数据结构期末

第一章

数据结构的基本概念

数据元素

数据的基本单位，例如一条记录

数据项

具有独立含义的**数据最小单位**，也称为字段、域、属性等

数据对象

性质相同的数据元素集合，如记录表

数据结构

带结构的数据元素的集合，包括有逻辑结构（数据元素之间的逻辑关系）、存储结构、数据的运算。

算法

一个算法应该是问题求解步骤的描述。程序不一定满足有穷性，入死循环等，但是算法的必要条件是有穷。程序是算法在计算机上的特定实现。

逻辑结构

表示方法

- 图表
- 二元组

逻辑结构类型

1. 集合
2. 线性结构
3. 树形结构
4. 图形结构

存储结构

1. 顺序存储结构
2. 链式存储结构
    - 链式存储设计时，结点内的存储单元地址一定连续
3. 索引存储结构
4. 哈希存储结构

算法性能分析

时间复杂度计算（Tn）

通常用O（Order，数量级）来表示。

1.求整数n(n>=0)的阶乘的算法如下，其时间复杂度是（）。

```
int fact (int n) {
    if(n<=1) return 1;
    return  n*fact(n-1);}
```

B O(n)

2.下列程序段的时间复杂度是（）

```
Count=0;
for(k=1; k<=n; k*=2)
   for(j=1; j<=n; j++)
       count++;
```

C O(nlogn)

3.下列函数的时间复杂度

```
int func  (int n)  {
    int  i=0,  sum=0;
   while(sum<n)  sum += ++i;
       return i;
```

B $O(n^{1/2})$

4.以下算法的时间复杂度（）

```
void fun(int n){
    int i=0;
    while(i*i*i<=n)
       i++;
```

C $O(n^{1/3})$

5.以下算法中最后一条语句的执行次数（）

```
int m=0, i, j;
for(i=1;i<=n;i++)
   for(j=1;j<=2i;j++)
       m++;
```

A n(n+1)

6.设n是描述问题规模的非负整数，下列的时间复杂度

```
x=0;
While (n>=(x+1)*(x+1))
x=x+1;
```

B $O(n^{1/2})$

第二章

线性表 略

第三章

栈和队列 略

第四章

串的定义

基本定义

1. 串是由零个或多个字符组成的有限序列。一般记为S=“a1a2...an”
2. 其中，S是串名；引号括起来的字符序列是串值；
3. ai（i≤n）是串中的字符，字符a在串中的位置为i； n是串中的字符个数，称为串的长度。
4. 长度为零的串称为空串，它不包含任何字符，记为S=“”

子串

串中任意个连续的字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。

例如：“abcde"的子串有：“，"a”，“ab”等。真子串是指不包含自身的所有子串。子串在主串中第一次出现时，子串的第一个字符在主串中的位置即为该子串在主串中的位置。特别地，空串是任意串的子串，任意串是其自身的子串

例如：字符串a=“BEI"b=“JING"c=" BEIJING"

串长：3,4,7

a在c中位置是：1，b在c中位置是：4

第五章

数组的顺序表示和实现

- 例如：定义数组int a[5]，每个元素占用4字节，假设a[0]存储在2000单元，则a[3]地址是多少？

  Loc(i)=Loc(0)+i*K ---> a[3]* = 2000+3*4 = 2012

- 二维数组的存储

  按行序为主序存放，Loc(i, j)=Loc(0,0)+(i*n+j)*K

特殊矩阵的压缩存储

对称矩阵

- 定义：在nxn的矩阵a中，满足性质：aij=aji (1<=i, j<=n)
- 存储方法：只存储下或上三角及主对角线的数据元素。共占用n(n+1)/2个元素空间

K	0	1	2	3	4	...	n(n+1)/2-1
Array[][]	a11	a21	a22	a31	a32	...	ann

三角矩阵

- 定义：在对角线以下（或以上）的数据元素，不包括对角线，全部为常数C
- 存储方法：重复元素C共享一个元素存储空间，上下三角矩阵和对称矩阵一样，所以共占用n(n+1)/2+1个元素空间，存放在一个一维数组sa[n(n+1)/2+1]

对角矩阵

- 定义：在nxn的矩阵中，所有非零元素都集中在以主对角线为中心的带状区域中，区域外的值全为0，则称为对角矩阵。常见的有三对角矩阵，五对角矩阵、七对角矩阵。
- 存储方法：用二维数组存，以对角线的顺序存储

*广义表的定义

基本定义

1. 广义表（Lists）：是n>=0个元素的有限序列，其中每个ai可以是原子，也可以是一个广义表。
2. 广义表记作：LS=(a1,a2,...an)。其中LS为表名，n是表长，ai是表的元素，一般用大写字母表示广义表，用小写字母表示原子。广义表**有括号包围**
3. 表头：若LS非空，则其**第一个元素a1就是表头**head(LS)=a1。表头可以是原子，也可以是子表。
4. 表尾：除**表头之外的其它元素组成的表**tail(LS)=a2,..,an表尾不是最后一个元素，而是一个子表。

计算示例

广义表	计算
A()	空表，长度为0。
B(())	长度为1，表头、表尾均为（）。
C(a,(b,c))	长度为2，有原子a和子表（b,c）构成。表头为a、表尾为（(b,c)）
D(x,y,z)	长度为3，每一项都是原子。表头为a、表尾为（y,z）。
E(C, D)	长度为2，每一项都是子表。表头为C、表尾为（D）。
F(a, F)	长度为2，第一项为原子，第二项是它本身。表头为a、表尾为（F）。

D=(E,F)=((a,(b,c)),F)

GetHead(D)=E;　　GetTail(D)=(F)

GetHead(E)=a;　　GetTail(E)=((b,c))

GetHead(((b,c)))=(b,c)　GetTail (((b,c)))=( )

注意：可以先将广义表用一个变量表示，带入计算

第六章

树

基本概念

- 结点：包含了数据元素及若干个指向其子树的分支。
- 根结点：非空树中无前驱结点的结点。
- 结点的度：结点的子树数目或分支个数。
- 树的度：树中各结点的度的最大值。
- 分支结点（非终端结点）：度大于0的结点。
- 树叶结点（叶子结点、终端结点）：度为0的结点。
- 结点的路径：根结点到该结点所经分支和结点构成结点的路径。
- 结点的路径长度：根结点到该结点路径上所经分支的数目。
- 结点的层次：设根结点的层次为1，则其子树的根结点层次为2；第*L* 层结点的子树的根结点层次为*L*+1。
- 树的深度：树中结点（该结点必为树叶结点）的最大层次。
- 孩子结点及双亲结点：结点的子树的根结点称为该结点的孩子结点，该结点又称为孩子结点的双亲结点。
- 兄弟结点：拥有同一个双亲结点的若干个结点互称为兄弟结点。
- 堂兄弟结点：在同一层次上，但双亲结点不同的若干个结点称为堂兄弟结点。
- 祖先结点：根结点到该结点路径上的所有结点均为该结点的祖先结点。
- 子孙结点：某结点的子树中所包含的所有结点均为该结点的子孙结点。
- 无序树：子树之间不存在次序关系，即子树能够调换，则称该树为无序树。
- 有序树：子树之间映射客观存在的次序关系（子树不能调换），则称该树为有序树。
- 森林：是m（m≥0）棵互不相交的树的集合。

***树的性质**

- 树总度数=总结点数-1

  例题：在一棵度为4的树中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树的叶子结点个数是：82

  计算：设叶子结点x个，有20*4+10*3+1*2+10*1=20+10+1+10+x-1，解得X=82

二叉树

基本概念

- 定义：二叉树是*n*（*n*≥0）个结点的有限集，当*n*=0时，二叉树为空；当*n*>0时，二叉树由一个根结点及至多两棵互不相交的左右子树组成，且左右子树都是二叉树。
- 特点：每个结点至多有二棵子树（即不存在度大于2的结点）；二叉树的子树有左、右之分，且其次序不能任意颠倒；二叉树可以是空集合，根可以有空的左子树或空的右子树。结点的子树要区分左子树和右子树，即使只有一棵子树也要说明它是左子树还是右子树。

***二叉树的性质**

- 在二叉树的第 i 层上至多有2^(i-1) 个结点。
- 深度为 *k* 的二叉树上至多含 2^k-1 个结点，至少有k个结点
- 对任何一棵二叉树，若它含有n0 个叶子结点、n2 个度为 2 的结点，则必存在关系式：*n*0 = n2+1。（利用结点数=度数+1，设n1即可）

特殊的二叉树

1. 满二叉树：一棵深度为k的二叉树若每一层上的结点数都达到最大（即$2k$-1个结点），则称其为满二叉树
2. 完全二叉树：一棵具有n个结点且深度为k的二叉树若前 k - 1层的结点数都达到最大，剩余的结点在第k层中从左至右连续分布，则称其为完全二叉树。 满二叉树是特殊的完全二叉树。

***特殊的二叉树性质**

- 具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor$ +1，其中$\lfloor x \rfloor$称作x的底，表示不大于x的最大整数。
- 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点，其双亲结点编号为 $\lfloor i / 2 \rfloor$ 。表现了完全二叉树中双亲结点编号和孩子结点编号之间的关系。

例题

1. 若一棵深度为6的完全二叉树的第6层有3个叶子结点，则该二叉树共有（17）个叶子结点。
2. 已知一棵完全二叉树的第6层（设根为第一层）有8个叶结点，则该完全二叉树的结点个数最多是（111）。第六层满层
3. **若一棵完全二叉树有768个结点，则该二叉树中叶结点的个数是（384）。最后一个分支结点的序号是768/2向下取整=384，所以叶子结点数=786-384=384.**
4. 设一棵非空完全二叉树T的所有叶结点均位于同一层，且每个非叶结点都有2个子结点。若T有k个叶结点，则T的结点总数是（2k-1）。高为h的满二叉树

存储结构

顺序存储结构

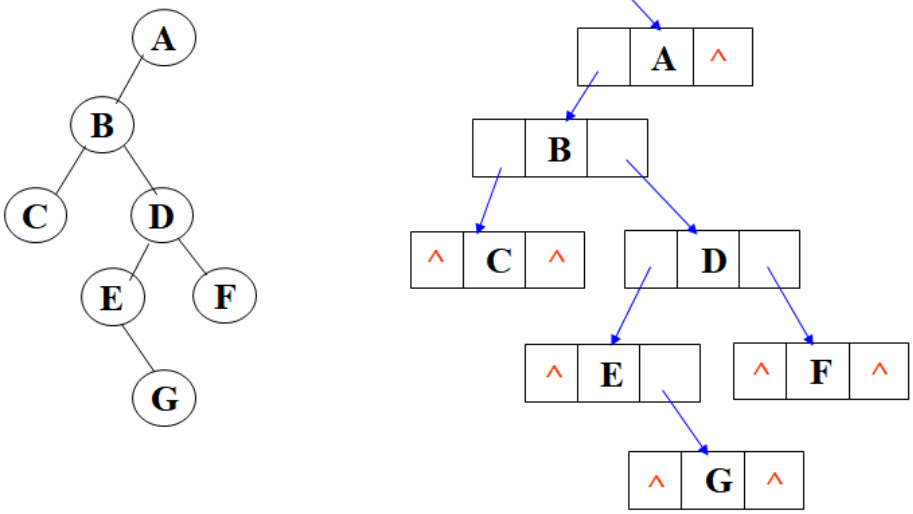
- 完全二叉树：按完全二叉树的结点层次编号，依次存放二叉树中的数据元素，即按照从上到下、从左到右的顺序依次存储所有结点。
- 非完全二叉树：若该二叉树为非完全二叉树，则必须将相应位置空出来或用0补充，使存放的结果符合完全二叉树形状。为方便存储常需要把二叉树中补充成完全二叉树形状。

0	1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	0	0	0	0	f	g

链式存储结构

- 图例

结点结构



遍历二叉树

含义

沿着某一条搜索路径巡访二叉树中的所有结点，使得每个结点均被且仅被访问一次。二叉树的遍历实际上就是要把一个非线性结构的二叉树转化为一个线性结构。

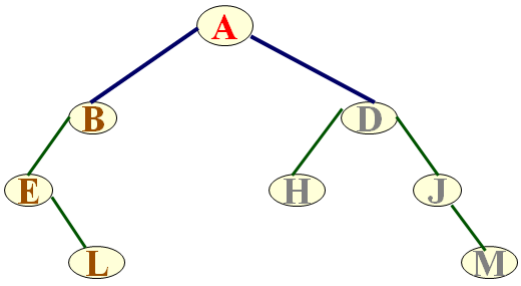
遍历类型

若规定了先左后右，则只有三种遍历类型：

- DLR——先（根）序遍历
- LDR——中（根）序遍历
- LRD——后（根）序遍历

若从层次关系遍历，则有层次遍历法。从根结点开始，按从上到下，从左到右的顺序访问每一个结点。每个结点仅访问一次。

图例



先序结果：ABELDHJM

中序结果：ELBAHDJM

后续结果：LEBHMJDA

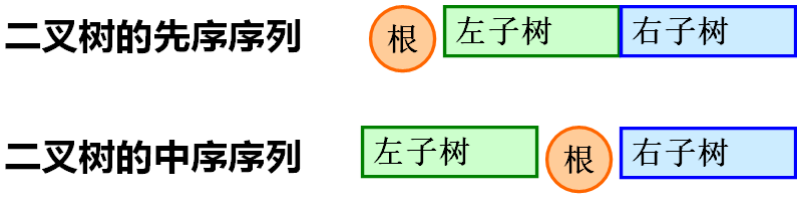
层次遍历：ABDEHJLM

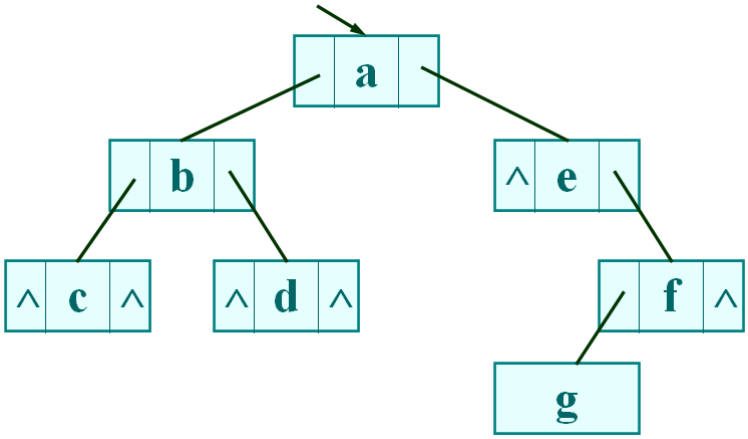
***遍历序列和二叉树的关系**

- 若二叉树中各结点的值均不相同，则二叉树结点的先序序列、中序序列和后序序列都是唯一的。
- 由二叉树的先序序列和中序序列，或由二叉树的后序序列和中序序列可以确定唯一一棵二叉树。

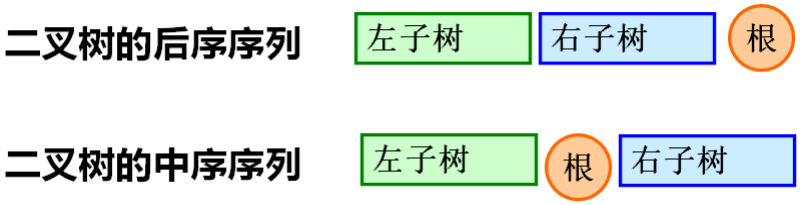
例题：

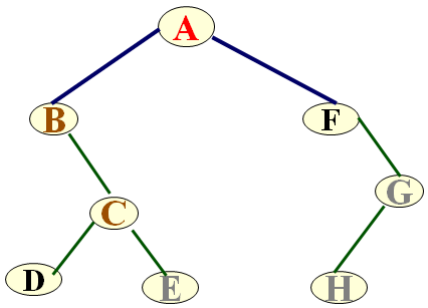
1. 已知先序序列：abcdefg；中序序列：cbdaegf，求由该序列确定的二叉树（解法：根据先序序列a在中序中位置确定分割cbd,egf分别为a的左右子树，再根据b在中序中的位置确定c,d为b的左右子树，依次类推即可）





2. 已知后序序列：decbhgfa；中序序列：bdceafhg，求由该序列确定的二叉树（解法：根据后序序列a在中序中位置确定分割bdce,fhg分别为a的左右子树，后续同理）





[遍历二叉树算法(ctrl+click)]

*线索二叉树

含义

利用二叉链表中的空指针域，对二叉树按某种遍历次序使其变成线索二叉树的过程称为线索化。

如果某个结点的左孩子为空，则将空的左孩子指针域改为指向其前趋；如果某结点的右孩子为空，则将空的右孩子指针域改为指向其后继。

为了区分lchild和rchild指针到底指向孩子还是前趋或后继，对二叉链表中每个结点增设两个标志域ltag和rtag，并做如下约定：

ltag=0，lchild指向结点的左孩子；

ltag=1，lchild指向结点的前趋；

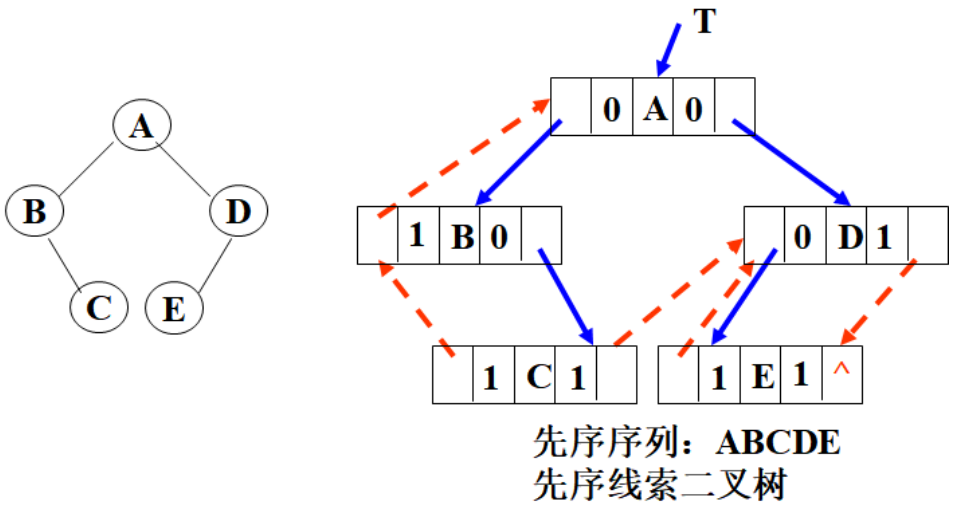
rtag=0，rchild指向结点的右孩子；

rtag=1，lchild指向结点的后继；

lchild	LTag	data	RTag	rchild

做法

以先序序列为例，先确定序列，再将空域构建线索



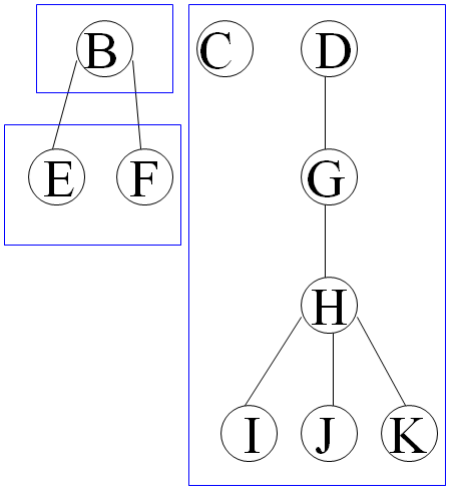
森林的遍历

森林的构成

1．森林中第一棵树的根结点；

2．森林中**第一棵树的子树森林**；

3．森林中其它树构成的森林。



先序遍历

若森林不空，则访问森林中第一棵树的根结点；先序遍历森林中第一棵树的子树森林；先序遍历森林中(除第一棵树之外)其余树构成的森林。即：依次从左至右对森林中的**每一棵树进行先根遍历。**

上图先序序列：BEFCDGHIJK

中序遍历

若森林不空，则中序遍历森林中第一棵树的子树森林；访问森林中第一棵树的根结点；中序遍历森林中(除第一棵树之外)其余树构成的森林。即：依次从左至右对森林中的**每一棵树进行后根遍历。**

上图中序序列：EFBCIJKHGD

Tips

树没有中根遍历：因为一棵非空树的根结点可能有2个以上子树，无法确定根结点的遍历次序

森林没有后序：若子树森林—森林—根，则把同一棵树割裂

*哈夫曼树

定义

在n个带权叶子结点构成的所有二叉树中，带权路径长度WPL最小的二叉树成为哈夫曼树/最优二叉树

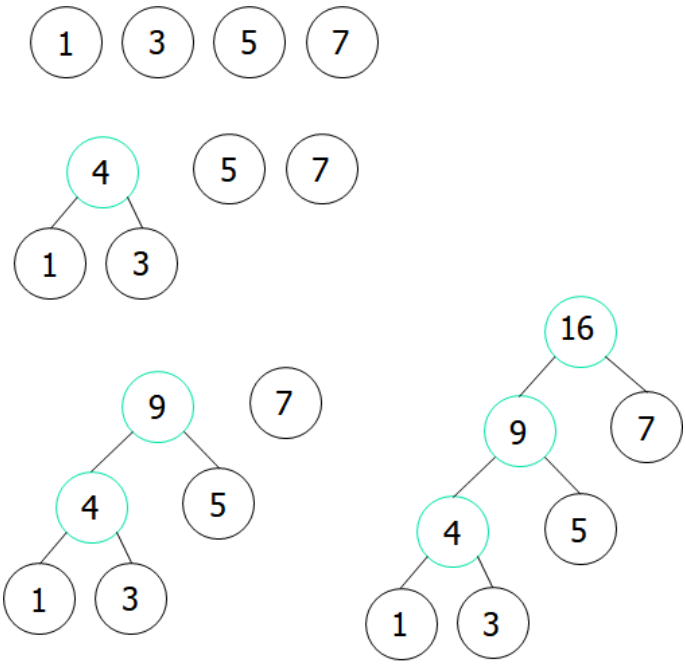
WPL计算

$$WPL = \sum_{k=1}^{n} w_k l_k$$

其中：

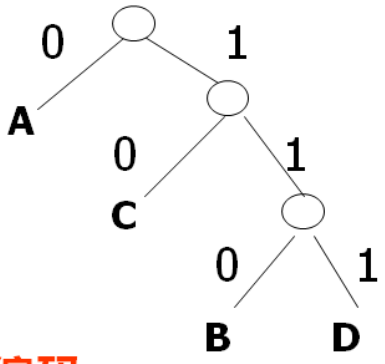
- n – 叶子结点的个数
- w_k —第k个叶子结点的权值
- l_k — 第k个叶子结点到根的路径长度

哈夫曼树的创建



哈夫曼编码

例：A: 3　B:1　C:2　D: 1



则：　A：0
　　C：10
　　B：110
　　D：111

由此可见该编码是前缀编码

待传电文ABACCDA 的编码：
‘0110010101110’，

总长13

哈夫曼编码：

约定： 树中的左分支表示字符 ‘0’
　　树中的右分支表示字符 ‘1’

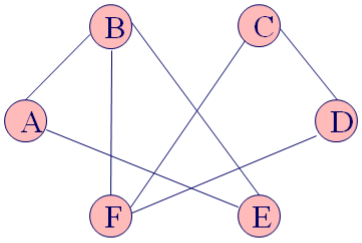
编码：从根到叶子的路径上分支字符组成的字符串作为该叶子结点字符的编码。

第七章

图的存储结构

（邻接矩阵）数组表示法

例1 无向图



顶点数组 vertices

0	A
1	B
2	C
3	D
4	E
5	F

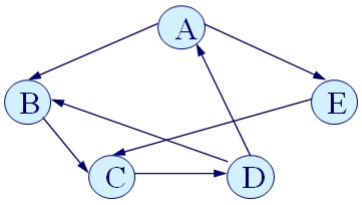
邻接矩阵 adjMatrix

0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	0	1
1	1	0	0	0	0
0	1	1	1	0	0

numVertices=6
numEdges=7
style=UDG(无向图)

- 无向图的邻接矩阵是对称的
- 顶点i的度=第i行（列）中1的个数
- 完全图的邻接矩阵中，对角元素为0，其余为1。

例2 有向图：



顶点数组 vertices

0	A
1	B
2	C
3	D
4	E

邻接矩阵 adjMatrix

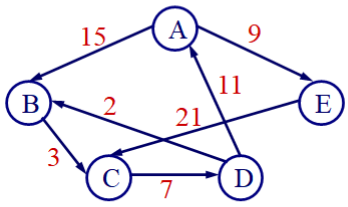
0	1	0	0	1
0	0	1	0	0
0	0	0	1	0
1	1	0	0	0
0	0	1	0	0

numVertices=5
numEdges=7
style=DG(有向图)

有向图的邻接矩阵不一定是对称矩阵

顶点出度=第i行元素之和
顶点入度=第i列元素之和
顶点度=第i行元素之和+第i列元素之和

例3 有向网



顶点数组 vertices

0	A
1	B
2	C
3	D
4	E

邻接矩阵 adjMatrix

∞	15	∞	∞	9
∞	∞	3	∞	∞
∞	∞	∞	7	∞
11	2	∞	∞	∞
∞	∞	21	∞	∞

numVertices=5
numEdges=7
style=DN(有向网)

数组表示法不足之处

不便于增加和删除顶点

浪费空间O(n2)：存稀疏图（点很多而边很少）有大量无效元素，但是稠密图（例如完全图）较为划算。

浪费时间：统计稀疏图中一共有多少条边

（邻接表）链表表示法

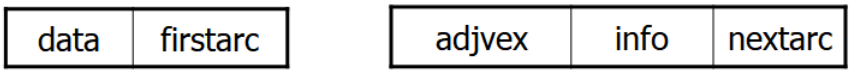
邻接表表示法

顶点表：按编号顺序将顶点数据存储在一维数组中。

关联同一顶点的边（以顶点为尾的弧）：用线性链表存储

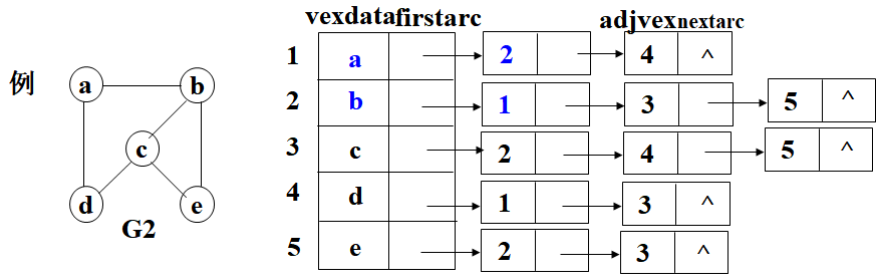
表头结点结构：数据域（data）用于存储顶点的名或其它有关信息；
链域（firstarc）用于指向链表中第一个顶点（即与顶点 v_i 邻接的第一个邻接点）

边结点结构：adjvex 与顶点 v_i 邻接的点在顶点表中的位置
info 存储和边相关的信息(若无，则置空NULL)
nextarc 指向下一条边的结点的指针

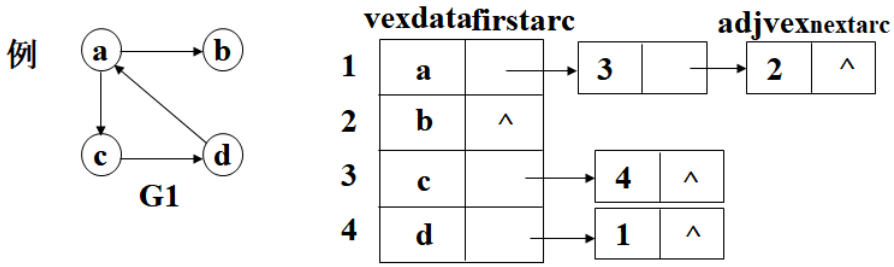


表头结点　　弧结点

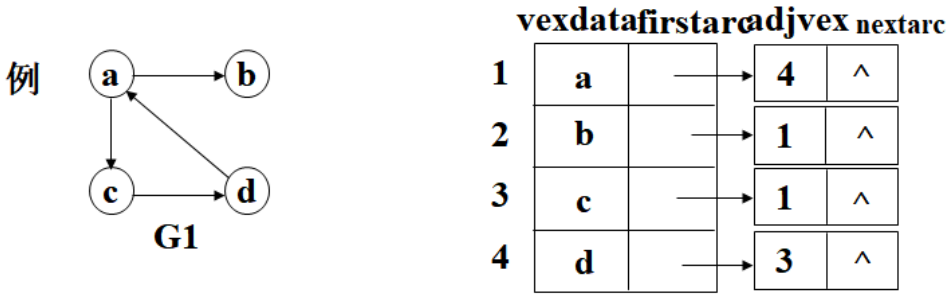
无向图



有向图



逆邻接表



######

最小生成树

图的遍历

深度优先遍历DFS

广度优先遍历BFS

非连通图的深度优先搜索遍历：首先将图中每个顶点的访问标志设为 FALSE, 之后搜索图中每个顶点，如果未被访问，则以该顶点为起始点，进行深度优先搜索遍历，否则继续检查下一顶点。

结论：

稠密图适于在邻接矩阵上进行深度遍历；

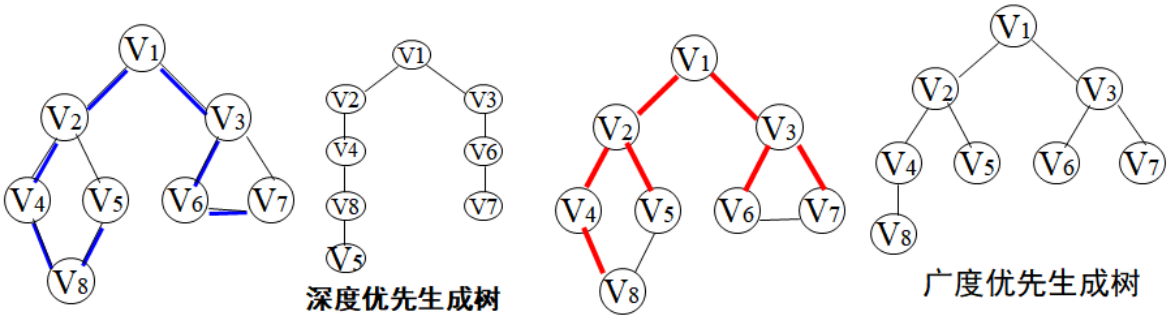
稀疏图适于在邻接表上进行深度遍历。

生成树定义

对连通图进行遍历，得到的是一个极小连通子图，即图的生成树。由深度优先搜索得到的生成树，称为深度优先搜索生成树。由广度优先搜索得到的生成树，称为广度优先搜索生成树。

对非连通图进行遍历，得到的是各连通分量的生成树，即图的生成森林。

生成树图示



生成树特点

生成树的顶点个数与图的顶点个数相同（是连通图的极小连通子图）

一个有n个顶点的连通图的生成树只有n-1条边

在生成树中再加一条边必然形成回路

生成树中任意两个顶点间的路径是唯一的

含n个顶点n-1条边的图不一定是生成树

最小代价生成树（无向图）

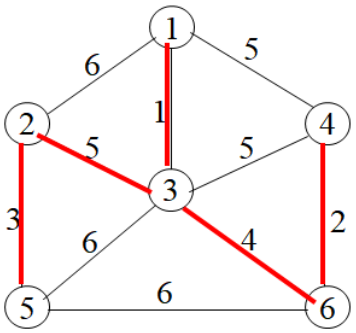
在一个连通网的所有生成树中， 各边的代价之和最小的那棵生成树称为该连通网的最小代价生成树（Minimum Cost Spanning Tree），简称为最小生成树(MST)

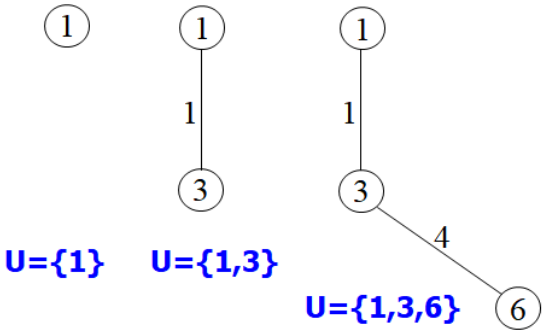
例题：假设要在 *n* 个城市之间建立通讯联络网，则连通 *n* 个城市只需要修建 *n-1*条线路，如何在最节省经费的前提下建立这个通讯网?

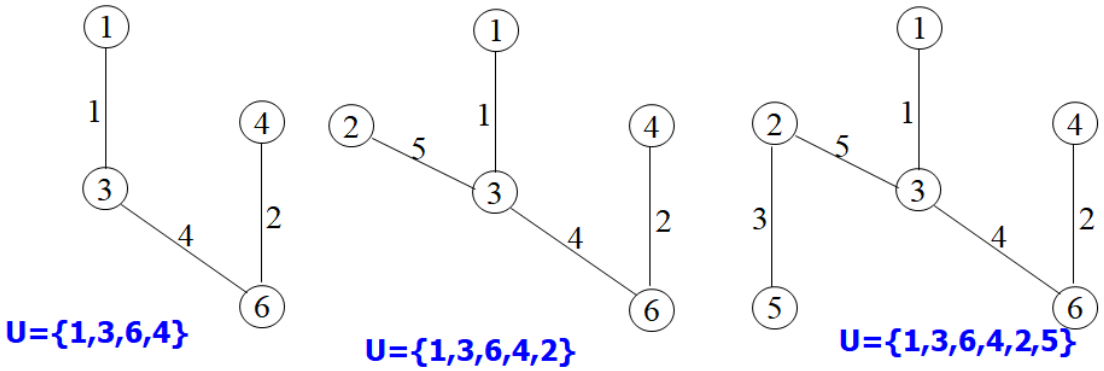
该问题等价于：在n个顶点的连通网中，构造网的一棵最小生成树, 即：在 e 条带权的边中选取 n-1 条边(不构成回路), 使“权值之和”为最小。

1. **Prim算法**

   先确定结点，找最小权值边依次加入







v / closedge	2	3	4	5	6	U	V-U
adjvex	1	1	1	1	1	{1}	{2,3,4, 5,6}
lowcost	6	1	5	∞	∞		
adjvex	3		1	3	3	{1, 3}	{2, 4, 5, 6}
lowcost	5	0	5	6	4		
adjvex	3		6	3		{1,3,6}	{2, 4,5}
lowcost	5	0	2	6	0		
adjvex	3			3		{1,3, 6,4}	{2, 5}
lowcost	5	0	0	6	0		
adjvex				2		{1,3,6, 4,2}	{5}
lowcost	0	0	0	3	0		
adjvex						{1,3,6, 4,2,5}	{}
lowcost	0	0	0	0	0		

2. **Kruskal算法**

初始状态为只有n个顶点而无边的非连通图T=(V,{F})，每个顶点自成一个连通分量。在E中选取代价最小的边，若该边依附的顶点落在T中两个不同的连通分量上，则将此边加入到T中；否则，舍去此边，选取下一条代价最小的边。依此类推,直至T中所有顶点都在同一连通分量上为止

算法比较

算法名	Prim 普里姆算法	Kruskal 克鲁斯卡尔算法
算法思想	选择顶点	选择边
时间复杂度	O(n2)	O(eloge)
适应范围	稠密图	稀疏图

拓扑排序（课程安排）

AOV网

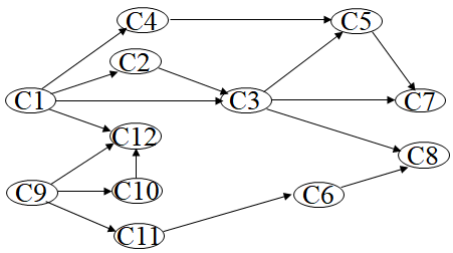
用顶点表示活动的网络。若用有向图表示一个工程，在图中用顶点表示活动，用弧表示活动间的优先关系。若图中存在有向边<vi,vj>，则Vi 必须先于活动Vj 进行。则这样的有向图称为用顶点表示活动的网络，简称AOV网。

实例

课程安排

课程代号	课程名称	先修棵
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构	C1,C2
C4	汇编语言	C1
C5	语言的设计和分析	C3,C4
C6	计算机原理	C11
C7	编译原理	C3.C5
C8	操作系统	C3,C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1,C9,C10

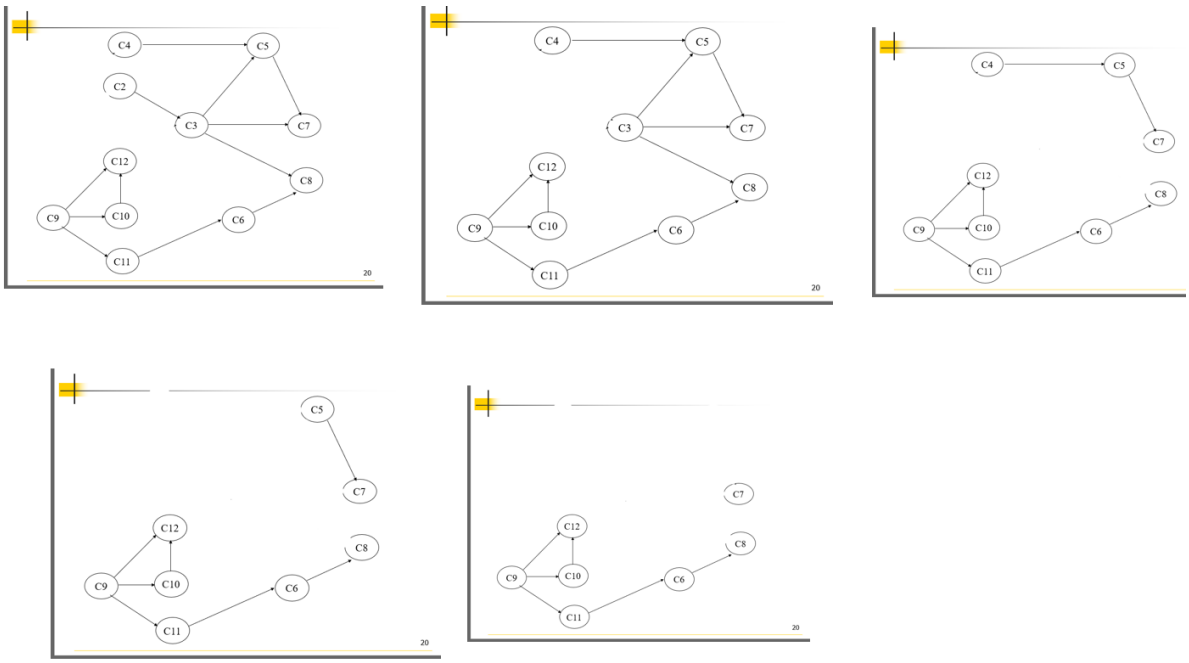
转化为图



方法

1. 在有向图中选一个没有前驱的顶点并输出之
2. 从图中删除该顶点和所有以它为尾的弧
3. 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止

部分示例



*关键路径（时间管理）

AOE网

用边表示活动的网络。如果在无环的带权有向图中， 用有向边表示一个工程中的活动，用边上权值表示活动持续时间，用顶点表示事件，则这样的有向图叫做用边表示活动的网络，简称 AOE网。

基本概念

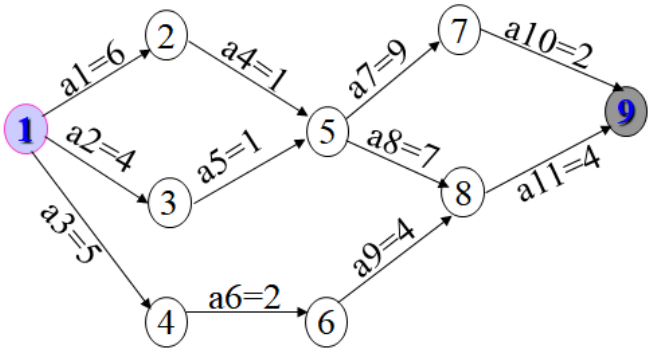
关键路径：完成整个工程所需的时间取决于从源点到汇点的**最长路径**长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)

事件 V1（源点）——表示整个工程开始

事件 V9（汇点）——表示整个工程结束

Ve[i]：事件Vi 的最早可能开始时间（正向最大时间）

Vl [i]：事件Vi 的最迟允许开始时间（逆向最小时间）



计算方法

1. 求顶点的Ve（最早时间）、Vl（最晚时间）。Ve求解从源点到汇点，Vl求解从汇点到源点。

顶点	Ve	Vl
V1	0	0
V2	6	6
V3	4	6
V4	5	8
V5	7	7
V6	7	10
V7	16	16
V8	14	14
V9	18	18

2. 求活动的e（Ve前节点的最早时间）、l（后Vl-ai）。计算出l-e = 0的活动即为关键活动

活动	e	l	l-e
a1	0	0	0 ✓
a2	0	2	2
a3	0	3	3
a4	6	6	0 ✓
a5	4	6	2
a6	5	8	3
a7	7	7	0 ✓
a8	7	7	0 ✓
a9	7	10	3
a10	16	16	0 ✓
a11	14	14	0 ✓

第八章

动态存储管理 略

第九章

二叉排序树

定义

- 若其左子树非空，则**左子树上所有结点的值均小于根结点的值；**
- 若其右子树非空，则**右子树上所有结点的值均大于等于根结点的值；**
- 其左右子树本身又各是一棵二叉排序树。

查找

若查找的关键字等于根结点，成功；

否则，若小于根结点，查找其左子树；若大于根结点，查找其右子树；

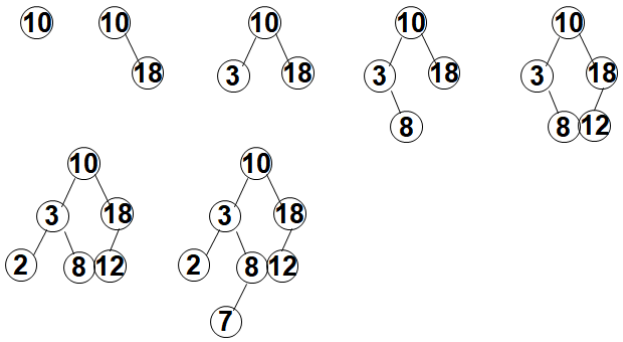
插入

若二叉排序树为空，则插入结点作为根结点插入到空树中。

否则，继续在其左右子树上查找直至某个叶子结点的左子树或右子树为空为止，则插入结点应为该叶子结点的左孩子或右孩子。**插入的元素一定是在叶结点上**

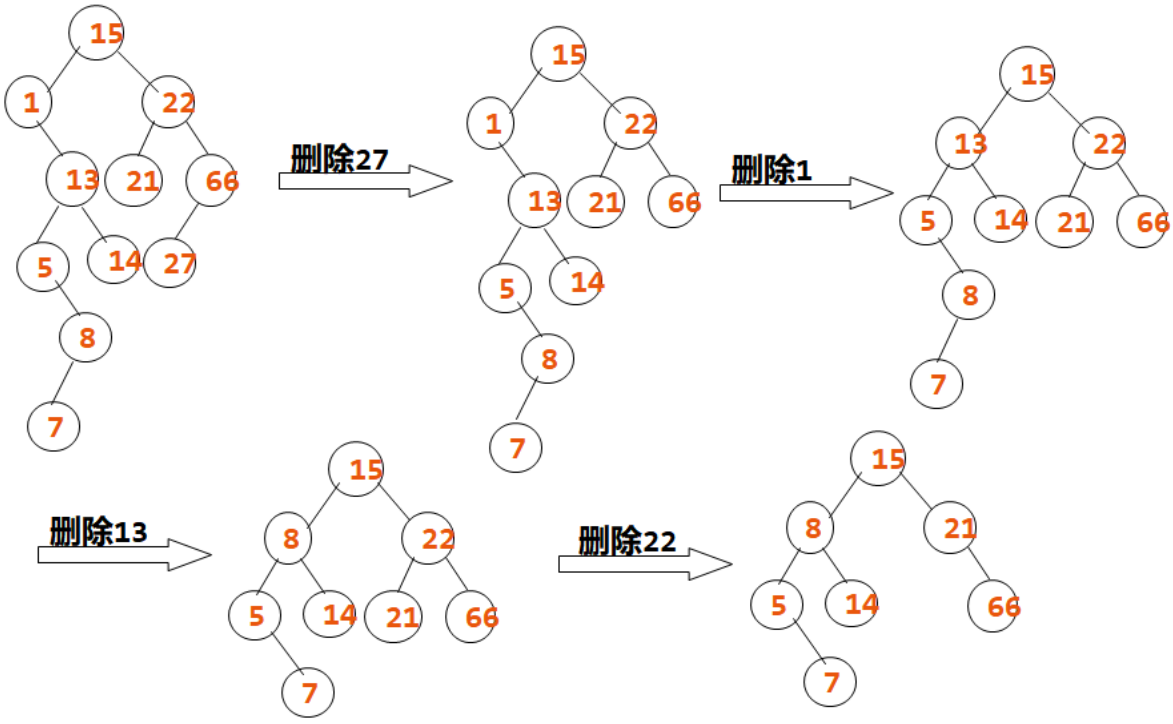
生成

关键字序列为{10,18,3,8,12,2,7,3}



***删除**

关键是要保证二叉排序树的性质不变，具体变化规则分情况而定



平衡二叉树

定义

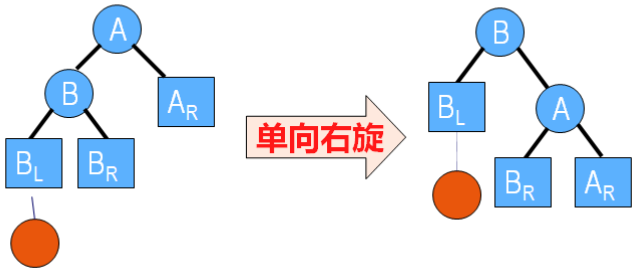
平衡二叉树又称AVL树（Adelson-Velskii and Landis），左子树与右子树的高度之差的绝对值小于等于1；左子树与右子树也是平衡二叉排序树。

平衡因子（BF）：结点的平衡因子 = 结点左子树的**高度** - 结点右子树的**高度**

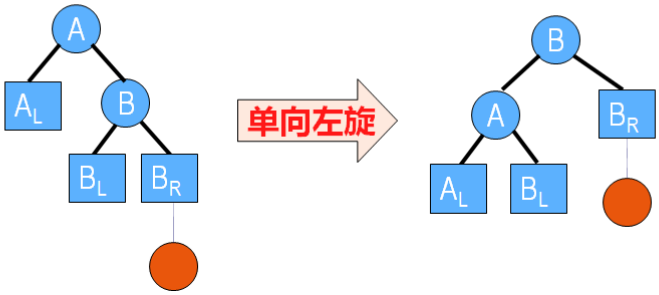
根据平衡二叉树的定义，平衡二叉树上所有结点的平衡因子只能是-1、0、1

调整方法

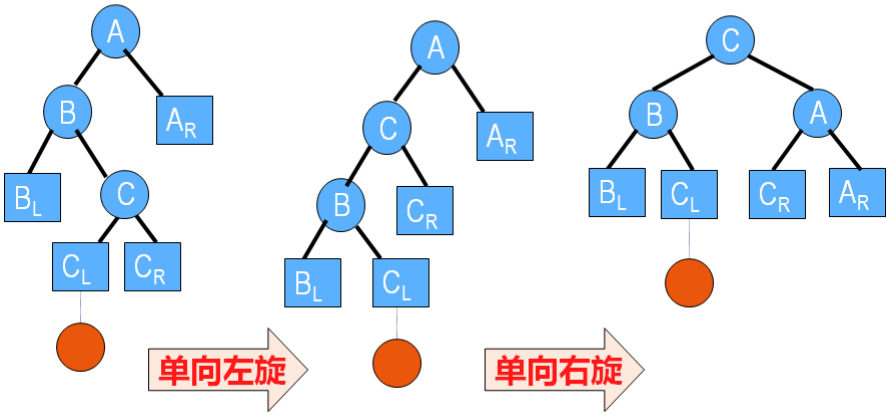
- 旋转法
    - LL型平衡调整（单向右旋）



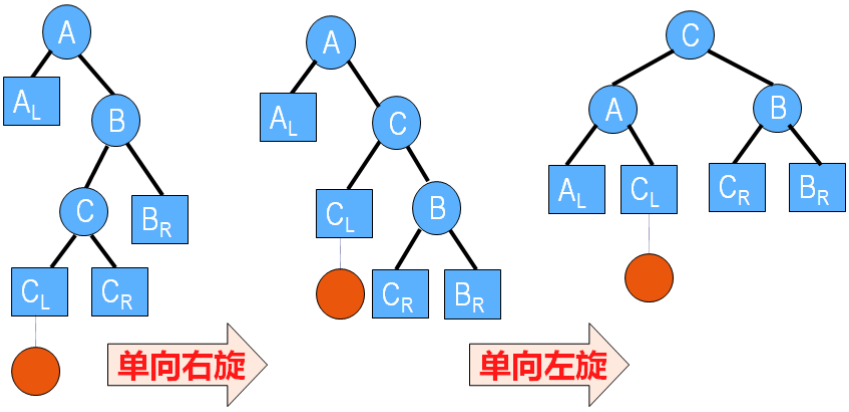
    - RR型平衡调整（单向左旋）



    - LR型平衡调整（先单向左旋再单向右旋）



- RL型平衡调整（先单向右旋再单向左旋）



哈希表

含义

在记录的存储地址和它的关键字之间建立一个确定的对应关系(哈希/散列)；这样，不经过比较，一次存取就能得到所查元素。

关键问题

1. 散列函数的设计。如何设计一个简单、均匀、存储利用率高的散列函数。
2. 冲突的处理。如何采取合适的处理冲突方法来解决冲突。

冲突

对于两个不同关键码ki≠kj，有H(ki) = H(kj)，即两个不同的记录需要存放在同一个存储位置，ki和kj相对于H称做同义词。

常用哈希函数

直接定址法

散列函数是关键码的线性函数，即：H(key) = a*key + b （a，b为常数）

除留余数法

散列函数为：H(key)=key mod p

一般情况下，选*p*为小于或等于表长（最好接近表长）的最小素数或不包含小于20质因子的合数。

处理冲突

开放定址法

由关键码得到的散列地址一旦产生了冲突，就去寻找下一个空的散列地址，并将记录存入

具体实现：

1. 线性探测法：当发生冲突时，从**冲突位置的下一个位置**起，依次寻找空的散列地址。
2. 二次探测法：$H_i=(H(key)+d_i)\%m$

   （$d_i=1^2，-1^2，2^2，-2^2，\ldots，q^2，-q^2$且$q\leqslant m/2$）

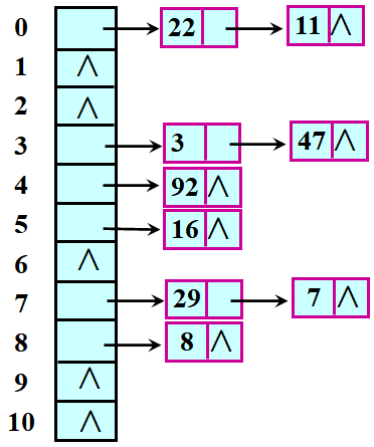
3. 随机探测法：

$$H_i=(H(key)+d_i)\ \%\ m$$

（d_i是一个随机数列，i=1，2，……，m−1）

拉链法

将所有散列地址相同的记录，即所有同义词的记录**存储在一个单链表**中



例题

例　已知一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
哈希函数为：H(key)=key MOD 13, 哈希表长为m=16，
设每个记录的查找概率相等

(1) 用线性探测再散列处理冲突，即Hi=(H(key)+di) MOD m

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			

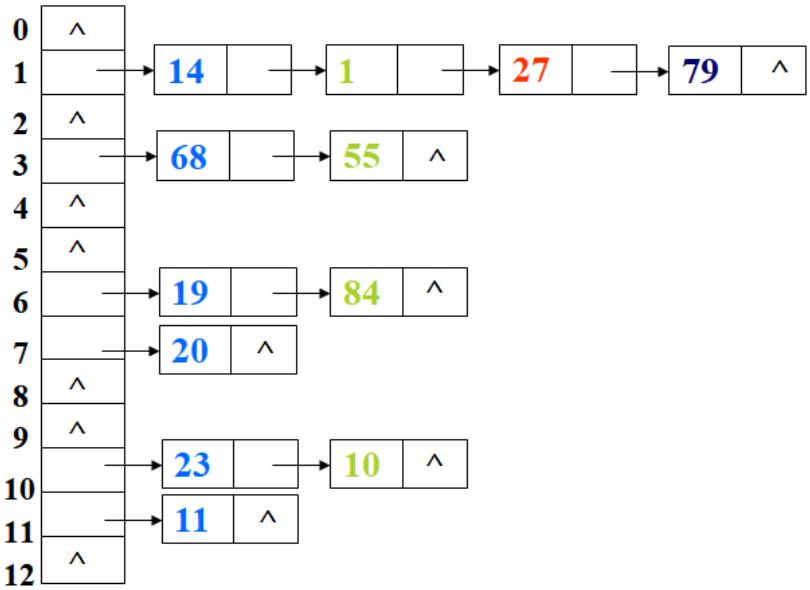
H(19)=6
H(14)=1
H(23)=10
H(1)=1　冲突，H1=(1+1) MOD16=2
H(68)=3
H(20)=7
H(84)=6　冲突，H1=(6+1)MOD16=7
　　冲突，H2=(6+2)MOD16=8
H(27)=1　冲突，H1=(1+1)MOD16=2
　　冲突，H2=(1+2)MOD16=3
　　冲突，H3=(1+3)MOD16=4
H(55)=3　冲突，H1=(3+1)MOD16=4
　　冲突，H2=(3+2)MOD16=5

H(11)=11
H(10)=10　冲突，H1=(10+1)MOD16=11
　　冲突，H2=(10+2)MOD16=12
H(79)=1　冲突，H1=(1+1)MOD16=2
　　冲突，H2=(1+2)MOD16=3
　　冲突，H3=(1+3)MOD16=4
　　冲突，H4=(1+4)MOD16=5
　　冲突，H5=(1+5)MOD16=6
　　冲突，H6=(1+6)MOD16=7
　　冲突，H7=(1+7)MOD16=8
　　冲突，H8=(1+8)MOD16=9

ASL=(1*6+2+3*3+4+9)/12=2.5

关键字(19,14,23,1,68,20,84,27,55,11,10,79)

(2) 用链地址法处理冲突



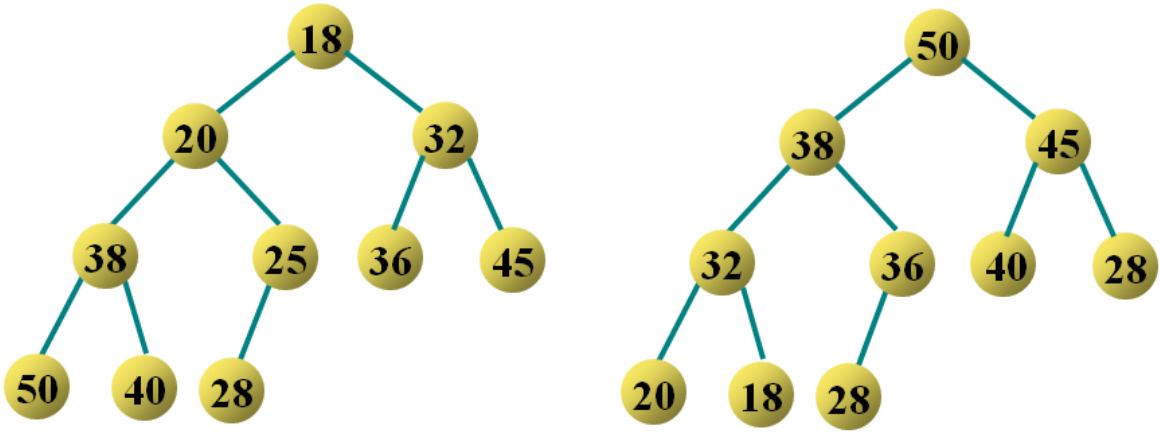
ASL=(1*6+2*4+3+4)/12=1.75

第十章

堆排序

堆的定义

堆是具有下列性质的完全二叉树：每个结点的值都小于或等于其左右孩子结点的值（称为小根堆），或每个结点的值都大于或等于其左右孩子结点的值（称为大根堆）



堆和序列的关系

将堆用顺序存储结构来存储，则堆对应一组序列。

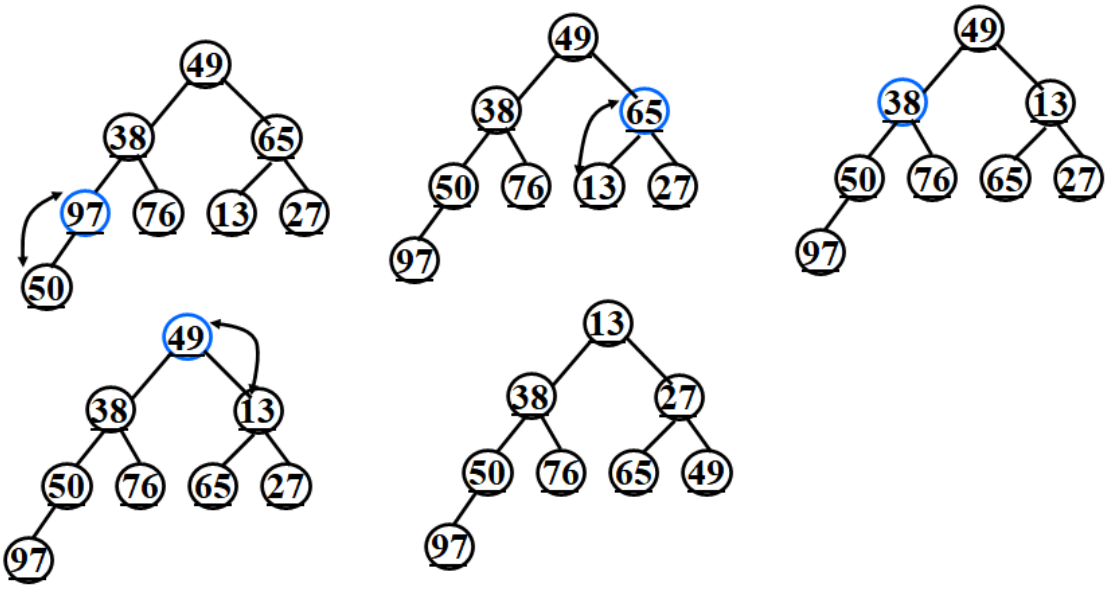
堆排序

含义

首先将待排序的记录序列构造成一个堆，此时，选出了堆中所有记录的**最大者**，然后将它从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的记录，以此类推，直到堆中只有一个记录。

堆有序化

以构建大根堆为例，从叶子结点所在的父子堆将大的节点移动到该父子堆的根节点，依次向大根堆的父节点进行。遍历完后取出根节点，将任意左右子堆父节点作为新的大根堆根节点



基数排序

含义

借助“分配”和“收集”对单逻辑关键字进行排序的一种方法，不需要关键字比较

排序方法比较

排序方法	最好情况	平均时间复杂度	最坏情况	辅助存储	稳定性
直接插入	O（n）	O（n2）	O（n2）	O（1）	稳定
希尔排序	O（n）	O（n1.3）	O（n2）	O（1）	不稳定
冒泡排序	O（n）	O（n2）	O（n2）	O（1）	稳定
快速排序	O(nlog2n)	O(nlog2n)	O（n2）	O(nlog2n)	不稳定
简单选择	O（n2）	O（n2）	O（n2）	O（1）	不稳定
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O（1）	不稳定
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O（n）	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O（n+rd）	稳定

算法

二叉树遍历算法

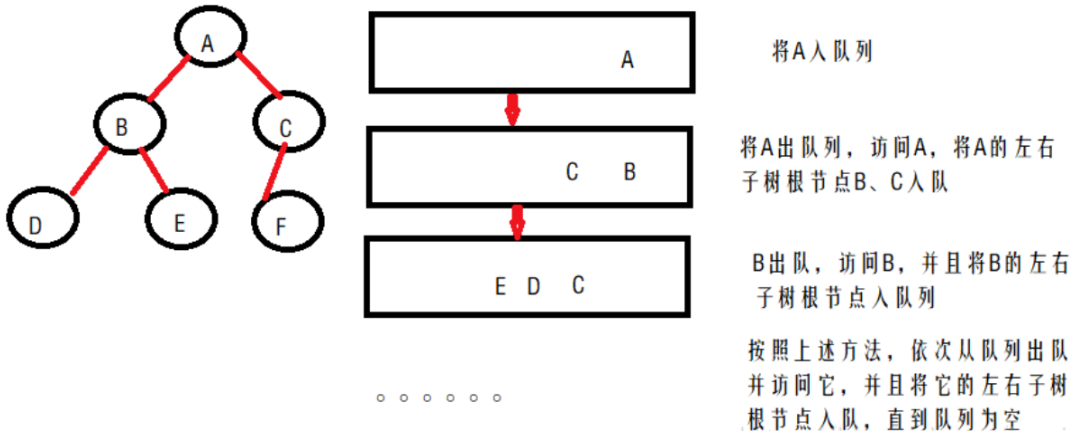
```
//先序遍历二叉树递归算法
Status PreOrderTraverse(BiTree T)
{
    if(T==NULL) return OK;//空二叉树
    else{
          visit(T->data);//访问根节点
          PreorderTraverse(T->lchild);//递归遍历左子树
          PreorderTraverse(T->rchild);//递归遍历右子树
    }
}
//中序递归遍历二叉树
Status InOrderTraverse(BiTree T)
{
    if(T==NULL) return OK;//空二叉树
    else{
          InorderTraverse(T->lchild);//递归遍历左子树
          visit(T->data);//访问根节点
          InorderTraverse(T->rchild);//递归遍历右子树
    }
}
//后序递归遍历二叉树
Status PostOrderTraverse(BiTree T)
{
    if(T==NULL) return OK;//空二叉树
    else{
          PostorderTraverse(T->lchild);//递归遍历左子树
          PoorderTraverse(T->rchild);//递归遍历右子树
          visit(T->data);//访问根节点
    }
}

//三种遍历算法不同之处仅在于访问根结点、遍历左子树、遍历右子树的先后关系。若不考虑visit()语句，则三种遍历方法完全相同（访问路径是相同的，只是访问结点的时机不同） 。
```

```
// 中序遍历二叉树的非递归算法(利用栈)
Status InorderTraverse(BiTree T)
{
    BiTree p; InitStack(S); p=T; //初始指向根结点
    while(p||!StackEmpty(S)) //树或者栈不为空时
     {
        if(p) {push(S,p);p=p->lchild;} //根不为空就入栈，访问左子树
        else {pop(S,q);printf(q->data); p=q->rchild;}
                //栈顶元素弹出放在q里，输出值，访问它的右孩子
    return OK;
}
```

二叉树层次遍历

利用队列




```
// 层次遍历二叉树的算法(利用队列)
Typedef struct{
   BTNode data[Maxsize];
   int front, rear;
} SqQueue;  //顺序循环队列

void LevelTraverse(BTNode *b)
{
    BTNode *p; SqQueue *qu;
    InitQueue(qu); //初始化队列
    enQueue(qu,b); // 根结点指针进队
    while(!QueueEmpty(qu)) { //队不空，则循环
        deQueu(qu,p);//出队结点p
        visit(p->data); // 访问p
        if(p->lchild)
            enQueue(qu,p->lchild); // 非空的左孩子指针进队
        if(p->rchild)
            enQueue(qu,p->rchild); // 非空的右孩子指针进队
     }
}
```

二叉树建立

```
//按先序次序输入结点值的方式建立二叉树T
Status CreateBiTree(BiTree &T)
{
   scan(&ch); //输入字符
   if(ch==c) //c为特殊数据(如#)用以表示空树
        T=NULL;
   else{
        T=new BTNode<ElemType>;
        T->data=ch; //生成根结点
        CreateBiTree(T->lchild); //左子树
        CreateBiTree(T->rchild); //右子树
    }
   return OK;
}
```

二叉树深度

```
//求二叉树的深度
int Depth(BiTree T)
{
    if(T==NULL)
       return 0; //如果是空树返回0
    else {
       h1=Depth(T->lchild);
       h2=Depth(T->rchild);
       return h1>h2 ? h1+1:h2+1;
}
```