《数据结构》期末复习

数组,串与广义列表

多维数组

- 多维数组是一维数组的推广,特点是可以有**多个直接前驱**与**多个直接后继**
- 数组的下标一般具有固定的上下界

二维数组

二维数组中元素的存储位置是常考的问题

二维数组中数组元素的顺序存放

- 行优先存放

  假设LOC(0,0) = a, 每个元素占用l个存储单元

```
LOC(j,k) = a + (j * m + k) * l
```

- 列优先存放

```
LOC(j,k) = a + (k * n + j) * l
```

三维数组

各维的元素个数为a,b,c

下标为i,j,k的数组元素的存储地址为:

```
LOC(i,j,k) = begin + sizeof(T) * (i * b * c + j * c + k)
```

推广至n维数组:

$$LOC(i_1, i_2, \ldots, i_n) = a + \left(\sum_{j=1}^{n-1} (i_j * \prod_{k=j+1}^{n} m_k) + i_n \right)$$

特殊矩阵

特殊矩阵是指非零或零元素分布有规律的矩阵,如:对称矩阵 三角矩阵

为节省存储空间,对可以不存储的元素不再存储

- 对称矩阵的压缩存储

  对称矩阵的元素关于主对角线对称

$$a_{ij} = a_{ji}, 0 <= i, j <= n - 1$$

将上三角部分存放在一个一维数组B中
称之为对称矩阵A的**压缩存储方式**

```
LOC(i,j)=(i+1)*i/2 + j;//case i>=j
LOC(i,j) = LOC(j,i);//case i<j
```

反过来,已知某元素为与B的第k个位置,可以寻找满足

$$i(i+1)/2 <= k < (i+1)(i+2)/2$$

的i, 此为该元素的行号.再由

$$j = k - i * (i+1)/2$$

找到该元素的列号.

- 三对角矩阵的压缩存储
  - 定义

    除了主对角线和在主对角线上,下两条对角线的元素外, 所有其他元素为0. 共有3n-2个非零元素
  - 非零元素a_{ij}满足$0 <= i <= n-1, i-1 <= j <= i+1$， 则A[i][j]在B中的位置是$2i+j-3$

稀疏矩阵

设A中有s个非零元素. 若s远小于元素总数(s << m*n), 则称A为稀疏矩阵

$e = \frac{s}{m*n}$为矩阵的稀疏因子

为了节省空间, 应只存储非零元素, 并记录它所在的行与列

- 稀疏矩阵的转置

  设矩阵列数为Cols, 对矩阵三元组扫描Cols次

  第k次扫描寻找列号为k的三元组, 交换行列值, 存储到输出三元组表中

矩阵的正交链表表示

有效表示动态变化的矩阵结构

稀疏矩阵表示为**行链表**与**列链表**的十字交叉, 称为正交链表

字符串

串的模式匹配

- 简单的模式匹配算法

  模式匹配:子串的定位操作

```
int Index(SString S, SString T){
    int i=1,j=1;
    while(i <= S.length && j <= T.length){
        if(S.ch[i] == T.ch[j]){
            i++;
            j++;
        }else{
            i = i - j + 2;
            j = 1;
        }
```

```
        if(j > T.length){
            return i - T.length;
        }else{
            return 0;
        }
    }
}
```

- KMP算法

  4.2.1中的匹配方式我们不难发现每趟匹配失败后都要从头比较, 我们可以优化这个算法

  如果*已经匹配相等*的前缀序列中有某个后缀正好是模式的前缀, 可以将模式向后滑动到与这些相等字符对齐的位置而无需回溯主串的i指针

  设模式P = $p_0p_1\dots p_{m-2}p_{m-1}$,则它的next特征函数定义如下:

$$next(j)=\begin{cases}-1 & j=0\\ k+1 & 0<=k<j-1 \& p_0p_1\dots p_k=p_{j-k-1}\dots p_{j-1}\\ 0 & else\end{cases}$$

```
void AString::getNext(int next[]){
    int j = 0, k = -1, lengthP = curLength;
    next[0] = -1;
    while(j<lengthP){
        if(k == -1 || ch[j] == ch[k]){
            j++;
            k++;
            next[j] = k;
        }else{
            k = next[k];
        }
    }
}
/*
1.如果当前ch[i] == ch[k], 那么根据定义, next(j)使得p(0,k) = p(j-k-1,j-1), 又有p(k+1)=p(j),
next(j+1) = next(j) + 1 = k + 1;
2.p(k+1)!=p(j),则我们找h使得p(0,h)=p(k-h,k)=
p(j-h-1, j-1),即h=next(k);
    (1)p(h+1) = p(j),由next(j+1)的定义,
    next(j+1)=h+1=next(k)+1=next(next(j))+1
    (2)p(h+1)!=p(j),则在p(0,h)中找更小的next(h)=1递      归到next(t)=-1
*/
```

根据得到的next函数, 我们可以得到KMP的快速匹配算法如下

```
int AString::fastFind(AString& pat, int k, int next[])const{
    int posP = 0; posT = k;
    int lengthP = pat.curLength;
    int lengthT = curLength;
    while(posP < lengthP && posT < lengthT){
        if(posP == -1 || pat.ch[posP] == ch[posT]){
            posP++;
            posT++;
        }else{
            posP = next[posP];
        }
    }
    if(posP < lengthP){
```

```
        return -1;
    }else{
        return posT - lengthP;
    }
}
```

广义表

关键在于会画广义表即可

- 广义表的定义和性质

  广义表的定义是递归的, 因为在表的描述中又用到了表, 允许表中有表

  习惯上称第一个表元素为**表头**,称除了表头外其他元素组成的表为**表尾**

- 广义表的表示

  通常用链表作为广义表的存储表示

  考虑到广义表中的表元素可能是子表的情形, 可以存放一个指向子表的指针

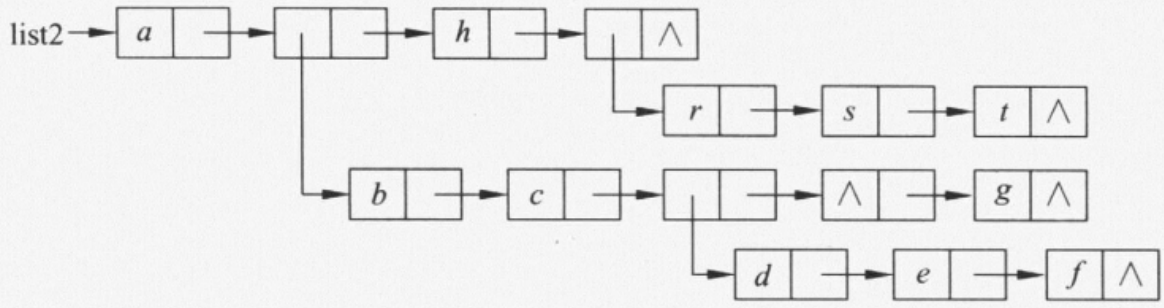


图 4.26　表中套表情形下的广义表链表表示

- 存储实现

  链表的每个表结点由3个域组成

  - 标志域utype

    标明该节点的类型: 0:附加头节点； 1:原子节点； 2:子表节点

```
- 信息域info

  utype = 0时, 该信息域存放引用计数;

  utype = 1时, 存放value

  utype = 2时, 存放指向子表标头的指针(hlink)

- 尾指针tlink
  utype = 0时, 存放指向该表表头元素节点的指针;
  utype != 0时, 存放同一层下一个表头节点的地址
```

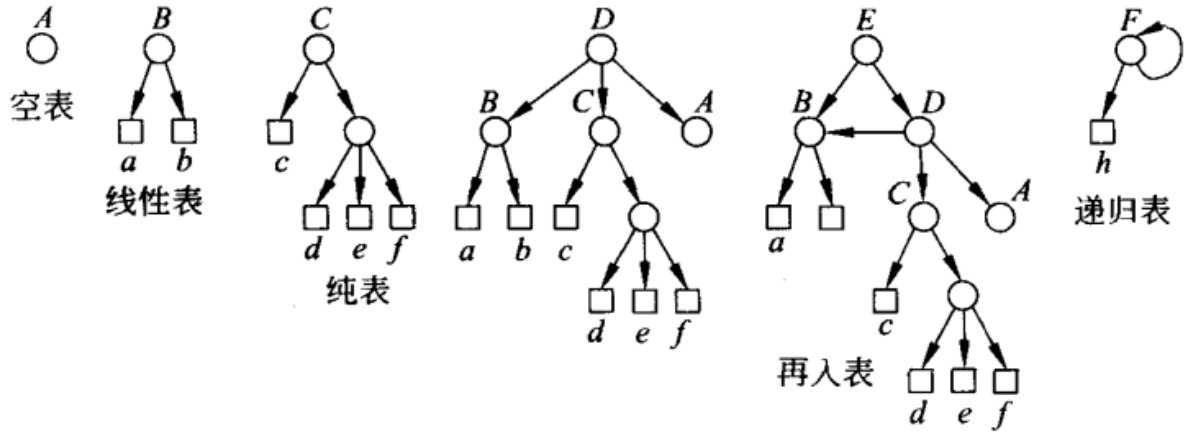


图 4.24　各种广义表的示意图

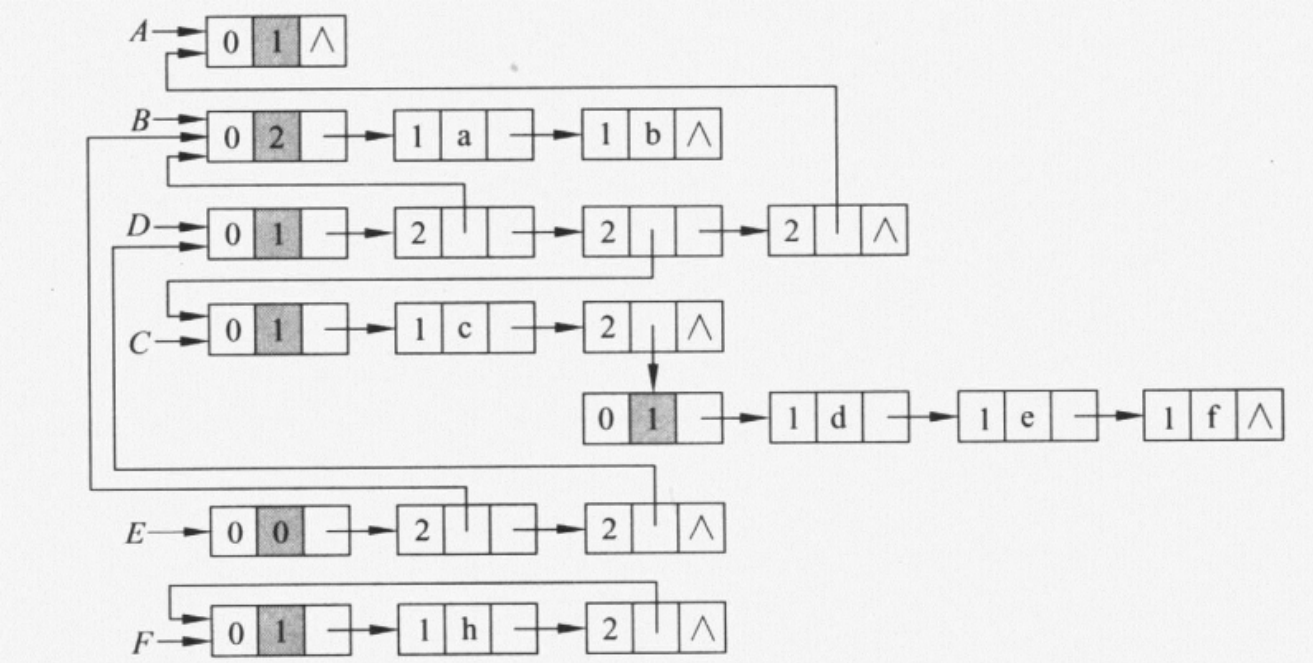


图 4.28　广义表的带附加头结点的存储表示

递归算法

对广义表的操作一般都是**递归的**，具体代码很容易

树与二叉树

树的基本概念

- 子女：子树节点的根
- 双亲：结点的父节点
- 兄弟：同一节点的子女之间互为兄弟
- 度：结点的子女个数；树中节点的度的最大值为**树的度**
- 分支节点：**度**不为0的节点
- 叶节点：**度**为0的节点
- 祖先：双亲以及双亲的祖先

- 子孙：子女和子女的子孙
- 结点的层：规定根节点是第一层
- 深度：结点的层次；最大层数为**树的深度**
- 高度：树的高度为根结点的高度
- 有序/无序树：结点的子女之间是否有次序

二叉树

- 性质：如果二叉树T的叶节点有n_0个，度为2的节点有n_2个，那么$n_0 = n_2 + 1$

  > $n = n_0 + n_1 + n_2$;
  >
  > $e = 2n_2 + n_1 = n - 1$;
  >
  > 变换可得

- 满二叉树：被节点~~注满~~的二叉树
- 完全二叉树：除了最后一层都达到了满层，且最后一层的节点从左向右依次顺序排列，中间不空（满二叉树是完全二叉树，反过来完全二叉树不一定是满二叉树！）
- 如果将一颗n个结点的完全二叉树自顶向下，自左向右按层次遍历序编号，有以下关系：
  - i = 1，则i无双亲
  - i > 1, 双亲为[i / 2]
  - 若2 * i <= n, i的左子女为 2 * i

    若2 * i + 1 <= n, i的右子女为2 * i + 1
  - 若i为奇数, i != 1; 左兄弟为i - 1
  - 若i为偶数, i != n, 右兄弟为i + 1
- 二叉链表/三叉链表表示

二叉树遍历

前序 中序 后序 层次序遍历(写烂了)

用栈进行非递归遍历

线索化二叉树

相当于是按遍历序确定每个结点的前驱和后继

为了节省空间, 可以增加ltag和rtag, tag = 0表示指针域是子女指针, tag = 1表示指针域是线索指针

树与森林

- 一般的树表示

  1. 广义表法
  2. 双亲表示法(每个节点上存储它的父节点)
  3. 子女链表表示法(每个节点存储它的子女信息(用链表))
  4. 子女指针
  5. 子女-兄弟链表法

```
struct TreeNode{
    int data;
    TreeNode* firstchild;
    TreeNode* nextSibling;//存储它的兄弟的链表
}
//要找到结点的某个子女,先走到firstchild,再在firstchild->nextSibling中遍历即可
```

- 森林的二叉树表示法

  会画即可

- 树的遍历

  深度遍历(先根 后根) 广度遍历

堆

- 堆的子女计算

  由于堆的节点被存储在线性数组中, 可以很方便地找到结点的父节点和子女节点

  1. i = 0, i是根节点; i > 0, i的父节点为[(i - 1) / 2]
  2. i的左子女为2 * i + 1
  3. i的右子女为2 * i + 2

- 堆的下滑调整算法

```
/*从start到m为止, 自上而下比较, 将关键码小的上浮*/
void MinHeap::siftDown(int start, int m){
    int i = start, j = 2 * i + 1;//j是i的左子女
    E temp = heap[i];
    while(j <= m){
        if(j < m && heap[j] > heap[j + 1]){j++;}
        if(temp <= heap[j]){
            break;
        } else {
            heap[i] = heap[j];
            i = j;
            j = 2 * j + 1;
        }
    }
    heap[i] = temp;
}
```

集合与字典

集合

位向量可以用来表示有限个可枚举的成员, 将成员a_i与整数i一一对应

$Vec[i] == 1, iff(a_i \subset S)$

对于需要无限增加成员的集合, 可以用**有序链表**

并查集(UFSets)

用来描述集合中的**等价关系**

支持操作:

1. `Union(Root1, Root2)` - 合并操作
2. `Find(x)` - 查找操作

```
//Methods of UFSets

//Find the root of x, x与y等价相当于Find(x) == Find(y)
void UFSets::Find(int x){
    if(parent[x] < 0) {
        return x;
    }
    return Find(parent[x]);
}

//Union x and y, 在集合中添加x与y的等价关系
void UFSets::Union(int Root1, int Root2){
    int r1 = Find(Root1);
    int r2 = Find(Root2);
    parent[r1] += parent[r2];//r1的集合大小增加r2
    parent[r2] = r1;//r2集合被归到r1下方
}

/*为了避免产生退化的树, 我们可以使用
带权的合并算法(因为把2插到1底下和把1插到2底下都是合法的算法)
如果r1的大小更大, 就以r1为根, 将r2插到r1的下方
(需要注意大小是以负数的形式存储的)*/
void UFSets::WeightedUnion(int Root1, int Root2) {
    int r1 = Find(Root1), r2 = Find(Root2), temp;
    if(r1 == r2){
        return;
    }
    if(r1 != r2) {
        temp = parent[r1] + parent[r2];
    }
    if(parent[r2] < parent[r1]){
        parent[r1] = r2;
        parent[r2] = temp;
    }else{
        parent[r2] = r1;
        parent[r1] = temp;
    }
};
```

字典

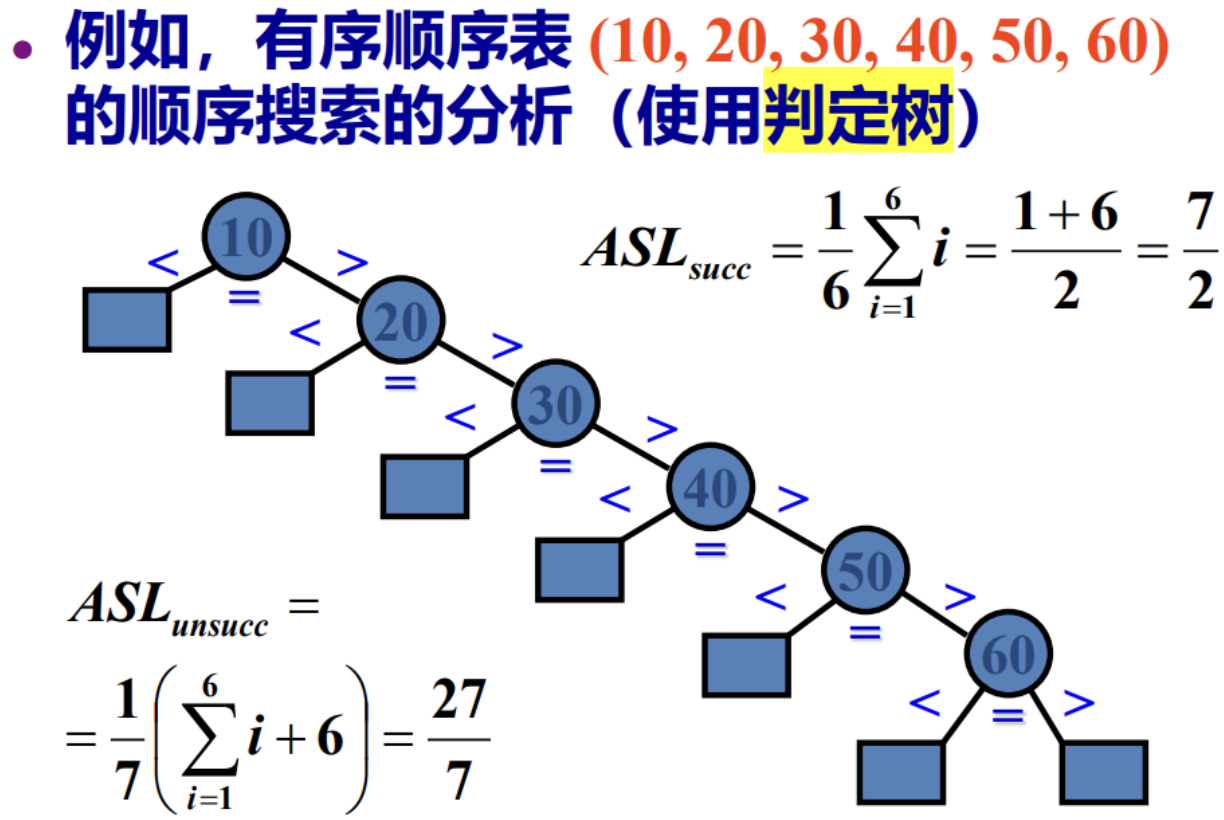
字典是<名字 - 属性>对的集合, 一般字典支持如下操作:

1. 确定名字key是否在字典中
2. 搜索key对应的value(并修改)
3. 增加/删除一对< key-value >

字典可以用**有序顺序表**和**有序链表**来描述

判定树

一种扩充二叉树, 了解其画法即可



散列表(Hash Table)

希望找到一种结构, 可以$O(1)$地从字典中取出key对应的value

构造一个`Hash Function`, 使得`Address = Hash(key)`

- 取址方式

  主要有以下几种取地址的方式:

  1. 直接定址

     取一个线性映射

     `Hash(key) = a * key + b`

     不产生冲突, 但对空间的要求严格

  2. 数字分析

     沙比公式看不懂, 应该不考

  3. 余数法(最常考!!!)

     将key进行一个模运算的映射

     `hash(key) = key % p`

     其中p是一最接近地址数m的素数

  4. 平方取中法

将key的计算机内码平方之后取中间几位作为地址

5. 折叠法

   将关键码分为位数等于散列表地址位数的几部分, 将其叠加

   一般当关键码的位数很多，而且关键码每一位上数字的分布大致比较均匀时，可用这种方法得到散列地址。

- 处理冲突的方法

  1. 线性探查法(闭散列)

     需要搜索/插入时, 计算出 `H0 = hash(key)`

     若发生冲突, 顺次找下一个位置 `Hi = (Hi-1 + 1) % m`

     搜索成功的平均搜索长度的分母为搜索次数

     搜索不成功的平均搜索长度的分母为m

  2. 链地址法(开散列)

     关键码按它的hash值被划分为一系列子集, 称为**桶**

搜索结构

搜索

在集合中寻找满足某种条件的数据对象

- 静态/动态搜索

  静态：搜索结构在插入和删除等操作的前后不发生改变

  动态：搜索结构可能会发生变化

- 顺序搜索

  就是最基本的挨个找过去

  监视哨：可以将位置n的元素作为“监视哨”（元素在1 - n-1的位置存放）

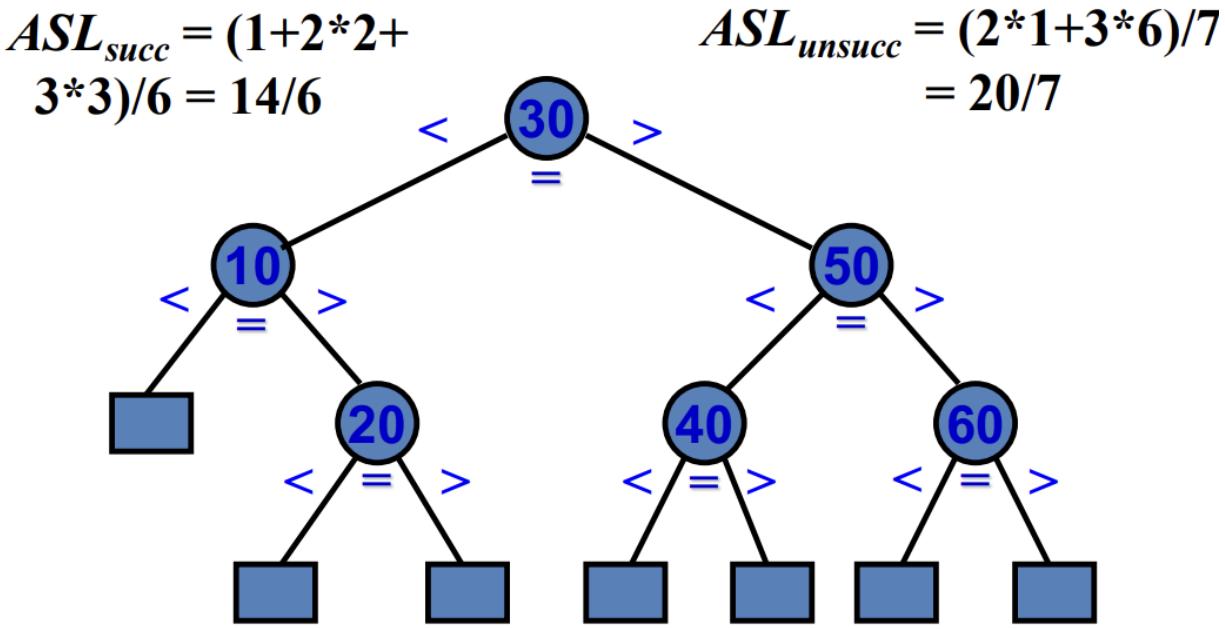
- 折半搜索

  基于一个**有序**的顺序表中

  每次求区间的中间值与搜索对象比较，这样每次可以直接排除半个区间

```
int BinarySearch(const K &x, const int low, const int high){
    int mid = -1;
    if(low <= high) {
        mid = (low + high) / 2;
        if(Element[mid] < x){
            mid = BinarySearch(x, mid + 1, high);
        }else if(Element[mid] > x){
            mid = BinarySearch(x, low, mid - 1);
        }
    }
    return mid;
}
```

搜索判定树:



- 二叉搜索树

  左子树的所有关键码小于根的关键码, 右子树的所有关键码大于根的关键码

  搜索和插入算法很简单

  删除算法: 对于两子树非空, 可以寻找右子树中序下的第一个节点(即右子树的最小节点), 将它的值填补到被删除的节点位置, 问题转化为删除那个最小的节点

AVL树

高度平衡的二叉树

> 高度平衡保证了搜索性能的稳定性, 普通的二叉搜索树在最坏的情况下依然需要$O(N)$级别的复杂度

AVL树的左子树和右子树的高度差绝对值不超过一, 且它们也都是AVL树

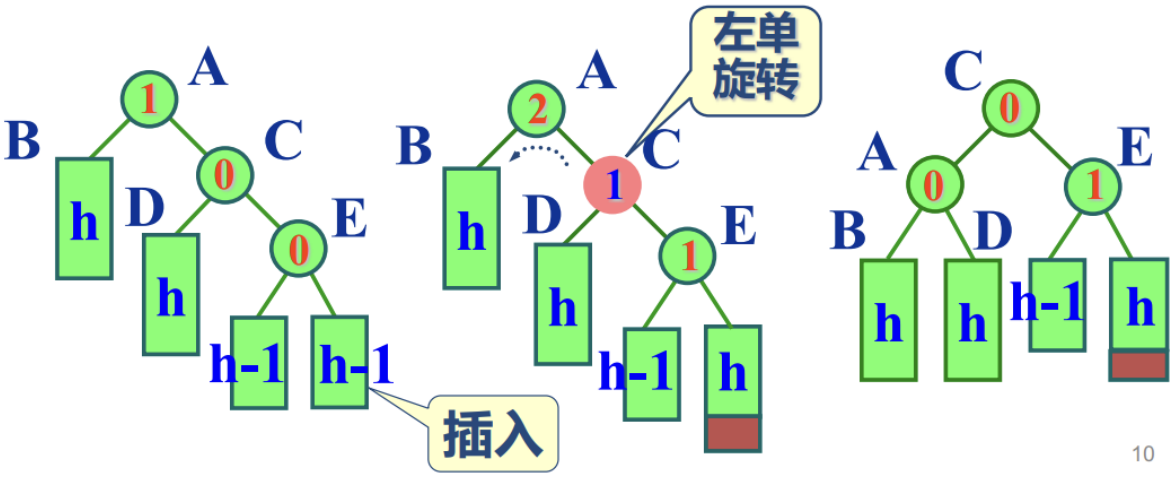
- 单旋转和双旋转

  为了维护AVL树的性质, 在插入和删除时, 如果遇到不平衡的情况, 我们需要进行单旋转和双旋转

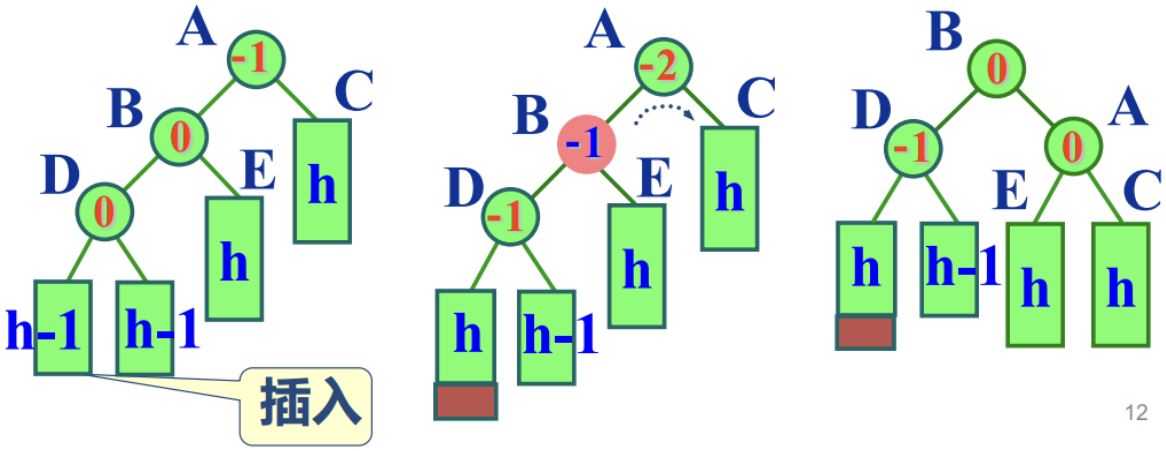
  单旋转始终用于三个点在同一边的情况(例如在右子树的右子树中插入), 双旋转始终用于尖括号形状的情况(例如在左子树的右子树 / 右子树的左子树中插入)

  > 记忆方法: 单旋转与插入方向相反(例如在右子树的右子树插入就是左单旋) 双旋转与插入方向相同(例如在左子树的右子树插入就是先左后右双旋)
  >
  > 实际上画图也能感知到是如何旋转的

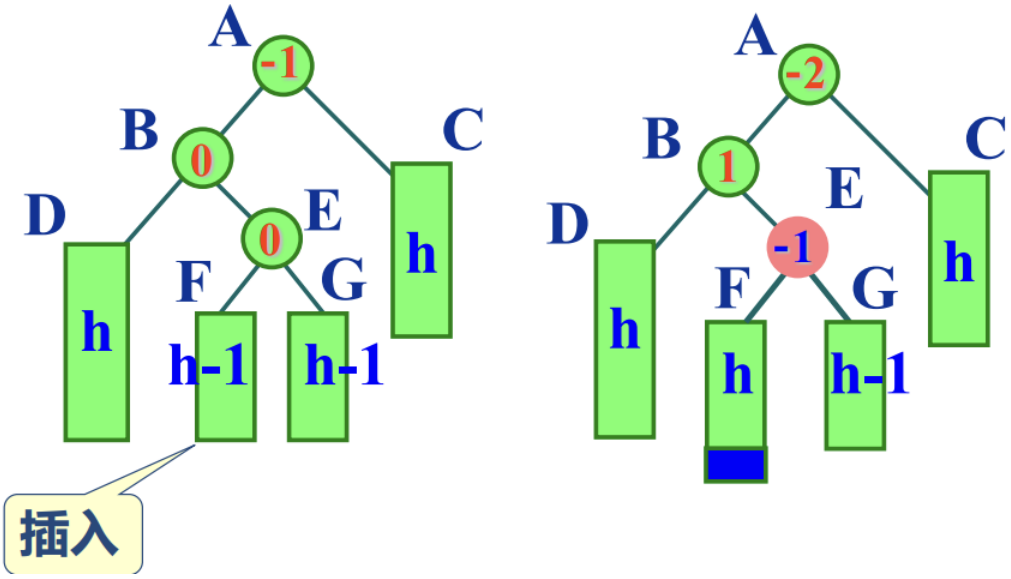
  左单旋

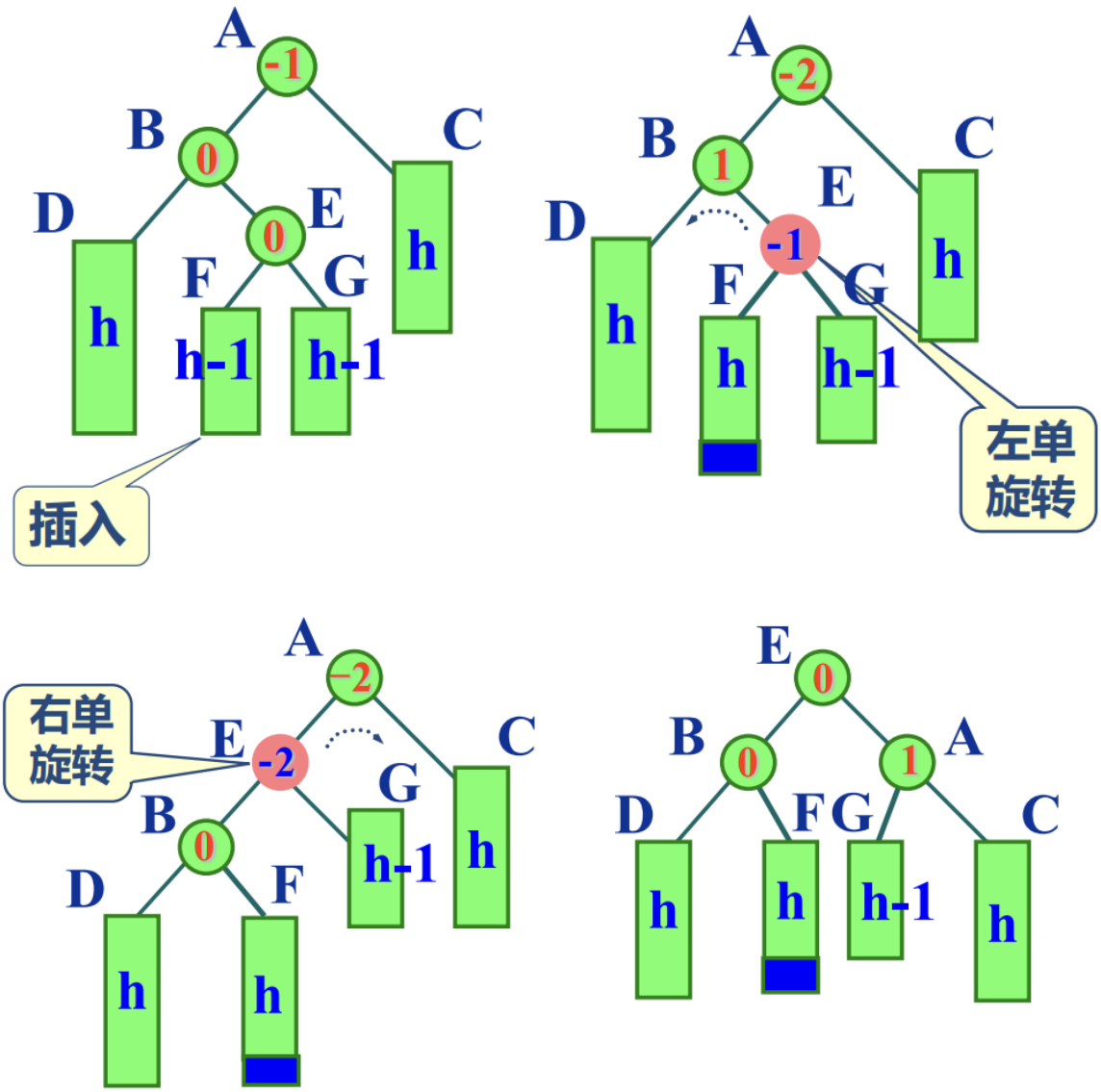

A
B
C
D
E
h
h
h
h-1
h-1
1
0
0
插入
左单旋转
2
1
1
10

右单旋


A
B
C
D
E
h
h
h
h-1
h-1
-1
0
0
插入
-2
-1
-1
12

先左后右


A
B
C
D
E
F
G
h
h
h
h-1
h-1
-1
0
0
插入
-2
1
-1



- 插入算法

  插入后需要向根节点回溯

  在某一节点发现不平衡则以这一节点为根节点做旋转

  如果节点的bf == 2, 则说明右子树高, 在右子树中寻找. 如果右子树的bf == 1, 说明右子树的右子树高, 左单旋; 反之说明左子树高, 进行先右后左双旋

- 删除算法

  基本的删除逻辑和普通的二叉搜索树一致

  区别在于需要**沿x的父节点上溯**来检测平衡是否被打破

  用一个`Bool shorter`来传递子树的高度是否被改变

  1. 当前结点的bf为0, 那么一个子树缩短显然不影响它的总高度和平衡性, 同时它以上的节点也感知不到它的子树的高度变化, 上溯停止
  2. 当前结点的bf为1, 且在较高的树删除

     该节点bf变化为0, 同时这棵树的高度减少了, shorter设置为`True`, 继续上溯
  3. 当前结点的bf为1, 且在较矮的树删除

这棵树的平衡性被打破了, 因此我们需要进行平衡化旋转, 之后再考察这颗树是否变矮了, 来传递shorter的值

图

基本概念

$Graph = (V, E)$，即图中的顶点与点之间的边

有向图与无向图

完全图：图中边数达到最大

权：边的数值，带权图称之为**网络**

路径：由一系列邻接点构成的序列

简单路径：点不重复的路径

回路：路径的始点与终点重合

连通图：任意顶点都连通的**无向图**

连通分量：非连通图的**极大连通子图**

强连通图：任意顶点都**互相连通**的**有向图**

强连通分量：**非强连通图**的**极大强连通子图**

> 即所有”强“都只与有向图有关，不带”强“的只与无向图相关

生成树: 无向连通图的生成树是他的**极小连通子图**

存储结构

- 邻接矩阵表示法

  建立一个矩阵E,其中$E[i][j] = (i, j) \subset E ? 1 : 0$

  > 由于从邻接矩阵来看图中的边数以及是否连通的时间开销很高, 我们转向邻接表等其他结构

- 邻接表表示法

  相当于是一个链表的数组. 其中第i个链表存放的是以顶点i为起始顶点的所有边, val存放的是终点j_i, 如果是有权图还需要存放边的权值

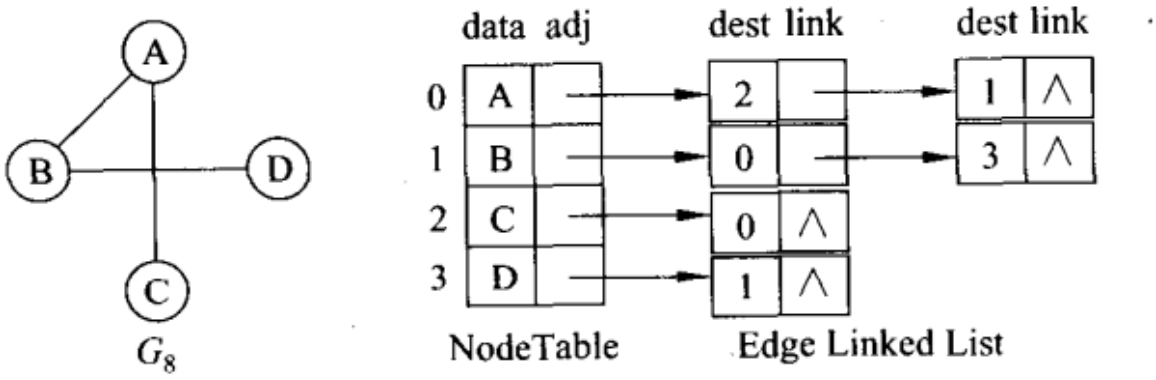


图 8.7　无向图的邻接表表示

这种表示相较于邻接矩阵的好处在于省去了大量的空余空间,只存放存在边的信息,对于点多边少的情况改善很大.

反过来当边很稠密的时候, 邻接矩阵占优, 因为它存储一个边只要占一个`int`的大小, 反而邻接矩阵要消耗一个`Linknode`的大小

对于有向图, 单个邻接表只能方便计算该点的出度. 如果还想要方便计算点的入度, 可以另外开一个逆邻接表(入边表)

> 邻接表仍有不满意的地方在于每条边被存储了两遍, 当我们需要标记某条边的时候需要标记两次. 因此我们又提出邻接多重表

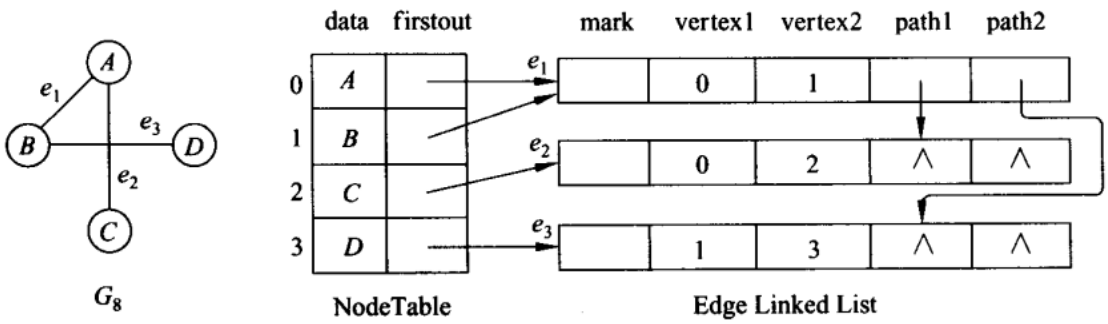
- 无向图的邻接多重表

  邻接多重表的边节点存储:

```
struct Edge {
    int mark;//该边是否被标记过
    Vertex *vertex1, *vertex2;//该边的两个邻接点
    Edge *path1, *path2;//依附于vertex1和vertex2的下一条边
};
```

  点节点存储:

```
struct Vertex{
    int data;
    Edge* firstout;//记录依附该点的第一条边
}
```



- 有向图的邻接多重表

  Edge的定义与无向边基本相同, Vertex增加一个firstin指针, 两个指针分别指向以该V为始顶点(终顶点)的第一条边

图的遍历算法

- DFS

```
void dfs(Graph &G, const T& v) {
    int i, loc, n = G.V.size();
    bool *visited = new bool[n];
    for(i = 0; i < n; i++) {
        visited[i] = false
    }
    loc = G.getVPos(v);
    DFS(G, loc, visited);
    delete[] visited;
}

void DFS(Graph &G, int v, bool visited[]) {
    visit(v);
    visited[v] = true;
    int w = G.getFirstNeighbor(v);
    while(w != -1) {
        if(visited[w] == false) {
            DFS(G, w, visited);
            w = G.getNextNeighbor(v,w);
        }
    }
}
```

> 相当于是走到一个点之后依次遍历该点的所有边, 一直遍历到头, 优先考虑遍历的深度, 因此叫"深度优先搜索"
>
> 如果采用邻接表来表示图, 顶点的邻接顶点可以被O(1)的时间内取出, 复杂度为O(n + e)
>
> 如果采用邻接矩阵来表示，则需要O(n^2)的时间复杂度

- 广度优先搜索

```
void BFS(Graph &G, T &v) {
    int w, n = G.NumberOfVertices;
    bool *visited = new bool[n];
    for(int i = 0; i < n; i++) {
        visited[i] = false;
    }
    int loc = G.getVertexPos(v);
    visited[loc] = true;
    queue<int> Q;
    Q.push(loc);
    while(!Q.empty()) {
        loc = Q.front();
        Q.pop();
        w = G.getFirstNeighbor(loc);
        while(w != -1) {
            if(visited[w] == false) {
                visit(w);
                visit[w] = true;
                Q.push(w);
            }
            w = G.getNextNeighbor(loc, w);
        }
```

```
    }
};
```

最小生成树

相当于是找权值和最小的生成树. 这个问题不平凡(不能直接选择最小的n-1条边)在于选择的边之间可能会构成**回路**, 我们需要避免这种情形

- Kruskal算法

  设有一个有 n 个顶点的连通网络 N = { V, E }, 最初先构造一个只有 n 个顶点、没有边的非连通图 T = { V, ∅ }, 图中每个顶点自成一个连通分量。当在 E 中选到一条具有最小权值的边时, 若该边 的两个顶点落在不同的连通分量上，则将此边加 入到 T 中; 否则将此边舍去，重新选择一条权值最小的边。如此重复下去, 直到所有顶点在同一个连通分量上为止。

  算法实现: 利用 *最小堆* 和 *并查集* 来实现

  > 最小堆是一种优秀的排序结构, 而并查集可以清晰表示图中连通分量的情况

```
void Kruskal(Graph &G, MST &mst) {
    int u, v, count;
    int n = G.NumberOfVertices();
    int m = G.NumberOfEdges();
    MSTEdgeNode edge;
    MinHeap<MSTEdgeNode> H(m);
    UFSets F(n);
    for(u = 0; u < n; u++){
        for(v = u + 1; v < n; v++){
            if(G.getWeight(u, v) != INT32_MAX){
                edge.tail = u;
                edge.head = v;
                edge.cost = G.getWeight(u, v);
                H.Insert(edge);
            }
        }
    }//初始化最小堆，将所有边插入
    count = 0;
    while(count <= n - 1) {
        H.Remove(edge);
        u = F.find(edge.tail);
        v = F.find(edge.head);
        if(u != v) {
            F.Union(u,v);
            mst.Insert(edge);
            count++;
        }
    }
}
```

- Prim算法

  从某一个顶点u出发，选择与它关联的**最小权值的边**， 将其顶点加入到生成树顶点集合U之中。

  之后的每一步从一个顶点在U中而另一个顶点不在的边的集合中选择权值最小的一个，把它的顶点加入到U中，如此往复。

```
//Prim算法
void Prim(Graph &G, const T u0, MST &mst) {
    MSTEdgeNode edge;
```

```
    int i,u,v,count;
    int n = G.NumberOfVertices();
    int m = G.NumberOfEdges();
    int u = G.getVertexPos(u0);
    MinHeap<MSTEdgeNode> H(m);
    bool Vmst = new bool[n];
    Vmst[u] = true;
    count = 1;
    do{
        v = G.getFirstNeighbor(u);
        while(v != -1){
            if(!Vmst[v]){
                edge.tail = u;
                edge.head = v;
                H.Insert(edge);
            }
            v = G.getNextNeighbor(u,v);
        }
        while(!H.IsEmpty() && count < n) {
            H.Remove(edge);
            if(!Vmst[edge.head]){
                MST.Insert[edge];
                u = edge.head;
                Vmst[u] = true;
                count++;
                break;
            }
        }
    }while(count < n);
}
```

> 复杂度分析: K算法适用于边稀疏的网络(这样使得并查集的体积小), P算法适用于边稠密的网络

最短路径

- Dijkstra算法

  用于解决**边权值非负**的单源最短路径问题

```
//Dijkstra算法
/*相当于是每次找到集合U外的dist最小值, 将它加入U, 再将U更新, 直到所有顶点都被加入到了U中*/
void ShortestPath(Graph &G, T v, E dist[], int path[]){
    int n = G.NumberOfVertices();
    bool *S = new bool[n];
    int i, j, k;
    E w, min;
    for(i = 0; i < n; i++) {
        dist[i] = G.getWeight(v,i);
        S[i] = false;
        if(i != v && dist[i] < maxValue)
            path[i] = v;
        else
            path[i] = -1;
    }//初始化dist, dist[i]为v-i之间边的值(-1表示无边)
    S[v] = true;
    dist[v] = 0;
    for(i = 0; i < n-1; i++){
        min = maxValue;
```

```
        int u = v;
        for(j = 0; j < n; j++){
            if(!S[j] && dist[j] < min){
                u = j;
                min = dist[j];
            }
        }
        S[u] = true;
    }
    for(k = 0; k < n; k++){
        w = G.GetWeight(u,k);
        if(!S[k] && w < maxValue && dist[u] + w < dist[k]){
            dist[k] = dist[u] + w;
            path[k] = u;
        }
    }
}
```

- Floyd算法

  求出的是每一对顶点之间的最短路径

  > 实际上是一种动态规划的思想. 对于n个顶点的图G, 求Vi -> Vj的最短路径被分为多个阶段:
  >
  > #0: 如果允许在V0中转, 最短路径?
  >
  > #1: 如果允许在V1中转, 最短路径?
  >
  > ...
  >
  > #n-1: 如果允许在V0 V1 ... Vn-1中转, 最短路径是?

```
void Floyd(Graph G, E a[][], int path[][]) {
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++){
            a[i][j] = G.getWeight(i,j);
            if(i != j && a[i][j] < maxValue)
                path[i][j] = i//path[i][j]中存放的是相应路径上j顶点的前一顶点的顶点号
            else
                path[i][j] = 0;
        }//初始化
        for(k = 0; k < n; k++){
            for(i = 0; i < n; i++){
                for(j = 0; j < n; j++){
                    if(a[i][k] + a[k][j] < a[i][j]){
                        a[i][j] = a[i][k] + a[k][j];
                        path[i][j] = path[k][j];
                    }
                }
            }
        }
    }
}
```

> Floyd算法允许带负值的边,但不能出现带负权值的边组成的回路

活动网络

- DAG图: 有向图中不存在环(Directed Acyclic Graph)
- 活动网络:

  可以用**有向图**来表示一个工程, V(i, j)表示i必须先于j进行

  AOV中显然不能出现有向回路

- 拓扑排序

  检测有向环的一种方法，它将各顶点排列成一个线性有序的序列（可以理解为映射到实数中， 先后关系被抽象化成了实数之间的大小关系）

  如果所有顶点都能被排入一个拓扑有序的序列中，网格中则不会出现有向环

- 拓扑排序的方法

  重复在AOV网络中选择一个**没有直接前驱**的节点(相当于是极大的节点), 输出, 并在图中删除该顶点与其发出的所有有向边

  重复上述操作, 如果全部顶点被输出, 那么拓扑排序则被完成; 反之如果图中还有未输出的顶点, 则说明图中存在有向环

```
void TopologicalSort(Graph &G){
    int i, j, w, v;
    int top = -1;
    int n = G.NumberOfVertices();
    int *count = new int[n];
    for(i = 0; i < n; i++)
        count[i] = 0;
    cin >> i >> j;
    while(i > -1 && i < n && j > -1 && j < n){
        G.insertEdge(i,j);
        count[j]++;
        cin >> i >> j;
    }
    for(i = 0; i < n; i++) {
        if(count[i] == 0){
            count[i] = top;
            top = i;
        }
    }

    for(i = 0 ; i < n; i++){
        if(-1 == top){
            exit(0);//有回路
        }
        v = top;
        top = count[top];
        w = G.GetFirstNeighbor(v);
        while(w != -1){
            count[w]--;
            if(count[w] == 0) {
                count[w] = top;
                top = w;
            }
            w = G.GetNextNeighbor(v,w);
        }
    }
}
```

- AOE网络

  用**边**表示活动的网络, 边的权值表示活动的持续时间

  AOE用于工程估算,例如:

  - 完成工程需要多少时间?
  - 为了缩短时间需要加快哪些活动?

  整个工程只有一个开始点和一个完成点, 开始点称**源点**, 结束点称**汇点**

  我们关心整个工程的完成时间, 这取决于从源点到汇点的**最长路径长度**,我们称其为**关键路径**

  定义几个辅助数学量:

  - 事件V_i的最早可能开始时间 $Ve(i)$

    从V_0到V_i的*最长路径长度*

  - 事件V_i的最迟可能开始时间 $Vl(i)$

    是保证V_{n-1}在$Ve(n-1)$时刻完成的前提下V_i的最迟开始时间

    > 显然由于选择的不同, 对于大部分的事件V_i我们可以早一些完成也可以迟一些完成, 因此V_i的完成有一个范围

  - 活动a_k的最早可能开始时间$Ae[k]$

    显然有$Ae[k]=Ve[k.start]$, 即活动k需要开始至少需要经过从始点到k的始边的最大路径

  - 活动a_k的最迟允许开始时间$Al[k]$

    在不引起延误的前提下, 该活动允许开始的最迟事件

    $Al[k]=Vl[j]-length(a_k)$

  - **时间余量** $Al[k]-Ae[k]$

    没有时间余量的活动即是**关键活动**

- 求解$Ve[i]$与$Vl[i]$

  从$Ve[0]=0$开始, 向后递推

  $Ve[j]=max(Ve[i]+dur<V_i,V_j>)$

  同样的, 从$Vl[n-1]=Ve[n-1]$开始, 向前递推

  $Vl[j]=min(Vl[k]-dur<V_j,V_k>)$

- 代码表示

```
void CriticalPath(Graph &G){
    int i, j, k;
    E Ae, Al, dur;
    int n = G.NumberOfVertices();
    E *Ve = new E[n];
    E *Vl = new E[n];
    for(i = 0; i < n; i++) {
        Ve[i] = 0;
    }

    for(i = 0; i < n; i++){
        j = G.getFirstNeighbor(i);
        while(j != -1){
            dur = G.getWeight(i,j);
            if(Ve[i] + dur > Ve[j]) {
                Ve[j] = Ve[i] + dur;
            }
```

```
            j = G.getNextNeighbor(i,j);
        }
    }
    Vl[n-1] = Ve[n-1];
    for(j = n-2; j > 0; j--) {
        k = G.getFirstNeighbor(j);
        while(k != -1){
            dur = G.getWeight(j,k);
            if(Vl[k] - dur < Vl[j]){
                Vl[j] = Vl[k] - dur;
            }
            k = G.getNextNeighbor(j, k);
        }
    }
}
```

排序

基本概念

- 排序算法的稳定性

  对于两个排序码相等的函数，如果排序不改变他们的前后关系，那么我们称这个排序方法**稳定**

- 内排序和外排序

  内排序指数据元素全部放在内存中的排序，外排序则是指元素个数太多以至于我们需要将元素在内外存之中进行移动

- 排序的时间开销

  排序的时间开销可用**数据比较次数KCN**与**数据移动次数RMN**来衡量

  一般按照**平均情况**估计，对于受到初始排列顺序较大的算法，需要按照**最好**和**最坏**的情况估算

  算法执行时所需要的附加存储也是评价算法好坏的一种标准

直接插入排序

当插入第 i 个元素时前面的元素已经排好序，用$V[i]$的排序码与前面的元素顺序比较，找到位置插入后，原来位置上的元素向后顺移

时间复杂度分析：

- 最好情况下已经排好序，比较次数为 n - 1， 移动次数为0
- 最坏情况下恰好按照逆序排列，每次都要比较i次并且做i次移动
- 平均的时间复杂度为$O(N^2)$
- 是**稳定的排序方法**

折半插入排序

与直接插入排序的不同之处在插入$V[i]$的时候采用**折半搜索法**

希尔排序

该算法的思想是取一个gap作为间隔，将元素分为gap个子列

所有距离为gap的元素放在同一个子序列中，在每一个子序列中实行直接插入排序

然后缩小间隔gap，重复上述操作。直到最后gap == 1，所有元素都被重新放在一个序列中排序

```
void ShellSort(dataList<T> &L, const int left, const int right) {
    int i, j, gap = right - left + 1;
    Eelement<T> temp;
    do{
        gap = gap / 3 + 1;
        for(i = left + gap; i <= right; i++){
            if(L[i] < L[i - gap]) {
                temp = L[i];
                j = i - gap;
                do{
                    L[j + gap] = L[j];
                    j = j - gap;
                }while ( j >= left && temp < L[j]);
                L[j + gap] = temp;
            }
        }
    }while(gap > 1);
}
```

- 分析

  gap的取法并不确定

  是一种**不稳定**的排序算法

快速排序

基本思想是取待定序列中的某个元素作为**基准**，按照该元素码的排序码大小作为基准，将整个元素划分为两个子序列：

- 左侧子序列的所有元素的排序码都比基准元素的排序码小
- 右侧子序列中的所有元素都比基准元素大

再分别对两个子序列重复这一算法

```
template<class T>
void QuickSort(dataList<T> &L, const int left, const int right){
    if(left < right){
        int pivotpos = L.Partition(left, right);
        QuickSort(L, left, pivotpos - 1);
        QuickSort(L, pivotpos + 1, right);
    }
}

int Partition(const int low, const int high) {
    int pivotpos = low;
    Element<T> pivot = Vector[low];
    for(int i = low + 1; i <= high; i++){
        if(Vector[i] < pivot){
            pivotpos++;
```

```
            if(pivotpos != i){
                Swap(Vector[pivotpos], Vector[i]);
            }
        }
    }

    Vector[low] = Vector[pivotpos];
    Vector[pivotpos] = pivot;

    return pivotpos;
}
```

- 分析

  快速排序是一个**递归的算法**

  根据递归树的大小，不难看出快排的平均时间复杂度在$O(nlog(n))$的级别

  但在最坏的情况下，待排序元素序列已经排好序，此时递归树退化成了一条单链，时间复杂度达到了$O(N^2)$

  是一种**不稳定**的排序

  在N很小的时候，直接用插入排序即可

起泡排序（冒泡排序）

基本方法：每趟排序从后向前，按次序两两比较，如果发生逆序就交换

```
void BubbleSort(dataList<T> &L, const int left, const int right){
    int pass = left + 1, exchange = 1;
    while(pass <= right && exchange){
        exchange = 0;
        for(int j = right; j >= pass; j--){
            if(L[j - 1] > L[j]) {
                Swap(L[j - 1], L[j]);
                exchange = 1;
            }
        }
        pass++;
    }
}
```

是一种稳定的排序方法

直接选择排序

每趟选择一个最小元素，与第一个元素对调，对子列继续执行直到只剩下一个元素

元素的比较次数与初始排列**无关**，移动次数与初始排列**有关**

不稳定

锦标赛排序

类似于体育比赛中的淘汰赛，每次两两比较，排序码较小的加入到**优胜者**中， 对优胜者再进行两两比较……

比较可以被转化为**胜者树**

该排序方法较快，但使用了较多的附加存储

是一种**稳定排序法**

堆排序

建立**最大堆**进行排序

```
void siftDown(dataList<T> &L, const int start, const int m){
    int i = start;
    int j = 2 * i;
    Element<T> temp = L[i];
    while(j <= m){
        if(j < m && L[j] < L[j + 1]){
            j++;
        }
        if(temp >= L[j]){
            break;
        } else {
            L[i] = L[j];
            i = j;
            j = 2 * j + 1;
        }
    }
    L[i] = temp;
}

void HeapSort(dataList<T> &L){
    int i, n = L.length();
    for(i = (n - 2) / 2; i >= 0; i--){
        siftDown(L, i, n-1);
    }

    for(i = n - 1; i >= 0; i--){
        L.Swap(0,i);
        siftDown(L, 0, i-1);
    }
}
```

归并排序

归并排序针对多个有序表进行排序，将他们合成一个新的有序表

- 迭代的归并排序算法

  把初始元素序列划分为长度为1的有序子列，两两归并，再做两两归并……

```
//两路归并算法，L1.Vector(left, mid)与L1.Vector(mid + 1, right)是两个有序表
void merge(dataList<T> &L1, dataList<T> &L2, const int left, const int mid, const int right){
    int k, i, j;
    i = left;
    j = mid + 1;
    k = left;
    while(i <= mid && j <= right){
        if(L1[i] <= L1[j]){
            L2[k++] = L1[i++];
        }else{
            L2[k++] = L1[j++];
        }
```

```
    }
    while(i <= mid){
        L2[k++] = L1[i++];
    }
    while(j <= right){
        L2[k++] = L1[j++];
    }
}
```

总结

一般来讲，时间复杂度越小，需要的附加存储就越多，一般找不到时间和空间都最优的排序

排序的平均复杂度不可能低于 O(nlogn)

附：排序比较表

排序方法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	$O(n)$	$O(n^2)$	0	$O(n^2)$	√	$O(1)$	
折半插入排序	$O(n\log_2 n)$		0	$O(n^2)$	√	$O(1)$	
起泡排序	$O(n)$	$O(n^2)$	0	$O(n^2)$	√	$O(1)$	
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n^2)$	×	$O(\log_2 n)$	$O(n)$
直接选择排序	$O(n^2)$		0	$O(n)$	×	$O(1)$	
锦标赛排序	$O(n\log_2 n)$		$O(n\log_2 n)$		√	$O(n)$	
堆排序	$O(n\log_2 n)$		$O(n\log_2 n)$		×	$O(1)$	
归并排序	$O(n\log_2 n)$		$O(n\log_2 n)$		√	$O(n)$	

文件 外部排序 外部搜索

主存储器与外存储器

主存又叫内存，外存与内存相比，价格较低且有永久存储能力，缺点是访外存慢得多。

磁带

顺序存取的设备

应用中使用文件进行数据处理的基本单位叫做**逻辑记录**，简称为记录；在磁带上物理地存储的记录被称作**物理记录**

在使用磁带或磁盘存放逻辑记录时，常把若干个逻辑记录打包进行存放，称作**块化**

在磁带上读写信息所用的时间$t_{IO}=t_a+t_b$，即延时时间加上对一个块读写所用的时间