1.概论

2.线性表 List

2.1概念

逻辑结构特征：

2.2链表

2.3.2循环链表

定义：链表中最后一个结点的指针指向头结点——从链表中的任一结点出发均可访问到其它结点。

循环判断条件：L->next=L

操作：

1. 合并循环链表

```
void circularMerge(LinkList& L, LinkList& R) {
    LinkList p = L->next;
    L->next = R->next->next;     //L的下一个节点为R的第二个节点
    R->data = R->next->data;     //将R的第一个节点数据存到头指针中，释放第一个节点
    free(R->next);
    R->next = p;
}
```

2. josepher问题

```
void josephus(LinkList& L,int n, int k, int m) {
    LinkList p=L,temp;
    int i=0;
    while (i < k) {
        p = p->next;
        i++;
    }
    for (int j = 0; j < n; j++) {
        i = 0;
        while (i < m-1)                  //得到第m个数的前一个节点指针
        {
            p = p->next;
            if (p != L)
                i++;
        }
        if (p->next == L)
            p = p->next;
        cout << p->next->data << " ";
        temp = p->next;
        p->next = p->next->next;
        free(temp);
```

```
    }
}
```

2.3.3双向链表

定义：链表的每个结点包含两个指针域，一个指向后继，一个指向前驱。

操作：

1. 初始化

```
void initDList(DLinkList& L) {
    L = (DLinkList)malloc(sizeof(DNode));
    if (!L) {
        cout << "init error";
        return;
    }
    L->data = 0;
    L->next = L;
    L->front = L;
}
```

2. 在双向链表L中的第i个元素之前插入一个数据元素e。

```
void insertDList(DLinkList& L,int n,int e) {
    DLinkList p = L,t;
    for (int i = 0; i < n; i++) {
        p = p->next;
        if (!p)
            return;
    }
    initDList(t);
    t->data = e;
    t->front = p->front;
    p->front->next = t;
    t->next = p;
    p->front = t;
}
```

3. 删除第i个元素

```
void deleteDList(DLinkList& L, int i) {
    DLinkList p = L;
    int k = 0;
    while (p->next&&k<i)
    {
        p = p->next; k++;
    }
    p->front->next = p->next;
    if (p->next)
        p->next->front = p->front;
    free(p);
}
```

2.3栈和队列

定义：栈和队列都是限制在“**端点**”进行插入操作和删除操作的线性表。

操作位置i	线性表	栈	队列
插入	i∈[1, n+1]	**i=**n+1**	**i=**n+1**
删除	i∈[1, n]	**i**=n**	**i**=1**

2.3.1栈

定义:栈是一种先进后出（LIFO）的线性表。

判断空条件：top=base

顺序栈操作：

1. 定义

```
typedef struct Stack {
    int* base;
    int* top;
    int stacksize;
}Stack;
```

2. 初始化

```
void initStack(Stack& S,int n) {
    S.base = (int*)malloc(n*sizeof(int));
    S.top = S.base;
    S.stacksize = n;
}
```

3. 进栈

```
void push(Stack& S, int e) {
    if (S.top - S.base > S.stacksize)
    {
        int* new_Stack = (int*)realloc(S.base,2 * S.stacksize *
sizeof(int));
        if (!new_Stack)
            return;
        S.base = new_Stack;
    }
    *(S.top++) = e;
}
```

4. 出栈

```
int pop(Stack& S) {
    if (S.top != S.base)
        return *(--S.top);  //是先--
    else return NULL;
}
```

链栈操作：

1. 定义

```
typedef struct SNode {
    int data;
    struct SNode* next;
}SNode,*LinkStack;
```

2. 初始化

```
void initLinkStack(LinkStack& L,int Data=NULL) {
    L = (LinkStack)malloc(sizeof(SNode));
    if (!L)
        return;
    L->data = Data;
    L->next = NULL;
}
```

3. 进栈

```
void push(LinkStack& L, int e) {
    LinkStack p;
    initLinkStack(p, e);
    p->next = L->next;
    L->next = p;
}
```

4. 出栈

```
int pop(LinkStack& L) {
    int n;
    LinkStack p;
    p = L->next;
    n = p->data;
    L->next = p->next;
    free(p);
    return n;
}
```

在使用栈时应注意：

1. 栈在使用之前必须初始化。
2. 顺序栈的大小——栈太小容易溢出，栈太大浪费空间。
3. 空栈问题。空栈常常是解决问题的起点，同时又是解决问题的终点。

2.3.2队列

定义：队列是一种先进先出(FIFO)的线性表。

判定为空条件：当front=rear时，队列空。

顺序队列操作：

1. 定义

```
typedef struct Queue {
    int* front;
    int* rear;
    int size;
}Queue;
```

2. 初始化

```
void initQueue(Queue& Q,int n) {
    Q.front = (int*)malloc(n * sizeof(int));
    Q.rear = Q.front;
    Q.size = n;
}
```

3. 入队

```
void inQueue(Queue& Q, int e) {
    if (Q.rear - Q.front == Q.size) {
        int* temp = (int*)realloc(Q.front, 2 * Q.size * sizeof(int));
        if (!temp)
            return;
        Q.front = temp;
        Q.rear = Q.front + Q.size;
        Q.size += Q.size;
    }
    *(Q.rear++) = e;
}
```

4. 出队

```
int outQueue(Queue& Q) {
    if (Q.rear != Q.front)
        return *Q.front++;
}
```

链队列操作：

1. 定义

```
typedef struct QNode {
    int data;
    struct QNode* next;
}QNode,*LinkQueue;
```

2. 初始化

```
void initQueue(LinkQueue &L,int e) {
    L = (LinkQueue)malloc(sizeof(QNode));
    if (!L)
        return;
    L->data = e;
    L->next = NULL;
}
```

3. 入队（由于方便按先后顺序打印，将新入队的元素放在链表最后，会比较方便）

```
void inQueue(LinkQueue& L,int e) {
    LinkQueue p = L,t;
    while (p->next)
        p = p->next;
    initQueue(t, e);
    p->next = t;
}
```

4. 出队

```
int outQueue(LinkQueue& L) {
    LinkQueue p = L->next;
    int n = p->data;
    L->next = p->next;
    free(p);
    return n;
}
```

例题：

1. 输入10进制数n，输出d进制数。

```
int _10toN(int e,int n) {
    int p = 0;
    Stack s;
    initStack(s,10);

    while (e) {
        push(s, e % n);
        e /= n;
    }
    while (!stackIsEmpty(s)) {
        p = p * 10 + pop(s);
    }
    return p;
}
```

2. 先产生n个[1，100]之间的随机整数，然后按照数的产生顺序依次配对输出奇数和偶数(即一次输出1个奇数和1个偶数)，直到奇数或者偶数输出完毕为止。

```
void twoPair(int n) {
    int i = 0;
    int p;
```

```
    Queue sin, dou;
    initQueue(sin,100);
    initQueue(dou, 100);

    for (i = 0; i < n; i++) {
        srand(i);
        p = rand() % 100 + 1;
        if (p % 2)
            inQueue(sin, p);
        else
            inQueue(dou, p);
    }
    i = 1;
    while (!queueIsEmpty(sin) && !queueIsEmpty(dou)) {
        cout << "第" << i++ << "组:" << outQueue(sin) << " " <<
outQueue(dou) << endl;
    }
}
```

2.3.3循环队列

定义： 循环队列是把顺序队列首尾相连，把存储队列元素的表从逻辑上看成一个环，成为循环队列。

判断条件：

- 空：rear=front
- 满： (rear+1)%MAXSIZE=front
- 根据以上判断条件可知，循环队列的最大元素个数为MAXSIZE-1，预留一个元素空间，用于处理队列"满"的情况。

操作：

1. 定义

```
#define MAXSIZE 10
typedef struct Squeue{
    int elem[MAXSIZE];
    int front;
    int rear;
}Squeue;
```

2. 入队

```
void inQueue(Squeue &S,int e) {
    if (IsFull(S)) {
        cout << "Queue is full!" << endl;
        return;
    }
    S.elem[S.rear] = e;
    S.rear = (S.rear + 1) % MAXSIZE;
}
```

3. 出队

```
int outQueue(Squeue& S) {
    if (IsEmpty(S))
        return NULL;
    int n = S.elem[S.front];
    S.front = (S.front + 1) % MAXSIZE;
    return n;
}
```

4. 打印

```
void queueToString(Squeue S) {
    int front = S.front;
    int rear = S.rear;
    while (front != rear) {
        cout << S.elem[front] << " ";
        front = (front + 1) % MAXSIZE;
    }
    cout << endl;
}
```

例题：

1. 有两个进程同时存在于一个程序中。其中第一个进程在屏幕上连续显示字符“-”,与此同时，程序不断检测键盘是否有输入，如果有，就读入用户键入的字符并保存到输入缓冲区中。在用户输入时，键入的字符并不立即回显在屏幕上。当用户键入一个逗号(,)时，表示第一个进程结束，第二个进程从缓冲区中读取那些已键入的字符并显示在屏幕上。第二个进程结束后，程序又进入第一个进程，重新显示字符“-”,同时用户又可以继续键入字符，直到用户输入一个分号(;)键，才结束第一个进程，同时也结束整个程序。

```
void twoProgress() {
    char ch;
    Squeue s;
    while (true) {
        if (_kbhit()) {
            ch = _getch();
            if (ch == *",") {
                while (!IsEmpty(s)) {
                    cout << (char)outQueue(s);
                }
                continue;
            }
            inQueue(s, ch);
            if (ch == *";")
                break;
        }
        Sleep(100);
        cout << "-";
    }
}
```

2. 迷宫问题

   假设迷宫由m行n列构成，有一个入口(1, 1)和一个出口(m，n)。试找出一条从入口通往出口的可行路径(如输出可通行路径方格的坐标序列)。

```
void labyrinth(int m,int n) {
    //结构体表示坐标的数据结构
    typedef struct Site {
        int x;
        int y;
    public:
        Site(int i, int j) {
            x = i; y = j;
        }
        Site() {
            x = 0; y = 0;
        }
    }Site;

    int i, j;//当前坐标位置

    int Map[10][10] = { 0 };//地图，数值为0表示不能通行，1表示可以通行
    for (i = 1; i < 9; i++)
        for (j = 1; j < 9; j++)
            Map[i][j] = 1;

    Map[1][3] = 0;  Map[1][4] = 0;  Map[1][6] = 0;  Map[2][6] = 0;  Map[3]
[1] = 0;
    Map[3][2] = 0;  Map[3][5] = 0;  Map[4][4] = 0;  Map[5][3] = 0;  Map[5]
[6] = 0;
    Map[5][8] = 0;  Map[6][3] = 0;  Map[6][6] = 0;  Map[7][5] = 0;  Map[7]
[6] = 0;
    Map[8][7] = 0;

    //输出地图
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
            cout << Map[i][j] << " ";
        cout << endl;
    }

    LinkStack<Site>  link;//用栈来存储上一个路过的节点坐标
    i = 1, j = 1;
    int k;
    Site temp;

    //循环寻找路径，按照右下左上的顺序
    while (i + j) {
        k = 0;
        if (i == m && j == n)k = 9;
        else if (Map[i][j + 1]==1)k = 2;
        else if (Map[i+1][j] == 1)k = 3;
        else if (Map[i][j - 1] == 1)k = 4;
        else if (Map[i-1][j] == 1)k = 5;
        else if (i==1&&j==1)k = 6;

        //如果当前节点存在除上一个节点之外还有其他节点，则将该节点加入栈，否则不加入。
        //如果直接增加可以前往的下一个节点，会导致有三个方向可通行的节点，在第一次回退之
后从栈中pop出，但不会再push进去的结果。也就是丢失坐标。
        switch (k) {
        case 0:
            Map[i][j] = -1;
```

```
            temp = link.pop(); i = temp.x; j = temp.y;  break;
        case 2:link.push(Site(i, j)); Map[i][j++] = 2; break;
        case 3:link.push(Site(i, j)); Map[i++][j] = 2; break;
        case 4:link.push(Site(i, j)); Map[i][j--] = 2; break;
        case 5:link.push(Site(i, j)); Map[i--][j] = 2; break;
        case 6:cout << "Search Failed" << endl; return;
        case 9:link.push(Site(i, j));
            cout << "Successful Serach,The Route is:" << endl; i = 0; j = 0;
break;
        }
    }
    temp = link.pop();
    cout << "(" << temp.x << "," << temp.y << ")";
    while (!link.isEmpty()) {
        temp = link.pop();
        cout << "<--(" << temp.x << "," << temp.y << ")";
    }

}
```

3. 括号匹配算法

设在表达式中允许包含[]和()两种括号，约定() 或[()]这类的括号匹配格式是正确的，而[(]) 或([()) 或(()]这类的括号匹配格式是不正确的。

```
void bracketMatch() {
    MList<char> p(100);
    cin >> p;//在MList中重载运算符号>>
    p.n = strlen(p.elem);
    LinkStack<char> chstack;
    char ch,temp;
    int i, n = p.n;
    for (i = 0; i < n; i++)
    {
        ch = p.elem[i];
        if (ch == *"[" || ch == *"{"||ch==*"(")
            chstack.push(ch);//左括号就进栈
        if (ch == *"]" || ch == *"}" || ch == *")")
        {
            temp = chstack.pop();//右括号就从栈中pop出上一个左括号，如果匹配则合法
            switch (temp) {
            case * "[":if (ch == *"]")break;
            case * "{":if (ch == *"}")break;
            case * "(":if (ch == *")")break;
            default:cout << "The expression is invalid." << endl; return;
            }
        }
    }
    //如果右括号都匹配，但左括号比右括号多的情况，也不合法。
    if (i == n && !chstack.isEmpty())
    {
        cout << "The expression is invalid." << endl;
        return;
    }
    cout << "The bracket is match!" << endl;
}
```

4. 算数表达式求解

```
void arthimetic() {
    MList<char> list(100);
    LinkStack<char> stack;
    Queue<char> queue;

    cin >> list;
    int n = strlen(list.elem);
    int i = 0;
    char ch,p;
    while (i < n) {
        ch = list.elem[i++];
        if (ch >= 48 && ch <= 57) {
            queue.push(ch);
            continue;
        }
        else if (queue.top() >= 48 && queue.top() <= 57) {
            queue.push(" ");
        }
        if (ch == "(") {
            stack.push(ch);
            continue;
        }
        if (ch == "*" || ch == "/") {
            if (stack.isEmpty()){
                stack.push(ch);
                continue;
            }
            p = stack.top();
            if (p == "(" || p == "+" || p == "-")
                stack.push(ch);
            else
            {
                queue.push(stack.pop());
                stack.push(ch);
            }
            continue;
        }
        if (ch == "+" || ch == "-") {
            if (stack.isEmpty() || stack.top() == "(")
                stack.push(ch);
            else {
                while (!stack.isEmpty() && stack.top() != "("){
                    queue.push(stack.pop());
                }
                stack.push(ch);
            }
            continue;
        }
        if (ch == "#"&&i==n) {
            while (!stack.isEmpty())
                queue.push(stack.pop());
            break;
        }
        else if(i==n){
            cout << "The arthimetic is invalid!!!" << endl;
```

```
            return;
        }
        while (ch == *")")
        {
            p = stack.pop();
            if (p != *"(")
                queue.push(p);
            else
                break;
        }
    }
    queue.toString();

    LinkStack<int> numStack;
    while (!queue.isEmpty()) {
        i = 0;
        while (queue.head() >= 48 && queue.head() <= 57)
        {
            i = i * 10 + (int)(queue.pop()-48);
        }
        if (i) {
            numStack.push(i);
            continue;
        }
        ch = queue.pop();
        int a = 0, b = 0,c=0;
        switch (ch) {
        case * " ":break;
        case * "+":b = numStack.pop(); a = numStack.pop(); c = a + b;
numStack.push(c); break;
        case * "-":b = numStack.pop(); a = numStack.pop(); c = a - b;
numStack.push(c); break;
        case * "*":b = numStack.pop(); a = numStack.pop(); c = a * b;
numStack.push(c); break;
        case * "/":b = numStack.pop(); a = numStack.pop(); c = a / b;
numStack.push(c); break;
        default:cout << "No choice!" << endl; break;
        }
    }
```

5. haoni问题（递归）

```
void move(char x, int n, char z)
{
    cout << "将" << x << "柱上的第" << n << "块移动到" << z << "柱上" << endl;
}

void hanoi(char x, int n, char y, char z) {
    if (n == 1)
        move(x, n, z);
    else {
        hanoi(x, n - 1, z, y);
        move(x, n, z);
        hanoi(y, n - 1, x, z);
    }
}
```

6. 背包问题

   设有n个物品，每个物品的重量为wi, i=1, 2, ..., n。试从这n个物品中选取若干个，使其重量之和 = 背包的容量T。

```
bool Knapsack(int T, int n,int w[]) {
    if (T == 0)return 1;
    if (T < 0 || n < 1)return 0;
    if (Knapsack(T - w[n - 1], n - 1, w)) {
        cout << w[n - 1] << " "; return 1;
    }
    return Knapsack(T, n - 1,w);
}

void bag() {
    int n = 7;
    int w[7] = { 27,24,17,51,28,32,63, };
    int T = 100;
    Knapsack(T, n, w);
}
```

7. 查找倒数第k个位置上的结点

```
LinkList LinkSearch(LinkList L, int k)
{   k0=1; p=L->next, q=p;  //q为所查结点
    while(p)
    {   if(k0<=k) k0++;
        else q=q->next;
        p=p->next;
    }
    return q;
}
```

2.4串（不考）

2.5数组与广义表

2.5.1定义

数组是一组偶对(下标值，数据元素值)的集合。在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推。

数组**是由n(n>1)个具有相同数据类型的数据元素a1，a2，...，an组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。

- 数组中的数据元素具有相同数据类型。
- 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
- 数组中的数据元素个数是固定的。

2.5.2数组的顺序表示和实现

数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就**不再发生变化**。因此，一般都是采用**顺序存储**的方法来表示数组。

问题：计算机的内存结构是**一维(线性)地址结构**，对于多维数组，将其存放(映射)到内存一维结构时，有个**次序约定问题**。即必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放到内存中。

二维数组是最简单的多维数组，以此为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。

通常有两种顺序存储方式：

1. 行优先顺序：将数组元素按行排列，第i+1个行向量紧接在第i个行向量后面。对二维数组，按行优先顺序存储的线性序列为：

   a11,a12,...,a1n, a21,a22,...a2n ,....., am1,am2,...,amn

   PASCAL和C是按行优先顺序存储的。

2. 列优先顺序：将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后，对二维数组，按列优先顺序存储的线性序列为：

   a11,a21,...,am1， a12,a22,...am2，....., an1,an2,...,anm

   FORTRAN是按列优先顺序存储的。

地址计算公式：

1. 以“行优先顺序”存储
   - 设有二维数组A=(aij)m×n，若每个元素占用的存储单元数为l(个)，LOC[a11]表示元素a11的首地址，即数组的首地址。

     二维数组中任一元素aij的(首)地址是：

     LOC[aij]=LOC[a11]+[(i-1)n+(j-1)]×l
   - 对于三维数组A=(aijk)m×n×p，若每个元素占用的存储单元数为l(个)，LOC[a111]表示元素a111的首地址，即数组的首地址。以“行优先顺序”存储在内存中。

     三维数组中任一元素aijk的（首）地址是：

     LOC[aijk]=LOC[a111]+[(i-1)×n×p+(j-1)×p+(k-1)]×l
   - 推而广之，对n维数组A=(aj1j2...jn) ，若每个元素占用的存储单元数为l(个)，LOC[a11 ...1]表示元素a11 ...1的首地址。则 以“行优先顺序”存储在内存中。

     n维数组中任一元素aj1j2...jn的(首)地址是：

     LOC[aj1j2...jn]=LOC[a11 ...1]+[(j1-1) (b2×...×bn)+ (j2-1)×(b3×...×bn) + ... + (j(n-1)-1)bn+(jn-1)]×l
2. 以“列优先顺序”存储
   - 和以上类似

2.5.3矩阵的压缩存储

对于高阶矩阵，若其中非零元素呈某种规律分布或者矩阵中有大量的零元素，若仍然用常规方法存储，可能存储重复的非零元素或零元素，将造成存储空间的大量浪费。对这类矩阵进行压缩存储：

- 多个相同的非零元素只分配一个存储空间；
- 零元素不分配空间。

2.5.3.1特殊矩阵

特殊矩阵：是指非零元素或零元素的分布有一定规律的矩阵。

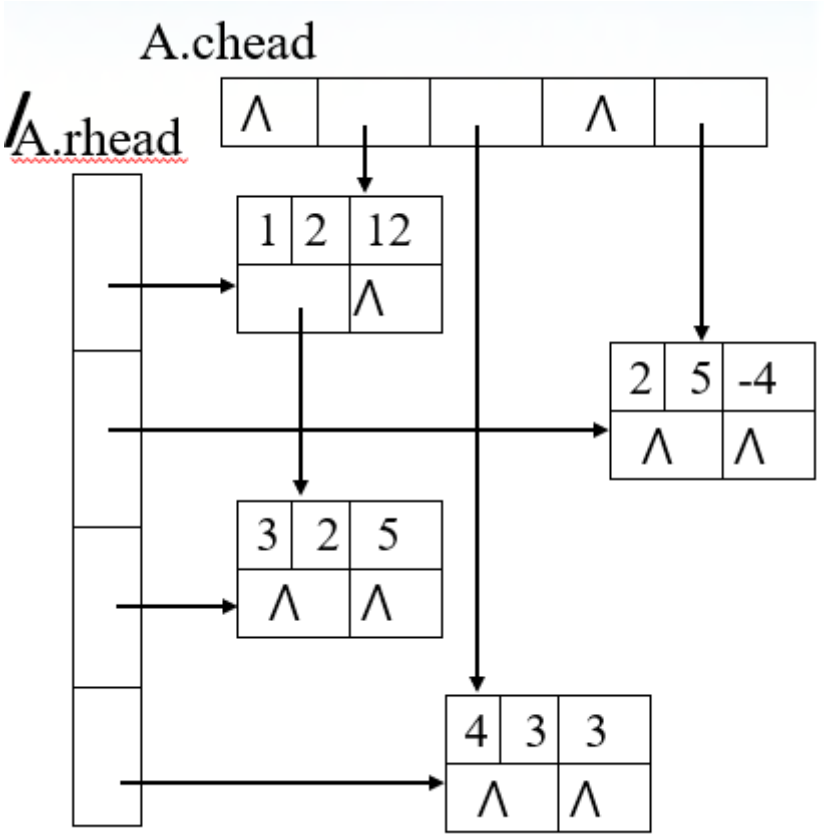
1. **对称矩阵**：

- 一个n阶方阵A=(aij)n×n中的元素满足性质：aij=aji 1≤i,j≤n且i≠j
- 元素关于主对角线对称，让每一对**对称元素**aij和aji(i≠j)分配一个存储空间，则n×n个元素压缩存储到n(n+1)/2个存储空间，能节约近一半的存储空间。
- **规定**：按照"**行优先顺序**"存储**下三角形**中的元素
- 设用一维数组(向量)sa[n(n+1)/2]存储n阶对称矩阵。为了便于访问，必须找出矩阵A中的元素的下标值（i,j）和向量sa[k]的下标值k之间的对应关系。
- 若i>=j，在下三角形中，前i-1行中共有1+2+...+(i-1)=i×(i-1)/2个元素，k=i×(i-1)/2+j-1；若i<j，在上三角形中，k=j×(j-1)/2+i-1。
- 称**sa**[n×(n+1)/2]为n阶对称矩阵A的**压缩矩阵**

2. **三角矩阵**：
   - 三角矩阵分为上三角矩阵和下三角矩阵
   - 三角矩阵中的重复元素c可共享一个存储空间，其余的元素正好有n(n+1)/2个，因此，三角矩阵可压缩存储到向量sa[0...n(n+1)/2]中，其中c存放在向量的第n(n+1)/2个分量中。
   - 下三角矩阵：k=i×(i-1)/2+j-1，i>=j;k=n×(n-1)/2,i<j
   - 上三角矩阵：k=(2n-i+2)×(i-1)/2+j-i,i<j;k=
3. **对角矩阵**：
   - 非零元素仅出现在主对角上，主对角线上的那条对角线，主对角线下的那条对角线上。
   - 元素个数为3n-2，存在一维矩阵sa[3n-2]中
   - 一个k对角矩阵(k为奇数)A是满足下述条件： 当|i-j|>(k-1)/2时， aij=0
   - 以**三对角矩阵**为例讨论：对于aij，在前面有i-1行元素，其中第一行只有两个元素，故共有3i-4个元素；在第i行里，是第j-i+2个元素，由于sa是从0开始计数，故aij下标k=3i-4+j-i+2-1=2i+j-3
   - 称**sa**[3n-2]为n阶对称矩阵A的**压缩矩阵**
4. **稀疏矩阵**
   - 对于稀疏矩阵，目前还没有一个确切的定义。设矩阵A是一个n×m的矩阵中有s个非零元素，设 δ=s/(n×m)，称δ为稀疏因子，如果某一矩阵的稀疏因子δ满足δ≤0.05时称为稀疏矩阵。
   - 对于稀疏矩阵，采用压缩存储方法时，**只存储非0元素**。必须存储非0元素的行下标值、列下标值、元素值。因此，一个**三元组**(i, j, aij)唯一确定稀疏矩阵的一个非零元素。
   - 例如：((1,2,12),(1,3,9),(3,1,-3),(3,8,4),(4,3,24),(5,2,18),(6,7,-7),(7,4,-6))
   - 十字链表：

$$A=\begin{pmatrix} 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix}$$

(a) 稀疏矩阵




- ```
  typedef struct Clnode
  {
      int row, col;    /*  行号和列号  */
      int value;     /*  元素值  */
      struct Clnode* down, * right;
  }OLNode;//非零结点

  typedef struct
  {
      int rn;          /*  矩阵的行数  */
      int cn;          /*  矩阵的列数  */
      int tn;          /*  非0元素总数  */
      OLNode* rhead;
      OLNode* chead;
  } CrossList;//头结点
  ```

2.5.4广义表

广义表是线性表的推广和扩充，在人工智能领域中应用十分广泛。

线性表是n(n≥0)个元素a1, a2 ,..., an的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项(Atom)。**所谓原子项可以是一个数或一个结构，是指结构上不可再分的。**若**放松对元素的这种限制**，容许它们具有其自身结构，就产生了广义表的概念。

广义表(Lists，又称为列表)：是由n(n ≥0)个元素组成的有穷序列： LS=(a1，a2，...，an)。其中**ai或者是原子项，或者是一个广义表。**LS是广义表的名字，n为它的长度。若ai是广义表，则称为LS的子表。

习惯上：原子用小写字母，子表用大写字母。

若广义表LS非空时：

- a1(表中第一个元素)称为表头；
- **其余元素组成的子表**称为**表尾**：(a2，a3，...，an)
  ```

- 广义表中所包含的元素(包括原子和子表)的个数称为表的长度。
- 广义表中括号的最大层数称为表深 (度)。

表5-2　　广义表及其示例

广　义　表	表长n	表深h
A=( )	0	0
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞
F=(( ))	1	2

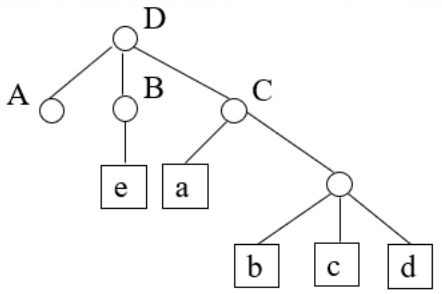


图5-12　　广义表的图形表示

重要结论:

(1) 广义表的元素可以是原子，也可以是子表，子表的元素又可以是原子，又可以是子表， ...。即广义表是一个多层次的结构。（表2中的广义表D的图形表示如图5-12所示。E是一个深度为无穷的元素，其实际表示就是E=(a,(a,(a,(a,...)))) ，是一个无限嵌套的元素。）

(2) 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。

(3) 广义表本身可以是一个递归表。

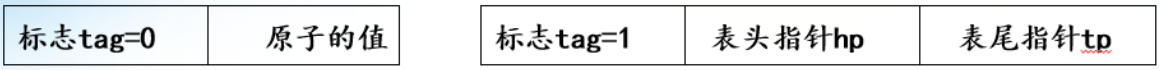
(4) 根据对表头、表尾的定义，任何一个非空广义表的表头可以是原子，也可以是子表， 而表尾必定是广义表。

存储结构:

第一种存储结构:

由于广义表中的数据元素具有不同的结构，通常用链式存储结构表示，每个数据元素用一个结点表示。因此，广义表中就有两类结点:

- 一类是**表结点**，用来表示**广义表项**，由**标志域**，**表头指针域**，**表尾指针域**组成;
- 另一类是**原子结点**，用来表示**原子项**，由**标志域**，原子的**值域**组成。如图5-13所示。

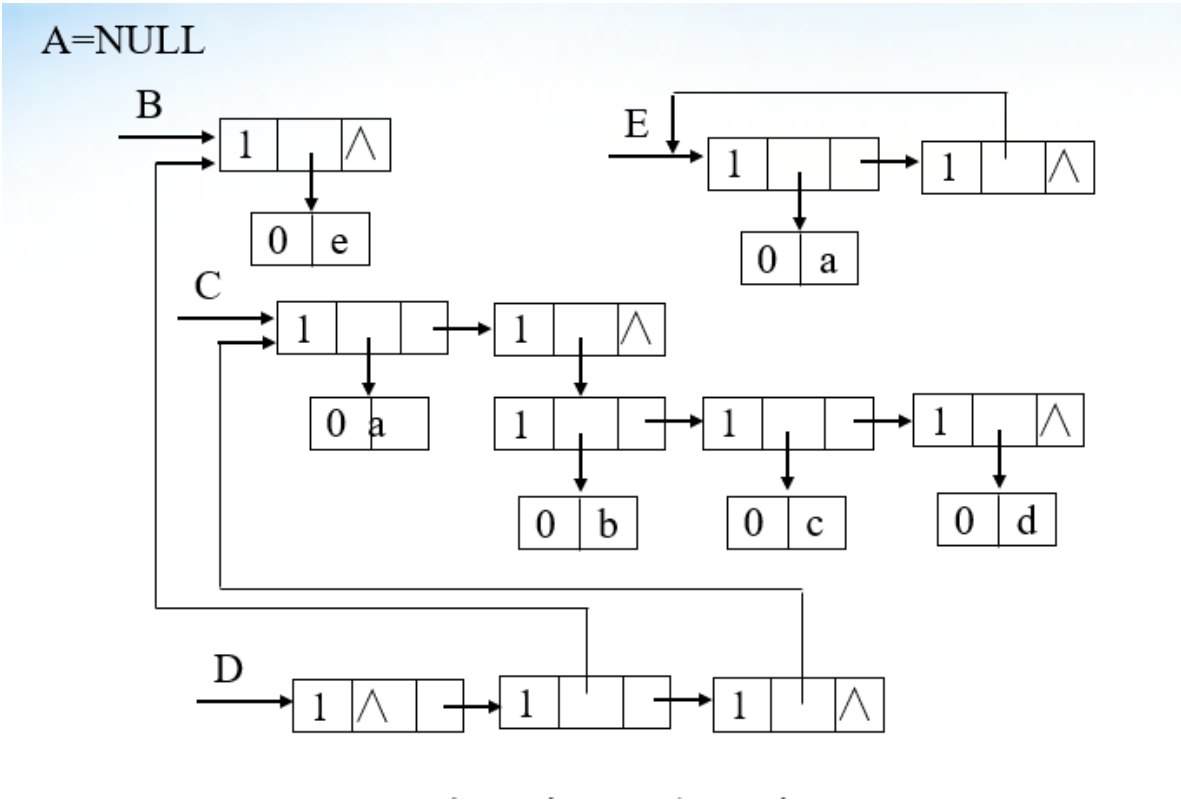


(a)　原子结点　　(b)　表结点

结构定义

```
typedef struct GLNode
{
    int tag; /*  标志域，为1：表结点;为0 ：原子结点  */
    union
    {
        elemtype value;     /* 原子结点的值域  */
        struct
        {
            struct GLNode* hp, * tp;
        }ptr;   /*  ptr和atom两成员共用  */
    }Gdata;
} GLNode;      /* 广义表结点类型  */
```

例： 对A=()，B=(e)，C=(a, (b, c, d))，D=(A, B, C)，E=(a, E)的广义表的存储结构如图所示。

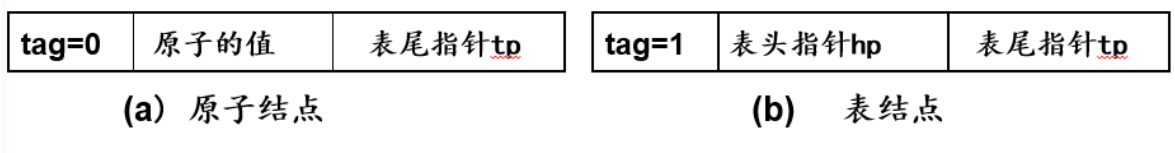


对于上述存储结构，有如下几个特点：

(1) 若广义表为空，表头指针为空；否则，表头指针总是指向一个表结点，其中hp指向广义表的表头结点(或为原子结点，或为表结点)，tp指向广义表的表尾(表尾为空时，指针为空，否则必为表结点)。

(2) 这种结构求广义表的长度、深度、表头、表尾的操作十分方便。

(3) 表结点太多，造成空间浪费。也可用以下所示的结点结构。



(a) 原子结点　　(b) 表结点

结构定义

```
template<class T>
class GLNode {
public:
```

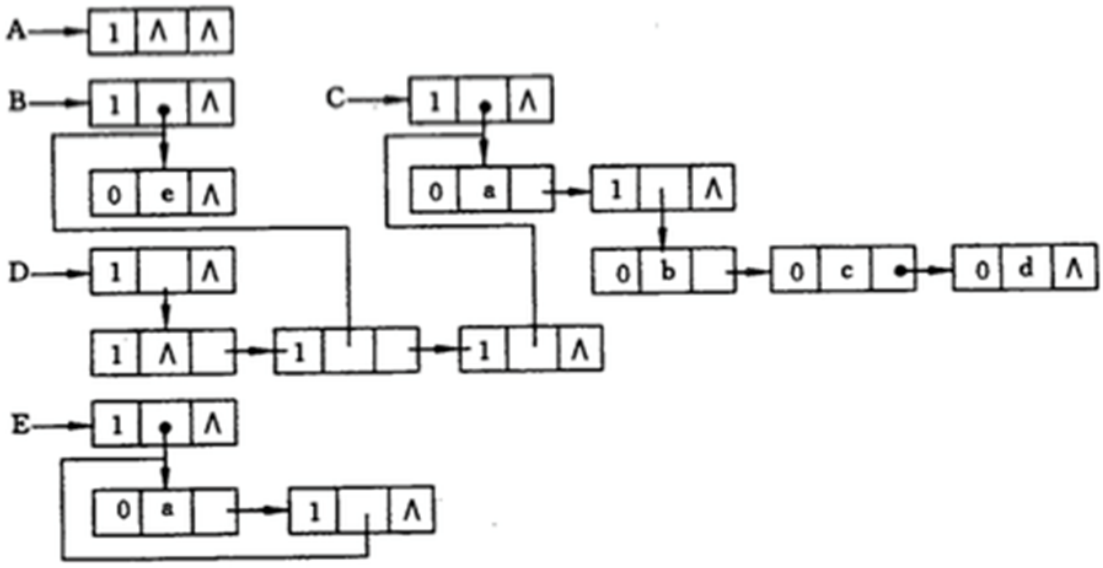
```
    GLNode(int type, T value);
    GLNode(int type,GLNode *value)
private:
    int tag;/*  标志域，为1：表结点;为0 ：原子结点  */
    union {
        T value;
        GLNode* hp;
    }data;
    GLNode* tp;
};/* 广义表结点类型  */

template<class T>
inline GLNode<T>::GLNode(int type, T value)
{
    this->tag = type;
    this->data = value;
    this->tp = NULL;
}

template<class T>
inline GLNode<T>::GLNode(int type, GLNode* value) {
    this->tag = type;
    this->data = value;
    this->tp = NULL;
}
```

例： 对A=()，B=(e)，C=(a, (b, c, d))，D=(A, B, C)，E=(a, E)的广义表的存储结构如图所示。

存储结构：



3.树 Tree

3.1基本概念

定义：n个结点的有限集合(n≥0)；空树：n=0的树(Tree)

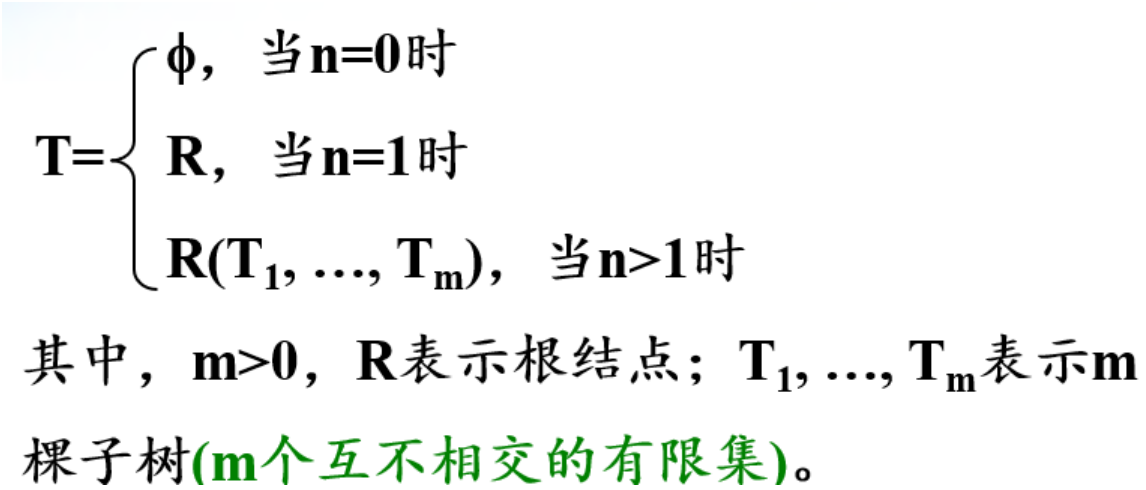
1. 对于非空树(n>0时)，**有且只有**1个根结点;

   ——根结点：没有前驱的结点

2. 当n>1时，除根结点之外，树中的其它任意1个结点**有且只有**1个前驱；

3. 树中每个结点可以有m个后继(m≥0)。

树的递归定义：



树的结点：包含一个数据元素和若干个指针(这些指针指向其拥有子树的根)

结点的度：结点拥有子树的个数

叶子结点：度=0的结点

分支节点：度>0的结点

树的度：max(结点的度)

结点的孩子：

结点的孩子：该结点的指针指向的结点——结点拥有的(某棵)子树的根。

孩子的双亲：孩子结点的前驱(结点)。

兄弟结点：同一个双亲结点的孩子。例如，在树A(B，C，D)中，结点C是结点B (或D)的兄弟结点。

结点的层次：根结点定义为第1层；根结点的孩子定义为第2层；第i层结点的孩子定义为第i+1层。

树的高度(或深度) =max(结点的层次)。例如树A(B(E(M, N), F), C(G, H(P), I), D(J, K))的高度= 4。**树的高度≠ 树的度**(不同概念)。

有序树：(树中每一个)结点的各棵子树按一定的次序从左向右排列的树。

无序树：结点各子树与次序无关的树。

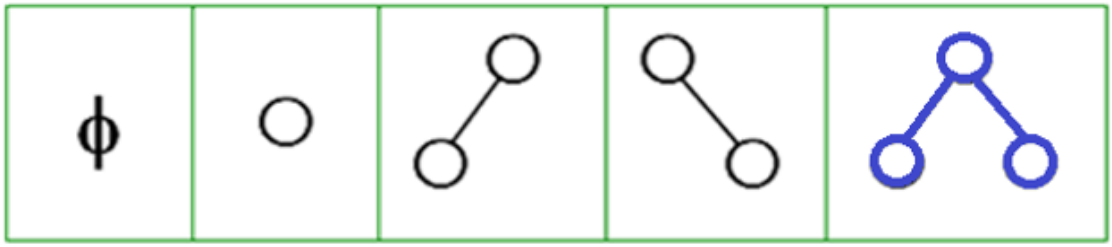
例如，对于有序树：A(B, C)≠A(C, B)，对于无序树：A(B, C) = A(C, B)。

森林：m棵互不相交的树的有限集， 即F={ T1, T2, ..., Tm }，m≥0。当m=0时，称为空森林。对于森林，如果将每棵树都赋予同一个双亲结点R，即F=R(T1, ..., Tm)，则森林成为一棵树。

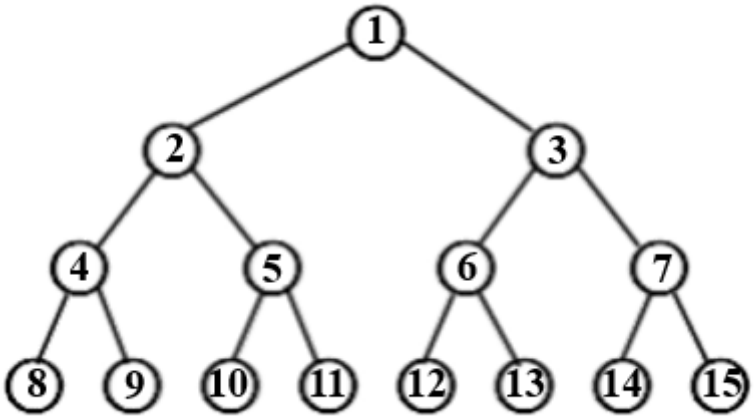
3.2二叉树

定义：结点的度<=2的(有序)树——二叉树中每个结点的子树最多2棵。

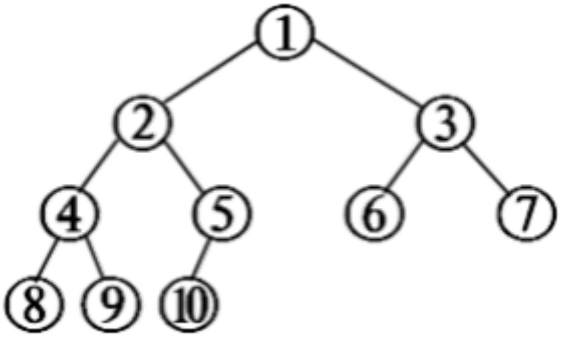
二叉树的5中基本形态：



满二叉树：一棵高度为k且具有2^k-1个结点的二叉树，例如下图是k=4的满二叉树。



完全二叉树：一棵高度为k、结点个数∈[2^(k-1),2^k-1]，且第k层的结点都集中在左侧的二叉树。(满二叉树也是完全二叉树)。



二叉树的性质：

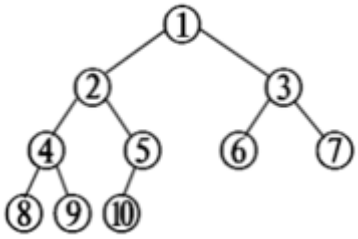
1. 在二叉树的第i层上最多有2^(i-1)个结点（i>=1)
2. 高度为k的二叉树最多有2^k-1个结点(k>=1)
3. 如果二叉树T的叶子结点数为n0，度为2的结点数为n2，则n0=n2+1

   证明：设二叉树T共有n个结点，叶子结点数为n0，度=1的结点数为n1，度=2的结点数为n2，T的分支数为m。

   有n=n0+n1+n2;m=n-1=n1+n2×2;

   即n0+n1+n2-1=n1+n2×2 =>n0=n2+1
4. ☞性质4：具有n个结点的完全二叉树T的高度为$\lfloor \log_2 n \rfloor + 1$。
5. 设**完全二叉树**含有n个结点。

1. 如果编号i=1，则结点i是根节点；如果编号i>1，则结点i的双亲结点的编号是(i/2)的向下取整
2. 结点i的左孩子的编号是2i；结点i的右孩子的编号是2i+1

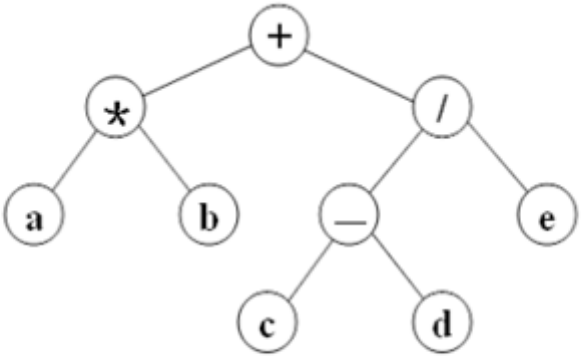


3. 如果2i<n，则结点i为分支结点；如果2i>n，则结点i为叶子结点。
4. 如果n为奇数，则每个**分支结点**都有左孩子和右孩子，如果n为偶数，则编号为n/2的分支结点只有左孩子，没有右孩子。

二叉树的存储结构

二叉树的**顺序存储结构**：

- 将二叉树的所有结点按照一定的次序存放到一组地址连续的存储单元中。（一般从根节点开始，自上而下，自左到右排成一个线性序列）



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	*	/	a	b	-	e					c	d		

- 将结点的编号和数组下标一一对应；结点的编号必须能反映其逻辑关系。
- 顺序存储结构适用于**完全二叉树**——结点的编号容易和数组下标对应，且浪费的存储空间较小。

二叉树的**链式存储结构**

操作：

1. 初始化

```
template<class T>
class BTree {
public:
    T data;
    BTree<T>* lc;
    BTree<T>* rc;

    BTree() { this->lc = this->rc = NULL; };
    BTree(T e);
};

template<class T>
inline BTree<T>::BTree(T e){
```

```
    this->data = e;
    this->lc = this->rc = NULL;
}
```

2. 根据树的字符形式来创建二叉树

```
template<class T>
void BTree<T>::create(string str,int &k)
{
    T ch = str[k];
    while (ch) {
        if (ch == '('&&str[k+1]==',')k+=2;
        else if (ch == '(' || ch == ',' && str[k + 1] == ')')k++;
        else if (ch == ')' || ch == ',') {
            k++; return;
        }
        else{
            BTree<T>* t = (BTree<T>*)malloc(sizeof(BTree<T>));
            //如果直接用构造函数初始化，地址不变，定义右孩子时左孩子也会变化
            t->data = ch;
            if (k != 0 && str[k - 1] == ',')this->rc = t;
            else this->lc = t;
            k++;
            t->create(str, k);
            ch = str[k];
            continue;
        }
        ch = str[k];
    }
}
```

3. 打印树

```
template<class T> //跳过头结点，进入根节点
void BTree<T>::toString() {
    this->lc->tostring();
    cout << endl;
}

template<class T>
void BTree<T>::tostring() {
    if (!this)return;//是空子树则返回
    cout << this->data ;
    if (this->lc) {
        cout << "(";
        this->lc->tostring();
    }
    else {
        if (this->rc)
            cout << "(";
        else return;
    }
    if (this->rc) {
```

```
        cout << ",";
        this->rc->tostring();
        cout << ")";
    }
    else if(!this->rc){
        cout <<")"; return;
    }
}
```

4. 求树的深度

```
template<class T>
int BTree<T>::depth() {
    if (!this)return -1;
    int ld = this->lc->depth();
    int rd = this->rc->depth();
    return ld > rd ? ld + 1 : rd + 1;
}
```

3.3二叉树遍历

遍历二叉树：按照 一定规则访问二叉树中的每个结点，使得每个结点均能被访问一次，而且仅被访问一次。

通过一次遍历，可以使二叉树中的结点由**非线性排列**变为**线性排列**——**遍历操作可以使二叉树线性化。**

二叉树由3个基本单元组成：**根结点、左子树和右子树**，因此，遍历二叉树的问题可归结为解决三个子问题：

1. 访问根结点(D)；
2. 遍历左子树(L)；
3. 遍历右子树(R)。

常用的三种遍历顺序：DLR（先序）、LDR（中序）、LRD（后序）

- (DLR)先序遍历

  若二叉树为空，则空操作；否则

  1. 访问根结点；
  2. 先序遍历左子树；
  3. 先序遍历右子树。

- (LDR)中序遍历
  1. 中序遍历左子树
  2. 访问根节点
  3. 中序遍历右子树
- (LRD)后序遍历
  1. 后续遍历左子树
  2. 后序遍历右子树
  3. 访问根节点
- 操作（递归遍历）：

```
template<class T>//先序遍历
```

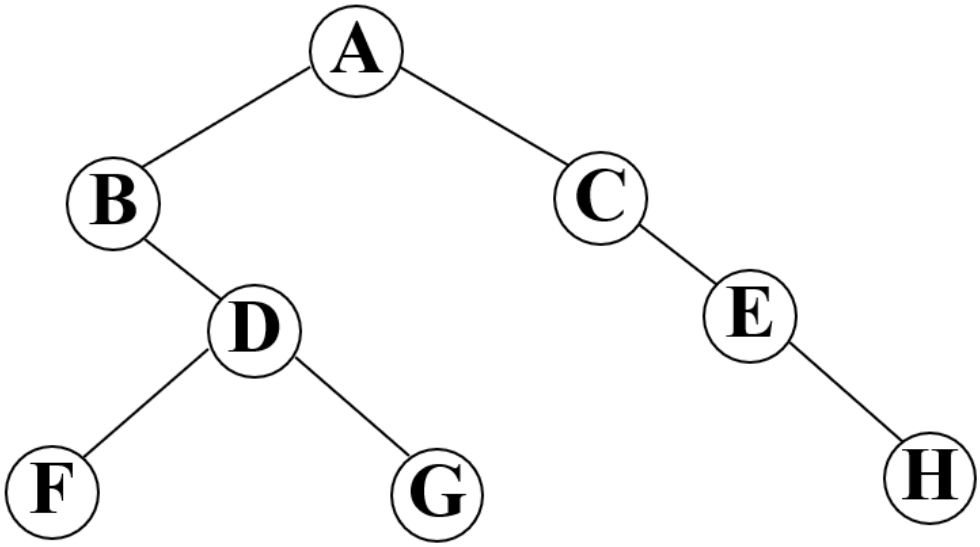
```
void BTree<T>::preOrder() {
    if (!this)return;
    cout << this->data;
    this->lc->preOrder();
    this->rc->preOrder();
}

template<class T>//中序遍历
void BTree<T>::inOrder() {
    if (!this)return;
    this->lc->inOrder();
    cout << this->data;
    this->rc->inOrder();
}

template<class T>//后序遍历
void BTree<T>::postOrder() {
    if (!this)return;
    this->lc->postOrder();
    this->rc->postOrder();
    cout << this->data;
}
```

题目:

1. 已知二叉树T的先序序列和中序序列分别为ABDFGCEH和BFDGACEH，试画出T。



2. 设二叉树T的中序序列和后序序列分别为 (中序) 3, 7, 11, 14, 18, 22, 27, 35; (后序) 3, 11, 7, 14, 27, 35, 22, 18。试画出二叉树T。

   T = 18 (14 (7 (3, 11)), 22 (, 35 (27)))

非递归遍历:

1. 先序

```
template<class T>
void preOrder(BTree<T> tree) {
    LinkStack<BTree<T>*> tstack;
    BTree<T>* p ;
    tstack.push(tree.lc);
```

```
    while (!tstack.isEmpty()) {
        p = tstack.pop();
        p->visit();
        if (p->rc)//先将右节点放入栈中，弹出时就是在左节点后
            tstack.push(p->rc);
        if (p->lc)
            tstack.push(p->lc);
    }
}
```

2. 中序

```
template<class T>
void inOrder(BTree<T> tree) {
    LinkStack<BTree<T>*> tstack;
    BTree<T>* p= tree.lc;
    while (p||!tstack.isEmpty()) {
        if (p) {
            tstack.push(p);
            p = p->lc;
        }
        else {
            p = tstack.pop();
            p->visit();
            p = p->rc;
        }
    }
}
```

3. 后序

```
template<class T>
void postOrder(BTree<T> tree) {
    LinkStack<BTree<T>*> lstack;
    LinkStack<BTree<T>*> rstack;
    BTree<T>* p;
    lstack.push(tree.lc);
    while (!lstack.isEmpty()) {
        p = lstack.pop();
        rstack.push(p);
        if (p->lc)
            lstack.push(p->lc);
        if (p->rc)
            lstack.push(p->rc);
    }
    while (!rstack.isEmpty()) {
        p = rstack.pop();
        p->visit();
    }
}
```

3.4线索二叉树

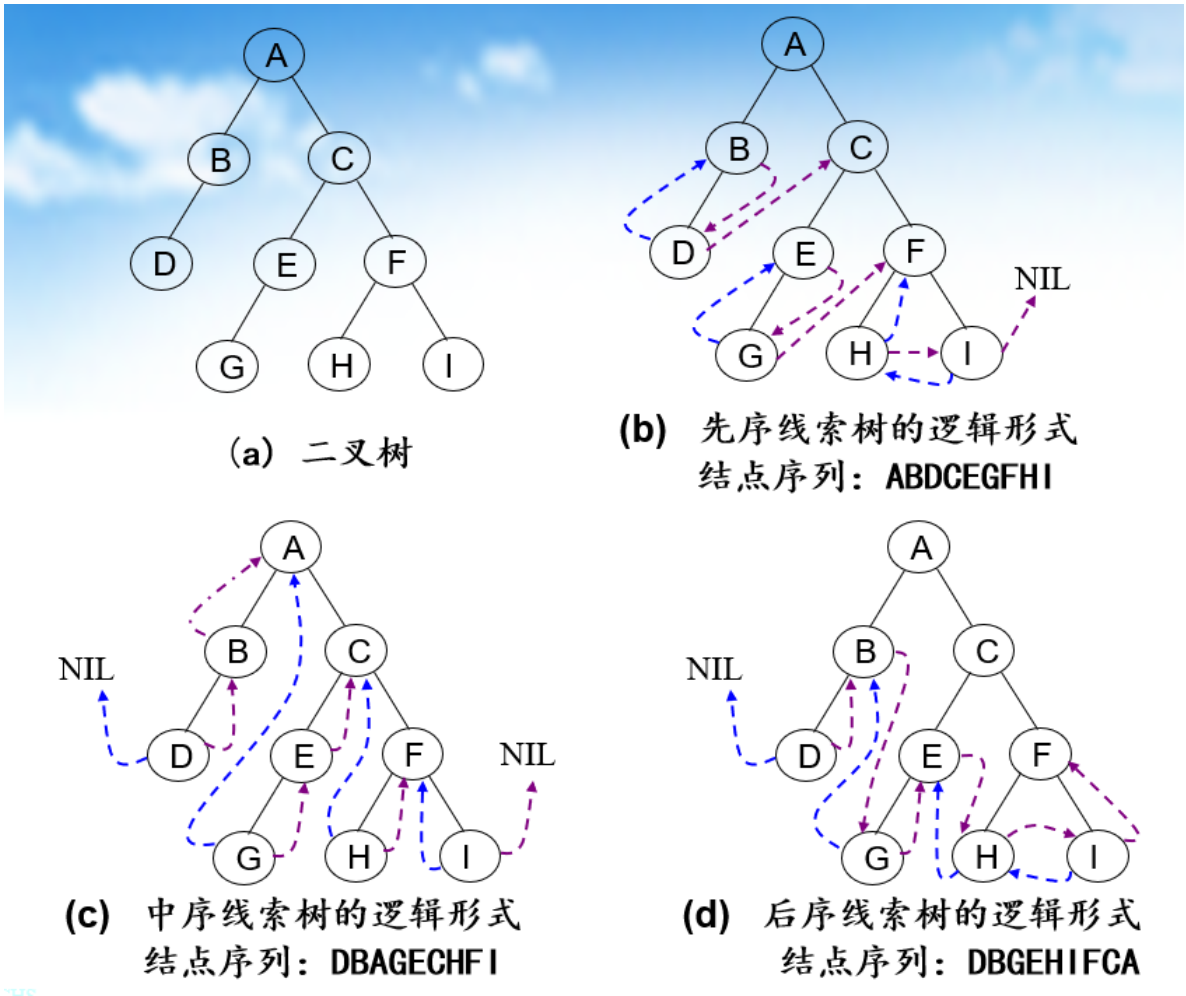
定义

线索：在结点的空指针域中存放指向某次**遍历**得到的**后继或前驱**的指针。

后继线索(右线索)：在空右指针域中存放指向某次遍历得到的后继结点的指针。

前驱线索(左线索)：在空左指针域中存放指向某次遍历得到的前驱结点的指针。

线索二叉树：根据某次遍历, 在二叉树中的相关空指针域都写入线索(后继线索或前驱线索)，即成为线索二叉树。线索二叉树可理解为已经线索化的二叉树。先序后继：先序遍历中得到的后继(先序前驱, 中序后继, 中序前驱, 后序后继, 后序前驱)。



线索化原则

先序后继：将右子树的根结点链入左子树的最后一个叶子结点。

中序后继：给定一个结点x，它是其左子树最后一个叶子结点的后继结点。

后序后继：

（1）若结点x是二叉树的根，则其后继为空；

（2）若结点x是其双亲的右孩子或是其双亲的左孩子且其双亲没有右孩子，则其后继即为双亲结点；

（3）若结点x是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点。

线索化实现

1. 先序线索化 遍历

```
void BTree<T>::preThreading(BTree<T>* B,BTree<T>* &pre) {
    if (!B || !pre)return;
```

```
    //处理根节点
    //如果当前子树的根节点的左子树为空 就把left指针指向前驱
    if (B->lc == NULL) {
        B->lc = pre;
        B->ltag = 1;
    }
    //如果当前子树的根结点的前驱的右子树为空 就把前驱的right指针指向根节点
    if (pre != NULL && pre->rc == NULL) {
        pre->rc = B;
        pre->rtag = 1;
    }

    pre = B;
    if (B->ltag == 0)
        preThreading(B->lc, pre);
    if (B->rtag == 0)
        preThreading(B->rc, pre);
    return;
}

template<class T>
void BTree<T>::preOrderTraverse(BTree<T>* B) {
    if (!B) return;
    BTree<T>* p = B;
    while (p) {
        while (p->ltag == 0) {
            p->visit();
            p = p->lc;
        }
        p->visit();
        p = p->rc;
    }
    cout << endl;
}
```

2. 中序线索化 遍历

```
template<class T>
void BTree<T>::inThreading(BTree<T>* B,BTree<T>* &pre)
{
    if (!B)return;
    inThreading(B->lc, pre);

    if (!B->lc) {        //没有左孩子
        B->ltag = 1;     //修改左标志域
        B->lc = pre;     //左孩子指向前驱结点
    }

    if (!pre->rc) {      //没有右孩子
        pre->rtag = 1;   //修改右标志域
        pre->rc = B;     //前驱结点右孩子指向当前节点
    }
    pre = B;             //保证pre指向下一节点的前驱

    inThreading(B->rc, pre);
}
```

```
template<class T>
void BTree<T>::inOrderTraverse(BTree<T>* B) {
    BTree<T>* p = B;
    while (p) {
        while (p->ltag == 0)p = p->lc;
        p->visit();
        while (p->rtag == 1) {
            p = p->rc;
            p->visit();
        }
        p = p->rc;
    }
}
```

3. 后序线索化 遍历

```
template<class T>
void BTree<T>::postThreading(BTree<T>* B, BTree<T>*& pre) {
    if (!B)return;

    postThreading(B->lc, pre);
    postThreading(B->rc, pre);

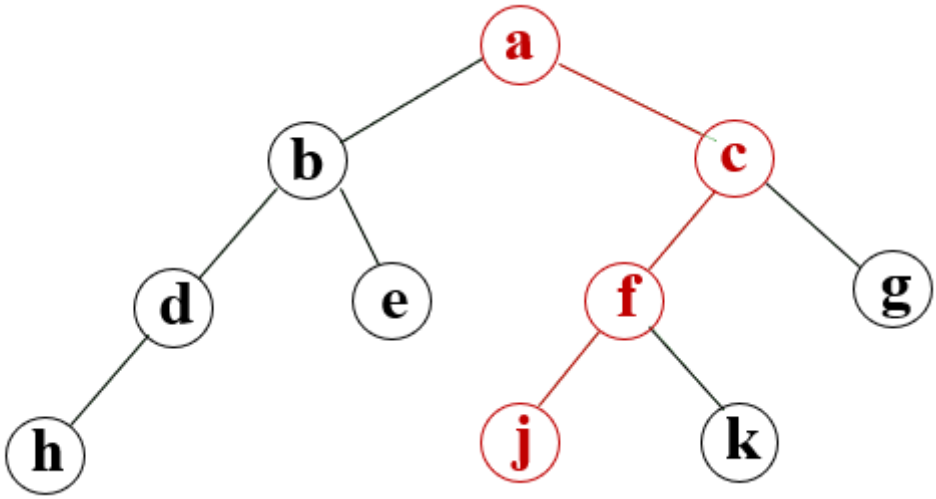
    if (!B->lc) {
        B->ltag = 1;
        B->lc = pre;
    }
    if (pre && !pre->rc) {
        pre->rtag = 1;
        pre->rc = B;
    }
    pre = B;
}

template<class T>
void BTree<T>::postOrderTraverse(BTree<T>* B) {
    if (!B)return;
    LinkStack<char> stack;
    BTree<T>* p = B;
    stack.push(p->data);
    while (p) {
        while (p->rtag == 0&&p->rc) {
            p = p->rc;
            stack.push(p->data);
        }
        while (p->lc&& p->ltag == 1) {
            p = p->lc;
            stack.push(p->data);
        }
        if (p->rc&&p->lc)
            continue;
        p = p->lc;
    }
    cout << endl;
    while (!stack.isEmpty()) {
        cout << stack.pop();
```

```
    }
}
```

3.5哈夫曼树

哈夫曼(Huffman)树又称**最优二叉树。**



路径：从一个结点到另一个结点之间的分支序列，如(a, j)=(a, c)+(c, f)+(f, j) 。

路径长度：路径上的分支数，如(a, j)=3。

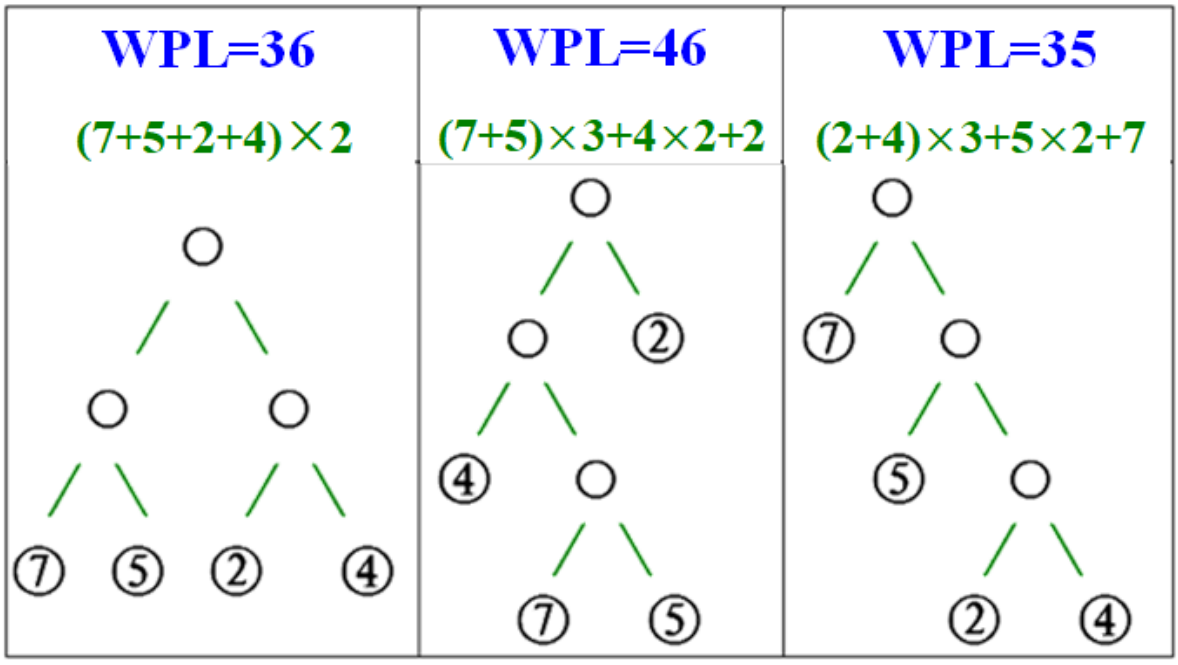
树的路径长度：从**根结点**到**树中每一个叶子结点**的路径长度之**和。**

结点的权值：1个具有某种含义的实数。

带权路径长度：从根结点到某个(叶子)结点之间的**路径长度**与该**结点的权**的**乘积。**

树的带权路径长度：树中所有叶子结点的带权路径长度之和，记为

WPL =∑wi li ， i=1, ..., n。其中, n表示叶子结点的数目；wi表示第i个叶子结点的**权值**；li 表示从根结点到第i个叶子结点之间的**路径长度。**

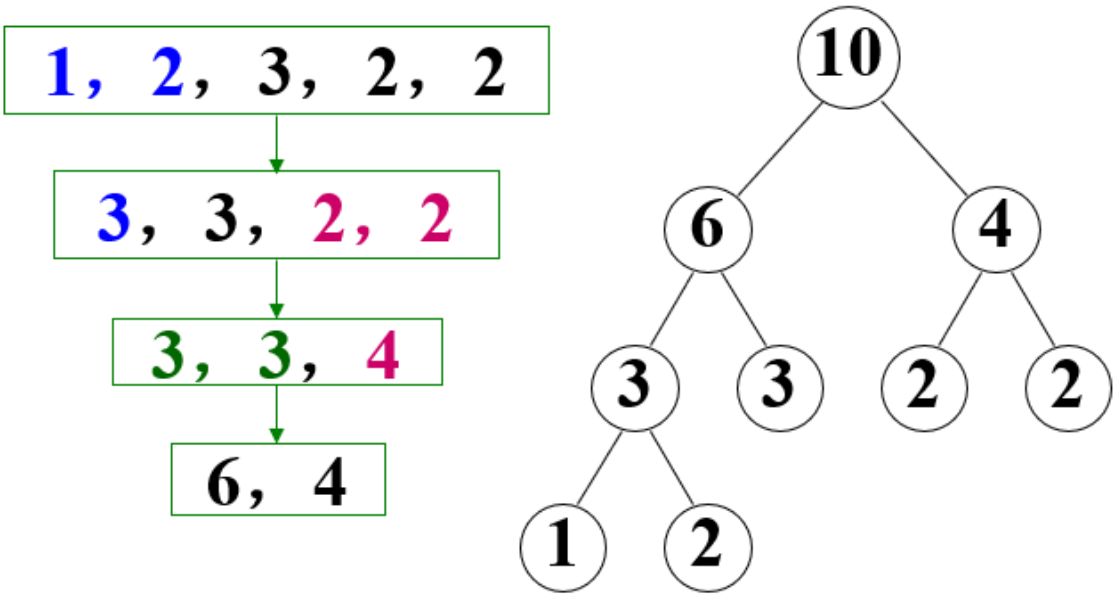


哈夫曼树：在n个叶子结点所构造的二叉树中，WPL最小的二叉树。一般不唯一。其中结点的度为0或2。

构造最优二叉树的算法

1. 根据给定的n个权值{ w1, w2, ..., wn }, 构成n棵二叉树集 T = { T1, T2, ..., Tn }，其中，Ti只含1个带权为wi的根结点；
2. 在T中选取两棵根结点的**权值最小**的二叉树Tj和Tk作为左、右子树，构成一棵新的二叉树Tjk , 且置Tjk**根结点的权值**为Tj和Tk根结点的**权值之和**；
3. 从T中删除二叉树Tj和Tk，并将二叉树Tjk添加到T中；
4. 重复(2)和(3)的操作，直到T中只含有一棵二叉树为止 => 哈夫曼树。

例：按照权值 { 1，2，3，2，2 } 构造一棵哈夫曼树。



构造说明：

- 在选取结点权值最小的两棵子树时, 如果出现权值相同，可以任选其中一棵子树;
- 两棵结点权值最小的二叉树组成新的二叉树时，没有规定左右子树次序；
- 在哈夫曼树中，权值越大的叶子结点离根结点越近——哈夫曼树的应用依据;
- 具有n个叶子结点的哈夫曼树，共有2n-1个结点(满足二叉树的性质)。
    - 在二叉树中，结点总数为n0+n1+n2，在哈夫曼树中，没有度为1的结点，即哈夫曼树的结点总数为n0+0+n2，根据二叉树性质n0=n2+1，即n2=n0-1=>哈夫曼树的结点总数n0+n2=2n0-1=2n-1。

代码实现：

1. 构造哈夫曼树

```
template<typename T>
MList<HuffmanNode<T>> Huffman(int n) {
    int i=0,j=0;
    T e;
    int weight;
    //由于C++不能直接根据变量名来定义数组，所以这边用了个线性表，实际上用数组的话会更加方便
    MList<HuffmanNode<T>> hufflist(2 * n - 1);

    for (i = 0; i < n; i++) {
        cin >> e >> weight;
        hufflist.elem[i]=HuffmanNode<T>(e, weight);
```

```
    }

    int min1, min2,locate1,locate2;
    for (i = n; i < 2*n - 1; i++) {
        min1 = INT_MAX, min2 = INT_MAX;
        locate1 = -1; locate2 = -1;
        for (j = 0; j < i; j++) {
            //挑选最小两个数，左子树比右子树小
            if (hufflist.elem[j].w <= min1 && hufflist.elem[j].parent ==
NULL) {
                min2 = min1;
                locate2 = locate1;
                min1 = hufflist.elem[j].w;
                locate1 = j;
            }
            else if (hufflist.elem[j].w <= min2 && hufflist.elem[j].parent
== NULL) {
                min2 = hufflist.elem[j].w;
                locate2 = j;
            }
        }
        hufflist.elem[locate1].parent = i;
        hufflist.elem[locate2].parent = i;
        hufflist.elem[i]=HuffmanNode<T>('A'+i, hufflist.elem[locate1].w +
hufflist.elem[locate2].w, locate1, locate2);
    }

    for (i = 0; i < 2 * n - 1; i++) {
        cout << i << ":\t";
        hufflist.elem[i].toString();
    }
    return hufflist;
}
```

2. 输出哈夫曼编码

```
template<typename T>
void Huffcode(MList<HuffmanNode<T>> hufflist,int n) {
    stack<char> code;
    int i, j;
    for (i = 0; i < n; i++) {
        j = i;
        while (hufflist.elem[j].parent != NULL) {
            if (hufflist.elem[hufflist.elem[j].parent].lc ==
j)code.push('0');
            else code.push('1');
            j = hufflist.elem[j].parent;
        }
        cout << i << ":" << hufflist.elem[i].data << "\t";
        while (!code.empty()) {
            cout << code.top();
            code.pop();
        }
        cout << endl;
    }
}
```

3.6树的存储结构

树的双亲表示法

用顺序表存储树的结点，每个结点包含数据域和指针域(双亲位置)。

也称为**树的静态链表**结构，它是以一组地址连续的存储空间存放树的结点，每个结点中包含**数据域**和**整型指针域**。

位置	0	1	2	3	4	5	6	7	8	9
数据域	R	A	B	C	D	E	F	G	H	K
指针域	-1	0	0	0	1	1	3	6	6	6

```
//树的双亲表示法
template<class T>
class TNode {
public:
    T data;  //数据元素
    int parent;//双亲位置
}; //结点结构

//树的存储结构
template<class T>
class PTree {
public:
    int length;
    TNode<T> tree[100];
    int nodenum;//结点数

    PTree();
};

template<class T>
PTree<T>::PTree() {
    this->length = 100;
    this->nodenum = 0;
}
```

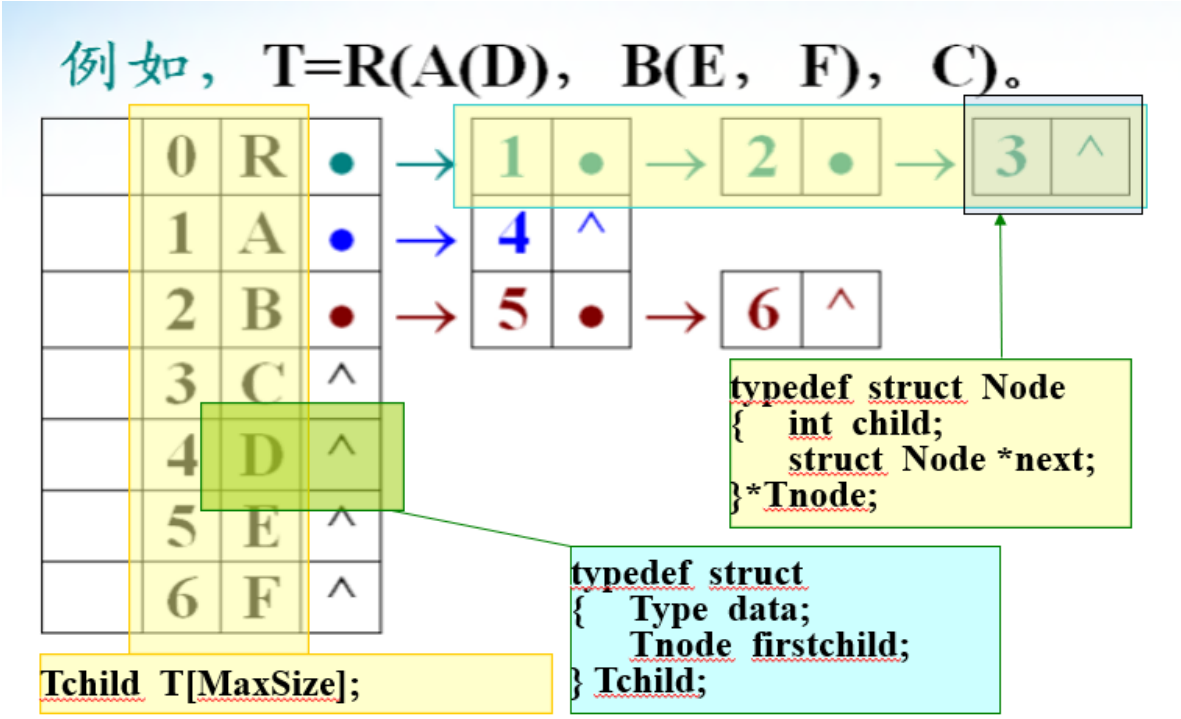
树的孩子表示法

每个结点建立1个链表，用于存储该结点的所有孩子结点。

每个结点建立一个链表，该链表由它的**孩子结点**构成。其中，**叶子结点产生空链表。**

在链表中，**孩子域(即数据域)**存放孩子结点在**顺序表**中的**位置**(下标)。

由**n个结点的元素值**和**指向孩子链表的头指针**再构成1个**线性表**(顺序存储结构)。




```
template<class T>
class CNode {
public:
    int child; // 孩子结点在顺序表中的位置
    CNode<T>* next;
};//孩子链表结构

template<class T>
class Tchild{
public:
    T data;
    CNode<T>  firstchild;//孩子链表头指针

    Tchild(T e) {
        this->data = e;
        this->firstchild = new CNode<T>;
    }
    ~Tchild() {
        delete this->firstchild;
    }
};//表头指针结构

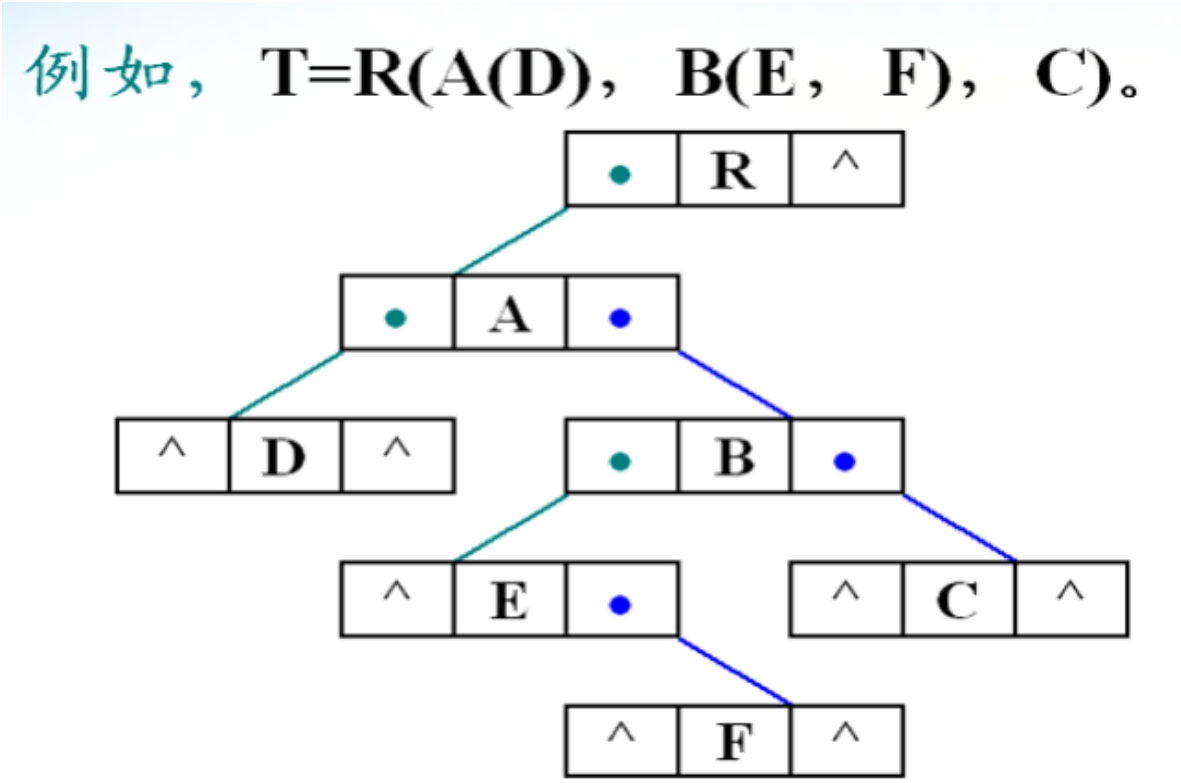
template<class T>
class CTree
{
public:
    Tchild<T>  T[100];
    int  r, n;//根的位置，结点数

    CTree() { r = 0; n = 0; }
};//树的存储结构
```

树的孩子兄弟表示法

用二叉树的存储结构表示树，第1个孩子为左孩子，其它兄弟依次存储为右孩子。也称为**树的二叉链表存储结构**

```
//树的孩子兄弟表示法
template<class T>
class Tnode
{
    T data;//数据元素域
    struct Tnode* Child1;//第1个孩子
    struct Tnode* Sibling;//下1个兄弟
};
```

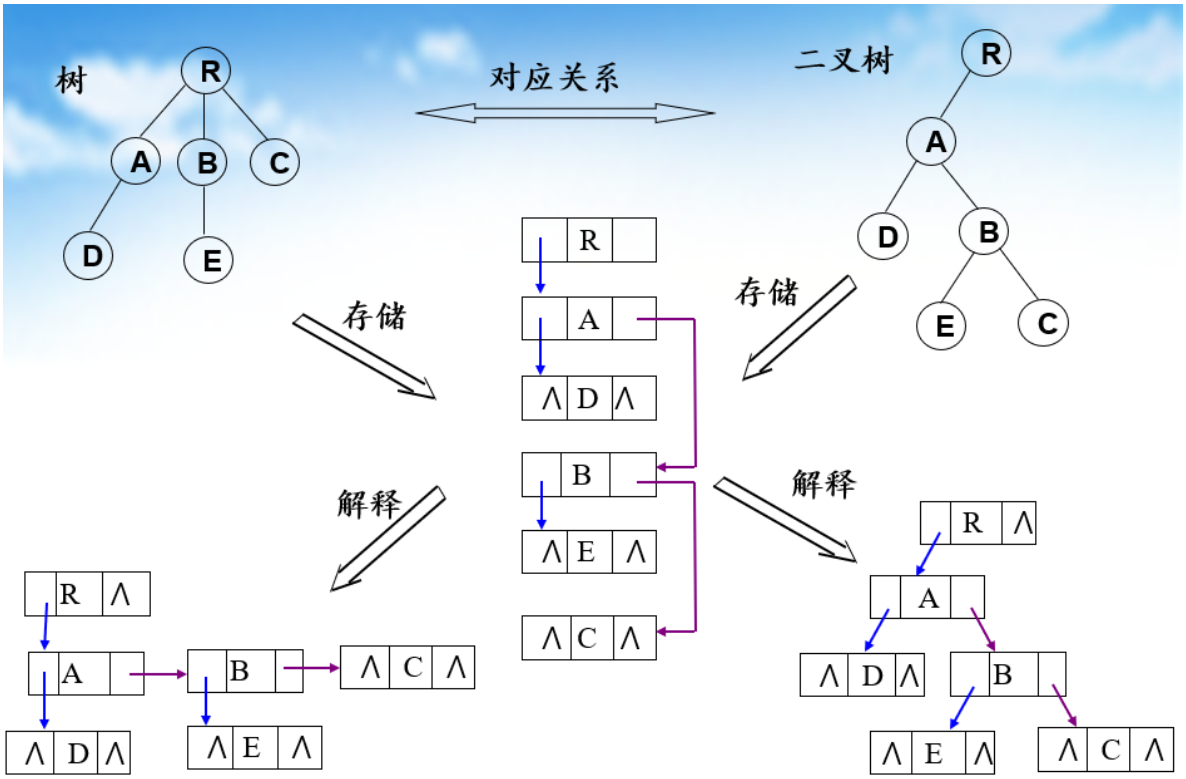


3.7森林与二叉树的转换

由于二叉树和树都可用**二叉链表**作为存储结构，对比各自的结点结构可以看出，以二叉链表作为媒介可以导出树和二叉树之间的一个对应关系。

- 从**物理结构**来看，树和二叉树的二叉链表是相同的，只是对指针的逻辑解释不同而已。
- 从树的**二叉链表**表示的**定义**可知，任何一棵和树对应的二叉树，其**右子树一定为空。**

下图直观地展示了树和二叉树之间的对应关系。



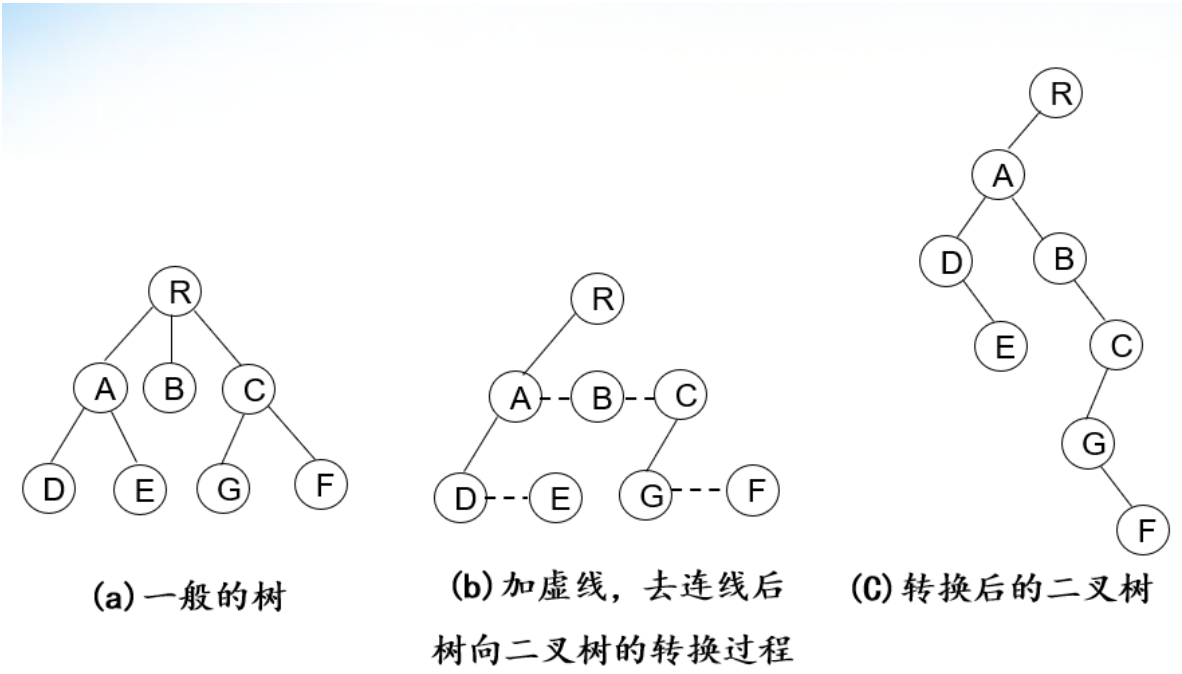
树转换成二叉树

对于一般的树，可以方便地转换成一棵唯一的二叉树与之对应。将树转换成二叉树在“孩子兄弟表示法”中已给出，其详细步骤是：

1. 加虚线。在树的每层按从“左至右”的顺序在兄弟结点之间加虚线相连。
2. 去连线。除最左的第一个子结点外，父结点与所有其它子结点的连线都去掉。
3. 旋转。将树顺时针旋转450，原有的实线左斜。
4. 整型。将旋转后树中的所有虚线改为实线，并向右斜。该转换过程如下图所示。

这样转换后的二叉树的特点是：

- 二叉树的**根结点没有右子树**，只有左子树；
- 左子结点仍然是原来树中相应结点的左子结点，而所有**沿右链往下的右子结点均是原来树中该结点的兄弟结点。**



二叉树转换成树

对于一棵转换后的二叉树，如何还原成原来的树。步骤是：

1. 加虚线。若某结点i是其**父结点的左子树的根结点**，则将该结点i的右子结点以及沿右子链不断地搜索**所有的右子结点**，将所有这些**右子结点**与**i结点的父结点**之间加虚线相连，如下图(a)所示。
2. 去连线。去掉二叉树中**所有父结点与其右子结点之间的连线**，如下图(b)所示。
3. 规整化。将图中各结点**按层次排列**且将所有的**虚线变成实线**，如下图(c)所示。

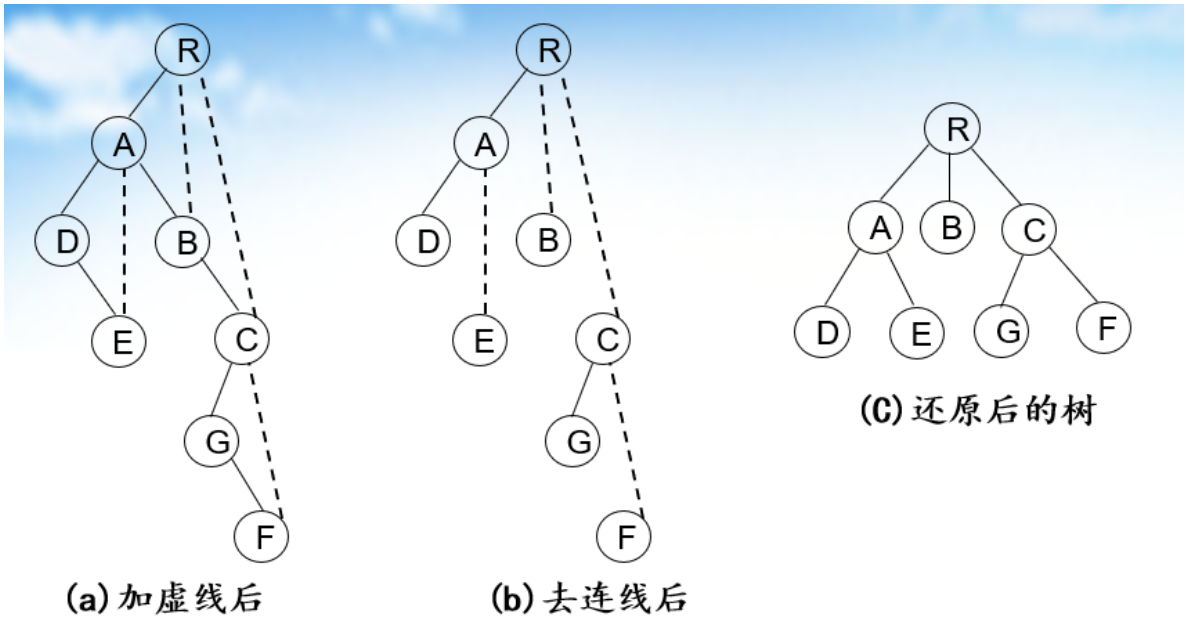


图 二叉树向树的转换过程

森林转化为二叉树

当一般的树转换成二叉树后，二叉树的右子树必为空。若把森林中的**第二棵树**(转换成二叉树后)的**根结点**作为**第一棵树(二叉树)的根结点的兄弟结点**，则可导出森林转换成二叉树的转换算法如下：

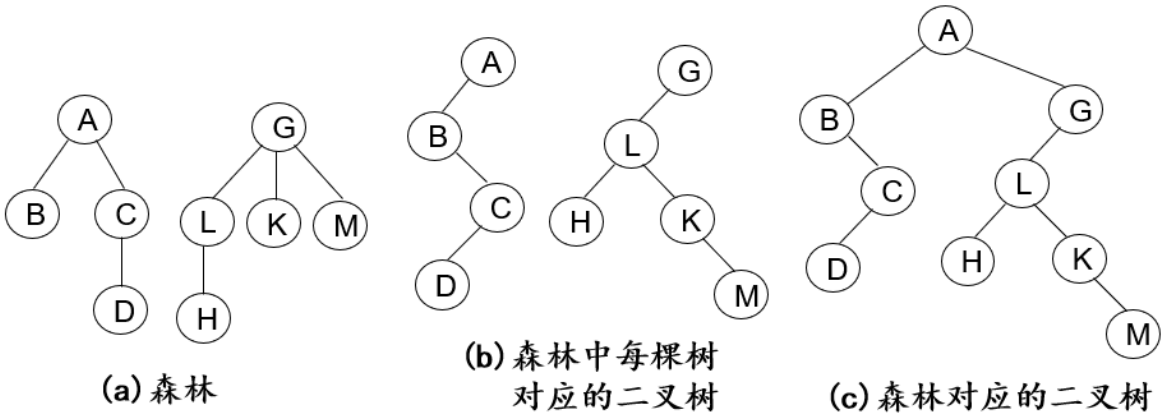
设F={T1, T2,…,Tn}是森林，则按以下规则可转换成一棵二叉树B=(root，LB，RB)

1. 若n=0，则B是空树。
2. 若n>0，则二叉树B的根是森林T1的根root(T1)，B的左子树LB是B(T11,T12, …,T1m)，其中T11,T12, …,T1m是T1的子树(转换后)，而其右子树RB是从森林F'={T2, T3,…,Tn}转换而成的二叉树。

转换步骤：

① 将F={T1, T2,…,Tn} 中的每棵树转换成二叉树。

② 按给出的森林中树的次序，从最后一棵二叉树开始，每棵二叉树作为前一棵二叉树的根结点的右子树，依次类推，则第一棵树的根结点就是转换后生成的二叉树的根结点，如下图所示。



二叉树转化为森林

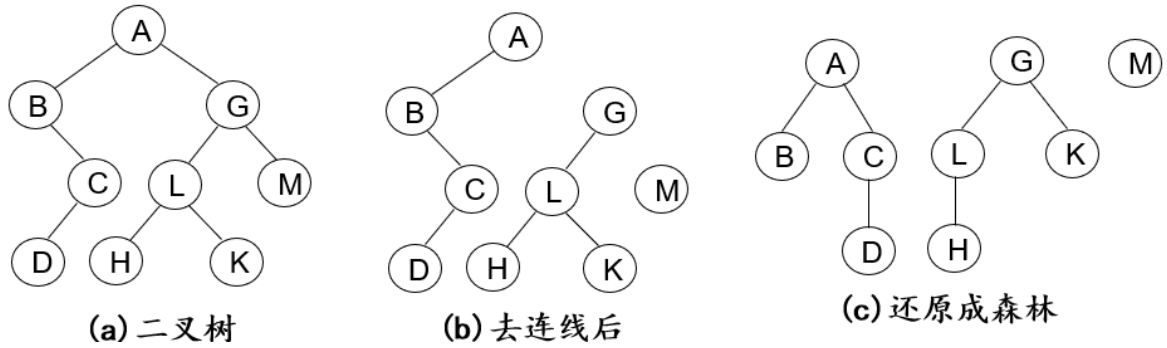
若B=(root，LB，RB)是一棵二叉树，则可以将其转换成由若干棵树构成的森林：F={T1, T2,⋯,Tn} 。

转换算法：

1. 若B是空树，则F为空。
2. 若B非空，则F中第一棵树T1的根root(T1)就是二叉树的根root， T1中根结点的子森林F1是由树B的左子树LB转换而成的森林；F中除T1外其余树组成的森林F'={T2, T3,⋯,Tn} 是由B右子树RB转换得到的森林。

上述转换规则是递归的，可以写出其递归算法。以下给出具体的还原步骤。

1. 去连线。将二叉树B的根结点与其右子结点以及沿右子结点链方向的所有右子结点的连线全部去掉，得到若干棵孤立的二叉树，每一棵就是原来森林F中的树依次对应的二叉树，如下图(b)所示。
2. 二叉树的还原。将各棵孤立的二叉树按二叉树还原为树的方法还原成一般的树，如下图(c)所示。



3.8树的应用

树的先根遍历

```
template<class T>
void CSnode<T>::preTree(){
    if (!this)return;
    cout << this->data;
    preTree(this->Child1);
    preTree(this->Sibling);
}
```

树的后根遍历

```
template<class T>
void CSnode<T>::postTree() {
    if (!this)return;
    preTree(this->Child1);
    preTree(this->Sibling);
    cout << this->data;
}
```

树的层序遍历

```
template<class T>
void CSnode<T>::levelTree(CSnode<T> &head) {
    CSnode<T>* p = &head;
    queue<CSnode<T>*> tqueue;
    while (p||!tqueue.empty()) {
        if (!p && !tqueue.empty()) {
            p = tqueue.front(); tqueue.pop(); }
        p->visit();
        cout << " ";
        if (p->Child1)tqueue.push(p->Child1);
        p = p->Sibling;
    }
}
```

例题

1. 编写递归算法：对于二叉树中每一个元素值为x的结点，删去以它为根的子树，并释放相应的空间。

   ```
   template<class T>
   void BTree<T>::deleteCTree(T e){
       if (!this)return;
       if (this->data == e)this->deleteCTree();
       else {
           this->lc->deleteCTree(e);
           this->rc->deleteCTree(e);
       }
   }

   template<class T>
   void BTree<T>::deleteCTree(){
       if (!this)return;
       BTree<T>* p = this;
       p->lc->deleteCTree();
       p->rc->deleteCTree();
       p->~BTree();
   }
   ```

2. 判断二叉树是否相似。

   二叉树的相似，是指俩二叉树有**相似的结构**，同样的节点位置可以有不同的值，但要么都是空节点，要么都不是空节点。只要有一个节点位置不相似，即一个是空节点，另一个是非空节点，则整个二叉树都不相似。

```
template<class T>
bool BTree<T>::likeTree(BTree<T>* A, BTree<T>* B) {
    if (!A && !B)return true;
    if (!A || !B)return false;
    int l, r;
    l = likeTree(A->lc, B->lc);
    r = likeTree(A->rc, B->rc);
    return (l && r);
}
```

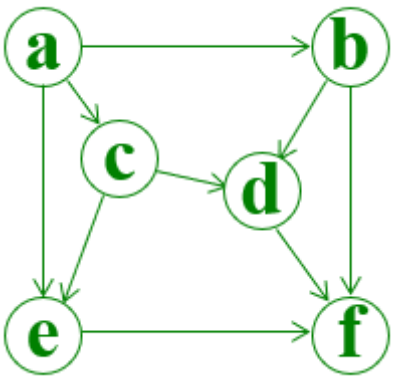
3. 求二叉链表T中值为x的数据元素所在的层次。

```
template<class T>
int BTree<T>::BTL(T e) {
    if (!this) return 0;
    if (this->data == e)return 1;
    int lk, rk;
    lk = this->lc->BTL(e);
    rk = this->rc->BTL(e);
    if (!lk && !rk)return 0;
    else return lk > rk ? lk+1 : rk+1;
}
```

4.图 Graph

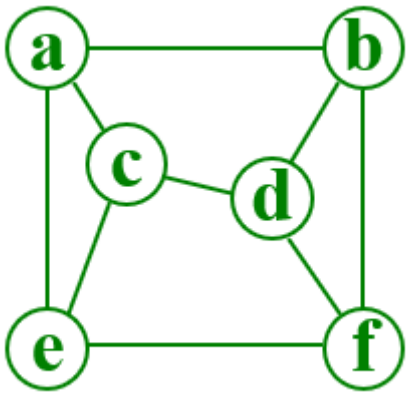
4.1图的基本概念

- 图是一种非线性数据结构；
- 图的**数据元素**是顶点；
- 图的**数据对象**是顶点集；
- 图结构主要研究两顶点之间的连通关系。
- 顶点(Vertex)：图的数据元素。
- 顶点集V：研究的所有顶点的集合。
- 关系集VR：两个顶点之间关系的集合。
- 例:顶点：a, b, c, d, e, f;V={ a, b, c, d, e, f }；VR:<a,b>,<d,f>等。

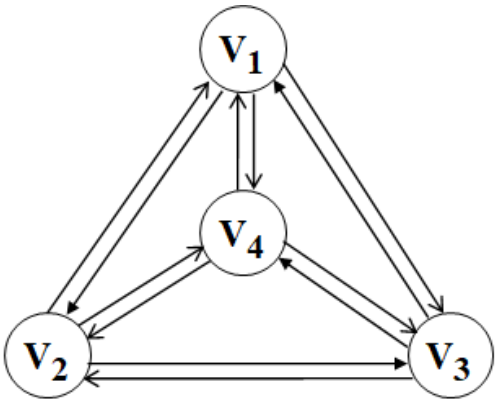


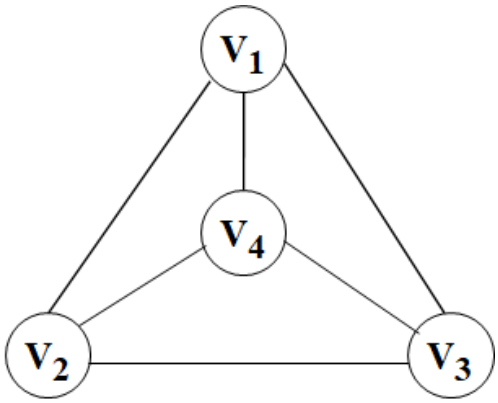
- 弧(Arc)：如果<v,w>∈VR，则<v,w>表示从v到w的一条弧(具有次序关系)。
- 弧头(Head)：箭头端顶点；

- 弧尾(Tail)：非箭头端顶点。
- 边(Edge)：如果<v,w>∈VR必有 <w,v>∈VR, 则称(v, w)是v和w之间的一条边;边**没有次序关系**，即(v, w)和(w, v)表示同一条边。

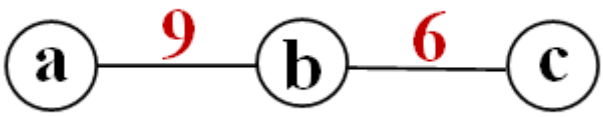


- 有向图(digraph)：由顶点集和**弧集**构成的图。
- 无向图(undigraph)：由顶点集和**边集**构成的图。
- 设：<vi,vj>∈VR，vi≠vj ； n表示图中**顶点数目**；m表示图中**边(或者弧)的数目**；
  - 则对于有向图：0≤m≤n(n-1)；
  - 对于无向图：0≤m≤n(n-1)/2。
  - n=0时V是空集；m=0时图或许有1个顶点。
- 有向完全图：m=n(n-1)的有向图。
- (无向)完全图：m=n(n-1)/2的无向图。

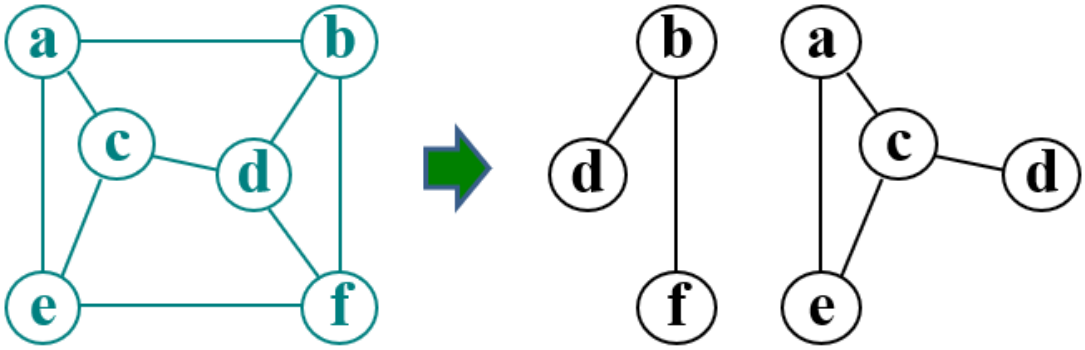




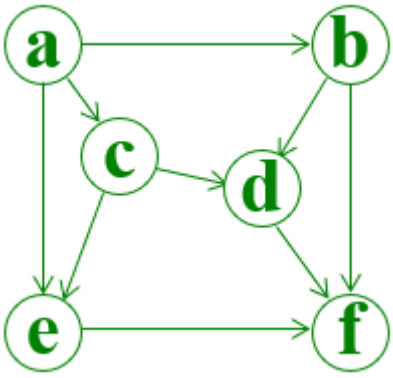
- 稀疏图：m<< n(n-1)的图。
- 稠密图：m接近于n(n-1)/2的图。
- 权值：标示边或弧某种特性的参数值。
  - 例如，



- 网：带权的图。（在不会引起歧义时，网也直接称为图。）
- 有向网：带权的有向图。
- 无向网：带权的无向图。
- 图的形式定义：Graph=(V，E)，其中，V是顶点的非空有限集，E是弧(或者边)的有限集。=>图是由一个**顶点集V**和一个**弧集(或边集)**构成的数据结构。**例**G=(V, E)，V={a, b}，E={<a,b>}。
- 子图：设图G=(V，E)，G'=(V'，E')。若V'∈V且E'∈E，则称G'为G的子图。



- 无向图
  - 邻接点(Adjacent)：对于G=(V，E)，如果边(v，v')∈E，则称顶点v和v'**互为邻接点**(即v和v'相邻接)，边<v，v'>与v和v'**相关联**。
  - 顶点v的度：和顶点v**相邻接的边的数目**，记为**TD(v)**。
- 有向图
  - 邻接点：对于G=(V, E)，如果弧<v,v'>∈E，则称v邻接**到**v'，v'邻接**自**v；弧<v,v'>和顶点v，v'相关联。
  - 入度ID(v)：以**v为弧头**（箭头指向）的**弧的数目**；
  - 出度OD(v)：以**v为弧尾**的**弧的数目**；
  - 顶点v的度**TD(v)**=**ID(v)+OD(v)**。



  - 对于有向图：ID(c)=1，OD(c)=2，即TD(c)=ID(c)+OD(c)=3。
- 设无向图有n个顶点vi，m条边，vi的度为TD(vi)，则

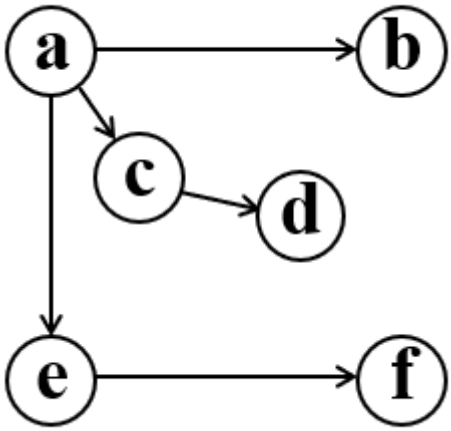
$$m = \frac{1}{2} \sum_{i=1}^{n} TD(v_i)$$

- 从v到v'的路径：v = v0, v1, ..., vk = v'，其中，(vj-1，vj)∈E，1≤j≤k。
- 路径长度：**路径**上的**边(或弧)**的数目。
- 回路(亦称为环)：第1个顶点和最后1个顶点相同的一条路径。
- 简单路径：j≠k时vj≠vk的路径。
- 简单回路：v0=vk的**简单路径**。
- v和v'连通：v和v'之间有路径。
- 连通图：无向图中任意两个顶点都连通。
- 连通分量：无向图中的**极大连通子图**。

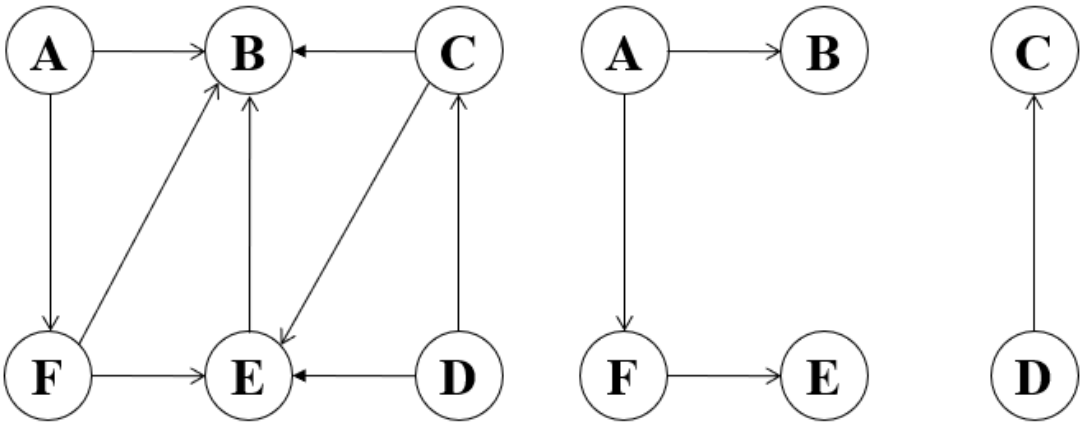
- 强连通图：在**有向图**中，对任意vi和vj(i≠j)，从vi到vj和从vj到vi都有路径
- 强连通分量：**有向图中**的极大连通子图。
- **例题**：设图G=(V, E)，V={ a, b, c, d, e }，E={<a, b>, <a, c>, <b, d>, <c, e>, <d, c>, <e, d>}

  回答下列问题：

  1. 求与顶点b相关联的弧；(有两条)
  2. 是否存在从c到b的路径? 不存在，因为 <c, e>, <e, d>, <d, c>
  3. 求ID(d)、OD(d)、TD(d)；ID(d)=2，OD(d)=1，TD(d)=3
  4. 该有向图是否是强连通图? (不是)
  5. 画出各个强连通分量。三个：①；②；③ **<c, e>，<e, d>，<d, c>。**
- **生成树：设连通图G共有n个顶点，则含有图G的n个顶点、且有n-1条边的连通图，称为图G的一棵生成树。**
- **一棵有n个顶点的生成树有且仅有n-1条边，但有n个顶点和n-1条边的图不一定是生成树。**
- **在有n个顶点的图中, 如果边多于n-1条, 则一定有环; 如果边少于n-1条, 则是非连通图。**
- **有向树：只有一个顶点的入度为0，其它顶点的入度都为1的有向图。**



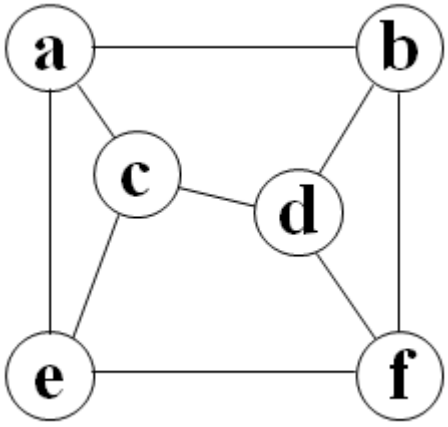
- **生成森林：含有图中全部n个顶点的m棵互不相交的有向树(共有n-m条弧)。**



4.2图的存储结构

图的邻接矩阵

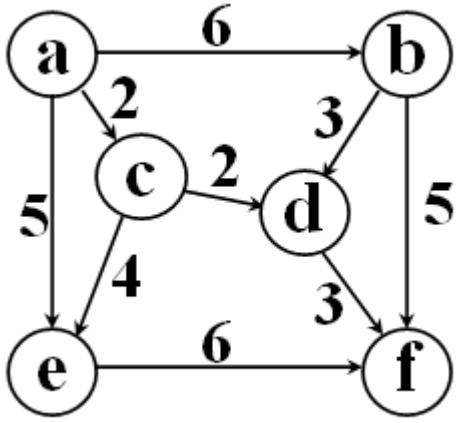
对于无向图 Ve={a, b, c, d, e, f }



其邻接矩阵为：

Vr	a	b	c	d	e	f
a	0	1	1	0	1	0
b	1	0	0	1	0	1
c	1	0	0	1	1	0
d	0	1	1	0	0	1
e	1	0	1	0	0	1
f	0	1	0	1	1	0

对于有向网Ve={a, b, c, d, e, f }



其邻接矩阵为：

Vr	a	b	c	d	e	f
a	0	6	2	0	5	0
b	0	0	0	3	0	5
c	0	0	0	2	4	0
d	0	0	0	0	0	3
e	0	0	0	0	0	6
f	0	0	0	0	0	0

定义：邻接矩阵，也称为数组表示法；n ——图中顶点的数目；Ve[n]——存放n个顶点的数据元素值；Vr[n] [m]=(wij)n×n——存放vi和vj的邻接值，如果vi和vj为非邻接点，记wij=0或∞。

操作：

1. **定义**

```
template<class T>
class MatrixGraph {
public:
    int type;//1为无向图，0为有向图
    int size;
    int n;
    T Ve[MAXSIZE];
    int Vr[MAXSIZE][MAXSIZE] = { 0 };
    MatrixGraph(int t=1) { this->type = t; this->n = 0; }
}
```

2. **创建矩阵**

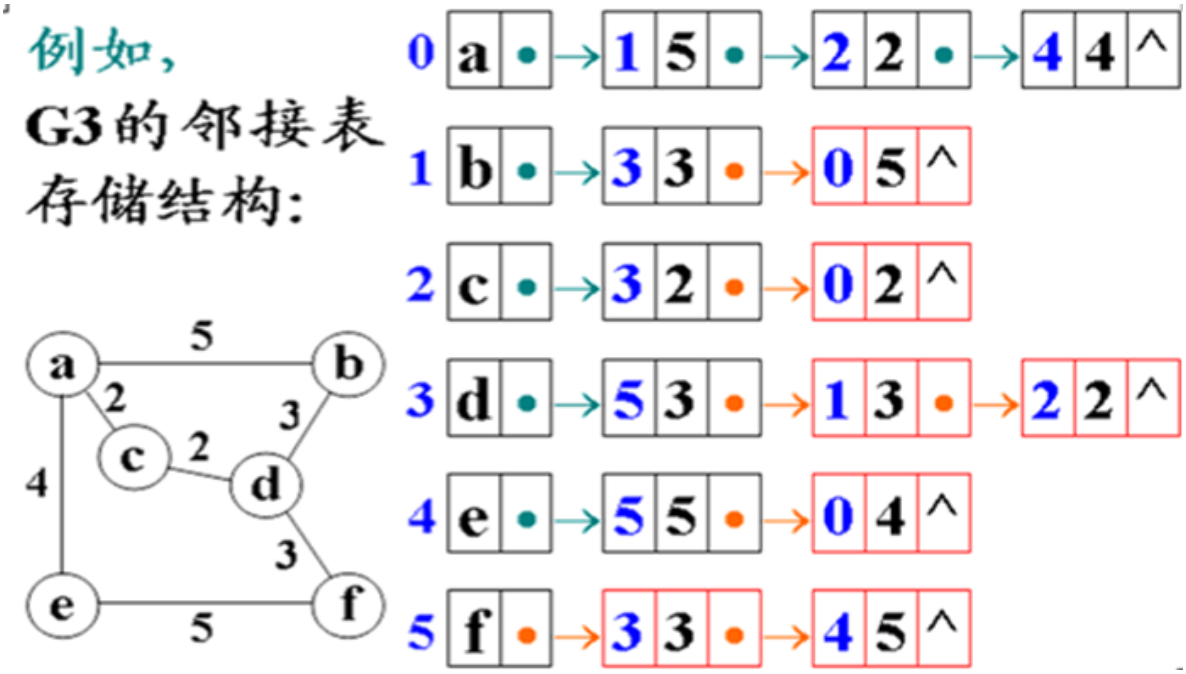
```
template<class T>
void MatrixGraph<T>::Create() {
    int v1, v2, w;
    this->n = 0;
    //根据表类型创建有向图或无向图
    if (this->type) {//有向图
        cin >> v1 >> v2;
        while (v1 != v2) {
            this->Vr[v1][v2] = 1;
            this->Vr[v2][v1] = 1;
            cin >> v1 >> v2;
            this->n++;
        }
    }
    else {//无像图
        cin >> v1 >> v2 >> w;
        while (v1 != v2) {
            this->Vr[v1][v2] = w;
            cin >> v1 >> v2 >> w;
            this->n++;
        }
    }
}
```

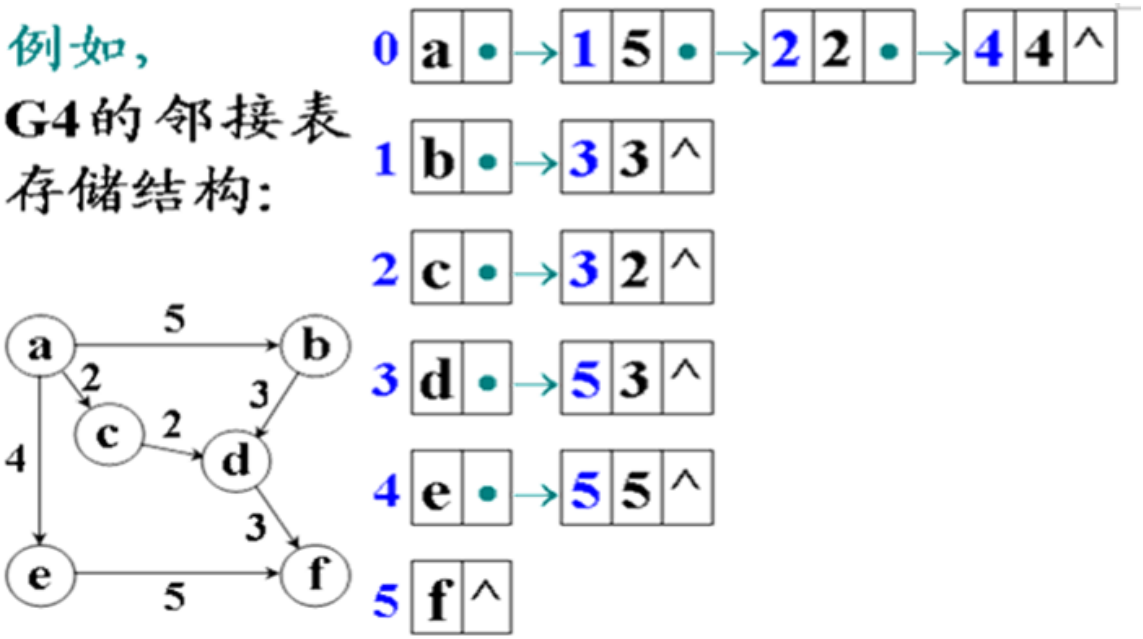
3. **求某一个结点的度**

```
template<class T>
int MatrixGraph<T>::TD(T e){
    int i, site = 0;
    int ID = 0,OD=0;
    for (i = 0; i < this->size; i++)
        if (Ve[i] == e) {
            site = i; break;
        }
    if (site == -1 || site == this->size) {
        cout << "No such element!";
        return NULL;
    }
    if (this->type) {
        for (i = 0; i < this->size; i++) {
            if (this->Vr[i][site])
                ID++;
        }
        return ID;
    }
    else {
        for (i = 0; i < this->size; i++) {
            if (this->Vr[i][site])
                ID++;
            if (this->Vr[site][i])
                OD++;
        }
        return ID + OD;
    }
}
```

图的邻接表

邻接表是图的一种链式存储结构；图中每一个顶点对应1个结点，并由所有顶点结点构成一个顺序表(结构)；图中每一个顶点建立一条链，该链由它的所有邻接点构成。





- **在无向图邻接表中，1条边对应2个表结点，即结点总数是边数的2倍；**
  - **顶点的度=邻接点链表中结点的数目。**
- **在有向图邻接表中，1条弧对应1个表结点，即结点总数=弧的数目；**
  - **顶点的出度=邻接点链表中结点数目。**

在有向图邻接表中，求顶点的入度比较复杂，需搜索所有的邻接点链表后才能确定。例在G4的邻接点链表中，ID(b)=1, ID(d)=2。可以通过对有向图建立逆邻接链表，则顶点的入度=逆邻接点链表中结点的数目。

操作：

1. **定义**

```
template<class T>
class ENode {
public:
    int Vi;
    int Wi;
    ENode<T>* next;

    ENode(int v, int w);
    ENode() {};
};

template<class T>
class VNode {
public:
    T data;
    ENode<T> *head;
};

template<class T>
class ListGraph {
public:
    VNode<T> Ve[MAXSIZE];
    int vn;
    int en;
    int type;//1为无向图，2为有向图
```

```
    bool visited[MAXSIZE];
};
```

2. **创建矩阵**

```
template<class T>
void ListGraph<T>::create() {
    int v1, v2, w;
    this->en = 0;
    ENode<T>* p;
    //根据表类型创建有向图或无向图
    if (this->type) {//无向图
        cin >> v1 >> v2;
        while (v1 != v2) {
            p = new ENode<T>(v2, 1);
            p->next = this->Ve[v1].head;
            this->Ve[v1].head = p;
            p = new ENode<T>(v1, 1);
            p->next = this->Ve[v2].head;
            this->Ve[v2].head = p;
            this->en++;
            cin >> v1 >> v2;
        }
    }
    else {//有像图
        cin >> v1 >> v2 >> w;
        while (v1 != v2) {
            p = new ENode<T>(v2, w);
            p->next = this->Ve[v1].head;
            this->Ve[v1].head = p;
            this->en++;
            cin >> v1 >> v2>>w;
        }
    }
}
```

4.3图的遍历

从图中某个顶点出发，按照一定规则访问图中的所有顶点，且图中的每个顶点仅被访问一次。

搜索有两种途径：

- **深度优先搜索(Depth First Search)**
- **广度优先搜索(Breadth First Search)**

深度优先搜索

基本思想：

①从图中某个顶点v出发，访问该顶点；

②查找顶点v的第一个未被访问的邻接点v1，访问该顶点，v<=v1；

③重复第②步操作，直到图中没有未被访问的顶点为止。

操作

1. **邻接矩阵中从序号为v的顶点出发深度优先搜索遍历连通图G**

```
template<class T>//找出第一个邻接点
int MatrixGraph<T>::firstAdjV(int v){
    for (int i = 0; i < this->size; i++)
        if (this->Vr[v][i] != 0)
            return i;
    return -1;
}

template<class T>//在当前邻接点基础上找下一个邻接点
int MatrixGraph<T>::nextAdjV(int v, int w){
    for (int i = w+1; i < this->size; i++)
        if (this->Vr[v][i] != 0)
            return i;
    return -1;
}

template<class T>
void MatrixGraph<T>::DFS(int v){
    this->visit(v); this->visited[v] = true;
    for (int k = this->firstAdjV(v); k >= 0; k = this->nextAdjV(v,k)) {
        if (!this->visited[k])
            this->DFS(k);
    }
}
```

2. **邻接矩阵中从序号为v的顶点出发深度优先搜索遍历非连通图**

```
template<class T>
void MatrixGraph<T>::DFSTraverse(int v) {
    this->setVisited();
    DFS(v);
    for (int i = 0; i < this->size; i++)
        if (!this->visited[i])
            DFS(i);
}
```

3. **邻接表中从序号为v的顶点出发深度优先遍历连通图**

```
template<class T>
void ListGraph<T>::DFS(int v){
    this->visit(v); this->visited[v] = true;
    ENode<T>* p = this->Ve[v].head;
    while (p) {
        if (!this->visited[p->Vi])
            DFS(p->Vi);
        p = p->next;
    }
}
```

4. **邻接表中从序号为v的顶点出发深度优先遍历非连通图**

```
template<class T>
void ListGraph<T>::DFSTraverse(int v) {
    this->setVisited();
    this->DFS(v);
    for (int i = 0; i < this->vn; i++)
        if (!this->visited[i])
            this->DFS(i);
}
```

广度优先搜索

基本思想：

- **从连通图中的某顶点v出发，访问顶点v；依次访问v的所有未被访问过的邻接点vk；**
- **按vk被访问的次序依次访问未被访问过的邻接点，直到访遍图中所有的顶点为止。**

操作：

1. **邻接矩阵中从序号为v的顶点出发深度优先搜索遍历连通图G**

```
template<class T>
void MatrixGraph<T>::BFS(int v){
    queue<int> vq;
    this->visited[v] = true;
    vq.push(v);
    while (!vq.empty()) {
        v = vq.front();
        vq.pop();
        this->visit(v);
        for (int i = 0; i < n; i++)
            if (this->Vr[v][i] && !this->visited[i]) {
                vq.push(i);
                this->visited[i] = true;
            }
    }
}
```

2. **邻接矩阵中从序号为v的顶点出发深度优先搜索遍历非连通图**

```
template<class T>
void MatrixGraph<T>::BFSTraverse(int v){
    this->setVisited();
    this->BFS(v);
    for (int i = 0; i < this->size; i++)
        if (!this->visited[i])
            BFS(i);
}
```

3. **邻接表中从序号为v的顶点出发深度优先遍历连通图**

```
template<class T>
void ListGraph<T>::BFS(int v){
    ENode<T>* p = NULL;
    queue<ENode<T>*> q;
    q.push(this->Ve[v].head);
    this->visit(v);
```

```
    this->visited[v] = true;
    while (!q.empty()) {
        p = q.front();
        q.pop();
        while (p) {
            if (!this->visited[p->Vi]) {
                this->visit(p->Vi);
                this->visited[p->Vi] = true;
                q.push(this->Ve[p->Vi].head);
            }
            p = p->next;
        }
    }
}
```

4. **邻接表中从序号为v的顶点出发深度优先遍历非连通图**

```
template<class T>
void ListGraph<T>::BFSTraverse(int v) {
    this->setVisited();
    this->BFS(v);
    for (int i = 0; i < this->vn; i++)
        if (!this->visited[i])
            this->BFS(i);
}
```

4.4最小生成树

顶点相通算法

- **j0=v1;**
- **①{ if j0=v2结束, else寻找j0的下1个邻接点v}**
- **if找到j0的下1个邻接点v，继续①的操作；else回溯到前1个邻接点再+1，继续①的操作，直到没有新的邻接点为止。**

代码：

1. **邻接矩阵**

```
template<class T>
bool MatrixGraph<T>::path(int v1, int v2){
    this->visited[v1] = true;
    for (int i = 0; i < this->size; i++)
        if (this->Vr[v1][i] && !this->visited[i]) {
            if (i == v2)
                return true;
            if (this->path(i, v2))
                return true;
        }
    return false;
}
```

2. **邻接表**

```
template<class T>
```

```
bool ListGraph<T>::path(int v1,int v2) {
    ENode<T>* p = this->Ve[v1].head;
    int t;
    while (p) {
        t = p->Vi;
        if (!this->visited[t]) {
            this->visited[t] = true;
            if (t == v2)
                return true;
            if (this->path(t, v2))
                return true;
        }
        p = p->next;
    }
    return false;
}
```

最小生成树

在遍历过程中，将所有经过的边记为T，没有经过的边(即回边)记为B。 则图G'=(V，T)是图G的一棵生成树（n个顶点，n-1条边，连通。）。

根据生成方式，有深度优先生成树和广度优先生成树两种。

最小生成树(Minimum Cost Spanning Tree)是在一个无向连通网的所有生成树中，代价之和最小的生成树。

性质：设G=(V, E)是一连通网，U是顶点集V的一个非空子集。若(u, v)是一条具有最小权值的边, 其中u∈U，v∈V-U，则存在一棵包含边(u, v)的最小生成树。

算法名称	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$ (n：顶点数)	$O(mn)$ (m：边数)
适应范围	稠密图	稀疏图

Prim算法——加点法

基本思想:

- **任取一个顶点v作为生成树的根，然后往生成树上添加新顶点u:**
- **添加的顶点u和已经在生成树上的某个顶点连通，并且在所有与生成树的连通边中，该边的权值最小;**
- **继续往生成树上添加顶点，直至生成树上含有n个顶点为止。**

代码实现:

```
template<class T>
void MatrixGraph<T>::prim(int v) {
    int i ,j,z,k;
    int lowcost[2][MAXSIZE];
    //第0行存储双亲结点，第1行存储当前树中结点与不在树中结点的邻接点的权值
    this->setVisited();
    this->visited[v] = true;

    //根据根节点，对最小代价表进行初始化
    for (i = 0; i < this->size; i++) {
        lowcost[0][i] = v;
        lowcost[1][i] = this->Vr[v][i];
    }

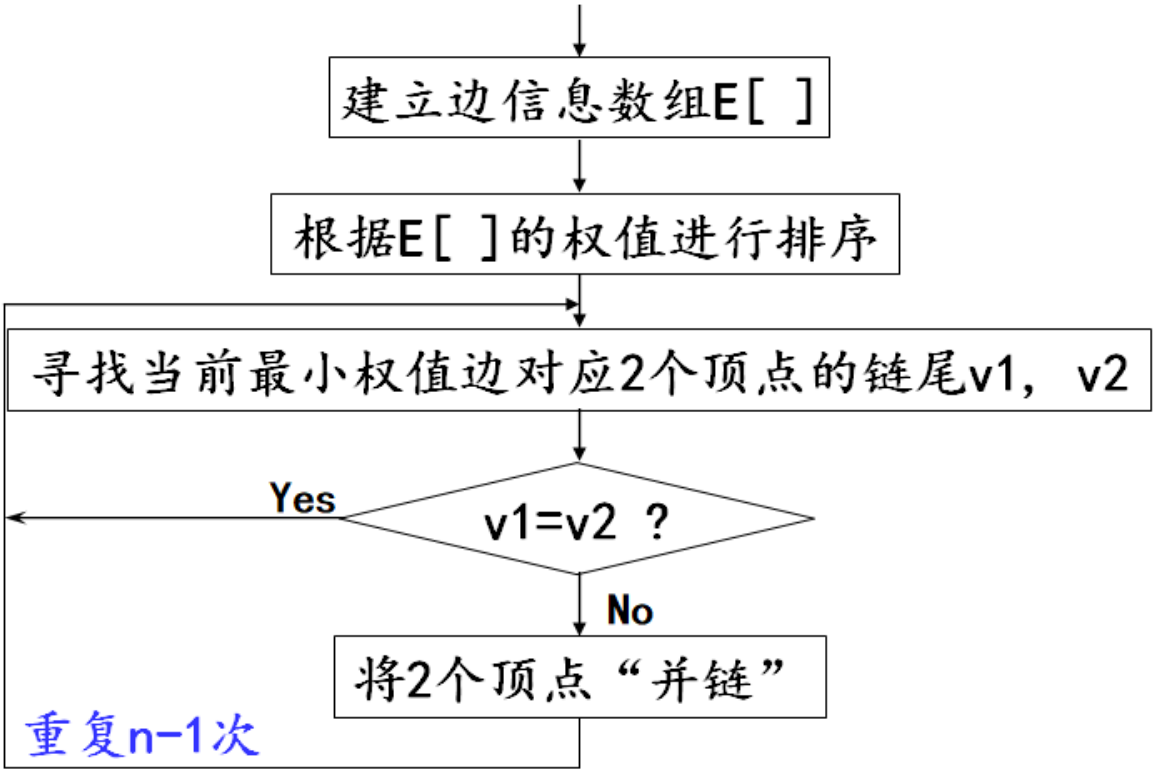
    //进行size-1次，找到除根节点外剩下的结点
    for (i = 0; i < this->size - 1; i++) {
        k = this->getMinEdge(lowcost[1]);//找到当前权值最小结点
        this->visited[k] = true;
        for (j = 0; j < this->size; j++) {
            //判断条件有两个，前提都是结点未访问，一个是lowcost中权值未赋值，仍未0，一种是
lowcost权值已赋值，但权值小于新加入结点与同一邻接点的权值
            if (!this->visited[j]&&!lowcost[1][j]||!this->visited[j] && this-
>Vr[k][j] < lowcost[1][j] && this->Vr[k][j]) {
                lowcost[0][j] = k;
                lowcost[1][j] = this->Vr[k][j];
            }
        }
    }
}

//根据lowcost表找到权值最小结点
template<class T>
int MatrixGraph<T>::getMinEdge(int lowcost[]){
    int min = INT_MAX,locate=-1;
    for (int i = 0; i < this->size; i++)
    {
        //权值不能为0
        if(lowcost[i]<min&&!this->visited[i]&&lowcost[i]){
            min = lowcost[i];
            locate = i;
        }
    }
    return locate;
}
```

Kruskal算法——加边法

基本思路：为使生成树上边的权值之和达到最小，应使生成树中每一条边的权值尽可能地小(都能最小则最佳)。

具体做法：先构造一个只含n个顶点的子图 S。然后从权值最小的边开始，若它的添加不使S中产生回路，则在S上加上这条边。如此重复，直至加上n-1条边为止。



算法实现：

```
struct Edge{
    int v1;
    int v2;
    int wi;
};

void edgeSort(Edge e[], int n) {
    int i,j;
    Edge p;
    for(i=0;i<n;i++)
        for(j=0;j<n-i-1;j++)
            if (e[j].wi > e[j + 1].wi) {
                p = e[j];
                e[j] = e[j + 1];
                e[j + 1] = p;
            }
}

static int edgeFind(int link[], int k) {
    while (link[k] >= 0)k = link[k];
    return k;
}

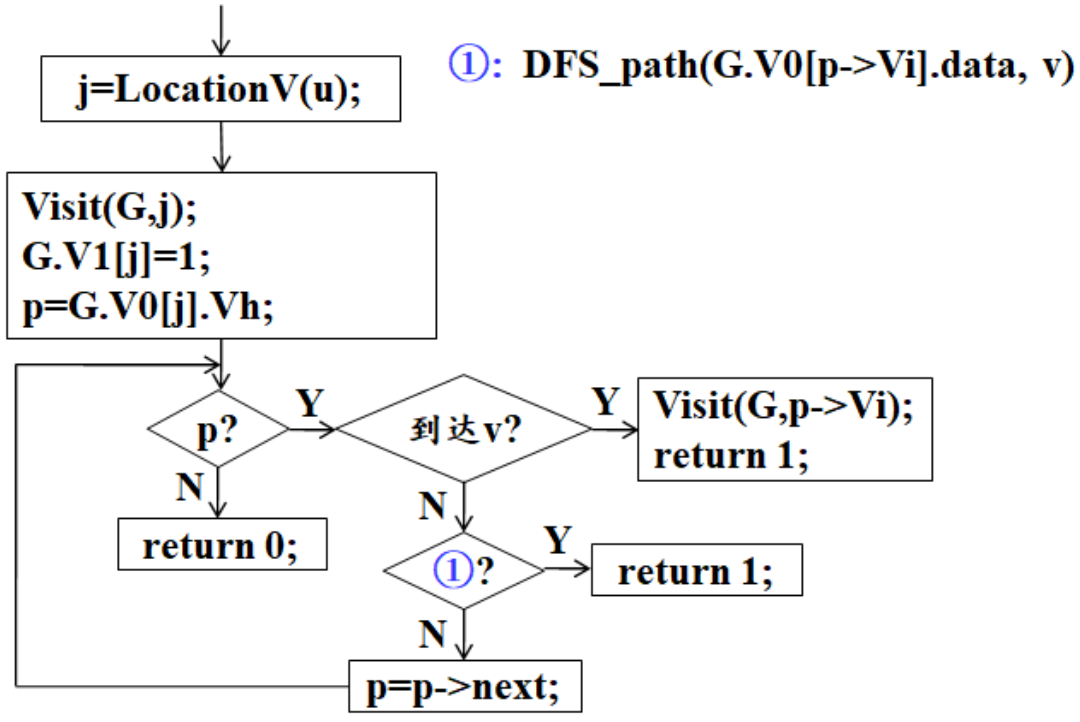
//获得所有边的信息
template<class T>
void MatrixGraph<T>::edges(Edge e[]){
    int n = 0;
    for(int i=0;i<this->size;i++)
        for (int j = i; j < this->size; j++)
            if(this->Vr[i][j]){
                e[n].v1 = i;
                e[n].v2 = j;
                e[n++].wi = this->Vr[i][j];
```

```
            }
}

template<class T>
void MatrixGraph<T>::Kruskal(Edge e[]){
    int link[MAXSIZE];
    edgeSort(e, this->size);/对边从小到大排序
    int i, v1, v2;
    for (i = 0; i < this->size; i++)
            link[i] = -1;
    for (i = 0; i < this->n; i++) {
        v1 = edgeFind(link, e[i].v1);
        v2 = edgeFind(link, e[i].v2);
        if (v1 != v2) {
            if(link[e[i].v2]>=0)
                link[e[i].v1] = e[i].v2;
            else
                link[e[i].v2] = e[i].v1;
            cout << e[i].v1 << " " << e[i].v2
                << " " << e[i].wi << endl;
        }
    }
}
```

4.5最短路径

例：求一条从顶点u到顶点v的简单路径 (如求从顶点b到顶点k的一条简单路径)。



代码实现：

```
template<class T>
bool ListGraph<T>::path_DFS(int v1,int v2){
    this->visit(v1);
    this->visited[v1] = true;
    ENode<T>* p = this->Ve[v1].head;
    while (p) {
```

```
        if (p->Vi == v2) {
            this->visit(v2); return true;
        }
        if (!this->visited[p->Vi] && this->path_DFS(p->Vi, v2))
            return true;
        p = p->next;
    }
    return false;
}
```

带权图中的最短路径问题

用顶点表示城市，边表示城市之间的通路，权值表示城市之间的距离。

- **如何判断两个城市之间是否有通路?**
- **如果有通路，走哪条路最短?**

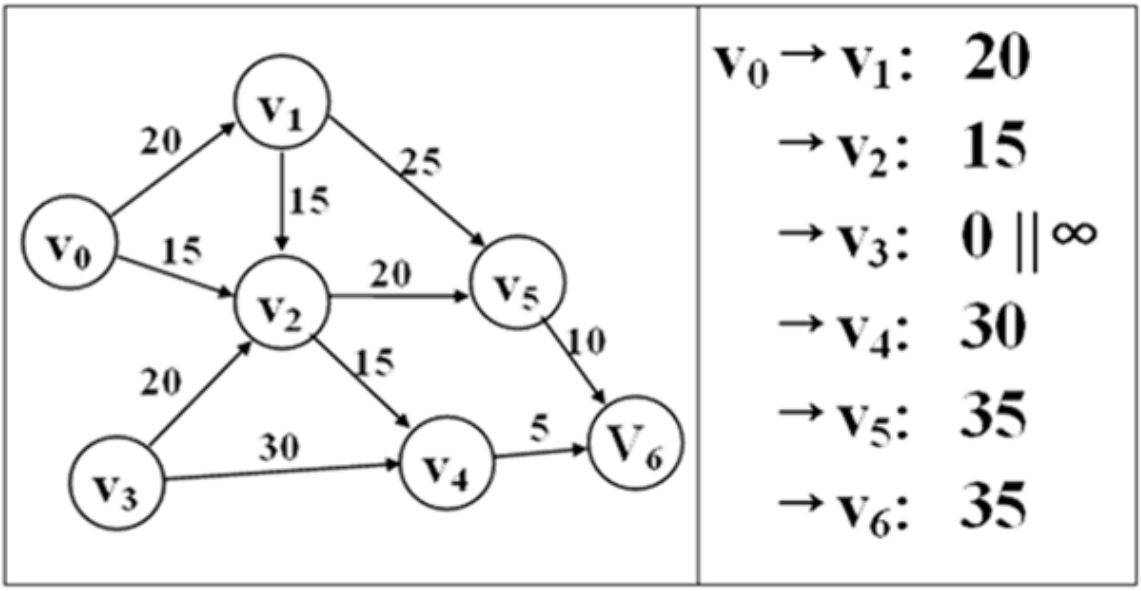
求最短路径时，所求的路径长度为路径上各边的权植之和。

假设如果城市之间有通路，其权值>0，否则，其权值 = 0 (或∞)。

单源最短路径问题

设顶点v是指定源点, 要求在有向网G中找到从源点v到其它各顶点的最短路径。

例：求下图中v0到其它各顶点的最短路径。



Dijkstra算法

基本思想：把图中所有顶点分成两组，第一组U包含已确定最短路径的顶点，初始值只含源点v；第二组V-U包含尚未确定最短路径的顶点，初始值含有图中除源点v之外的其它顶点。

按路径长度递增的顺序计算源点u到其它各顶点的最短路径，逐个把第二组中的顶点加到第一组中去，直至U=V为止。

代码实现：

1. **邻接矩阵**

```
template<class T>
void MatrixGraph<T>::Dijkstra(int v){
```

```
    int i,j,k,s,min,length[MAXSIZE],list[MAXSIZE];
    this->setVisited();
    this->visited[v] = true;
    for (i = 0; i < this->vn; i++) {
        length[i] = this->Vr[v][i];
        list[i] = v;
    }
    list[v] = -1;
    for (i = 0; i < this->vn; i++) {
        min = INT_MAX; k = 0;
        for (j = 0; j < this->vn; j++)
            if (!this->visited[j] && length[j] < min&&length[j]>0) {
                k = j; min = length[j];
            }
        this->visited[k] = true;
        for (j = 0; j < this->vn; j++) {
            if (this->Vr[k][j] == 0|| this->visited[j])
                continue;
            s = length[k] + this->Vr[k][j];
            if (s < length[j]||length[j]==0) {
                length[j] = s;
                list[j] = k;
            }
        }
    }
}

//输出所有最短路径
templaet<class T>
void pathPrint(int list[],int length[]){
    for (i = 0; i < this->vn; i++) {
        j = i;
        cout << "Path:" << j;
        j = list[j];
        while (j!=-1) {
            cout << "->" << j;
            j = list[j];
        }
        cout <<"\tlength:"<<length[i]<<endl;
    }
}
```

2. **邻接表**

```
template<class T>
void ListGraph<T>::Dijkstra(int v) {
    int i, j, k, s, min, length[MAXSIZE] = { 0 }, list[MAXSIZE];
    this->setVisited();
    this->visited[v] = true;
    ENode<T>* p = this->Ve[v].head;
    while (p) {
        length[p->Vi] = p->Wi;
        list[p->Vi] = v;
        p = p->next;
    }
    list[v] = -1;
    for (i = 0; i < this->vn; i++) {
```

```
        min = INT_MAX; k = 0;
        for (j = 0; j < this->vn; j++)
            if (!this->visited[j] && length[j] < min && length[j]>0) {
                k = j; min = length[j];
            }
        this->visited[k] = true;
        p = this->Ve[k].head;
        while (p) {
            if (!this->visited[p->Vi])
            {
                s = length[k] + p->Wi;
                if (s < length[p->Vi] || length[p->Vi] == 0) {
                    length[p->Vi] = s;
                    list[p->Vi] = k;
                }
            }
            p = p->next;
        }
    }
}

//输出所有最短路径
templaet<class T>
void pathPrint(int list[],int length[]){
    for (i = 0; i < this->vn; i++) {
        j = i;
        cout << "Path:" << j;
        j = list[j];
        while (j!=-1) {
            cout << "->" << j;
            j = list[j];
        }
        cout <<"\tlength:"<<length[i]<<endl;
    }
}
```

Floyd算法

问题描述：求图中任意一对顶点之间的最短路径。

Floyd算法的基本思想：

- **从vi到vj的所有可能存在的路径中，选出一条加权长度最短的路径**
- **=>若<vi,vj>存在，则存在路径{vi, vj}——路径中不含其它顶点**
- **=>若<vi, u0>,<u0, vj>存在，则存在路径{vi, u0, vj}; ——路径中还包含u0顶点**
- **=>若<ui, u0>，<u0, u1>，<u1,vj>存在，则存在路径{vi, u0, u1, vj}；——路径中还包含u0和u1两个顶点**
- **=>依次类推。**
- **=>vi到vj的最短路径是上述这些路径中的加权长度最小者。**

问题求解：

- **顶点数：n**
- **路径数：n*n-n**
- **n*n条最短路径长度：Wi[n] [n]**
- **n*n条最短路径：path[n] [n] [n]**

- **——每条路径包含n个顶点的标示：1表示对应顶点在路径上，0表示该顶点不在路径上。例如，path[a] [f] = { 1-0-0-1-1-1-0}表示从顶点a到顶点f的最短路径包括a, d, e和f四个顶点——但没有描述四个顶点出现的次序。**

算法实现：

1. **邻接矩阵**

```
template<class T>
void MatrixGraph<T>::Floyd(){
    int Wi[MAXSIZE][MAXSIZE] = {0}, path[MAXSIZE][MAXSIZE][MAXSIZE] = { 0
};
    int i, j, k,x,size=this->vn;
    for (i = 0; i < size; i++)
        for (j = 0; j < i; j++) {
            Wi[i][j] =Wi[j][i]= this->Vr[i][j];
            if (Wi[i][j]) {
                path[i][j][i] = path[i][j][j] = 1;
                path[j][i][i] = path[j][i][j] = 1;
            }
        }
    for(i=0;i<size;i++)
        for (j = i; j < size; j++) {
            x = -1;
            for(k=0;i!=j&&k<size;k++)
                if (Wi[i][k] && Wi[k][j] && (Wi[i][k] + Wi[k][j] < Wi[i]
[j] || !Wi[i][j])) {
                    x = k;
                    Wi[i][j] = Wi[i][k] + Wi[k][j];
                    Wi[j][i] = Wi[i][k] + Wi[k][j];
                }
            if (x > -1) {
                path[i][j][i] = path[i][j][j] = path[i][j][x] = 1;
                path[j][i][i] = path[j][i][j] = path[j][i][x] = 1;
            }
        }
}
```

2. **邻接表**

```
template<class T>
void ListGraph<T>::Floyd() {
    int Wi[MAXSIZE][MAXSIZE] = { 0 }, path[MAXSIZE][MAXSIZE][MAXSIZE] = {
0 };
    int i, j, k, x, size = this->vn;
    ENode<T>* p;
    for (i = 0; i < size; i++){
        p = this->Ve[i].head;
        while (p) {
            Wi[i][p->Vi] = Wi[p->Vi][i] = p->Wi;
            path[i][p->Vi][i] = path[i][p->Vi][p->Vi] = 1;
            path[p->Vi][i][i] = path[p->Vi][i][p->Vi] = 1;
            p = p->next;
        }
    }
```

```
    for (i = 0; i < size; i++)
        for (j = i; j < size; j++) {
            x = -1;
            for (k = 0; i != j && k < size; k++)
                if (Wi[i][k] && Wi[k][j] && (Wi[i][k] + Wi[k][j] < Wi[i]
[j] || !Wi[i][j])) {
                    x = k;
                    Wi[i][j] = Wi[i][k] + Wi[k][j];
                    Wi[j][i] = Wi[i][k] + Wi[k][j];
                }
            if (x > -1) {
                path[i][j][i] = path[i][j][j] = path[i][j][x] = 1;
                path[j][i][i] = path[j][i][j] = path[j][i][x] = 1;
            }
        }
}
```

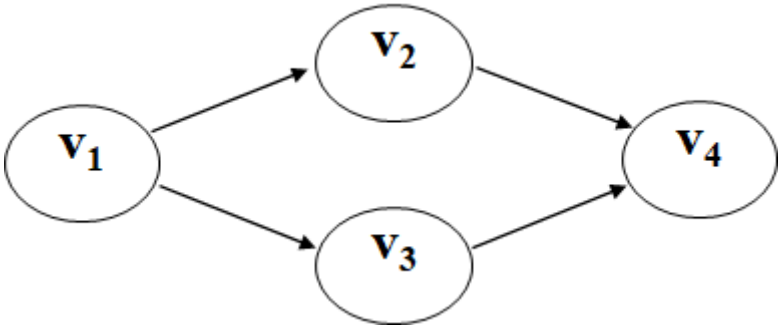
小结

单源最短路径：从指定顶点v, 要求在图G中找到从顶点v到其它各顶点的最短路径——Dijkstra算法 时间复杂度为O(n2)。

求图G中任意一对顶点之间的最短路径——Floyd算法时间 复杂度为O(n3)。

4.6有向无环图

有向无环图：一个没有回路 (即无环)的有向图。主要用于研究工程项目的工序问题、工程时间进度问题等。



一个工程(project)都可分为若干个称为活动(active)的子工程(或工序)，各个子工程受到一定的条件约束：某个子工程必须开始于另一个子工程完成之后；整个工程有一个开始点(起点)和一个终点。人们关心：

- **工程能否顺利完成？（拓扑排序问题）**
- **估算整个工程完成所必须的最短时间是多少?影响工程的关键活动是什么？（关键路径问题）**

AOV网：图中顶点表示活动，有向边表示活动之间的优先关系，这样的有向图称为顶点表示活动的网(Activity On Vertex Network ， AOV网) 。

检查有向图中是否存在环的方法:

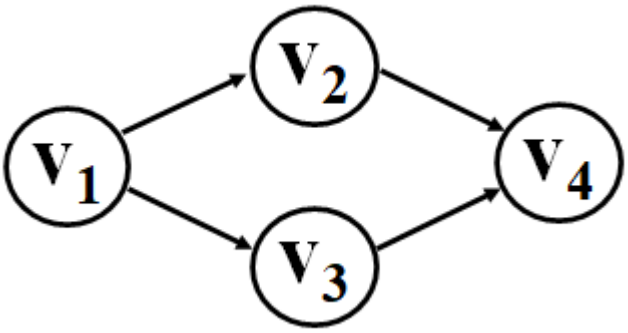
1. **如果从有向图G的某个顶点v出发，进行深度优先搜索遍历时产生顶点u到顶点v的回边，则图G存在包含顶点u和顶点v的环。**
2. **对有向图G进行拓扑排序。如果所有顶点都包含在"拓扑序列"中，则图G不存在环。**

4.7拓扑排序

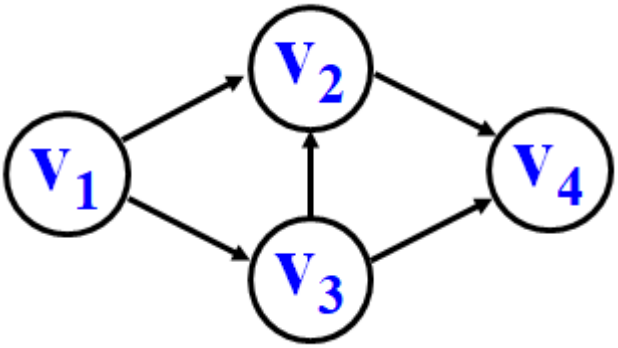
拓扑序列：假设对有向图进行操作：按照有向图的弧 (次序关系)，将图中顶点排成一个线性序列(对于图中没有限定次序关系的顶点, 可以人为地添加上任意的次序关系)，由此得到的顶点序列称为拓扑序列。

例:

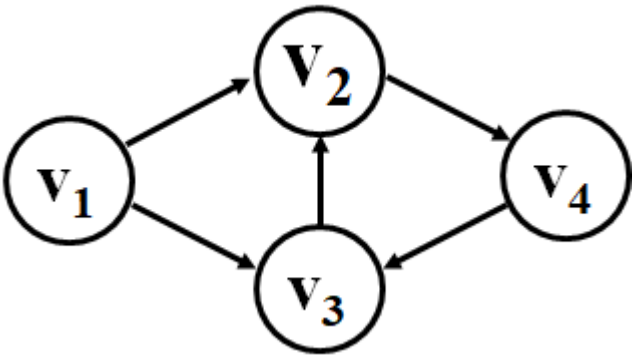
- **在由顶点v1，v2，v3和v4组成的有向图中，可以得到两个拓扑序列：v1，v2,v3,v4和 v1,v3,v2,v4。**



- **而在下面的有向图中，拓扑序列唯一：v1,v3,v2,v4**



- **在下面的有向图中，不能求得拓扑序列——因为图中存在一个环 {v2,v4,v3}**



拓扑排序的一般操作方法：

1. **从有向图中选取一个入度为0的顶点，输出它;**
2. **从有向图中删去该顶点以及所有以它为尾的弧;**
3. **重复上述两步，直至图空或者找不到入度为0的顶点为止。例如，**

拓扑排序算法的基本操作步骤：

- **求各顶点的入度, 将入度=0的顶点入队;**
- **当队列非空时，进行下列操作：**
    1. **输出队首元素v;**
    2. **将顶点v的所有邻接点的入度减1。如果出现入度=0的顶点，将该顶点入队。**

算法实现：

1. **邻接矩阵**

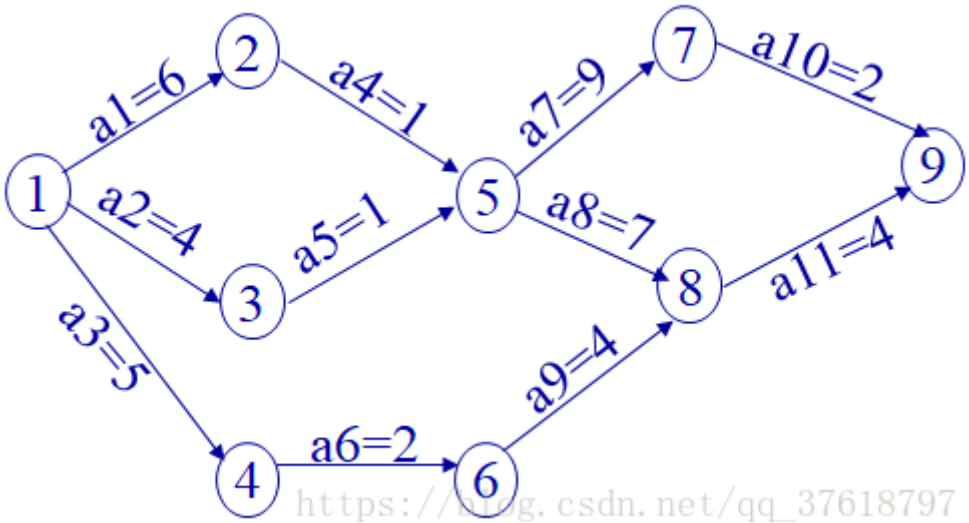
```
template<class T>
void MatrixGraph<T>::TopoSort() {
    int n = 0, r = 0, i, j, size = this->vn, p;
    int topolist[MAXSIZE], ID[MAXSIZE] = { 0 };
    for (i = 0; i < size; i++) {
        ID[i] = 0;
        for (j = 0; j < size; j++)
            if (this->Vr[j][i])
                ID[i]++;
        if (ID[i] == 0)
            topolist[r++]=i;
    }
    while (r > n) {
        p = topolist[n++];
        for (i = 0; i < size; i++)
            if (this->Vr[p][i]) {
                ID[i]--;
                if (ID[i] == 0)
                    topolist[r++] = i;
            }
    }
}
```

2. **邻接表**

```
template<class T>
void ListGraph<T>::TopoSort() {
    int n = 0, r = 0, i, j, size = this->vn;
    int topolist[MAXSIZE], ID[MAXSIZE] = { 0 };
    ENode<T>* p;
    for (i = 0; i < size; i++) {
        p = this->Ve[i].head;
        while (p) {
            ID[p->Vi]++;
            p = p->next;
        }
    }
    for (i = 0; i < size; i++)
        if (ID[i] == 0)
            topolist[r++] = i;
    while (r > n) {
        p = this->Ve[topolist[n++]].head;
        while (p) {
            ID[p->Vi]--;
            if (ID[p->Vi] == 0)
                topolist[r++] = p->Vi;
            p = p->next;
        }
    }
}
```

4.8关键路径

AOE网（Activity on Edge）：AOE网是一个带权有向无环图。其中，顶点表示事件，用弧表示活动，权表示活动持续的时间。例如，



问题描述：

假设以AOE网表示1个施工图,权值表示完成该项子工程（或称活动）所需时间

1. **完成整个工程需要多少时间?**
2. **哪些活动是影响工程进度的关键?**

源点：入度=0的顶点(工程的开始点)。

汇点：出度=0的顶点(工程的结束点)。

关键路径(工程完成最短时间)：从起点到终点的最长路径长度(路径上各活动持续时间之和) 。长度最长的路径称为关键路径

关键活动: 关键路径上的活动称为关键活动。关键活动是影响整个工程的关键。

事件vi的最早发生时间: 设v0是起点，从v0到vi的最长路径长度称为事件vi的最早发生时间，即是以vi为尾的所有活动的最早发生时间。

若活动ai是弧<j,k>，持续时间是dut(<j,k>)，设：

- **e(i)：表示活动ai的最早开始时间；**
- **l(i)：在不影响进度的前提下，表示活动ai的最晚开始时间； 则l(i)-e(i)表示活动ai的时间余量，若l(i)-e(i)=0，表示活动ai是关键活动。**
- **ve(i)：表示事件vi的最早发生时间，即从起点到顶点vi的最长路径长度；**
- **vl(i)：表示事件vi的最晚发生时间。**

则有以下关系：

$$\begin{cases} e(i)=ve(j) \\ l(i)= vl(k)-dut(<j,\ k>) \end{cases}$$

$$ve(j)=\begin{cases}0 & j=0，表示v_j是起点\\ Max\{ve(i)+dut(<i,\ j>)|<v_i,\ v_j>是网中的弧\}\end{cases}$$

含义是：源点事件的最早发生时间设为0；除源点外，只有进入顶点vj的所有弧所代表的活动全部结束后，事件vj才能发生。即只有vj的所有前驱事件vi的最早发生时间ve(i)计算出来后，才能计算ve(j) 。

方法是：对所有事件进行拓扑排序，然后依次按拓扑顺序计算每个事件的最早发生时间。

$$vl(j)=\begin{cases}ve(n-1) & j=n-1，表示v_j是终点\\ Min\{vl(k)-dut(<j,\ k>)|<v_j,\ v_k>是网中的弧\}\end{cases}$$

含义是： 只有vj的所有后继事件vk的最晚发生时间vl(k)计算出来后，才能计算vl(j)

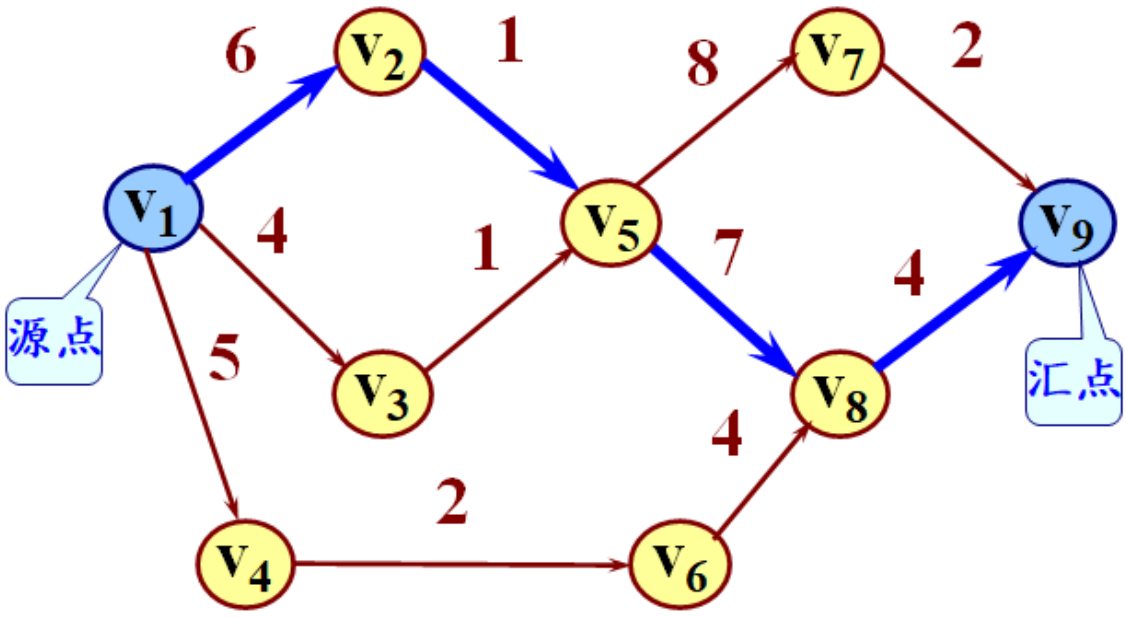
方法是：按拓扑排序的逆顺序，依次计算每个事件的最晚发生时间。

求AOE中关键路径和关键活动算法思想

1. **利用拓扑排序求出AOE网的一个拓扑序列；**
2. **从拓扑排序的序列的第一个顶点(源点)开始，按拓扑顺序依次计算每个事件的最早发生时间ve(i)；**
3. **从拓扑排序的序列的最后一个顶点(汇点)开始，按逆拓扑顺序依次计算每个事件的最晚发生时间vl(i)；**

关键路径：从源点到汇点的最长路径。

例如，v1→v2→v5→v8→v9是关键路径。



代码实现：

```
template<class T>
void MatrixGraph<T>::CriticalPath(){
    int i,j,v,size=this->vn;
    int ve[MAXSIZE] = { 0 }, vl[MAXSIZE];
    for (i = 0; i < size; i++) {
```

```
        vl[i] = -1;
        v = this->topolist[i];
        for (j = 0; j < size; j++)
            if(this->vr[v][j])
                if (this->vr[v][j] + ve[v] > ve[j] || ve[j] == 0)
                    ve[j] = this->vr[v][j] + ve[v];
    }
    vl[size - 1] = ve[size - 1];
    for (i = size-1; i >=0; i--) {
        v = this->topolist[i];
        for (j = 0; j < size; j++)
            if (this->vr[j][v])
                if (vl[v]- this->vr[j][v] < vl[j] || vl[j] == -1)
                    vl[j] = vl[v]- this->vr[j][v];
    }
    for (i = 0; i < size; i++)
        if (vl[this->topolist[i]] == ve[this->topolist[i]])
            cout << this->topolist[i] << " ";
}
```

5查找 Search

5.1查找的基本概念

查找：在数据表中查找满足给定值的第一个记录或全部记录。

查找成功：在数据表中查找到满足条件的记录——返回查找结果所在记录的指针。

查找不成功：在数据表中没有查找到满足条件的记录——返回空指针。

静态查找：没有改变数据表的查找。

动态查找：可能需要在数据表中进行插入操作或删除操作的查找。

关键字：记录中某个成员(数据项)的值，用以识别一个记录。

- **若关键字可以唯一识别一个记录，则称之为主关键字(如学号);**
- **若关键字可能识别若干个记录，则称之为次关键字(如姓名、成绩)。**

平均查找长度(Average Search Length)

ASL= $\sum_{i=1}^{n} P_i C_i$

其中, n为数据表长度, Pi为查找数据表中第i个记录的概率， $\sum_{i=1}^{n} P_i = 1$ **，Ci为找到第i个记录时已经进行比较的次数。**

5.2静态查找

顺序查找

顺序查找法：依据顺序查找的方法实现对数据表的查找算法。

顺序查找过程：从数据表中的最后一个记录(或第一个记录)开始，逐个比较关键字和给定值，并判断查找是否成功。

例：在数据表L[1..n]中，顺序查找关键字=key的记录。若找到，返回该记录在数据表中的位置，否则返回0。

算法分析：从n到1，逐个比较关键字=key，并根据比较情况输出查找结果。

代码实现：

```
int Search(Type L[],Type key,int n)
{
    L[0]=key;  // 哨兵
    for(i=n; L[i]!=key; --i);
    return i;
} //算法的时间复杂度为O(n)
```

顺序查找的时间性能分析:

对数据表而言，Ci = i，即

$$ASL=P_n+2P_{n-1}+\ldots+(n-1)P_2+nP_1$$

在等概率查找的情况下，

$$P_i=\frac{1}{n}$$

即顺序表查找的平均查找长度为:

$$ASL=\frac{1}{n}\sum_{i=0}^{n} i=\frac{n+1}{2}$$

折半查找

有序表：在数据表中, 记录按关键字值从小到大递增(或从大到小递减)有序。

折半查找法：用折半查找的方法实现对有序表的查找算法=>折半查找法也称为二分查找法。

```
例： 折半查找算法。
int BiSearch(Type L[],Type key,int n)
{   left=1,right=n; //置区间初值
    while(left<=right)
    {   mid=(left+right)/2;
        if(key==L[mid]) return mid;
        if(key<L[mid]) right=mid-1;
        else left=mid+1;
    } return -1;    //查找不成功
} //折半查找算法的时间复杂度为O(logn)
```

分块查找

分块查找算法

将数据按照一定原则存入数据表，使数据表划分为若干个逻辑子表。

逻辑子表由索引表划分。

索引表：由每个逻辑子表的最大关键字 项构成的一个有序表。

例：数据表L={22, 12, 8, 18,|33, 42, 23, 35,|48, 55, 60, 86, 77}包含三个逻辑子表。

索引表LI={22, 1, 4; 42, 5, 4; 86, 9, 5};分别是子表中最大值，主表位置，子表位置

分块查找：先用顺序查找法在索引表中查找索引记录，再依据该记录值在数据表中的相应逻辑子表查找所需要的记录。

代码实现：

1. **索引表存储结构**

```
typedef struct
{   KeyType index;
    int start;
    int length;
}IndexList;
IndexList LI[M];
```

2. **分块查找**

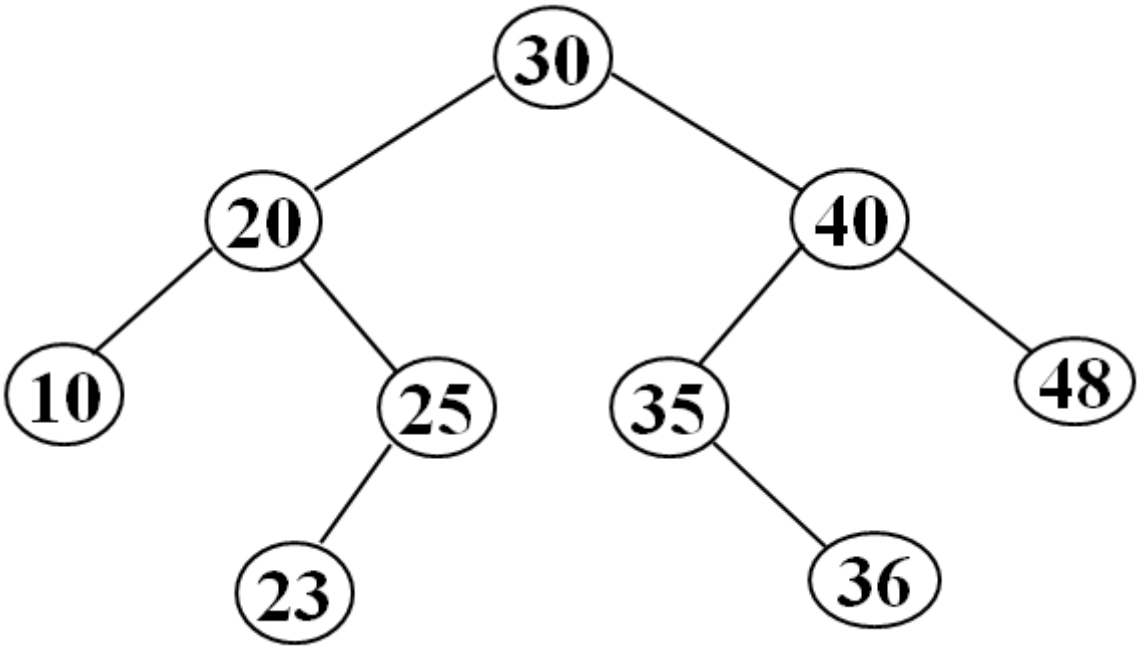
```
int BlockSearch(int m,int key)  //m为LI的长度
{   for(i=0;i<m;++i)
        if(key<=LI[i].index) break;
    if(i==m) return 0;  //没有找到相应逻辑子表
    j=LI[i].start;      //逻辑子表开始记录
    while(j<LI[i].start+LI[i].length)
        if(key==L[j].key) return j;     //查找成功
        else ++j;
    return 0;   //查找不成功
}
```

5.3动态查找

二叉排序树

定义：或是一棵空树，或是具有如下特性的二叉树：

1. **若左子树不空，则左子树上所有结点的值均小于根结点的值；**
2. **若右子树不空，则右子树上所有结点的值均大于(或等于)根结点的值；**
3. **左、右子树分别是二叉排序树。**



代码实现：

1. **采用二叉链表存储结构**

```
template<class T>
class BinaryTree{
public:
    T key; //关键字
    //ElemType* otherinfo;
    struct Tnode* lc; //左指针
    struct Tnode* rc; //右指针
};
```

2. **查找算法**

```
template<typename T>
int BinaryTree<T>::SearchT(T e)
{
    if (!this) return 0;  //查找不成功
    if (e == this->key) return 1;  //查找成功
    if (e < this->key)  //继续查找
        return SearchT(this->lc, key);
    return SearchT(this->rc, key);
}
```

3. **查找算法**

```
template<class T>
BinaryTree<T>* BinaryTree<T>::SearchT_(T e){
    if (!this) return 0;  //查找不成功
    if (e == this->key) return this;  //查找成功
    if (e < this->key)
        return this->lc->SearchT_(e);
    return this->rc->SearchT_(e);
}
```

4. **插入算法**

   如果查找成功，返回0(不进行插入操作);否则，插入值为key的结点并返回1。

```
template<class T>
int BinaryTree<T>::insert(T e) {
    BinaryTree<T>* p=this;
    if (this->SearchT(e, p))return 0;
    else {
        BinaryTree<T>* t = new BinaryTree<T>(e);
        t->lc = t->rc = NULL;
        if (e < p->key)p->lc = t;
        else p->rc = t;
        return 1;
    }
}
```

5. **删除结点**

   删除操作在查找成功之后进行，并且要求在删除操作后，仍然保持二叉排序树的特性。被删除的结点只能是下列三种情况之一:

   1. **叶子结点;**
   2. **只有左子树或只有右子树;**
   3. **既有左子树，也有右子树。**

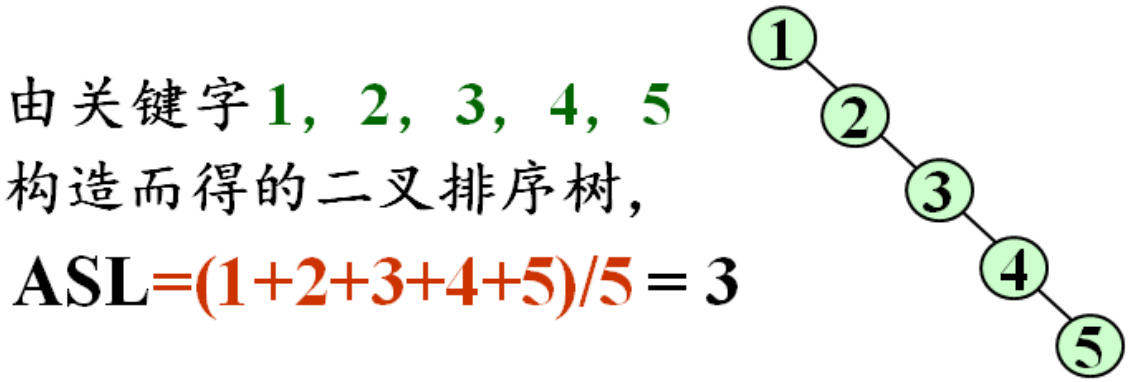
```
template<class T>
int BinaryTree<T>::deleteT(T e) {
    BinaryTree<T>* p = this;
    BinaryTree<T>* q = this->SearchT(e, p);
    if (!q)return 0;
    if (!q->lc && !q->rc) delete q;//叶子结点
    else if (q->lc && q->rc) {//左右子树都有
        BinaryTree<T>* next;
        p = q; next = q->lc;
        while (next->rc) {
            q = next;
            next = next->rc;
        }
        p->key = next->key;
        if (p == q)
            p->lc = next->lc;
        else q->rc = next->lc;
        delete next;
    }
```

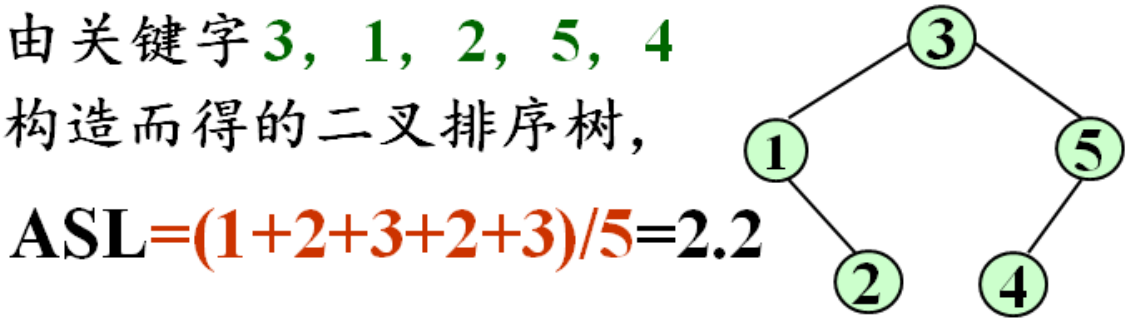
```
    else if (q->lc) {//只有左子树
        if (p->lc == q)
            p->lc = q->lc;
        else if (p->rc==q)
            p->rc = q->lc;
        delete q;
    }
    else if (q->rc) {//只有右子树
        if (p->lc == q)
            p->lc = q->rc;
        else if (p->rc == q)
            p->rc = q->rc;
        delete q;
    }
    return 1;
}
```

查找性能分析

对于每一棵二叉排序树，均可按照平均查找长度的定义来计算它的ASL值。

由n个关键字值构造所得的不同形态的二叉排序树，其平均查找长度不一定相同，甚至可能差别很大。





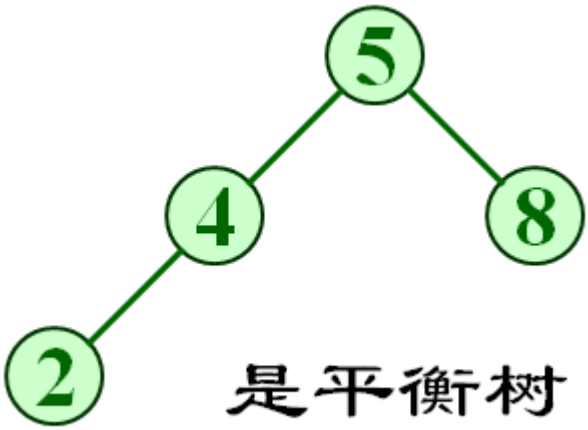
平衡二叉树

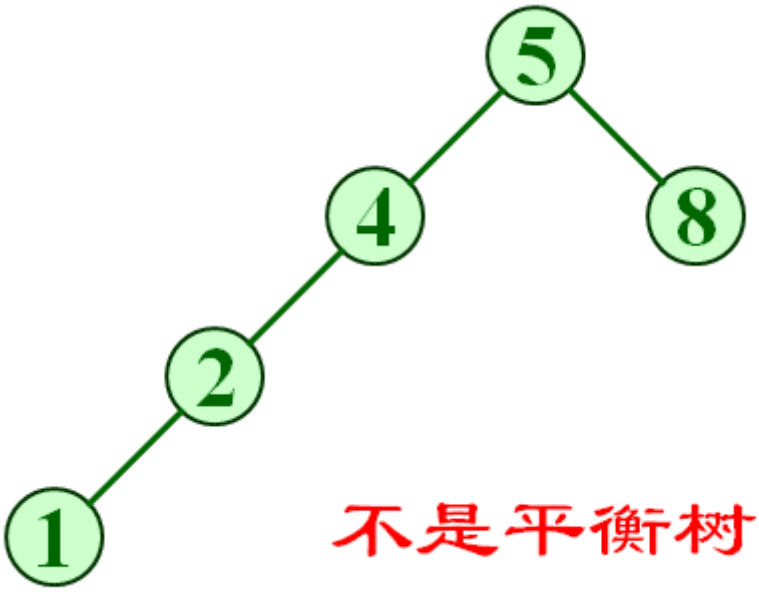
平衡二叉树定义：

或者是一棵空树，或者是具有下列性质的二叉树：

1. **左子树和右子树都是平衡二叉树；**
2. **左子树和右子树的深度之差≤1。**

Balanced Binary Tree，又称AVL树（Adelson-Velskii and Landis）





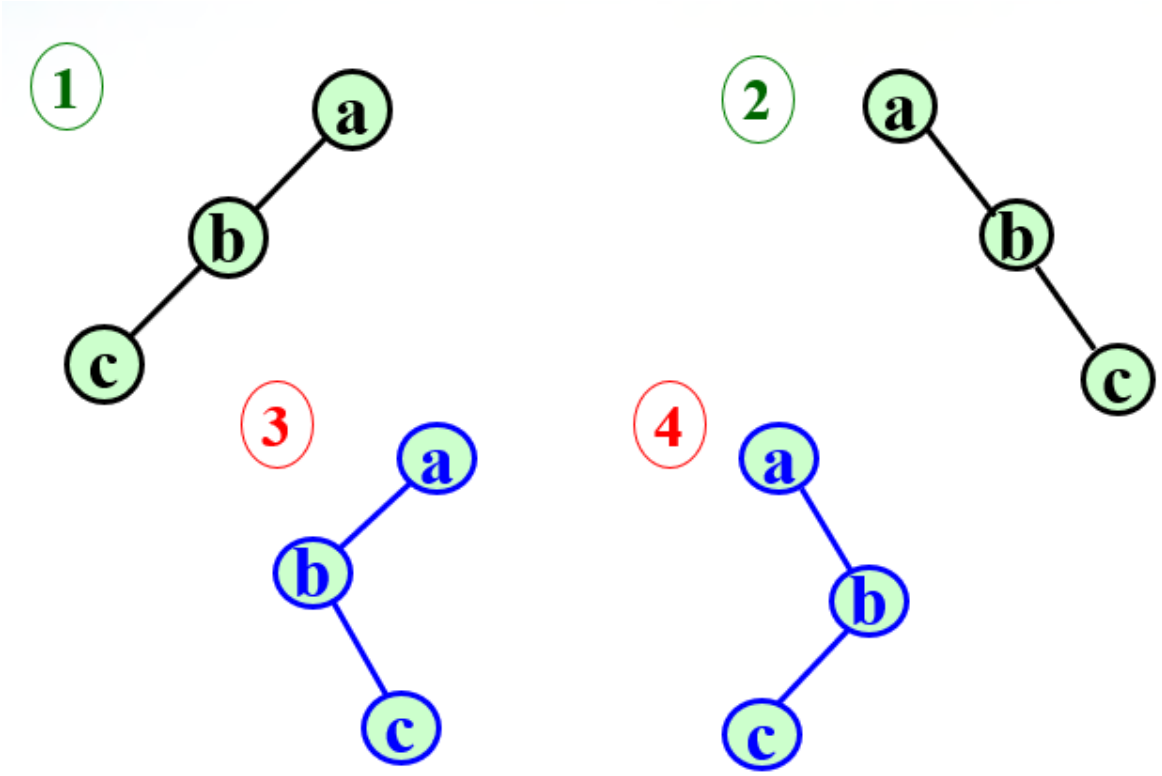
平衡因子(Balance Factor)=左子树的深度 - 右子树的深度

- **平衡二叉树上：平衡因子= -1、0或1。**
- **如果二叉树上所有结点的平衡因子的绝对值都≤1，则它是一棵平衡二叉树**
- **如果二叉树上有一个结点的平衡因子的绝对值>1，则它不是平衡二叉树。**

构造平衡二叉树：

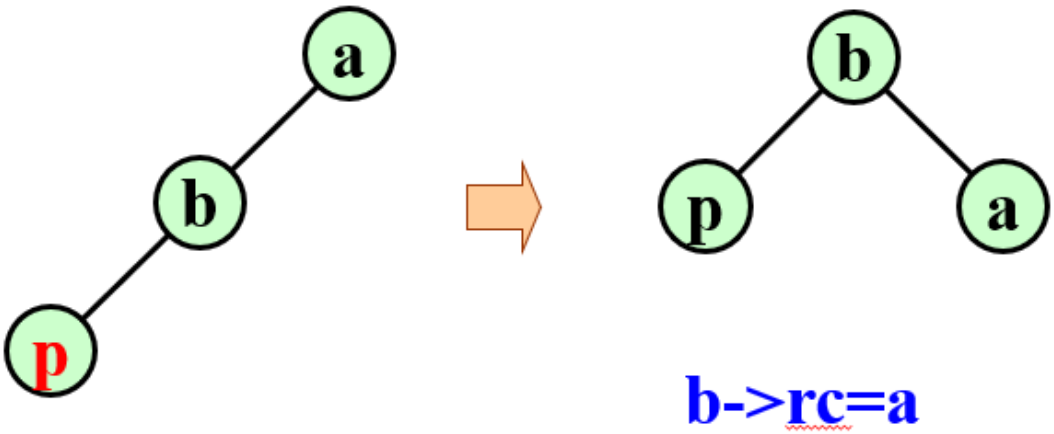
在构造二叉排序树的过程中，如果出现不平衡结点，需采用平衡方法将二叉树调整成平衡二叉树，其要点是：将关键字中间值的结点作为根结点，再重新连接相关结点，构成平衡二叉树。

不平衡二叉树的基本形态：

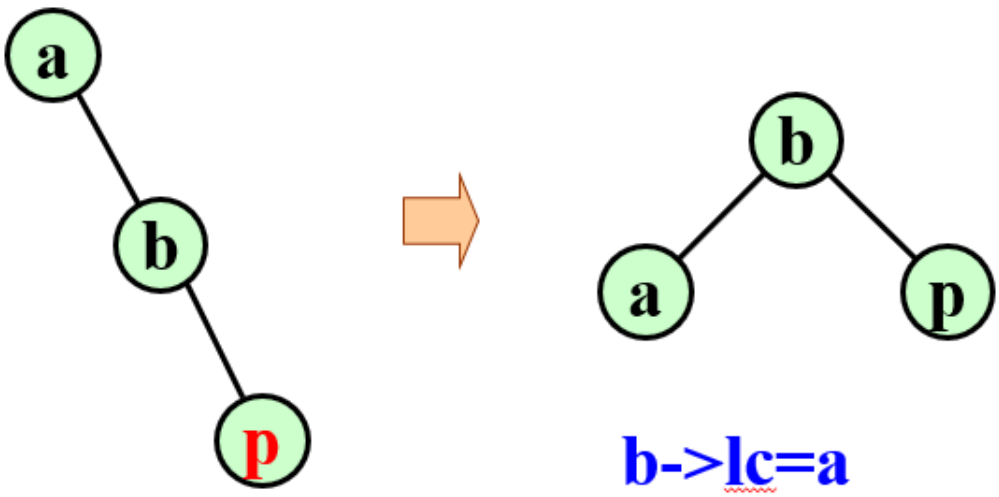


二叉排序树的基本平衡方法：

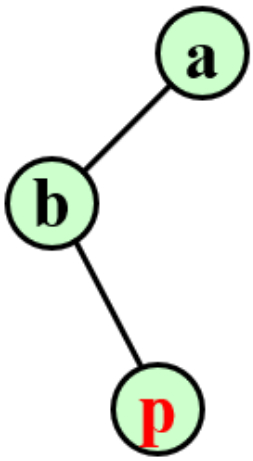
1.

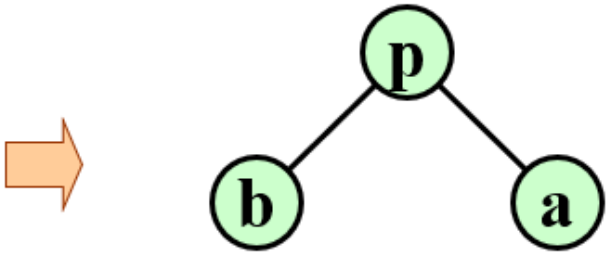


2.



3.



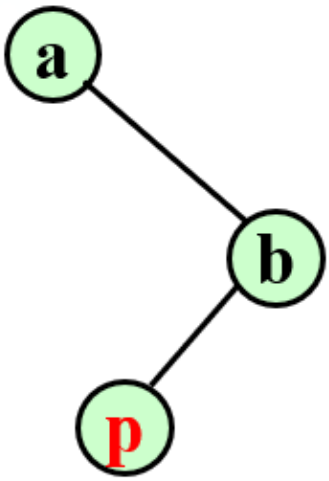


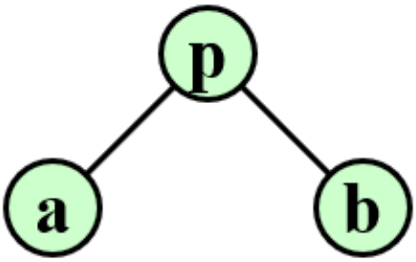
[b<p<a]

p->lc=b

p->rc=a

4.



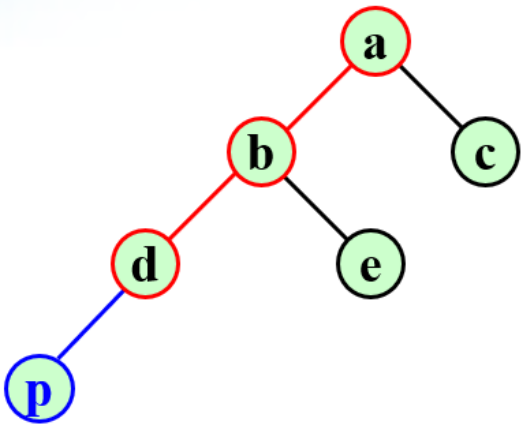


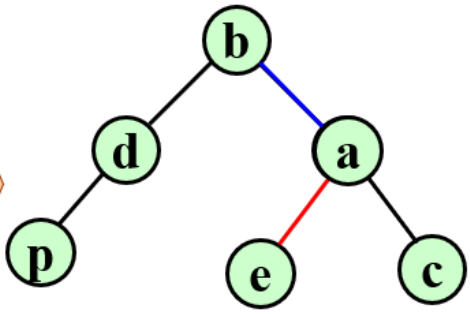
[a<p<b]

p->rc=b

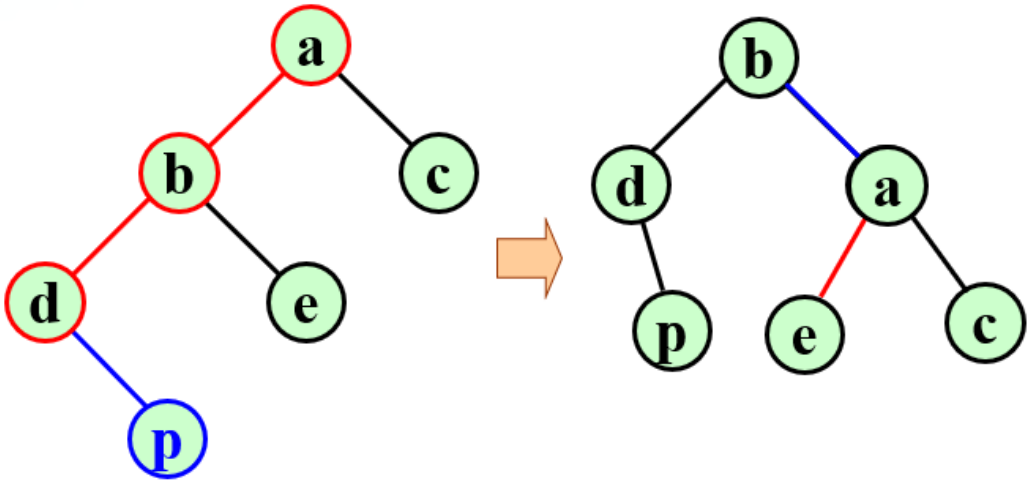
p->lc=a

5.



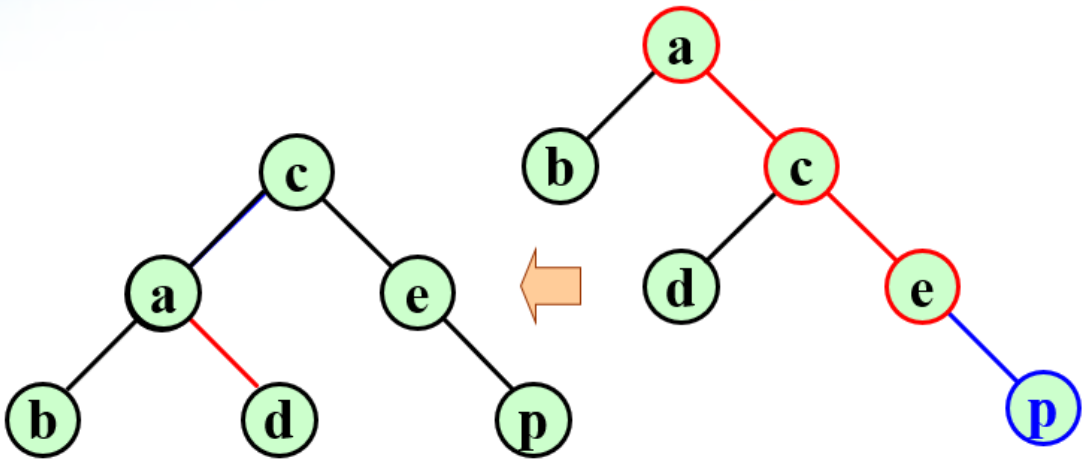


q=b; a->lc=e; q->rc=a;

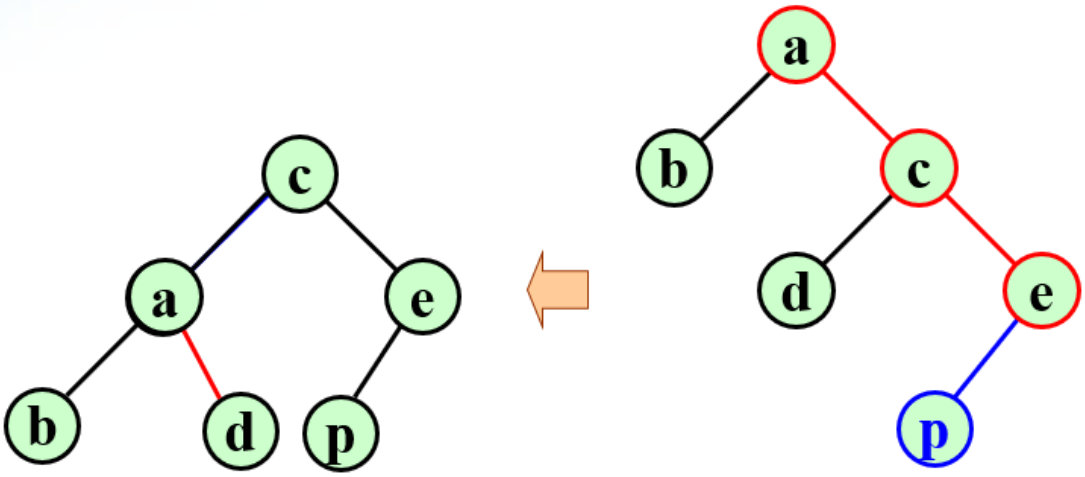


q=b; a->lc=e; q->rc=a;

6.

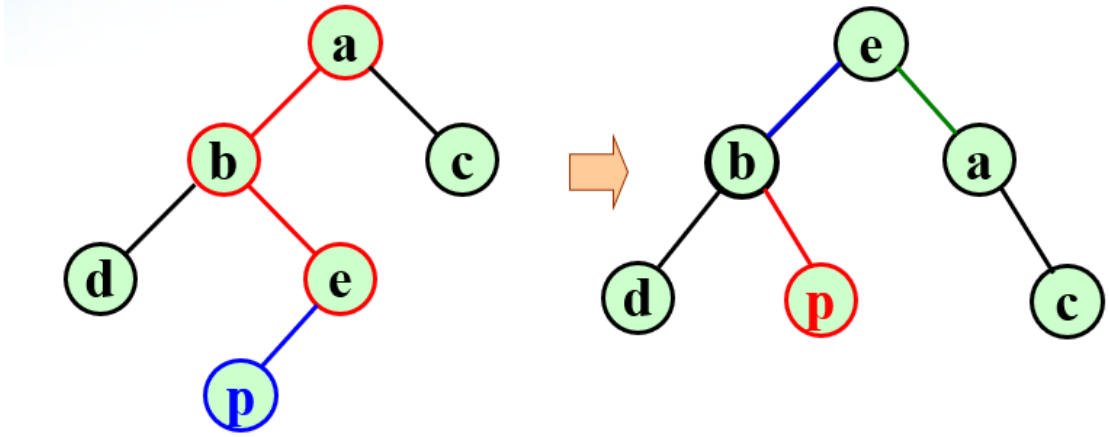


q=c; a->rc=d; q->lc=a;

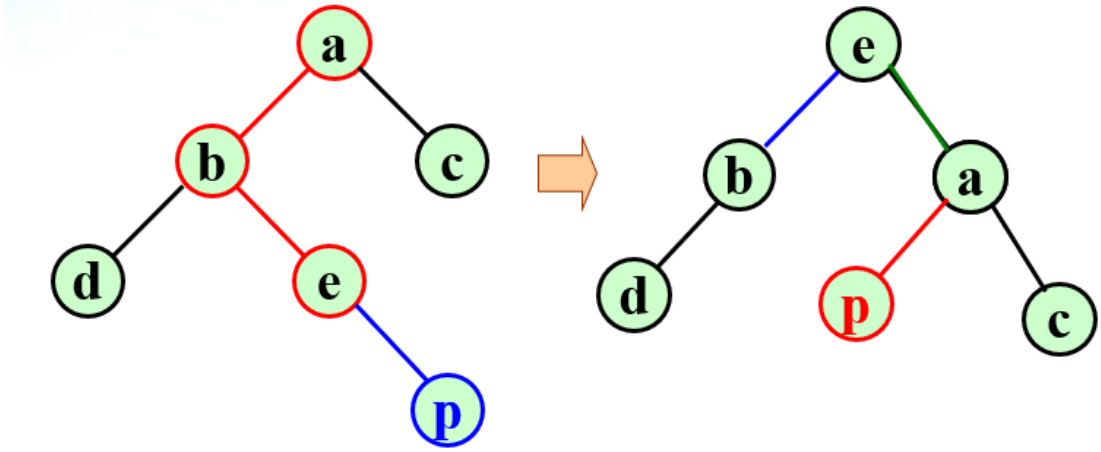


q=c; a->rc=d; q->lc=a;

7.

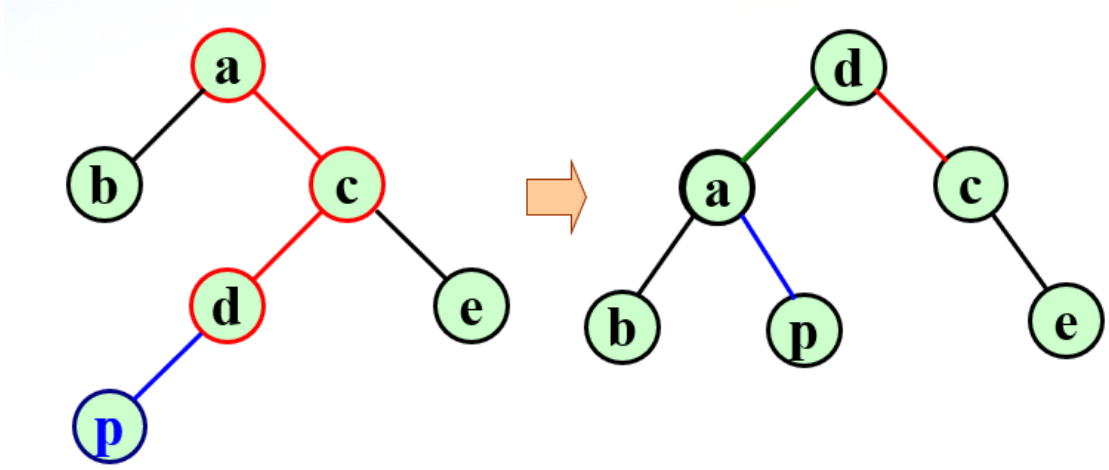


q=e; b->rc=p; q->lc=b; q->rc=a;

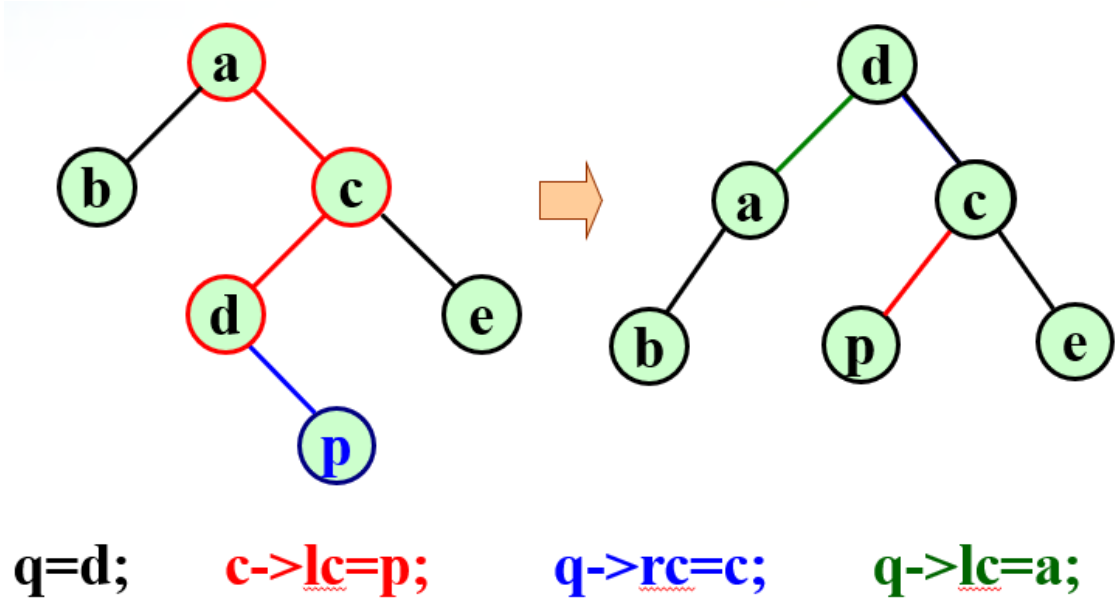


q=e; q->lc=b; a->rc=p; q->rc=a;

8.



q=d; q->rc=c; a->rc=p; q->lc=a;



q=d; c->lc=p; q->rc=c; q->lc=a;

平衡二叉树的查找性能分析：

在平衡树上进行查找的过程和二叉排序树相同，因此，查找过程中和给定值进行关键字的比较次数不超过平衡树的深度。

问：含有n个关键字的平衡二叉树可能达到的最大深度是多少?

答：[log2n]+1

在平衡二叉树上进行查找的时间复杂度为O(logn)

5.4散列表

基本概念

哈希表主要包括构造哈希函数、处理冲突和哈希表查找等内容。

基本思想：

- 以数据元素的关键字key作为自变量构造一个哈希函数Hash(key)，并将函数值定义为该数据元素的存储地址。
- 目的：为数据元素在列表中的位置和它的关键字值建立一个确定的关系。

哈希表(Hash List，也称为散列表)：基于哈希函数建立的顺序表。

哈希地址：哈希函数的值。

在哈希表中进行查找时，先计算给定值对应的哈希地址，再根据该地址从哈希表中取出相应的数据元素。

哈希表举例：设线性表A= { 18, 75, 43, 55, 60, 46 }，哈希函数 H(key)=key%m。如果取m=6 (最小存储空间)，则H(18)=0，H(75)=3，H(43)=1，H(55)=5，H(60)=0，H(46)=4；这时，H(60)=0和H(18)=0发生冲突，必须重新分配哈希地址(即需要处理冲突)。

哈希函数

哈希函数的构造方法

1. 直接定址法
2. 除留余数法

直接定址法

取关键字的线性函数作为哈希函数：H(key) = a × key + b

其中，a和b为常数。此方法适合于：地址集合的大小≥关键字集合的大小。

例如，关键字集={11, 22, 33, 44, 55}，m=5

(地址集合的大小=关键字集合的大小)， 存储地址依次是0, 1, 2, 3和4。

=> 取哈希函数H(key) = key/11-1，即取a=1/11，b= –1。

除留余数法

设哈希表表长为m，取哈希函数为H(key) = key % p，p≤m。

要点：选择一个合适的p值，使H(key)的同义词尽可能少(尽可能少冲突)。一般情况下，可以选择p为素数。

设关键字集={ 15, 45, 18, 39, 24, 33, 21 }，哈希表表长m=20。

则若取p=9(含质因子3)，则哈希地址为{ 6, 0, 0, 3, 6, 6, 3 }，冲突现象严重；

若取p=11(质数)，则哈希地址为{ 4, 1, 7, 6, 2, 0, 10 }，没有产生冲突现象。

处理冲突方法

处理冲突的含义：为产生冲突的哈希地址寻找下一个哈希地址。

1. **开放定址法**
2. **链地址法**

开放地址法

为产生冲突的哈希地址H(key)求得一个地址序列：H0，H1，…，Hs，1≤s≤m-1

其中，H0 = H(key)，Hi =(H0+di)%m，i=1，2，…，s (di称为增量)

1. **线性探测再散列：di = c × i，其中c为常数，i =1,…,m-1**
2. **二次探测再散列：di = 1^2，-1^2，2^2，-2^2，……**

注意：增量di应具有"完备性"：s个Hi均不相同，且都是哈希表的有效地址。

例：表长m=11，关键字集={ 19, 1, 23, 14, 55, 68, 11, 82, 36 }。

设定哈希函数H(key) = key % 11，采用线性探测再散列处理：di = i，则

关键字值	**19**	**1**	**23**	**14**	**55**	**68**	**11**	**82**	**36**
地址初值	8	1	1	3	0	2	0	5	3
哈希地址	8	1	2	3	0	4	5	6	7

ASL = (1 + 1 + 2 + 1 + 1 + 3 + 6 + 2 + 5)/n=22/9

链地址法

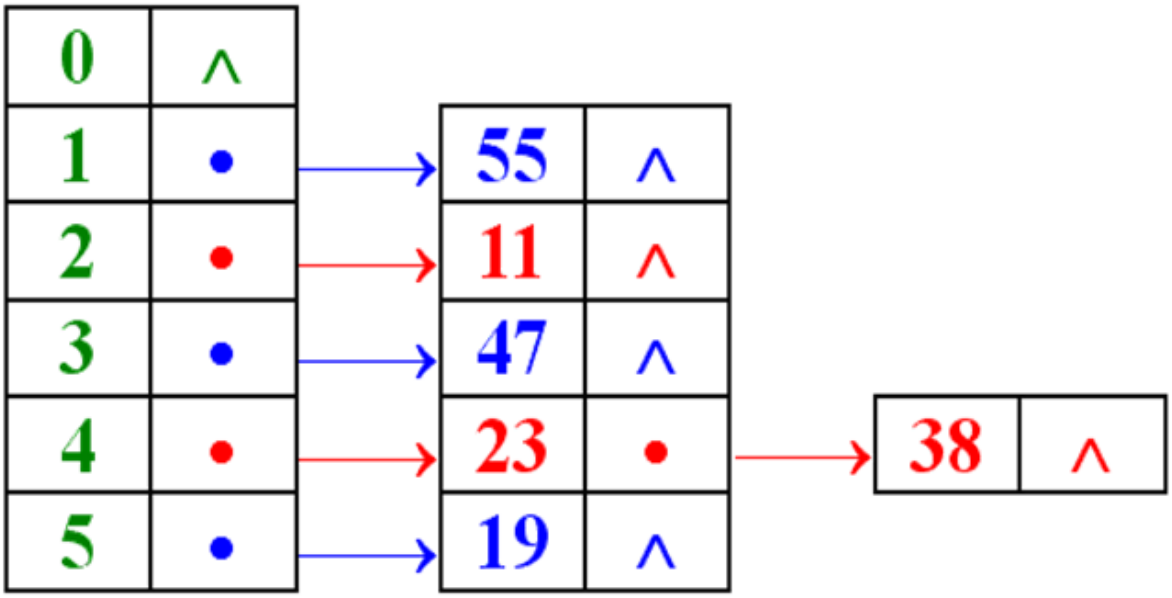
- **将所有哈希地址相同的数据元素都存放在同一个链表中；**
- **链地址法的平均查找长度较短；**
- **链地址法更适合于：构造表前无法确定表长的情况(动态申请链结点)；**
- **在用链地址法构造的哈希表中，删除结点的操作易于实现。**

```
template<class T>
struct LNode
{
    T key;//ElemType data
    LNode* next;
};
```

例题：关键字集={19, 23, 55, 11, 47, 38}，表长m=6。

取哈希函数H(key)=key%5+1，则哈希地址={5, 4, 1, 2, 3, 4}

由链地址法产生的哈希表



哈希表的查找过程

对于给定值k0，计算哈希地址Hi=H(k0)；

- **如果Hash[Hi]=NULL，则查找不成功；**
- **如果Hash[Hi].key=k0，则查找成功；**
- **否则，按原处理冲突方法求下一个哈希地址Hi，直至Hash[Hi].key=k0，或者Hash[Hi]=NULL为止。**

代码实现：

1. **在开放地址哈希表H中查找关键字为Key的记录**

```
int Hash[M]; //定义哈希表数组
int  SearchHash(int key, int &p)
{   int p=Hash(key); //计算哈希地址
    while(Hash[p])
    {   if(Hash[p]=key)  return 1;
        p=NextHash(key); //求下一个地址
    }
    return 0;
} //SearchHash 算法结束
```

2. **在链地址表中查找关键字为Key的记录。**

```
template<class T>
int SearchHashL(T key, LNode<T>* p){
    int Hi = Hash(key);  //计算哈希地址
    p = HashL[Hi].next;
    while (p)
        if (p.key = key) return 1;
        else p = p.next;  //下一个关键字
    return 0;
} //SearchHashL算法结束
```

哈希表的查找性能分析

从查找过程得知，哈希表的平均查找长度实际上并不等于1。

决定哈希表平均查找长度ASL的因素：处理冲突的方法和装填因子。

例如，对于表长m=6，关键字集={ 19，23， 55， 11， 47， 38 }，如果采用链地址法，ASL=(1×5+2×1)/6 >1

装填因子定义：

$$\alpha = \frac{\text{哈希表中实际填入的记录数}}{\text{哈希表的长度}}$$

一般情况下，a越小，发生冲突的可能性越小；a越大，填记录就越容易发生冲突。

哈希表的一个特点：

用哈希表构造列表时，可以选择一个适当的装填因子α，使得平均查找长度限定在某个范围内。一般情况下，哈希表的平均查找长度是α的函数。如利用线性探测再散列处理冲突法的平均查找长度

$$ASL=\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

例题：已知列表L中的1000个关键字各不相同。请给出列表L的一个哈希表设计方案，要求它在等概率情况下，查找成功时的平均查找长度不超过3。

1. 选择处理冲突的方法，并求出α——“线性探测再散列”处理冲突法。
2. 由于要求平均查找长度≤3，则 $ASL=\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)\leqslant 3$ 得到

   α≤0.8
3. 确定哈希表的长度m，一般取m≥k/α中的最小素数。

   ∵ k/a=1000/0.8=1250=> m=1259 (根据给定的素数表找出满足要求的最小质数)。

6排序 Sort

基本概念

排序定义：假设含n个元素的数据表为{r1, r2, ..., rn}，试确定1, 2, ..., n的一种排列p1, p2, ..., pn，使数据表满足非递减关系：$r_{p_1} \leq r_{p_2} \leq \ldots \leq r_{p_n}$

即数据表$\{r_{p_1}, r_{p_2}, \ldots, r_{p_n}\}$递增有序。

排序的稳定性：对于数据表{r1, r2, ..., rn}，当ri=rj且i<j时,如果排序前ri领先于rj，排序后ri仍然领先于rj，则称该排序方法是稳定的；否则称该排序方法是不稳定的。

排序的分类：

- 物理排序：根据关键字值在数据表中重新排列记录的存储位置。
- 索引排序：根据关键字值建立索引表，在输出数据表的记录时，由索引表描述输出顺序(索引表当作静态链表用)。
- 比较式排序：根据数据表中两个元素值的大小，确定这两个元素的存储位置。
  - 两种基本操作：
    1. 比较两个关键字的大小
    2. 交换两个记录的位置
  - 两类数据区：

- 计算式排序：根据元素值，计算它在数据表中的存储位置。

6.1基本排协

插入排序

插入排序：在有序区找第i个元素的插入位置

```
template<class T>
void insertSort(T list[], int n) {
    int i, j;
    for (i = 2; i <= n; i++) {
        list[0] = list[i];
        for (j = i - 1; list[j] > list[0]; j--)
            list[j + 1] = list[j];
        list[j + 1] = list[0];
    }
}
```

时间复杂度为O(n^2)

插入排序算法复杂度分析

最好情况(关键字在数据表中有序):

- **比较次数:** $\sum_{i=2}^{n} 1 = n - 1$
- **移动次数: 0**

最坏情况(关键字在数据表中逆序有序):

- **比较次数:** $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- **移动次数:** $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$

希尔排序

希尔排序: 分逻辑区的插入排序, 又被称为缩小增量排序。

Shell基本思想:

- **将整个数据表分成m个子序列, 对每个子序列分别进行插入排序。**
- **逐步减少子序列数m, 直至m=1为止。**

举例: 将n个元素分成 m个子序列:

- **{L[1], L[1+m], L[1+2m], ..., L[1+km]};**
- **{L[2], L[2+m], L[2+2m], ..., L[2+km]};**
- **.....**
- **{L[m], L[m+m], L[3m], ..., L[(k+1)m]}**

其中, 增量m的值在排序过程中逐渐缩小, 直至最后一趟排序减至1。

```
template<class T>
void ShellInsert(T L[], int n, int m)
```

```
{
    int i, j, k;
    for (i = 1; i <= m; ++i)
    {
        for (j = i + m; j <= n; j += m)
        {
            L[0] = L[j];
            for (k = j - m; k > 0 && L[k] > L[0];k -= m)
                L[k + m] = L[k];  //记录后移
            L[k + m] = L[0];  //插入到相应位置
        }
    }
} //算法时间复杂度为O(n^2/m)

template<class T>
void ShellSort(T L[], int n, int delta[], int t) {
    //t为delta长度
    for (int m = 0; m < t; m++)
        ShellInsert<T>(L, n, delta[m]);
}
```

选择排序

选择排序：在无序区找第i小值的元素

```
template<class T>
void SelectSort(T L[], int n) {
    int i, j, min;
    for (i = 1; i <= n; i++) {
        min = i;
        for (j = i + 1; j <= n; j++)
            if (L[j] < L[min])
                min = j;
        j = L[i]; L[i] = L[min]; L[min] = j;
    }
}//算法时间复杂度为O(n^2)
```

冒泡排序

冒泡排序：将无序区的最大值元素交换到最后

```
template<class T>
void BubbleSort(T L[],int n) {
    int i, j;
    T t;
    for (i = n; i > 1; i--) {
        int count = 0;
        for (j = 1; j < i; j++)
            if (L[j] > L[j + 1]) {
                t = L[j]; L[j] = L[j + 1];
                L[j + 1] = t; count = j;
            }
        if (count == 0)
            break;
    }
}
```

冒泡排序算法时间复杂度分析

最好情况(关键字在数据表中递增有序)：只需进行一趟冒泡。

- **比较的次数：n-1**
- **移动的次数：0**

最坏情况(关键字在数据表中递减有序)：需进行n-1趟冒泡。

- **比较的次数：** $\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2}$
- **移动的次数：** $3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$

地精排序

地精排序：单重循环的排序算法

例 问题：数组L[n]存放n个整数，用一重循环将L[n]中的整数从小到大排序。

基本思路：前进冒泡，遇到冒泡时往回冒泡，直到把这个数字放好为止。

```
template<class T>
void GnomeSort(T L[], int n)
{
    int i = 0;
    T temp;
    while (i < n)
    {
        if (i == 0 || L[i - 1] <= L[i]) i++;
        else
        {
            temp = L[i];
            L[i] = L[i - 1];
            L[i - 1] = temp;
            i--;
        }
    }
} //算法的时间复杂度为O(n^2)
```

小结

插入排序，O(n2)

希尔排序，时间复杂度优于插入排序

选择排序，O(n2)

冒泡排序，O(n2)，实际效率可能较高

地精排序，O(n2)，单重循环的排序算法

6.2归并排序

举例：采用归并排序算法，将数据表L中的记录按关键字值从小到大排序。

基本思想：将两个(2路)或两个以上的有序(子)序列合并为一个有序序列。

1. **合并操作代码**

```
template<class T>
void Merge(T L[], int p, int q, int r) {
    int i = p, j = q + 1,k=0;
    T* temp = new T[r-p+1];
    while (i <= q && j <= r)
        if (L[i] <= L[j])
            temp[k++] = L[i++];
        else temp[k++] = L[j++];
    while (i <= q)
        temp[k++] = L[i++];
    while (j <= r)
        temp[k++] = L[j++];
    for (i = p; i <= r; i++)
        L[i] = temp[i - p];
    delete []temp;
}
```

2. **排序**

```
template<class T>
void MergeSort(T L[],int s,int t,int n) {
    if (s >= t)return;
    int m = (s + t) / 2;
    MergeSort(L, s, m,n+1);
    MergeSort(L, m + 1, t,n+1);
    Merge(L, s, m, t);
}//时间复杂度为O(nlog2n)
```

6.3快速排序

基本思想

- **通过一趟快速排序将待排数据分割成独立的左中右三个部分，其中，中部分只含一个数，左边部分的所有关键字都比这个数小，右边部分的所有关键字都比这个数大。**
- **然后再按此方法对这两部分记录分别进行快速排序，直到所有记录整理成有序序列。**

选取一个数据L[i]，以它的关键字值key作为基准元素(枢轴)进行划分：将顺序表L[s..t]划分成三个部分L[s..i-1]，L[i]和L[i+1..t]：

$$L[j].key \le L[i].key \le L[k].key$$

$$(s \le j \le i-1) \qquad (i+1 \le k \le t)$$

1. **找基准元素、**

```
template<class T>
int QuickPass(T L[], int l, int r) {
    T x = L[l];
    while (l < r) {
        while (l<r && L[r]>x)r--;
        L[l] = L[r];
        while (l < r && L[l] < x)l++;
        L[r] = L[l];
    }
    L[l] = x;
    return l;
}
```

2. **快速排序**

```
template<class T>
void QuickSort(T L[], int l, int r) {
    if (l > r)return;
    int m = QuickPass(L, l, r);
    QuickSort(L, l, m - 1);
    QuickSort(L, m + 1, r);
}
```

3. **合并写法**

```
template<class T>
void QuickSort(T L[], int l, int r) {
    if (l > r)return;
    T x = L[l];
    while (l < r) {
        while (l<r && L[r]>x)r--;
        L[l] = L[r];
        while (l < r && L[l] < x)l++;
        L[r] = L[l];
    }
    L[l] = x;
    int m = QuickPass(L, l, r);
    QuickSort(L, l, m - 1);
    QuickSort(L, m + 1, r);
}
```

快速排序算法时间复杂度分析：

假设一次划分所得枢轴位置i=k，则对n个记录进行快速排序所需时间为

T(n) = Tpass(n) + T(k-1) + T(n-k)

若待排序列中的关键字随机分布，则k 取1至n中任意一个值是等概率的。

由此可得快速排序所需时间的平均值为：

$$T(n)=\mathrm{cn}+\frac{1}{n}\sum_{k=1}^{n}(T(k-1)+T(n-k))$$

$$T(n)=\mathrm{cn}+\frac{2}{n}\sum_{k=0}^{n-1}T(k)\qquad T(n-1)=\mathrm{c}(\mathrm{n}-1)+\frac{2}{n-1}\sum_{k=0}^{n-2}T(k)$$

$$T(n)=\frac{2n-1}{n}\mathrm{c}+\frac{n+1}{n}\mathrm{T}(\mathrm{n}-1)$$

$$T(n)<\frac{n+1}{n}\mathrm{T}(1)+2(\mathrm{n}+1)\left(\frac{1}{2}+\cdots+\frac{1}{n+1}\right)\mathrm{c}$$

$$T(n)\leq\frac{n+1}{n}\mathrm{b}+2\mathrm{c}(\mathrm{n}+1)\ln(\mathrm{n}+1)\leq 2\mathrm{d}(\mathrm{n}+1)\ln(\mathrm{n}+1)$$

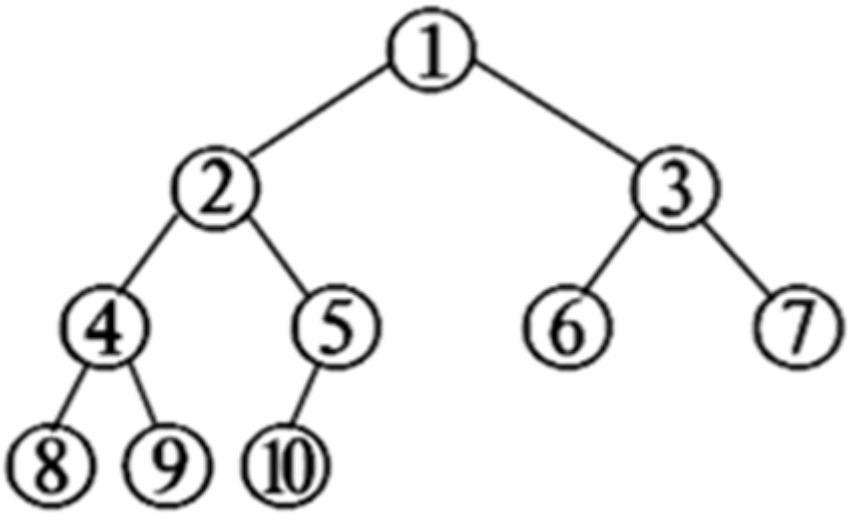
若待排记录的初始状态按关键字有序，快速排序将蜕化为冒泡排序，其时间复杂度为O(n^2)。

为避免出现这种情况，需在进行一次划分之前，进行“预处理”。例如，先对L(s).key, L(t).key和L[(s+t)/2].key进行相互比较，然后取关键字为“三者之中”的记录作为枢轴记录。

6.4堆排序

首先，“堆”是一棵完全二叉树。

r1	r2	r3	r4	r5	r6	r7	r8	r9	r10
1	2	3	4	5	6	7	8	9	10



- **结点i的孩子编号为2i和2i+1。**
- **结点i的双亲编号为i/2。**
- **最大分支编号为n/2。**
- **树高为[log2n] +1。**

其次，堆是满足下列性质的序列{r1, r2, ..., rn}:

$$\begin{cases} r_i \le r_{2i} \\ r_i \le r_{2i+1} \end{cases} \text{（小顶堆）} \qquad \begin{cases} r_i \ge r_{2i} \\ r_i \ge r_{2i+1} \end{cases} \text{（大顶堆）}$$

$$(\ i=1,\ 2,\ \ldots,\ n/2\ )$$

小顶堆: {12, 36, 27, 65, 40, 34, 98, 81, 73, 55}

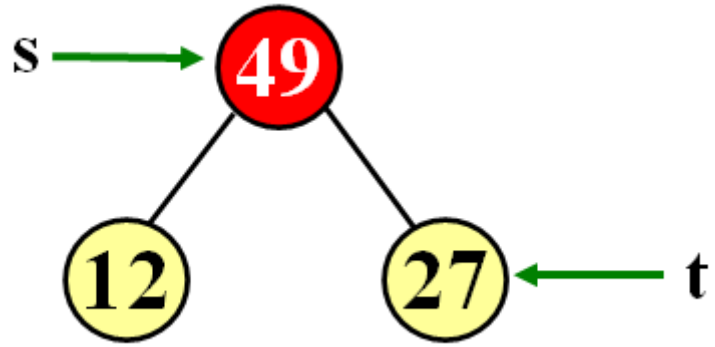
不是堆: {12, 36, 27, 65, 40, 14, 98, 81, 73, 55}

排序思路：

堆，一般采用顺序表存储数据。

如何建堆?——重点是将顺序表r[ ]整理成堆。

- **设r[s..t]中的关键字除r[s]外均满足小顶堆的特征(二叉树的基本形状满足要求)。**



- **自上而下调整关键字r[s]的位置，使r[s..t]满足小顶堆特征。**

代码实现：

1. **调整一个顶点使其符合小顶堆**

```
template<class T>
void HeapAdjust(T L[], int s, int t) {
    int j;
    for (j = 2 * s; j <=t; j *= 2) {
        if (L[j] > L[j + 1]&&j+1<=t)j++;
        if (L[s] < L[j])break;
        swap<T>(L[s], L[j]);
        s = j;
    }
}
```

2. **进行堆排序**

```
template<class T>
void HeapSort(T L[], int n) {
    for (int i = n / 2; i > 0; i--) //从下到上调整每个父结点
        HeapAdjust<T>(L, i, n);
    for (int i = n; i > 1; i--) {//输出根节点，然后重新调整
        cout << L[1] << " ";
        swap<T>(L[1], L[i]);
        HeapAdjust<T>(L, 1, i-1);
    }
    cout << L[1]<<endl;
}
```

6.5计数排序

计数排序是一种计算式算法，要求数据元素为整型。

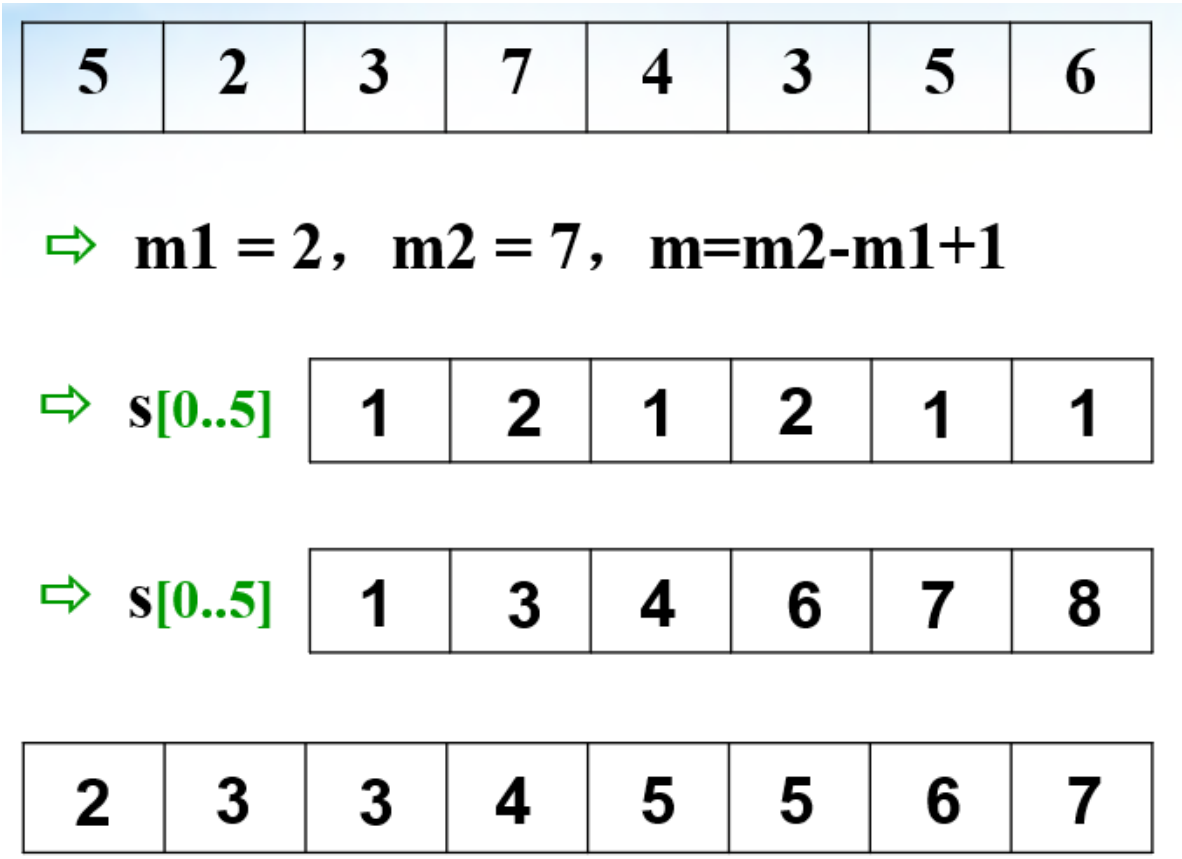
设数据元素(整数)的个数为n，数据元素最大值和最小值的差值为m-1。

如果m<log2n，计数排序算法的时间复杂度小于比较式排序算法。

基本思路：

对于i=1,...,n，计算r[i-1]属于n个整数中第几小的数，并据此将它存放到相应的位置上。

1. **找出数组r中最小值m1和最大值m2;**
2. **统计数组r中每个值为i的整数出现的次数，存入数组s的s[i-m1]项;**
3. **对i=2, ..., m2-m1+1，计算s[i]+=s[i-1];**
4. **填充目标数组：将每个值为i的整数存放在数组的L[s[i]]项，并执行s[i]-1。**



代码实现：

```
template<class T>
void CountSort(T L[], int n) {
    int min = INT_MAX, max = INT_MIN;
    int i, j;
    T* r = new T[n];
    for (i = 1; i <= n; i++) {
        if (L[i] < min)min = L[i];
        if (L[i] > max)max = L[i];
    }
    int m = max - min + 1;
    T* s = new T[m]{ 0 };
    for (i = 1; i <= n; i++)
        s[L[i] - min]++;
    for (i = 1; i < m; i++)
        s[i] += s[i - 1];
    for (i = n; i > 0; i--) {
        r[s[L[i] - min]-1] = L[i];
        s[L[i] - min]--;
    }
    for (i = 0; i < n; i++)
        L[i+1] = r[i];
    delete s, r;
}
```

6.6树形选择排序

树形选择排序又称锦标赛排序，是一种按照锦标赛的思想进行选择排序的方法。首先对n个记录的关键字进行两两比较，然后在n/2个较小者之间再进行两两比较，如此重复，直至选出最小的记录为止。

树形选择排序的思路来自于锦标赛/淘汰赛/选拔赛。

树形选择排序是一种在"完全二叉树"上完成的排序算法。

难点：如何生成和保存这棵完全二叉树?

举例：找出{70, 73, 69, 23, 93, 18, 11}序列中的最小值 (n=7) => 形成一棵完全二叉树。

完全二叉树的前h-1层是1棵满二叉树，共有2^(h-1-1个结点。n个数据存放在完全二叉树的底层，编号从2^(h-1)到2^(h-1)+n-1。

n个数据需要多高的完全二叉树? (h=[log2n])

基本思路：

1. **初始化 (时间复杂度为O(n))**
    - **计算满二叉树的高度 h = [log2] （向上取整）**
    - **满二叉树的结点值都置成 ∞ (令k=2^h-1)：for (i=1; i<=k; i++) L[i]= ∞;**
    - **n个数据存放在底层：L[k+1..k+n]。**
    - **注意，还需设置L[k+n+1]= ∞。**

代码实现：

```
template<class T>
void TreeSort(T L[],int n) {
    int h = (int)(log(n) / log(2))+1;
    int k = (int)pow(2, h) - 1, i, j,p;
    int start=0, end=0;
    T* tree = new T[long(k + n + 2)];
```

```
    //初始化树
    for (i = 1; i <= k; i++)
        tree[i] = INT_MAX;
    tree[k + n + 1] = INT_MAX;
    for (i = k + 1; i <= k + n; i++)
        tree[i] = L[i - k];

    //构建完全二叉树
    for (i = h; i >= 1; i--) {
        start = (int)pow(2, i);
        end = (int)pow(2, i + 1);
        for (j = start; j < end && j <= k + n; j += 2)
            tree[j / 2] = tree[j] < tree[j + 1] ? tree[j] : tree[j + 1];//将2n
与2n+1中较小值赋给父结点
    }

    for (i = 1; i <= n; i++) {
        cout << tree[1]<<" ";
        j = 1;
        while (tree[2 * j] == tree[1] || tree[2 * j + 1] == tree[1]) {
            j *= 2;
            if (tree[j] != tree[1])j++;
        }
        tree[j] = INT_MAX;

        //用最小值结点的兄弟结点来依次比较父结点的值，兄弟结点值小则覆盖
        for (p = j; p > 1; p /= 2) {
            if (p % 2)j = tree[p - 1];
            else j = tree[p+1];
            if (j < tree[p])tree[p / 2] = j;
            else tree[p/ 2] = tree[p];
        }
    }
}
```

6.7基数排序

假如多关键字的记录序列中，每个关键字的取值范围相同(如十进制数)，则按 LSD (Least Significant Digit first，最低位优先法) 法进行排序时，可以采用 “分配-收集” 法，该方法不需要进行关键字之间的比较。

基数排序方法的基本思路是，将单关键字看成是由多个数位 (或多个字符)构成的多关键字，并采用 “分配-收集”的方法进行排序。

基本步骤：给定序列209, 386, 768, 185, 606, 230, 83, 539

1. **首先按个位数取值分别为0, 1, ..., 9分配成10组，之后按从0至9的顺序将它们收集在一起；**
2. **然后按十位数取值分别为0, 1, ..., 9分配成10组，之后按从0至9的顺序将它们收集在一起；**
3. **最后按百位数取值分别为0, 1, ..., 9分配成10组，之后按从0至9的顺序将它们收集在一起；**
4. **.....**

基数排序一般采用链表存储结构，即链式基数排序。操作：

1. **将待排记录建成一个链表；**
2. **分配时，将当前关键字位值相同的记录分配到同一个链队列中；**
3. **收集时，按当前关键字位取值从小到大顺序将各链队列链成一个链表；**

4. **对每个关键字位重复(2)和(3)两步。**

代码实现：

```
template<class T>
void RadixSort(RadixNode<T>* L, int n) {
    int radix = 10,i,j,k=0,l;
    RadixNode<T>* p,*r;
    RadixHead<T> head;
    p = L->next;
    while (p) {
        i = p->data;j = 0;
        while (i) {
            i /= radix;j++;
        }
        if (j > k)k = j;
        p = p->next;
    }
    for (l = 0; l < k; l++) {
        head = RadixHead<T>();
        p = L->next;
        //分配算法
        while (p) {
            i = p->data1 % radix;
            r = new RadixNode<T>(p->data, p->data1/radix);
            if (!head.H[i].next)head.H[i].next = r;
            else head.R[i].next->next = r;
            head.R[i].next = r;
            p = p->next;
        }
        //收集算法
        j = 0;
        while (!head.H[j].next)j++;
        L->next = head.H[j].next;
        for (i = j + 1; i < radix; i++) {
            if (!head.H[i].next)continue;
            head.R[j].next->next = head.H[i].next;
            j = i;
        }
    }
}
```

基数排序的时间复杂度为O(k(n+r))

其中：分配算法时间复杂度为O(n)，收集为O(r)，r=Radix为“基数”，k为“分配-收集”的趟数

6.8桶排序(先不学了)

定义：假设输入是由随机过程产生的[0, 1)区间上均匀分布的实数。将区间[0, 1)划分为n个大小相等的桶(子区间)，每桶大小为1/n。将n个元素分配到这些桶中，对桶中元素进行排序，然后依次连接桶。