数据结构习题整理

1.介绍

1、数据结构是一门研究非数值计算的程序设计问题中所涉及的计算机的（ ）以及它们之间的关系和运算的学科

A 操作对象

B 计算方法

C 逻辑存储

D 数据映像

2. 要求同一逻辑结构的所有数据元素具有相同的特性, 这意味着( )

A 数据元素具有同一特点

B 不仅数据元素包含的数据项的个数要相同,而且对应数据项的类型要一致

C 每个数据元素都一样

D 数据元素所包含的数据项的个数要相等

3、线性表的顺序存储结构是一种（ ）的存储结构

A 随机存取

B 顺序存取

C 索引存取

D **HASH**存取

4. 算法必须具备输入、输出和( )等五个特征

A 可行性、可移植性和可扩充性

B 可行性、确定性和有穷性

C 确定性、有穷性和稳定性

D 易读性、稳定性和安全性

5. 算法分析的目的是( )

A 找出数据结构的合理性

B 研究算法中的输入和输出之间的关系

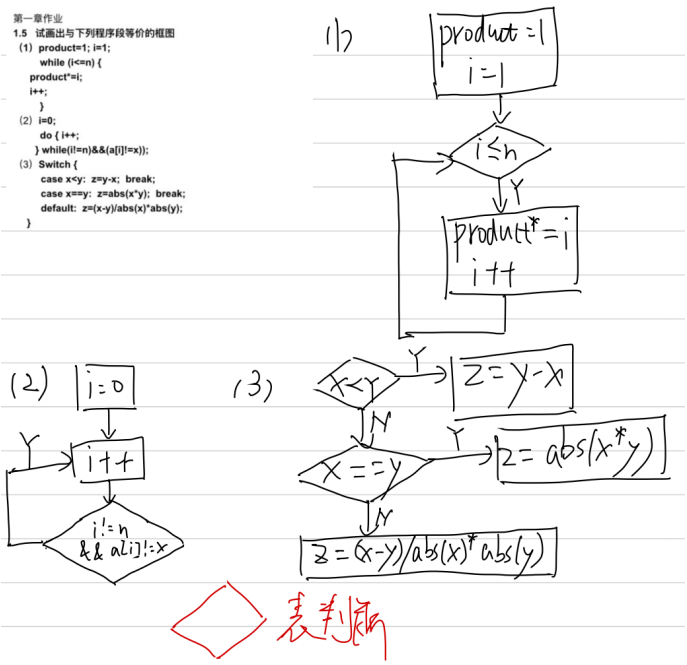
C 分析算法的效率以求改进

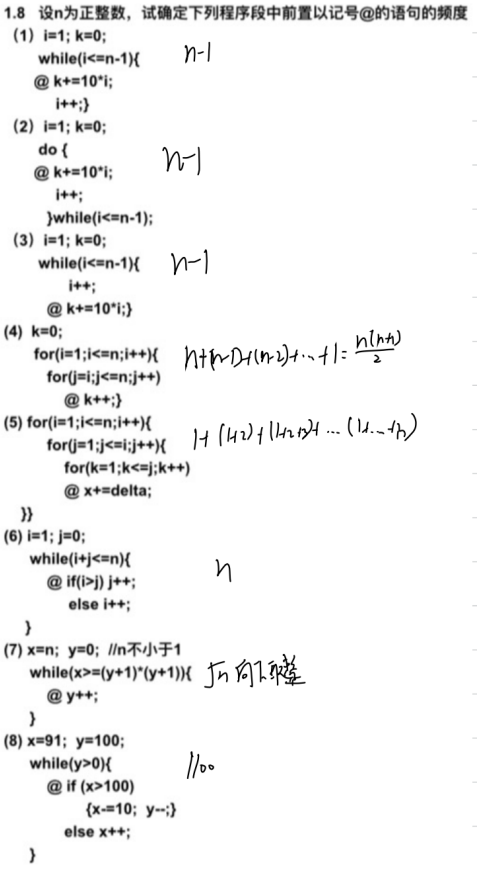
D 分析算法的易懂性和文档性

第一章作业

1.5 试画出与下列程序段等价的框图

```
(1) product=1; i=1;
      while (i<=n) {
    product*=i;
    i++;
      }
(2) i=0;
      do { i++;
    } while(i!=n)&&(a[i]!=x));
(3) Switch {
      case x<y:  z=y-x;  break;
      case x==y:  z=abs(x*y);  break;
      default:  z=(x-y)/abs(x)*abs(y);
    }
```





2.线性表

1、线性表$L=(a_1,a_2,a_3\cdots a_i,\cdots a_n)$，下列说法正确的是（ ）

A 每个元素都有一个直接前驱和直接后继

B 线性表中至少要有一个元素

C 表中诸元素的排列顺序必须是由小到大或由大到小的

D 除第一个元素和最后一个元素外其余每一个元素都有且仅有一个直接前驱和直接后继

2、对于顺序表，以下说法错误的是（ ）

A 顺序表是用一维数组实现的线性表，数组的下标可以看成是元素的绝对地址

B 顺序表的所有存储结点按相应数据元素间的逻辑关系决定的次序依次排列

C 顺序表的特点是：逻辑结构中相邻的结点在存储结构中仍相邻

D 顺序表的特点是：逻辑上相邻的元素，存储在物理位置也相邻的单元中

下表为数组中的位置，而不是绝对地址，绝对地址为内存分配的位置

3、对顺序表上的插入、删除算法的时间复杂度分析来说，通常以（ ）为标准操作。

A 条件判断

B 结点移动

C 算术表达式

D 赋值语句

4、对于顺序表的优缺点， 以下说法错误的是（ ）

A 无需为表示结点间的逻辑关系而增加额外的存储空间

B 可以方便地随机存取表中的任一结点

C 插入和删除操作较方便

D 由于顺序表要求占用连续的空间，存储分配只能预先进行静态分配

5、在一个单链表中，若删除***p**结点的后继结点，则执行（ ）

A q=p->next; p->next=q->next; free(q);

B p=p->next; p->next=p->next->next; free(p);

C p->next=p->next; free(p->next);

D p=p->next->next; free(p->next);

6、在一个单链表中，已知***q**结点是 ***p**结点的前驱结点，若在 ***q** 和 ***p** 之间插入结点***s,** 则执行（ ）

A s->next=p->next ; p->next=s;

B p->next=s->next; s->next=p;

C q->next=s; s->next=p;

D p->next=s; s->next=q;

7、在线性表的下列存储结构中，读取元素花费时间最少的是（ ）

A 单链表

B 双链表

C 循环链表

D 顺序表

3.栈和队列

1、设有一个顺序栈S，元素S1、S2、S3、S4、S5、S6依次入栈，如果6个元素出栈的顺序是S2、S3、S4、S6、S5、S1，则堆栈的容量至少应该是（ ）

A 2

B 3

C 5

D 6

看最多能存储多少个元素

2、一个栈的入栈序列是 a 、b、c 、d 、e 、则栈的不可能的输出序列是（ ）

A e 、d、c 、b 、a

B d 、e、c 、b 、a

C d 、c、e 、a 、b

D a 、b、c 、d 、e

a在b前面

3、以下说法正确的是（ ）

A 因链式栈本身没有容量限制，故在用户内存空间的范围内不会出现栈满的情况

B 因顺序栈本身没有容量限制，故在用户内存空间的范围内不会出现栈满的情况

C 对于链式栈而言，在栈满状态下，如果再进行入栈操作，则会发生“上溢”

D 对于顺序栈而言，在栈满状态下，如果再进行入栈操作，则会发生“下溢”

4、设计一个判断表达式中左、右括号是否匹配的算法，采用（ ）数据结构最佳

A 线性表的顺序存储结构

B 栈

C 队列

D 线性表的链式存储结构

5、已知一个算式的中缀表达式为**a+(b-c)/d**，则其后缀表达式是（ ）

A a+(b-c)/d

B abc-d/+

C bc-d/a+

D a+bc-d/

6、一个队列的入队序列是**1**、**2**、**3**、**4**，则队列的可能的输出序列是（ ）

A 4、3、2、1

B 1、2、3、4

C 1、4、3、2

D 3、2、4、1

7、顺序队列（rear指向队尾元素）的入队操作应该是（ ）

A sq.rear=sq.rear+1; sq.data[sq.rear]=x;

B sq.data[sq.rear]=x; sq.rear=sq.rear+1;

C sq.rear=(sq.rear+1)%maxsize;
sq.data[sq.rear]=x;

D sq.data[sq.rear]=x;
sq.rear=(sq.rear+1)%maxsize;

8、循环队列的队满条件是（ ）

A **(sq.rear+1)%maxsize==(sq.front+1)%maxsize;**

B **(sq.rear+1)%maxsize==sq.front+1;**

C **(sq.rear+1)%maxsize==sq.front;**

D **sq.rear==sq.front;**

9、在一个链式队列中，若f、r 分别为队头、队尾指针，则插入s 所指结点的操作为（ ）

A f->next=s; f=s;

B r->next=s; r=s;

C s->next=r; r=s;

D s->next=f; f=s;

队列

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。

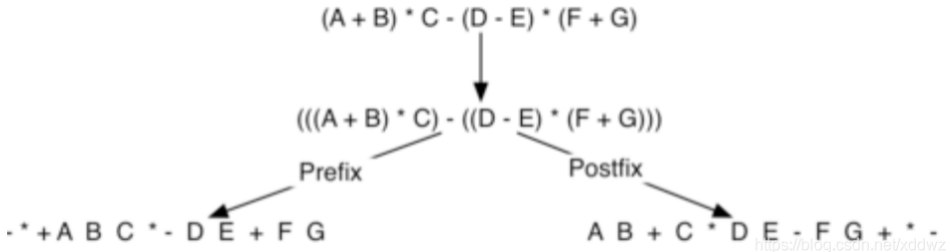
表达式转换

对中缀表达式：A+B，

前缀表达式：将操作符移到前面，变为：+AB
后缀表达式：或将操作符移到后面，变为：AB+

- 无论表达式多复杂，转化成前缀或者后缀时，只需要两个步骤：

1. 将中缀表达式转化为全括号的形式。
2. 将所有的操作符移动到对应的左括号（前缀）或者右括号（后缀）处，替换之，再删除所有的括号。



3.19 假设一个算术表达式中可以包含三种括号：园括号“(”和“)”、方括号“[”和“]”、花括号“{”和“}”，且这三种括号可按任意的次序嵌套使用（如：...[...{...}...[...]...]...[...]...(...)...)。编写判别给定表达式中所含括号是否正确配对出现的算法（已知表达式已存入数据元素为字符的顺序表中）。

```
#include <iostream>
#include <stack>
#include <stdio.h>
using namespace std;
stack<char> a;

int main()
{
    int i, n, p = 0, q = 0;
    char s[1024], b;
    gets_s(s);
    n = strlen(s);
    for (i = 0; i < n; i++)
    {
        if (s[i] == '{' || s[i] == '[' || s[i] == '(')
            a.push(s[i]);
        else if (s[i] == '}')
        {
            if (a.empty())
            {
                p = 1;
                break;
            }
            b = a.top();
            if (b == '{')
                a.pop();
            else
            {
                p = 1;
                break;
            }
        }
        else if (s[i] == ']')
        {
            if (a.empty())
            {
                p = 1;
                break;
            }
            b = a.top();
            if (b == '[')
                a.pop();
            else
            {
                p = 1;
                break;
            }
        }
        else if (s[i] == ')')
        {
            if (a.empty())
            {
                p = 1;
                break;
            }
            b = a.top();
            if (b == '(')
                a.pop();
            else
            {
                p = 1;
                break;
            }
        }
    }
    if (!a.empty())
        q = 1;
    if (p == 1 || q == 1)
        printf("NO\n");
    else
        printf("YES\n");
    return 0;
}
```

4.串

1、串是一种特殊的线性表，其特殊性表现在（ ）

A 可以顺序存储

B 数据元素是一个字符

C 可以链式存储

D 数据元素可以是多个字符

模式匹配

2、设有两个串P和Q，求Q在P中首次出现的位置的操作称为（ ）

A 连接

B 模式匹配

C 求子串

D 求串长

3、设串S1= ‘ABCDEFG’ ,S2= ‘PQRST，函数stringconcat(X,Y)返回X和Y的连接串，substring(S,i,j)返回串S中从序号i的字符开始的j个字符组成的子串，stringlength(S)返回串S的长度。则stringconcat(substring(S1,2,stringlength(S2)),substring(S1,stringlength(s2),2))的结果串是（ ）

A BCDEF

B BCDEFG

C BCDQRST

D BCDEFEF

4、KMP算法的最大特点是指示主串的指针不需要回溯

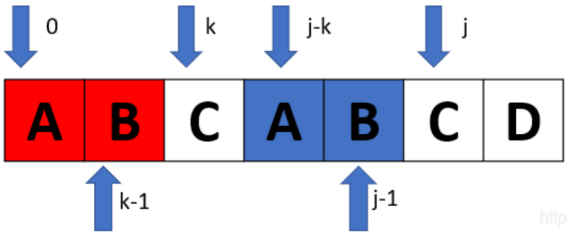
A 正确

B 错误

KMP 算法是 D.E.Knuth、J,H,Morris 和 V.R.Pratt 三位神人共同提出的，称之为 Knuth-Morria-Pratt 算法，简称 KMP 算法。该算法相对于 Brute-Force（暴力）算法有比较大的改进，主要是消除了主串指针的回溯，从而使算法效率有了某种程度的提高。

1. 特殊情况
   当 j 的值为 0 或 1 的时候，它们的 k 值都为 0，即 next[0] = 0、next[1] =0。但是为了后面 k 值计算的方便，我们将 next[0] 的值设置成 -1。

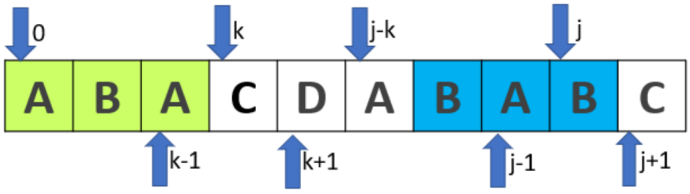
2. 当 t[j] == t[k] 的情况
   举个栗子



   观察上图可知，当 t[j] == t[k] 时，必然有"t[0]…t[k-1]" == " t[j-k]…t[j-1]"，此时的 k 即是相同子串的长度。因为有"t[0]…t[k-1]" == " t[j-k]…t[j-1]"，且 t[j] == t[k]，则有"t[0]…t[k]" == " t[j-k]…t[j]"，这样也就得出了next[j+1]=k+1。

3. 当t[j] != t[k] 的情况
   关于这种情况，在代码中的描述就是“简单”的一句 k = next[k]; 。我当时看了之后，感觉有点蒙，于是就去翻《数据结构教程》。但是这本书里，对于这行代码的解释只有三个字：k 回退…！于是我从“有点蒙”的状态升级到了“很蒙蔽”的状态，我心想，k 回退？我当然知道这是 k 退回，但是它为什么要会退到 next[k] 的位置？为什么不是回退到k-1？？？巴拉巴拉巴拉…此处省略一万字。

   我绞尽脑汁，仍是不得其解。于是我就去问度娘…
   在我看了众多博客之后，终于有了一种拨云见日的感觉，看下图



   由第2中情况可知，当 t[j] == t[k] 时，t[j+1] 的最大子串的长度为 k，即 next[j+1] = k+1。但是此时t[j] != t[k] 了，所以就有 next[j+1] < k，那么求 next[j+1] 就等同于求 t[j] 往前小于 k 个的字符（包括t[j]，看上图蓝色框框）与 t[k] 前面的字符（绿色框框）的最长重合串，即 t[j-k+1] ~ t[j] 与 t[0] ~ t[k-1] 的最长重合串（这里所说“最长重合串”实不严谨，但你知道是符合 k 的子串就行…），那么就相当于求 next[k]（只不过 t[k] 变成了 t[j],但是 next[k] 的值与 t[k] 无关）!!!。所以才有了这句 k = next[k]，如果新的一轮循环（这时 k = next[k] ，j 不变）中 t[j] 依然不等于 t[k] ，则说明倒数第二大 t[0~next[k]-1] 也不行，那么 k 会继续被 next[k] 赋值（这就是所谓的 k 回退…），直到找到符合重合的子串或者 k == -1。

至此，算是把求解数组 next 的算法弄清楚了（其实是，终于把 k = next[k] 弄懂了…）

因为这个算法神奇难解之处就在k=next[k]这一处的理解上，网上解析的非常之多，有的就是例证，举例子按代码走流程，走出结果了，跟肉眼看的一致，就认为解释了为什么k=next[k]；很少有看到解释的非常清楚的，或者有，但我没有仔细和耐心看下去。我一般扫一眼，就大概知道这个解析是否能说的通。仔细想了三天，搞的千转百折，山重水复，一头雾气缭绕的。搞懂以后又觉得确实简单，但是绕人，烧脑。

KMP算法讲解：https://www.bilibili.com/video/BV1jb411V78H

1、字符串S1='abcdefghijklmnopqrstuvw'，由如下运算分别得到S2和S3，请给出S2和S3的值

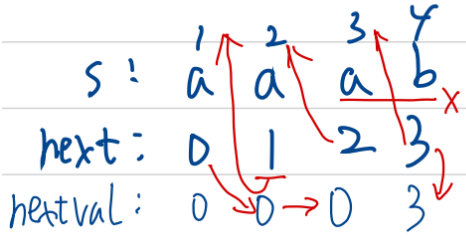
合并字符串 S1中从1位数起取3个字符

S2=StringConcat(substring(S1,19,3),substring(S1,4,2),substring(S1,14,1),substring(S1,20,1))

S3=substring(S1,stringlength(S2),stringlength(S2))

字符串长度

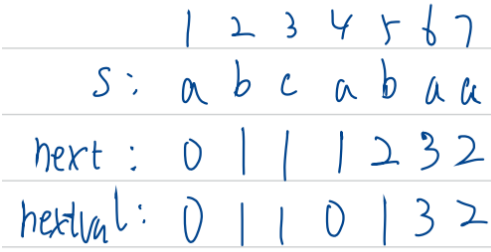
2、令s='aaab'，t='abcabaa'，分别求出它们的next函数值和nextval函数值。



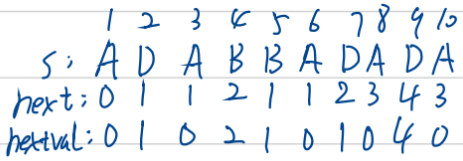
next[1]和next[2]永远为0，1

next值的最左与最右匹配最大字符数+1

nextval未活 不匹配为next值，匹配则为指向nextval值



3、已知模式串pat='ADABBADADA'，写出模式串的next函数值和nextval函数值



5.数组

1、二维数组A[10..20, 5..10]采用行序为主序的方式存储，每个数据元素占4个存储单元，并且A[10,5]的存储地址是1000，则A[18,9]的存储地址是 （ ）

A 1208

B 1212

C 1368

D 1364

1000+（18-10）×4×（10-5+1）+（9-5）*4=1208

2、二维数组**A**中，每个元素的长度为**4**个字节，行下标从**0**到**4**，列下标从**0**到**5**， **A**按行序为主序存储时元素**A[3,5]**的地址与**A**按列序为主序存储时元素（ ）的地址相同。

A A[2,4]

B A[3,4]

C A[3,5]

D A[4,4]

[0...4,0...5], 好难算...画图数就完事了

3、对矩阵压缩存储是为了（ ）

A 方便运算

B 节省空间

C 方便存储

D 提高运算速度

4、稀疏矩阵的压缩存储方法通常有两种，即（ ）

A 二元数组和三元数组

B 三元组和散列

C 三元组和十字链表

D 散列和十字链表

4、二维数组**A[1..10, 1..20]**采用列序为主序方式存储，每个元素占一个存储单元，并且**A[1,1]**的存储地址是**200**，则**A[6,12]**的地址是[填空1]

填空1:315

200+(12-1)×10+(6-1)=315

6.树和二叉树

1、设深度为**K**的二叉树上只有度为**0**和度为**2**的结点，则这类二叉树上所含结点总数最少（ ）个

A k+1

B 2k

C 2k-1

D 2k+1

度：该节点的孩子节点的个数

2、深度为**6**的二叉树最多有（ ）个结点

A 64

B 63

C 32

D 31

二叉树最多节点数量：2的深度次方-1

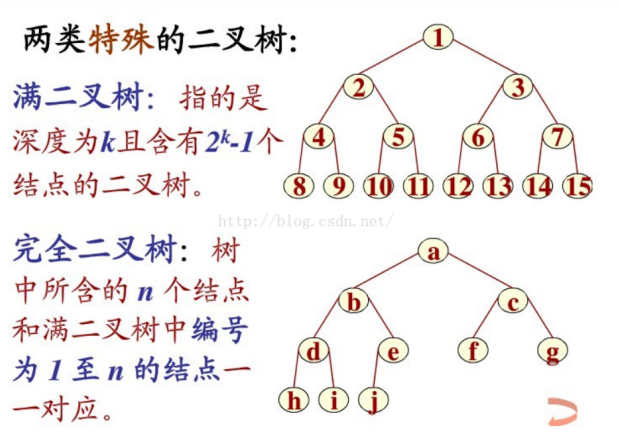
3、已知完全二叉树有**26**个结点，则整棵二叉树有（ ）个度为**1**的结点。

A 0

B 1

C 2

D 13



画出完全二叉树即可求得

4、设二叉树有**n**个结点，则其深度为（ ）

A n-1

B n

C $\lfloor \log_2 n \rfloor + 1$

D 无法确定

没有规定是什么二叉树

5、设有一棵**22**个结点的完全二叉树，那么整颗二叉树有（ ）个度为**0**的结点。

A 6

B 7

C 8

D 11

画完全二叉树之后再数

6、任何一棵二叉树的叶子结点在其先序、中序、后序遍历序列中的相对位置（ ）

A 肯定发生变化

B 有时发生变化

C 肯定不发生变化

D 无法确定

相对次序发生变化的都是子树的根,也就是分支结点

7、已知某二叉树的先序遍历的结点访问顺序是abdgcefh，中序遍历的结点访问顺序是dgbaechf，则其后序遍历的结点访问顺序是（ ）

A bdgcefha

B gdbecfha

C bdgechfa

D gdbehfca

前序遍历就是先访问根节点，再访问左节点，最后访问右节点，
中序遍历就是先访问左节点，再访问根节点，最后访问右节点，
后序遍历就是先访问左节点，再访问右节点，最后访问根节点。

直接把二叉树画出来

8、n个结点的线索二叉树中的线索数目为（ ）个

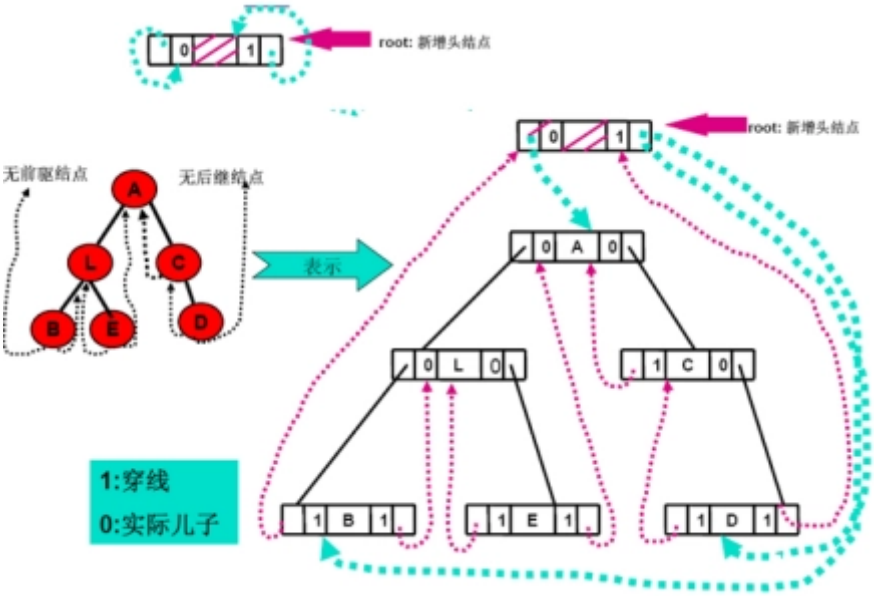
A n-1

B n

C n+1

D n+2

对于n个结点的二叉树，在二叉链存储结构中有n+1个空链域，利用这些空链域存放在某种遍历次序下该结点的前驱结点和后继结点的指针，这些指针称为线索，加上线索的二叉树称为线索二叉树。



这种加上了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树(Threaded BinaryTree)。根据线索性质的不同，线索二叉树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树三种。

n个节点的线索二叉树中的线索数目为n+1

9、树最适合用来表示（ ）

A 有序数据元素

B 无序数据元素

C 元素之间具有分支层次关系的数据

D 元素之间无联系的数据

10、设森林T中有4棵树，结点个数分别是n1,n2,n3,n4，当把森林T转换成一棵二叉树后， 根结点的右子树上有（ ）个结点

A n1-1

B n1

C n1+n2+n3

D n2+n3+n4

只有第一棵数在左子树上，右子树为剩下的树

11、设有30个权值，用它们构造一棵哈夫曼树，则该哈夫曼树中共有多少个（ ）结点。

A 59

B 60

C 58

D 61

n个权值构造出的哈夫曼树共有2n-1个节点

12、以数据集{4、5、6、7、10}为叶子节点的权值所构造的哈夫曼树的WPL为 [填空1]

WPL=73

构造之后将各叶子结点×各路径长度=WPL

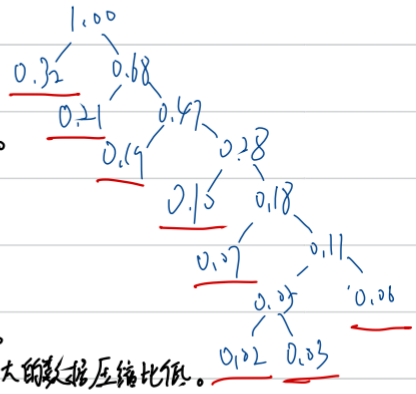
哈夫曼树

6.26 假设用于通信的电文仅由8个字母组成，字母在电文中出现的频率分别为0.07,0.19,0.02,0.06,0.32,0.03,0.21,0.10。试为这8个字母设计哈夫曼编码。使用0—7的二进制表示形式是另一种编码方案，对于上述实例，比较两种方案的优缺点。

频率：0.32 0.21 0.19 0.10 0.07 0.06 0.03 0.02

哈夫曼编码：0 10 110 1110 11110 111110 1111101 1111100

二进制：000 001 010 011 100 101 110 111



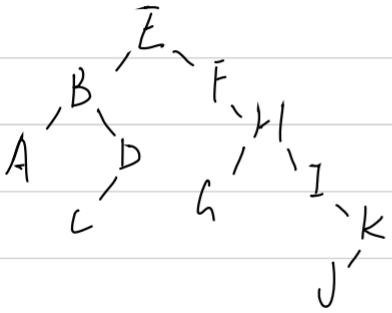
哈夫曼编码占用空间：$1\times0.32+2\times0.21+3\times0.19+4\times0.10+5\times0.07+6\times0.06+7\times(0.03+0.02)$
$=2.77$

二进制占用空间：$3\times(0.32+0.21+0.19+0.1+0.07+0.06+0.03+0.02)$
$=3$

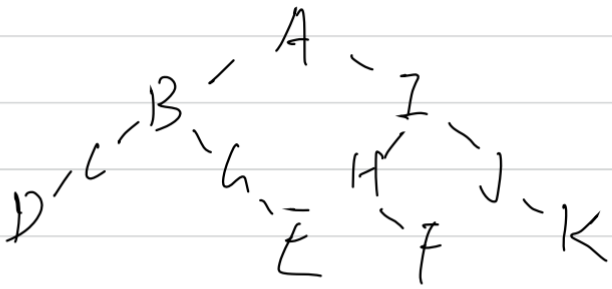
哈夫曼编码：优点：对出现概率较大数字差距较大的数据更易压缩数据，压缩比大。
缺点：需要先遍历一遍构建频次表，效率较低。且对数字差异不大的数据压缩比低。

二进制编码：优点：简单，效率高，压缩比稳定。
缺点：对大量数据处理较慢，压缩比普通低。

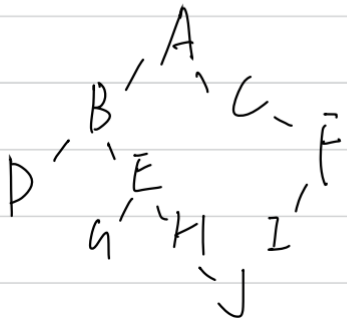
6.27 假设一颗二叉树的先序序列为EBADCFHGIKJ和中序序列为ABCDEFGHIJK，请画出该树。



6.28 假设一颗二叉树的中序序列为DCBGEAHFIJK和后序序列为DCEGBFHKJIA，请画出该树。



6.29 假设一颗二叉树的层序序列为ABCDEFGHIJ和中序序列为DBGEHJACIF，请画出该树



6.65 已知一颗二叉树的前序序列和中序序列分别存于两个一维数组中，试编写算法建立该二叉树的二叉链表。

```
public:
    TreeNode *buildTree(vector<int> &preorder, vector<int>
&inorder) {
        if(preorder.empty()||inorder.empty())
        return NULL;
        return
build(preorder,inorder,0,preorder.size()-1,0,inorder.size()-1);
    }
    TreeNode* build(vector<int> preorder,vector<int>
inorder,int preL,int preR,int inL,int inR){
        if(preL>preR)
        return NULL;
//先序第一个值即为根的值
        int rootValue=preorder[preL];
        TreeNode* root=new TreeNode(rootValue);

        //寻找左右子树的划分
        int k;
        for(int i=0;i<inorder.size();++i){
            if(rootValue==inorder[i]){
                k=i;
                break;
            }
        }
        int numLeft=k-inL;
        root-
>left=build(preorder,inorder,preL+1,preL+numLeft,inL,k-1);
        root-
>right=build(preorder,inorder,preL+numLeft+1,preR,k+1,inR);
        return root;
    }
};
```

6.68 已知一颗树的由根至叶子结点按层次输入的结点序列及每个结点的度（每层中自左至右输入），试写出构造此树的孩子—兄弟链表的算法

```
CSTree CreateCSTreeLevelDegree(char *levelstr, int *num) {
 int cnt,i,parent;
 CSNode *p;
 CSNode *tmp[maxSize];

 //创建结点
 for (i=0; i < strlen(levelstr); ++i) {
  p = (CSNode *)malloc(sizeof(CSNode)); if (!p) exit(OVERFLOW);
  p->data = levelstr[i];p->firstchild=p->nextsibling=NULL;
  tmp[i]=p;
 }

 //连接
//父母
 parent=0;
 //找到孩子的个数
 cnt=0;
 i=1;

 while (i<strlen(levelstr)) {
  if (num[parent]==0 || cnt==num[parent]) { //这个父母没有孩子 || parent的孩子
已经找完了
   cnt=0; //计数器归0
   parent++; //位移一位
   continue;
  }
  //这个父亲有孩子（i是parent的孩子）
  cnt++;
  if (cnt==1) { //i是parent的第一个孩子
   tmp[parent]->firstchild = tmp[i];
  } else { //不是第一个孩子
   tmp[i-1]->nextsibling = tmp[i]; //它是前面的兄弟
  }

  i++;
 }

 return tmp[0];
}
```

7.图

1、在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和的（ ）倍

A 1/2

B 1

C 2

D 4

有向图中，入度之和=出度之和

2、用邻接矩阵存储图时，所占用的存储空间大小仅与图中的结点个数有关

A 对

B 错

3、对于一个具有N个顶点和E条边的无向图，若采用邻接表表示，则所有邻接表中的表结点总数是 （ ）

A E/2

B E

C 2E

D N+E

N个定点和E条边的无向图，邻接表中的表节点总数为2E

4、采用邻接表存储的图的广度优先遍历算法类似于二叉树的（ ）遍历

A 先序遍历

B 中序遍历

C 后序遍历

D 层次遍历

深度优先类似先序遍历，广度优先类似层次遍历

5、任何一个带权的无向连通图的最小生成树（ ）

A 只有一棵

B 有一棵或多棵

C 一定有多棵

D 可能不存在

任何一个带权的无向连通图的最小生成树有一棵或多棵

6、对于含有n个顶点e条边的无向连通图。利用kruskal算法生成最小生成树其时间复杂度为（ ）

A O(n)

B O(eloge)

C O(n*n)

D O(n*e)

7、一个有向图G中若有弧<vi,vj>、<vj,vk> 和<vi,vk>，则在图G的拓扑序列中，顶点vi, vj,和vk的相对位置为vi, vj, vk

A 对

B 错

8、以下说法正确的是（ ）

A 连通图的生成树是该连通图的一个极小连通子图

B 无向图的邻接矩阵是对称的，有向图的邻接矩阵一定是不对称的

C 任何一个有向图，其全部顶点可以排成一个拓扑序列

D 有回路的图不能进行拓扑排序

9、判断一个有向图是否存在回路除了可以利用拓扑排序方法以外， 还可以利用（ ）

A 求关键路径的方法

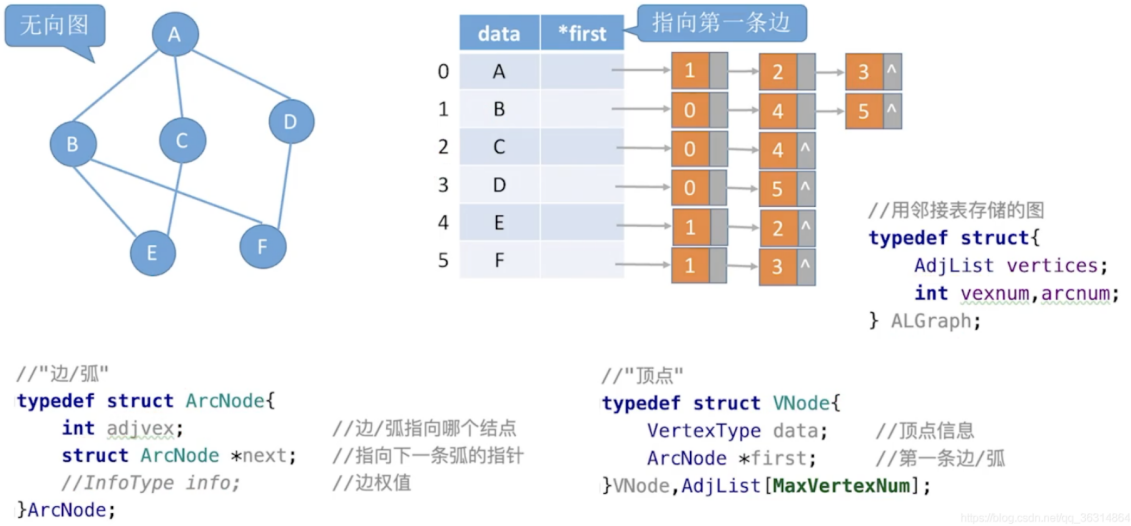
B 求最短路径的Dijkstra方法

C 广度优先遍历算法

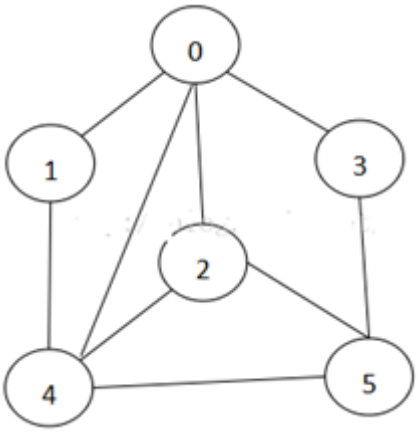
D 深度优先遍历算法

邻接表（顺序+链式存储）

当一个图为稀疏图时，使用邻接矩阵表示显然要浪费大量的存储空间，而图的邻接表法结合了顺序存储和链式存储方法，大大减少了这种不必要的浪费。



邻接矩阵和邻接表、深度优先遍历和广度优先遍历



转化成邻接矩阵，则表示如下：

$$
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\left[\begin{array}{cccccc}
0 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0
\end{array}\right]
\end{array}
$$

0号顶点到2号顶点存在边

深度优先遍历基本思想

图的深度优先搜索(Depth First Search) 。

1) 深度优先遍历，从初始访问结点出发，初始访问结点可能有多个邻接结点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始结点，访问它的第一个邻接结点， 可以这样理解：每次都在访问完**当前结点**后首先访问**当前结点的第一个邻接结点**。
2) 我们可以看到，这样的访问策略是优先往纵向挖掘深入，而不是对一个结点的所有邻接结点进行横向访问。
3) 显然，深度优先搜索是一个递归的过程

```
/**
    * 深度优先遍历，此时不考虑起始点，即以0号序列的顶点为起始顶点
```

```
     */
    public void dfs() {
        System.out.println("Graph.dfs");
        boolean[] visited = new boolean[numberOfVertex];
        Arrays.fill(visited, false);
        for (int i = 0; i < numberOfVertex; i++) {
            if (!visited[i]) {
                dfs(i, visited);
            }
        }
        System.out.println();
    }
    /**
     * 从指定顶点进行深度优先遍历
     *
     * @param vertex 开始顶点的序号
     */
    public void dfs(int vertex) {
        boolean[] visited = new boolean[numberOfVertex];
        Arrays.fill(visited, false);

        dfs(vertex, visited);
        System.out.println();
    }
    /**
     * @param vertex 深度优先遍历的开始顶点所在的序号
     */
    private void dfs(int vertex, boolean[] visited) {
        System.out.print(vertexs[vertex] + "->");
        visited[vertex] = true;
        int w = getFirstNeighbour(vertex);
        while (w != -1) {
            if (!visited[w]) {
                dfs(w, visited);
            } else {
                // 如果w已经被访问过，则访问w的下一个邻接顶点
                w = getNextNeighbour(vertex, w);
            }
        }
    }
```

深度优先遍历算法步骤

1) 访问初始结点v，并标记结点v为已访问。
2) 查找结点v的第一个邻接结点w。
3) 若w存在，则继续执行4，如果w不存在，则回到第1步，将从v的下一个结点继续。
4) 若w未被访问，对w进行深度优先遍历递归（即把w当做另一个v，然后进行步骤123）。
5) 查找结点v的w邻接结点的下一个邻接结点，转到步骤3。

广度优先遍历基本思想

图的广度优先搜索(Broad First Search)。
类似于一个分层搜索的过程，广度优先遍历需要使用一个队列以保持访问过的结点的顺序，以便按这个顺序来访问这些结点的邻接结点

广度优先遍历算法步骤

1) 访问初始结点v并标记结点v为已访问。
2) 结点v入队列
3) 当队列非空时，继续执行，否则算法结束。
4) 出队列，取得队头结点u。
5) 查找结点u的第一个邻接结点w。
6) 若结点u的邻接结点w不存在，则转到步骤3；否则循环执行以下三个步骤：
6.1 若结点w尚未被访问，则访问结点w并标记为已访问。
6.2 结点w入队列
6.3 查找结点u的继w邻接结点后的下一个邻接结点w，转到步骤6。

```
/**
     * 广度优先遍历
     */
    public void bfs() {
        System.out.println("Graph.bfs");

        boolean[] visited = new boolean[numberOfVertex];
        Arrays.fill(visited, false);

        for (int i = 0; i < numberOfVertex; i++) {
            if (!visited[i]) {
                bfs(i, visited);
            }
        }
    }
    /**
     * 从指定顶点vertex开始进行广度优先遍历
     *
     * @param vertex 从vertex顶点开始进行广度优先遍历
     */
    public void bfs(int vertex) {
        boolean[] visited = new boolean[numberOfVertex];
        Arrays.fill(visited, false);

        bfs(vertex, visited);

    }
    /**
     * 从顶点vertex开始进行广度优先遍历
     *
     * @param vertex  顶点序号
     * @param visited 辅助遍历数组
     */
    private void bfs(int vertex, boolean[] visited) {
        System.out.print(vertexs[vertex] + "->");
        visited[vertex] = true;

        LinkedList<Integer> queue = new LinkedList<>();
        queue.addLast(vertex);
        while (!queue.isEmpty()) {
```

```
            // 此时head所在的顶点已经访问过了
            int head = queue.remove();
            int w = getFirstNeighbour(head);

            while (w != -1) {
                if (!visited[w]) {
                    // 深度优先遍历从此处开始递归，但广度优先不进行递归
                    System.out.print(vertexs[w] + "->");
                    visited[w] = true;
                    queue.addLast(w);
                }
                w = getNextNeighbour(head, w);
            }
        }
    }
```

7.22

试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点**vi****到顶点****vj****的路径****(i<>j)****。注意：算法中涉及的图的基本操作必须在此存储结构上实现。**

```
#include <stdio.h>
#include <stdlib.h>
//弧结点
typedef struct ArcNode {
  int ajdVex;  //弧终点指针位置
  ArcNode* next;   //下一条弧的指针
};

//顶点结点
typedef struct Node {
  ArcNode* Arc;   //指向依附该顶点的弧指针
};

//邻接表
typedef struct Graph {
  Node adjlist[100];   //点集
  bool visited[100];   //是否被遍历
  int n, m;  //顶点数和弧数
};

Graph* G = (Graph*)malloc(sizeof(Graph));
int i, j, k; bool hasPath = false;
//创建邻接表
void createLink() {
  printf("请输入顶点数n 弧数m: ");
  scanf("%d %d", &G->n, &G->m);
  printf("创建的所有点: ");
  for (i = 0; i < G->n; i++) { G->adjlist[i].Arc = 0; printf("V%d ", i);
}printf("\n");
  for (j = 0; j < G->m; j++) {
    printf("第%d条弧起点编号、终点编号: ", j + 1);
    scanf("%d %d", &i, &k);
    ArcNode* p = (ArcNode*)malloc(sizeof(ArcNode));
    p->ajdVex = k; //将边所指向的点编号赋值
    p->next = G->adjlist[i].Arc;
```

```
    G->adjlist[i].Arc = p;
  }
}
//输出邻接表
void print_Graph() {
  ArcNode* p;
  for (i = 0; i < G->n; i++) {
    p = G->adjlist[i].Arc;
    printf("V%d", i);
    while (p) {
      printf("->V%d", p->ajdVex);
      p = p->next;
    }printf("\n");
  }
}
//两点是否连通
void existPath(int d, int z) {
  G->visited[d] = true;
  ArcNode* p = G->adjlist[d].Arc;
  if (d == z)hasPath = true;
  while (p) {

    if (p->ajdVex == z)hasPath = true;
    if (!G->visited[p->ajdVex])existPath(p->ajdVex, z);
    p = p->next;
  }
}
//输出两点连接路径

void print_Path(int d, int z) {
  ArcNode* p = G->adjlist[d].Arc;
  printf("路径为：V%d", d);
  while (p) {
    printf("->V%d", p->ajdVex);
    if (p->ajdVex == z)break;
    p = G->adjlist[p->ajdVex].Arc;
  }printf("\n");

}
//visited置零
void setFlase() {
  hasPath = false;
  for (i = 0; i < G->n; i++) G->visited[i] = false;
}

int main() {
  createLink();
  while (true) {
    printf("\n");
    printf("请输入操作(1.打印信息 2.判断路径 3.退出)：");
    int choice; scanf("%d", &choice); setFlase();
    if (choice == 1) print_Graph();
    if (choice == 2) {
      printf("请输入两点编号："); scanf("%d %d", &i, &k); existPath(i, k);
      if (hasPath) { printf("两点连通!\n"); print_Path(i, k); }
      else printf("两点不连通!\n");
    }
    if (choice == 3)break;
```

```
  }
  return 0;
}
```

7.23

同**7.22** **题要求，试基于广度优先搜索策略写一算法。**

```
BfsReachable(ALGraph g, int i, int j) {

  Queue q;
  int k, n;
  ArcNode* p;
  InitQueue(q);
  EnQueue(q, i);
  while (!QueueEmpty(q))
  {
•     DeQueue(q, k);
•     visited[k] = 1;
•     for (p = g.vertices[k].firstarc; p; p = p->nextarc)
•     {
•       n = p->adjvex;
•       if (n == j)return 1;
•       if (visited[n] != 1)EnQueue(q, n);
•     }
  }
  return 0;
}
```

7.27

采用邻接表存储结构，编写一个判别无向图中任意给定的两个顶点之间是否存在一条长度为**k****的简单路径的算法。**

```
void DFS(ALGraph G, int i, int j, int k, Status on[], Status& found)

{
  Static int n = 0;//记录当前路径上的顶点个数
  ArcNode* p;
  on[i] = TRUE;
  n++;
  if (i == j && n == k + 1)
•     found = TRUE;
  else
  {
•     p = G.vertices[i].firstarc;
•     while (!found && p != NULL)
•     {
•       if (!on[p->adjvex])
•         DFS(G, p->adjvex, j, k, on, found);
•       p = p->nextarc;
•     }
  }
  on[i] = FALSE;//回退操作
  n--;
}
```

```
Status SimplePath(ALGraph G, int i, int j, int k)
{
  int m;
  Status on[MAX_VERTEX_NUM];
  Status found;
  for (m = 1; m <= G.vexnum; m++)
      on[m] = FALSE;
  found = FALSE;
  DFS(G, i, j, k, on, found);
  return found;
}
```

9.搜索

1、对采用折半查找法进行查找操作的查找表，要求按（ ）方式进行存储

A 顺序存储

B 链式存储

C 顺序存储并且结点按关键字有序

D 链式存储并且结点按关键字有序

2、如果要求一个线性表既能较快地查找，又能适应动态变化（插入删除方便）的要求，可以采用（ ）查找方法

A 分块

B 顺序

C 折半

D 哈希

3、设有n个结点的二叉排序树中查找一个元素时，最坏情况下的时间复杂度为（ ）

A O(n)

B O(1)

C $O(\log_2 n)$

D O(n^2)

4、有数据（49，32，40，6，45，12，56），从空二叉树开始依次插入数据形成二叉排序树，若希望高度最小，则应该选择下列（ ）输入序列

A 45，12，49，6，40，56，32

B 40，12，6，32，49，45，56

C 6，12，32，40，45，49，56

D 32，12，6，40，45，56，49

按升序输入序列

5、已知一棵深度为K的平衡二叉树，其每一个非终端结点的平衡因子均为0，则该树共有（ ）结点

A $2^{(K-1)}-1$

B $2^{(K-1)}+1$

C $2^{k}-1$

D $2^{k}+1$

深度为k的平衡二叉树，其每个非终端节点的平衡因子均为0,该树共有2^k - 1个节点

6、深度为6的AVL树至少有（ ）个结点

A 10

B 12

C 20

D 21

对于一棵平衡树，如果以N_h表示深度为h时含有的最少结点数。有如下的规律：

$$N_0 = 0, N_1 = 1, N_2 = 2;$$
$$N_h = N_{h-1} + N_{h-2} + 1$$

7、m阶B-树每一个结点的子树个数大于或等于m/2

A 对

B 错

B-树即为B树

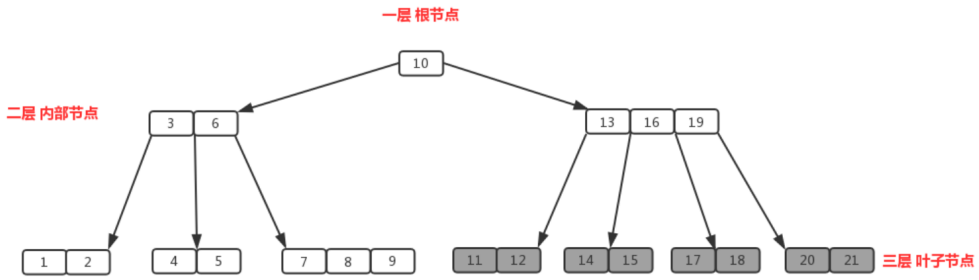
B树是一种平衡的多分树，通常我们说m阶的B树，它必须满足如下条件：

- 每个节点最多只有m个子节点。
- 每个非叶子节点（除了根）具有至少⌈ m/2⌉子节点。
- 如果根不是叶节点，则根至少有两个子节点。
- 具有*k*个子节点的非叶节点包含*k* -1个键。
- 所有叶子都出现在同一水平，没有任何信息（高度一致）。

第一次看到这个定义的时候，在想什么鬼？。。。。什么是阶？子节点、飞叶子点、根？？？啥意思！少年别慌。。。

什么是B树的阶？

B树中一个节点的子节点数目的最大值，用m表示，假如最大值为10，则为10阶，如图



所有节点中，节点【13,16,19】拥有的子节点数目最多，四个子节点（灰色节点），所以可以定义上面的图片为4阶B树，现在懂什么是阶了吧

什么是根节点？

节点【10】即为根节点，特征：根节点拥有的子节点数量的上限和内部节点相同，如果根节点不是树中唯一节点的话，至少有俩个子节点（不然就变成单支了）。在m阶B树中（根节点非树中唯一节点），那么有关系式2<= M <=m，M为子节点数量；包含的元素数量 1<= K <=m-1,K为元素数量。

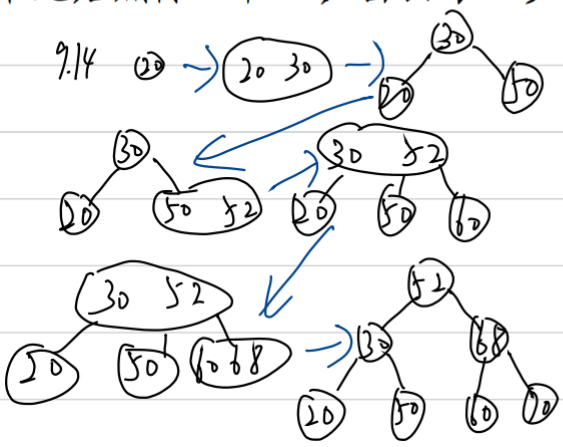
什么是内部节点？

节点【13,16,19】、节点【3,6】都为内部节点，特征：内部节点是除叶子节点和根节点之外的所有节点，拥有父节点和子节点。假定m阶B树的内部节点的子节点数量为M，则一定要符合（m/2）<= M <=m关系式，包含元素数量M-1；包含的元素数量 （m/2）-1<= K <=m-1,K为元素数量。m/2向上取整。

什么是叶子节点?

节点【1,2】、节点【11,12】等最后一层都为叶子节点，叶子节点对元素的数量有相同的限制，但是没有子节点，也没有指向子节点的指针。特征：在m阶B树中叶子节点的元素符合（m/2）-1<= K <=m-1。

9.14 试从空树开始，画出按以下次序向2—3树即3阶B—树中插入关键码的建树过程：20、30、50、52、60、68、70，如果此后删除50和68，画出每一步执行后2—3树的状态。



8、m阶B-树具有k个子树的非叶子结点含有k-1个关键字

- A 对
- B 错

9、有K个关键字互为同义词（即求出的HASH函数的值都一样），若用线性探测把这K个关键字存入哈希表中，至少要进行（ ）次探测

- A K-1
- B K
- C K+1
- D K*(K+1)/2

10、哈希表的平均查找长度（ ）

- A 与处理冲突的方法有关，而与表的长度无关
- B 与处理冲突的方法无关，而与表的长度有关
- C 与处理冲突的方法有关，并且与表的长度有关
- D 与处理冲突的方法无关，并且与表的长度无关

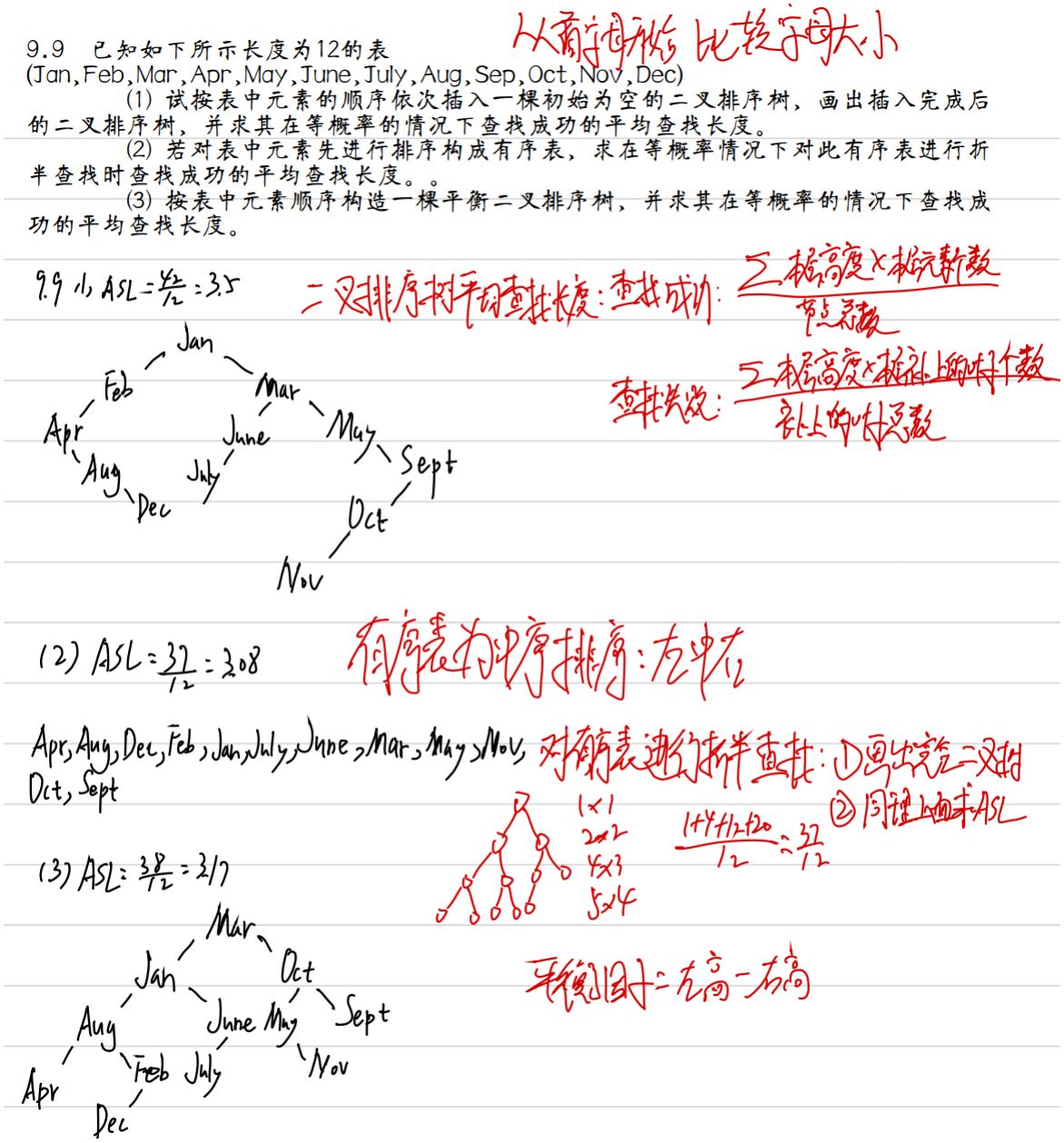
二叉排序树

二叉排序树要么是空二叉树，要么具有如下特点：

- 二叉排序树中，如果其根结点有左子树，那么左子树上所有结点的值都小于根结点的值；
- 二叉排序树中，如果其根结点有右子树，那么右子树上所有结点的值都大小根结点的值；
- 二叉排序树的左右子树也要求都是二叉排序树；

平衡二叉树

1. 可以是空树。
2. 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过1。



哈希表

散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

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。

构造好的哈希函数的方法，应能使冲突尽可能地少，因而应具有较好的随机性。这样可使一组关键字的散列地址均匀地分布在整个地址空间。根据关键字的结构和分布的不同，可构造出许多不同的哈希函数。

1.直接定址法

直接定址法是以关键字k本身或关键字加上某个数值常量c作为哈希地址的方法。该哈希函数H(k)为：

H(k)=k+c (c≥0)

这种哈希函数计算简单，并且不可能有冲突发生。当关键字的分布基本连续时，可使用直接定址法的哈希函数。否则，若关键字分布不连续将造成内存单元的大量浪费。

2.除留余数法

取关键字k除以哈希表长度m所得余数作为哈希函数地址的方法。即：

H(k)=k%m

这是一种较简单、也是较常见的构造方法。这种方法的关键是选择好哈希表的长度m。使得数据集合中的每一个关键字通过该函数转化后映射到哈希表的任意地址上的概率相等。理论研究表明，在m取值为素数（质数）时，冲突可能性相对较少。

3.平方取中法

取关键字平方后的中间几位作为哈希函数地址（若超出范围时，可再取模）。

4.折叠法

这种方法适合在关键字的位数较多，而地址区间较小的情况。

将关键字分隔成位数相同的几部分。然后将这几部分的叠加和作为哈希地址（若超出范围，可再取模）。

例如，假设关键字为某人身份证号码430104681015355，则可以用4位为一组进行叠加。即5355+8101+1046+430=14932，舍去高位。则有H(430104681015355)=4932为该身份证关键字的哈希函数地址。

处理冲突：

1.开放定址法

用开放定址法处理冲突就是当冲突发生时，形成一个地址序列。沿着这个序列逐个探测，直到找出一个“空”的开放地址。将发生冲突的关键字值存放到该地址中去。

如 Hi=(H(k)+d（i）) % m, i=1，2，...k (k<m-1)

其中H(k)为哈希函数，m为哈希表长，d为增量函数，d(i)=dl，d2...dn-l。

增量序列的取法不同，可得到不同的开放地址处理冲突探测方法。

（1）线性探测法

线性探测法是从发生冲突的地址（设为d）开始，依次探查d+l，d+2，...m-1（当达到表尾m-1时，又从0开始探查）等地址，直到找到一个空闲位置来存放冲突处的关键字。

若整个地址都找遍仍无空地址，则产生溢出。

线性探查法的数学递推描述公式为：

d0=H(k)

di=(di-1+1)% m (1≤i≤m-1)

（2）平方探查法

设发生冲突的地址为d，则平方探查法的探查序列为：d+12，d+22，...直到找到一个空闲位置为止。

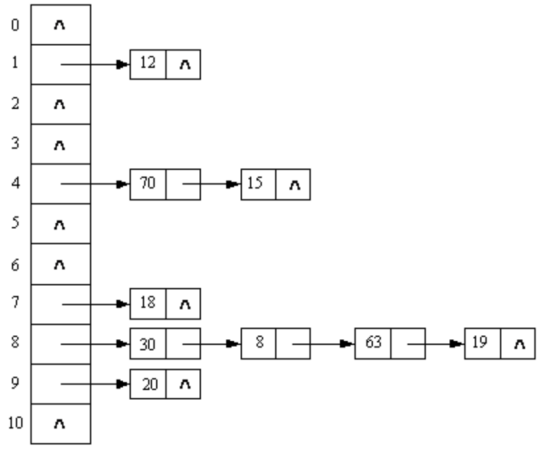
平方探查法的数学描述公式为：

d0=H(k)

di=(d0+i*i) % m (1≤i≤m-1)

2.链地址法

用链地址法解决冲突的方法是：把所有关键字为同义词的记录存储在一个线性链表中，这个链表称为同义词链表。并将这些链表的表头指针放在数组中（下标从0到m-1）。这类似于图中的邻接表和树中孩子链表的结构。



9.19 选取哈希函数H(k)=(3k) MOD 11。用开放定址法处理冲突，di=i ((7k) MOD 10 +1) (i=1,2,3…)。试在0—10的散列地址空间中对关键字序列（22、41、53、46、30、13、01、67）构造哈希表，并求等概率情况下查找成功时的平均查找长度。

9.19 H(22)=0

H(41)=2

H(53)=5

H(30)=2 H1(30)=3

H(13)=6 H1(13)=8

H(01)=3 H1(01)=0 H2(01)=8

H3(01)=5 H4(01)=2 H5(01)=10

H(67)=3 H1(67)=2 H2(67)=1

Hi(key)=(H(key)+di) mod m (i=1,2,... k(k≤m-1))

∴

	0	1	2	3	4	5	6	7	8	9	10
	22	67	41	30		53	46		13		01
Ci:	1	3	1	2		1	1		2		8

ASL=∑(查找次数×该次数)/查找总数

ASL成功=(1×4+2×2+3+6)/8=17/8

关键建字个数
散列表长度

9.20 试为下列关键字建立一个装载因子不小于0.75的哈希表，并计算你所构造的哈希表的平均查找长度。（ZHAO、QIAN、SUN、LI、ZHOU、WU、ZHANG、WANG、CHANG、CHAO、YANG、JIN）

9.20 装载因子 $= \frac{\text{记录数}}{\text{表长}}$ 表长 $\le 12/0.75 = 16$

将字符key转化为数字num表示：

```
int num = 0
for(i=0; i<key.length; i++)
  { num = 37 * num + (key[i] - 'A' + 1)
```

∴ num(ZHAO) = 1327982　num(ZHAO) = 1328506

num(QIAN) = 873473　num(WU) = 872

num(SUN) = 26802　num(ZHENG) = 49140280

num(LI) = 4453　num(WANG) = 1166913

num(CHANG) = 602960　num(CHAO) = 162963

num(YANG) = 1268219　num(JIN) = 14037

设哈希函数为

h(ZHAO) = 14, h(QIAN) = 1, h(SUN) = 2, h(LI) = 5

h(ZHOU) = 10, h(WU) = 8, h(ZHENG) = 12, h(WANG) = 1

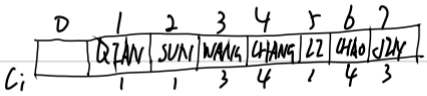
h(WANG) = 2, h(WANG) = 3, h(CHANG) = 1, h(CHANG) = 2

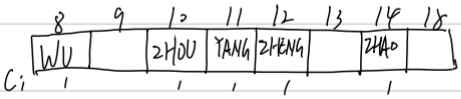
h(CHANG) = 3, h(CHANG) = 4, h(CHAO) = 3,

h(CHAO) = 4, h(CHAO) = 5, h(CHAO) = 6,

h(YANG) = 11, h(JIN) = 5, h(JIN) = 6,

h(JIN) = 7





ASL = 1.8

9.21 在地址空间为0—16的散列区中，对以下关键字序列构造两个哈希表：
(Jan,Feb,Mar,Apr,May,June,July,Aug,Sep,Oct,Nov,Dec)
(1) 用线性探测开放定址法处理冲突
(2) 用链地址法处理
并分别求这两个哈希表在等概率情况下查找成功和不成功时的平均查找长度。设哈希函数为H(x)= i/2 ，其中i为关键字中第一个字母在字母表中的序号。

9.21 (1)

H(Jan)=10/2=5;
H(Feb)=6/2=3;
H(Mar)=13/2=6;
H(Apr)=1/2=0;
H(May)=13/2=6;冲突;H1=6+1=7;
H(June)=10/2=5;冲突;H1=5+1=6;冲突;H2=7;H3=8;
H(July)=5;H1=6;H2=7;H3=8;H4=9
H(Aug)=0;H1=1;
H(Sep)=9;H1=10;
H(Oct)=7;H1=8;H2=9;H3=10;H4=11;
H(Nov)=7;H1=8;H2=9;H3=10;H4=11;H5=12
H(Dec)=2
ASL=(1+2+1+1+1+1+2+4+5+2+5+6)/12=31/12

(2)

H(Jan)=5;
H(Feb)=3;
H(Mar)=6;
H(Apr)=0;
H(May)=6
H(June)=5;
H(July)=5;
H(Aug)=0;;
H(Sep)=9;
H(Oct)=7;
H(Nov)=7;
H(Dec)=2
0->Apr->Aug
1->
2->Dec
3->Feb
4->
5->Jan->June->July
6->Mar->May
7->Oct->Nov
8->
9->Sep
ASL=(1+2+1+1+1+2+3+1+2+1+2+1)/12=18/12

二分查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列。

首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

```
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) { // 注意
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
```

```
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
        }
    return -1;
}
```

10.分类

1、初始序列已经按键值有序时，用直接插入排序算法进行排序，需要比较的次数为（ ）

A n-1

B log_2n

C $2log_2n$

D n^2

2、一组记录的关键字为（**46**、**79**、**56**、**38**、**40**、**84**），则利用快速排序的方法以第一个记录为基准得到的一次划分结果为（ ）

A 38、40、46、56、79、84

B 40、38、46、79、56、84

C 40、38、46、56、79、84

D 40、38、46、84、56、79

3、快速排序在最坏情况下的时间复杂度是（ ）

A $O(log_2n)$

B $O(nlog_2n)$

C O(n^2)

D O(n^3)

快速排序是C.R.A.Hoare于1962年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

该方法的基本思想是：

1．先从数列中取出一个数作为基准数。

2．分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。

3．再对左右区间重复第二步，直到各区间只有一个数。

4、一组记录的关键字为（**46**、**79**、**56**、**38**、**40**、**84**），则利用堆排序的方法进行递增排序建立的初始堆为（ ）

A 79、46、56、38、40、84

B 84、79、56、38、40、46

C 84、79、56、46、40、38

D 84、56、79、40、46、38

5、以下四种排序方法，要求附加内存空间最大的是（ ）

A 插入排序

B 选择排序

C 快速排序

D 归并排序

6、在排序过程中，键值比较的次数与初始序列的排序顺序无关的是（ ）

A 直接插入排序和快速排序

B 直接插入排序和归并排序

C 简单选择排序与归并排序

D 快速排序与归并排序

7、（ ）排序方法每做一趟排序，都会有一个元素被放到最终位置上。

A 归并排序

B 插入排序

C 快速排序

D 基数排序

8、（ ）排序方法是从无序序列中依次取出元素与有序序列中的元素进行比较，将其放入到有序序列的正确位置上。

A 归并排序

B 插入排序

C 快速排序

D 选择排序

9、设表中元素的初始状态是按健值递增的，分别用堆排序、快速排序、冒泡排序、归并排序进行递增排序，则____ [填空1] _____排序算法最节省时间、_____ [填空2] _____排序算法最费时间。

填空1：冒泡排序，填空2：快速排序

20、外部排序是把外存文件调入内存，利用内部排序的方法进行排序，因此排序所花的时间取决于内部排序的时间

A 对

B 错

冒泡排序

0.如果遇到相等的值不进行交换，那这种排序方式是稳定的排序方式。

1.原理：比较两个相邻的元素，将值大的元素交换到右边

2.思路：依次比较相邻的两个数，将比较小的数放在前面，比较大的数放在后面。

(1)第一次比较：首先比较第一和第二个数，将小数放在前面，将大数放在后面。

(2)比较第2和第3个数，将小数 放在前面，大数放在后面。

……

(3)如此继续，知道比较到最后的两个数，将小数放在前面，大数放在后面，重复步骤，直至全部排序完成

(4)在上面一趟比较完成后，最后一个数一定是数组中最大的一个数，所以在比较第二趟的时候，最后一个数是不参加比较的。

(5)在第二趟比较完成后，倒数第二个数也一定是数组中倒数第二大数，所以在第三趟的比较中，最后两个数是不参与比较的。

(6)依次类推，每一趟比较次数减少依次

```
void BubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
            for (int j = 0; j < n - i - 1; j++)
            {
                    if (arr[j] > arr[j + 1])
            {
                            int temp = arr[j];
                            arr[j] = arr[j + 1];
                            arr[j + 1] = temp;
                        }
                }
        }
}
```

快速排序

1. 从数列中挑出一个元素，称为 "基准"（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

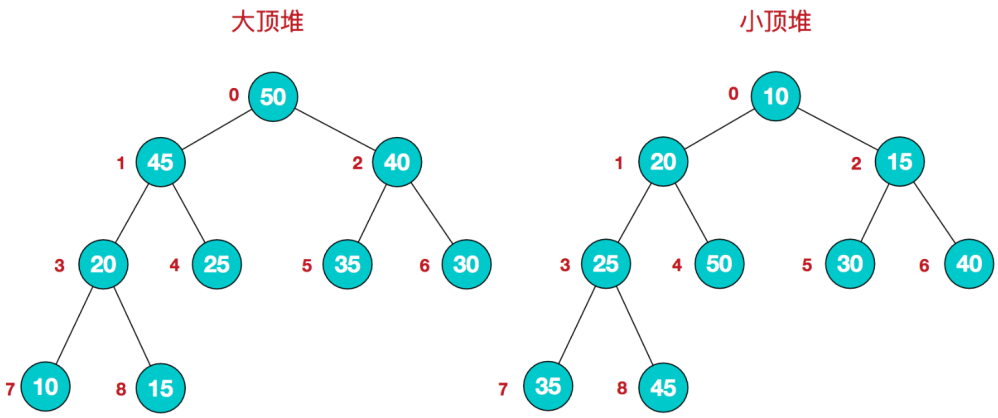
```
Paritition1(int A[], int low, int high) {
   int pivot = A[low];
   while (low < high) {
     while (low < high && A[high] >= pivot) {
       --high;
     }
     A[low] = A[high];
     while (low < high && A[low] <= pivot) {
       ++low;
     }
     A[high] = A[low];
   }
   A[low] = pivot;
   return low;
 }
```

```
void QuickSort(int A[], int low, int high) //快排母函数
{
  if (low < high) {
    int pivot = Paritition1(A, low, high);
    QuickSort(A, low, pivot - 1);
    QuickSort(A, pivot + 1, high);
  }
}
```

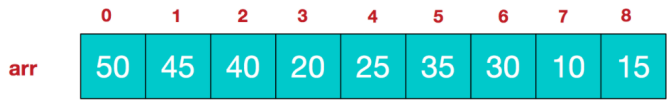
堆排序

堆排序是利用**堆**这种数据结构而设计的一种排序算法，堆排序是一种**选择排序，**它的最坏，最好，平均时间复杂度均为O(nlogn)，它也是不稳定排序。首先简单了解下堆结构。

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。如下图：



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



该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

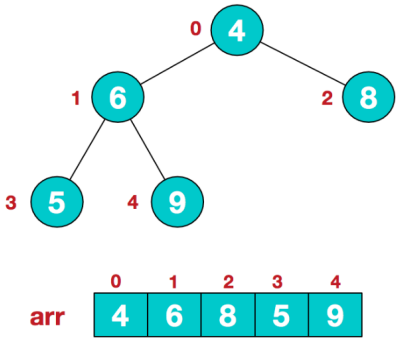
大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

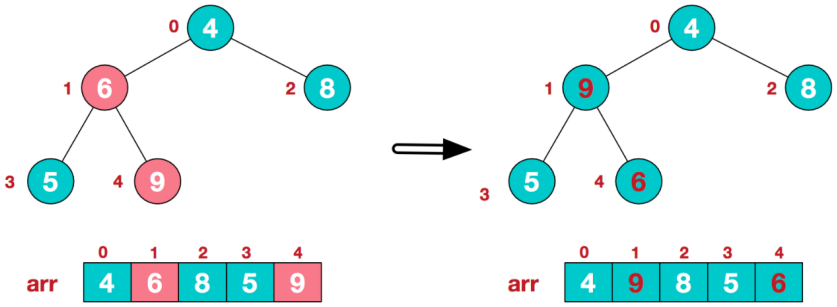
堆排序基本思想及步骤：

步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆)。

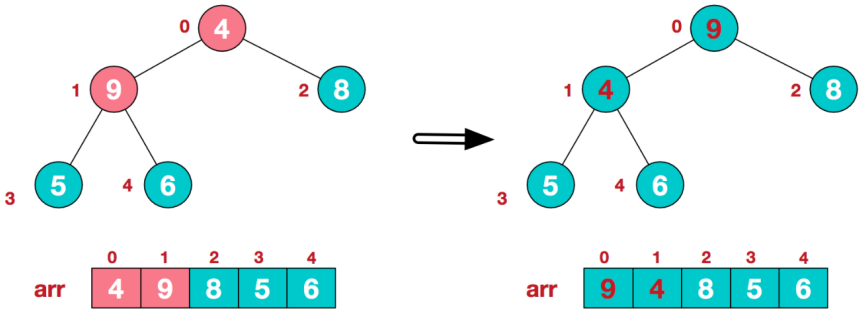
a.假设给定无序序列结构如下



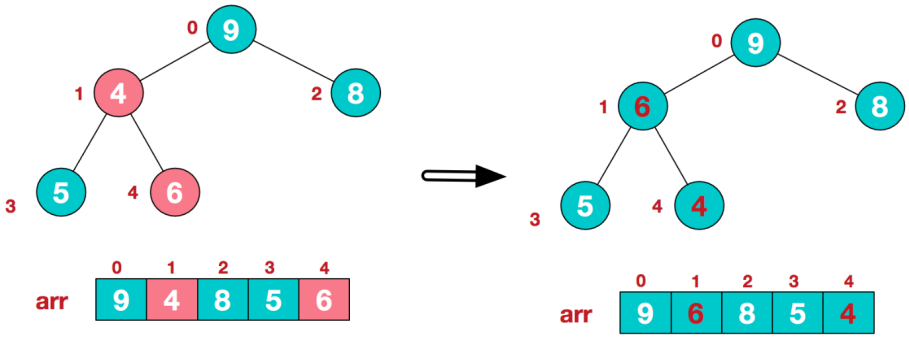
2.此时我们从最后一个非叶子结点开始（叶结点自然不用调整，第一个非叶子结点 arr.length/2-1=5/2-1=1，也就是下面的6结点），从左至右，从下至上进行调整。



4.找到第二个非叶节点4，由于[4,9,8]中9元素最大，4和9交换。



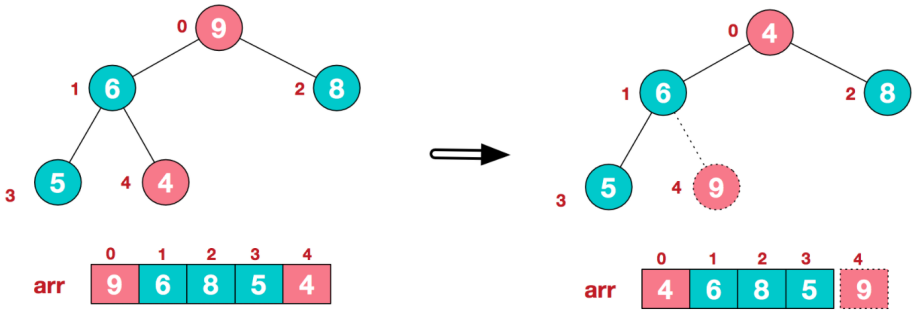
这时，交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中6最大，交换4和6。



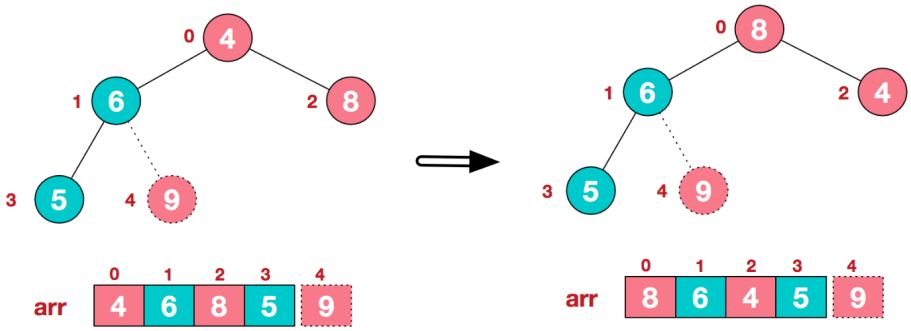
此时，我们就将一个无需序列构造成了一个大顶堆。

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

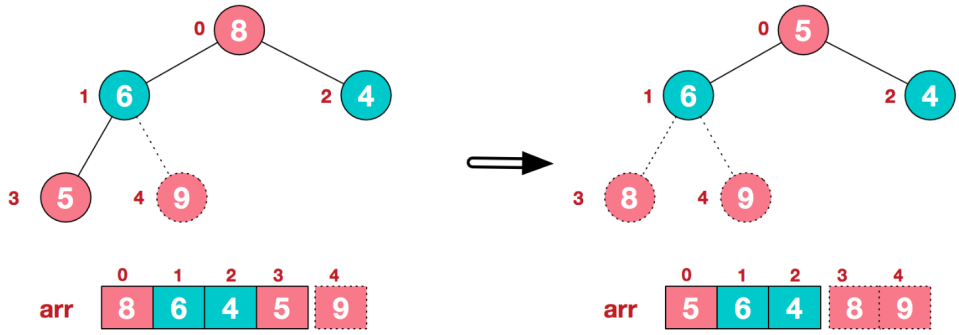
a.将堆顶元素9和末尾元素4进行交换



b.重新调整结构，使其继续满足堆定义



c.再将堆顶元素8与末尾元素5进行交换，得到第二大元素8



后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序