的权值和频率是一个意思。

哈夫曼编码树是贪心构造的，先选取的是频率最小的依次构造的，所以频率越高的越接近树根，所以深度越小的权值越大，顶多只能出现深度更小的权值可能等于深度更大的结点的权值，不可能小于。

**AVL树**

**规模为n的AVL一次插入操作最坏情况下会引起logn次局部重构** (2020年)

* 正确
* 错误

错误，插入操作只需要O(1)次旋转操作，子树高度复原，其被影响的祖先也复原了。

AVL树的删除操作在最坏情况下会引起O(logn)次局部重构，插入操作则不会。

**在 AVL 树中，（）可能会发生两次旋转调整？** (2016年)

A 添加、删除节点操作

B 仅删除节点操作

C 仅添加节点操作

D 添加、删除节点都不

选A。

添加、删除操作都可能会发生旋转。

添加结点可能的旋转操作至多两次，删除结点可能的旋转操作至多Ω(logn)。

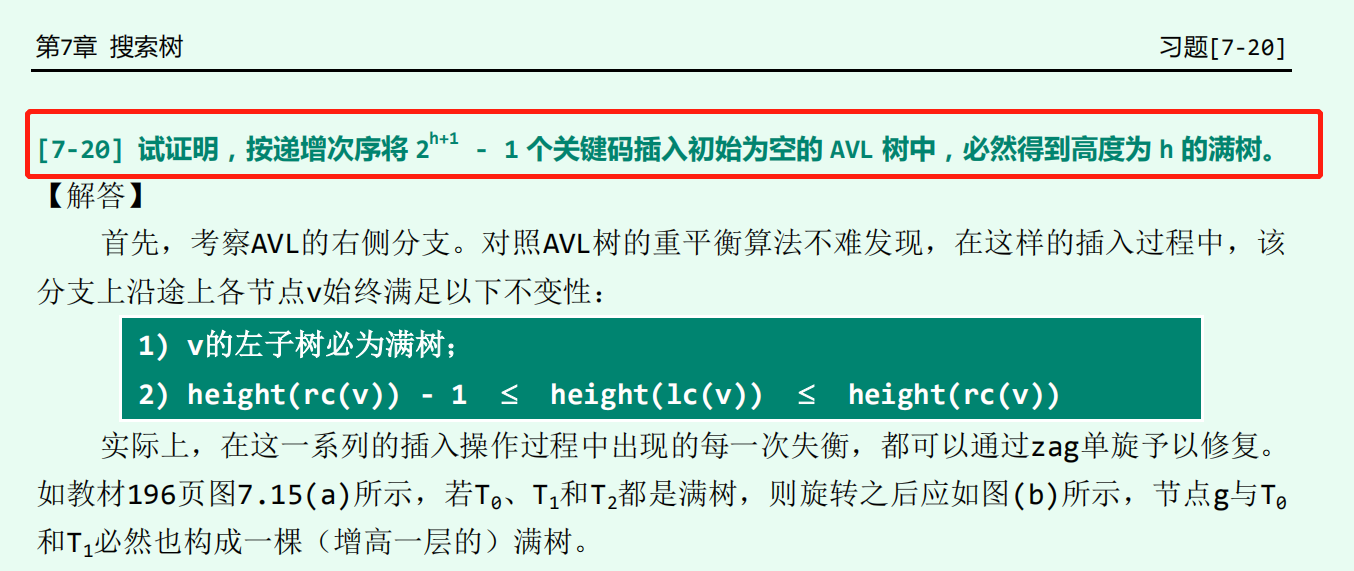
**对于同一个长度为 n 的序列分别按照递增和递减的顺序构造 AVL 树，那么“存在正整数k，使 n=2^k-1 ”是“两次构造的堆相同”的（ ）**(2018年)

A.充分不必要条件             B. 必要不充分条件             C. 充分必要条件             D. 既不充分也不必要条件

选C，充要条件。

只有当树为满树时，递增，递减序列构造的AVL树是相等的。

设得到的AVL的树高为h，即 k = h+1，n = 2^(h+1) - 1。



**将关键字1，2，3...，2016插入初始为空的平衡二叉树中，假设只有一个根节点的二叉树高度为0，那么最终二叉树的高度是多少？**(2017年)

A9.               B.10               C.11               D.12

解答：

将n个结点按递增/递减次序插入AVL树，且n满足 ，则此AVL高度为 。

，故得到的AVL高度为 h = 10。

**Splay伸展树**

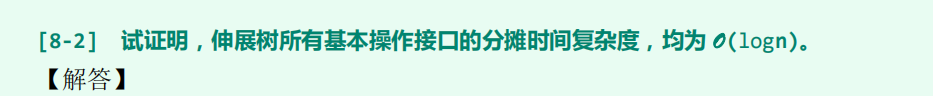
**对于不符合局部性原理的访问，splay的分摊复杂度不是O(logn)。**(2019年)

* 正确
* 错误

错误。

习题解析8-2有证明：伸展树所有基本操作接口的分摊时间复杂度均为*O(logn)*。

只是当局部性强，缓存命中率高时，分摊复杂度甚至可以更高到 O(logk) ( k << n)



**伸展树插入操作的分摊时间复杂度O(logn)** (2018年)

* 正确
* 错误

正确，伸展树的插入删除操作都是要先进行搜索，搜索操作用双层伸展分摊时间复杂度都是O(logn)。

伸展树内含的含义就是双层伸展。

**B树**

**7阶B-树根节点常驻内存，则对规模为2017的B-树最多需要几次访外存？**(2018年)

A.          B.          C.              D.

含有N个关键码m阶B树的树高：

此处 m = 7，N = 2017，代入公式计算得到 h <= log4(1009) + 1 = 5.99 ，h取5

由于根节点已经在内存中了，外部节点不计入访问，故只需要 5-1 =4 次访外存。

**搜索7阶B树的第2016个关键字，假设B树根节点在内存中，则共需启动几次I/O.** (2017年)

A.  B.  C.  D.

含有N个关键码m阶B树的树高h的范围 ：

此处 m = 7，N = 2017，代入公式得 4 <= h <= 5  
由于根节点已经在内存中了，外部节点不计入访问，故需要3到4次I/O。

**红黑树**

**红黑树上所有节点的黑深度和黑高度之和必相等。**(2020年)

* 正确
* 错误

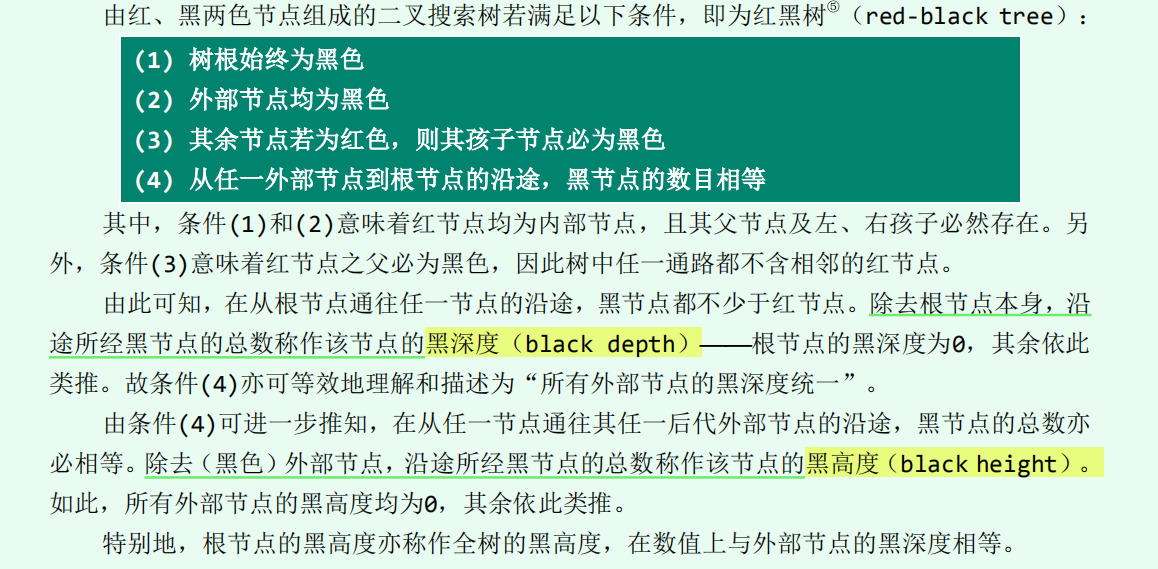
错误。

红黑树必须满足条件：从任一外部节点到根节点的沿途，黑节点的数目相等。

对于**黑节点**来说，其 黑高度+黑深度=全树的黑高度；

对于**红节点**来说，其 黑高度+黑深度=全树的黑高度 - 1。

具体地，黑高度和黑深度的定义见以下邓书截图：



**试举出红黑树优于AVL树的场景，红黑树相比AVL树的优势是什么？**(2019年)

1. AVL树需要维护平衡因子，而红黑树仅需对结点重染色。
2. 相比AVL树，红黑树每次插入删除操作所进行的旋转操作都是O(1)的，而AVL的删除操作是O(logn)的。因此由于红黑树的拓扑结构变化非常小，在持久性、历史版本的维护上具有优势，可以最大可能的节省空间。

**BST二叉搜索树**

**二叉搜索树中最大的节点** (2016年)

A 仅有左孩子，没有右孩子

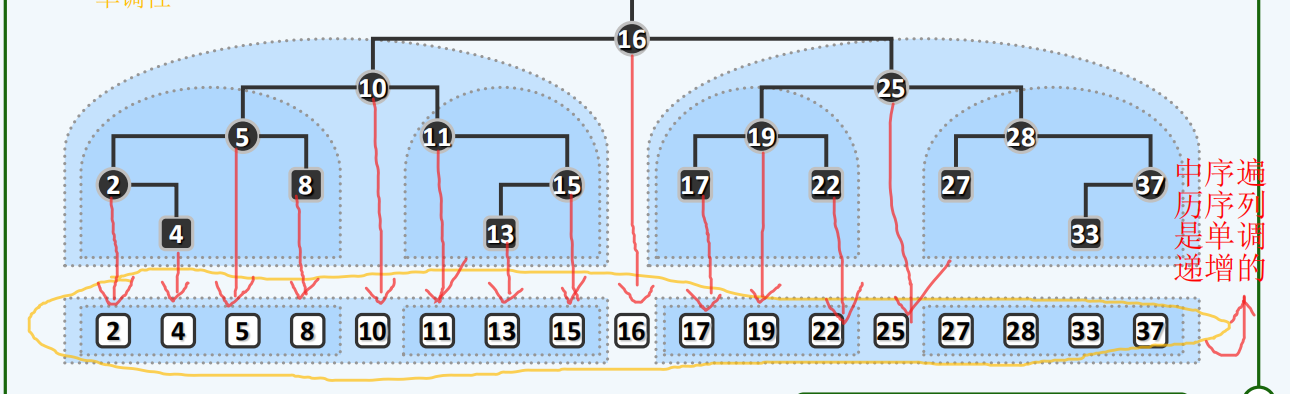
B 仅有右孩子，没有左孩子

C 既有左孩子，又有右孩子

D 没有左孩子，也没有右孩子

此题有问题。只能确定没有右孩子，左孩子可能有也可能没有。

二叉搜索树的中序遍历就是结点大小的顺序



由上图容易看出，最大结点在最右子树的最右结点，其一定是没有右孩子的

同理，最小结点一定是在最左子树的最左结点，其一定是没有左孩子的。

**叶节点数量为2018的二叉树，层次遍历时队列容量必然小于2018。**(2019年)

* 正确
* 错误

错误❌。应该是小于等于。

假如此二叉树是一颗完全二叉树，其叶结点数为2018。

完全二叉树层次遍历队列容量必然 <= 叶子节点个数，即队列容量小于等于2018。

完全二叉树是二叉树中结点数最多的树了，它都如此，何况其他树，只会比它更少。

参考《习题解析》【5-18】【5-19】

**对于二叉树，通过先序遍历和后序遍历不能确定其层次遍历。** (2019年)

* 正确
* 错误

错误❌。

先序遍历和后序遍历不可以确定唯一的一棵二叉树。

但是先序和后序可以确定层序遍历。

**判断与证明 给定一棵二叉树的先序和后序遍历序列，通过先序和后序遍历序列能否确定唯一层次遍历序列？若可以给出证明，不可以则说明理由。**(2020年)

首先给出结论：先序和后序可以确定层序遍历序列 。

证明：（递归生成法）

* 设其先序遍历为：V，\*\*\*L\*\*\*\*，\*\*\*\*R\*\*\*\*
* 后序遍历为：\*\*\*L\*\*\*\*，\*\*\*\*R\*\*\*\*，V
* 当遍历到某一节点 v
  + ① v无孩子，显然无歧义
  + ② v既有左孩子lc，也有右孩子rc
    - 局部先序为 v，lc，rc ；后序为 lc，rc ，v
    - 分离根节点v之后，左右子树先后顺序明确，确定层次遍历 v，lc，rc
    - 递归左子树\*\*\*L\*\*\*\*和右子树\*\*\*\*R\*\*\*\*
  + ③ v只有左孩子lc，没有右孩子
    - 局部先序为 v，lc；后序为 lc，v
    - 确定层次遍历 v，lc
    - 递归左子树\*\*\*L\*\*\*\*
  + ④ v只有右孩子rc，没有左孩子
    - 局部先序为 v，rc；后序为 rc，v
    - 确定层次遍历 v，rc
    - 递归右子树\*\*\*\*R\*\*\*\*
* 综上，因为可以递归生成唯一的层次遍历序列，显然就确定了层次遍历序列。

**判断与证明 给出中序序列{D B A E C F}和层次序列{A B C D E F}能否唯一确定一颗二叉树？若能给出步骤，不能的话请构造其中一棵** (2016年)

答：中序+层序 -> 能唯一确定一棵二叉树。

对于任意一棵二叉树

* 中序：\*\*\*L\*\*\*\*，V , \*\*\*\*R\*\*\*\*
* 层次：**VLR**########……

① 根据层次遍历可确定根节点在中序中的位置

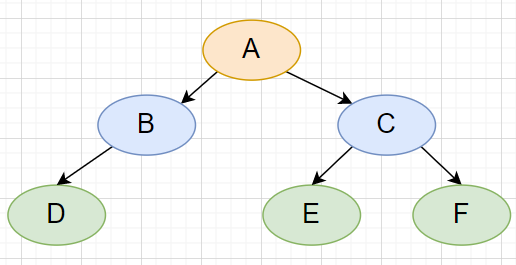
② 根据根节点将中序遍历序列划分为左右两个子序列

③ 递归遍历左右子序列的层次和中序序列，按照①②

根结点，左右子树的顺序能够毫无歧义的确定，故中序和层序能确定一棵二叉树。

就本题而言，

* D B **A** E C F；**A** B C D E F
* D **B** **A** E **C** F；**A** **B** **C** D E F
* **D B A E C F；A B C D E F**



**🌞重构总结**

**任意给出树的两种遍历序列，能否确定唯一的一颗二叉树或二叉树的另一种遍历序列的一些结论如下：**

* **中序 + 先序|后序|层序 —> 唯一二叉树** （中序带谁都能组队打boss）
  + （中序 + 层序 -> 唯一二叉树）2016年已经考过证明
* **层序 + 先序|后序 —❌ —> 唯一二叉树**
* **先序+后序 —❌ —> 唯一二叉树**
* **(先序+后序)&&真二叉树 —> 唯一二叉树**（先序和后序在一起要加条件）
* **先序+后序 —> 层序**
  + （先序 + 后序 ->层序）2019年已经考过判断，2020年已经考过证明

**算法题**

**1.下图给出二叉树的结构体声明**：A,B,C,D和F都是3分每题，E为5分。 (2020年)

struct BinNode { //二叉树节点

BinNodePosi(T) lc;

BinNodePosi(T) rc;

int size; //当前节点和孩⼦总数

}

A.完全二叉树左子树的规模为\_\_\_\_\_\_\_，请给出递推公式；

B.给出A的伪代码实现 lSize()；

C.search(k)找出中序遍历序列第k个节点（中序遍历起始下标为0，模拟中序遍历不得分）；

D.splyto(a, x) ，a为x的祖先，通过zigzag操作将x调整变为a的孩子，若a为NULL，x调为根节点；

E.将一颗splay树调成完全二叉树（可用上述函数），要求时间复杂度为O(nlogn)，迭代深度不超过O(logn);

F.证明你在E中给出的算法满足复杂度和迭代深度的要求。

**解答：**

A.完全二叉树左子树的规模为\_\_\_\_\_\_\_，请给出递推公式；

注意本小题是为后面服务的

设完全二叉树的规模为n，其高度：

高度为h-1的满二叉树节点个数为：

此二叉树的最后一层节点个数为：

而高度为h的满二叉树最后一层结点个数为 ：

综上，完全二叉树左子树规模：

B.给出A的伪代码实现 lSize()；

int lSize(int n){

int h = log(n)/log(2);

int b = pow(2,h);

int a = n-b + 1;

int lSzie = b/2 - 1 + min(a,b/2);

return lSzie;

}

C.search(k)找出中序遍历序列第k个节点（中序遍历起始下标为0，模拟中序遍历不得分）；

之所以不要模拟中序遍历，是因为我们每个结点中已知了size情况，也即包含当前结点和孩子结点总数，所以要充分利用这个信息。注意结合中序遍历的访问顺序和结点数的特点找规律。

解答：中序遍历的访问顺序是 左 ->根 ->右，而每个结点的个数 = 左子树节点个数 + 右结点个数 + 1。

1.若 k == leftSize + 1：直接返回根节点；

2.若 k < leftSize + 1：在左子树上寻找第 k 个节点；

3.若 k > leftSize + 1：在右子树上寻找第 k-leftSize -1个节点。

BinNodePosi(T) searchK( BinNodePosi(T) root, int k){

int leftSize = root->lc->size;

if(leftSize + 1 < k)

return searchK(root->rc, k-leftSize-1);

else if(leftSize + 1 == k)

return root;

else

return searchK(root->lc, k);

}

D.splyto(a, x) ，a为x的祖先，通过zigzag操作将x调整变为a的孩子，若a为NULL，x调为根节点；

解答：

1.x已经使a的孩子了 —> 什么都不用做

2.x是a的孙子 —> 只需一次旋转

① x是左孩子 —> zig旋转，使x称为a的孩子

② x是右孩子 —> zag旋转，使x称为a的孩子

3.x不是a的直接孙子 —> splay算法的双层伸展 —> 直到x为a的孩子

设x的父亲为p，p的父亲为g

① px为左左 —> zig，zig旋转

② px为右右 —> zag，zag旋转

③ px为左右 —> zag，zig旋转

④ px为右左 —> zig，zag旋转

void splyto(BinNodePosi(T) a, BinNodePosi(T) x){

if(a == x->parent) return;//x已经使a的孩子了，什么都不用做

else if(a == x->parent->parent){//x是a的孙子，只需一次旋转

if(isLchild(x)) zig(x->parent);

else zag(x->parent);

}else {//x不是a的直接孙子

BinNodePosi(T) p = x->parent;

BinNodePosi(T) g = p->parent;

if(isLchild(p)&&isLchild(x)) {zig(g);zig(p);}

else if(isRchild(p)&&isRchild(x)) {zag(g);zag(p);}

else if(isLchild(p)&&isRchild(x)) {zag(p);zig(g);}

else if(isRchild(p)&&isLchild(x)) {zig(p);zag(g);}

splyto(a,x);//继续检查a和x的关系,做递归的旋转

}

}

E.将一颗splay树调成完全二叉树（可用上述函数），要求时间复杂度为O(nlogn)，迭代深度不超过O(logn);

 解答：此问要结合前面几问来回答

* 由B问的 lSize(int n)函数可以得到规模为n的完全二叉树的左子树的规模
* 由C问的 searchK(BinNodePosi(T) root, int k)函数可以得到中序遍历第K个节点
* 由D问的 splyto(BinNodePosi(T) a, BinNodePosi(T) x)函数可以把 x 结点旋转成为 a 的孩子
* 并且每个结点内部的size属性就已知了以其为根节点的子树的规模

1.先从这棵splay树里找一个适合做完全二叉树根节点的结点

①已知splay树的规模为 n = root->size，规模为n的完全二叉树的左子树规模为 lSize(n)

②通过 searchK函数对splay树做中序遍历找到第 lSize(n) + 1 个结点，将其作为要构建的完全二叉树的根节点

2.通过splyto函数将原树中这个结点旋转成为新的根节点

3.按步骤12递归新的根节点的的左子树和右子树，其左子树和右子树同样需要满足完全二叉树的定义。

BinNodePosi(T) to\_Compl\_BT(BinNodePosi(T) root){

if(!root) return NULL;

if(!root.lc && !root.rc) return root;

int leftSize = lSize(root->size);

BinNodePosi(T) newRoot = searchK(root,leftSize+1);//找到适合做完全二叉树根节点的结点

splyto(root->parent, newRoot);//将newRoot旋转为根节点，取代root原来的位置，作为原来root的父亲的孩子

to\_Compl\_BT(newRoot->lc);//递归构建左子树也为一棵完全二叉树

to\_Compl\_BT(newRoot->rc);//递归构建右子树也为一棵完全二叉树

return newRoot;

}

F.证明你在E中给出的算法满足复杂度和迭代深度的要求。

解答：

lSize函数的复杂度为 O(1)

searchK函数的复杂度为 O(logn)

splyto函数的复杂度为 O(logn)

所以 to\_Compl\_BT函数单次迭代的复杂度为 O(1) + O(logn) + O(logn) = O(logn)

总计有n个结点需要迭代，故to\_Compl\_BT函数总体时间复杂度为 n× O(logn) = O(nlogn)

而二叉树高度 h = O(logn)，故迭代深度不超过O(logn)。

**返回后序遍历的第 K 个节点，时间复杂度不超过 x 的深度，Ο(depth(x))** (7+3+4 = 14) (2019年)

struct BinNode{

int size; //当前节点和孩⼦总数

BinNode \*lchild,\*rchild;

};

BinNode \*rank(BinNode\* t,int k){

//有效代码⾏数不超过 12 ⾏

//不要尝试模拟后序遍历，时间复杂度会超时。

}

⼀，给出具体算法实现。

⼆，解释你的算法。

三，分析时间复杂度和空间复杂度。

解答：

①若 root->size == k：直接返回根节点；

②若 leftSize <= k ：在左子树上寻找第k个节点；

③若 leftSize > k： 在右子树上寻找第 k-leftSize 个节点；

一：代码如下

BinNode \*rank(BinNode\* root, int k){

//检查参数是否合法

if(root == NULL || root->size < k) return NULL;

if(root->size == k) return root;

//如果没有左子树，则直接进入右子树搜索

if(root->lchild == NULL)

return rank(root->rchild, k);

int leftSize = root->lchild->size;

//左子树不为空，且根据左子树规模和目标比较，进入左子树搜索

if(k <= leftSize)

return rank(root->lchild, k);

else

return rank(root->rchild, k-leftSize);

}

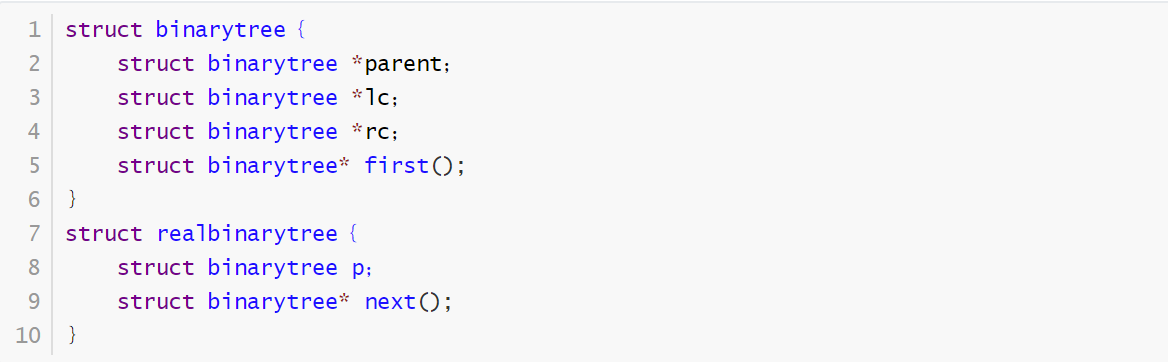
二：算法解释：

后序遍历的顺序是：左->右->根。现在已知了每个结点孩子包含其本身的个数。

结合已知，不难发现这样的规律：当 root->size == k 此根节点就是要找的结点；当 leftSize <= k ：在左子树上寻找第k个节点；当 leftSize > k： 在右子树上寻找第 k-leftSize 个节点。

三：时间复杂度：O(depth(x))。空间复杂度O(1)。

**3.若二叉树的数据结构如下** (2017年)



（1）若first（）函数是取二叉树后序遍历节点的第一个节点，请写出first（）函数代码。

（2）若next（）函数是取该节点的后序遍历的后继，请写出next（）函数代码。

（3）在调用first（）函数和next（）函数对二叉树进行后序遍历时，证明遍历时间复杂度为o(n)。

解答：

（1）若first()函数是取二叉树后序遍历节点的第一个节点，请写出first（）函数代码。

即求以当前节点为根节点的后序遍历的第一个结点。

binarytree\*first() {

while(this->lc || this->rc){//优先遍历左子树，左子树为空时，才转到右子树

if(this->lc) this = this->lc;

else this = this->rc;

}

return this;

}

（2）若next（）函数是取该节点的后序遍历的后继，请写出next（）函数代码。

1.如果当前结点无父亲，其实当前结点就是根节点，其后序遍历后继为空。

2.如果当前结点是左孩子，其后序遍历后继为以其兄弟结点为根节点的子树的后序遍历的第一个结点（若有）

3.如果当前结点是右孩子，其后序遍历后继为其父亲

//返回以当前结点在后序遍历当中的后继结点

binarytree\* realbinarytree::next(){

if(!this->parent) return NULL;//后继为空

if(this == this->parent->lc && !this->parent->rc){ //当前结点是左孩子并且其右兄弟不为空

return this->parent->rc->first();

}else{//【是右孩子】或者【是左孩子但右兄弟为空的情况】

return this->parent;

}

}

（3）在调用first()函数和next()函数对二叉树进行后序遍历时，证明遍历时间复杂度为O(n)。

对于任意一个节点，遍历过程中最多被涉及3次，第1次进入左子树，第2次从左子树进入右子树，第3次从右子树退出。

因而整个遍历的消耗=O(3n)=O(n)。 待考虑清楚 待解答

//从结点x开始对此棵二叉树做后序遍历

void PostTravel(realbinarytree\* x) {

while(!x){

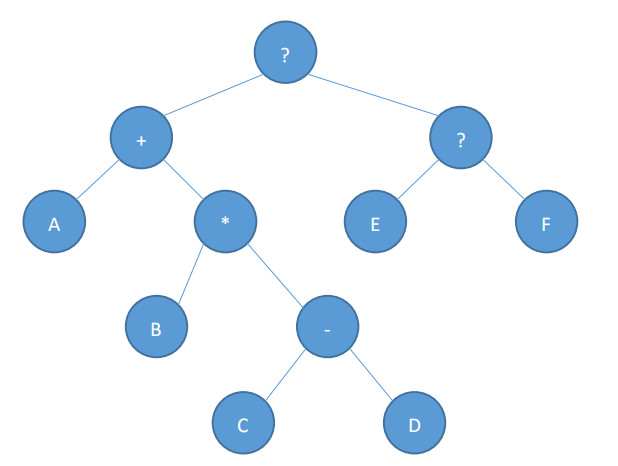
visit(x);

x->next();

}

}

**设计一个算法，把一个中序遍历 ABCD-\*+EF??(后面三个符号忘记了不过不重要)构造成如下图所示的二叉树** (2016年)



a) 描述算法思想

b) 伪代码实现

解答：

和逆波兰表达式求值的思想整体一致，只不过这里还需要把每一个值包装在一个结点上。

a）

* 扫描序列，操作数入栈。
* 当遇到符号，取出栈顶两个元素，作为此符号结点的左右子结点。
* 然后把这个结点入栈。
* 直到扫描完序列，栈变空

b）代码如下

BinNode\* RPN2BinTree(char\* s) {

stack<BinNode\*> sta;

char\*p = s;

while(\*p != '\0'){

BinNode\* node = new BinNode(\*p);

if(\*p == 操作符 ) {

BinNode\* rc = sta.top(); sta.pop();

BinNode\* lc = sta.top(); sta.pop();

node->lc =lc;

node->rc =rc;

}

sta.push\_back(node);

}

return sta.top();

}

**给了一个算法，问访问节点的顺序，树的样子是一颗深度为 4 的二叉树。** (2016年)

typedef struct binNode {

char data;

struct binNode \*rc,\*lc;//左子树右子树

}binNode;

twist(node x) {

if(!x) return;

if(x->rc) {

twist(x->lc->rc);

putchar(x->data);

twist(忘了);

putchar(忘了);

}

else{

Twist(忘了);

Putchar(忘了);

}

}

解答： 题目不全，无法做

**四、堆**

**完全二叉堆**

**完全二叉堆删除操作平均时间复杂度为O(1),最坏情况下为O(logn)。**(2020年)

* 正确
* 错误

错误 ❌

错误。删除元素需要用下滤操作，该下滤在平均情况下的时间复杂度为O(logn)。

另外，需要注意的是，在完全二叉堆中插入元素需要进行上滤操作，在关键码均匀独立分布时，其时间复杂度在平均情况下为O(1)，详情可参考邓【习题10-6】。



**将n个元素组成一个完全二叉堆，时间复杂度至少为O(nlogn)。**(2020年)

* 正确
* 错误

错误 ❌

使用Floyd建堆算法只用O(n)的时间。

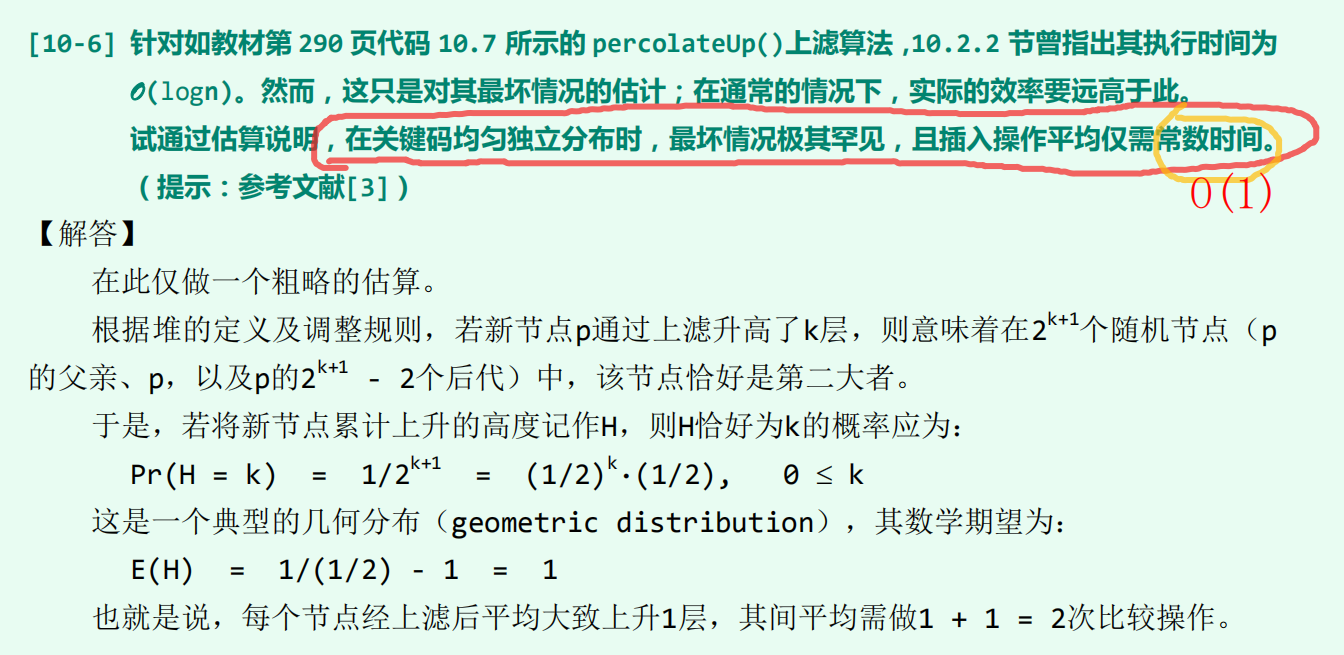
**如果插入的关键码独立均匀分布，堆的插入操作平均时间复杂度为O(1)。** (2019年)

* 正确
* 错误

正确 ✔

由于完全二叉堆插入采用上滤操作，并且每一层的节点数是指数型的，因此若完全随机地插入，则只期望上升1层。

参考邓【习题10-6】



**输入随机的情况下完全二叉堆的插入平均时间是常数** (2018年)

* 正确
* 错误

正确 ✔

同上题。由于完全二叉堆插入采用上滤操作，并且每一层的节点数是指数型的，因此若完全随机地插入，则只期望上升1层。

参考邓【习题10-6】

**胜者树-败者树**

**败者树删除操作的时间复杂度在常系数上优于胜者树。** (2020年)

* 正确
* 错误

正确 ✔

重赛过程中，败者树只与其父结点比较，不用交替访问沿途结点及其兄弟，减少了访存时间。

插入删除操作都要重构，重新选则胜者。

故不论是插入还是删除操作，其常系数上复杂度都小于胜者树。

**简答 相比于锦标赛树，败者树的优势是什么？**(2019年)

答：重赛过程中，败者树只与其父结点比较，不用交替访问沿途结点及其兄弟，减少了访存时间，时间效率高。

**左式堆**

**Crane算法合并左式堆A和B为H，H右侧链节点未必都来自A或B右侧链。** (2020年)

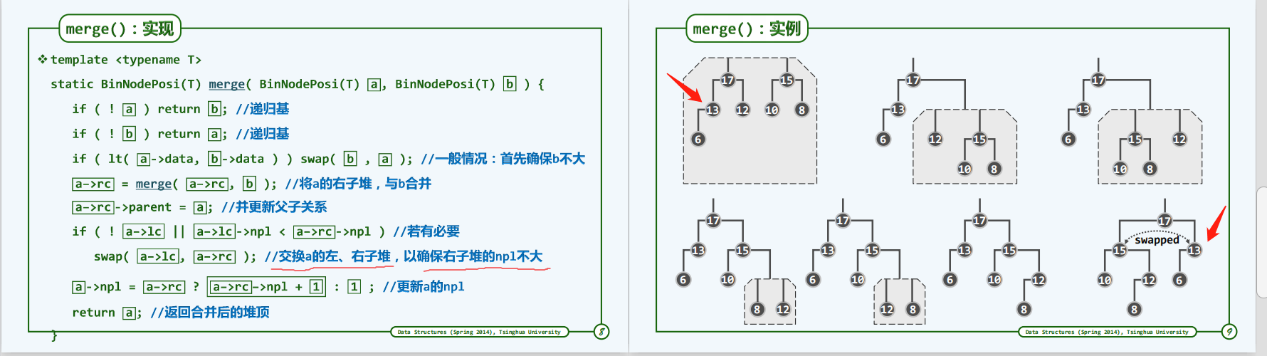
* 正确
* 错误

正确 ✔

H右侧链节点可能来自A的左右结点和B的左右结点，未必都来自A或B右侧链。

左式堆合并过程中为了保持左倾性，可能会交换左右子树。

如下图，13原本不是A的右侧链中的元素，但是合并后的H中其成为了右侧链元素。



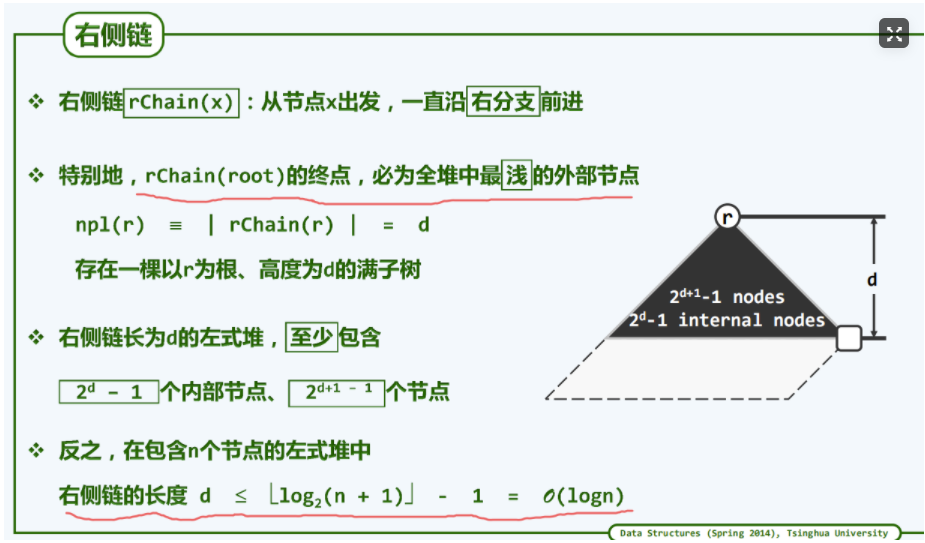
**左式堆最右侧链长度为k，则左式堆\_\_\_\_\_\_含有\_\_\_\_\_\_个内部节点。** (2018年)

A.最少 2^k          B.最少 2^k-1              C.最多 \*\*                  D.最多 \*\*

应该是选B

至少包含

* 2^k - 1 个内部结点
* 2^(k+1) - 1 个结点



**五、哈希、散列**

**开放式散列比封闭式散列可以更有效地利用局部缓存。** (2020年)

* 正确
* 错误

错误 ❌。

**闭散列**：可用的散列地址仅限于散列表所覆盖的范围之内。

**开散列：**可用的散列地址除了散列表，还引入了别的附加空间，如多槽位，独立链，公共溢出区。

封闭式散列可用的散列地址仅限于散列表所覆盖的范围之内，能保证物理上的关联性，因此可以更有效地利用局部缓存。而开放式散列使用的策略如独立链等引入了次级关联，因而不能保证物理上的关联性。

**简答：相比开散列，闭散列的优势是什么，说明两点。**(2019年)

**答：**

1. 闭散列结构本身保持简洁，在散列表内部解决冲突，无需附加空间
2. 闭散列可更有效的利用局部缓存

**对长度为m=4k+3素数的散列表双平方探测一定能访问其全部元素** (2018年)

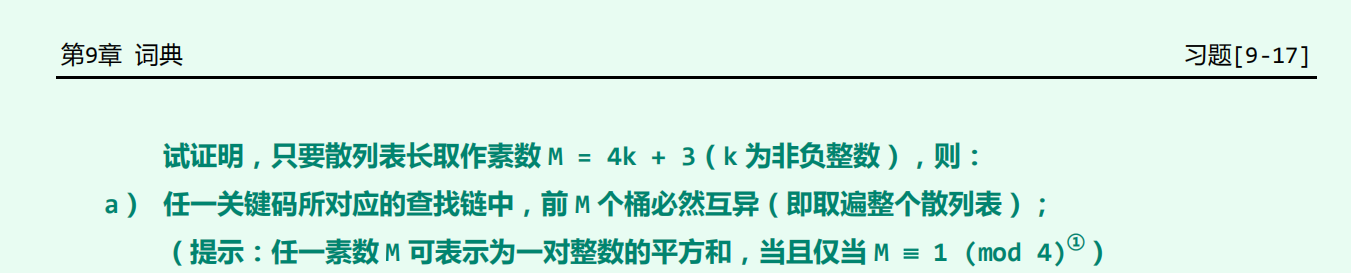
* 正确
* 错误

正确 ✔

表长取做素数，⭐

必然可以保证查找链的前M项均互异（即取遍整个散列表）；

在装填因子未增至100%之前，插入操作必然成功（而不致因无法抵达空桶而失败）。



**散列表用不超过长度的素数，即使分布理想，使用取余法仍然会堆积** (2017年)

* 正确
* 错误

正确 ✔

除余法 hash(key) = key % M

设散列表长度为N，素数为Q<N，则所有的关键码都必然落在散列表的区域内。

存在一个永远散列不到的区域，显然分布不均。

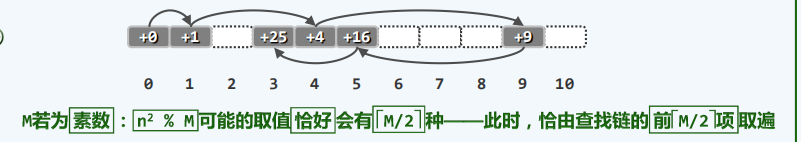
本来应该映射到区域 [Q,M) 的元素，现在被映射到 [0,Q)，堆在其他元素上了，大大增加了和其他元素冲突的几率，所以会堆积。

**散列长为2017，采用单平方探测，已经存入1000个元素，问此时最多有( )个懒惰删除的桶单元** (2018年)

A.8         B.9         C.1016             D.1017

选 B。

2017是素数。可访问元素数=1009。因此最多有9个懒惰删除的桶单元。



**简答：散列表长为 13，采用双散列函数解决冲突：** (2016年)

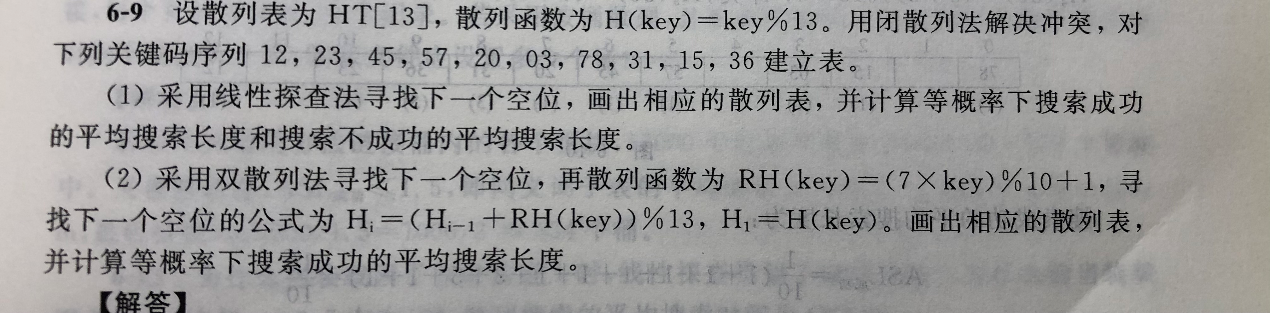
散列表长为 13，采用双散列函数解决冲突：H(key) = key % 13，H’(key) = ( 7 \* key % 10 ) + 1。

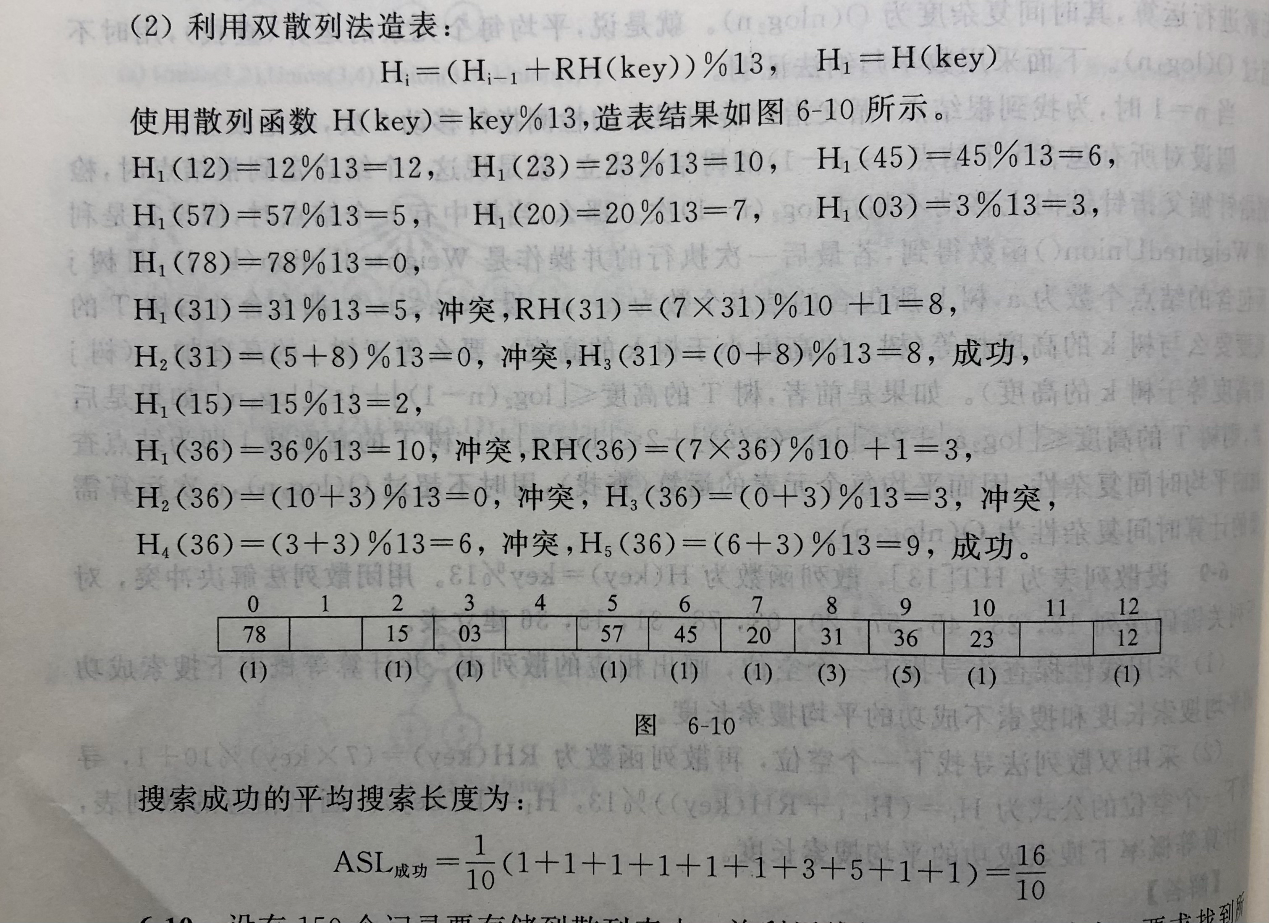
输入顺序为：12，23，45，57，20，03，78，31，15，36。

【注：回忆版题面已按《数据结构习题解析(第2版)-殷人坤》习题6-9对应题面修正，实际考试中可能有所简化】

1) 构造散列表

2) 求等概率下搜索成功的平均查找长度





**六、图**

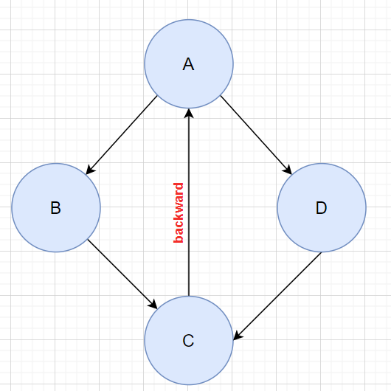
**有向图DFS后有k条边被标记为后向边，图中未必恰含k个环路。**(2020年)

* 正确
* 错误

正确 ✔

如下图，从A开始访问，只有一条后向边，但图中有两个环路。

DFS中有边被标记为后向边，就一定有环路。但是有K条后向边，不代表恰有K个环路。⭐



**有向无环图DFS后各节点按（ ）拓扑排序。** (2020年)

A.被发现的顺序

B.被发现的逆序

C.回溯的顺序

D.回溯的逆序

选D

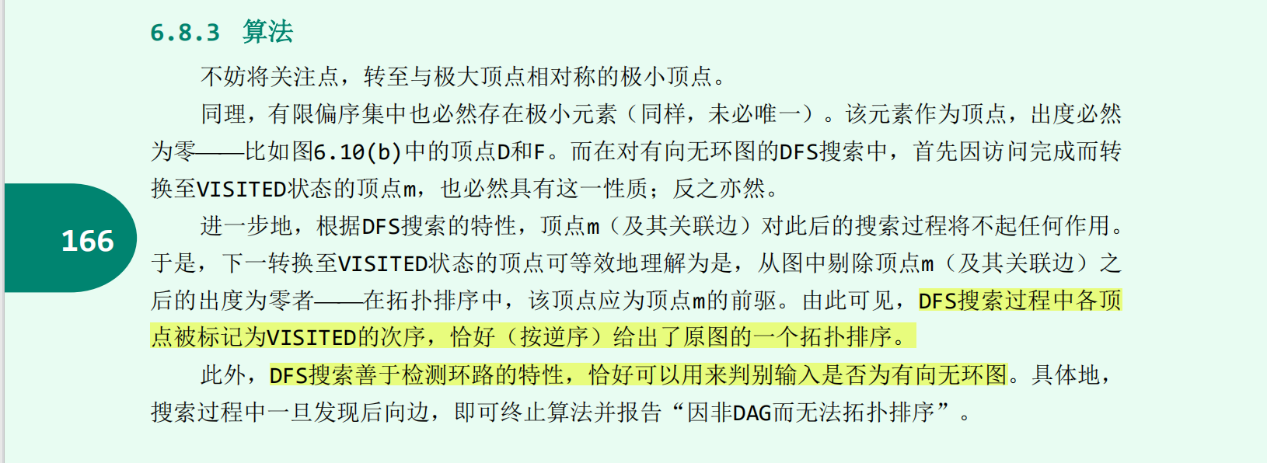
DFS搜索过程中各顶点被标记为VISITED的次序，恰好（按逆序）给出了原图的一个拓扑排序。

而被标记VISITED的顶点就是回溯的顶点。

例如：A->B->C

回溯的顺序就是CBA，其逆序就是ABC，也正好是此图的拓扑排序。

如果此算法用栈实现，则拓扑排序就是栈内各顶点自顶向底排序。（教材代码6.5）

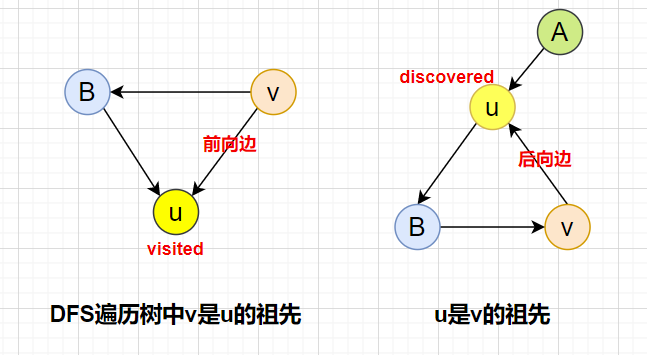


**简答：DFS过程中何时标记前向边？何时标记后向边？**(2019年)

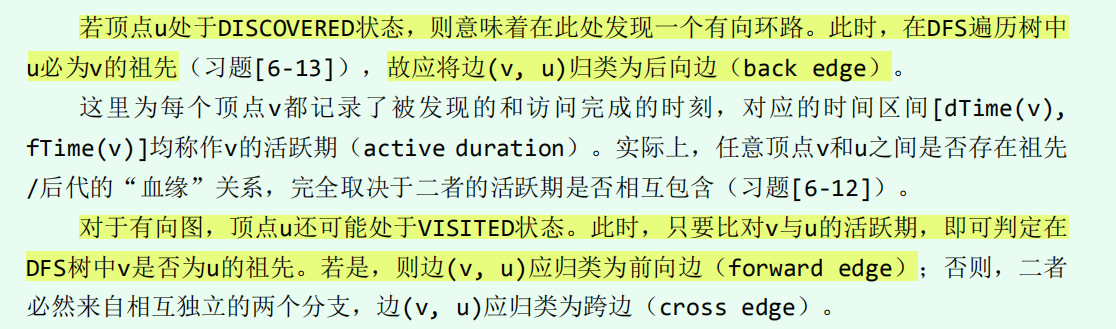
答：设两个顶点分别为u和v

发现当前顶点v，枚举其邻居u的时候

* 发现u被标记为VISITED状态，且v的发现时间早于u的，则此条边v->u被标记为前向边，v为u的祖先
* 发现u被标记为DISCOVERD状态，则此条边u->v被标记为后向边，u为v的祖先







**权值都为正整数的图能用迪杰斯特拉构造出最短路径**(2016年)

* 正确
* 错误

正确 ✔ Dijkstra是单源最短路径算法。

注意：Dijkstra只能处理正权边，不能处理负权边。

**简答：对于稠密图，迪杰斯特拉算法使用多叉堆替换二叉堆，为什么？多叉堆分叉数m怎么确定？**(2019年)

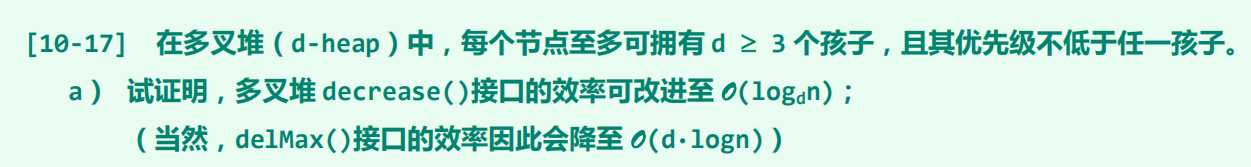
对于稠密图，使用多叉堆替换二叉堆能带来更优的性能。

原因在于：多叉堆的高度更底，相应的上滤成本降低，对应的插入操作的成本降低；虽然因此下滤成本增加，但是对于稠密图来说，插入操作远远多余删除操作，利大于弊，所以整体性能优化。

当分叉数 m = e/n + 2 时 ⭐

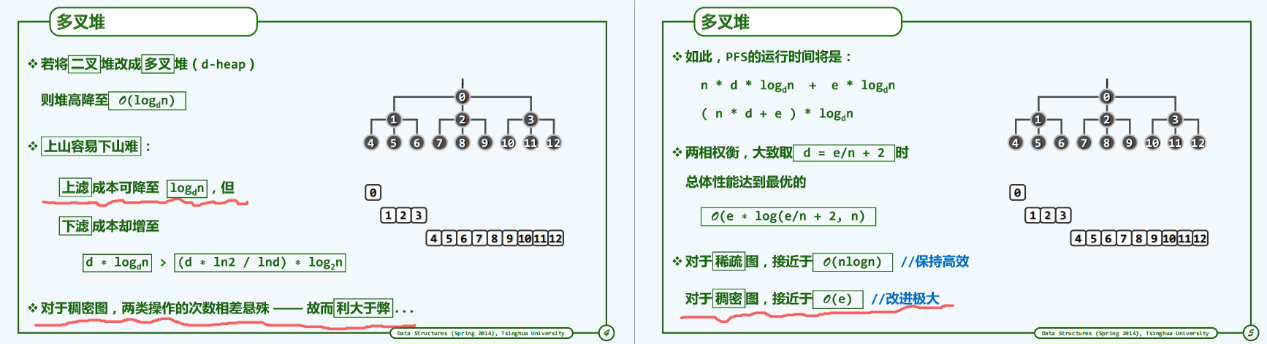
基于d叉堆实现的PFS优先级搜索 - Prim算法、Dijkstra算法性能能达到最优

* 对于稀疏图，接近于 O(nlogn)
* 对于稠密图，接近于 O(e)









**算法题**

**请利用图的广度优先遍历找出图中的最小环，若不存在环则输出+oo,要求时间复杂度为O(n×e)，空间复杂度为O(n)，最小环即环中边数最少的环。** (2017年)

（1）请描述你的算法思想。

（2）请用伪代码写出算法。

（3）说明你的算法的时间复杂度和空间复杂度。

解答：参考

每一次CROSS都if(CROSS) Ring = min(Ring, a.depth + b.depth + 1)。

每轮就能计算出经过根节点的所有环的最小环。把每一个节点依次当作根节点BFS即可。

int ans = INF;

int least\_ring(){//只考虑记录最小环的大小

for 所有的节点:

bfs(节点)；

}

void bfs(节点){

利用队列执行算法

若遇到CROSS边(a,b):

ans = min(ans, a.depth + b.depth + 1)

}

**简答：程序应该是 prim 算法**，问是否能够构成最小生成树，如果能就证明，不能举出例子驳斥。

V 表示图的点集，U 表示已经确定路径的点集，初始时 U 为空，F 为已经确定的路径，初始也为空。先任意取一点 u 放入 U，然后在 V-U 中遍历 u 的邻接点，选权值最小的边 e 和点 v 放入 F和 U 中，具体算法就请翻书吧(2016年)

prim算法

**七、字符串**

**KMP**

**随机英文字母串匹配，最好情况下蛮力 \_\_\_\_\_\_ KMP，平均复杂度蛮力 \_\_\_\_\_\_KMP。（ ）** (2020年)

A.坏于 坏于

B.相等 坏于

C.坏于 相等

D.相等 相等

选 D

KMP复杂度为O(n+m)，即使是在最坏情况下也为O(n+m)

随机字符匹配，字符集较大时，蛮力算法的平均复杂度为O(m+n)，与KMP算法相差无几。

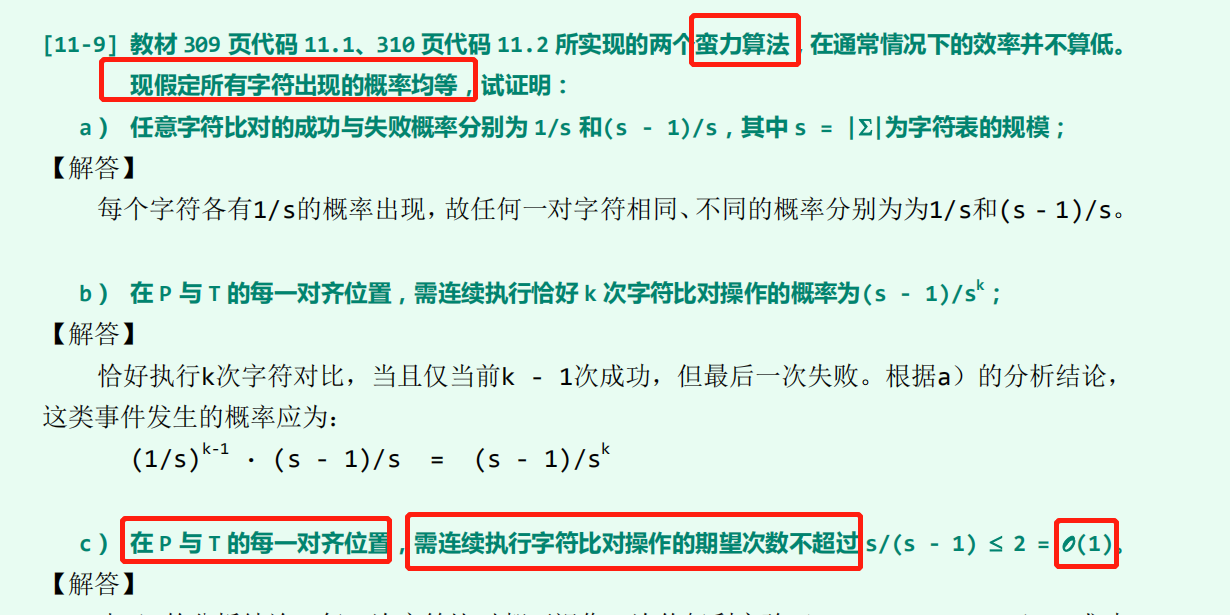
具体可参考【习题解析11-9】和【11-10】。

**判断：在字符集各字符出现概率相同时，kmp算法时间渐进程度接近蛮力算法。** (2017年)

* 正确
* 错误

正确 ✔

根据习题解析【习题解析11-9】，蛮力算法在所有字符出现概率均等的前提下，P与T的每一对齐位置，需连续执行字符比较操作的期望次数不超过O(1)，则总体就是O(n)，接近KMP算法。



**简答：在何种情形下KMP优于蛮力算法，为什么？**(2019年)

单次匹配概率越大（字符集越小）的场合，KMP的优势越明显；否则与蛮力算法的性能相差无几。

因为KMP算法在匹配过程中遇到不匹配的字符，充分利用以往对比所提供的信息，避免了模式串完全回退，从而可以快速前进。

**模式串HHFBHHFHHFBSHF改进后的next表，以下正确的是（ ）**(2020年)

A.next[13]=1, next[0]=-1

B.next[13]=1, next[0]=0

C.next[13]=0, next[0]=-1

D.next[13]=-1, next[0]=-1

选 A

next[0]一定等于-1。

模式串A[13] = F 的最长前缀匹配为A[0]和A[12] = H，长度为1；又A[1] != A[13]，故改进的next[13] = 1。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **rank** | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| **P[ ]** | \* | **H** | **H** | **F** | **B** | **H** | **H** | **F** | **H** | **H** | **F** | **B** | **S** | **H** | **F** |
| **未改进Next表** | N/A | -1 | 0 | 1 | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 0 | 1 |
| **改进Next表** | N/A | -1 | -1 | 1 | 0 | -1 | -1 | 1 | 3 | -1 | 1 | 0 | 4 | -1 | 1 |

下面的程序可以用来验算手工构造的next表是否正确  


#include <iostream>

#include <stdio.h>

#include <string.h>

using namespace std;

//非改进版next

int\* buildNext1(char\* p) {

int m = strlen(p);

int j=0;

int\* N = new int[m];

int t = N[0] =-1;

while( j < m-1){

if(0>t || p[j] == p[t]){

j++;t++;

N[j] = t;

}else

t = N[t];

}

return N;

}

//改进版next

int\* buildNext2(char\* p) {

int m = strlen(p);

int j=0;

int\* N = new int[m];

int t = N[0] =-1;

while( j < m-1){

if(0>t || p[j] == p[t]){

j++;t++;

N[j] = (p[j]!=p[t]?t:N[t]);

}else

t = N[t];

}

return N;

}

int main(){

char\* p = "HHFBHHFHHFBSHF";

int len = strlen(p);

int\* N = new int[len];

cout<<"模式串：";

for(int i=0;i<len;i++)

cout<<p[i];

cout<<endl;

N = buildNext1(p);

cout<<"非改进版Next表"<<endl;

for(int i=0;i<len;i++)

cout<<N[i]<<" ";

cout<<endl;

N = buildNext2(p);

cout<<"改进版Next表"<<endl;

for(int i=0;i<len;i++)

cout<<N[i]<<" ";

cout<<endl;

return 0;

}

**即使不使用改进的next表，KMP依然可以达到线性的时间复杂度**(2019年)

* 正确
* 错误

**正确 ✔。**无论是否是否next表，KMP算法复杂度都是O(n+m)

**没改进的next算法时间复杂度也是O(n)** (2018年)

* 正确
* 错误

**正确 ✔。**无论是否是否next表，KMP算法复杂度都是O(n+m)。

**BM算法**

**对于长度为 m 的串进行串匹配时好后缀数组中 gs[0]=1 的概率为** (2018年)

A.1/m   B.1/2^(m-1)                C.1/2^m                   D.1/2^(m+1)

选B。

gs[ j ]表示在BM\_GS算法中，当模式串的位置 j 与目标串某某失配时，此时模式串应当移动的位移量。

由gs[0]=1 知当模式串与目标串自右向左匹配时，模式串在其位置0处失配（而位置1~m-1均已匹配成功, P[1,m-1] = T[i，j] ），此时模式串应当向右移动1位，移动1位也意味着 P[0, m-2 ] = T[i，j] = P[1,m-1]

说明模式串的前m-1 个字符和后m-1个字符是完全匹配的，从而说明模式串每个字符都相等

设字符集大小为T，则每种字符的概率为 1/T，长度为m的串的字符概率就是 (1/T)^m

由于有T种字符，所以串中所有字符全等出现的概率为 T\*(1/T)^m = 1/T^(m-1)

似乎回忆题中漏掉了字符集大小的条件。可以猜测此处T=2。故选B。

**八、栈**

**逆波兰表达式0!1+23!4+^\*56!7\*8!?/-9+值等于2017，则？处的运算符为（   ）** (2018年)

A.加号         B.减号          C.乘号         D.除号         E.乘方          F.阶乘

选D，具体过程如下

0!1+23!4+^\*56!7\*8!?/-9+

1 1+23!4+^\*56!7\*8!?/-9+

2 23!4+^\*56!7\*8!?/-9+

2 2 6 4+^\*56!7\*8!?/-9+

2 2 10 ^\*56!7\*8!?/-9+

2 1024 \*56!7\*8!?/-9+

2048 56!7\*8!?/-9+

2048 5 720 7\*8!?/-9+

2048 5 5040 8!?/-9+

2048 5 5040 40320 ?/-9+

2048 5 ( 5040 ? 40320) /-9+

2048 [ 5/( 5040 ? 40320) ] -9+

[ 2048 - 5/( 5040 ? 40320) ] 9 +

[ 2048 - 5/( 5040 ? 40320) ] + 9 = 2017

5 / ( 5040 ? 40320) = 40

? 处应为 除号

**非法表达式(12)3+!4\*+5,执行evaluate算法后的结果 （  ）** (2018年)

A.99         B.89         C.88         D.98

选B

习题解析[4-12]



**有如下逆波兰式结果为2016，问?中的运算符号是多少（）** (2017年)

**2  0 ！ \*  2  2  \*  6  +  ^  18  8  ?  9  /  \***

A.+           B.\*                  C.^               D. !                  E./

2  0 ！ \*  2  2  \*  6  +  ^  18  8  ?  9  /  \*

2  1 \*  2  2  \*  6  +  ^  18  8  ?  9  /  \*

2  2  2  \*  6  +  ^  18  8  ?  9  /  \*

2 4  6  +  ^  18  8  ?  9  /  \*

2 10  ^  18  8  ?  9  /  \*

1024  18  8  ?  9  /  \*

若 ？ 为二元运算符

1024  (18 ? 8)    9  /  \*

1024  [(18 ? 8) /9]   \*

1024 \*  [(18 ? 8) /9]   = 2016 无解

若 ？ 为一元运算符 !

1024  18  8  ! 9  /  \*

1024  18  40320 9  /  \*

1024  18  4480  \* 无解

此题回忆版有问题，暂时作废。

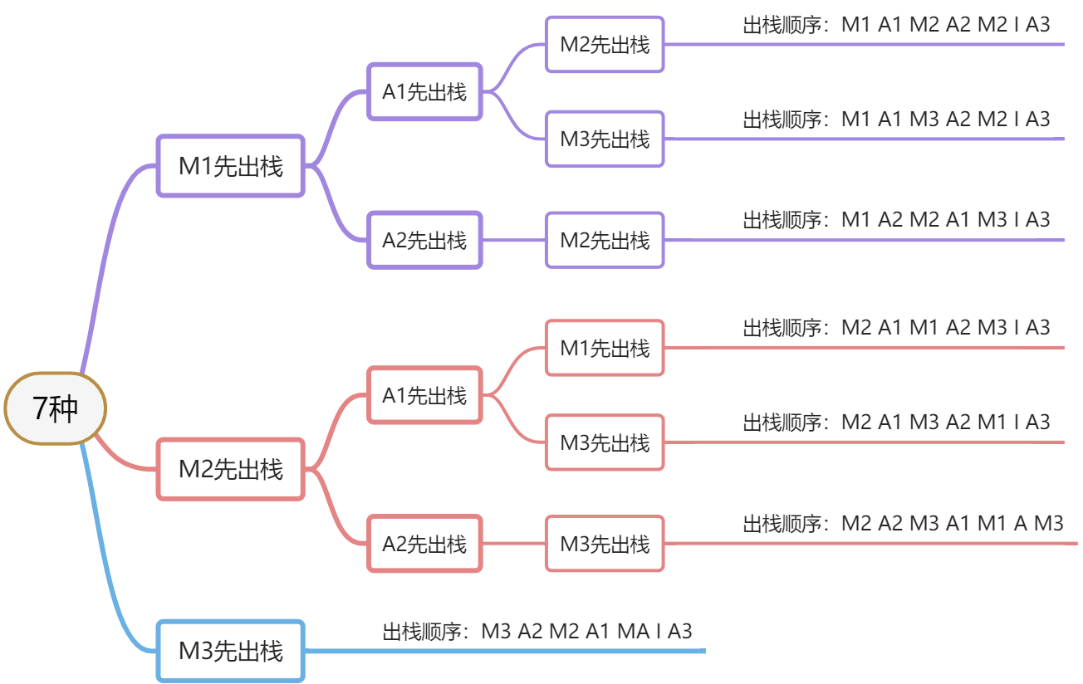
**一组输入 MAMAMIA 入栈，要求出栈顺序也为 MAMAMIA，共有几种方案？** (2016年)

A 4 B 5 C 6 D 7

答：选D，共有7种方案

注意到M和A有重复的，将其重新标记为

M1 ，A1， M2， A2， M3， I， A3



**简答**

**逆波兰表达式的优点是什么？既然中缀转换为逆波兰消耗的时间就可以计算出表达式值，那逆波兰意义何在？**(2019年)

1.逆波兰表达式可以不用事先约定好运算符之间的优先级，去掉括号后表达式无歧义。直接一趟扫描就可以计算，可以实时计算。

即RPN的运算符优先级表述能力强，计算效率高。

2.求出逆波兰表达式之后，可以代入不同的数值进行多次计算，就不用每次都中缀求值了，速度更快。

即因为转换表达式只需一次，而求值可能多次。

3.适合用栈操作运算

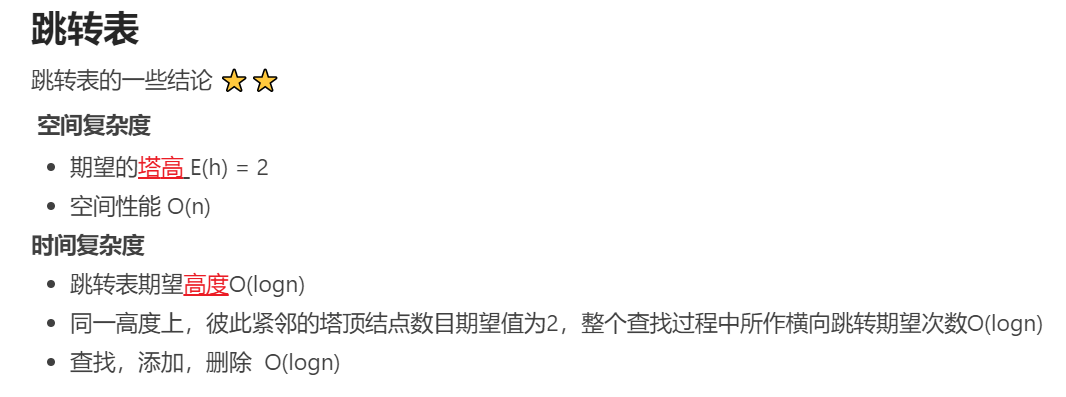
**其他**

**在n个节点的跳转表中，单个词条的期望塔高是θ(logn)。**(2020年)

* 正确
* 错误

错误 ❌。

期望的塔高 E(h) = 2



**函数的调用栈中如果有相同的函数，则他们必然紧邻。**(2019年)

* 正确
* 错误

错误 ❌。

不一定，是否紧邻是看它们是否挨着调用，没有挨着调用，就不会紧邻。

**Fib查找时以前后黄金分割点作为轴点的常系数相同** (2018年)

* 正确
* 错误

错误 ❌。

显然是不同的。因为左右的查找代价不同。

**算法题**

**最大和区间(13')** (2018年)

给定一个整数序列，求出连续子序列和的最大值

1)说明算法思路  
2)伪代码描述算法  
3)说明时间复杂度和空间复杂度  
题注(大致意思)：蛮力算法就不要用啦，是O(n^3),只有设计出O(n)算法才有可能满分，O(n^2)酌情给分。

Leetcode上可以练习这题 【[剑指 Offer 42. 连续子数组的最大和](https://leetcode-cn.com/problems/lian-xu-zi-shu-zu-de-zui-da-he-lcof/)】

如下：时间复杂度O(n)，空间复杂度O(1)

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int tempSum = 0; //作为某连续子数组的累加和

int max = nums[0];

for (int i = 0; i < nums.size(); i++) {

tempSum += nums[i];

if(tempSum > max)

max = tempSum;

if(tempSum < 0) //子数组和小于0,可以丢掉了，因为对后面都是负贡献

tempSum = 0;

}

return max;

}

};

**求一个数组 A 中连续相同数字的和等于 s 的最长子数组长度** (2016年)

例如 A={1,1,2,1,1,1,2,1}，s=3， 则所求子数组长度为 3，要求算法时间复杂度不超过 O(n)，空间复杂度不超过 O(1)

a) 描述算法思想

b) 伪代码实现

c) 计算程序的算法复杂度。

Leetcode上本题的一个简单版本 【[485. 最大连续1的个数](https://leetcode-cn.com/problems/max-consecutive-ones/)】

待解决  待优化

int maxSubLenS(vector<int>& A,int s){

int maxLen = 0,curLen = 0;

int remain = s;

if(remain-A[i] >0 ) {

remain -= A[i];

curLen++;

}

for(int i=1;i<A.size();i++) {

if(A[i] == A[i-1]) {

if(remain > A[i]){

remain -= A[i];

curLen++;

}else if(remain == A[i])

maxLen = curLen > maxLen ? curLen:maxLen;

}else{

if(s >= A[i] ){

remain = s - A[i]；

curLen = 1;

}else if(s == A[i]){

maxLen = 1;

}else {

remain = s;

curLen = 0;

}

}

}

return maxLen;

}