数据结构与程序设计笔记

版权说明

本数据结构与程序设计笔记由：

Yang Hui，LZR结合课堂内容整理

目录

指针导论

运算优先级

C语言运算符优先级 rightrat

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式)/函数名(形参表)		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		单目运算符
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	表达式%表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	表达式>>表达式	左到右	双目运算符
	>>	右移	表达式<<表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!=表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式1?表达式2:表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式1, 表达式2...	左到右	从左向右运算

特例说明

表达式	表示含义
int *a()	一个函数，返回值为int指针
int (*a)()	一个函数指针，返回值为int
int *a[10]	一个10元数组，里面是int指针
int (*a)[10]	一个int指针，指向一个10元数组
*a ++和 *(a ++)	先取a的元素，然后a+1
* ++a	先a+1，然后取a的元素
* (a++)->key	先取a指针中key指针的元素，然后a+1

对于取地址、自增等操作，从右往左运行；取成员优先进行；后置的自增运算符最后执行

其他零散知识点

1. 字符串数组可以在定义的时候初始化直接赋值一个常量字符串；字符指针可以直接指向一个字符串常量，但是不可以修改里面的数值；
2. 常量字符串的末尾如果有空间会自动加上\0，如果给出的字符串空间不足自动截取尾部；
3. 所有的数组名全部是常量指针，不可以重新赋值和更改位移；
4. struct类型的定义方法：

```
struct node
{
    int num;
    char c;
};
```

定义了一个struct node类型；

```
struct node
{
    int num;
    char c;
} Knode;
```

定义了一个struct node类型和该类型的一个变量knode;

```
struct
{
    int num;
    char c;
} Knode;
```

定义了一个无名称结构类型，该结构有一个变量Knode；

```
typedef struct
{
    int num;
    char c;
} Node;
```

定义了一个无名称结构类型，然后取名为Node;

线性表

线性表的顺序实现与算法

线性表的链表实现与算法

数据结构

```
typedef struct Node
{
    int num;
    struct Node *next;
} Node;
```

顺序插入

```
void OrderInsert(Node *Head,Node *Insert)
{
    Node *p = Head->next,*ptr = Head;
    for (;p != NULL && p->num < Insert->num;ptr = p,p = p->next);
    if (p == NULL && ptr == Head) //空表插入
        Head->next = Insert;
    else if (p == NULL && ptr != Head) //末尾插入
        ptr->next = Insert;
    else
    {
        ptr->next = Insert;
        Insert->next = p;
    }
    return;
} //Head需要事先分配空间，由小到大排列
```

查找可以通过修改顺序插入的办法来做，效率O(n)

队列

循环队列实现与算法

数据结构

```
int Queue[MAXLEN];
int Head = 0,Rear = MAXLEN - 1,cnt = 0;
```

算法

```
int OutQueue()
```

```
{
    if (cnt == 0)
        return ERROR;
    return Queue[++ Rear];
}

int InQueue(int num)
{
    if (cnt == MAXLEN)
        return ERROR;
    Queue[Head ++] = num;
    return 1;
}
```

栈

栈的理论知识点

特点：LIFO（Last in First out）、元素间呈线性关系、插入删除在一端进行。

定义：只允许在表的一端进行插入操作和删除操作的线性表。允许操作的一端称为栈顶，栈顶元素的位置由一个称为栈顶位置的变量给出。当表中没有元素时，称之为空栈。

操作：入栈、出栈、判断栈空/满、取栈顶元素。

储存方式：顺序存储和链式存储

1.顺序存储（一维数组Stack[]+int型栈顶指针top）

初始化：

```
Elemtype Stack[MAXSIZE];
```

```
int top=-1; //调用时“++top”;
```

```
int top=0;  //调用时“top++”;
```

2.链式存储（链表+指向栈顶元素的指针p）

结构类型：

```
typedef struct node
{
    Elemtype data;
    struct node *link;
} Node, *Nodeptr;
```

初始化：

```
 Nodeptr top=NULL;
```

多栈共享空间问题（实际coding时候大概率用不到）：

定义top1、top2分别从-1和MAXSIZE向中间移动（入栈1则++top1，入栈2则--top2，出栈类推）

当top1==top2 - 1时候栈满，top1==-1时栈1空，top2==MAXSIEZE时栈2空。

栈的相关算法

0. 定义部分：

```
#define MAXSIZE 1024//设置最大长度为1024
int top=-1;
Elemtype STACK[MAXSIZE];//Elemtype为自己设定的数据类型
```

```
typedef struct node
{
    Elemtype data;
    struct node *link;
}Node, *Nodeptr;
Nodeptr top=NULL;
```

1. 初始化栈：

```
void  initStack( )
{
   Top= -1;
}
```

```
void initStack(  )
{
   Top=NULL;
}
```

2. 判断栈空：

```
int  isEmpty( )
{
       return Top== -1;
}
```

```
int  isEmpty( )
{
       return Top==NULL;
}
```

3. 判断栈满：

```
int isFull(  )
{
      return Top==MAXSIZE-1;
}
```

```
//只要内存充足，链栈元素个数没有上限
```

4. 入栈:

```
void push(Elemtype STACK[],Elemtype item)
{
    if(isFull())
    {
        fprintf(stderr,"FULL!");
        exit(0);
    }
    else
        STACK[++top]=item;
}
```

```
void  push( ElemType item )
{
    Nodeptr p;
    if( (p=(Nodeptr)malloc(sizeof(Node)))==NULL)
    {
        fprintf(stderr,"NO MEMORY!");
        exit(0);
    }//可能性不大 if可以注释掉
    else
    {
        p->data=item;           /*将item送新结点数据域*/
        p->link=Top;            /*将新结点插在链表最前面*/
        top=p;                  /*修改栈顶指针的指向*/
    }
}
```

5. 出栈:

```
ElemType pop( ElemType s[ ])
{
    if(isEmpty()){
        fprintf(stderr,"EMPTY!");
        exit(0);
    }
    else
        return s[top --]
}
```

```
ElemType  pop( )
{
```

```
    Nodeptr p;
    ElemType item;
    if ( isEmpty() )
    {
        fprintf(stderr,"EMPTY!");
        exit(0);
    }                                           /* 栈中无元素*/
    else
    {
        p=Top;
        item=Top->data;                         /*保存被删栈顶的数据信息*/
        Top=Top->link;                          /* 删除栈顶结点 */
        free(p);                                /* 释放被删除结点*/
        return item;                            /* 返回出栈元素*/
    }
}
```

表达式计算

中缀表达式->后缀表达式

定义+ -优先级为3，* /优先级为2，（）优先级为1

那么运算方法如下：

1. 如果遇到数字，则直接打出到输出流中，计算进入数据栈
2. 如果遇到符号，则进行如下判定：
    1. 如果不是右括号：对之前的栈进行遍历，如果前面的符号优先级小于等于当前优先级，则这个符号出栈运算，并且打出到输出流；直到后一个符号的优先级大于前一个或者为左括号
    2. 如果是右括号，那么直接打到左括号为止，并且舍弃掉左括号，右括号不入栈
3. 注意有可能出栈会有空，一旦检测到出栈为空就终止出栈，并且把该符号入栈
4. 如果遇到了等于号，所有的符号全部出栈打入输出流并且计算数值，等于号本身不入栈。

后缀表达式->中缀表达式

构造表达式树：

所有的字符（数字、符号）全部视作一个二叉树节点，放在一个输入的流里面，然后开始遍历这个流：如果遇到一个数字，则直接把数字生成一个节点然后放入到栈中；如果遇到一个符号，那么出栈两个节点然后把这两个节点挂在符号的左右子树上，随后该符号节点入栈。

最后栈顶的唯一元素就是表达式树

表达式树生成中缀表达式需要注意括号：

1. 如果父节点的优先级大于左子节点的优先级，那么输出一个括号，然后在该子树遍历完之后补全括号；
2. 如果父节点为/ %或者父节点为* -并且右子节点为+ -，那么输出一个括号，右子树遍历完毕以后补全括号。

```
#include <stdio.h>
#include <ctype.h>
#define MAXSIZE 512
```

```
enum type{NUM,OP,EQ,EMP};
enum op{EQU,ADD,SUB,MUL,DIV,MOD,LEFT,RIGHT};
typedef struct node
{
    int num;
    enum type type;
    enum op op;
} node;
int num_stack[MAXSIZE];
enum op op_stack[MAXSIZE];
int num_top = -1,op_top = -1;
int priority[] = {-1,0,0,1,1,1,2,2};
node getnode();
void process(node);
int caculate(enum op);

int main()
{
    node p;
    while(p = getnode(),p.type != EQ)
    {
        if (p.type == EMP)
            continue;
        else if (p.type == NUM)
            num_stack[++ num_top] = p.num;
        else if (p.type == OP)
            process(p);
    }
    while(op_top >= 0)
    {
        caculate(op_stack[op_top --]);
    }
    printf("%d",num_stack[0]);
}

int caculate(enum op op)
{
    int b = num_stack[num_top --];
    int a = num_stack[num_top --];
    int t;
    switch (op)
    {
        case ADD: t = a + b; break;
        case SUB: t = a - b; break;
        case MUL: t = a * b; break;
        case DIV: t = a / b; break;
        case MOD: t = a % b; break;
    }
    num_stack[++ num_top] = t;
    return 1;
}
void process(node p)
{
    if (p.op != RIGHT)
    {
        while(priority[p.op] <= priority[op_stack[op_top]] && op_stack[op_top]
!= LEFT)
            caculate(op_stack[op_top --]);
```

```
        op_stack[++ op_top] = p.op;
    }
    else
    {
        while(op_stack[op_top --] != LEFT)
            caculate(op_stack[op_top + 1]);
    }
}

node getnode()
{
    node p;
    p.type = EMP;
    int num = 0,flag = 0;
    char c = getchar();
    while(isdigit(c))
    {
        flag = 1;
        num *= 10;
        num += c - '0';
        c = getchar();
    }
    if (flag == 1)
    {
        p.type = NUM;
        p.num = num;
        ungetc(c,stdin);
        return p;
    }
    switch (c)
    {
        case '+':
            p.op = ADD;
            p.type = OP;
            break;
        case '-':
            p.op = SUB;
            p.type = OP;
            break;
        case '*':
            p.op = MUL;
            p.type = OP;
            break;
        case '/':
            p.op = DIV;
            p.type = OP;
            break;
        case '%':
            p.op = MOD;
            p.type = OP;
            break;
        case '(':
            p.op = LEFT;
            p.type = OP;
            break;
        case ')':
            p.op = RIGHT;
            p.type = OP;
```

```
            break;
        case '=':
            p.op = EQU;
            p.type = EQ;
            break;
        default:
            break;
    }
    return p;
}
```

串

串的基本相关算法

串的中间插入和删除

```
void StringInsert(char *str,char *new,int pos) //在pos前面插入new字符串，如果
n>strlen-1，直接末尾插入
{
    if (pos > strlen(str) - 1)
        strcat(str,new);
    else
    {
        char *p = (char *) malloc (sizeof(MAXLEN));
        strcpy(p,str + pos);
        *(str + pos) = 0;
        strcat(str,new);
        strcat(str,p);
        free(p);
    }
}

void StringDelete(char *str,int pos,int n) //删除包括pos在内的n个字符
{
    if (pos > strlen(str))
        return;
    if (pos + n >= strlen(str))
        *(str + pos) = 0;
    else
    {
        *(str + pos) = 0;
        strcat(str,str + pos + n);
    }
    return;
}
```

KMP算法

next数组的生成

```
void getNext(int *next,char *str_sub)
{
    next[0] = -1;
    int i = 0,j = -1;
    int len = strlen(str_sub)
    while(i < len)
    {
        if (j == -1 || str_sub[j] == str_sub[i])
        {
            i ++;
            j ++;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

主匹配算法

```
int KMPCompare(char *str,char *str_sub)
{
    int i = 0,j = 0;
    int len = strlen(str);
    int len_sub = strlen(str_sub);
    int *next = (int *) malloc (sizeof(int) * len_sub);
    getNext(next,str_sub);
    while(i < len && j < len_sub)
    {
        if (j == 0 || str[i] == str_sub[j])
        {
            i ++;
            j ++;
        }
        else
            j = next[j];
    }
    if (j >= len_sub)
        return i - len_sub;
    else
        return -1;
}
```

主匹配算法返回字串第一次出现的位置的下标。

树

树的理论知识点

树的高度——层数

树的度：最大的子节点数量

树的节点数量N和边数量L满足：（N_x表示有x个孩子的节点）

$$0 * N_0 + 1 * N_1 + \ldots + n * N_n = N - 1$$

森林和二叉树：

- 对于一棵树，某个节点的右兄弟等于二叉树中的右孩子，第一个孩子等于二叉树中的左孩子
- 对二叉树进行先序遍历等于对树进行先序遍历，对二叉树进行后序遍历等于对树进行后序遍历
- 没有孩子的节点没有左子树，没有右兄弟的节点没有右子树

二叉树的理论知识点

#define N 节点数

#define N_x 某行节点数

#define h 深度

#define l 边数

完全二叉树：层次全满，有关系式：

$$N = 2^h - 1$$

$$N_x = 2^{x-1}$$

满二叉树：只有最下一层不满：

$$N <= 2^h - 1$$

$$N_l <= 2^{h-1}$$

对于顺序存储的二叉树（完全二叉树）

（0下标开始）有：

$$\text{左孩子：} Tree[2i + 1]$$

$$\text{右孩子：} Tree[2i + 2]$$

$$\text{双亲节点：} Tree[(i - 1)/2]$$

（1下标开始）有：

二叉树的相关算法

1. 树的生成
2. 树的前序遍历

   先访问根结点，然后访问左子树，再访问右子树

```
void FirstRootSearch(TNode *Root)
{
    if (Root == NULL)
        return;
    visit(Root);
    FirstRootSearch(Root->left);
    FirstRootSearch(Root->right);
}
```

3. 树的中序遍历

   先访问左子树，然后访问根节点，再访问右子树

```
void FirstRootSearch(TNode *Root)
{
    if (Root == NULL)
        return;
    FirstRootSearch(Root->left);
    visit(Root);
    FirstRootSearch(Root->right);
}
```

4. 树的后序遍历

   先访问左子树，然后访问根节点，再访问右子树

```
void FirstRootSearch(TNode *Root)
{
    if (Root == NULL)
        return;
    FirstRootSearch(Root->left);
    FirstRootSearch(Root->right);
    visit(Root);
}
```

哈夫曼树

生成哈夫曼树的方法：

1. 统计字符数量，然后排序
2. 每个字符构造为一个二叉树节点，节点中记录一个数值为当前节点下所有字符的总数量
3. 选取最小的两个节点，创造一个新的节点连接这两个节点，新节点的数值为两个小节点之和，原来的两个节点从排序数组中删除，然后把新节点插入到排序数组中，插入后依然有序
4. 重复步骤3一共n-1次，直到只剩下一个节点，该节点就是哈夫曼树的根

5. 如果要生成哈夫曼编码，对哈夫曼树进行前序遍历，并且记录路程中的左右顺序（01序列），找到以后直接输出

树的相关算法

数据结构

```
#define MAXNUM 16
typedef struct Tnode
{
    int num;
    struct Tnode *child[MAXNUM];
} Tnode;
```

1. 一般树的先序遍历

```
void FirstSearch(Tnode *Root)
{
    int i;
    if (Root != NULL)
    {
        visit(Root);
        for (i = 0;i < MAXNUM;i ++)
            FirstSearch(Root->child[i]);
    }
    return;
}
```

2. 一般树的后序遍历

```
void RearSearch(Tnode *Root)
{
    int i;
    if (Root != NULL)
    {
        for (i = 0;i < MAXNUM;i ++)
            FirstSearch(Root->child[i]);
        visit(Root);
    }
    return;
}
```

3. 一般树的层次遍历

```
Tnode *queue[MAXLEN];
int Head = 0,Rear = MAXLEN - 1,cnt = 0;
void Search(Tnode *Root)
{
    int i;
    if (Root != NULL)
    {
        visit(Root);
        for (i = 0;i < MAXNUM;i ++)
```

```
            InQueue(Root->child[i]);
        Search(OutQueue());
    }
    return;
}

void InQueue(Tnode *p)
{
    if (cnt != MAXLEN)
        queue[Head ++] = p;
}

Tnode *OutQueue()
{
    if (cnt == 0)
        return NULL;
    return queue[-- Rear];
}
```

图

图的理论知识点

最小单元为顶点，连接的关系有无向图和有向图。

顶点和其他顶点的边数量叫做度，对于有向图，由该顶点指出的边数目叫做出度，指向该顶点的边叫做入度。无向图可以理解为任何一个边都是双向的。

无向图最多有n(n-1)/2个边，有向图可以有n(n-1)个。（不考虑自连和多重连接）

如果考虑连接的权重，会构成一个新的关系叫做网。

最大连通子图：对于一个图可以根据路径的方向走通的所有节点构成一个最大连通子图。如果有向图任何两个顶点连通，则为强连通图。

路径：在一个最大连通子图中，从一个顶点到另外一个顶点经过的顶点序列，没有重复顶点的叫做简单路径。

图的存储结构

顶点存储：数组

边存储：

1. 邻接矩阵——二维数组存储：用一个二维数组，Graph [i] [j] 表示i->j的边。

   如果是无向图表示i-j，则Graph [i] [j] = 1; Graph [j] [i] = 1，不存在边关系的对应位置为0；

   如果是有向图表示i->j，则Graph [i] [j] = 1，不存在边关系的对应位置为0；

   如果是网，则定义INF=1145141919810，有边的地方存储权重，没有边的地方存储INF

   无向图一定要记得矩阵的对称位置要赋值啊

2. 邻接链表——链式存储：数组本身定义为一个链表类型，数组中每一个链表中除了头节点（表示顶点自己）其他节点表示该顶点出度边所指向的顶点。

图的基本相关算法

构造算法

包括邻接矩阵和邻接表

遍历算法

包括BFS和DFS

数据结构：

```
typedef struct Node
{
    int data; //表示数组表示的顶点的信息
    int num; //表示作为出度对应顶点的编号
    struct Node *next;
} Node;
Node Graph[MAXLEN]; //末尾节点初始化NULL
char Visited[MAXLEN]; //表示已经遍历过的节点
```

DFS

```
void DFS(int node) //递归写法
{
    Visited[node] = 1;
    visit(&Graph[node]);
    Node *p = Graph[node].next;
    while(p != NULL)
    {
        if (Visited[p.num] == 0)
            DFS(p.num);
        p = p->next;
    }
    return;
}

void DFS_2(int node)
{
    Visited[node]
}
```

BFS

```
void BFS(int node)
{
    Visited[node] = 1;
    visit(&Graph[node]);
    Node *p = Graph[node].next;
    while(p != NULL)
    {
        if (Visited[p.num] == 0)
        {
            Visited[p.num] = 1;
            inQueue(p.num);
            Depth[p.num] = Depth[node] + 1;
```

```
        }
        p = p->next;
    }
    BFS(outQueue());
    return;
}

void BFS_2(int node)
{
    inQueue(node);
    while(!isEmptyQueue())
    {
        int num = outQueue();
        visit(&Graph[num]);
        Node *p = Graph[node].next;
        while(p != NULL)
        {
            if (Visited[p.num] == 0)
            {
                Visited[p.num] = 1;
                inQueue(p.num);
                Depth[p.num] = Depth[node] + 1;
            }
            p = p->next;
        }
    }
    return;
}
```

如果使用邻接矩阵存储，那么计算方法类似，对该点所有节点的搜索转为对Graph矩阵第node行的遍历。

最小生成树

包含具有n个顶点的连通图G的全部n个顶点,仅包含其n-1条边的极小连通子图称为G的一个生成树

性质：

1. 包含n个顶点的图：连通且仅有n-1条边

<=>无回路且仅有n-1条边

<=>无回路且连通

<=>是一棵树

2. 如果n个顶点的图中只有少于n-1条边，图将不连通
3. 如果n个顶点的图中有多于n-1条边，图将有环（回路）
4. 一般情况下，生成树不唯一

Prim算法

复杂度n^2

数据结构

```
int Graph[MANLEN][MAXLEN]; //图
int Tree[MAXLEN]; //表示树：i的父节点路径为Route[i]，初始化为源点序号
int Weight[MAXLEN]; //表示当前到达已有生成树的最小权重，初始化为第一个节点的权重位置
int Visited[MAXLEN];
```

随后不进行代码说明：

循环n-1次：

1. 首先找到Weight中最小的点k，然后Visited[k] = 1;
2. 然后在Graph中遍历k的出度，如果有Visited[i] == 0 && Graph [k] [i] < Weight[i]，也就是i节点走k到树的距离比直接到当前的点更快，执行：

```
Weight[i] = Graph[k][i];
Tree[i] = k;
```

最终Tree为生成树。

Kruskal算法

复杂度$n\log_2 n$

数据结构

```
typedef struct Side
{
    int Weight;
    int From;
    int to;
} Side;
int Graph[MAXLEN][MAXLEN];
Side SideNode[MAXLEN * MAXLEN];
int Tree[MAXLEN];
```

随后不进行代码说明：

初始化中，把Graph中所有信息输入到SideNode数组中去，然后对该数组进行排序

随后开始循环：

1. 选取SideNode中未访问的最小的边，然后对From和to进行检查溯源，溯源方法是通过Tree数组找回其是否有共同的根（根的Tree数组值为0，用此来判定是否为根），如果不同源，则Tree[to] = From，并且计数+1
2. 如果计数达到了n-1，循环终止，Tree中为生成树

最短路径

选取一个顶点为源点。

构造数据结构：

```
int Route[MAXLEN]; //用Route[i]表示i的父节点路径为Route[i]，初始化为源点序号
int Distance[MAXLEN]; //表示i到源点的距离，初始化直接从Graph中获取
int Graph[MAXLEN][MAXLEN]; //图
int Visited[MAXLEN];
```

随后不进行代码说明：

循环n-1次：

1. 首先找到Distance中最小的点k，然后Visited[k] = 1;
2. 然后在Graph中遍历k的出度，如果有Visited[i] == 0 && Graph[k] [i] + Distance[k] < Distance[i]，也就是i节点走k到源点比直接到源点更快，那么执行：

```
Distance[i] = Distance[k] + Graph[k][i];
Route[i] = k;
```

循环完毕以后，如果寻找点k的最短路径，那么就是从Route[k]开始一直找到源点的编号；

AOV图

AOV的拓扑序列：

寻找一个没有入度的顶点，加入到序列以后删除它所有的出路径，递归，直到不存在没有入度的点。如果所有点全部入序列，则为AOV图，否则不是。

关键路径：耗时最长的路径，不止一条。关键路径的延迟一定会带来整个任务的延迟，非关键路径不一定；关键路径全部提前一定会带来整个任务的提前完成，只有一个提前完成不一定。

最早发生时间和最晚发生时间；

时间容差；

最早发生时间的计算：在该顶点的所有入度中选取max{T_n+t_n}（入度顶点的最早时刻+任务需要的时间：最大值）

最晚发生时间的计算：在该顶点的所有出度中选取min{T_n-t_n}（出度顶点的最晚时刻-任务需要的时间：最小值）

最早时间和最晚时间差叫做时间容差，两个相同的顶点连接成关键路径。

查找

基本概念

1.ASL（平均查找长度）：确定一个记录在查找表中的位置所需要进行的关键字值的比较次数的期望值(平均值)。

其中，pi为查找第i个记录的概率，ci为查找第i个记录所进行过的关键字的比较次数。

2.判定树：若把当前查找范围内居中的记录的位置作为根结点，前半部分与后半部分的记录的位置分别构成根结点的左子树与右子树,则由此得到一棵称为“判定树”的二叉树，利用它来描述折半查找的过程。

成功的查找过程正好等于走了一条从根结点到被查找结点的路径，经历的比较次数恰好是被查找结点在二叉树中所处的层次数 。

顺序查找

描述：从表的第一个记录开始,将用户给出的关键字值与当前被查找记录的关键字值进行比较，若匹配，则查找成功，给出被查到的记录在表中的位置，查找结束。若所有n 个记录的关键字值都已比较，不存在与用户要查的关键字值匹配的记录，则查找失败，给出信息0。

```
int  search(keytype key[ ],int n,keytype k)
{
       int  i;
       for(i=0;i<n; i++)
             if(key[i]==k)
                  return i;
       return -1;
}
```

对于具有n个记录的顺序表，若查找概率相等，则有:

二分查找

将要查找的关键字值与当前查找范围内位置居中的记录的关键字的值进行比较。

若匹配，则查找成功，给出被查到记录在文件中的位置，查找结束。

若要查找的关键字值小于位置居中的记录的关键字值，则到当前查找范围的前半部分重复上述查找过程，否则，到当前查找范围的后半部分重复上述查找过程，直到查找成功或者失败。
若查找失败，则给出错误信息（0）。

平均查找长度为:

对于n个元素，查找长度为:

二分查找的插入

二分查找使用以下代码，插入点为low，如果low>=length，则末尾插入，同时low也是第一个大于key的元素

```
int n,low,high,mid;//n为记录的个数
low=0;
high=n-1;
mid=low+high/2;//实际上是向下取整
```

```
int  binsearch(keytype key[ ], int n, keytype k)
{
     int low=0, high=n-1, mid;
       while(low<=high){//注意判定条件!
              mid=(low+high)/2;
              if(k==key[mid])
                    return mid;              /*  查找成功  */
              if(k>key[mid])
                    low=mid+1;             /*  准备查找后半部分 */
              else
```

```
                    high=mid-1;            /* 准备查找前半部分 */
        }
        return -1;                                /*    查找失败   */
}
```

```
int binsearch2(keytype key[ ], int low, int high, keytype k)
{
     int  mid;
     if(low>high)
           return -1;
     else{
           mid=(low+high)/2;
           if(k==key[mid])
                 return mid;
           else
                 if(k<key[mid])
                     return  binsearch2(key,low,mid-1,k);
                 else
                     return  binsearch2(key,mid+1,high,k);
           }
 }
```

进化之插值查找：mid=low + (high-low)*(k-a[low])/(a[high]-a[low])

缺点是只能进行从某个元素开始进行查找，不能限制双侧范围

二分查找树

搜索操作：

```
BTNodeptr  searchBST(BTNodeptr p, Datatype item)
{
    if(p == NULL){
        p = (BTNodeptr)malloc(sizeof(BTNode));
        p->data = item;
        p->lchild = p->rchild = NULL;
    }
    else if( item < p->data)
        p->lchild = insertBST(p->lchild, item);
    else if( item > p->data)
       p->rchild = insertBST(p->rchild,item);
    else
       do-something; //找到该元素
    return p;
}
```

Trie树

一种用于描述单词的trie结构定义

```
struct tnode {  // word tree
    char isword;    // is or not a word
    char isleaf;    // is or not a leaf node
    struct tnode *ptr[26];
};
```

基于trie结构的单词树的构造

```
void wordTree(struct tnode *root,char *w)  /* install w at or below p */
{
    struct tnode *p;
    for(p=root; *w != '\0'; w++){
        if(p->ptr[*w-'a'] == NULL) {
            p->ptr[*w-'a'] = talloc();
            p->isleaf = 0;
        }
        p = p->ptr[*w-'a'];
    }
    p->isword = 1;
}
struct tnode *talloc()
{   int i;
    struct tnode *p;
    p = (struct tnode *)malloc(sizeof(struct tnode));
    isword = 0; isleaf = 1;
    for(i=0; i<26; i++)
        ptr[i] = NULL;
    return p;
}
```

一个读入字符序列的函数：

```
char *fget_word(char *word_cache,FILE *input){
    char c;
    int top = 0;
    while(!isalpha(c = fgetc(input)) && c != EOF);
    if (c == EOF)
        return NULL;
    if (isupper(c))
        c = tolower(c);
    word_cache[top ++] = c;
    while(isalpha(c = fgetc(input)) && c != EOF)
    {
        if (isupper(c))
            c = tolower(c);
        word_cache[top ++] = c;
    }
    word_cache[top] = '\0';
    return word_cache;
}
```

B-树

所有叶子都在同样深度

数据结构模式：

n（分叉数-1）p_0 Key_1 p_1 … Key_n p_n

其中p是指针，p_i指向的节点的所有key的大小小于Key_i

```
typedef struct BNode
{
    int num;
    int Key[MAX + 1];
    struct BNode *Childe[MAX + 1];
    struct BNode *parent;
}
```

B-树的查找

从根节点出发，开始查找。查找的时候给一个虚拟的节点INF在key数组的最后，然后在key中找到第一个不小于待查找数num的key：如果key==num，查找完毕，如果key>num，查找key左边的子树，直到查找成功或者子树为NULL查找失败

B-树的插入

先使用查找，如果查找失败，就在对应的叶子节点里面做二分查找插入。如果key的数量大于树的阶数M-1，从M/2节点两侧分裂为两个子树，M/2自身插入到双亲节点里面去，重复如此，直到没有分裂或者到根节点。

如果到了根节点，说明父亲节点为NULL，则生成一个新的节点，两边挂载。

B+树

所有的元素全部在叶节点里面，数据结构和B-树一样，但是Key代表的是右边p所指的子树的最大节点数值

B+树的查找

有两个方法，一个是从叶子节点的链表进入，一个是从顶点进入

顶点查找方法一样，二分查找找到第一个大于Key的数值然后进入它右边的子树查找，一定会在叶子节点的位置查找完毕，如果递归节点为NULL就返回查找失败

B+树的插入

先是二分查找插入，如果叶子节点的数量大于阶数M，那么节点从M/2分裂，M/2包括自己之前的分为一组，其余的分为另外一组，M/2自身复制一个Key插入到双亲节点中去，然后双亲节点插入点右边的指针移动一下给新树留下位置。如果双亲结点也需要分裂则继续。

如果到了根节点，说明父亲节点为NULL，则同时把M/2和M的数值作为Key插入到新的根节点中去，两边挂载

关键：节点分裂和连锁反应

B-树和B+树区别：

1. B-树的每个分支结点中含有该结点中关键字值的个数,B+树没有;
2. B-树的每个分支节点都含有指向关键字值对应记录的指针，而B+树只有叶结点有指向关键字值对应记录的指针;
3. B-树只有一个指向根节点的入口，而B+树的叶结点被链接成为一个不等长的链表， 因此，B+树有两个入口，一个指向根结点，另一个指向最左边的叶结点(即最小关键字所在的叶结点)。

哈希（散列）

根据构造的散列函数与处理冲突的方法将一组关键字映射到一个有限的连续地址集合上，并以关键字在该集合中的“象”作为记录的存储位置，按照这种方法组织起来表称为散列表;建立表的过程称为哈希造表或者散列，得到的存储位置称为散列地址或者杂凑地址。

（将元素和位置建立对应关系）

散列函数：对元素进行处理得到对应值（通常做下标处理）

常用：直接定址法/除留余数法/数字分析法/平方取中法/叠加法/基数转换法

冲突处理：当两个元素有相同的hash值的时候考虑冲突

常用：开放地址法（向后寻找位置：线性探测再散列/二次探测再散列/伪随机再散列）/再散列法/链地址法（较常用，同位置建立链表）

```
typedef struct T
{
    char word[80];
    int num,num1,num2;
    struct T *link;
}Node,*Nodeptr;
```

```
Nodeptr Hashlist[NHASH+1];
```

```
unsigned int hash(char *str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * a + (*str++);
        a *= b;
    }

    return (hash & 0x7FFFFFFF);
}
```

```
Nodeptr insert(Nodeptr t, char str[])
{
    if (t == NULL)
    {
        t = (Nodeptr)malloc(sizeof(Node));
        t->num = 1;
```

```
        strcpy(t->word, str);
        t->lchild = NULL;
        t->rchild = NULL;
        return t;
    }
    if (strcmp(str, t->word) > 0)
    {
        t->lchild = insert(t->lchild, str);
    }
    else if (strcmp(str, t->word) < 0)
    {
        t->rchild = insert(t->rchild, str);
    }
    else
    {
        t->num++;
    }
    return t;
}
```

排序

选择排序

代码实现

```
void SelectSort(int *array,int n) //n总元素数量
{
    int min,min_pos,i,j;
    for (i = 0;i < n - 1;i ++)
    {
        min = array[i];
        for (j = i + 1;j < n;j ++)
        {
            if (array[j] < min)                    //比较comp函数插入
            {
                min = array[j];
                min_pos = j;
            }
        }
        swap(&array[min_pos],&array[i]);
    }
}
```

插入排序

冒泡排序

代码实现:

```
void BubbleSort(int *array,int n) //n总元素数量
{
    int flag = 1,i,j;
    for (i = n - 1;i > 0 && flag == 1;i --)
```

```
    {
        flag = 0;
        for (j = 0;j < i;j ++)
        {
            if (array[j] > array[j + 1])                //比较compare函数插入
            {
                flag = 1;
                swap(&array[i],&array[j]);
            }
        }
    }
}
```

希尔排序

堆排序

代码实现：

1. 堆调整：

```
void HeapAdjust(int *array,int i,int n) //i待调整的堆的顶端，n总元素数量
{
    int j,temp;
    temp = array[i];
    j = i * 2 + 1;                                      //获取子树的位置
    while(j < n)                                        //如果子树存在
    {
        if (j + 1 < n && array[j] < array[j + 1])   //比较两个子树大小，比较函数插入
            j ++;
        if (temp < array[j])                            //检查是否符合堆要求，比较函数插入
        {
            array[(j - 1) / 2] = array[j];          //不符合交换元素位置
            j = j * 2 + 1;                              //寻找下一子树
        }
        else
            break;
    }
    array[(j - 1) / 2] = temp;                          //原来的顶插入到对应的位置上
}
```

2. 堆排序

```
void HeapSort(int *array,int n) //n总元素数量
{
    int i,temp;
    for (i = n / 2;i >= 0;i --)
        HeapAdjust(array,i,n);                    //先构造一个堆
    for (i = n - 1;i > 0;i --)
    {
        swap(&array[i],array[0]);                 //获取堆顶元素并交换
        HeapAdjust(array,0,i);                    //重新构造堆
    }
}
```

二路归并

代码实现：

```
void Merge(int *array,int n)                      //n总元素个数
{
    int *temp;
    temp = (int *) malloc (sizeof(int) * n);
    MergeSort(array,temp,0,n - 1);
    free(temp);
}

void MergeSort(int* array,int *temp,int left,int right) //left左元素下标，right右元素下标
{
    int center;
    if (left < right)
    {
        center = (left + right) >> 1;
        MergeSort(array,temp,center + 1,right);
        MergeSort(array,temp,left,center);        //分而治之，依次迭代
        MergeGather(array,temp,left,center,right);//整合一起，类似于后序遍历
    }
}

void MergeGather(int *array,int *temp,int left,int leftend,int rightend) //left左序列起始下标，leftend左序列终止下标，right右序列终止下标
{
    int i = left,j = leftend + 1;
    int top = left;
    while(i <= leftend && j <= rightend)          //依次整合
    {
        if (array[i] < array[j])                  //比较函数插入
            temp[top ++] = array[i ++];
        else
            temp[top ++] = array[j ++];
    }
    while(i <= leftend)                           //如果右边没有了直接转入左边
        temp[top ++] = array[i ++];
    while(j <= rightend)                          //如果左边没有了直接转入右边
        temp[top ++] = array[j ++];
    for (i = left;i <= rightend;i ++)             //复制一次
        array[i] = temp[i];
}
```

快速排序

代码实现：

```
void QuickSort(int *array,int low,int high) //low左元素下标，high右元素下标
{
    int i,last;                                          //last小于索引的最后一个元素下标
    if (low < high)
    {
        last = low;
        for (i = last + 1;i <= high;i ++)                //从last开始寻找，找到小于目标的
        {
            if (array[i] < array[low])                   //比较函数插入
            {
                last ++;
                swap(&array[i],&array[last]);            // 找到多一个元素last先++然后交换插入
            }
        }
        swap(&array[low],&array[last]);                  //last是小于索引的，交换last和索引
        QuickSort(low,last - 1);
        QuickSort(last + 1,high);
    }
}
```

桶排序