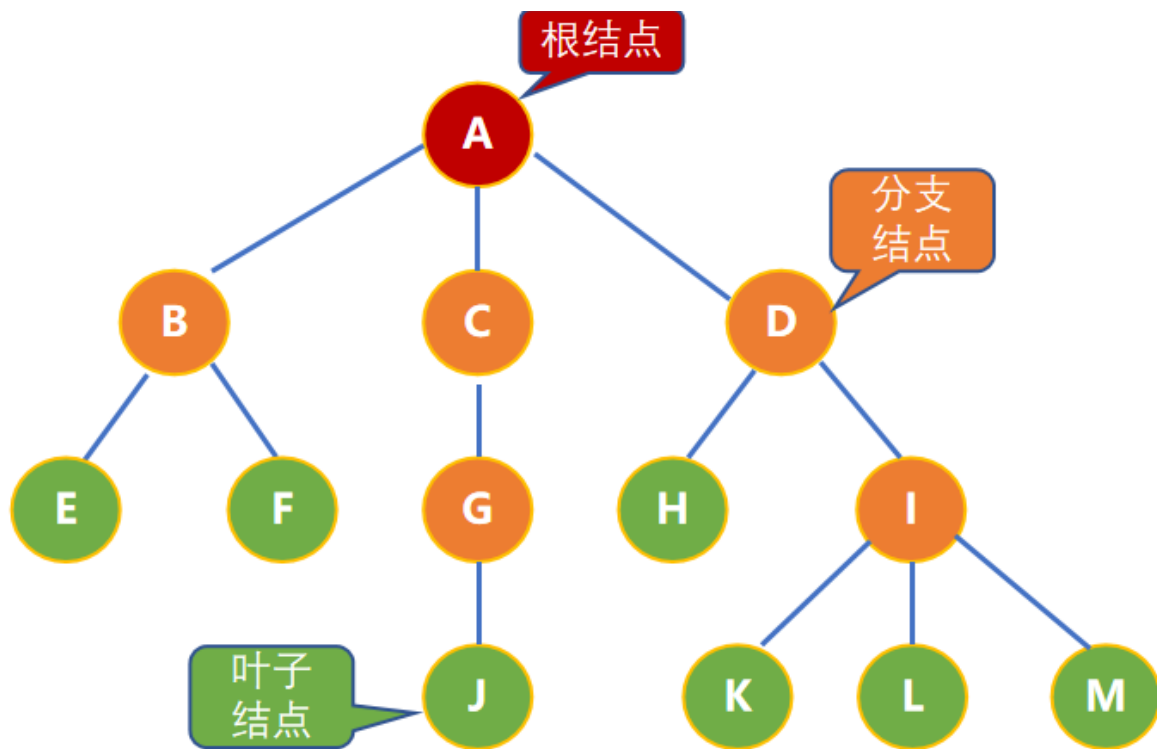


Para.6 树与二叉树

基本概念



空树：没有结点的树

非空树：有且仅有一个根结点

叶子结点：没有后继结点；分支结点：有后继结点

除了根节点之外，任何一个结点都有且仅有一个前驱，每个结点可以有0/多个后继。

结点的度：树中一个结点的子树个数

树的度：树中各结点度的最大值

m 叉树：度最大为 m 的树

树中的每个结点都处在一个层次上。结点的层次从树根开始定义，根结点为第1层，它的孩子结点为第2层，以此类推。

树中结点的最大层次称为树的高度（或树的深度）

有序树:在树T中,如果各子树T之间是有先后次序的,则称为有序树。

ADT

数据对象D：一个集合，该集合中的所有元素具有相同的特性。

数据关系R：若D为空集，则为空树。若D中仅含有一个数据元素，则R为空集；

否则 $R=\{H\}$ ，H是如下的二元关系：

- 1) 在D中存在唯一的称为根的数据元素 $root$ ，它在关系H下没有前驱。
- 2) 除 $root$ 以外，D中每个结点在关系H下都有且仅有一个前驱。

基本操作：

InitTree: 初始化空树

DestroTree: 销毁树

CreateTree: 创建树

TreeEmpty: 判断树是否为空树

Root: 返回树的根

Parent(Tree,x): 树Tree存在, x是Tree中的某个结点。若x为非根结点, 则返回它的双亲, 否则返回“空”

FirstChild(Tree,x): 树Tree存在, x是Tree中的某个结点。若x为非叶子结点, 则返回它的第一个孩子结点, 否则返回“空”

NextSibling(Tree,x): 树Tree存在, x是Tree中的某个结点。若x不是其双亲的最后一个孩子结点, 则返回x后面的下一个兄弟结点, 否则返回“空”

InsertChild(Tree,p,child): 树Tree存在, p指向Tree中某个结点, 非空树Child与Tree不相交。将Child插入Tree中, 做p所指向结点的子树

DeleteChild(Tree,p,i): 树Tree存在, p指向Tree中某个结点, $1 \leq i \leq d$, d为p所指向结点的度。删除Tree中p所指向结点的第i棵子树

TraverseTree(Tree,visit()): 树Tree存在, visit() 是对结点进行访问的函数。按照某种次序对Tree的每个结点调用visit() 函数访问一次且最多一次。若visit() 失败, 则操作失败。

二叉树

定义

二叉树要满足以下两个条件：

1. 每个结点的度都不大于2
2. 每个结点的孩子结点次序不能任意颠倒（有序孩子结点排列）

性质

1. 二叉树的第i层最多有 2^{i-1} 个结点
2. 深度为k的二叉树至多有 $2^k - 1$ 个结点

♣ **性质3**：对任意一棵二叉树T，若终端结点数为 n_0 ，而其度数为2的结点数为 n_2 ，则：
• $n_0 = n_2 + 1$

♣ **证明**：设二叉树中结点总数为n， n_1 为二叉树中度为1的结点总数。因为二叉树中所有结点的度小于等于2，所以有： $n = n_0 + n_1 + n_2$

♣ 设二叉树中分支数目为B，因为除根结点外，每个结点均对应一个进入它的分支，所以有： $n = B + 1$ 。

♣ 又因为二叉树中的分支都是由度为1和度为2的结点发出，所以分支数目为： $B = n_1 + 2n_2$

♣ 整理上述两式可得到： $n = B + 1 = n_1 + 2n_2 + 1$

♣ 将 $n = n_0 + n_1 + n_2$ 代入上式得出 $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$ ，整理后得 $n_0 = n_2 + 1$ ，故结论成立。

完全二叉树：

对一棵具有n个结点的二叉树T按层序编号，如果编号为i ($1 \leq i \leq n$) 的结点与同样深度的满二叉树中编号为i的结点在二叉树中的位置完全相同，则称T为完全二叉树。

具有N个结点的完全二叉树深度为：

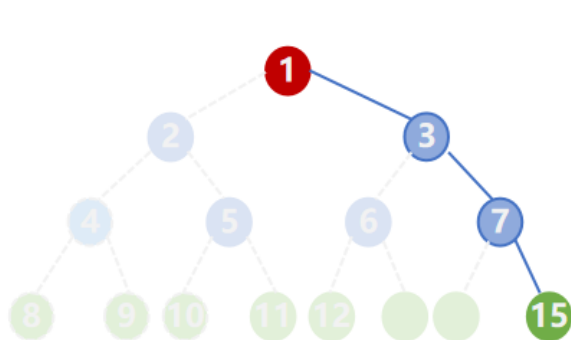
$$\log_2 N + 1$$

存储结构

顺序存储

```
#define MaxSize 100
struct TreeNode{
    ElemType value;//结点数据
    bool isEmpty;//结点是否为空
};
TreeNode t[MaxSize];
//定义一个数组。按照从上到下从左到右的顺序存储完全二叉树里的各个结点
for(int i=0;i<MaxSize;i++){
    t[i].isEmpty = true;
}
```

#二叉树的顺序存储中，一定要把二叉树的结点编号与完全二叉树对应起来。



i 二叉树的顺序存储中，一定要把二叉树的结点编号与完全二叉树对应起来

i的左孩子 —— $2i$

i的右孩子 —— $2i+1$

i的父结点 —— $\lfloor i/2 \rfloor$

最坏情况：高度为h且只有h个结点的单分支树（所有结点只有右孩子），也至少需要 $2^h - 1$ 个存储单元

顺序存储更适合存储完全二叉树

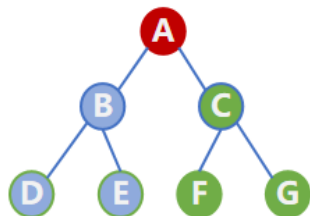
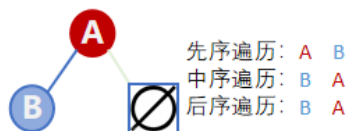
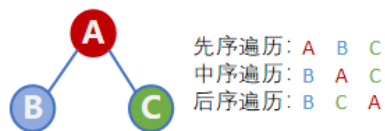


```
typedef struct Node{
    DataType data;
    struct Node * Lchild;
    struct Node * Rchild;
}BiTNode,*BiTree;
//n个结点的二叉链表共有(n+1)个空链域
```

链式存储

```
typedef struct Node{
    DataType data;
    struct Node * Lchild;
    struct Node * Rchild;
    struct Node * parent;
}BiTNode,*BiTree;//三叉链表
```

遍历原理

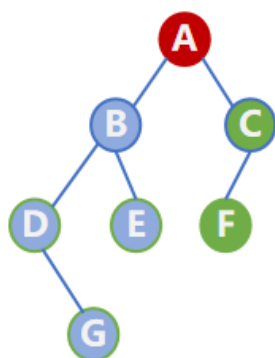


先序遍历: A B D E C F G
中序遍历: D B E A F C G
后序遍历: D E B F G C A

先序遍历: 根左右 (NLR)

中序遍历: 左根右 (LNR)

后序遍历: 左右根 (LRN)



先序遍历: 根 左 右
根 (根 左 右) (根左)
根 (根 (根右) 右) (根左)

A B C
A B D E C F
A B D G E C F

中序遍历: 左 根 右
(左 根 右) 根 (左根)
((根右) 根 右) 根 (左根)

B A C
D B E A F C
D G B E A F C

后序遍历: 左 右 根
(左 右 根) (左 根) 根
((右 根) 右 根) (左 根) 根

B C A
D E B F C A
G D E B F C A

分支结点逐层展开法

//先序遍历: 如二叉树非空: 访问根结点; 先序遍历左子树; 先序遍历右子树

```
typedef struct Node
```

```
{
```

```
    DataType data;
```

```
    struct Node * LChild;
```

```
    struct Node * RChild;
```

```
}BiTNode,*BiTree;
```

```
void PreOrder(BiTree root){
```

```
    if(root!=NULL){
```

```
        Visit(root->data); //printf("%c", root ->data);
```

```
        PreOrder(root->LChild); //递归访问左子树
```

```
        PreOrder(root->RChild); //递归访问右子树
```

```
    }
```

```
}
```

//中序遍历: 如二叉树非空: 中序遍历左子树; 访问根结点; 中序遍历右子树

```
void InOrder(BiTree root){
```

```
    if(root!=NULL){
```

```
        InOrder(root->LChild);
```

```
        Visit(root->data);
```

```
        InOrder(root->RChild);
```

```
    }
```

```
}
```

//后序遍历: 如二叉树非空: 后序遍历左子树; 后序遍历右子树; 访问根结点;

```
void PostOrder(BiTree root){
```

```
    if(root!=NULL){
```

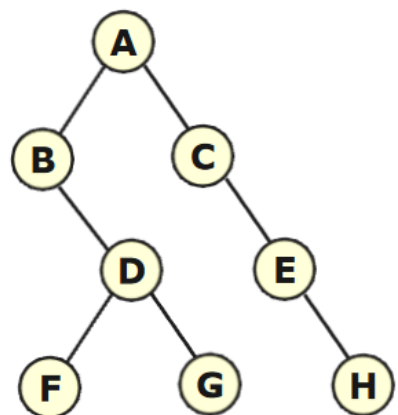
```

        PostOrder(root -> LChild);
        PostOrder(root -> RChild);
        Visit(root -> data);
    }
}
//输出二叉树中的叶子结点: 判断结点有无左, 右孩子, 则输出该结点
void PreOrder(BiTree root){
    if(root!=NULL){
        if(root ->LChild == NULL && root ->RChild == NULL){
            printf ("%c ", root ->data);
        }
        PreOrder(root ->LChild);    /* 先序遍历左子树 */
        PreOrder(root ->RChild);    /* 先序遍历右子树 */
    }
}
//统计叶子结点数目
//算法1
void leaf_a(BiTree root){
    if(root!=NULL){
        leaf_a(root->LChild);
        leaf_a(root->RChild);
        if(root->LChild==NULL && root ->RChild == NULL)
            LeafCount++;
    }
}
//算法2: 后序遍历(分治): 如果是空树返回0; 如果只有一个结点, 返回1; 否则为左右子树的叶子结点数之和
int leaf_b(BiTree root){
    int LeafCount;
    if(root==NULL)
        LeafCount = 0 ;
    else if((root->LChild==NULL)&&(root->RChild==NULL))
        LeafCount = 1;
    else
        LeafCount = leaf_b(root->LChild) + leaf_b(root->RChild);
    return LeafCount;
}
//算法3:

```

4、建立二叉链表方式存储的二叉树

- 给定一棵二叉树, 可以得到它的遍历序列; 反过来, 给定一个遍历序列, 也可以创建相应的二叉链表。
- 在这里所说的遍历序列是一种“扩展的遍历序列”, 通常用特定的元素表示空子树。例如:
AB.DF..G..C.E.H..
- 其中用小圆点表示空子树



```

void CreateBiTree (BiTree *bt){
    char ch;
    ch = getchar();
    if(ch!=".")
        *bt=NULL;
    else{
        *bt = (BiTree)malloc(sizeof(BiNode));
        (*bt)->data=ch;
        CreateBiTree(&((*bt)->LChild)); //生成左子树
        CreateBiTree(&((*bt)->RChild)); //生成右子树
    }
} //二叉树（二叉链表方式）

```

二叉树高度

```

int PostTreeDepth(BiTree bt){
    int hl,hr,max;
    if(bt!=NULL){
        hl=PostTreeDepth(bt->LChild);
        hr=PostTreeDepth(bt->RChild);
        max=hl>hr?hl:hr;
        return(max+1);
    }
    else return(0);
}

```

```

void PreTreeDepth(BiTree bt,int h){
    if(bt!=NULL){
        if(h>depth)
            depth = h; /*如果该结点层次值大于depth，更新depth的值*/
    }
    PreTreeDepth(bt->LChild, h+1); /* 遍历左子树 */
    PreTreeDepth(bt->RChild, h+1); /* 遍历右子树 */
}

/* 前序遍历求二叉树bt高度的递归算法，h为bt指向结点所在层次，初值1*/

```

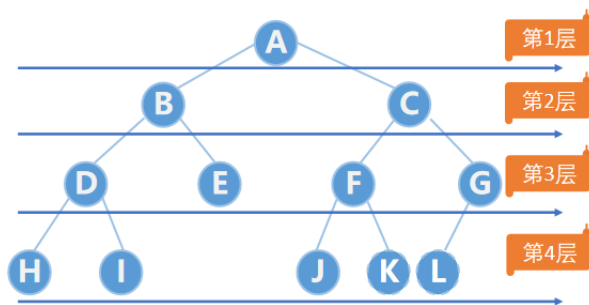
递归消除

在大量复杂的情况下，递归的问题无法直接转换成循环。此时可以用队列消除递归或采用工作栈。

工作栈提供一种控制结构：

当递归算法进层时将信息保留；当递归算法出层时需要从栈区退出信息。

层次遍历(非递归)



实现二叉树的层序遍历 -
-从根开始，依次向下，
对于每一层从左向右遍历。



首先将根结点的指针入队

循环取队首元素，执行如下操作，直至队空为止：

访问该元素（结点）的数据部分

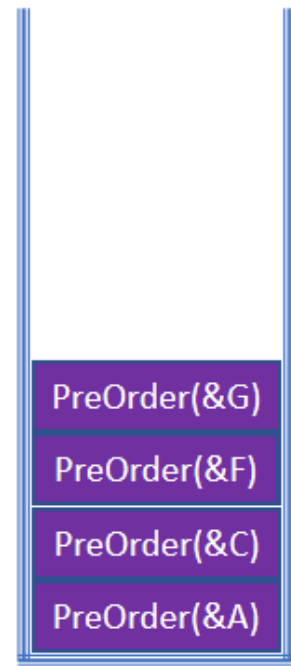
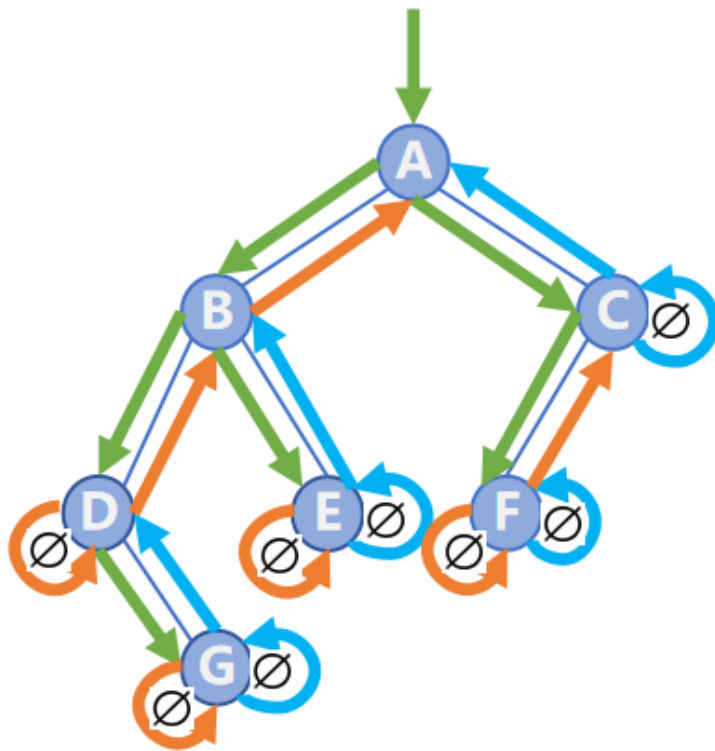
若该结点有左孩子，则将其入队

若该结点有右孩子，则将其入队

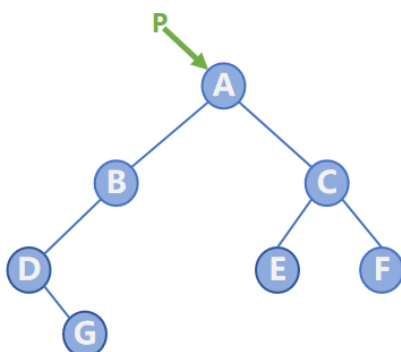
```
void LevelOrder(BiTree bt){
    BiTree Queue[MAXNODE]; //定义队列
    int front, rear;
    if(bt==NULL)
        return;
    Queue[rear]=bt; //根结点入队
    rear++;
    while(rear!=front){ //队列非空则继续遍历
        visit(Queue[front++] -> data); //访问刚出队的元素
        if(Queue[front] -> lchild != NULL){
            Queue[rear]=Queue[front] -> lchild;
            rear++;
        }
        if(Queue[front] -> rchild != NULL){
            Queue[rear]=Queue[front] -> rchild;
            rear++;
        }
    }
}
```

基于栈的递归消除

```
void PreOrder(BiTree root){
    if(root!=NULL){
        visit(root -> data);
        PreOrder(root -> Lchild);
        PreOrder(root -> Rchild);
    }
}
```



函数调用栈



非递归先序遍历

- ① 访问并进栈，取左孩子直到无法再取到左孩子
- ② 栈顶出栈
- ③ 取右孩子并且再回到循环开始进行同样处理

自己构造Stack实现非递归版先序遍历:

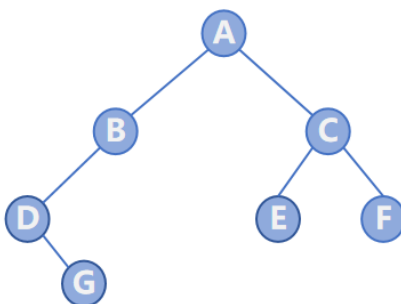


Stack

关键代码

```
while (!StackEmpty(st) || p!=NULL)
{ while (p!=NULL) {
    printf("%c ", p->data);
    Push(st, p);
    p=p->lchild;
}
if (!StackEmpty(st)) {
    Pop(st, p);
    p=p->rchild;
}
}
```

遍历结果 **A B D G C E F**



非递归中序遍历

- ① 不访问进栈，取左孩子直到无法再取到左孩子
- ② 栈顶出栈，访问结点
- ③ 取右孩子并且再回到循环开始进行同样处理

自己构造Stack实现非递归版中序遍历:

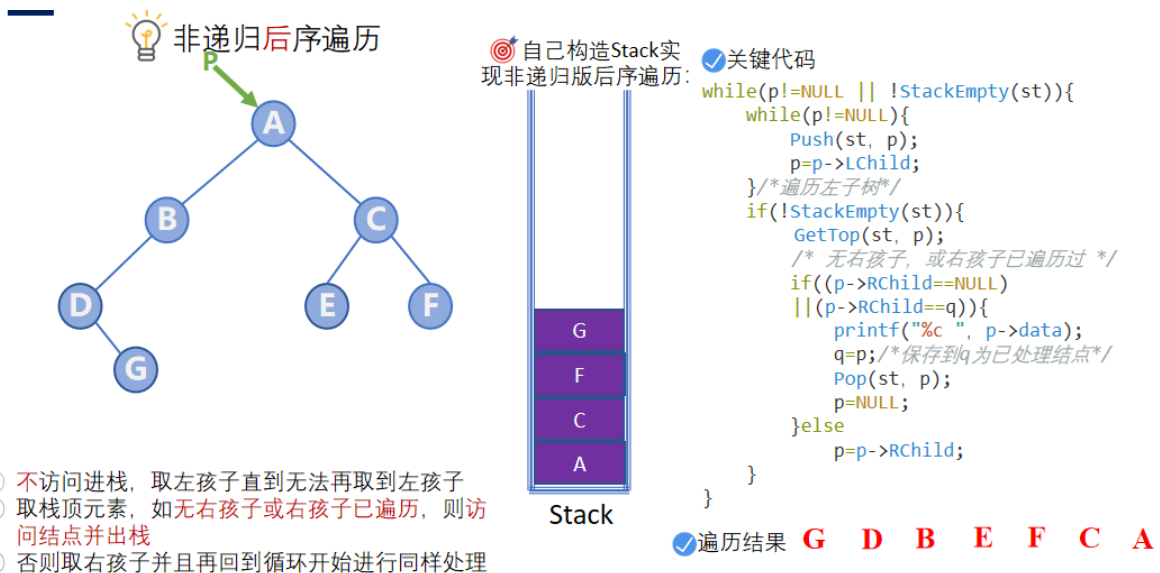


Stack

关键代码

```
while (!StackEmpty(st) || p!=NULL)
{ while (p!=NULL) {
    printf("%c ", p->data);
    Push(st, p);
    p=p->lchild;
}
if (!StackEmpty(st)) {
    Pop(st, p);
    p=p->rchild;
}
}
```

遍历结果 **D G B A E C F**



线索二叉树

线索: 指向前驱和后继结点的指针

线索链表: 以这种结构组成的二叉链表作为二叉树的存储结构, 称为线索链表

线索化: 对二叉树以某种次序进行遍历并且加上线索的过程叫做线索化

```

typedef struct Node{
    DataType data;
    struct Node * LChild;
    struct Node * RChild;
}BiTNode, *BiTree; /*一般二叉树*/

typedef struct Node{
    DataType data;
    struct Node * LChild;
    struct Node * RChild;
    int ltag, rtag; /*左右线索*/
}BiTNode, *BiTree; /*线索二叉树*/

```



tag为0表示指针域指向孩子
tag为1表示指针域是“线索”

LTAG=0: 有左孩子, RTAG=0: 有右孩子;

LTAG=1: 有前驱, RTAG=1: 有后继 (即当没有左孩子时, *LChild指针则指向前驱节点。RTAG同理)

前驱节点: 对一棵二叉树进行中序遍历, 遍历后的顺序, 当前节点的前一个节点为该节点的前驱节点;

后继节点: 对一棵二叉树进行中序遍历, 遍历后的顺序, 当前节点的后一个节点为该节点的后继节点;

线索二叉树的画法: https://www.bilibili.com/video/BV1kd4y1471C/?spm_id_from=333.337.search-card.all.click&vd_source=4e2c42e50717d2840103a5c4b3f7092f

```

typedef struct Node{
    DataType data;

```

```

    struct Node *Lchild;
    struct Node *Rchild;
    int Ltag,Rtag;
}BiTNode,*BiTree;
BiTree pre;//存放当前访问结点的前驱
void Inthread(BiTree root){
    //中序线索化
    if(root!=NULL){
        Inthread(root->Lchild);//线索化左子树
        if(root->Lchild==NULL){//判断是否到达左树最左端；起始时叶子结点的左右指针都为空，
            所以我们就利用叶子结点的左右指针域来建立线索
            root->Ltag=1;
            root->Lchild=pre;
        }
        if(pre!=NULL&&pre->Rchild==NULL){
            pre->Rchild=root;
            pre->Rtag=1;
        }
        pre=root;
        Inthread(root->Rchild);
    }
}

```

```

//寻找前驱
BiTNode * InPre(BiTNode *p){
    BiTNode *q;
    if(p->Ltag==1)
        pre=p->Lchild;
    else
    {
        for(q=p->Lchild;q->Rtag==0;q=q->Rchild)
            pre=q;
    }
    return(pre);
}

```

先序+中序，中序+后序可以确定唯一二叉树；但先序+后序不可以。

```

void Prethread(BiTree root)//前序遍历
{
    if(root!=NULL)
    {
        if(root->Lchild==NULL)
        {
            root->Ltag=1;
            root->Lchild=pre;
        }
        if(pre!=NULL && pre->Rchild == NULL)
        {
            pre->Rchild=root;
            pre->Rtag=1;
        }
        pre=root;
        if(root->Ltag==0)
    }
}

```

```

        {
            Prethread(root->Lchild);
            Prethread(root->Rchild);
        }
    }
}

void Postthread(BiTree root)
{
    if(root!=NULL)
    {
        Postthread(root->Lchild);
        Postthread(root->Rchild);
        if(root->Lchild==NULL)
        {
            root->Ltag=1;
            root->Lchild=pre;
        }
        if(pre!=NULL && pre->Rchild==NULL)
        {
            pre->Rchild=root;
            pre->Rtag=1;
        }
        pre=root;
    }
}

```

双亲表示法

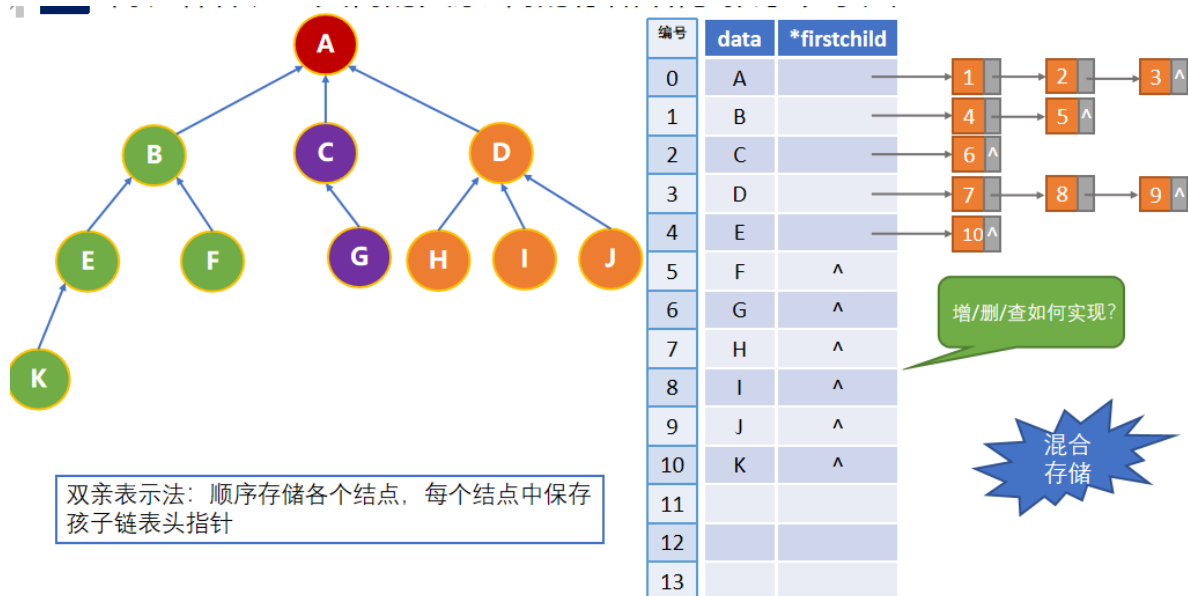
每个结点中保存指向双亲的“指针”，即parent域内保存双亲的编号。

```

#define MAX 100
typedef struct TNode{
    DataType data;
    int parent;
}TNode;
typedef struct{
    TNode tree[MAX];
    int nodenum; //结点数
}ParentTree;
//当需要删除结点时直接将其parent值变为-1；同时新增元素不需要按照逻辑顺序存储

```

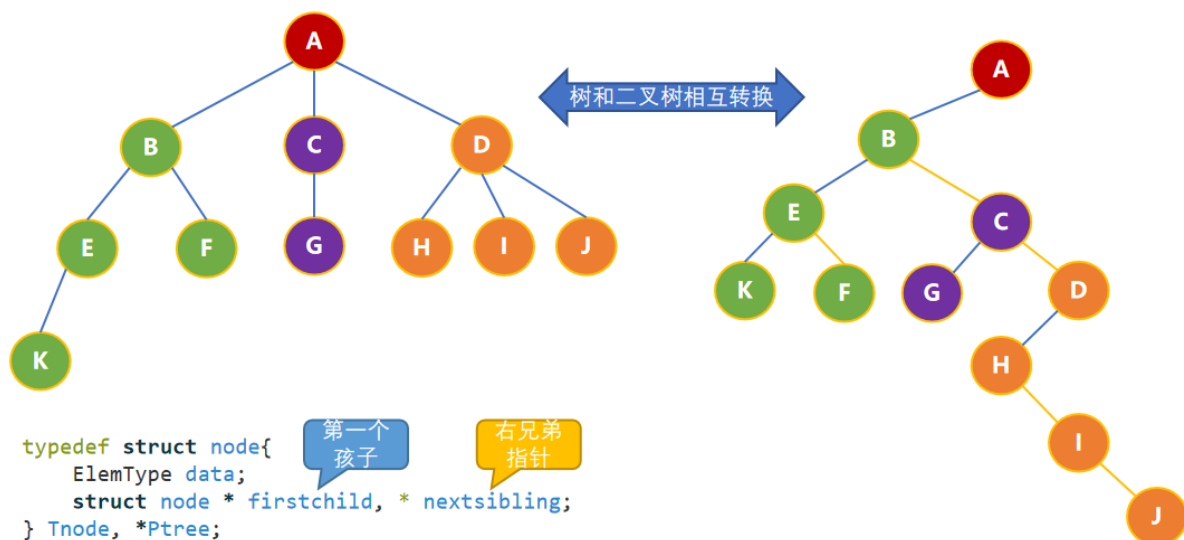
孩子表示法



```
typedef struct Childnode//孩子链表
{
    int Child;
    struct Childnode * next;//指向下一个孩子结点的指针
}Childnode;
typedef struct{//顺序表的结点
    DataType data;
    Childnode * FirstChild;//指向孩子链表的头指针
}DataNode;
typedef struct{
    Datanode nodes[MAX];
    int root,num;//根节点位置以及结点个数
}ChildTree;
```

孩子兄弟表示法

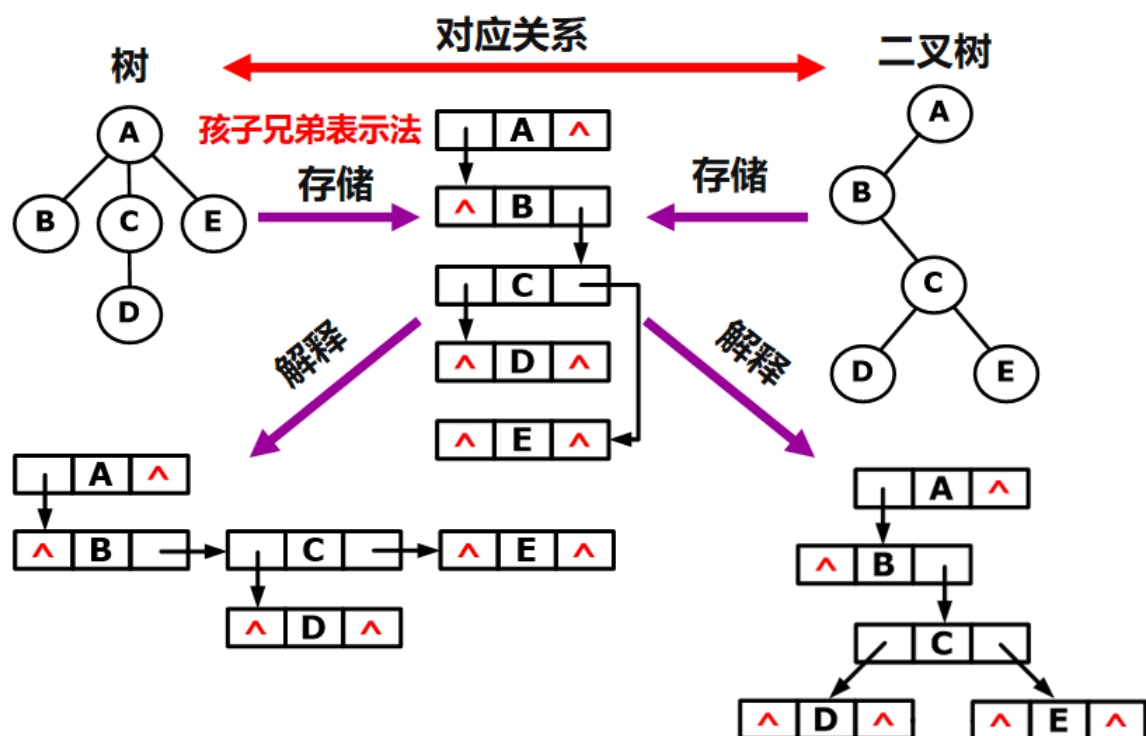
```
typedef struct node{
    ElemType data;
    struct node * firstchild, * nextsibling;
}TNode,*PTree;
typedef struct Node{//二树的结点
    DataType data;
    struct Node * LChild;
    struct Node * RChild;
}BiTNode,*BiTree;
```

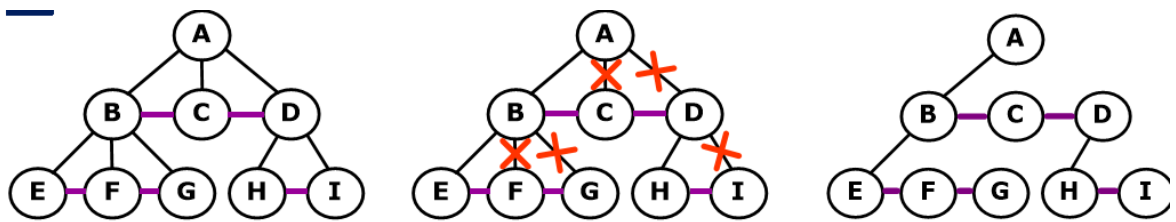


可利用二叉链表导出树和二叉树的对应关系：

对一颗采用孩子兄弟链表表示法存储的树可以找到一颗二叉树的二叉链表与之对应；

二者的物理存储方式一致但解释不同；

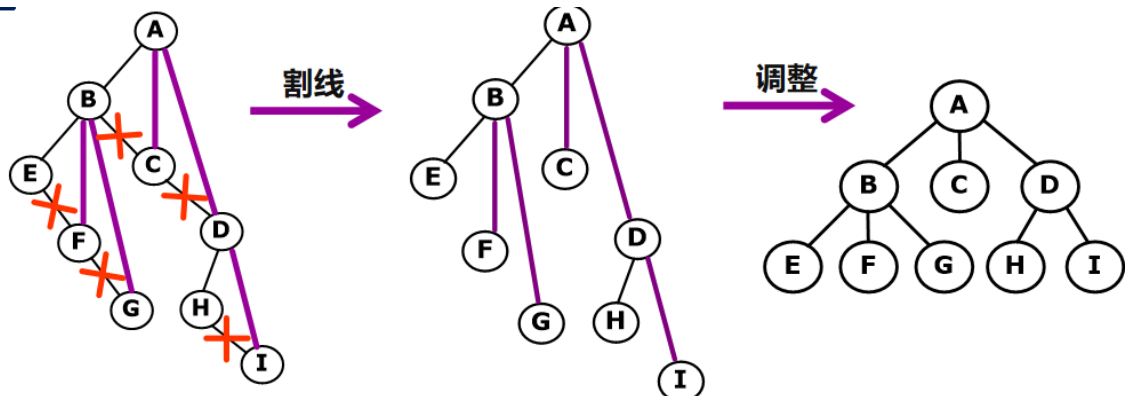
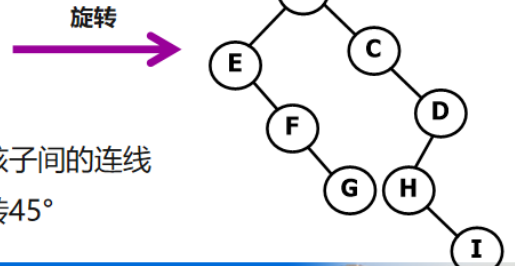




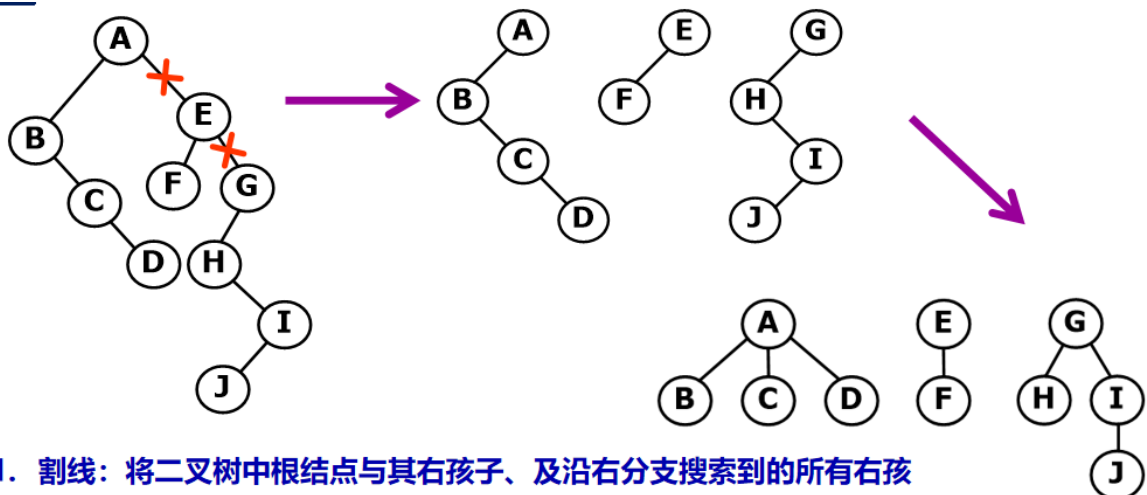
特点：树转换成的二叉树其右子树一定为空

树的根结点没有兄弟

1. 加线：在所有兄弟节点之间加一条连线
2. 割线：对每个结点，去掉除左孩子外它和其余孩子间的连线
3. 旋转：以树的根结点为轴心，将整棵树顺时针转45°



1. 加线：若结点是父结点p的左孩子，则将该结点的右孩子，右孩子的右孩子(.....所有右孩子)，都与父结点p进行连线
2. 割线：去掉原二叉树中父结点与右孩子之间的连线
3. 调整：将结点按层次排列，形成树形结构



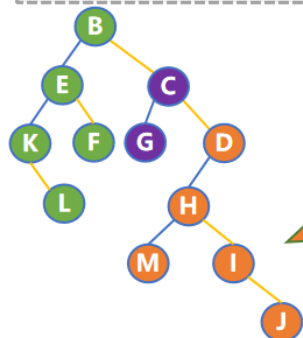
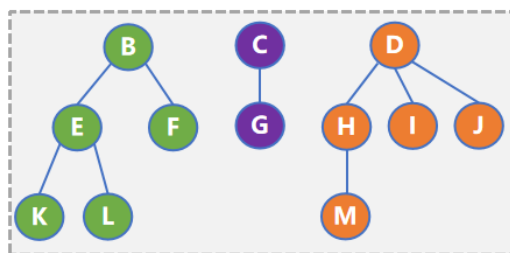
1. 割线：将二叉树中根结点与其右孩子、及沿右分支搜索到的所有右孩子间的连线全部去掉，使之变成孤立的二叉树
2. 还原：将孤立的二叉树还原成树

树的遍历

树的先根遍历=>树转换为二叉树后前序遍历

树的后根遍历=>树转换为二叉树后中序遍历

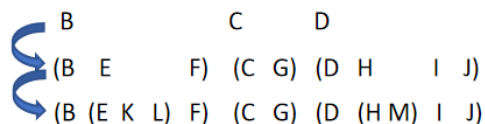
森林的遍历



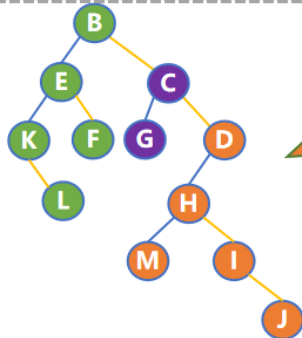
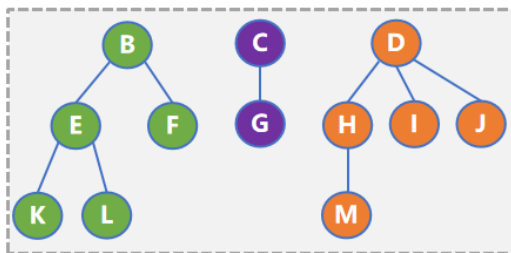
效果还等同于对应二叉树进行先序遍历

先序(先根)遍历森林的方法：若森林非空

- 访问森林中第一棵树的根节点
- 先序遍历第一棵树中根节点的子树森林
- 先序遍历除去第一棵树之后剩余的树构成的森林



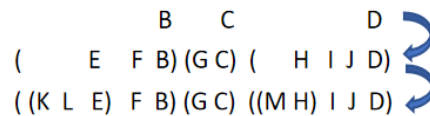
效果等同于依次对各个树进行先根遍历



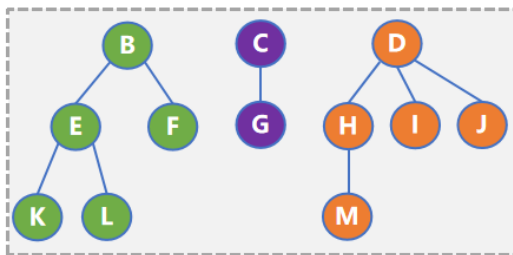
效果还等同于对应二叉树进行中序遍历

中序(后根)遍历森林的方法：若森林非空

- 中序遍历森林中第一棵树的根结点的子树森林
- 访问第一棵树的根结点
- 中序遍历除去第一棵树之后剩余的树构成的森林

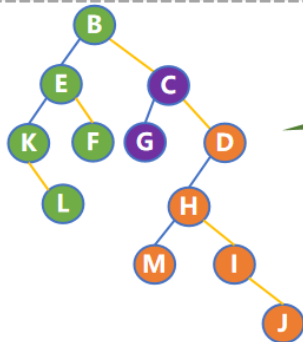


效果等同于依次对各个树进行后根遍历



后序遍历森林的方法：若森林非空

- 🌲 后序遍历森林中第一棵树的根结点的子树森林
- 🌲 后序遍历除去第一棵树之后剩余的树构成的森林
- 🌲 访问第一棵树的根结点



效果等同于对应二叉树进行后序遍历

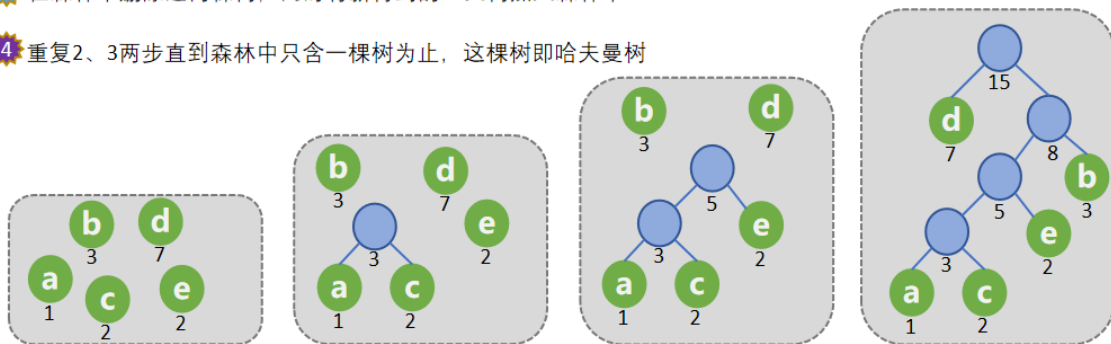
L K F E G M J I H D C B

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历
	后序遍历	后序遍历

哈夫曼树

定义

- 1 根据给定的 n 个权值： $\{w_1, w_2, \dots, w_n\}$ ，构造 n 棵只含根结点的二叉树，令每棵树的权值为相应的结点权值 (w_i)
- 2 在森林中选取两棵根结点权值最小的树作为左右子树，构造一棵新的二叉树，新树根节点权值为其左右子树根结点权值之和
- 3 在森林中删除这两棵树，同时将新得到的二叉树加入森林中
- 4 重复2、3两步直到森林中只含一棵树为止，这棵树即哈夫曼树



结点的带权路径长度：从树的根到该结点的路径长度（经过的边数）与该结点上权值的乘积

树的带权路径长度：树种所有叶结点的带权路径长度之和(WPL, Weighted Path Length)

*. 权值小的二叉树作为新构造的二叉树的左子树, 权值大的二叉树作为新构造的二叉树的右子树; 在权值相等时, 深度小的二叉树作为新构造的二叉树的左子树, 深度大的二叉树作为新构造的二叉树的右子树


```

#define N 50//叶子结点数
#define M 2*N-1//总结点数
typedef struct TreeNode{
    char data;//结点值
    int weight;//权重值
    int parent;
    int lchild;
    int rchild;
}HTNode;

```

```

//哈夫曼树
void CreateHT(HTNode ht[],int n){
    int i,k,lnode,rnode;
    double min1,min2;
    for(i=0;i<2*n-1;i++)
    {
        ht[i].parent=ht[i].lchild=ht[i].rchild=-1;//所有结点的域初始值为-1（初始化）
    }
    for(k=0;k<=i-1;k++)
    {
        min1=min2=32267;//lnode和rnode为权重最小的两个节点
        lnode=rnode=-1;
        if(ht[k].parent == -1)
        {
            min2=min1;rnode=lnode;
            min1=ht[k].weight;lnode=k;
        }
        else if(ht[k].weight<=min2)
        {
            min2=ht[k].weight;rnode=k;
        }//在0到i-1的范围内寻找权值最小的两个结点
    }
    ht[i].weight=ht[lnode].weight+ht[rnode].weight;//ht[i]为双亲结点
    ht[i].lchild=lnode;ht[i].rchild=rnode;
    ht[lnode].parent=i;ht[rnode].parent=i;
}

```

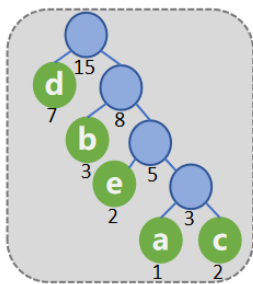
哈夫曼编码

对一棵具有n个叶子的哈夫曼树，若对树中的每个左分支赋予0，右分支赋予1（也可左1右0），则从根到每个叶子的通路上，各分支的赋值分别构成一个二进制串，该二进制串就称为哈夫曼编码；哈夫曼编码是最优前缀码。

```

#define LEN 100//编码字符个数
typedef struct{
    char ch;//存储字符
    char code[LEN]; //存放编码
}TCode;
TCode CodeBook[LEN]; //编码本

```



a 1110 **a** 1 1 1 0
b 10 **b** 1 0
c 1111 **c** 1 1 1 1
d 0 **d** 0
e 110 **e** 1 1 0

	0	1	2	3	4	5	6	7	8
Data	a	b	c	d	e				
Weight	1	3	2	7	2	3	5	8	15
Parent	5	7	5	8	6	6	7	8	-1
Lchild	-1	-1	-1	-1	-1	0	4	1	3
Rchild	-1	-1	-1	-1	-1	2	5	6	7

- 1 选择哈夫曼树的某个叶结点
- 2 利用parent指针找到其父结点
- 3 利用父结点的孩子指针域判断该结点是左孩子还是右孩子,如果是左孩子则生成0, 反之则生成1
- 4 重复步骤(2)~(3) 直至回溯到根节点, 得到一个0/1序列该序列是Huffman编码的逆序: 将其反序得到字符编码
- 5 重复步骤(1)~(4), 实现对全部叶结点的编码

```

void encoding (HTNode ht[], TCode book[],int n){
    char *str = (char *)malloc(n+1);
    int i,j,idx,p;
    for(i=0;i<n,i++){
        {
            book[i].ch =ht[i].ch;//内容复制到编码本
            idx = i;
            j=n;
            while(p = ht[idx].parent > 0)//双亲结点存在
            {
                if(ht[p].lchild == idx)
                {
                    j--;str[j]="0";
                }
                else{
                    j--;str[j]="1";
                }
                idx=p;
            }
            strcpy(book[i].code,&str[j]);
        }
    }
}
  
```

```

void decoding(HTNode ht[],char* codes,int n)
{
    int i,p;
    i=0;
    p=2*n-2;//从根节点开始
    while(codes[i]!="0")
    {
        while(ht[p].lchild!=-1 &&ht[p].rchild!=-1)
        {
            if(codes[i]=="0")
                p=ht[p].lchild;
            else
                p=ht[p].rchild;
            i++;
        }
        printf("%c",ht[p].ch);
    }
}
  
```

```

        p=2*n-2;
    }
    printf("\n");
}

```

Para.7 图

基本概念

图G由顶点集V和边集E组成，记为 $G=(V,E)$ ，其中 $V(G)$ 表示图G中顶点的有限非空集； $E(G)$ 表示图G中顶点之间的关系（边）集合。若 $V=\{v_1, v_2, \dots, v_n\}$ ，则用 $|V|$ 表示图G中顶点的个数，也称图G的阶， $E=\{(u,v) | u \in V, v \in V\}$ ，用 $|E|$ 表示图G中边的条数。

若E是无向边（简称边）的有限集合时，则图G为无向图。边是顶点的无序对，记为 (v, w) 或 (w, v) ，因为 $(v,w)=(w, v)$ ，其中 v, w 是顶点。可以说顶点 w 和顶点 v 互为邻接点。边 (v,w) 依附于顶点 w 和 v ，或者说边 (v, w) 和顶点 v, w 相关联。

若E是有向边（也称弧）的有限集合时，则图G为有向图。弧是顶点的有序对，记为 $\langle v, w \rangle$ ，其中 v, w 是顶点， v 称为弧尾， w 称为弧头， $\langle v, w \rangle$ 称为从顶点 v 到顶点 w 的弧，也称 v 邻接到 w ，或 w 邻接自 v 。 $\langle v, w \rangle \neq \langle w, v \rangle$

ADT

ADT Graph {

数据对象V：一个集合，该集合中的所有元素具有相同的特性。

数据关系R： $R=\{VR\}$ $VR=\{\langle x, y \rangle \mid P(x, y) \wedge (x, y \in V)\}$

基本操作：

(1) CreateGraph(G)：创建图G。

(2) DestroyGraph(G)：销毁图G。

(3) LocateVertex(G, v)：确定顶点 v 在图G中的位置。若图G中没有顶点 v ，则函数值为“空”。

(4) GetVertex(G, i)：取出图G中的第 i 个顶点的值。若 $i >$ 图G中顶点数，则函数值为“空”。

(5) FirstAdjVertex(G, v)：求图G中顶点 v 的第一个邻接点。若 v 无邻接点或图G中无顶点 v ，则函数值为“空”。

(6) NextAdjVertex(G, v, w)：已知 w 是图G中顶点 v 的某个邻接点，求顶点 v 的下一个邻接点（紧跟在 w 后面）。若 w 是 v 的最后一个邻接点，则函数值为“空”

(7) InsertVertex(G, u)：在图G中增加一个顶点 u 。

(8) DeleteVertex(G, v)：删除图G的顶点 v 及与顶点 v 相关联的弧。

(9) InsertArc(G, v, w)：在图G中增加一条从顶点 v 到顶点 w 的弧。

(10) DeleteArc(G, v, w)：删除图G中从顶点 v 到顶点 w 的弧。

(11) TraverseGraph(G)：按照某种次序，对图G的每个结点访问一次且最多一次。

} ADT Graph;

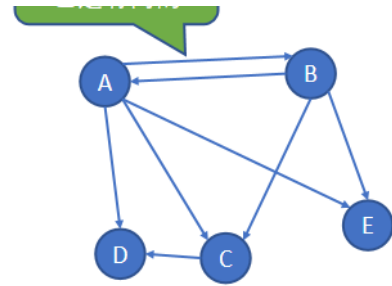
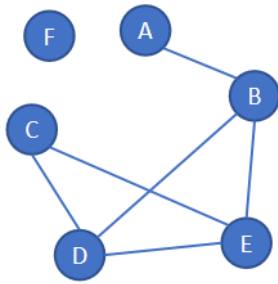
其余概念

无向图：顶点 v 的度是指依附于该顶点的边的条数，记为 $TD(v)$ 。

无向图全部顶点的度是边数的两倍

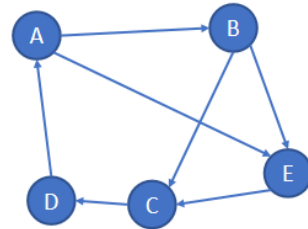
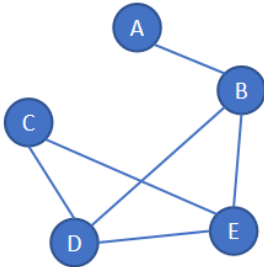
有向图：入度是以顶点 v 为终点的有向边的数目；出度是以顶点 v 为起点的有向边的数目；点 v 的度等于其入度和出度之和

$$\text{edges} = \sum_{i=1}^n \text{in-degree}(v_i) = \sum_{i=1}^n \text{out-degree}(v_i)$$



- 路径**——顶点 V_p 到顶点 V_q 之间的一条路径是指顶点序列, $V_p, V_{i1}, V_{i2}, \dots, V_{im}, V_q$
- 回路**——第一个顶点和最后一个顶点相同的路径称为回路或环
- 简单路径**——在路径序列中, 顶点不重复出现的路径称为简单路径
- 简单回路**——除第一个顶点和最后个顶点外, 其余顶点不重复出现的回路称为简单回路。
- 路径长度**——路径上边的数目
- 点到点的距离**——从顶点 u 出发到顶点 v 的最短路径若存在, 则此路径的长度称为从 u 到 v 的距离。若从 u 到 v 根本不存在路径, 则记该距离为无穷(∞)

- 无向图中**, 若从顶点 v 到顶点 w 有路径存在, 则称 v 和 w 是**连通的**
- 有向图中**, 若从顶点 v 到顶点 w 和从顶点 w 到顶点 v 之间都有路径, 则称这两个顶点是**强连通的**



若图 G 中任意两个顶点都是连通的, 则称图 G 为**连通图**, 否则称为**非连通图**。

注意:

对于 n 个顶点的无向图 G ,

若 G 是**连通图**, 则**最少**有 $n-1$ 条边

若 G 是**非连通图**, 则**最多**可能有 C_{n-1}^2 条边

若图中任何一对顶点都是强连通的, 则称此图为**强连通图**。

注意:

对于 n 个顶点的有向图 G ,

若 G 是**强连通图**, 则**最少**有 n 条边 (形成回路)

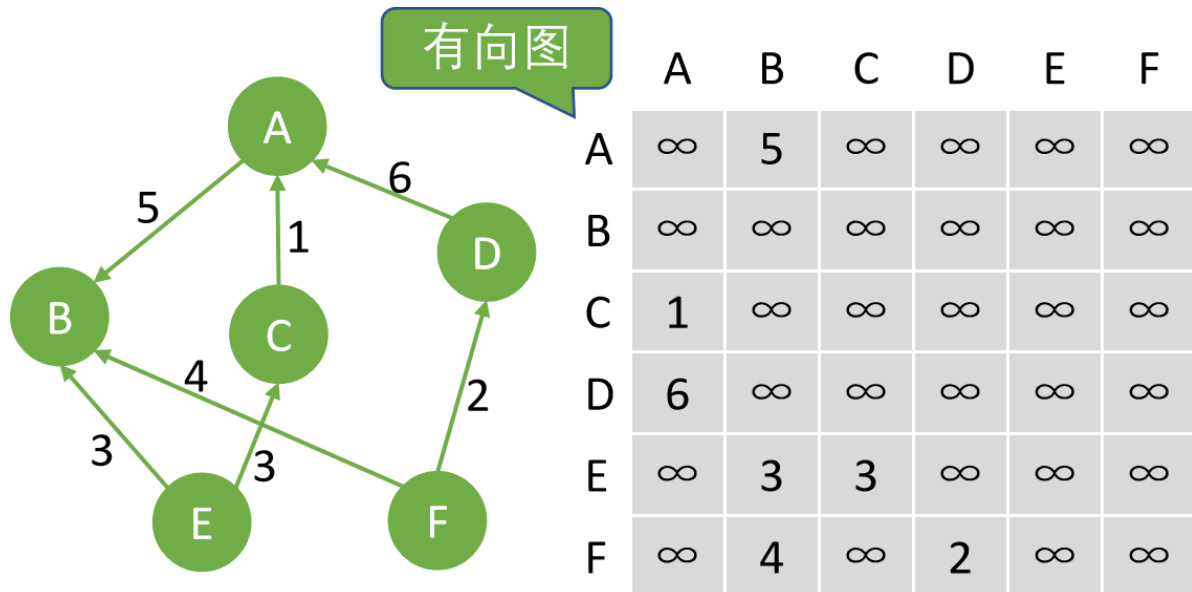
邻接矩阵

```
//无权图
#define MAX_VERTEX_NUM 20
typedef struct{
    char vertex[MAX_VERTEX_NUM]; //顶点向量
    int [MAX_VERTEX_NUM] [MAX_VERTEX_NUM]; //邻接矩阵
    int vexnum, arcnum; //图的顶点数和弧数
}AdjMatrix; //邻接矩阵
```

时间复杂度 $O(n)$

```
//带权图
#define MAX_VERTEX_NUM 100
#define UNFINITY 32768
typedef enum {DG, DN, UDG, UDN} GraphKind; //分别为有向图, 无向图, 有向网, 无向网
typedef char VertexData; //假设顶点数据 (这里是字符型)
typedef struct ArcNode{
    AdjType adj;
}
```

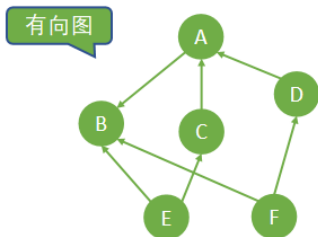
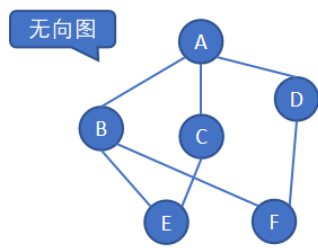
```
typedef struct{
    char vertex[MAX_VERTEX_NUM]; //顶点向量
    int [MAX_VERTEX_NUM][MAX_VERTEX_NUM]; //邻接矩阵
    int vexnum, arcnum; //图的顶点数和弧数
    GraphKind kind; //图的种类
}AdjMatrix; //邻接矩阵
```



时间复杂度 $O(|V|^2)$

邻接表

```
typedef struct ArcNode{
    int adjvex; //指向顶点位置
    struct ArcNode *nextarc; //指针指向下一条弧
    OtherInfo info; //与弧相关的信息
}ArcNode;
typedef struct Vertex{
    VertexData data;
    ArcNode *firsstarc; //指针指向下一条弧
}VertexNode;
typedef struct{
    VertexNode vertex[MAX_VERTEX_NUM];
    int vexnum, arcnum; //图的顶点数和弧数
    GraphKind kind; //种类标志
}AdjList;
```



编号	data	*first
0	A	1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1

边结点的数量是 $2|E|$, 整体空间复杂度为 $O(|V| + 2|E|)$

编号	data	*first
0	A	1
1	B	^
2	C	0
3	D	0
4	E	1
5	F	1

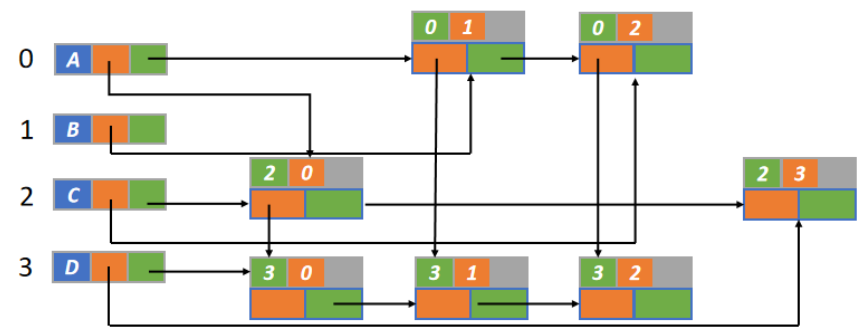
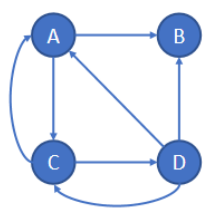


思考
如何求顶点的度、出度、入度?
如何找到一个顶点相连的边/弧?

边结点的数量是 $|E|$, 整体空间复杂度为 $O(|V| + |E|)$

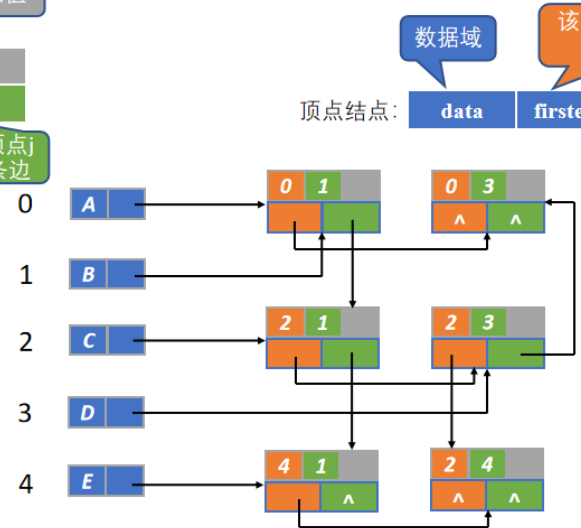
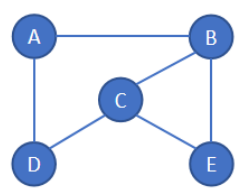
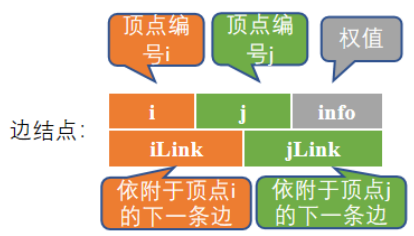
十字链表

2 图的存储结构-十字链表



空间复杂度 $O(V+E)$

邻接多重表



每条边只对应一份数据
空间复杂度: $O(|V| + |E|)$

删除边、删除节点等操作很方便

注意: 邻接多重表只适用于存储无向图

深度优先遍历DFS

```
bool visited[MAX_VERTEX_NUM];
void DepthFirstSearch(Graph g,int v0)
{
    visit(v0);
    visited[v0]=True;
    w=FirstAdjVertex(G,v0);
    while(w!=-1)
    {
        if(!visited[w])
            DepthFirstSearch(g,w);
        w=NextAdjVertex(g,v0,w);
    }
}

//递归版
bool visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph g){
    for (vi=0;vi<g.vexnum;vi++)
        visited[vi]= false ;    /*初始化访问标志数组*/
    for( vi=0;vi<g.vexnum;vi++)    /*调用深度遍历连通子图的操作*/
        if (! visited[vi] )
            DepthFirstSearch(g, vi);
    /*若图g是连通图，则此循环只执行一次*/
}

void DepthFirstSearch (Graph g, int v0) { /*深度遍历v0所在的连通子图*/
    visit (v0) ;
    visited[v0] =True;    /*访问顶点v0，并置访问标志数组相应分量值*/
    w=FirstAdjVertex(g,v0);
    while ( w!=-1) {    /*邻接点存在*/
        if(! visited [w] )
            DepthFirstSearch(g,w);    /*递归调用DepthFirstSearch*/
        w=NextAdjVertex(g,v0,w);    /*找下一个邻接点*/
    }
}
```

```
//非递归版
bool visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph g){
    for (vi=0;vi<g.vexnum;vi++)
        visited[vi]= false ;    /*初始化访问标志数组*/
    for( vi=0;vi<g.vexnum;vi++)    /*调用深度遍历连通子图的操作*/
        if (! visited[vi] )
            DepthFirstSearch(g, vi);
    /*若图g是连通图，则此循环只执行一次*/
}

void DFS1(Graph G, int v0) {
    InitStack(&S);
    visit(v0);visited[v0]=true; Push(&S,v0);
    while(!IsEmpty(&S)) {
        GetTop(&S,&v);
        w=FirstAdjVertex(G,v);
```

```

        while(w!=-1) {
            if(!visited[w]) {
                visit(v); visited[v]=true; Push(&S,v); break; }
            else
                w=FirstAdjVertex(G,v,w);
        }
        if(w==--1) Pop(&S,v);
    }
}

```

同一个图的邻接矩阵表示方式唯一，因此深度优先遍历序列唯一，深度优先生成树也唯一。同一个图邻接表表示方式不唯一，因此深度优先遍历序列不唯一，深度优先生成树也不唯一。

对无向图进行DFS遍历调用DFS函数的次数=连通分量数。对于连通图，只需调用1次DFS。

对于强连通图，从任一结点出发都只需调用1次DFS。

广度优先遍历BFS

FirstAdjVertex(G,v)：求图G中顶点v的第一个邻接点。若v无邻接点或图G中无顶点v，则函数值为“空”。
 NextAdjVertex(G,v,w)：已知w是图G中顶点v的某个邻接点，求顶点v的下一个邻接点（紧跟在w后面）。若w是v的最后一个邻接点，则函数值为“空”。

```

void BFSTraverse(Graph g) { /*对图g进行深度优先搜索*/
    for (vi=0;vi<g.vexnum;vi++)
        visited[vi]= false ; /*初始化访问标志数组*/
    for( vi=0;vi<g.vexnum;vi++) /*调用深度遍历连通子图的操作*/
        if (! visited[vi] )
            BreadthFirstSearch(g, vi);
    /*若图g是连通图，则此循环只执行一次*/
}/* TraverseGraph */

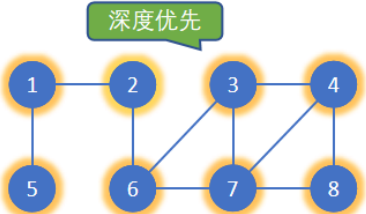
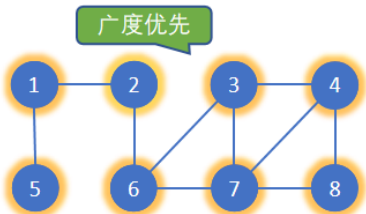
bool visited[MAX_VERTEX_NUM]; /*访问标志数组*/

void BreadthFirstSearch(Graph g,int v0) { /*广度优先搜索图g中v0所在的连通子图*/
    visit(v0);
    visited[v0]=True;
    InitQueue(&Q); /*初始化空队*/
    EnterQueue(&Q,v0); /* v0进队*/
    while(!Empty(Q)) {
        DeleteQueue(&Q,&v); /*队头元素出队*/
        w=FirstAdjVertex(g,v); /*求v的第一个邻接点*/
        while(w!=-1) {
            if(!visited(w)) {
                visit(w);
                visited[w]=True;
                EnterQueue(&Q, w);}
            w=NextAdjVertex(g,v,w); /*求v相对于w的下一个邻接点*/
        }
    }
}

```

两种搜索算法的时间复杂度：对于邻接矩阵均为 V^2 ，对于邻接表均为 $(V+E)$ 。

空间复杂度：两者视情况在1~V之间。

	
深度优先遍历	广度优先遍历
类似于树的先根遍历（树的深度优先遍历）	类似于树的层序遍历（树的广度优先遍历）
1.深入探索未访问的邻接点 2.通过firstAdjVertex和nextAdjVertex找到所有邻接点 3.利用visited数组防止重复访问 4.非连通图需要每个连通分量调用DFS	1.需要一个辅助队列 2.通过firstAdjVertex和nextAdjVertex找到所有邻接点 3.利用visited数组防止重复访问 4.非连通图需要每个连通分量调用BFS
由深度优先遍历过程生成深度优先生成树 邻接表存储的图，深度遍历序列不唯一，深度优先生成树不唯一 非连通图的深度优先遍历形成深度优先生成森林	由广度优先遍历过程生成深度优先生成树 邻接表存储的图，广度遍历序列不唯一，广度优先生成树不唯一 非连通图的广度优先遍历形成广度优先生成森林
空间复杂度为 $O(V)$ 时间复杂度：邻接矩阵 $O(V ^2)$,邻接表 $O(V + E)$	空间复杂度为 $O(V)$ 时间复杂度：邻接矩阵 $O(V ^2)$,邻接表 $O(V + E)$
无向图：调用次数根据连通分量个数有关 有向图：调用次数根据起始顶点有关，强连通只需一次	无向图：调用次数根据连通分量个数有关 有向图：调用次数根据起始顶点有关，强连通只需一次

最小生成树

连通图的生成树：包含全部顶点的一个极小连通子图

对于一个带权连通无向图 $G=(V,E)$ ，生成树不同，每棵树的权（即树中所有边上的权值之和）也可能不同。设 R 为 G 的所有生成树的集合，若 T 为 R 中边的权值之和最小的生成树，则 T 称为 G 的最小生成树（Minimum-Spanning-Tree, MST）。

如果一个连通图本身就是一棵树，则其最小生成树就是它本身

只有连通图才有生成树，非连通图只有生成森林

Prim算法：

Prim算法（普里姆）：从某一个顶点开始构建生成树；每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。

时间复杂度： $O(|V|^2)$ 适用于边稠密图

Kruskal算法：

每次选择一条权值最小的边，使这条边的两头连通（原本已经连通的就不选），直到所有结点都连通

时间复杂度： $O(|E|\log|E|)$ 适用于边稀疏图

拓扑排序

定义

由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：①每个顶点出现且只出现一次。②若顶点 A 在序列中排在顶点 B 的前面，则在图中不存在从顶点 B 到顶点 A 的路径。

实现

- ①从AOV网中选择一个没有前驱（入度为0）的顶点并输出。
- ②从网中删除该顶点和所有以它为起点的有向边。
- ③重复①和②直到当前的AOV网为空或当前网中不存在无前驱的顶点为止。

```
#define MaxVertexNum 100//图中顶点数目的最大值
```

```

typedef struct ArcNode{ //边表结点
    int adjvex; //该弧所指向的顶点的位置
    struct ArcNode *nextarc; //指向下一条弧的指针
    //InfoType info; //网的边权值
}ArcNode;
typedef struct VNode
{
    //顶点表结点
    vertexType data; //顶点信息
    ArcNode *firstarc; //指向第一条依附该顶点的弧的指针
} VNode, AdjList[MaxVertexNum];
typedef struct
{
    AdjList vertices; //邻接表
    int vexnum, arcnum; //图的顶点数和弧数
} Graph; //Graph是以邻接表存储的图类型

bool Topologicalsort(Graph G)
{
    InitStack(S); //初始化栈，存储入度为0的顶点
    for (int i = 0; i < G.vexnum; i++)
        if (indegree[i] == 0)
            push(s, i); //将所有入度为0的顶点进栈
    int count = 0; //计数，记录当前已经输出的顶点数
    while (!IsEmpty(s))
    {
        //栈不空，则存在入度为0的顶点
        Pop(s, i); //栈顶元素出栈
        print[count++] = i; //输出顶点i
        for (p = G.vertices[i].firstarc; p; p = p->nextarc){
            //将所有i指向的顶点的入度减1，并且将入度减为0的顶点压入栈s
            v = p->adjvex;
            if (--indegree[v])
                Push(s, v); //入度为0，则入栈
        }
    } //while
    if (count < G.vexnum)
        return false; //排序失败，有向图中有回路
    else
        return true; //拓扑排序成功
}

```

逆拓扑排序

```

bool visited[MAX_VERTEX_NUM]; //访问标志数组*/
void DFSTraverse(Graph g) { //对图g进行深度优先搜索*/
    for (vi=0;vi<g.vexnum;vi++)
        visited[vi]= false; //初始化访问标志数组*/
    for( vi=0;vi<g.vexnum;vi++) //调用深度遍历连通子图的操作*/
        if (! visited[vi] )
            DepthFirstSearch(g, vi);
    /*若图g是连通图，则此循环只执行一次*/
}/* TraverseGraph */
void DepthFirstSearch (Graph g, int v0) { /*深度遍历v0所在的连通子图*/
    visit (v0);
    visited[v0] =True; //访问顶点v0，并置访问标志数组相应分量值*/
    w=FirstAdjVertex(g,v0);
}

```

```

while ( w!=-1) {    /*邻接点存在*/
    if(! visited [w] )
        DepthFirstSearch(g,w); /*递归调用DepthFirstSearch*/
    w=NextAdjvertex(g,v0,w);    /*找下一个邻接点*/
}
}

```

有向无环图的关键路径（略）

最短路径求法

DFS简单路径

```

int pre[]; /*记录从第u个顶点到v个顶点的简单路径*/
void one_path(Graph *G, int u, int v) {
/*在连通图G中找一条从第u个顶点到v个顶点的简单路径*/
    int i;
    pre=(int *)malloc(G->vexnum*sizeof(int));
    for(i=0;i<G->vexnum;i++) /* 初始化pre数组的各个元素为不可能的下标值-1*/
        pre[i]=-1;
    pre[u]=-2; /*将pre[u]置为-2，表示初始顶点u已被访问，且没前驱*/
    DFS_path(G, u, v); /*用深度优先搜索找一条从u到v的简单路径。*/
    free(pre);
}

int DFS_path(Graph *G, int u, int v) {
/*在连通图G中用深度优先搜索策略找一条从u到v的简单路径。*/
    int j;
    for(j=firstadj(G,u);j>=0;j=nextadj(G,u,j))
        if(pre[j]==-1) {
            pre[j]=u;
            if(j==v){
                print_path(pre ,v); /*从v开始沿着pre输出路径u到v*/
                return 1;
            }
            else if(DFS_path(G,j,v))
                return 1;
        }
    return 0;
}

```

DFS最短路径

```

void BFS_MIN_Distance(Graph G, int u)
{
    // d[i] 表示从u到i结点的最短路径
    for (i = 0; i < G.vexnum; ++i){
        d[i] = INF;    // 初始化路径长度
        path[i] = -1; //最短路径从哪个顶点过来
    }
    d[u] = 0;
    visited[u] = TRUE;
    EnQueue(Q, u);
    while (!isEmpty(Q)){    //BFS算法主过程

```

```

        DeQueue(Q, u);          //队头元素u出队
        for (w = FirstAdjVertex(G, u); w >= 0;
            w = NextAdjVertex(G, u, w))
            if (!visited[w]){ // w为u的尚未访问的邻接顶点
                d[w] = d[u] + 1;    //路径长度加1
                path[w] = u;        //最短路径应从u到w
                visited[w] = TRUE;  //设已访问标记
                EnQueue(Q, w);      //顶点w入队
            } // if
        } // while
    }
}

```

Para.8 排序

基本信息

列表（查找表）——由同一类型的数据元素（或记录）构成的集合，可利用任意数据结构实现

关键字——数据元素中唯一标识该元素的某个数据项的值，使用基于关键字的查找，查找结果应该是唯一的。

查找——根据给定的关键字值，在特定的列表（查找表）中确定一个其关键字与给定值相同的数据元素，并返回该数据元素在列表中的位置。

查找长度——在查找运算中，需要对比关键字的次数称为查找长度

平均查找长度（ASL, Average Search Length）——所有查找过程中进行关键字的比较次数的期望（平均值）

顺序查找

```

typedef struct
{
    KeyType key;
    OtherType other_data;
}RecordType;
typedef struct
{
    RecordType r[LIST_SIZE+1];
    /* r[0]为工作单元 */
    int length;
}RecordList;
int SeqSearch(RecordList l, KeyType k)
/*不用"监视哨"法，在顺序表中查找关键字等于k的元素*/
{
    int i;
    i=l.length;
    while (i>=1&&1.r[i].key!=k) i--;
    if (i>=1)
        return(i);
    else
        return (0);
}
//倒序遍历查找
//监视哨查找

```

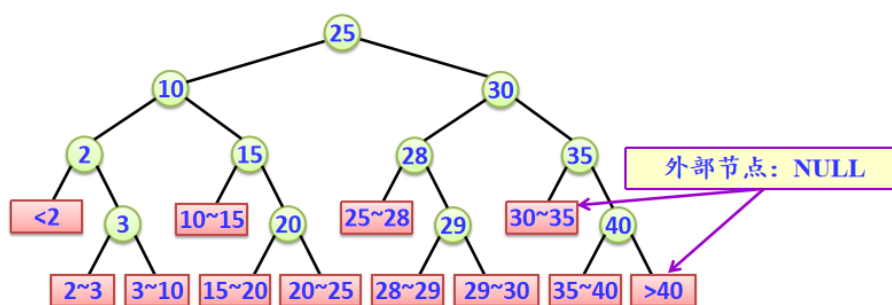
```
int SeqSearch(RecordList l, KeyType k)
/*在顺序表l中顺序查找其关键字等于k的元素，若找到，则函数值为该元素在表中的位置，否则为0*/
{
    int i;
    l.r[0].key=k;
    i=l.length;
    while (l.r[i].key!=k) i--;
    return(i);
}
//该方法通过队头元素监视哨去掉了最终判断是否越界的代码
```

二分查找（顺序表）

```
int BinSearch(RecType R[], int n, KeyType k)
{
    int low = 0, high = n - 1, mid;
    while (low <= high)
    { //当前区间存在元素时循环
        mid = (low + high) / 2;
        if (R[mid].key == k) //查找成功返回其逻辑序号mid+1
            return mid;
        if (k < R[mid].key) //继续在R[low..mid-1]中查找
            high = mid - 1;
        else
            low = mid + 1; //继续在R[mid+1..high]中查找
    }
    return -1;
}
```

二分查找也可以使用树：

带外部节点的判定树



成功二分查找：若查找给定值为20的元素，依次与25、10、15、20元素比较，共比较4次。

不成功二分查找：若查找给定值为32的元素，依次与25、30、35元素比较，关键字共比较3次。

折半查找的时间复杂度= $O(\log_2 n)$

分块查找

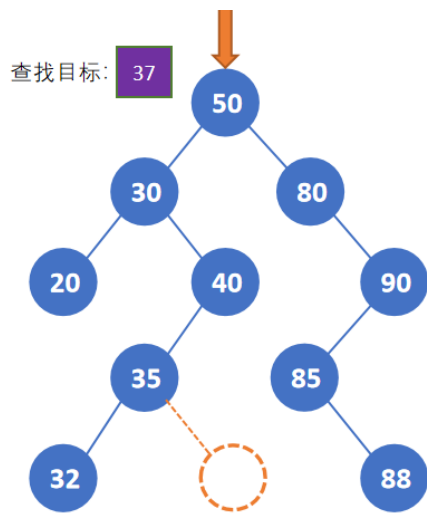
- ☺ 分块查找法要求将列表组织成以下索引顺序结构：
- ☺ 首先将列表分成若干个块（子表）。
 - 🎵 一般情况下，块的长度均匀，最后一块可以不满。
 - 🎵 每块中元素任意排列，即块内无序，但块与块之间有序。
- ☺ 构造一个索引表。
 - 🎵 其中每个索引项对应一个块并记录每块的起始位置，和每块中的最大关键字（或最小关键字）。
 - 🎵 索引表按关键字有序排列。

三类查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构要求	有序表 无序表	有序表	分块有序表
可用存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

二叉排序树

二叉排序树又称二叉查找树(BST,Binary Search Tree)，它是一种特殊的二叉树。其定义为：①若它的左子树非空，则左子树上所有结点的值均小于根结点的值②若它的右子树非空，则右子树上所有结点的值均大于或等于根结点的值③它的左右子树也分别为二叉排序树。



- 二叉排序树可看做是一个有序表，在二叉排序树bt上查找关键字为k的记录，
- 成功时返回该结点指针，否则返回NULL

lchild	key	rchild
--------	-----	--------

```
typedef struct node{
    KeyType key; /*关键字的值*/
    struct node *lchild,*rchild; /*左右孩子指针*/
}BSTNode,*BSTree;

BSTree SearchBST(BSTree bt, KeyType k) {
    if (bt==NULL || bt->key==k) //递归出口
        return bt;
    if (k<bt->key)
        return SearchBST(bt->lchild, k); //在左子树中递归查找
    else
        return SearchBST(bt->rchild, k); //在右子树中递归查找
}
```

```
void InsertBST(BSTree *bst, KeyType key) {
    /*若在二叉排序树中不存在关键字等于key的元素，插入该元素*/
    BiTree s;
    if (*bst==NULL) { /*递归结束条件*/
        s=(BSTree)malloc(sizeof(BSTNode)); /*申请新的结点s*/
        s->key=key;
        s->lchild=NULL;
        s->rchild=NULL;
        *bst=s;
    }
    else if (key < (*bst)->key)
        InsertBST(&(*bst)->lchild, key); /*将s插入左子树*/
    else if (key > (*bst)->key)
        InsertBST(&(*bst)->rchild, key); /*将s插入右子树*/
}
```

```
void CreateBST(BSTree *bst) {
    /*从键盘输入元素的值，创建相应的二叉排序树*/
    KeyType key;
    *bst=NULL;
    scanf("%d", &key);
    while (key!=ENDKEY) { /*ENDKEY为自定义常数*/
        InsertBST(bst, key);
        scanf("%d", &key);
    }
}
```

AVL树

平衡二叉排序树又称为AVL树（Adelson-Velskii and Landis）——树上任一结点的左子树和右子树的高度之差不超过1结点

平衡因子=左子树高-右子树高

若树高为h，则最坏情况下，查找一个关键字最多需要对比h次，即查找操作的时间复杂度不可能超过O(h)

由于平衡二叉树的特性，n个结点平衡二叉树的最大深度为O(log₂n)故平衡二叉树的平均查找长度为O(log₂n)

哈希法

哈希法又称散列法、杂凑法以及关键字地址计算法等，相应的表称为哈希表

特点：数据元素的关键字与其存储地址直接相关

具有相同哈希值的两个不同的关键字称为该函数的同义词

构造方法

$$\begin{array}{r} 5864 \\ 4220 \\ 04 \\ \hline 10088 \end{array}$$

$H(\text{key}) = 0088$

移位叠加

$$\begin{array}{r} 5864 \\ 0224 \\ 04 \\ \hline 6092 \end{array}$$

$H(\text{key}) = 6092$

边界叠加

例：关键字为：

0442205864

哈希地址位数为4

- 构造方法
 - 将关键字分割成位数相同的几部分
 - 然后取这几部分的叠加和（舍去进位）做哈希地址
 - 移位叠加：将分割后的几部分低位对齐相加
 - 边界叠加：从一端沿分割界来回折送，然后对齐相加
- 适于关键字位数很多，且每一位上数字分布大致均匀情况

关键字的哈希地址采用如下方式计算得到

关键字被某个不大于哈希表表长 m 的数 p 除后所得余数

即： $\text{Hash}(\text{key}) = \text{key} \bmod p \quad (p \leq m)$

特点：简单、常用，可与上述几种方法结合使用

p 的选取很重要（ p 选的不好容易产生同义词）通常选择一个不大于 m 的素数

例如：给定关键字序列 {121,123,136,214,286,334}

取hash函数为： $\text{Hash}(\text{key}) = \text{key} \bmod 7$

得到相应的hash函数值：{2, 4, 3, 4, 6, 5}

例如：有一个解放后出生人口调查表，每个记录包含：年份、人数等数据项，其中年份为关键字

哈希地址	01	22
年份	1949	1970
人数			

$\text{Hash}(\text{key}) = \text{key} + (-1948)$ 这样就可以方便地存储和查找1948年后任一年的记录

构造方法：首先对关键字进行分析，取关键字或关键字的某个线性函数作为哈希地址，即：

$\text{Hash}(\text{key}) = a \cdot \text{key} + b$ (a, b 为常数)

最简单的情况： $\text{Hash}(\text{key}) = \text{key}$

直接定址法的特点

①地址集合与关键字集合大小相等

②若关键字彼此不冲突，则哈希地址不会发生冲突

实际应用中很少使用这种形式的哈希函数，适于关键字事先可预知的情况

例：有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

①	②	③	④	⑤	⑥	⑦	⑧	
8	1	3	4	6	5	3	2	n=80, d=8, r=10, s=2
8	1	3	7	2	2	4	2	分析：① 只含数字8;
8	1	3	8	7	4	2	2	② 只含数字1;
8	1	3	0	1	3	6	7	③ 只含数字3、4;
8	1	3	2	2	8	1	7	⑧ 只含数字2、7、5;
8	1	3	3	8	9	6	7	④⑤⑥⑦数字分布近乎随机
8	1	3	6	8	5	3	7	所以：取④⑤⑥⑦任意两位（或其中两
8	1	4	1	9	3	5	5	位与另两位的叠加）作哈希地址

例如：可以取第4、6两位组成的2位十进制数作为每个数据的哈希地址，则图中列出的关键字的哈希地址分别为：45, 72, 84, 03, 28, 39, 51, 65, 13

构造方法：取关键字平方值的中间几位作为哈希地址，即：
Hash (key) = “key² 的中间几位”
其中：所取的位数由哈希表的大小确定
求关键字的平方值的目的是：“扩大差别”和“贡献均衡”
关键字的各位都对平方值的中间几位有所贡献
Hash值中应该包含有各位的信息
适于不知道全部关键字情况

关键字	内部编码	内部编码的平方值	H(k)关键字的哈希地址
KEYA	11052501	122157778355001	778
KYAB	11250102	126564795010404	795
AKEY	01110525	001233265775625	265
BKEY	02110525	004454315775625	315

取关键字的随机函数值作哈希地址即：

Hash(key) = rand(key)

适用于关键字长度不等的情况

冲突处理

**开放定址之
线性探测法**
线性探测法的数学递推描述公
式为：
 $d_0 = h(k)$
 $d_i = (d_{i-1} + 1) \bmod m \quad (1 \leq i \leq m-1)$

					60	17	29	38		
0	1	2	3	4	5	6	7	8	9	10
H(38)=38 mod 11=5 冲突										
H1=(5+1) mod 11=6 冲突										
H2=(5+2) mod 11=7 冲突										
H3=(5+3) mod 11=8 不冲突										

- 例：表长为11的哈希表中已填入三条记录
- 表中记录的关键字分别为：17, 60, 29
 - 已知哈希函数为：Hash(key) = key mod 11
 - 请将第4个关键字38填入表中（冲突采用线性探测法处理）
- 线性探测法： $d_i = 1, 2, 3, \dots, m-1$

链地址法

- 将所有哈希地址为*i*的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第*i*个单元中，因而查找、插入和删除主要在同义词链中进行。
- 链地址法的优点：
 - ◆ 冲突处理简单，不会像开放定址法那样发生关键字堆积现象
 - ◆ 不会产生溢出（因为链表可扩展）
 - ◆ 适用于无法确定哈希表长度的情况
 - ◆ 删除结点的操作易于实现

Para.9 排序

基本信息

将一个数据元素（记录）的任意序列，重新排列成一个按关键字有序的序列，称为排序。

设：给定一个包含*n*个记录的序列 { *R*₁, *R*₂, ..., *R*_{*n*} }其相应的关键字序列为 { *K*₁, *K*₂, ..., *K*_{*n*} }这些关键字之间存在偏序关系（可相互比较）： $K_{p1} \leq K_{p2} \leq \dots \leq K_{pn}$

按此偏序关系将上式记录序列重新排列为：{ *R*_{*p*1}, *R*_{*p*2}, ..., *R*_{*p**n*} }将上述操作称为排序。

稳定排序和不稳定排序：

假设 $K_i = K_j (1 \leq i \leq n, 1 \leq j \leq n, i \neq j)$ ，若在排序前的序列中 *R*_{*i*} 领先于 *R*_{*j*} (即 $i < j$)，经过排序后得到的序列中 *R*_{*i*} 仍领先于 *R*_{*j*}，则称所用的排序方法是稳定的；反之，当相同关键字的领先关系在排序过程中发生变化者，则称所用的排序方法是不稳定的。

插入排序

```
//直接插入排序
void InsertSort(int A[], int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++) //将各元素插入已排好序的序列中
        if (A[i] < A[i - 1]){ //若A[i]关键字小于前驱
            temp = A[i]; //用temp暂存A[i]
            for(j=i-1; j>=0 && A[j]>temp; --j) //检查所有前面已排好序的元素
                A[j + 1] = A[j]; //所有大于temp的元素都向后挪位
            A[j + 1] = temp; //复制到插入位置
        }
}

//带哨哨
void InsSort(RecordType r[], int length)
/* 对记录数组r做直接插入排序，length为数组中待排序记录的数目 */
{
    int i, j;
    for (i=2; i<=length; i++)
    {
        r[0]=r[i]; /*将待插入记录存放到监视哨r[0]中*/
```

```

        j=i-1;
        while ( r[0].key< r[j].key ) /* 寻找插入位置 */
        {
            r[j+1]= r[j];
            j=j-1;
        }
        r[j+1]=r[0]; /*将待插入记录插入到已排序的序列中*/
    }
} /* InsSort */

```

```

//折半插入排序
void BinSort (RecordType r[], int length)
/*对记录数组r进行折半插入排序，length为数组的长度*/
{
    int i,j;
    RecordType x;
    int low,high,mid;
    for ( i=2; i<=length ; ++i )
    {
        x= r[i];
        low=1; high=i-1;
        while (low<=high ) /* 确定插入位置*/
        {
            mid=(low+high) / 2;
            if ( x.key< r[mid].key )
                high=mid-1;
            else
                low=mid+1;
        }
        for ( j=i-1 ; j>= low; --j ) r[j+1]= r[j];/* 记录依次向后移动 */
        r[low]=x; /* 插入记录 */
    }
} /*BinSort*/

```

希尔排序

希尔排序：先将待排序表分割成若干形如 $L[i, i + d, i + 2d, \dots, i + kd]$ 的“特殊”子表(子序列)，对各个子表分别进行直接插入排序。缩小增量 d ，重复上述过程，直到 $d=1$ 为止。当序列达到基本有序时，对其执行一次直接插入排序。

```

void ShellInsert(RecordType r[], int length, int delta)
/*对记录数组r做一趟希尔插入排序，length为数组的长度,delta 为增量*/
{
    int i,j;
    for(i=1+delta;i<= length; i++) /* 1+delta为第一个子序列的第二个元素的下标 */
        if(r[i].key < r[i-delta].key)
        {
            r[0]= r[i]; /* 备份r[i] (不做监视哨) */
            for(j=i-delta; j>0 &&r[0].key < r[j].key; j-=delta)
                r[j+delta]= r[j]; //后移,查找插入位置
            r[j+delta]= r[0]; //插入
        }
}

```

```

}/*ShellInsert*/
void ShellSort(RecordType r[], int length, int delt[], int n)
/*对记录数组r做希尔排序，length为数组r的长度，
delta 为增量数组，n为delt[]的长度 */
{
    int i;
    for(i=0 ; i<=n-1; ++i)
        ShellInsert(r, length, delt[i]);
}

```

其他排序方式

```

//冒泡排序
void BubbleSort(RecordType r[], int length)
/*对记录数组r做冒泡排序，length为数组的长度*/
{
    int n, i, j;
    int change;
    RecordType x;
    n = length;
    change = TRUE;
    for (i = 1; i <= n - 1 && change; ++i)
    {
        change = FALSE;
        for (j = 1; j <= n - i; ++j)
            if (r[j].key > r[j + 1].key)
            {
                x = r[j];
                r[j] = r[j + 1];
                r[j + 1] = x;
                change = TRUE;
            }
    }
} /* BubbleSort */

```

```

//快速排序
int QKPass(RecordType r[],int left,int right)
/*对记录数组r 中的r[left]至r[right]部分进行一趟排序，并得到基准的位置，使得排序后的结果满足其
之后（前）的记录的关键字均不小于（大于）于基准记录*/
{
    RecordType x;
    int low,high;
    x= r[left]; /* 选择基准记录*/
    low=left;
    high=right;
    while ( low<high )
    {
        while (low< high && r[high].key>=x.key )
            /* high从右到左找小于x.key的记录 */
            high--;
    }
}

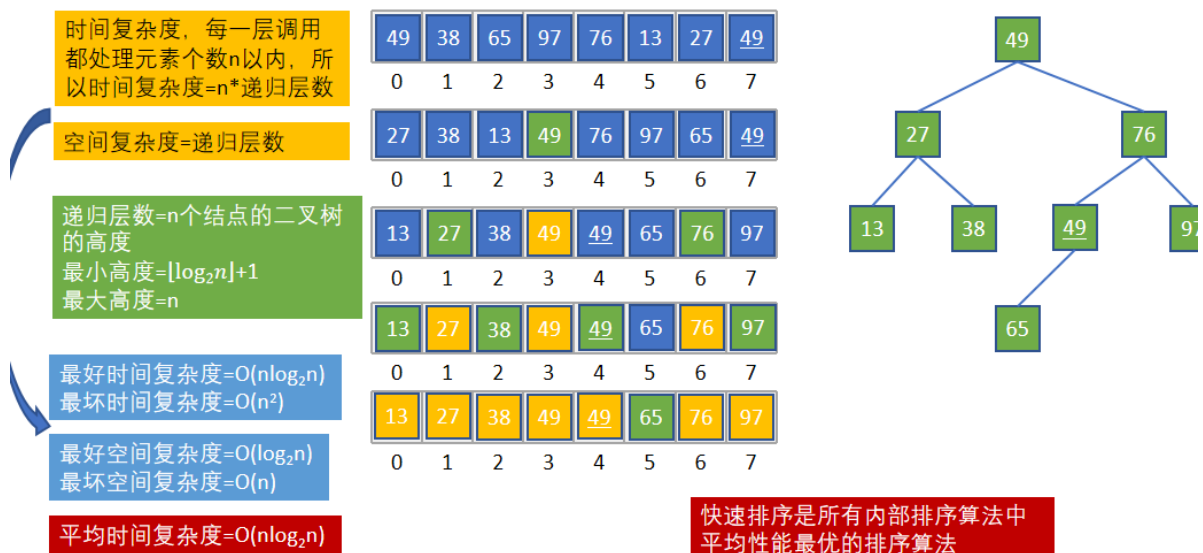
```

```

    if ( low < high )
    {
        r[low]= r[high];
        low++;
    } /* 找到小于x.key的记录, 则进行交换*/
    while (low<high && r[low].key<x.key )
    /* low从左到右找大于x.key的记录 */
        low++;
    if ( low<high )
    {
        r[high]= r[low];
        high--;
    } /* 找到大于x.key的记录, 则交换*/
    }
    r[low]=x; /*将基准记录保存到low=high的位置*/
    return low; /*返回基准记录的位置*/
} /* QKPass */

void QKSort(RecordType r[],int low, int high )
/*对记录数组r[low..high]用快速排序算法进行排序*/
{
    int pos;
    if(low<high)
    {
        pos=QKPass(r, low, high); /*调用一趟快速排序, 将枢轴元素
        为界划分两个子表*/
        QKSort(r, low, pos-1); /*对左部子表快速排序*/
        QKSort(r, pos+1, high); /*对右部子表快速排序*/
    }
}

```



https://www.bilibili.com/video/BV18T4y197xL/?spm_id_from=333.337.search-card.all.click&vd_source=4e2c42e50717d2840103a5c4b3f7092f

```

//简单选择排序
void SelectSort(RecordType r[], int length)
/*对记录数组r做简单选择排序, length为数组的长度*/
{

```

```
int i,j,k;
int n;
RecordType x;
n=length;
for ( i=1 ; i<= n-1; ++i)
{
    k=i;
    for ( j=i+1 ; j<= n ; ++j)
        if ( r[j].key < r[k].key )
            k=j;//找位置
        if(k!=i)
        {
            x= r[i];
            r[i]= r[k];
            r[k]=x;
        }//交换
    }
} /* SelectSort */
```

- 对 n 个记录进行简单选择排序, 所需进行的关键字间的比较次数为:

$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

- **移动**记录的次数, 最小值为 0, 最大值为 $3(n-1)$ 。

空间复杂度= $O(1)$

时间复杂度= $O(n^2)$

稳定性:不稳定

原始序列

996

第一趟排序后序列

6 9 9

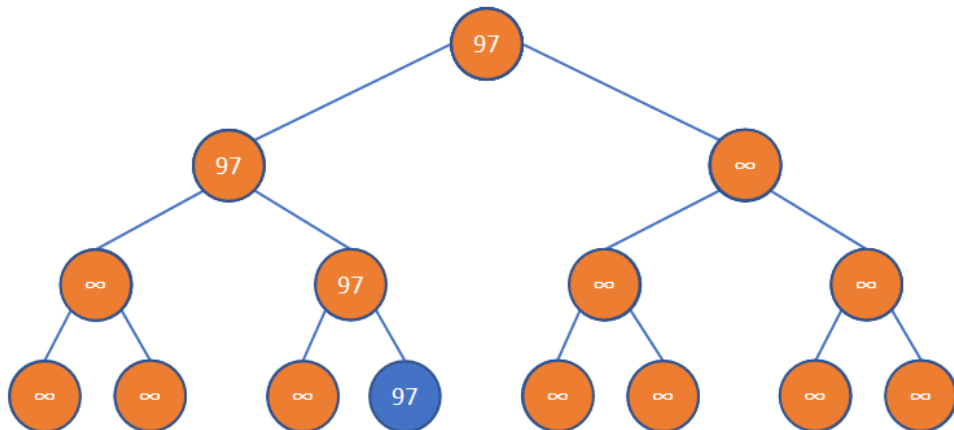
第二趟排序后序列

6 9 9

树形选择排序

待排
序列

49	38	65	97	76	13	27	<u>49</u>
----	----	----	----	----	----	----	-----------



排序
结果

13 27 38 49 49 65 76 97

在树型选择排序中，被选中的关键字都是走了一条由叶子结点到根结点的比较的过程,由于含有n个叶子节点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

则在树型选择排序中，每选择一个小关键字需要进行 $\lfloor \log_2 n \rfloor$ 次比较

因此其时间复杂度为： $O(n \log_2 n)$

移动记录次数不超过比较次数，故总的算法时间复杂度为 $O(n \log_2 n)$ 。

稳定性： 稳定，可通过代码对当左右子树相等情况规定优先左子树。

空间复杂度： $O(n)$ ，n个叶子结点构造完全二叉树的所需空间（ $2n$ 或者 $2n-1$ 取决于度为1的结点数量）。

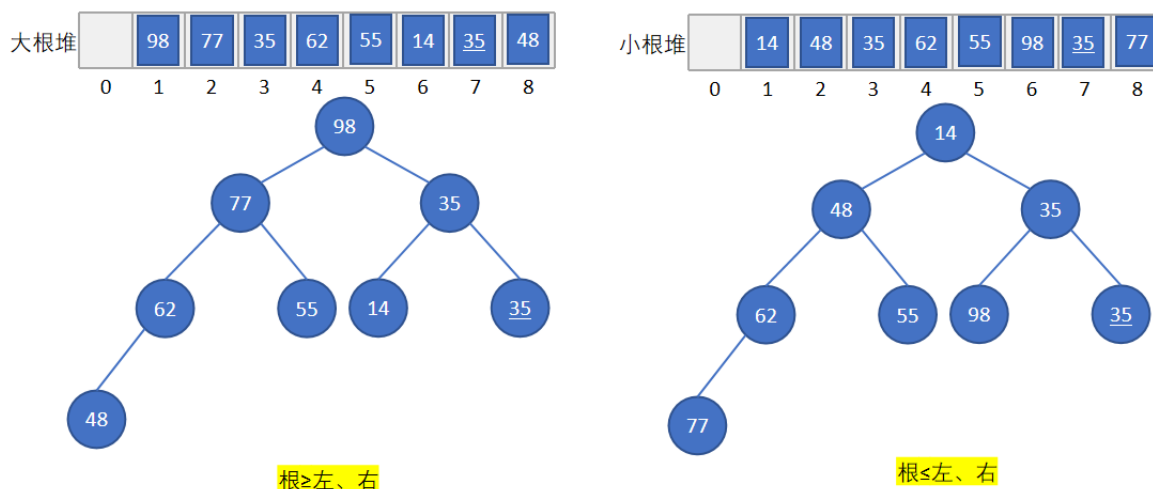
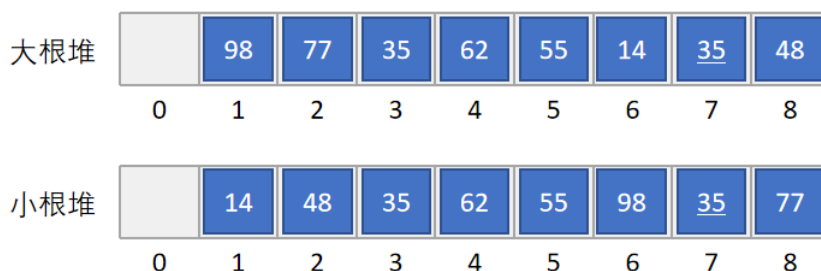
堆排序

堆排序是对树型选择排序的进一步改进，弥补树形选择排序占用空间多的问题。采用堆排序时，只需要一个记录大小的辅助空间。

堆是什么？

若n个待排记录序列R[1...n]满足下面某一条性质，则称为堆(Heap)：

- ① 若满足： $R(i).key \geq R(2i).key$ 且 $R(i).key \geq R(2i+1).key$ ($1 \leq i \leq \lfloor n/2 \rfloor$)——大根堆(大顶堆)
- ② 若满足： $R(i).key \leq R(2i).key$ 且 $R(i).key \leq R(2i+1).key$ ($1 \leq i \leq \lfloor n/2 \rfloor$)——小根堆(小顶堆)



基于大根堆求最大就是取堆顶（根、数组第一个元素）

基于小根堆求最小就是取堆顶（根、数组第一个元素）

```
//建立堆
void crt_heap(RecordType r[], int length)
/*对记录数组r建堆，length为数组的长度*/
```

```

{
    int i,n;
    n= length;
    for ( i=n/2; i >= 1; --i) /* 自第[n/2]个记录开始进行筛选建堆 */
        sift(r,i,n);
}

void sift(RecordType r[], int k, int m)
/* 假设 r[k..m]是以 r[k]为根的完全二叉树，且分别以 r[2k]和 r[2k+1]为根的左、右子树为大根堆，
调整r[k]，使整个序列r[k..m]满足堆的性质 */
{
    RecordType t; int i,j; int x; int finished;
    t=r[k];/* 暂存"根"记录r[k] */
    x=r[k].key;
    i=k;
    j=2*i;
    finished=FALSE;
    while( j<=m && !finished ) {
        if ( j<m && r[j].key< r[j+1].key )
            j=j+1; /* 若存在右子树，且右子树根的关键字大，
                    则沿右分支"筛选" */
        if (x>= r[j].key)
            finished=TRUE; /* 筛选完毕 */
        else {
            r[i] = r[j]; i=j; j=2*i;
        } /* 继续筛选 */
    }
    r[i] =t; /* r[k]填入到恰当的位置 */
}

```

```

void HeapSort(RecordType r[],int length)
/* 对r[1..n]进行堆排序，执行本算法后，r中记录按关键字由大到小有序排列 */
{
    int i,n;
    RecordType b;
    crt_heap(r, length);
    n= length;
    for ( i=n ; i>= 2; --i)
    {
        b=r[1]; /* 将堆顶记录和堆中的
                  最后一个记录互换 */
        r[1]= r[i];
        r[i]=b;
        sift(r,1,i-1); /* 进行调整，
                        使r[1..i-1]变成堆 */
    }
} /* HeapSort */

```


合并

```
void Merge ( RecordType r1[], int low, int mid, int high, RecordType r[])
{
    /*已知r1[low..mid]和r1[mid+1..high]分别按关键字有序排列，将它们合并成一个有序序列，存
    放在r[low..high] */
    i=low; j=mid+1; k=low;
    while ( (i<=mid)&&(j<=high) ) {
        if ( r1[i].key<=r1[j].key ) {
            r[k]=r1[i] ; ++i;
        }
        else {
            r[k]=r1[j] ; ++j;
        }
        ++k ;
    }
    while ( i<=mid ){
        r[k] =r1[i]; k++; i++; }
    while ( j<=high){
        r[k] =r1[j]; k++; j++;
    }
} /* Merge */

void MergeSort (RecordType r1[], int low, int high, RecordType r[]) {
    /* r1[low..high]经过排序后放在r[low..high]中, r2[low..high]为辅助空间 */
    RecordType *r2;
    r2=(RecordType*)malloc(sizeof(RecordType)*(high-low+1));
    if ( low==high ) r[low]=r1[low];
    else{
        mid=(low+high)/2;
        MergeSort(r1, low, mid, r2);
        MergeSort(r1, mid+1, high, r2);
        Merge (r2, low, mid, high, r);
    }
    free(r2);
} /* MergeSort */
```

基数排序

```
typedef struct node{//含next指针的待排元素
    int data;
    node *next;
}TNode;

typedef struct{//首尾指针组合
    node *front;
    node *rear;
}TPointer;

// 根据数组R 构建带头结点的单链表
TNode* build_list(int R[], int n){
    int i; TNode* p, ph;
    ph = (TNode*)malloc(sizeof(TNode)); // 判空略
    ph->next = NULL;
    for(i = 0; i < n; ++i ){
        p = (TNode*)malloc(sizeof(TNode)); // 判空略
        p->data = R[i];
        p->next = ph->next;
        ph->next = p;
    }
    return ph;
}
```

```
void radix_sort(int* R, int n){
    int i; TNode* p; TPointer Q[RADIX]; //表示每个关键字
    取值个数 (基数)
    int max_val = findmax(R, n); // 求最大值
    TNode* ph = build_list(R, n); // 构建链表
    for( i = 0; max_val; max_val/=10, ++i){
        dispatch(ph, Q, i);
        collect(ph, Q); // 分配收集
    } // 迭代次数由关键字位数决定
    p = ph->next; // 将排序结果写入数组R
    for( i = 0; p; ++i){
        R[i] = p->data;
        p = p->next;
    }
    destroy(ph); // 销毁辅助链表
}
```

```

void dispatch (TNode * ph, TPointer Q[], int d){
    int i, idx; TNode * p = NULL;
    for( i = 0; i < RADIX; ++i ){
        Q[i].front = NULL; Q[i].rear = NULL; }
    p = ph->next; // 取原始链队列中第一个结点
    if(p){ ph->next = p->next; p->next = NULL; }
    while(p){
        idx = p->data; // 取出*p中的第d位数字
        for(i = 0; i < d; ++i) idx = idx / RADIX;
        idx = idx % 10;
        if( Q[idx].front == NULL ){ // 将*p分配到相应队列中
            Q[idx].front = p; Q[idx].rear = p; }
        else{
            Q[idx].rear->next = p; Q[idx].rear = p; }
        p = p->next; // 取原始链队列中下一个结点
        if(p){ ph->next = p->next; p->next = NULL; }
    }
}

```

```

void collect(TNode* ph, TPointer * Q){
    int i; TNode* p;
    for(i = 0; !Q[i].front; ++i); // 找出Q数组中第一个
    指向非空队列的元素
    // 将其链接到新的链表中
    ph->next = Q[i].front; p = Q[i].rear; i++;
    // 寻找其余非空队列, 并将其顺序链接到主队列
    for(; i < RADIX; ++i){
        if(Q[i].front){
            p->next = Q[i].front; p = Q[i].rear;
        }
    }
    p->next = NULL; // 修改链表尾结点
}

```