基本概念和术语：

1、数据(data)

所有能输入到计算机中去的描述客观事物的符号。

2. 数据元素(data element)

数据的基本单位， 也称结点(node)或记录(record)。

3、数据项(data item)

有独立含义的数据最小单位也称域(field)。

4、数据对象(Data Object)

是性质相同的数据元素的集合，是数据的一个子集。

5 、数据结构（Data Structure）

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。

数据结构的两个层次：

1、逻辑结构

数据元素间抽象化的相互关系，与数据的存储无关，独立于计算机，它是从具体问题抽象出来的数学模型。

1）线性结构

有且仅有一个开始和一个终端结点，并且所有结点都最多只有一个直接前趋和一个后继。线性表，栈，队列

2）非线性结构

一个结点可能有多个直接前趋和直接后继。树，图

2、存储结构

数据元素及其关系在计算机存储器中的存储方式。

1） 顺序存储结构

借助元素在存储器中的相对位置来表示数据元素间的逻辑关系。顺序表

2） 链式存储结构

借助指示元素存储地址的指针表示数据元素间的逻辑关系。链表

时间复杂度：

语句频度：一条语句的重复执行次数。

算法的执行时间 =

该算法中所有语句的频度之和。

```
for( i = 0; i < n; i++) n
    for( j = 0; j < n; j++) n
     c[i][j] = a[i][j] + b[i][j]; n*n
```

$T(n) = O(n^2)$

空间复杂度:

算法所需存储空间的度量，记作：

S(n)=O(f(n))

线性表

线性表的定义:

由 n(n≥0)类型相同的数据元素构成的有限序列，称为线性表。

一、线性表的顺序存储表示

线性表的顺序存储表示是指用一组地址连续的存储单元依次存放线性表中的数据元素，称这种存储结构的线性表为顺序表。

1.存储结构：

```
typedef  struct {
  ElemType *elem; //存储空间基址
  int  Length;    // 当前长度
  int  listSize;  // 分配的存储容量
} SqList;  // 顺序表
```

2.初始化：

```
Status  InitList( SqList & L,  int maxsize ) {
L.elem = new ElemType[maxsize];
if (!L.elem)
  exit(OVERFLOW);
```

```
L.length = 0;
L.listsize = Maxsize;
return OK;
}
```

算法时间复杂度：O(1).

3.查找：

```
int LocateElem (SqList L, ElemType e)
{
for (i=0;i<L.length;i++)
      if(L.elem[i]==e) return i+1;
return 0;
}
```

算法的时间复杂度：

最好情况： T(n)=O(1)

最坏情况： T(n)=O(n)

平均情况： （n+1）/2 , T(n)=O(n)

4.插入：

```
Status ListInsert(SqList &L, int i,
ElemType e){
if (i < 1 || i > L.length+1)
```

```
    return ERROR;
if (L.length >= L.listsize)
    return  OVERFLOW;
for (j=L.length-1; j>=i-1; --j)
     L.elem[j+1] = L.elem[j];
    L.elem[i-1] = e;
     ++L.length;
 return TRUE;
}
```

算法的时间复杂度：

T(n)=O(n)

5.删除：

```
Status ListDelete(SqList &L, int i)
{
if ((i < 1) || (i > L.length))
     return ERROR;
for (j = i; j<L.length; j++)
     L.elem[j-1] = L.elem[j];
--L.length;
return TRUE;
}
```

T(n) = O(n)

单链表

1.存储结构：

```
typedef struct  LNode {
        ElemType  data;  // 数据域
        struct LNode  *next;  // 指针域
} LNode, *LinkList;
```

2.初始化

构造一个带头结点的空链表。

```
Void InitList(Linklist &L)
{
L= new LNode;
L->next=NULL;
}
```

3.查找

```
Status  Search(LinkList  L,  int  i,
ElemType &e){
p = L->next;
j = 1;
```

```
while (j<i)  {
p = p->next;
++j;
}
if ( !p || j>i )
    return ERROR;
e = p->data;
return OK;
}
```

算法时间复杂度为:O(n)

```
LNode *Searchs (LinkList L,Elemtype e)
{
 p=L->next;
 while(p &&p->data!=e)
       p=p->next;
 return p;
}
```

4.插入：

```
Status ListInsert_L(LinkList &L, int i,
ElemType e) {
p = L;
j = 0;
```

```
while (p && j < i-1)
      { p = p->next;  ++j; }
if (!p || j > i-1)
return ERROR;
s = new LNode;
if ( s == NULL)
return ERROR;
s->data = e;
s->next = p->next;
p->next = s;
return OK;
}  //O(n)
```

5.删除：

```
Status ListDelete(LinkList L, int i,
ElemType &e) {
p = L;
j = 0;
while (p->next && j < i-1)
 { p = p->next;  ++j; }
if  (!(p->next) || j > i-1)
    return ERROR;
```

```
q = p->next; p->next = q->next;
e = q->data;
delete(q);
return OK;
}
```

6.前插法：

```
void CreateList(LinkList &L,int n)
{
    L=new LNode;
   L->next=NULL;
   for(i=n; i>0 ;--i)
          {
             p=new LNode;
            cin>>p->data;
            p->next=L->next;
             L->next=p;
          }
}
```

7.尾插法：

```
void CreateList(LinkList &L, int n)
```

```
 {
    L=new LNode;
    L->next=NULL;
     r=L;
     for(i=0; i<n; ++i)
           {  p=new LNode
               cin>>p->data;
               r->next=p;
               r=p;
           }
p->next=NULL;
}
```

8.置空

```
void ClearList(&L)
{
while (L->next)
    {
      p=L->next;
      L->next=p->next;
    }
}//O（n）
```

三、循环链表

最后一个结点的指针域的指针又指回第一个结点的链表。

四、双向链表

```
typedef struct  DuLNode
 {
ElemType    data;  // 数据域
struct DuLNode  *prior;
struct DuLNode  *next;
 } DuLNode, *DuLinkList;
```

栈

一、顺序栈

1. 定义

只能在表的一端（栈顶）进行插入和删除运算的线性表.

2. 逻辑结构

与线性表相同，仍为一对一关系

3. 存储结构

用顺序栈或链栈存储均可，但以顺序栈更常见。

/ 后进先出

运算规则：

只能在栈顶插入、 删除；

只能存取栈顶元素

存储结构：

```
typedef struct
 { SElemType    *base;
   SElemType    *top;
   int stacksize;
} SqStack;
```

栈空标志：　　**栈满标志：**

base==top S.top - S.base== S.stacksize

初始化：

```
Status  InitStack(  SqStack  &S ,  int
MAXSIZE )
{
  S.base =new SElemType[MAXSIZE];
  if( !S.base ) return OVERFLOW;
  S.top = S.base;
  S.stacksize = MAXSIZE;
  return OK;
}
```

进栈：

```
Status Push( SqStack &S, SElemType e)
{  if( S.top - S.base== S.stacksize )
            return ERROR;
  *S.top++=e;
   return OK;
}
```

取栈顶元素：

```
Status Pop( SqStack &S, SElemType &e)
{ if( S.top == S.base ) // 栈空
        return ERROR;
```

```
  e= *--S.top;
  return OK;
}
```

栈顶判空：

```
bool StackEmpty( SqStack S )
{
  if(S.top == S.base) return true;
   else return false;
}
```

求栈长度：

```
int StackLength( SqStack S )
{
  return  (S.top  -  S.base) ;
}
```

二、链栈

存储结构：

```
typedef  struct StackNode {
       SElemType  data;
       struct StackNode *next;
 } StackNode,  *LinkStack;
```

初始化：

```
void InitStack(LinkStack &S )
{
  S=NULL;
}
```

入栈：

```
StatusPush(LinkStack& S, ElemType e)
   {
      p=new StackNode; //生成新结点 p
      if (!p) return OVERFLOW;
      p->data=e;
      p->next=S;
       S=p;
      return OK;
 }
```

出栈：

```
 Status Pop (LinkStack &S, SElemType
&e)
 { if (S==NULL) return ERROR;
    e = S-> data;
    p = S;
    S =  S-> next;
```

```
    delete p;
    return OK;
 }
```

取栈顶元素：

```
Status SElemType GetTop(LinkStack S ,

SElemType &e)
    {  if (S==NULL) return ERROR;
    e=S->data;
       return OK;
     }
```

判空：

```
Status StackEmpty(LinkStack S)
{
    if (S==NULL) return TRUE;
    else return FALSE;
}
```

队 列

定义：

队列是一种先进先出(FIFO) 的线性表. 它只允许在表的一端进行插入,而在另一端删除元素。

顺序队列：

存储结构：

```
Typedef struct {
    QElemType *base;   //初始化的动态分配存储空间
    int  front;              //头指针
    int  rear;               //尾指针
}SqQueue;
```

空队标志：front= =rear

问题：假溢出

循环队列：

入队：

```
  base[rear]=x;
  rear=(rear+1)%M;
```

出队：

```
  x=base[front];
  front=(front+1)%M;
```

队空：front==rear

队满：(rear+1)%M==front

初始化：

```
Status InitQueue (SqQueue &Q)
{  Q.base =new QElemType[MAXQSIZE]
   if(!Q.base) exit(OVERFLOW);
    Q.front=Q.rear=0;
    return OK;
}
```

入队：

```
Status EnQueue(SqQueue &Q,QElemType e)
  { if((Q.rear+1)%MAXQSIZE==Q.front)
                  return ERROR;
      Q.base[Q.rear]=e;
      Q.rear=(Q.rear+1)%MAXQSIZE;
```

```
    return OK;
  }
```
出队：
```
Status DeQueue (LinkQueue &Q,QElemType
&e)
 { if(Q.front==Q.rear) return ERROR;
   e=Q.base[Q.front];
   Q.front=(Q.front+1)%MAXQSIZE;
   return OK;
}
```
取列首元素：
```
Status   GetQueue   (LinkQueue      Q,
QElemType &e)
 { if(Q.front==Q.rear) return ERROR;
   e=Q.base[Q.front];
   return OK;
}
```
长度：
```
int  QueueLength (SqQueue Q)
{
  return                          (Q.rear-
Q.front+MAXQSIZE)%MAXQSIZE;
```

```
 }
```

二、链队列

存储结构：

```
typedef struct QNode
 { QElemType   data;
    struct Qnode  *next;
 } Qnode, *QueuePtr;

typedef struct {
QueuePtr  front;//队头指针
QueuePtr  rear; //队尾指针
}LinkQueue;
```

初始化：

```
Status InitQueue (LinkQueue &Q)

{  Q.front=(QueuePtr)malloc(sizeof(QNo
de));
     if(!Q.front) exit(OVERFLOW);
     Q.rear=Q.front;
     Q.front->next=NULL;
      return OK;
```

```
}
```

入队：

```
Status EnQueue(LinkQueue &Q, QElemType
e)
 { p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(OVERFLOW);
    p->data=e; p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return OK;
}
```

出队：

```
Status DeQueue (LinkQueue &Q,QElemType
&e)
 { if(Q.front==Q.rear) return ERROR;
   p=Q.front->next;
   e=p->data;
   Q.front->next=p->next;
   if(Q.rear==p) Q.rear=Q.front;
   free(p);
   return OK;
}
```

取队头元素：

```
Status GetHead (LinkQueue Q, QElemType
&e)
  { if(Q.front==Q.rear) return ERROR;
     e=Q.front->next->data;
     return OK;
  }
```

判空：

```
Status QueueEmpty (LinkQueue Q)
  { return (Q.front==Q.rear); }
```

摧毁：

```
Status DestroyQueue (LinkQueue &Q)
   {while(Q.front)
          { Q.rear=Q.front->next;
             free(Q.front);
             Q.front=Q.rear;
          }
   return OK;
}
```

串、数组和广义表

串(String)：

是零个或多个字符组成的有限序列。

串名 串值 串长

空串 空格串

子串：

是“串”中任一个连续的字符子序列.

主串：

包含子串的串相应地称为主串。

字符位置：

是字符在序列中的序号。

子串位置：

是子串的第一个字符在主串中的位置。

串相等：

两串长度相等,且对应位置的字符都相等。

串 的 存 储 结 构：

1. 顺序存储
2. 链式存储

模式匹配算法：

```
typedef struct
 { char *ch;
  int  length;
}SString;
int  Index(Sstring S, Sstring T, int
pos)
 {
i=pos-1;
j=1-1;
while (i<S.length && j <T.length)
      {
        if ( S[ i ]==T[ j ])
           {++i;  ++j; }
         else
           { i=i-j+1;    j=1-1; }
   if ( j==T.length)
      return i-T.length+1;//序号
   else return 0;
}
```

数组

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

行优先存储

列优先存储

矩 阵 的 压 缩 存 储：

- 什么是压缩存储？

若多个数据元素的值都相同，则只分配一个元素值的存储空间，且零元素不占存储空间。

- 什么样的矩阵能够压缩？

一些特殊矩阵，如：对称矩阵，对角矩阵，三角矩阵，稀疏矩阵等。

- 什么叫稀疏矩阵？

矩阵中非零元素的个数较少（一般小于 5%）

1. 对称矩阵
2. 三角矩阵

(a)上三角矩阵　　(b)下三角矩阵

广义表：

n（≥ 0）个表元素组成的有限序列，记作$LS=(a_0, a_1, a_2, \cdots, a_{n-1})$。$LS$是表名，$a_i$是表元素，它可以是表 (称为子表)，可以是数据元素(称为原子)。n为表的长度。n = 0 的广义表为空表。

基本运算：

（1）求表头 GetHead(L)：非空广义表的第一个元素，可以是一个单元素，也可以是一个子表。

（2）求表尾 GetTail(L)：非空广义表除去表头元素以外其它元素所构成的表。表尾一定是一个表。

A=(a,b,(c,d),(e,(f,g)))　　d

GetHead(GetTail(GetHead(GetTail(GetTail(A)))))

树和二叉树

树（tree）：

是n（n≥0)个结点的有限集合，

它或为空树（n=0）或为非空树，对于非空树T：

（1）有且仅有一个称为根的结点（root）；

（2）除了根以外，其余结点可分为m（m≥0)个互不相交的有限集T_1，T_2，…，T_m，其中每个集合本身又是一棵树，并且称为根的子树（SubTree）。

树的基本术语：

结点：

数据元素+若干指向子树的分支。

结点的度：

分支的个数。

树的度：

树中所有结点的度的最大值。

叶子结点：

度为零的结点。

非终端结点（分支结点）：

度大于零的结点。

路径：

由从根到该结点所经分支和结点构成。

孩子结点、双亲结点、

兄弟结点、堂兄弟结点

祖先结点、子孙结点

结点的层次:

假设根结点的层次为 1,第 *l* 层的结点的子树根结点的层次为 *l+1*。

树的深度：

树中叶子结点所在的最大层次

森林：

是 m（m≥0）棵互不相交的树的集合。

二叉树（Binary Tree）:

是 n(n≥0)个结点构成的集合,它或为空树,或是由一个根结点加上两棵分别称为左子树和右子树的互不交的二叉树组成。

二叉树的性质：

性质 1 ： 在二叉树的第 i 层上至多有 2^{i-1} 个结点（i≥1）。

性质 2 ： 深度为 *k* 的二叉树上至多含 2^k-1 个结点（k≥1）。

性质 3 ： 对任何一棵二叉树，若它含有 n_0 个叶子结点、n_2 个度为 2 的结点，则必存在关系式：$n_0 = n_2+1$。

性质 4 ： 具有 *n* 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor +1$。

性质 5 ： 若对含 *n* 个结点的完全二叉树从上到下且从左至右进行 *1* 至 *n* 的编号，则对完全二叉树中任意一个编号为 *i* 的结点：

（1） 若 *i=1*，则该结点是二叉树的根，无双亲，

否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点；

（2） 若 *2i>n*，则该结点无左孩子，

否则，编号为 *2i* 的结点为其左孩子结点；

（3） 若 *2i+1>n*，则该结点无右孩子结点，

否则，编号为 *2i+1* 的结点为其右孩子结点。

两类特殊的二叉树：

1.满二叉树：

指的是深度为 k 且含有 2^k-1 个结点的二叉树。

2.完全二叉树：

树中所含的 n 个结点和满二叉树中编号为 1 至 n 的结点一一对应。

二叉树的顺序存储结构：

```
#define MAXSIZE 100
typedef TElemType SqBiTree [MAXSIZE ];
SqBiTree bt;
```

二叉树的链式存储结构：

```
typedef struct {
    TElemType       data;
    struct BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
```

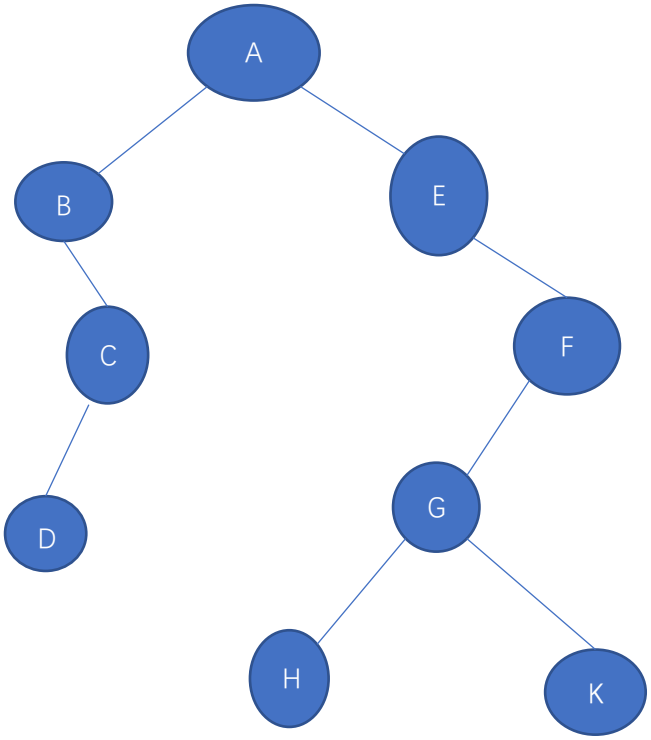
二叉树的遍历：

先序的遍历算法

中序的遍历算法

后序的遍历算法

按层次遍历



先序序列：

A B C D E F G H K

中序序列：

B D C A E H G K F

后序序列：

D C B H K G F E A

算法描述：

先序：

```
void Preorder (BiTree T)
{
   if (T) {
      visit(T->data);
      Preorder(T->lchild);
      Preorder(T->rchild);
   }
}
```

中序：

```
void Inorder (BiTree T)
{
   if (T) {
   Inorder(T->lchild);
   visit(T->data);
   Inorder(T->rchild);
   }
}
```

中序非递归：

```
void Inorder1 (BiTree T)
{ InitStack(S); p=T;
while(1){
while(p) {Push(S,p);p=p->lchild; }
if( StackEmpty(S))  return;
Pop (S, p);
cont << P->data;
p=p->rchild; }
}
```

应用：

1.统计二叉树中叶子结点的个数：

```
void  CountLeaf  (BiTree  T,    int  &
count){
   if ( T ) {
      if ((!T->lchild)&& (!T->rchild))
         count++;
      CountLeaf( T->lchild, count);
      CountLeaf( T->rchild, count);
   }
}
```

2、求二叉树的深度

```
int Depth (BiTree T ){
 if ( !T )     depthval = 0;
 else {
    depthL= Depth( T->lchild );
    depthR=   Depth(   T->rchild   );
    depthval=1+(depthL>depthR?depthL:
    depthR);
    }
 return depthval;
}
```

3、建立二叉树的存储结构

```
void CreateBiTree(BiTree &T)
 {  scanf(( “%c”, &ch);
    if (ch== ‘  ') T = NULL;
    else
      { T = new BiTNode;
         T->data = ch;
         CreateBiTree(T->lchild);
         CreateBiTree(T->rchild);
      }
```

```
}
```

4、查询二叉树中某个结点

```
bool Preorder (BiTree T, ElemType x,
BiTree &p) {
if (T) {
    if (T->data==x) { p = T; return
TRUE;}
else {
      if  (Preorder(T->lchild, x, p))
return TRUE;
else
return(Preorder(T->rchild, x, p)) ;
}
}
else { p = NULL;  return FALSE; }
}
```

线索二叉树：

在中序线索二叉树中，查找结点*p 的中序后继结点

1. 若 P->Rtag 为 1，　则 P 的右线索指向其后继结点*q；

2. 若 P->Rtag 为 0，　则其后继结点*q 是右

子树中的最左结点。

森林和树之间的转换

哈夫曼树与哈夫曼编码

最优二叉树(哈夫曼树)的定义：

结点的路径长度：

从根结点到该结点的路径上分支的数目。

树的带权路径长度定义为：

树中所有叶子结点的带权路径长度之和

$WPL(T) = \Sigma w_k l_k$ (对所有叶子结点)

最优二叉树(哈夫曼树,Huffman Tree)

在所有含 n 个叶子结点、并带相同权值的二叉树中，必存在一棵其带权路径长度取最

小值的二叉树，称为“最优二叉树”

构造最优二叉树算法(哈夫曼算法)

哈夫曼编码：左 0 右 1 原则

图

图（Graph)的定义：

图G由两个集合V和E组成,记为G=(V,E)，其中， V 是顶点(数据元素)的有穷非空集合，E是V中顶点偶对的有穷集合，这些顶点偶对称为边 。

无向图：每条边都是无方向的图。

有向图：每条边都是有方向的图。

图的基本术语：

1） 子图：

设有两个图G=(V,E) 和G′=(V′,E′)， 且 V′⊆V, E′⊆E, 则称 G′ 为 G 的子图。

2） 无向完全图和有向完全图

3） 稀疏图和稠密图：

边或弧的个数 e<nlogn 的图称为稀疏图,否则称作稠密图。

4） 权和网：

图中的边可标上具有某种含义的数值，该数值称为边上的权。权可表示为从一个顶点到另一顶点的距离或耗费。这种边带权的图称网（网络）。

5）邻接点：

在无向图 G 中，假若边(v ， w) ∈G，则称顶点 v 和 w 互为邻接点。而边(v ， w) 与顶点 v 和 w 相关联。

6）度、入度和出度：

顶点 v 的度：是和顶点 v 关联的边的数目。记为 TD(v)。

入度：以顶点 v 为头的弧的数目。记为 ID(v)。

出度：以顶点 v 为尾的弧的数目。记为 D(v)。

7）路径和路径长度：

若从顶点 u 到顶点 w 之间存在一条路径。路径上边或弧的数目称作路径长度。

8）回路或环：

第一个顶点和最后一个顶点相同的路径称为回路或环。

9）简单路径、简单回路：

简单路径:指序列中顶点不重复出现的路径。

简单回路:指序列中第一个顶点和最后一个顶点相同的简单路径。

10）连通、连通图、连通分量：

在无向图 G 中，如果顶点 v 到顶点 w 有路径，则称 v 和 w 是连通的。
在无向图 G 中，如果任意两个顶点之间都有路径，则称此图为连通图。
若无向图为非连通图，则图中各个极大连通子图称作此图的连通分量。

11）强连通图、强连通分量：

对有向图， 若任意两个顶点之间都存在一条有向路径，则称此有向图为强连通图。否则，其各个强连通子图称作它的强连通分量。

12）连通图的生成树：

假设一个连通图有 n 个顶点和 e 条边，其中 n-1 条边和 n 个顶点构成一个极小连通子图，称该极小连通子图为此连通图的生成树。
对非连通图，称由各个连通分量的生成树的集合为此非连通图的生成森林。

13)有向树和生成森林：

有一个顶点的入度为 0，其余顶点的入度均为 1 的有向图称为有向树。

一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，但只有足以构

成若干棵有向树的弧。

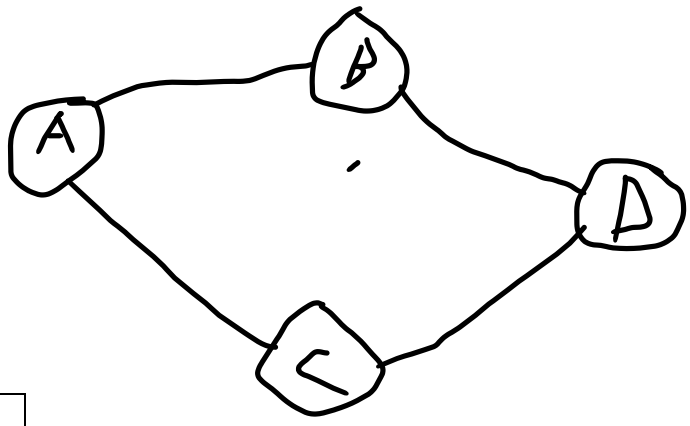
图的存储表示

邻接矩阵：

存储结构：

```
typedef struct
{vextype vexs[n];
 int arcs[n][n];
 int vexnum,arcnum;
}AMGraph;
```

A
B
C
D



创建无向邻接矩阵算法：

```
CREATGRAPH(AMGraph &G)
{ cin>>G.vexnum>>G.arcnum;
  for(i=0; i< G.vexnum ; i++)
       cin>> G.vexs[i];
  for(i=0; i< G.vexnum ; i++)
     for(j=0; j< G.vexnum ; j++)
          G.arcs[i][j]=MaxInt;
  for(k=0; k< G.arcnum ; k++)
     { cin>>i>> j>>w;
       G.arcs[i][j]=w;
          G.arcs[j][i]=w;
     }}
```

邻接表：

边结点结构：

```
typedef struct ArcNode {
   int  adjvex;
   struct ArcNode  *nextarc;
} ArcNode;
```

顶点的结点结构：

```
typedef struct VNode {
   VertexType  data;
```

```
  ArcNode  *firstarc;
} VNode, AdjList[MVNum];
```

存储结构：

```
typedef struct
{ AdjList vertices;
    int vexnum,arcnum;
} ALGraph;
```

建邻接表的算法：

```
CREATADJLIST(ALGraph &G)
 { cin>>G.vexnum>>G.arcnum;
    for (i=0; i<G.vexnum; i++)
     {cin>>G.vertices[i].data;
       G.vertices[i]. firstarc =NULL; }
for (k=0; k<G.arcnum; k++)
   {cin>>i, j;
     s=new ArcNode;
     s->adjvex=j;
     s->nextarc=G.vertices[i].firstarc;
     G.vertices[i].firstarc=s;
     s=new ArcNode;
     s->adjvex=i;
     s->nextarc=G.vertices[j].firstarc;
```

```
        G.vertices[j].firstarc=s;
        }
 }
```

图的遍历

1.深度优先搜索

从图中某个顶点 V_0 出发，访问此顶点，然后依次从 V_0 的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和 V_0 有路径相通的顶点都被访问到。

算法：（基于邻接表）

```
int visited[n];//初值为 FALSE
void DFS(ALGraph G, int i)
{cout<<G.vertices[i].data;
  visited[i]=TRUE;
  p= G.vertices[i].firstarc;
  while(p!=NULL)
    {w=p->adjvex;
       if (!visited[w])  DFS(G, w);
      p=p->nextarc; }
}
```

广度优先搜索：

从图中的某个顶点 V_0 出发，并在访问此顶点

之后依次访问 V_0 的所有未被访问过的邻接点，之后按这些顶点被访问的先后次序依次访问它们的邻接点，直至图中所有和 V_0 有路径相通的顶点都被访问到。若此时图中尚有顶点未被访问,则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

算法：（基于邻接表）

```
void BFS(ALGraph G, int v)
{ InitQueue (Q) ;
  cout<<G.vertice[v].data;
  visited[v]=TRUE;
  EnQueue(Q,v);
  while(! QueueEmpty(Q))
    {DeQueue (Q, i);
     p= G.vertices[i].firstarc;
     while(p!=NULL)
      { w= p->adjvex;
        if(! visited[w])
         {cout<<G.vertices[w].data);
          visited[w]=TRUE;
          ENQUEUE(Q,w);}
```

```
            p=p->next;
        }
    }
}
```

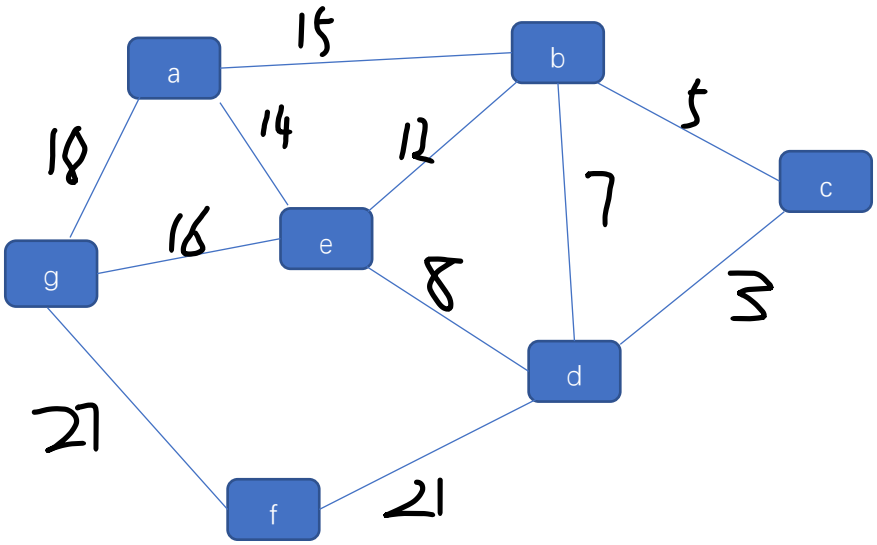
最小生成树：

普里姆算法(Prim)

(1) 取图中任意一个顶点 v 作为生成树的根。

(2) 选择一个顶点在生树中，另一个顶点不在生成树中的边中权最小的边(v,w)，往生成树上 添加新顶点 w 及边(v,w) 。

(3) 继续执行(2)，直至生成树上含有 n-1 个边为止。



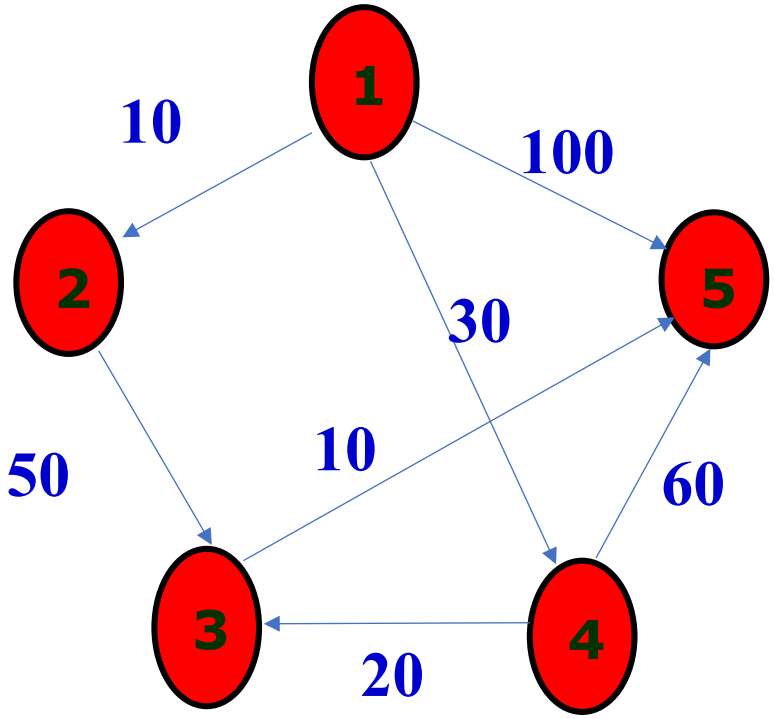
PRIM（AMGraph G）

```
{for(j=1;j<G.vexnum;j++)
   {T[j-1].formvex=1;
     T[j-1].endvex=j+1;
     T[j-1].length=G.arcs[0][j]; }
 for (k=0; k<G.vexnum-1; k++)
   { min=max;
     for (j=k; j<G.vexnum-1; j++)
         if(T[j].length<min)
            {min=T[j].length;  m=j; }
     e=T[m]; T[m]=T[k]; T[k]=e;
     v=T[k].endvex;
    for (j=k+1; j<G.vexnum-1; j++)
      {d=G.arcs[v-1][T[j].endvex-1];
           if (d<T[j].length)
               {T[j].length=d;
                 T[j].fromvex=v;}
        }
   }
}
```

克鲁斯卡尔(Kruskal)算法

求某源点到其余各顶点的最短路径问题

迪杰斯特拉（Dijkstra)算法：



算法：

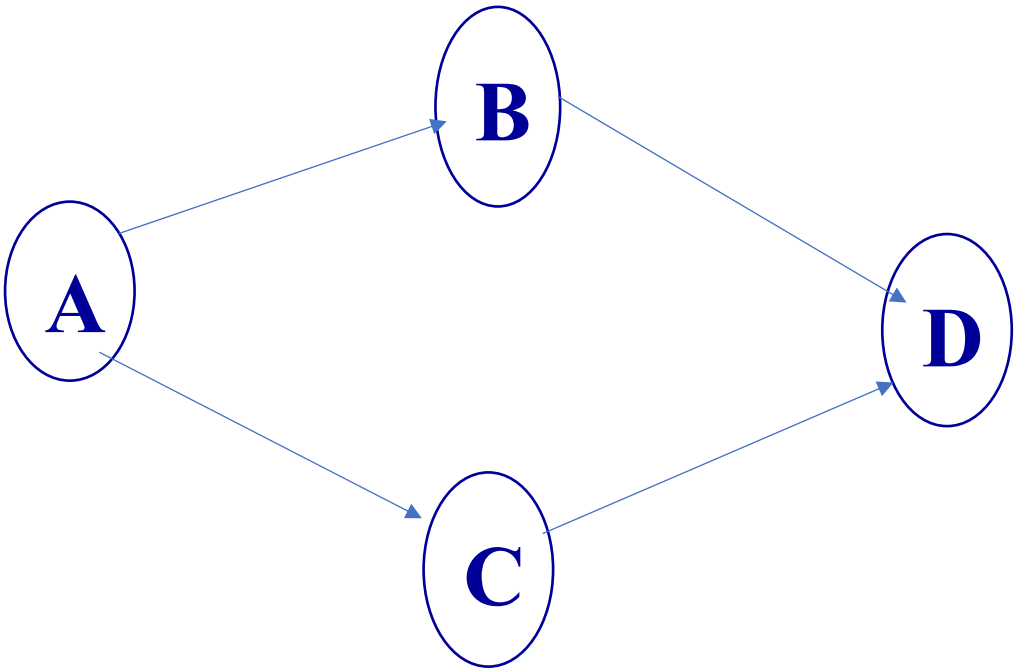
```
float Dist[n];
int p[n], s[n];
DIJKSTRA(AMGraph G, int v)
{v1= v-1;
 for (i=0; i<G.vexnum; i++)
  {Dist[i]=G.arcs[v1][i];
    s[i]=0;
    if (Dist[i]!=max) p[i]=v;
```

```
    else                                p[i]=0;
    }
s[v1]=1; Dist[v1]=0;
for (i=0; i<G.vexnum-1; i++)
   {min=inf; /* 先令 min=∞ */
     for (j=0; j<G.vexnum; j++)
 if ((!s[j] )&&(Dist[j]<min))
     {min=Dist[j]; k=j;}
S[k]=1;
for (j=0; j<G.vexnum; j++)
    if((!s[j]) &&(Dist[j]>Dist[k]+G.arcs[k][j]))
         { Dist[j]=Dist[k]+G.arcs[k][j];
           p[j]=k+1;
         }
  }
}
```

拓 扑 排 序

对于下列有向图 G 可求得拓扑序列：

A B C D 或 A C B D



查找

查找表：

查找表是由同一类型的数据元素(或记录)构成的集合。

关键字：

是数据元素（或记录）中某个数据项的值，

用以标识（识别）一个数据元素（或记录）。

查找：

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或（记录）。

平均查找长度：

关键字的平均比较次数，也称平均搜索长度。

1. 顺序查找

数据元素类型的定义：

```
typedef struct {
    keyType key;      // 关键字域
      … …             // 其它属性域
} ElemType ;
```

存储结构：

```
typedef  struct {
 ElemType *R;// 数据元素存储空间基址，建表时按实际长度分配
int length;     // 表的长度
…
} SSTable;
```

算法：

```
int LocateELem(SSTable ST, KeyType key)
{  for (i=1;i< =ST.length; i++)
       if (ST.R[i].key==key)
                return i;
  return 0;
}
```

改进算法：（“哨兵”）

```
int Search_Seq(SSTable ST, KeyType key)
{ ST.R[0].key = key;    // 设置“哨兵”
 for(i=ST.length;ST.R[i].key!=key;--i)
// 从后往前找
return i;          // 找不到时，i 为 0
}
```

平均查找长度:(n+1)/2;

2.折半查找

- low 指示查找区间的下界;
- high 指示查找区间的上界;
- mid = (low+high)/2。

算法：

```
int Search_Bin ( SSTable ST, KeyType
key ) {
   low = 1;  high = ST.length;
   while (low <= high) {
      mid = (low + high) / 2;
      if  (key == ST.R[mid].key )
          return  mid;
     else if( key < ST.elem[mid].key) )
        high = mid - 1;
     else  low = mid + 1;
   }
   return 0;
} // Search_Bin
```

平均查找长度:log2(n+1)-1

3.索引顺序表(分块查找)

在建立顺序表的同时，建立一个索引。

```
typedef  struct {
    KeyType maxkey;
    int              stadr;
}indexItem;  // 索引项
```

```
typedef  struct {
     indexItem *elem;
     int              length;
}indexTable;     // 索引表
```

4.二叉排序树（二叉查找树）

定义：

二叉排序树(Binary Sort Tree)或者是一棵空树；或者是具有下列性质的二叉树：

（1）若它的左子树不空，则左子树上所有结点的值均小于根结点的值；

（2）若它的右子树不空，则右子树上所有结点的值均大于根结点的值；

（3）它的左、右子树也都分别是二叉排序树。

存储结构：

```
typedef struct {
     keyType key;     // 关键字域
        … …           // 其它属性域
} ElemType ;

typedef struct BiTNode { // 结点结构
   ElemType      data;
```

```
    struct BiTNode   *lchild, *rchild;
// 左右孩子指针
} BSTNode, *BSTree;
```

1. 二叉排序树的查找：

```
BSTree SearchBST(BSTree T, KeyType key)
{   if (!T)   return T;
    else  if ( key==T->data.key) )
        return T;
    else  if ( key<T->data.key )
     return SearchBST(T->lchild,key );
    else
      return   SearchBST   (T->rchild,
key );}
```

2. 二叉排序树的插入

```
void Insert BST(BSTree &T, ElemType e )
{ if (!T)
  {
        S =   new  BSTNode;
        S->data = e;
        S->lchild = S->rchild = NULL;
        T=S;
```

```
  }
else  if ( e.key==T->data.key) return ;
else  if ( e.key<T->data.key)
            Insert BST(T->lchild ,e);
//将*S 插入左子树
else  if ( e.key>T->data.key)
            Insert BST(T->rchild ,e);
}
```

3.二叉排序树的创建

```
void CreatBST(BSTree &T)
{ T=NULL;
   cin>>e;
   while(e.key!=ENDFLAG)
        { InsertBST(T,e); cin>>e; }}
```

4.二叉排序树的删除

三种情况:

1）被删除的结点是叶子;

2）被删除的结点只有左子树或者只有右子树;

3)被删除的结点既有左子树,也有右子树。

```
void DeleteBST ( BSTree &T,KeyType key )
{p=T; f=NULL;
```

```
  while(p)
     {if(p.data.key==key) break;
       f=p;
       if(p.data.key>key) p=p->lchild;
       else                              p=p->rchild;
      } //定位：*p 为被删除结点
  if(!p) return ;//找不到被删结点
  if (p->lchild && p->rchild)
      {q=p; s=p->lchild;
        while (s->rchild)
            {q=s; s=s->rchild;}
        p->data=s->data;
        if(q!=p) q->rchild=s->lchild;
        else q->lchild=s->lchild;
        delete s;
       }
else
    {if (!p->rchild)
           {q=p; p=p->lchild;}
                    //用 q 记住要删结点,p 指向要代替被删结点的结点
       else if (!p->lchild)
                  {q=p; p=p->rchild;}
       if (!f)   T=p; //被删结点为根结点
       else if (q==f->lchild) f->lchild=p;
       else f->rchild=p;
      delete q;
    }
}
```

5. 散列表的查找

散列查找法的基本思想：

通过对元素的关键字进行某种运算，直接求出元素的地址，　　即使用关键字到地址的直接转换法，而不需反复比较。

（1）散列函数与散列地址

若令关键字为 key 的记录在表中的存储位置 p= H(key) ，则称这个函数 H 为散

列函数，p 为散列地址。

（2）散列表

通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

（3）冲突与同义词：

key1≠ key2，而 H(key1) = H(key2)的现象称为“冲突”现象 。

构造散列函数的方法 ：

最常用(常考)：除留余数法

设定散列函数为：H(key) = key MOD p

例如： 给定一组关键字为：12，39，18，24，33，21，

若取 p=9，则他们对应的散列函数值将为：3，3，0，6，6，3。

处理冲突的方法 ：

开放地址法中的线性探测再散列：

排序

排序(Sorting)

就是整理文件中的记录，使得它按关键字递增（或递减）的次序排列起来。

排序的稳定性

如果待排序文件中，存在有多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，则称这种

排序方法是稳定的；反之，则称这种排序方法是不稳定的。

内排序与外排序：

若在排序过程中，整个文件都是放在内存中处理，排序时不涉及数据的内、外存交换，则称之为内部排序（简称内排序);反之，若排序过程中要进行数据的内、外存交换，则称之为外部排序。

一、插入排序

插入排序的基本思想：

每次将一个待排序的记录按其关键字大小插入到前面已经排好序的文件中的适当位置，直到全部记录插入完为止。

1. 直接插入排序

```
typedef struct
{ int key;
   datatype other;
} rectype;
typedef struct
{Rectype r[n+1];
 int length;
} SqList;
INSERTSORT(SqList &L)
{for(i=2;i<=L.lenth; i++)
   {L.r[0]=L.r[i];
     j=i-1;
    while(L.r[0].key<L.r[j].key)
              L.r[j+1]=L.r[j--];
    L.r[j+1]=L.r[0];
     }
```

```
    }
```

时间复杂度：T(n)= O(n^2)

稳定性：稳定

2. 希尔排序

又称缩小增量排序。

算法：

```
void ShellInsert(SqList &L, int dk)
{for(i=dk+1;i<=L.length; i++)
     {L.r[0]=L.r[i];
       j=i-dk;
       while(j>0&&L.r[0].key<L.r[j].key;
             {L.r[j+dk]=L.r[j];
               j-=dk; }
L.r[j+dk]=L.r[0];
     }
}
void ShellSort(SqList &L,int dt[],int t)
{for(k=0;k<t;k++)
     ShellInsert(L, dt[k]);
}
```

不稳定

二、交换排序

交换排序的基本思想：

两两比较待排序记录的关键字，发现这两个记录是逆序时则进行交换，直到全部记录无逆序为止。

1.冒泡排序

算法

```
BUBBLESORT(SqList &L)
{ int i,j, noswap;
for (i=1; i<n; i++)
    {noswap=TRUE;
      for (j=L.length-1; j>=i; j--)
          if (L.r[j+1].key<L.r[j].key)
           { L.r[0]=L.r[j+1];
                  L.r[j+1]=L.r[j];
                  L.r[j]=L.r[0];
             noswap=FALSE;
               }
      if (noswap) break;
     }
}
```

稳定

$T(n)= O(n^2)$

2.快速排序

算法：

```
int PATITION (L, /, h)
{L.r[0]=L.r[/];
while(/<h)
{   while((L.r[h].key>=
              L[0].key) &&(/<h))
            h--;
     if(/<h) L.r[l++]=L.r[h];
     while((L.r[/].key<=
              L.r[0].key) &&(/<h))
            /++;
     if(/<h) L.r[h--]=L.r[/];
 }
 L.r[/]=L.r[0];
 return /;
}
QUICKSORT(&L, /, h)
{ if(/<h)
   { i=PARTITION(L,l,h);
     QUICKSORT(L, /, i-1);
```

```
      QUICKSORT(L, i+1, h);
    }
}
```

不稳定

T(n)= O(n log $_2$n)

三、选择排序

1.直接选择排序

算法：

```
SELECTSORT(SeList &L)
{ for(i=1;i<L.length; i++)
    {k=i;
     for (j=i+1;j<=Length; j++)
       if(L.r[j].key<L.r[k].key)
            k=j;
        if ( k!=i)
             {L.r[0]=L.r[i];
                L.r[i]=L.r[k];
               L.r[k]=L.r[0];
             }
     }
}
```

T(n)=O(n^2)

不稳定

2.堆排序

大根堆 小根堆

算法：

```
SIFT(SqList &L, int s, int m)
{     L.r[0]=L.r[s];
      for(j=2*s; j<=m; j*=2)
     {
            if ((j<m)&&
                  (L.r[j].key<L.r[j+1].key))
                   j++;
             if (L.r[0].key<L.r[j].key)
            {
                   L.r[s]=L.r[j];
                   s=j;
             }
             else break;
       }
       L.r[s]=L.r[0];
 }
HEAPSORT (SqList &L)
```

```
{ n=L.length;
  for (i=n/2; i>0; i--)
        SIFT(L, i, n);
  for (i=n; i>1; i--)
       { L.r[0]=L.r[1]; L.r[1]=L.r[i];
         L.r[i]=L.r[0];
          SIFT(L, 1, i-1);
        }
}
```

不稳定

$T(n) = O(n \log n)$

四、归并排序

二路归并：

算法：

```
MERGE(RecType R[ ], RecType T[ ], int low, int mid, int high)
{i=low; j=mid+1; k=low;
  while ((i<=mid) && (j<=high))
          if (R[i].key <= R[j].key)
                  T[k++]=R[i++];
```

```
          else   T[k++]=R[j++];
   while (i<=mid)
          T[k++]=R[i++];
   while (j<=high)
          T[k++]=R[j++];
}
MERGEPASS(RecType R[ ], RecType T[ ], int length, int n)
{ int i=1, j;
 while((i+2*length-1<=n){
       MERGE(R,T,i,i+length-1,i+2*length-1);
       i=i+ 2*length;
    }
 if(i+length-1<n)
       MERGE(R, T, i, i+length-1,     n);
 else
    for(j=i; j<=n; j++)
              T[j]=R[j];
}
MERGESORT(RecType R[ ],int n)
{int length=1;
  while (length<n)
  {MERGEPASS(R, R1,length,n);
```

```
    length = 2*length;
    MERGEPASS(R1,R,length,n);
    length = 2*length;
  }
}
```

稳定

T(n)=O(nlogn)

五、基 数 排 序

1.多关键字的排序

例子：

无序序列：

|3,2,30|1,2,15|3,1,20|2,3,18|2,1,20|

2. 链式基数排序

例子：p→369→367→167→239→237→138→230→139