数据结构 与 算法

一、线性表

二、栈 和 队列

三、串

四、数组 和 广义表

五、树 和 二叉树

5.1 树的基本概念

1、结点的度（Degree）：结点的字数个数

2、数的度：数的所有结点中最大的度数

3、叶结点（Leaf）：度为0的结点

4、夫结点（Parent）：有子数的结点是其子树的根结点的夫结点

5、子结点（Child）：若A结点是B结点的夫结点，则称B结点是A结点的子结点

6、兄弟结点（Sibling）：具有同一父结点的各结点彼此是兄弟结点

7、路径和路径长度：从结点n1到nk的路径为一个结点序列n1,n2,...,nk，ni是ni+1的父结点。路径所包含边的个数为路径的长度。

8、祖先结点（Ancestor）：沿数根到某一结点路径上的所有结点都是这个结点的祖先结点

9、子孙结点（Descendant）：某一结点的子树中所以结点是这个结点的子孙

10、结点的层次（Level）：规定根结点在1层，其他任一结点的层数是其父结点的层数加1

11、树的深度（Depth）：数中所有结点中的最大层次是这棵树的深度

5.2 二叉树

5.2.1 定义与性质

1、一个有穷的结点集合，这个集合可以为空，若不为空，则它是由根结点和称为其左子树和右子树的两个不相交的二叉树组成

2、特殊二叉树：
（1）斜二叉树：相当于链表
（2）完美二叉树：也称满二叉树
（3）完全二叉树

3、重要性质：
（1）一个二叉树第i层的最大结点数为：2^(i-1),i>1
（2）深度为k的二叉树有最大结点总数为：2^k-1,k>=1
（3）对任何非空二叉树，若n0表示叶结点的个数、n2是度为2的非叶结点个数，那么两者满足关系n0=n2+1

5.2.2 存储结构

1、顺序存储结构：
（1）适用于完全完全二叉树
（2）非根结点（序号i>1）的父结点的序号是[i/2]
（3）结点（序号为i）的左孩子的结点的序号为2i（若2i<=n，则没有左孩子），右孩子的序号为2i+1

2、链表存储

```
typedef struct TreeNode *BinTree;
typedef BinTree Position;
struct TreeNode{
    ElementType Data;
    BinTree Left;
    BinTree Right;
}
```

5.2.3 遍历

先序、中序、后序遍历过程中经过的结点的路线是一样的，只是访问各个结点的时机不同，即cout的时机不同

非递归算法的思路是使用堆栈

遍历的核心问题：二维结构的线性化

需要一个存储结构保存暂时不访问的结点：堆栈、队列

1、先序遍历

（1）访问根结点
（2）先序遍历其左子树
（3）先序遍历其右子树

递归算法

```
void PreOrderTraversal(BinTree BT)
{
    if(BT != nullptr){
        cout<<BT->Data<<endl;
        PreOrderTraversal(BT->Left);
        PreOrderTraversal(BT->Right);
    }
}
```

非递归算法

```
void PreOrderTraversal(BinTree BT)
{
    BinTree p = BT;
    SqStack s(20);  //建立容量为20、元素类型为整型的空栈
    while( p || !s.StackEmpty() ){
        if(p){
            //遍历左子树
            cout<<p->Data<<endl;
            s.Push(p);
            p = p->Left;
        }else{
            //访问根结点，遍历右子树
            p = s.Pop();
            p = p->Right;
        }
    }
}
```

2、中序遍历

（1）中序遍历其左子树；
（2）访问根结点；
（3）中序遍历其右子树。

递归算法

```
void InOrderTraversal(BinTree BT)
{
    if(BT != nullptr){
        InOrderTraversal(BT->Left);
        cout<<BT->Data<<endl;
        InOrderTraversal(BT->Right);
    }
}
```

非递归算法

```
void InOrderTraversal(BinTree BT)
{
    BinTree p = BT;
    SqStack s(20);  //建立容量为20、元素类型为整型的空栈
    while( p || !s.StackEmpty() ){
        if(p){
```

```
            s.Push(p);
            p = p->Left;
        }else{
            p = s.Pop();
            cout<<p->Data<<endl;
            p = p->Right;
        }
    }
}
```

3、后序遍历

（1）后序遍历其左子树
（2）后序遍历其右子树
（3）访问根结点

递归算法

```
void InOrderTraversal(BinTree BT)
{
    if(BT != nullptr){
        InOrderTraversal(BT->Left);
        InOrderTraversal(BT->Right);
        cout<<BT->Data<<endl;
    }
}
```

非递归算法

```
struct BiTNode_1{
    BiTNode *bt;
    int tag;
}
void InOrderTraversal()
{
    BiTNode *p = bt;
    SqStack s(20);
    BiTNode_1 *temp;
    while( p || !s.StackEmpty() ){
        if(p){
            BiTNode_1 *t = new BiTNode_1;
            t->bt = p;
            t->tag = 1;
            s.Push(t);
            p = p->Left;
        }else{
            temp = s.Pop();
            if(temp->tag == 1){
                s.Push(temp);
                temp->tag-2;
                p = temp->bt->Right;
            }else{
                cout<<temp->bt->Data<<endl;
                p = nullptr;
            }
```

```
        }
    }
}
```

4、层序遍历

遍历从根结点开始，首先将根结点入队，然后开始执行循环：结点出队、访问该结点、其左右儿子入队

最后出来的结点序列是一层一层的

```
void LevelOrderTraversal(BinTree BT)
{
    Queue Q;
    BinTree T;
    if(!BT) return;
    Q = CreatQueue(MaxSize);
    AddQ(Q,BT); //根结点入队
    while( !IsEmptyQ(Q) ){
        T = DeleteQ(Q); //结点出队
        cout<<T->Data;  //访问该结点
        if(T->Left) AddQ(Q,T->Left);
        if(t->Right) AddQ(Q,T->Right);
    }
}
```

5、应用例子

求二叉树的高度

```
int PostOrderGetHeight(BinTree BT)
{
    int HL,HR,MaxH;
    if(BT){
        HL = PostOrderGetHeight(BT->Left);
        HR = PostOrderGetHeight(BT->Right);
        MaxH = (HL > HR)?HL:HR;
        return(MaxH + 1);
    }else{
        return 0;
    }
}
```

5.3 二叉搜索树

查找问题：

（1）静态查找和动态查找

（2）针对动态查找，数据如何组织

二叉搜索树满足以下性质：
（1）非空左子树的所有键值小于其根结点的键值
（2）非空右子树的所有键值大于其根结点的键值
（3）左、右子树都是二叉搜索树

1、查找操作

尾递归方法

```
BinTree Find( ElementType X, BinTree BST )
{
    if( !BST ) return nullptr;  //查找失败
    if( X > BST->Data ) return Find( X, BST->Right );    //在右子树中继续查找
    else if( X < BST->Data ) return Find( X, BST->Left );//在左子树中继续查找
    else return BST;   //查找成功，返回找到结点的地址
}
```

迭代函数方法

```
BinTree IterFind( ElementType X, BinTree BST )
{
    while(BST){
        if( X > BST->Data ) BST = BST->Right;
        else if( X < BST->Data ) BST = BST->Left;
        else return BST;
    }
    return nullptr;
}
```

查找的效率取决于树的高度

查找最大和最小元素：

（1）查找最小元素的递归函数

```
BinTree FindMin( BinTree BST )
{
    if(!BST) return nullptr;
    else if(!BST->Left) return BST;
    else return FindMin(BST->Left);
}
```

（2）查找最大元素的迭代函数

```
BinTree FindMax( BinTree BST )
{
    if(BST){
        while(BST->Right){
            BST = BST->Right;
        }
    }
    return BST;
}
```

2、插入

关键是要找到元素应该插入的位置，可以采用与Find类似的方法

```
BinTree Insert( ElementType X, BinTree BST )
{
    if( !BST ){
        //若原树为空，生成并返回一个结点的二叉搜索树
        BST = malloc(sizeof(struct TreeNode));
        BST->Data = X;
        BST->Left = BST->Right = nullptr;
    }else{
        if(X < BST->Data)    BST->Left = Insert(X,BST->Left);
        else if(X>BST->Data) BST->Right = Insert(X,BST->Right);
    }
    return BST
}
```

3、删除

考虑三种情况：
（1）要删除的是叶结点：直接删除，并再修改其父结点指针
（2）要删除的结点只有一个孩子结点：将其父结点的指针指向要删除结点的孩子结点
（3）要删除得结点有左、右两颗子树：用另一结点替代被删除结点：右子树的最小元素或左子树的最大元素

```
BinTree Delete(ElementType X,BinTree BST)
{
    BinTree Tmp;
    if(!BST) cout<<"要删除的元素未找到"<<endl;
    else if(X < BST->Data)  BST->Left = Delete(X,BST->Left);   //左子树递归删除
    else if(X < BST->Data)  BST->Right = Delete(X,BST->Right); //右子树递归删除
    else{  //找到要删除结点
        if(BST->Left && BST->Right){      //被删除结点右左右两个子结点
            Tmp = FindMin( BST->Right );  //在右子树中找最小元素填充删除结点
            BST->Data = Tmp->Data;
            BST->Right = Delete(BST->Data, BST->Right);//在删除结点的右子树中删除最小
元素
        }else{  //被删除结点有一个或无子结点
            Tmp = BST;
            if(!BST->Left)    BST = BST->Right; //有右孩子或无子结点
            else(!BST->Right) BST = BST->Left;  //有左孩子或无子结点
            free(Tmp);
        }
    }
    return BST;
}
```

5.4 平衡二叉树

1、搜索树结点不同插入次序，将导致不同的深度和平均查找长度ASL

2、平衡因子（Balance Factor，简称BF）：BF(T) = hL - hR

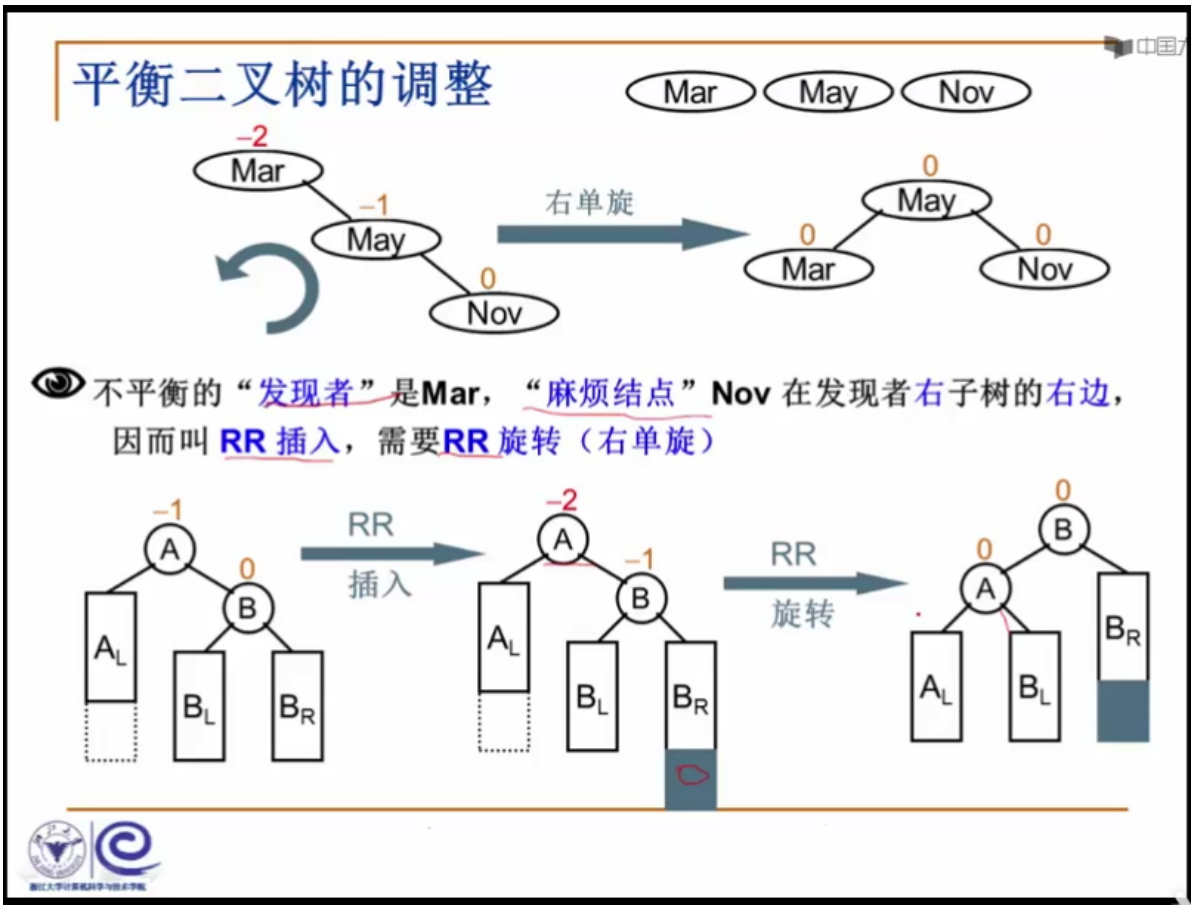
3、AVL树：空树 或 任一结点左、右子树高度差的绝对值不超过1

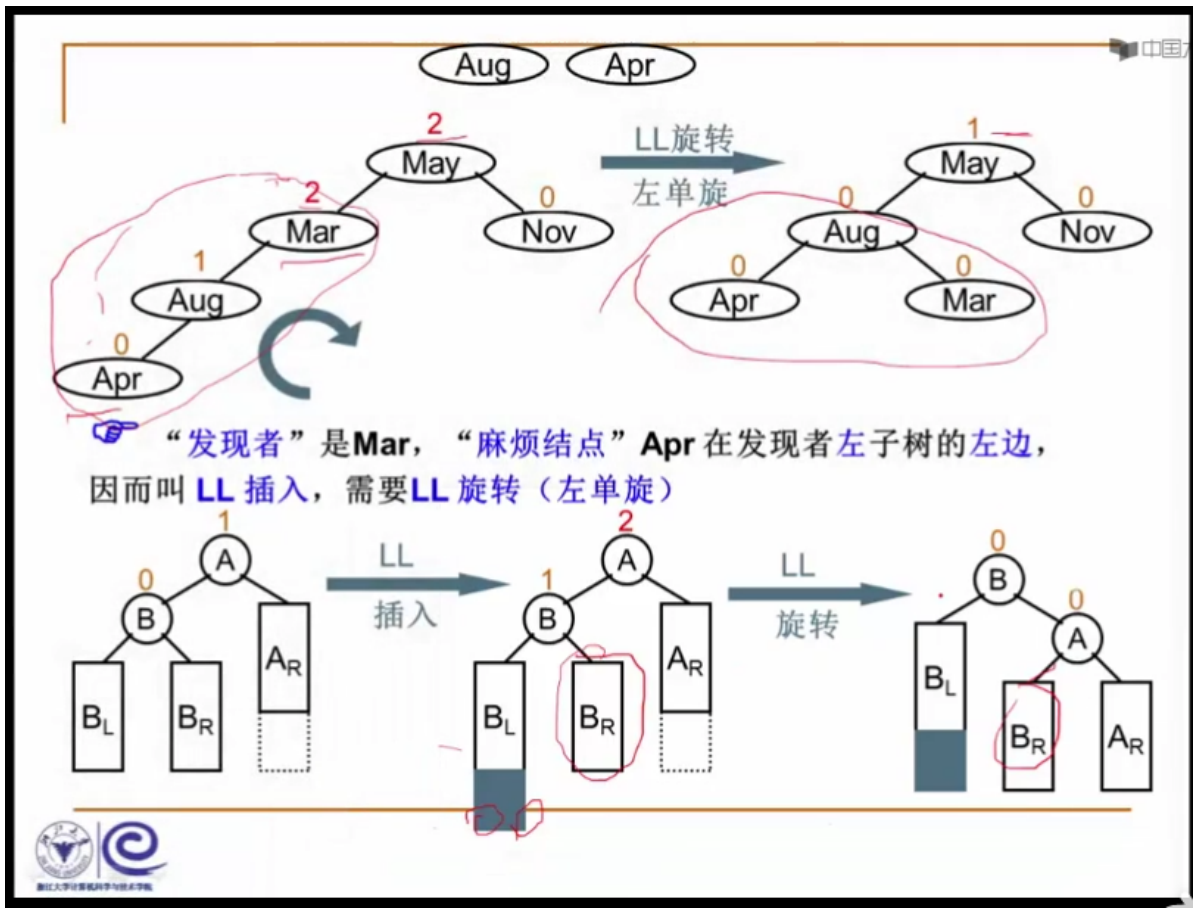
4、设nh是高度为h的平衡二叉树的最小节点数

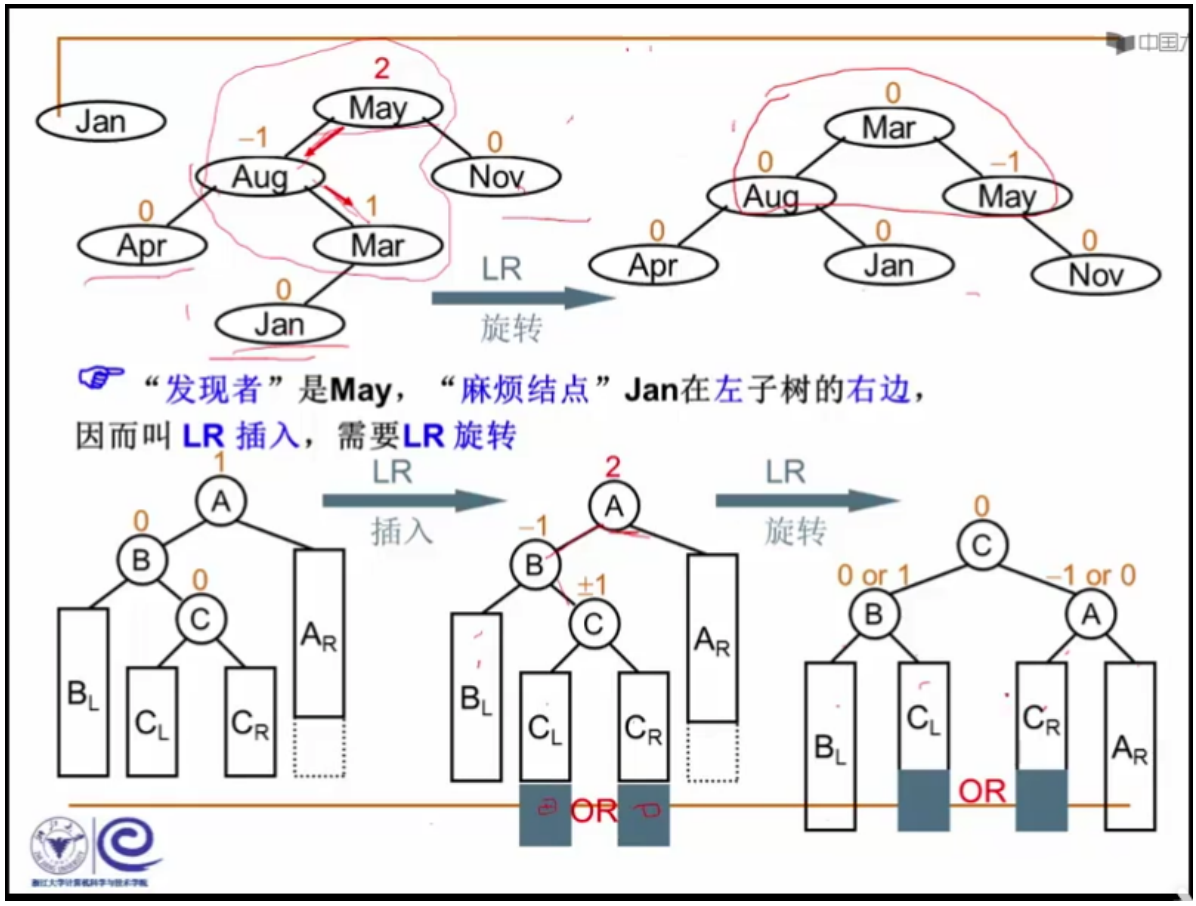
$$n_h = n_{h-1} + n_{h-2} + 1$$

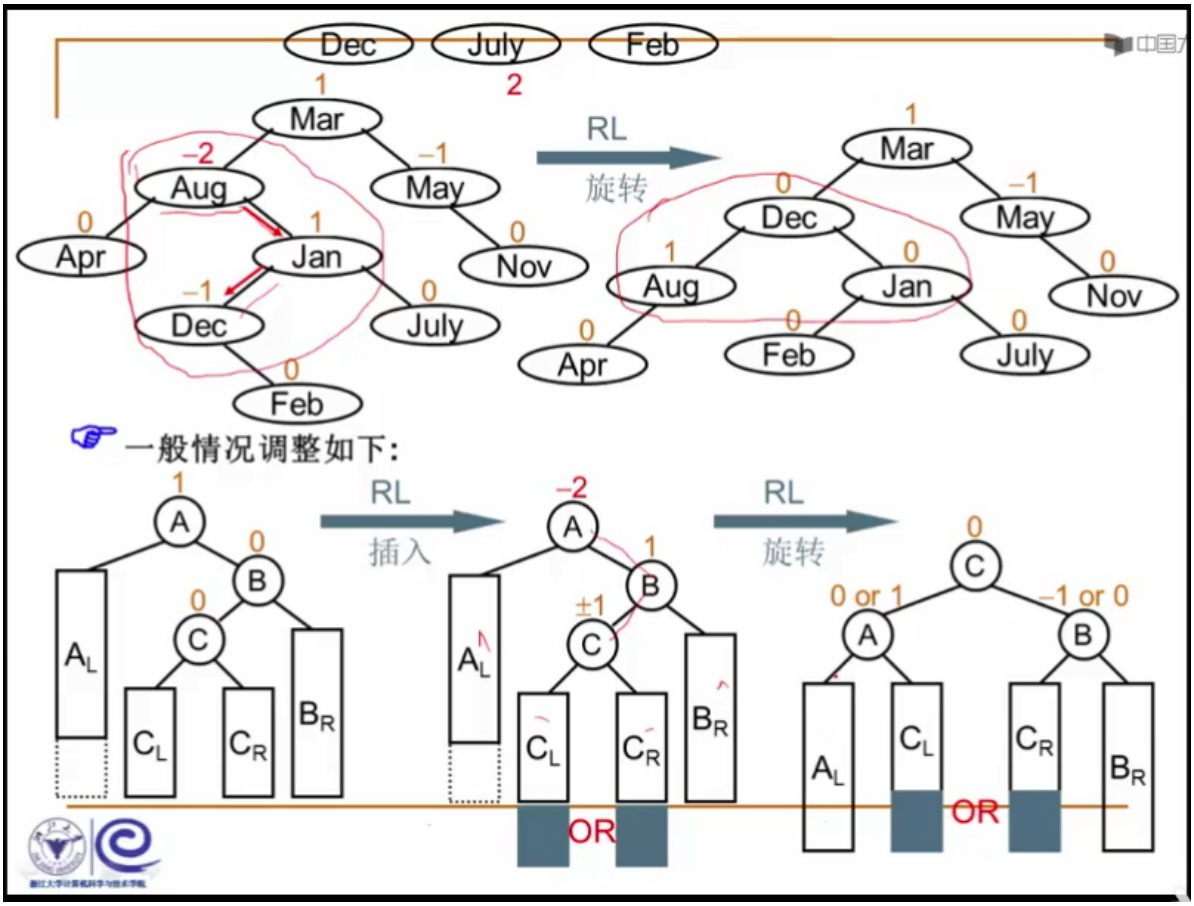
给定结点数为n的AVL树的最大高度为O(log2 n)

5、调整：




Aug
Apr
May
Mar
Aug
Apr
Nov
LL旋转
左单旋
May
Aug
Nov
Apr
Mar
“发现者”是Mar，“麻烦结点”Apr在发现者左子树的左边，
因而叫 LL 插入，需要LL 旋转（左单旋）
A
B
A_R
B_L
B_R
LL
插入
LL
旋转


Jan
May
Aug
Nov
Apr
Mar
Jan
LR
旋转
Mar
Aug
May
Apr
Jan
Nov
“发现者”是May，“麻烦结点”Jan在左子树的右边，
因而叫 LR 插入，需要LR 旋转
A
B
C
A_R
B_L
C_L
C_R
LR
插入
LR
旋转
±1
OR
0 or 1
−1 or 0
OR



5.5 堆

1、优先队列（Priority Queue）：特殊的队列，取出元素的顺序是依照元素的优先权（关键字）大小，而不是元素进入队列的先后顺序。

2、堆的两个特性：
（1）结构性：用数组表示的完全二叉树
（2）有序性：任一结点的关键字是其子树所有结点的最大值（或最小值）
每个结点的值都小于（大于）或等于其左右孩子节点的值，称为小（大）顶堆

3、最大堆的创建

```
typedef struct HeapStruct *MaxHeap;
struct HeapStruct{
    ElementType *Elements;     //存储堆元素的数组
    int Size;                  //堆当前元素个数
    int Capacity;              //堆的最大容量
}
MaxHeap Creat(int MaxSize)     //创建容量为MaxSize的空的最大堆
{
    MaxHeap H = malloc(sizeof(struct HeapStruct));
    H->Elemnets = malloc((MaxSize+1)*sizeof(ElementType));
    H->Size = 0;
    H->Capacity = Maxsize;
    H->Elememts[0] = MaxData;
    //定义“哨兵”为大于堆中所有可能元素的值，便于以后更快操作
    return H;
}
```

4、最大堆的插入

```
void Insert(MaxHeap H, ElementType item)
{
    //将元素item插入最大堆H，其中H->Elements[0]已经定义为哨兵
    int i;
    if(IsFull(H)){
        cout<<"最大堆已满"<<endl;
        return;
    }
    i = ++H->Size;    //i指向插入后堆中的最后一个元素的位置
    for(; H->Elements[i/2] < item; i/=2){
        //如果插入的元素比父结点要大则跟父结点交换位置
        H->Elemnets[i] = H->Elements[i/2];
    }
    H->Elements[i] = item;
}
```

5、最大堆的删除

```
//从最大堆H中取出键值为最大的元素，并删除
ElementType DeleteMax(MaxHeap H)
{
    int Parent,Child;
    ElementType MaxItemm, temp;
    if(IsEmpty(H)){
        cout<<"最大堆已为空"<<endl;
        return;
    }
    MaxItem = H->Elements[1];
    //用最大堆中最后一个元素从根结点开始从上过滤下层结点
    temp = H->Elements[H->Size--];
    for(Parent = 1; Parent*2 <= H->Size; Parent = Child){
        Child = Parent *2;
        if( (Child != H->Size) && (H->Elements[Child] < H->Elements[Child+1]) ){
            Child++;  //Child指向左右子结点的较大者
        }
        if( temp >= H->Elements[Parent] = H->Elements[Child] ) break;
        else H->Elements[Parent] = H->Elements[Child];
    }
    H->Elements[Parent] = temp;
    return MaxItem;
}
```

6、最大堆的建立

将已经存在的N个元素按最大堆的要求存放在一个一维数组中

方法一：通过插入操作，将N个元素一个个相继插入到一个初始为空的堆中去，其时间复杂度为O(NlogN)

方法二：在线性时间复杂度下建立最大堆：
（1）将N个元素按输入顺序存入，先满足完全二叉树的结构特性
（2）调整各结点位置，以满足最大堆的有序特性。----->主要是运用删除堆时候的思路
（3）从最后一个元素的父节点开始进行调整

5.6 哈夫曼树和哈夫曼编码

1、概念

（1）带权路径长度（WPL）：

$$WPL = \sum_{k=1}^{n} W_k * L_k$$

（2）WPL最小的树

2、哈夫曼树的构造

思路：每次把权值最小的两颗二叉树合并，使用最小堆进行操作效率高

时间复杂度：O(NlogN)

```
typedef struct TreeNode *HuffmanTree;
struct TreeNode{
    int Weight;
    HuffmanTree Left, Right;
}
HuffmanTree Huffman(MinHeap H)
{
    int i;
    HuffmanTree T;
    BuildMinHeap(H);
    for(i = 1; i < H->Size; i++){
        T = malloc(sizeof(struct TreeNode));
        T->Left = DeleteMin(H);
        T->right = DeleteMin(H);
        T->Weight = T->Left->Weight + T->Right->Weight;
        Insert(H, T);
    }
    T = DeleteMin(H);
    return T;
}
```

3、哈夫曼树的特点

（1）没有度为1的结点

（2）n个叶子结点的哈夫曼树共有2n-1个结点

（3）哈夫曼树的任意非叶结点的左右子树交换后仍是哈夫曼树

（4）对同一组权值，可能存在不同结构的两颗哈夫曼树

4、哈夫曼编码

（1）根据字符出现的频率来构造哈夫曼树

（2）在哈夫曼树上求叶子结点的编码

5.7 B-树

```
// B树的C++实现
#include<iostream>
using namespace std;

// B树节点
```

```
class BTreeNode
{
    int *keys;  // 关键字数组
    int t;      // 最小度 (定义关键字数量)
    BTreeNode **C; //子节点指针数组
    int n;     // 当前的关键字数量
    bool leaf; // 如果节点是叶子节点，则为true，否则为false
public:
    BTreeNode(int _t, bool _leaf);   // 构造函数

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in
    // child array C[].  The Child y must be full when this function is called
    void splitChild(int i, BTreeNode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {  root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {  if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {  return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;
```

```
    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
```

```
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now.  Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);

            // Change root
            root = s;
        }
        else  // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
```

```
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two.  See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];
```

```
    // Increment count of keys in this node
    n = n + 1;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constucted tree is ";
    t.traverse();

    int k = 6;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    k = 15;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    return 0;
}
```

5.8 红黑树

六、图

6.1 表示方法

1、邻接矩阵

G [N] [N]

问题：对于无向图，有一半的空间是被浪费的

解决方法：用一个长度为 N(N+1)/2的1维数组A存储 { G00, G10, G11,....., Gn-1 0, ... , G n-1 n-1 }
则Gij在A中对应的下标 (i*(i+1)/2 + j)

好处：直观、简单、好理解
方便检查任意一对顶点间是否存在边
方便找任一顶点的所有”邻接点“（有边直接相连的顶点）
方便计算任一顶点的”度“（从该顶点发出的边数为”出度“，指向该点的边数为”入度“）
---无向图：对应行或列非零元素的个数
---有向图：对应行非0元素的个数是”出度“；对应列非0元素的个数是”入度“

缺点：浪费空间---存稀疏图
浪费时间---统计稀疏图中一共有多少条边

```
#define MAX_VERTEX_NUM 20
const int infinty = INT_MAX;
struct ArcCell
{
    int adj;          //对无权图有1、0表示是否相邻，对带权图，则为权值类型
    char *info;       //该弧的相关信息
};

struct MGraph
{
    string vexs[MAX_VERTEX_NUM];                    //顶点表
    ArcCell arcs[MAX_VERTEX_NUM][MAX_VERTEX_NUM]; //邻接矩阵，即边表
    int vexnum;    //顶点数
    int arcnum;    //边数
    int kind;      //邻接矩阵存储的图的种类
};

class Graph
{
    MGraph m_Graph;
    bool visited[MAX_VERTEX_NUM];
public:
    Graph();
    ~Graph();
    int LocateVex(string u);
    bool CreateUDG();
};

int Graph::LocateVex(string u)
{
    for(int i = 0; i < MAX_VERTEX_NUM; i++){
        if(u == m_Graph.vexs[i])  return i;
    }
    return -1;
}

bool Graph::CreateUDG()
{
    int i,j;
    string v1, v2;
    cout<<"请输入无向图的顶点个数，边的个数：  "<<endl;
    cin>>m_Graph.vexnum>>m_Graph.arcnum;
    cout<<"请输入各个顶点："
    for(i = 0; i < m_Graph.vexnum; i++){  //构造顶点向量
        cin>>m_Graph.vexs[i];
    }
    for(i = 0; i < m_Graph.vexnum; i++){
        for(j = 0; j < m_Graph.vexnum; j++){
            m_Graph.arcs[i][j].adj = 0;
            m_Graph.arcs[i][j].info = false;
        }
    }
    for(i = 0; i < m_Graph.arcnum; i++){  //构造邻接矩阵
        cout<<"请输入一条边依附的两个顶点"<<endl;
```

```
        cin>>v1>>v2;
        int m = LocateVex(v1);
        int n = LocateVex(v2);
        m_Graph.arcs[m][n].adj = 1;
        m_Graph.arcs[n][m].adj = 1;
    }
    m_Graph.kind = 2;
    return true;
}
```

2、邻接表

G[N]为指针数组，对应矩阵每行一个链表，只存非0元素

特点：方便找任一顶点的所有"邻接点"
节约稀疏图的空间：需要N个头指针 + 2E个结点（每个结点至少2个域）
方便计算任一顶点的"度"?
---对无向图：是，但对有向图
---对有向图：只能计算"出度"；需要构造"逆邻接表"来计算"入度"
不方便检查任意一对顶点间是否存在边

```
struct ArcNode  //定义边结点
{
    int adjvex;
    ArcNode* next;
};

struct VertexNode //定义表结点
{
    int vertex;
    ArcNode* firstedge;
};
```

6.2 遍历

1、深度优先搜索（DFS）

是数的先序遍历的一种推广

```
void DFS( Vertex V )    //伪代码
{
    visited[ V ] = true;
    for( V的每个邻接点W ){
        if( !visited[ W ] ) DFS( W );
    }
}
void Graph::DFSTraversal(int v)
{
    cout<<m_Graph.vexs[v]<<endl;
    visited[v] = 1;
    for(int i = 0; i < m_Graph.vexnum; i++){
```

```
        if(m_Graph.arcs[v][i] == 1 && visited[i] == 0){
            DFSTraversal(i);
        }
    }
}
```

若有N个顶点、E条边，时间复杂度：
（1）邻接表存储：O(N+E)
（2）邻接矩阵存储：O(N^2)

2、广度优先搜索（BFS）

类似于数的层次遍历，使用的是队列

```
void BFS( Vertex V )
{
    visited[V] = true;
    Enqueue ( V,Q );
    while( !IsEmpty(Q) ){
        V = Dequeue( Q );
        for( V的每个邻接点 ){
            if( !visited[W] ){
                visited[W] = true;
                Enqueue( W, Q );
            }
        }
    }
}
```

若有N个顶点、E条边，时间复杂度：
（1）邻接表存储：O(N+E)
（2）邻接矩阵存储：O(N^2)

3、连通分量 和 重连通分量

关节点：当且仅当删去v以及依附于v的所有边之后，G将被分割成至少两个连通分量

重连通图：一个没有关节点的连通图； 图G的重连通分量事实上把G的边划分到互不相交的边的子集中

```
void ListComponents(Graph G)
{
    for(each V in G){
        if(!visited[V]){
            DFS(V);
        }
    }
}
```

6.3 最短路径问题

单源最短路径问题：从某固定源点出发，求其到所有其他顶点的最短路径
多源最短路径问题：求任意两顶点间的最短路径

1、无权图的单源最短算法

按照递增的顺序找出各个顶点的最短路

该方法的思路与BFS有相似的地方

```
void UnWeighted( Vertex S )
{
    Enqueue(S,Q);
    while(!IsEmpty(Q)){
        V = Dequeue(Q);
        for( V的每个邻接点W ){
            if(dist[W] == -1){  //dist[]初始化时为-1，代表还没有被访问
                dist[W] = dist[V] + 1;
                path[W] = V;  //记录路径上该顶点的上一个顶点
                Enqueue(W,Q);
            }
        }
    }
}
```

用邻接表下的时间复杂度为O(|N|+|E|)

2、有权图的单源最短路算法

不讨论有负值圈存在的图

按照递增的顺序找出到各个顶点的最短路

Dijkstra算法：
（1）令S = { 源点s + 已经确定了最短路径的顶点vi }
（2）对任一未收录的顶点v，定义 dist[v] 为 s 到 v 的最短路径长度，但该路径仅经过 S 中的顶点，即路径{ s->(vi属于S)->v }的最小长度
（3）若路径时按照递增的顺序生成的，则：
真正的最短路必须只经过S中的顶点
每次从未收录的顶点中选一个 dist 最小的收录（贪心算法）
增加一个v进入S，可能影响另外一个w的dist值：dist[w] = min{ dist[w], dist[v] + <v,w>的权重 }
（4）dist[]的初始化，与源点没有直接相连的点，设为正无穷；与源点直接相连的点，则设为其长度

```
void Dijkstra( Vertex V )
{
    while(1){
        V = 未收录顶点中dist最小者;
        if( 这样的V不存在 ) break;
        collected[V] = true;
        for( V的每个邻接点W ){
            if( collected[W] == false ){
                if( dist[V] + E<v,w> < dist[W] ){
                    dist[W] = dist[V] + E<v,w>;
                    path[W] = v;
                }
            }
```

```
        }
    }
}
```

时间复杂度取决于搜索未收录顶点的算法和路径更新方法

3、多源最短路算法

方法一：直接将单源最短路算法调用V遍：T=O(V^3+E+V)，对于稀疏图效果好
方法二：Floyd算法

```
void Graph::ShortestPath_Floyd(PathMatrix &P, DistanceMatrix &D)
//若P[v][w][u] = true，则u是从v到w当前球的最短路径上的顶点
{
    int u,v,w,i;
    for(v = 0; v<m_Graph.vexnum; v++){
        for(w = 0; w<m_Graph.vexnum; w++){
            D[v][w] = m_Graph.arcs[v][w].adj;
            for(u = 0; u<m_Graph.vexnum; u++){
                P[v][w][u] = false;
            }
            if(D[v][w]<infinty){
                P[v][w][v] = P[v][w][w] = true;
            }
        }
    }

    for(u=0; u<m_Graph.vexnum; u++){
        for(v=0; v<m_Graph.vexnum; v++){
            for(w=0; w<m_Graph.vexnum; w++){
                if(D[v][u] + D[u][w] < D[v][w]){
                    D[v][w] = D[v][u] + D[u][w];
                    for(i=0; i<m_Graph.vexnum; i++){
                        P[v][w][i] = P[v][u][i] || P[u][w][i];
                    }
                }
            }
        }
    }
}
```

6.4 最小生成树

1、概念：
（1）是一棵树：无回路，V个顶点一定有V-1条边
（2）是生成树：包含全部顶点，V-1条边都在图里，向生成树中任加一条边都一定构成回路
（3）边的权重和最小

2、贪心算法：
（1）贪：每一步都要最好
（2）好：权重最小的边
（3）需要约束：只能用图里有的边；只能正好用掉V-1条边；不能有回路

3、并查集

4、Prim算法
与最短路径中的dijkstra有相似之处
由小变大

```
void Prim()
{
    MST = {s};
    while(1){
        V = 未收录顶点中dist最小者;
        if( 这样的V不存在 ) break;
        将V收录进MST：dist[V] = 0;
        for( V的每个邻接点W ){
            if( dist[W] != 0 ){
                if( E(V,W) < dist[W] ){
                    dist[W] = E(V,W);
                    parent[W] = V;
                }
            }
        }
    }
    if( MST中收到的顶点不到V个 ) Error();
}

// dist[]的初始化：dist[V] = E(s,V)或正无穷
// parent[s] = -1;
```

5、Kruskal算法
将森林合并成树

```
void Kruskal(Graph G)
{
    MST = {};
    while( MST中不到V-1条边 && E中还有边){
        从E中取一条权重最小的边E(V,W);    //最小堆实现
        将E(V,W)从E中删除;
        if( E(V,W)不在MST中构成回路 ) 将E(V,W)加入MST; //使用并查集方法
        else 彻底无视E(V,W);
    }
    if( MST中不到V-1条边 ) Error();
}
```

6.5 拓扑排序

1、概念
（1）拓扑序：如果图中从V到W有一条有向路径，则V一定排在W之前。满足此条件的顶点序列称为一个拓扑序
（2）拓扑排序：获得一个拓扑序的过程
（3）AOV（Activity On Vertex）如果有合理的拓扑序，则必定是有向无环图DAG

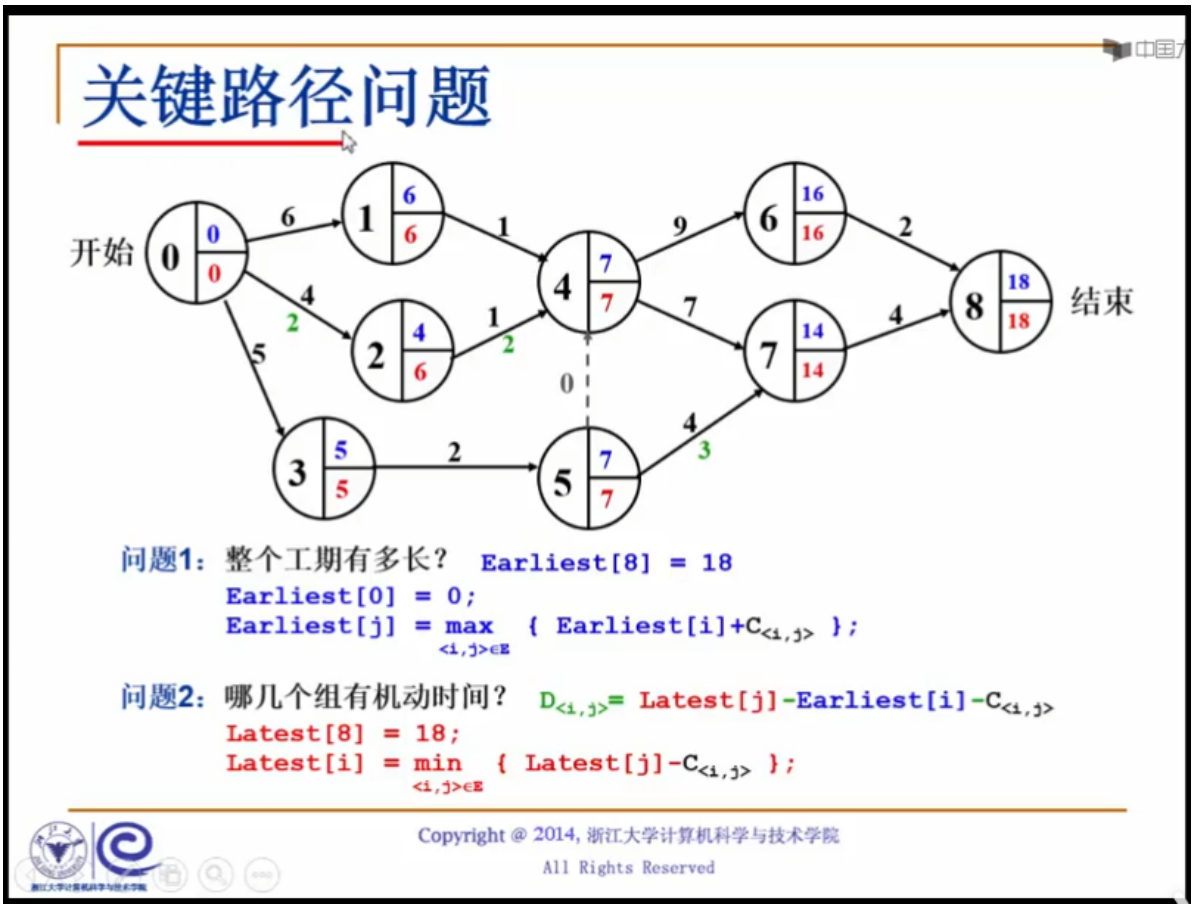
2、算法

```
void TopSort()
{
```

```
    for( 图中每个顶点V ){
        if( Indegree[V] == 0 )  Enqueue( V,Q );
    }
    while( !IsEmpty(Q)){
        V = Dequeue(Q);
        输出V，或者记录V的输出序号;
        cnt++
        for( V的每个邻接点W ){
            if( --Indegree[W] == 0) Enqueue(W,Q);
        }
    }
    if(cnt != v) Error("图中有回路");
}
```

6.6 关键路径

1、AOE（Activity On Edge）网络
（1） 由绝对不允许延误的活动组成的路径
（2）一般用于安排项目的工序



七、查找

7.1 哈希表查找

7.1.1 概念

1、一些概念
（1）已知的几种查找方法：顺序查找O(N)、二分查找(静态查找)O(log2N)、二叉搜索树O(h)或平衡二叉树O(log2N)
（2）查找的本质：已知对象找位置：
有序安排对象：全序、半序
直接"算出"对象位置：散列
（3）哈希查找法的两项基本工作：
计算位置：构造哈希函数确定关键词存储位置；
解决冲突：应用某种策略解决多个关键词位置相同的问题
时间复杂度几乎是常量：O(1)，即查找时间与问题规模无关

2、哈希表
（1）基本思路：构造一个大小确定的表，根据哈希函数，把数据对象放进去，问题在于如何解决冲突问题
（2）装填因子：设散列表空间大小为m，填入表中元素个数是n，则称 a = n/m 为散列表的装填因子
（3）思路总结：以关键字key为自变量，通过一个确定的函数h计算出对应的函数值h(key)，作为数据对象的存储地址

7.1.2 哈希函数的构造方法

1、一个好的哈希函数一般应考虑的两个因素：
（1）计算简单，以便提高转换速度；
（2）关键词对应的地址空间分布均匀，以尽量减少冲突

2、数字关键词的散列函数构造：
（1）直接定址法：取关键词的某个线性函数值为哈希地址，即h(key) = a*key+b
（2）取余数法：h(key) = key mod p
（3）数字分析法：分析数字关键字在各位上的变化情况，取比较随机的位作为哈希地址
（4）折叠法：把关键词分割成位数相同的几个部分，然后折叠
（5）平方取中法

3、字符关键词的哈希函数构造
（1）简单---ASCII码加和法：h(key) = (求和key[i]) mod TableSize ----冲突严重
（2）改进---前三个字符移位法：h(key) = (key[0]x27^2 + key[1]x27 + key[2]) mod TableSize
（3）好的---移位法：设计关键词的所有n个字符，并且分布得很好：

$$h(hey) = (\sum_{i=0}^{n-1} key[n-i-1] * 32^i) mod(TableSize)$$

```
Index Hash(const char *Key, int TableSize)
{
    unsigned int h = 0;
    while(*key != '\0')
        h = ( h<<5 ) + *Key++;
    return h%TableSize;
}
```

7.1.3 冲突处理方法

1、开放定址法

（1）若发生了第i次冲突，试探的下一个地址将增加di，基本公式：

$$h_i(key) = (h(key) + d_i) mod(TableSize)(1 <= i < TableSize)$$

（2）di决定了不同的解决冲突的方案：线性探测、平方探测、双哈希

（3）平方探测下的实现：

```
typedef struct HashTbl *HashTable;
struct HashTbl{
    int TableSize;
    Cell *TheCells;
}H;
HashTable Initialize(int TableSize)
{
    HashTable H;
    int i;
    if(TableSize < MinTableSize){
        Error("哈希表太小");
        return nullptr;
    }
    /*分配哈希表*/
    H = (HashTable)malloc(sizeof(struct HashTbl));
    if(H==nullptr)
        FatalError("空间溢出");
    H->TableSize = NextPrime(TableSize);
    /*分配哈希表cells*/
    H->TheCells = (Cell*)malloc(sizeof(Cell)*H->TableSize);
    if(H->TheCells == nullptr)
        FatalError("空间溢出");
    for(i = 0l i<H->TableSize; i++)
        H->TheCells[i].Info = Empty;
    return H;
}
Position Find(int key, HashTable H)
{
    Position CurrentPos, NewPos;
    int CNum = 0; //记录冲突次数
    NewPos = CurrentPos = Hash(Key,H->TableSize);
    while(H->TheCells[NewPos].Info != Empty && H->TheCells[NewPos].Element !=
key){
        if(++CNum % 2){  //判断冲突的奇偶次
            NewPos = CurrentPos + (CNum+1)/2*(CNum+1)/2;
            while(NewPos>=H->TableSize)
                NewPos -= H->TableSize;
        }else{
            NewPos = CurrentPos - CNum/2 * CNum/2;
            while(NewPos<0)
                NewPos += H->TableSize;
        }
    }
    return NewPos;
}
void Insert(int key,HashTable H)
{
```

```
    Position Pos = Find(key,H);
    if(H->TheCells[Pos].Info != Legitimate){
        /*确认在此插入*/
        H->TheCells[Pos].Info = Legitimete;
        H->TheCells[Pos].Element = key;
        /*字符串类型的关键词需要strcpy函数*/
    }
}

/* 在开放地址哈希表中，删除操作要很小心。通常只能懒惰删除，即需要增加一个”删除标记（deleted）“，而并不是真正删除它。以便查找时不会”断链“，其空间可以在下次插入时重用 */
```

（4）再哈希：
当哈希表元素太多（即装填因此太大）时，查找效率会下降
当装填因子过大时，解决的方法是加倍扩大散列表，这个过程叫做”再哈希“

2、分离链接法

（1）将相应位置上冲突的所有关键词存储在同一个单链表中
（2）实现如下：

```
typedef struct ListNode *Position,*List;
struct ListNode{
    ElementType Element;
    Position Next;
}
typedef struct HashTbl *HashTable;
struct HashTbl{
    int TableSize;
    List TheLists;
};
Position Find(ElementType Key,HashTable H)
{
    Position P;
    int Pos = Hash(Key,H->TableSize);
    P = H->TheLists[Pos].Next;
    while(P != nullptr && strcmp(P->Element,Key))
        p = p->Next;
    return P;
}
```

7.1.4 哈希表的性能分析

1、哈希表查找性能分析：
（1）成功平均查找长度(ASLs)：查找表中关键词的平均查找比较次数（其冲突次数加1）
不成功平均查找长度(ASLu)：不在哈希表中的关键词的平均查找次数，一般方法：将不在哈希表中的关键词分若干类
（2）关键词的比较次数，取决于产生冲突的多少，影响产生冲突多少有以下三个因素：
哈希函数是否均匀；处理冲突的方法；散列表的填充因子

2、各种冲突处理方法下的性能：

（1）开放地址法：
哈希表是一个数组，存储效率高，随机查找；但又聚集现象

1. 线性探测法的查找性能

可以证明，线性探测法的期望探测次数 满足下列公式：

$$p=\begin{cases}\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] & \text{（对插入和不成功查找而言）}\\ \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) & \text{（对成功查找而言）}\end{cases}$$

当α= 0.5时，

- 插入操作和不成功查找的期望 ASLu = 0.5*(1+1/(1-0.5) ^{2}) = 2.5 次
- 成功查找的期望 ASLs = 0.5*(1+1/(1-0.5)) = 1.5次

H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12
key	11	30		47				7	29	9	84	54	20
冲突次数	0	6		0				0	1	0	3	1	3

α= 9/13=0.69，于是

期望 ASLu = 0.5*(1+1/(1-0.69) ^{2}) = 5.70次

期望 ASLs = 0.5*(1+1/(1-0.69)) = 2.11次（实际计算ASLs =2.56）

2. 平方探测法和双散列探测法的查找性能

可以证明，平方探测法和双散列探测法探测次数 满足下列公式：

$$p=\begin{cases}\frac{1}{1-\alpha} & \text{（对插入和不成功查找而言）}\\ -\frac{1}{\alpha}\ln(1-\alpha) & \text{（对成功查找而言）}\end{cases}$$

当α= 0.5时，

- 插入操作和不成功查找的期望 ASLu = 1/(1-0.5) = 2 次
- 成功查找的期望 ASLs = -1/0.5* ln(1-0.5) ≈ 1.39 次

H(key)	0	1	2	3	4	5	6	7	8	9	10
key	11	30	20	47			84	7	29	9	54
冲突次数	0	3	3	0			2	0	1	0	0

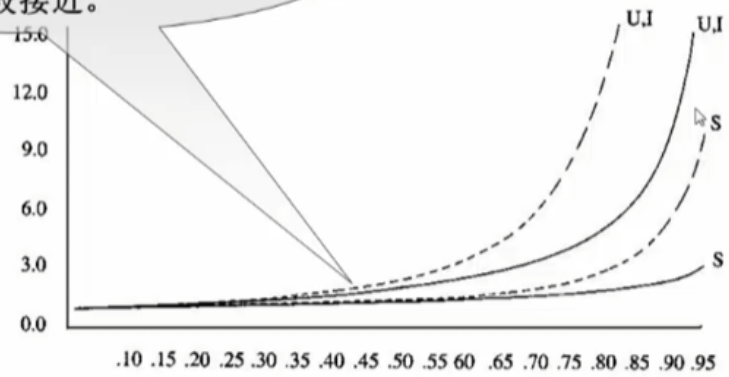
α= 9/11=0.82，于是

期望 ASLu = 1/(1-0.82) ≈ 5.56次

期望 ASLs = -1/0.5* ln(1-0.5) ≈ 2.09次（例中ASLs =2）。

❖ 期望探测次数与装填因子α的关系。

当装填因子α< 0.5的时候，各种探测法的期望探测次数都不大，也比较接近。



线性探测法（虚线）、双散列探测法（实线）
U表示不成功查找，I表示插入，S表示成功查找

（2）分离链法：

哈希表是顺序存储和链式存储的结合，链表部分的存储效率和查找效率都比较低。

3. 分离链接法的查找性能

所有地址链表的平均长度定义成装填因子α，α有可能超过1。
不难证明：其期望探测次数 p为：

$$p = \begin{cases} \alpha + e^{-\alpha} & \text{（对插入和不成功查找而言）} \\ 1 + \dfrac{\alpha}{2} & \text{（对成功查找而言）} \end{cases}$$

当α = 1时，

- 插入操作和不成功查找的期望 ASLu = $1+e^{-1}$ = 1.37 次，
- 成功查找的期望 ASLs = 1+1/2 = 1.5 次。

➢ 前面例子14个元素分布在11个单链表中，所以α= 14/11≈1.27， 故
期望 ASLu = $1.27+e^{-1.27}$ ≈ 1.55次
期望 ASLs = 1+1.27/2 ≈ 1.64次（例中ASLs =1.36）。

3、选择合适的h(key)，哈希法的查找效率期望是常数O(1)，它几乎与关键字的空间的大小n无关！也适合于关键字直接比较计算量大的问题

4、它是以较小的装填因子为前提。因此哈希方法是一个以空间换时间的方法

5、哈希方法的存储堆关键字是随机的，不方便顺序查找关键字，也不适合范围查找，或最大值最小值查找

7.2 顺序查找

1、查找步骤
（1）将所要查找的值key放入数组的第一个存储单元（即下标为0的单元）
（2）从数组最后一个数据元素开始，向前一个一个比较记录是否等于key，若相等，则返回该记录所在的数组下标；若扫描完所有记录都没有与key相等的记录，则返回0
（3）若返回值>0，则查找成功，否则查找失败

2、实现

```
int Search(int key)
{
    for(int i = st.length-1; i>=0; i--){
        if(st.elem[i]==key){
            cout<<"查找成功"<<i+1<<"位置上"<<endl;
            return i+1;
        }
    }
    cout<<"未找到"<<endl;
    return 0;
}
```

3、算法分析
（1）成功平均查找长度为：(n+1)/2
（2）当n很大时，查找效率非常低，为了提高查找效率，查找表需要根据 ”查找概率越高，比较次数越低“ 的原则进行设计

7.3 有序表的查找

1、核心思路：二分查找，前提田间是查找表中必须是采用顺序存储结构的有序表

2、具体步骤：
（1）表长n，下界low，上界high，中间点mid，给定的待查值key
（2）设置初始区间指针：下界指针low = 1，上界指针high = n，执行（3）
（3）若low > high，查找失败，返回查找失败信息；否则令m=[(low+high)/2]取下整数，并执行下面的操作：
---> 若待查值 key 小于 st.elm[mid].key ，则在m的左半区进行查找，令 high = m-1；执行（3）
---> 若待查值 key 大于 st.elm[mid].key ，则在m的右半区进行查找，令 high = m+1；执行（3）
---> 若待查值 key 等于 st.elm[mid].key ，则查找成功，返回记录在表中的位置

3、实现如下：

```
int Search(int key)
{
    int low=0, high=st.length-1;
    int mid;
    while(low<=high){
        mid = (high-low)/2;
        if(st.elem[mid] == key){
            cout<<"查找成功，处于第"<<mid+1<<"位置上"<<endl;
            return mid + 1;
        }
        else if(st.elem[mid]>key) high = mid - 1;
```

```
            else low = mid +1;
        }
        cout<<"查找失败"<<endl;
        return 0;
    }
```

4、算法分析
（1）相当于在一个搜索二叉树上进行搜索，其比较次数即为该结点所在树中的层数，其查找成功时进行关键字比较的次数至多为[log2(n+1)]+1
（2）假设有序表的长度n=2^h-1，则描述二分查找的判定树的深度为h的满二叉树。假设表中每个记录的查找概率相等，则平均查找长度为：

$$ASL = \sum_{i=1}^{n} P_i C_i = \frac{n+1}{n} log_2(n+1) - 1 \approx log_2(n+1) - 1$$

（3）二分查找的效率通常比顺序查找要高，但这方法有使用的条件作为局限

7.4 分块查找

1、概念：
（1）又称为索引顺序查找
（2）核心也是一种二分查找，而是把无序查找表分成若干个有序的查找表块

2、步骤：
（1）对于待查值key，在索引表中按某种查找算法将key与各个子表的最大关键字ki，kj进行比较。若ki<=key<kj，则可以可能在kj所对应的子表中。
（2）在（1）所找到的子表中进行顺序查找。若找到关键字与key相等的记录，则查找成功，返回该记录所在的位置；否则返回失败

3、算法分析
（1）在进行分块查找时，通常将长度为n的表均匀地分为b块，每块又m个记录，因此有b = [n/m]取上整数
（2）若假设用顺序查找来确定块的位置，平均查找长度为：ASL = 1/2 * (n/m +m) + 1
（3）若采用二分查找来确定块，则 ASL = log2(n/m+1)+m/2

八、内部排序

8.1 简单排序

1、冒泡排序

```
void Bubble_Sort(int* A, int N)
{
    for(int P = N-1; P>=0; P--){
        flag = 0;
        for(int i = 0; i<p; i++){
            if( A[i] > A[i+1] ){
                Swag(A[i],A[i+1]);
                flag = 1;
            }
        }
        if(flag == 0) break;
    }
}
```

时间复杂度：最坏时：O(n^2)

2、插入排序

```
void Insertion_Sort(int* A, int N)
{
    for (int P = 1; P < N; P++) {
        int Tmp = A[P];
        int i;
        for (i = P; i > 0 && A[i - 1] > Tmp; i--) {
            A[i] = A[i - 1];
        }
        A[i] = Tmp;
    }
}
```

从第二个开始抽牌，往前插

时间复杂度：最坏时：O(n^2)

3、时间复杂度下界

（1）逆序对：对于下标i<j，如果A[i]>A[j]，则称（i，j）是一对逆序对
（2）交换两个相邻元素正好消去一个逆序对
（3）任意N个不同元素组成的序列平均具有N(N-1)/4个逆序对
（4）任何仅以交换相邻两元素来排序的算法，其平均时间复杂度为OMIGA(N^2)
（5）要提高效率，我们必须：每次消去不止一个逆序对；每次交换相隔较远的两个元素

8.2 希尔排序

1、基本思路：
（1）定义增量序列DM>DM-1>...D1=1
（2）对每个Dk进行"Dk-间隔"排序

2、原始希尔排序：DM=[N/2], Dk=[Dk+1 / 2]

```
void Shell_Sort(ElementType *a, int n)
{
    for(int d = n/2; d>0; d/=2){
        for(int p = d; p<n; p++){
            int tmp = a[p];
            int i;
            for(i=p; i>=d && a[i-d]>tmp; i-=d){
                a[i]=a[i-d];
            }
        }
    }
}
```

时间复杂度：最坏时：O(n^2)

增量元素不互质，则小增量可能根本不起作用

3、Hibbard序列：

Dk = 2^k -1 相邻元素不互质

时间复杂度：最坏时：O(n^3/2)

8.3 堆排序

1、选择排序

```
void Selection_Sort(int *a, int n)
{
    for(int i = 0; i<n; i++){
        int MinPos = ScanForMin(a,i,n-1);
        Swap(a[i],a[MinPos]);
    }
}
```

关键在于如何快速找到最小值

2、堆排序

思路：将一个二叉树调整为一个最大堆，然后将根结点和最后一个结点互换，然后把最后一个结点拿掉并输出，对剩下的树进行最大堆调整，最后重复，直到没有了结点

```
void Heap_Sort(int *a,int n)
{
    for(int i = n/2; i>=0; i--){//建立堆
        PercDown(a,i,n);//调整
    }
    for(int i = n-1; i>0; i--){
        Swap(a[0],a[i]);
        PercDown(a,0,i);
    }
}
```

定理：堆排序处理N个不同元素的随机排序的平均比较次数是：2NlogN-O(NloglogN)

虽然堆排序给出最佳平均时间复杂度，但实际效果不如用Sedgewick增量序列的希尔排序

8.4 归并排序

1、核心思路：有序子列的归并

```
void Merge(int *a,int *Tmpa, int l,int r,int rend)
{
    int lend = r-1; //左边终点位置，假设左右两列挨着
    int tmp = l;    //存放结果的数组的初始位置
    int NumElements = rend - l + 1;
    while(l<=lend && r<=rend){
        if(a[l]<=a[r]) tmpa[tmp++] = a[l++];
        else tmpa[tmp++] = a[r++];
    }
    while(l<=lend){
        tmpa[tmp++] = a[l++];
    }
    while(r<=rend){
        tmpa[tmp++] = a[r++];
    }
    for(int i=0; i<NumElements; i++,rend--){
        a[rend]=tmpa[rend];
    }
}
```

2、递归算法：分而治之

```
void MSort(int *a,int *Tmpa, int l,int rend)
{
    int center;
    if(l<rend){
        center = (l + rend)/2;
        MSort(a,tmpa,l,center);
        MSort(a,tmpa,center+1;rend);
        Merge(a,tmpa,l,center+1,rend);
    }
}
```

统一函数接口后

```
void Merge_Sort(int *a,int n)
{
    int *tmpa = new int[n*sizeof(int)];
    if(tmpa != nullptr){
        MSort(a,tmpa,0,n-1);
        delete[] tmpa;
    }
    else Error("空间不足");
}
```

3、非递归算法

```
void Merge_Pass(int *a,int *tmpa,int n,int length)
{
    for(int i=0; i<=n-2*length; i += 2*length){
        Merge(a,tmpa,i,i+length, i+2*length-1);
```

```
    }
    if(i+length<n) Merge(a,tmpa,i,i+length,n-1);
    else for(int j=i; j<n;j++) tmpa[j] = a[j];
}
void Merge_Sort(int *a,int n)
{
    int length = 1;
    int *tmpa = new int[n*sizeof(int)];
    if(tmpa != nullptr){
        while(length<n){
            Merge_Pass(a,tmpa,n,length);
            length *=2;
            Merge_Pass(tmpa,a,n,length);
            length *=2;
        }
        delete[] tmpa;
    }
    else Error("空间不足");
}
```

8.5 快速排序

1、选主元

取头、中、尾的中位数

```
int Median3(int *a, int left, int right)
{
    int Center = (left + right)/2;
    if(a[left]>a[Center])  Swap(a[left], a[Center]);
    if(a[left]>a[right])   Swap(a[left], a[Center]);
    if(a[Center]>a[right]) Swap(a[center], a[Center]);
    Swap(a[Center], a[right-1]);
    return a[right-1];
}
```

2、子集划分

双指针的方法

3、小规模数据的处理：
（1）快速排序的问题：用递归；对小规模的数据可能还不如插入排序快
（2）解决方案：当递归的数据规模充分小，则停止递归，直接调用简单排序（例如插入排序）；再程序中定义一个Cut_Off的阈值

4、算法实现

```
void Quicksort(int *a, int l, int r)
{
    if(Cut_Off <= r-l){
        int Pivot = Median3(a,l,r);
        int i = l;
        int j = r-1;
        for( ; ; ){
            while(a[++i]<Pivot){}
```

```
            while(a[--j]<Pivot){}
            if(i<j)  Swap(a[i],a[j]);
            else break;
        }
        Swap(a[i],a[r-1]);
        Quick_Sort(a,l,i-1);
        Quick_Sort(a,i+1,r);
    }
    else Insertion_Sort(a+l,r-l+1);
}
void Quick_Sort(int *a, int N)
{
    Quicksort(a,0,N-1);
}
```

8.6 表排序

1、应用场景：存储数据的结构体复杂，移动的时候耗费时间，因此做交换位置的时间不能忽略

2、采用的方法：间接排序---定义一个指针数组作为"表"，根据表和key值来进行排序

3、定理：N个数字的排列由若干个独立的环组成

8.7 基数排序

1、桶排序

数值范围不大的情况下

```
void Bucket_Sort(int *a, int n)
{
    count[] 初始化一个存放指针的数组，下标为成绩;
    while(读入一个学生的成绩 grade ){
        将该生插入count[grade]链表;
    }
    for(int i=0; i<M; i++){
        if( count[i] ){
            输出整个count[i]链表;
        }
    }
}
```

2、基数排序

（1）根据排序的数字的基数来创建相应长度的桶，按照次位优先

（2）见下图

3、多关键字的排序

（1）应用例子：扑克牌排序：花色，面值

（2）先用”次位优先“排序，根据面值建立13个桶；然后将结果合并，再为花色建立4个桶，最后按顺序输出就可

8.8 排序算法的比较

排序方法	平均时间复杂度	最坏情况下时间复杂度	额外空间复杂度	稳定性
简单选择排序	O(N^2)	O(N^2)	O(1)	N
冒泡排序	O(N^2)	O(N^2)	O(1)	Y
直接插入排序	O(N^2)	O(N^2)	O(1)	Y
希尔排序	O(N^d)	O(N^2)	O(1)	N
堆排序	O(NlogN)	O(NlogN)	O(1)	N
快速排序	O(NlogN)	O(N^2)	O(logN)	N
归并排序	O(NlogN)	O(NlogN)	O(N)	Y
基数排序	O(P(N+B))	O(P(N+B))	O(N+B)	Y

九、算法设计 与 分析

9.4 动态规划

1、概念

（1）基本思想：将待求问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解

（2）问题特点：经过分割得到的子问题往往不是独立的。希望能保存已解决的子问题的答案，在需要时找出已求得的答案，避免大量重复计算

（3）总结：将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。但在递增生成子集过程中，力图朝最优方向进行而且也不回溯

2、步骤

（1）找出最优解的性质，并刻画其结构特征；

（2）递归定义最优值；

（3）自底向上的方式计算得出最有值

（4）根据计算最优值时得到得信息构造最优解

3、子问题重叠性