基本概念

- **数据**

数据是信息的载体，在计算机科学中是指所有能**输入**到计算机中并能被计算机程序识别和**处理**的符号集合。

- **数据元素**

数据元素也称为结点，是表示数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。

- **数据项**

数据项是构成数据元素的不可分割的最小单位。

- **数据对象**

数据对象是具有相同性质的数据元素的集合，是数据的子集。

注意：在不产生混淆的情况下，将数据对象简称为数据。

- **数据结构**

数据结构是指相互之间存在一定关系的数据元素的集合，即数据结构是一个二元组 DataStructure = (D, R)，其中 D 是数据元素的集合，R 是 D 上关系的集合。按照视点的不同，数据结构分为逻辑结构和存储结构。

- **数据的逻辑结构**

数据的逻辑结构是指数据元素之间逻辑关系的整体。根据数据元素之间逻辑关系的不同，数据结构分为四类：

⑴ 集合：数据元素之间就是“属于同一个集合”，除此之外，没有任何关系；

⑵ 线性结构：数据元素之间存在着一对一的线性关系；

⑶ 树结构：数据元素之间存在着一对多的层次关系；

⑷ 图结构：数据元素之间存在着多对多的任意关系。

注意：数据结构分为两类：线性结构和非线性结构。

- **数据的存储结构**

数据的存储结构又称为物理结构，是数据及其逻辑结构在计算机中的表示。通常有两种存储结构：顺序存储结构和链接存储结构。

顺序存储结构的基本思想是：用一组**连续**的存储单元**依次**存储数据元素，数据元素之间的逻辑关系是由元素的存储位置来表示的。

链接存储结构的基本思想是：用一组**任意**的存储单元存储数据元素，数据元素之间的逻辑关系是用指针来表示的。

注意：存储结构除了存储数据元素之外，必须存储数据元素之间的逻辑关系。

- **抽象数据类型**

抽象数据类型是一个数据结构以及定义在该结构上的一组操作的总称。抽象数据类型提供了使用和实现两个不同的视图，实现了封装和信息隐藏。

- **算法的定义**

通俗地讲，算法是解决问题的方法，严格地说，算法是对特定问题求解步骤的一种描述，是指令的有限序列。

- **算法的特性**

⑴ 输入：一个算法有零个或多个输入（即算法可以没有输入），这些输入通常取自于某个特定的对象集合。

⑵ 输出：一个算法有一个或多个输出（即算法必须要有输出），通常输出与输入之间有着某种特定的关系。

⑶ 有穷性：一个算法必须总是（对任何合法的输入）在执行有穷步之后结束，且每一步都在有穷时间内完成。

⑷ 确定性：算法中的每一条指令必须有确切的含义，不存在二义性。并且，在任何条件下，对于相同的输入只能得到相同的输出。

⑸ 可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

- **线性表的定义**

线性表简称表，是零个或多个具有相同类型的数据元素的有限序列。数据元素的个数称为线性表的**长度**，长度等于零时称为空表。

- **线性表的逻辑关系**

在一个非空表 $L=(a_1, a_2, \cdots\cdots, a_n)$中，任意一对相邻的数据元素 a_{i-1} 和 a_i 之间（$1<i\leqslant n$）存在序偶关系（a_{i-1}，a_i），且 a_{i-1} 称为 a_i 的前驱，a_i 称为 a_{i-1} 的后继。在这个序列中，a_1 无前驱，a_n 无后继，其它每个元素有且仅有一个前驱和一个后继。

- **顺序表的存储结构定义**

用 MaxSize 表示数组的长度，顺序表的存储结构定义如下：

```
#define MaxSize 100
typedef struct
{
   ElemType data[MaxSize];   // ElemType 表示不确定的数据类型
   int length;               //length 表示线性表的长度
} SeqList;
```

- **顺序表是随机存取结构**

设顺序表的每个元素占用 c 个存储单元，则第 i 个元素的存储地址为：

$$LOC(a_i) = LOC(a_1) + (i-1) \times c$$

- **顺序表的优缺点**

顺序表利用了数组元素在物理位置上的邻接关系来表示线性表中数据元素之间的逻辑关系，这使得顺序表具有下列优点：

⑴ 无需为表示表中元素之间的逻辑关系而增加额外的存储空间；

⑵ 可以快速地存取表中任一位置的元素（即随机存取）。

同时，顺序表也具有下列缺点：

⑴ 插入和删除操作需移动大量元素。在顺序表上做插入和删除操作，等概率情况下，平均要移动表中一半的元素。

⑵ 表的容量难以确定。由于数组的长度必须事先确定，因此，当线性表的长度变化较大时，难以确定合适的存储规模。

⑶ 造成存储空间的“碎片”。数组要求占用连续的存储空间，即使存储单元数超过所需的数目，如果不连续也不能使用，造成存储空间的“碎片”现象。

- **单链表的存储结构定义**

单链表的存储结构定义如下：

```
Struct    Node
 {     ElemType data;   // ElemType 表示不确定的数据类型
        struct Node *next;
 } *first;                //first 为单链表的头指针
```

- **双链表的存储结构定义**

双链表存储结构定义如下：

```
struct DulNode
{
   ElemType data;   // ElemType 表示不确定的数据类型
   struct DulNode *prior, *next;      // prior 为前驱指针域，next 为后继指针域
} *first;                  //first 表示双链表的头指针
```

- **栈的定义**

栈是限定仅在表尾进行插入和删除操作的线性表。允许插入和删除的一端称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈。

- **栈的操作特性**

栈的操作具有**后进先出**的特性。

- **队列的定义**

队列是只允许在一端进行插入操作，而另一端进行删除操作的线性表。允许插入的一端称为队尾，允许删除的一端称为队头。

- **队列的操作特性**

队列的操作具有**先进先出**的特性。

- **循环队列中解决队空队满的判断条件**

方法一：附设一个存储队列中元素个数的变量 num，当 num=0 时队空，当 num=QueueSize 时为队满；

方法二：修改队满条件，浪费一个元素空间，队满时数组中只有一个空闲单元；即队空的条件是 front=rear，队满的条件是(rear+1) % QueueSize=front，队列长度为(rear−front+QueueSize) % QueueSize。

方法三：设置标志 flag，当 front=rear 且 flag=0 时为队空，当 front=rear 且 flag=1 时为队满。

- **串的定义**

串是零个或多个字符组成的有限序列。

- **空格串和空串的定义**

只包含空格的串称为**空格串**。串中所包含的字符个数称为串的长度，长度为 0 的串称**空串**，记作 ""。

- **串的比较**

串的比较是通过组成串的字符之间的比较来进行的。

给定两个串：

X="$x_1x_2\cdots x_n$"

Y="$y_1y_2\cdots y_m$"

则当 $n=m$ 且 $x_1=y_1$，…，$x_n=y_m$ 时，称 $X=Y$；

当下列条件之一成立时，称 $X<Y$：

⑴ $n<m$，且 $x_i=y_i$（i=1，2，…，n）；

⑵ 存在某个 $k\leqslant\min(m,n)$，使得 $x_i=y_i$（i=1，2，…，k−1），$x_k<y_k$。

- **改进的模式匹配算法中 next[j]的求法**

用 next[j]表示 t_j 对应的 k 值（$1\leqslant j\leqslant m$），其定义如下：

$$\text{next}[j]=\begin{cases}0 & j=1\\ \max\{k \mid 1\leqslant k<j \text{ 且 } "t_1t_2\ldots t_{k-1}" = "t_{j-k+1}t_{j-k+2}\ldots t_{j-1}"\} & \\ 1 & \text{其它情况}\end{cases}$$

- **数组的基本操作**

数组是一个具有固定格式和数量的数据集合，在数组上一般不能做插入、删除元素的操作。因此，在数组中通常只有两种操作：

⑴ 读取：给定一组下标，读取相应的数组元素；

⑵ 修改：给定一组下标，存储或修改相应的数组元素。

- **二维数组的寻址**

按行优先，设二维数组的行下标与列下标的范围分别为［l_1，h_1］与［l_2，h_2］，则任一元素 a_{ij} 的存储地址可由下式确定：

$$\mathrm{LOC}(a_{ij})=\mathrm{LOC}(a_{l_1l_2})+((i-l_1)\times(h_2-l_2+1)+(j-l_2))\times c$$

- **特殊矩阵的定义**

特殊矩阵是指矩阵中有很多值相同的元素并且它们的分布有一定的规律。

- **矩阵压缩存储的基本思想**

压缩存储的基本思想是：⑴ 为多个值相同的元素只分配一个存储空间；⑵ 对零元素不分配存储空间。

- **对称矩阵的压缩存储中：**下三角元素 a_{ij}（$i\geqslant j$）在一个数组 SA 中的下标为：$k=i\times(i-1)/2+j-1$。上三角中的元素 a_{ij}（$i<j$），则访问和它对应的下三角中的元素 a_{ji} 即可，即：$k=j\times(j-1)/2+i-1$。
- **三角矩阵的压缩存储中：**下三角矩阵中任一元素 a_{ij} 在一个数组 SA 中的下标 k 与 i、j 的对应关系为：

$$k=\begin{cases} i\times(i-1)/2+j-1 & \text{当 } i\geqslant j \\ n\times(n+1)/2 & \text{当 } i<j \end{cases}$$

上三角矩阵元素 a_{ij} 在 SA 中的下标为：$k=(i-1)\times(2n-i+2)/2+(j-i)$。

- **稀疏矩阵的压缩存储方式**

三元组顺序表和十字链表

- **三元组的定义**

```
struct element
 {
   int    row, col;
   ElemType    item
 };
```

- **广义表的定义**

广义表是 n（$n\geqslant 0$）个数据元素的有限序列。

- **表头**

当广义表 *LS* 非空时，称第一个元素为 *LS* 的表头；

- **表尾**

称广义表 *LS* 中除去表头后其余元素组成的广义表为 *LS* 的。

- **长度**

广义表 *LS* 中的直接元素的个数称为 *LS* 的**长度**；

- **深度**

广义表 *LS* 中括号的最大嵌套层数称为 *LS* 的**深度**。

- **树的定义**

树是 n（$n\geqslant 0$）个结点的有限集合。当 $n=0$ 时，称为空树；任意一棵非空树满足以下条件：

⑴ 有且仅有一个特定的称为**根**的结点；

⑵ 当 $n>1$ 时，除根结点之外的其余结点被分成 m（$m>0$）个**互不相交**的有限集合 T_1，T_2，…，T_m，其中每个集合又是一棵树，并称为这个根结点的子树。

- **结点的度、树的度**

某结点所拥有的子树的个数称为该结点的度；树中各结点度的最大值称为该树的度。

- **叶子结点、分支结点**

度为 0 的结点称为叶子结点，也称为终端结点；度不为 0 的结点称为分支结点，也称为非终端结点。

- **孩子结点、双亲结点、兄弟结点**

某结点的子树的根结点称为该结点的孩子结点；反之，该结点称为其孩子结点的双亲

- **路径、路径长度**

如果树的结点序列 $n_1, n_2, \cdots, n_k$ 满足如下关系：结点 n_i 是结点 n_{i+1} 的双亲（$1 \leqslant i < k$），则把 $n_1, n_2, \cdots, n_k$ 称为一条由 n_1 至 n_k 的路径；路径上经过的边的个数称为路径长度。

- **祖先、子孙**

如果从结点 x 到结点 y 有一条路径，那么 x 就称为 y 的祖先，而 y 称为 x 的子孙。

注意：某结点子树中的任一结点都是该结点的子孙。

- **结点的层数、树的深度（高度）**

规定根结点的层数为 1，对其余任何结点，若某结点在第 k 层，则其孩子结点在第 $k+1$ 层；树中所有结点的最大层数称为树的深度，也称为树的高度。

- **二叉树的定义**

二叉树是 n（$n \geqslant 0$）个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

- **二叉树的特点**

二叉树的特点是：⑴ 每个结点最多有两棵子树，所以二叉树中不存在度大于 2 的结点；⑵ 子树的次序不能任意颠倒，某结点即使只有一棵子树也要区分是左子树还是右子树。

注意：二叉树和树是两种树结构。

- **二叉树的基本形态**

二叉树具有五种基本形态：⑴ 空二叉树；⑵ 只有一个根结点；⑶ 根结点只有左子树；⑷ 根结点只有右子树；⑸ 根结点既有左子树又有右子树。

- **斜树**

所有结点都只有左子树的二叉树称为左斜树；所有结点都只有右子树的二叉树称为右斜树；左斜树和右斜树统称为斜树。

斜树的特点：①每一层只有一个结点，即只有度为 1 和度为 0 的结点并且只有一个叶子结点；② 斜树的结点个数与其深度相同。

- **满二叉树**

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

满二叉树的特点：① 叶子结点都在最下一层；② 只有度为 0 和度为 2 的结点。

- **完全二叉树**

对一棵具有 n 个结点的二叉树按层序编号，如果编号为 i（$1 \leqslant i \leqslant n$）的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。

完全二叉树的特点是：① 叶子结点只能出现在最下两层，且最下层的叶子结点都集中在左面连续的位置；② 如果有度为 1 的结点，只可能有一个，且该结点只有左孩子。

- **二叉树的基本性质**

性质 1　二叉树的第 i 层上最多有 2^{i-1} 个结点（$i \geqslant 1$）。

性质 2　在一棵深度为 k 的二叉树中，最多有 2^k-1 个结点，最少有 k 个结点。

性质 3　在一棵二叉树中，如果叶子结点的个数为 n_0，度为 2 的结点个数为 n_2，则

$$n_0 = n_2 + 1。$$

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　对一棵具有 n 个结点的完全二叉树中的结点从 1 开始按层序编号，则对于任意的编号为 i（$1 \leqslant i \leqslant n$）的结点（简称为结点 i），有：

⑴ 如果 $i>1$，则结点 i 的双亲的编号为 $\lfloor i/2 \rfloor$；否则结点 i 是根结点，无双亲；

⑵ 如果 $2i \leqslant n$，则结点 i 的左孩子的编号为 $2i$；否则结点 i 无左孩子；

⑶ 如果 $2i+1 \leqslant n$，则结点 i 的右孩子的编号为 $2i+1$；否则结点 i 无右孩子。

- **二叉树的存储**

包括：二叉树的顺序存储和二叉树的链式存储。

二叉链表的存储结构定义如下：

```
struct BiNode
{
   ElemType data;
   BiNode *lchild, *rchild;
} *root;                  //root 表示二叉链表的头指针

struct TriNode
{
   ElemType data;
   TriNode *lchild, *rchild, *parent;     // parent 指向该结点的双亲
} *root;                  //三叉链表的头指针
```

- **遍历的含义**

所谓遍历就是无重复无遗漏地访问。二叉树的遍历是指从根结点出发，按照某种次序访问二叉树中的所有结点，使得每个结点被访问一次且仅被访问一次。

- **二叉树的遍历次序定义**

前序遍历（或称前根遍历、先序遍历）

若二叉树为空，则空操作返回；否则

⑴ 访问根结点；

⑵ 前序遍历根结点的左子树；

⑶ 前序遍历根结点的右子树。

中序遍历（或称中根遍历）

若二叉树为空，则空操作返回；否则

⑴ 中序遍历根结点的左子树；

⑵ 访问根结点；

⑶ 中序遍历根结点的右子树。

后序遍历（或称后根遍历）

若二叉树为空，则空操作返回；否则

⑴ 后序遍历根结点的左子树；

⑵ 后序遍历根结点的右子树；

⑶ 访问根结点。

层序遍历

二叉树的层序遍历是从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

- **线索二叉树的定义**

在一个具有 *n* 个结点的二叉链表中，利用 *n*+1 个空指针域存放指向该结点在某种遍历序列中的前驱和后继结点的指针，这些指向前驱和后继结点的指针称为**线索**，加上线索的二叉树称为**线索二叉树**，相应地，加上线索的二叉链表称为**线索链表**。

- **线索二叉树的存储结构定义**

线索链表中的结点定义如下：

```
enum flag {Child, Thread};     //枚举类型，枚举常量 Child=0，Thread=1
struct ThrNode
{
   ElemType data;       // ElemType 表示不确定的数据类型
   ThrNode *lchild, *rchild;
   flag ltag, rtag;
}*root;                    //root 表示线索链表的头指针
```

- **树的存储结构**

包括：双亲表示法、孩子表示法、孩子兄弟表示法。

双亲表示法的存储结构定义如下：

```
#define MaxSize 100;   //树中最大结点个数
struct PNode               //数组元素的类型
{
   ElemType data;          //树中结点的数据信息，
   int parent;              //该结点的双亲在数组中的下标
};
PNode Tree[MaxSize];
```

孩子表示法的存储结构定义如下：

```
struct CTNode        //孩子结点
{
   int child;
   CTNode *next;
};
struct CBNode       //表头结点
{
   ElemType data;
   CTNode *firstchild;   //指向孩子链表的头指针
};
```

孩子兄弟表示法又称为二叉链表表示法，存储结构定义如下：

```
struct TNode
{
   ElemType data;       // ElemType 表示不确定的数据类型
   TNode *firstchild;   //firstchild 指向该结点的第一个孩子
   TNode *rightsib;     //rightsib 指向该结点的右兄弟
};
```

- **树转换为二叉树**

树转换为二叉树的方法是：

⑴ **加线**——树中所有相邻兄弟结点之间加一条连线；

⑵ **去线**——对树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其它孩子结点之间的连线；

⑶ **层次调整**——以根结点为轴心，将树顺时针转动一定的角度，使之层次分明。

- **森林转换为二叉树**

森林转换为二叉树的方法如下：

⑴ 将森林中的每棵树转换成二叉树；

⑵ 从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连起来后，所得到的二叉树就是由森林转换的二叉树。

- **二叉树转换为树或森林**

树和森林转换为二叉树的过程是可逆的，将一棵二叉树还原为树或森林的方法如下：

⑴ **加线**——若某结点 x 是其双亲 y 的左孩子，则把结点 x 的右孩子、右孩子的右孩子、……，都与结点 y 用线连起来；

⑵ **去线**——删去原二叉树中所有的双亲结点与右孩子结点的连线；

⑶ **层次调整**——整理由⑴、⑵两步所得到的树或森林，使之层次分明。

树的遍历序列与二叉树的遍历序列之间的对应关系

根据树与二叉树的转换关系以及树和二叉树遍历的操作定义可知，树的遍历序列与由树转化成的二叉树的遍历序列之间具有如下对应关系：树的前序遍历序列等于二叉树的前序遍历序列，树的后序遍历序列等于二叉树的中序遍历序列。

- **哈夫曼树中叶子结点的权值**

叶子结点的权值是指对叶子结点赋予的一个有意义的数值量。

- **二叉树的带权路径长度**

设二叉树具有 n 个带权值的叶子结点，从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和称做二叉树的带权路径长度，记为：

$$WPL=\sum_{k=1}^{n} w_k l_k$$

其中，w_k 为第 k 个叶子结点的权值；l_k 为从根结点到第 k 个叶子结点的路径长度。

- **哈夫曼树定义**

给定一组具有确定权值的叶子结点，可以构造出不同的二叉树，将其中带权路径长度最小的二叉树称为**哈夫曼树**，也称为最优二叉树。

- **哈夫曼算法的基本思想**

哈夫曼算法的基本思想是：

⑴ 初始化：由给定的 n 个权值$\{w_1，w_2，…，w_n\}$构造 n 棵只有一个根结点的二叉树，从而得到一个二叉树集合 $F=\{T_1，T_2，…，T_n\}$；

⑵ 选取与合并：在 F 中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和；

⑶ 删除与加入：在 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中；

⑷ 重复⑵、⑶两步，当集合 F 中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

- **图的定义**

图是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：

$$G=(V，E)$$

其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中顶点之间边的集合。

- **无向图与有向图**

若顶点 v_i 和 v_j 之间的边没有方向，则称这条边为无向边，用无序偶对(v_i，v_j)来表示；若从顶点 v_i 到 v_j 的边有方向，则称这条边为有向边（也称为弧），用有序偶对<v_i, v_j>来表示，v_i 称为弧尾，v_j 称为弧头。如果图的任意两个顶点之间的边都是无向边，则称该图为**无向图**，否则称该图为**有向图**。

- **简单图**

若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。

- **邻接、依附**

在无向图中，对于任意两个顶点 v_i 和 v_j，若存在边(v_i，v_j)，则称顶点 v_i 和 v_j 互为邻接点，同时称边(v_i，v_j)依附于顶点 v_i 和 v_j。

在有向图中，对于任意两个顶点 v_i 和 v_j，若存在弧<v_i，v_j>，则称顶点 v_j 是 v_i 的邻接点，同时称弧<v_i，v_j>依附于顶点 v_i 和 v_j。

- **无向完全图、有向完全图**

在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有 n 个顶点的无向完全图有 $n\times(n-1)/2$ 条边。

在有向图中，如果任意两顶点之间都存在方向互为相反的两条弧，则称该图为有向完全图。含有 n 个顶点的有向完全图有 $n\times(n-1)$ 条边。

- **稠密图、稀疏图**

称边数很少的图为稀疏图，反之，称为稠密图。

- **顶点的度、入度、出度**

在无向图中，顶点 v 的度是指依附于该顶点的边的个数，记为 $TD(v)$。在具有 n 个顶点 e 条边的无向图中，有下式成立：

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

在有向图中，顶点 v 的入度是指以该顶点为弧头的弧的个数，记为 $ID(v)$；顶点 v 的出度是指以该顶点为弧尾的弧的个数，记为 $OD(v)$。在具有 n 个顶点 e 条边的有向图中，有下式成立：

$$\sum_{i=1}^{n} ID(v_i) = \sum_{i=1}^{n} OD(v_i) = e$$

- **连通图、连通分量**

在无向图中，若任意顶点 v_i 和 v_j($i\neq j$)之间有路径，则称该图是连通图。非连通图的极大连通子图称为连通分量。

- **强连通图、强连通分量**

在有向图中，对任意顶点 v_i 和 v_j ($i\neq j$)，若从顶点 v_i 到 v_j 和从顶点 v_j 到 v_i 均有路径，则称该有向图是强连通图。非强连通图的极大强连通子图称为强连通分量。

- **邻接矩阵的存储结构定义**

假设图 $G=(V, E)$ 有 n 个顶点，则邻接矩阵是一个 $n\times n$ 的方阵，定义为：

$$\text{arc[i][j]}=\begin{cases}1 & \text{若}(v_i, v_j)\in E \text{ 或} <v_i, v_j>\in E\\ 0 & \text{否则}\end{cases}$$

邻接矩阵的存储结构定义如下：

```
#define MaxSize 10
typedef struct
{
   ElemType vertex[MaxSize];   //存放图中顶点的信息，ElemType 表示不确定的数据类型
   int arc[MaxSize][MaxSize];   //存放图中边的信息
```

```
    int vertexNum, arcNum;          //图的顶点数和边数
} MGraph;
```

- **邻接表的存储结构定义**

邻接表是一种顺序存储与链接存储相结合的存储方法，具体方法为：将顶点 v_i 的所有邻接点链成一个单链表，称为顶点 v_i 的边表（对于有向图则称为出边表），边表的头指针和顶点的数据信息采用顺序存储（称为顶点表）。所以，在邻接表中存在两种结点：顶点表结点和边表结点。

顶点表结点　　边表结点

邻接表表示的结点结构

其中，vertex：数据域，存放顶点信息；

firstedge：指针域，边表的头指针；

adjvex：邻接点域，存放边该顶点的邻接点在顶点表中的下标；

next：指针域，指向边表中的下一个结点。

邻接表的存储结构定义如下：

```
struct ArcNode     //定义边表结点
{
   int adjvex;         //邻接点域
   ArcNode *next;
};
struct VertexNode     //定义顶点表结点
{
   ElemType vertex;       // ElemType 表示不确定的数据类型
   ArcNode *firstedge;
};
#define MaxSize 10
typedef struct
{
   VertexNode adjlist[MaxSize];     //顶点表
   int vertexNum, arcNum;            //图的顶点数和边数
} ALGraph;
```

- **图的遍历次序定义**

深度优先遍历

从图中某顶点 v 出发进行深度优先遍历的基本思想是：

① 访问顶点 v；

② 从 v 的未被访问的邻接点中选取一个顶点 w，从 w 出发进行深度优先遍历；

③ 重复上述两步，直至图中所有和 v 有路径相通的顶点都被访问到。

广度优先遍历

从图中某顶点 v 出发进行广度优先遍历的基本思想是：

① 访问顶点 v；

② 依次访问 v 的各个未被访问的邻接点 v_1，v_2，……，v_k；

③ 分别从 v_1，v_2，…，v_k 出发依次访问它们未被访问的邻接点，直至图中所有与顶点 v 有路径相通的顶点都被访问到。

- **最小生成树的定义**

设 $G=(V, E)$ 是一个无向连通网，生成树上各边的权值之和称为该生成树的代价，在 G 的所有生成树

中，代价最小的生成树称为**最小生成树**。

- **普里姆（Prim）算法的基本思想**

设 $G=(V，E)$是一个无向连通网，令 $T=(U，TE)$是 G 的最小生成树。T 的初始状态为 $U=\{v_0\}$（$v_0\in V$），$TE=\{\ \}$，然后重复执行下述操作：在所有 $u\in U$，$v\in V-U$ 的边中找一条代价最小的边$(u，v)$并入边集 TE，同时 v 并入顶点集 U，直至 $U=V$ 为止。

- **克鲁斯卡尔（Kruskal）算法的基本思想**

设无向连通网为 $G=(V，E)$，令 G 的最小生成树为 $T=(U，TE)$，其初态为 $U=V$，$TE=\{\}$，然后按照边的权值由小到大的顺序，依次考察边集 E 中的各条边。若被考察边的两个顶点属于 T 的两个不同的连通分量，则将此边加入到 TE 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路。如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

- **迪杰斯特拉（Dijkstra）算法的基本思想**

设置集合 S 存放已经找到最短路径的顶点，S 的初始状态只包含源点 v，对 $v_i\in V-S$，假设从源点 v 到 v_i的有向边为最短路径。以后每求得一条最短路径 v，…，v_k，就将 v_k加入集合 S 中，并将路径 v，…，v_k，v_i 与原来的假设相比较，取路径长度较小者为当前最短路径。重复上述过程，直到集合 V 中全部顶点加入到集合 S 中。

- **Floyd 算法的基本思想**

假设从 v_i到 v_j的弧（若从 v_i到 v_j的弧不存在，则将其弧的权值看成∞）是最短路径，然后进行 n 次试探。若 v_i，…，v_k和 v_k，…，v_j分别是从 v_i 到 v_k和从 v_k到 v_j中间顶点的序号不大于 $k-1$ 的最短路径，则将 v_i，…，v_k，…，v_j和已经得到的从 v_i到 v_j中间顶点的序号不大于 $k-1$ 的最短路径相比较，取长度较短者为从 v_i到 v_j中间顶点的序号不大于 k 的最短路径。

- **AOV 网的定义**

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，称这样的有向图为顶点表示活动的网，简称 **AOV 网**。

- **拓扑序列的定义**

设 $G=(V，E)$是一个具有 n 个顶点的有向图，V 中的顶点序列 v_1，v_2，…，v_n称为一个**拓扑序列**，当且仅当满足下列条件：若从顶点 v_i到 v_j有一条路径，则在顶点序列中顶点 v_i必在顶点 v_j之前。

- **拓扑排序的基本思想**

对 AOV 网进行拓扑排序的基本思想是：

⑴ 从 AOV 网中选择一个没有前驱的顶点并且输出它；

⑵ 从 AOV 网中删去该顶点，并且删去所有以该顶点为尾的弧；

⑶ 重复上述两步，直到全部顶点都被输出，或 AOV 网中不存在没有前驱的顶点。

- **查找算法的时间性能**

查找算法用关键码的比较次数来度量查找算法的时间性能。对于查找成功的情况，将关键码比较次数的数学期望值定义为**平均查找长度**，即：

$$ASL=\sum_{i=1}^{n}p_ic_i$$

其中，n 表示问题规模，即查找集合中的记录个数；p_i表示查找第 i 个记录的概率；c_i表示查找第 i 个记录所需的关键码的比较次数。

- **顺序查找算法的时间复杂度**

对于具有 n 个记录的顺序表，查找第 i 个记录时，需进行 $n-i+1$ 次关键码的比较。设每个记录的查找概率相等，查找成功时，顺序查找的平均查找长度为：$O(n)$；查找不成功时，关键码的比较次数是 $n+1$ 次，则查找失败的平均查找长度为 $O(n)$。

- **顺序查找的适用情况**

顺序查找对表中记录的存储没有任何要求，顺序存储和链接存储均可应用；对表中记录的有序性也没有要求，无论记录是否按关键码有序均可应用。

- **折半查找的适用情况**

折半查找（也称对半查找、对分查找、二分查找）要求线性表中的记录必须按关键码有序，并且必须采用顺序存储。

- **折半查找的基本思想**

取有序表的中间记录作为比较对象，则

（1）若给定值与中间记录的关键码相等，则查找成功；

（2）若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；

（3）若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。

不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

- **折半查找的时间复杂度**

具有 n 个结点的折半查找判定树的深度为$\lfloor \log_2 n \rfloor+1$。

最好情况：比较 1 次，即查找的关键码是判定树的根结点；

最坏情况：比较次数为$\lfloor \log_2 n \rfloor+1$，即查找的关键码是判定树的最下一层结点；

平均情况：折半查找的平均时间复杂度为 $O(\log_2 n)$。

查找不成功的比较次数最多不超过树的深度，最多为$\lfloor \log_2 n \rfloor+1$次。

- **二叉排序树的定义**

二叉排序树或者是一棵空的二叉树，或者是具有下列性质的二叉树：

⑴ 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；

⑵ 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；

⑶ 它的左右子树也都是二叉排序树。

- **二叉排序树的查找性能**

如果二叉排序树是平衡的，则其查找效率为 $O(\log_2 n)$。如果二叉排序树为一棵斜树，则其查找效率为 $O(n)$。因此，二叉排序树的查找性能在 $O(\log_2 n)$ 和 $O(n)$ 之间。

- **平衡二叉树的定义**

平衡二叉树或者是一棵空的二叉排序树，或者是具有下列性质的二叉排序树：

⑴ 根结点的左子树和右子树的深度最多相差 1。

⑵ 根结点的左子树和右子树也都是平衡二叉树。

- **构造平衡二叉树的基本思想**

在构造二叉排序树的过程中，每当插入一个结点时，首先检查是否因插入而破坏了树的平衡性，若是，则找出最小不平衡子树，在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。

- **平衡调整的四种类型**

设结点 A 为最小不平衡子树的根结点，对该子树进行平衡化调整有以下四种情况：

⑴ LL 型：结点 x 插在根结点 A 的左孩子的左子树上。

⑵ RR 型：结点 x 插在根结点 A 的右孩子的右子树上。

⑶ LR 型：结点 x 插在根结点 A 的左孩子的右子树上。

⑷ RL 型：结点 x 插在根结点 A 的右孩子的左子树上。

- **散列查找的基本思想**

散列查找也称为哈希查找、Hash 查找，其基本思想是：在记录的存储位置和它的关键码之间建立一个

确定的对应关系 H，使得每个关键码 key 和唯一的一个存储位置 $H(key)$ 相对应。在查找时，根据这个确定的对应关系找到给定值 k 的映射 $H(k)$，若查找集合中存在这个记录，则必定在 $H(k)$ 的位置上。

- **散列查找的基本概念**

采用散列技术将记录存储在一块连续的存储空间中，这块连续的存储空间称为**散列表**，将关键码映射为散列表中适当存储位置的函数称为**散列函数，**所得的存储位置址称为**散列地址**。

对于两个不同的关键码 $k_1 \neq k_2$，有 $H(k_1)=H(k_2)$，即两个不同的记录需要存放在同一个存储位置，这种现象称为**冲突**，k_1 和 k_2 相对于 H 称做**同义词**。

- **散列查找的关键问题**

采用散列技术需要考虑的两个关键问题是：

⑴ 散列函数的设计。如何设计一个简单、均匀、存储利用率高的散列函数。

⑵ 冲突的处理。如何采取合适的处理冲突方法来解决冲突。

- **处理冲突的方法**

开放定址法

用开放定址法处理冲突得到的散列表叫做**闭散列表**。

所谓开放定址法，就是由关键码得到的散列地址一旦产生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

① **线性探测法**

当发生冲突时，线性探测法从冲突位置的下一个位置起，依次寻找空的散列地址，即对于键值 key，设 $H(key)=d$，闭散列表的长度为 m，则发生冲突时，寻找下一个散列地址的公式为：

$$H_i=(H(key)+d_i)\ \%\ m \quad （d_i=1，2，\cdots，m-1）。$$

线性探测法会出现非同义词之间对同一个散列地址争夺的现象，称为**堆积**或**聚集。**

② **二次探测法**

当发生冲突时，二次探测法寻找下一个散列地址的公式为：

$$H_i=(H(key)+d_i)\%\ m（d_i=1^2，-1^2，2^2，-2^2，\cdots，q^2，-q^2 \text{ 且 } q\leqslant m/2）$$

③ **随机探测法**

当发生冲突时，随机探测法探测下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

$$H_i=(H(\text{key})+d_i)\%\ m \quad （d_i \text{ 是一个随机数列，} i=1，2，\cdots\cdots，m-1）$$

拉链法（链地址法）

用拉链法处理冲突构造的散列表叫做**开散列表**。

拉链法的基本思想是：将所有散列地址相同的记录，即所有关键码为同义词的记录存储在一个单链表中——称为同义词子表，在散列表中存储的是所有同义词子表的头指针。

- **直接插入排序的基本思想**

直接插入排序的基本思想是：依次将待排序序列中的每一个记录插入到一个已排好序的序列中，直到全部记录都排好序。

- **直接插入排序算法的性能**

• 时间性能

最好情况：待排序序列为正序，时间复杂度为 $O(n)$；

最坏情况：待排序序列为逆序，时间复杂度为 $O(n^2)$。

平均情况：待排序序列中各种可能排列的概率相同，时间复杂度为 $O(n^2)$。

• 空间性能

直接插入排序只需要一个记录的辅助空间。

• 稳定性

直接插入排序是一种稳定的排序方法。

- **希尔排序的基本思想**

希尔排序的基本思想是：先将整个待排序记录序列分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列基本有序时，再对全体记录进行一次直接插入排序。

- **希尔排序算法的性能**

• 时间性能

希尔排序算法的时间性能是所取增量的函数，其时间性能在 $O(n^2)$ 和 $O(n\log_2 n)$ 之间，当 n 在某个特定范围时，希尔排序的时间性能约为 $O(n^{1.3})$。

• 空间性能

希尔排序只需要一个记录的辅助空间，用于暂存当前待插入的记录。

• 稳定性

希尔排序是一种不稳定的排序方法。

- **起泡排序的基本思想**

起泡排序的基本思想是：两两比较相邻记录的关键码，如果反序则交换，直到没有反序的记录为止。

- **起泡排序算法的性能**

• 时间性能

最好情况：待排序记录序列为正序，时间复杂度为 $O(n)$；

最坏情况：待排序记录序列为反序，时间复杂度为 $O(n^2)$；

平均情况：时间复杂度为 $O(n^2)$。

• 空间性能

起泡排序只需要一个记录的辅助空间，用来作为记录交换的暂存单元。

• 稳定性

起泡排序是一种稳定的排序方法。

- **快速排序的基本思想**

快速排序又称为分区交换排序，其基本思想是：首先选一个轴值（即比较的基准），将待排序记录分割成独立的两部分，左侧记录的关键码均小于或等于轴值，右侧记录的关键码均大于或等于轴值，然后分别对这两部分重复上述过程，直到整个序列有序。

- **快速排序的性能**

• 时间性能

最好情况：时间复杂度为 $O(n\log_2 n)$。

最坏情况：待排序记录序列为正序或逆序，时间复杂度为 $O(n^2)$。

平均情况：时间复杂度为 $O(n\log_2 n)$。

• 空间性能

最好情况下为 $O(\log_2 n)$；最坏情况下，栈的深度为 $O(n)$；平均情况下，栈的深度为 $O(\log_2 n)$。

• 稳定性

快速排序是一种不稳定的排序方法。

- **简单选择排序的基本思想**

简单选择排序的基本思想是：第 i 趟（$1\leqslant i\leqslant n-1$）排序通过 $n-i$ 次关键码的比较，在 $n-i+1$ 个记录中选取关键码最小的记录，并和第 i 个记录交换作为有序序列的第 i 个记录。

- **简单选择排序算法的性能**

• 时间性能

简单选择排序最好、最坏和平均的时间复杂度均为 $O(n^2)$。

• 空间性能

在简单选择排序过程中，只需要一个用来作为记录交换的暂存单元。

• 稳定性

简单选择排序是一种不稳定的排序方法。

- **堆的定义**

堆是具有下列性质的完全二叉树：每个结点的值都小于或等于其左右孩子结点的值（称为**小根堆**）；或者每个结点的值都大于或等于其左右孩子结点的值（称为**大根堆**）。

- **堆排序的基本思想**

首先将待排序的记录序列构造成一个堆（假设利用大根堆），此时，选出了堆中所有记录的最大者即堆顶记录，然后将它从堆中移走（通常将堆顶记录和堆中最后一个记录交换），并将剩余的记录再调整成堆，这样又找出了次大的记录，以此类推，直到堆中只有一个记录为止。

- **堆排序算法的性能**

• 时间性能

堆排序最好、最坏和平均的时间复杂度为 $O(n\log_2 n)$。

• 空间性能

在堆排序算法中，只需要一个用来交换的暂存单元。

• 稳定性

堆排序是一种不稳定的排序方法。

- **二路归并排序的基本思想**

将若干个有序序列进行两两归并，直至所有待排序记录都在一个有序序列为止。

- **二路归并排序算法的性能**

• 时间性能

归并排序算法最好、最坏、平均的时间性能的时间代价是 $O(n\log_2 n)$。

• 空间性能

二路归并排序在归并过程中需要与原始记录序列同样数量的存储空间，以便暂存归并的中间结果，因此其空间复杂度为 $O(n)$。

• 稳定性

二路归并排序是一种稳定的排序方法。