数据结构

数据结构:
data structure
是带有结构的数据元素的集合。
结构指的是数据之间的相互关系，即数据的组织形式，结构中的数据元素称为节点。

- 相关概念
  - 数据（data）
    - 描述客观事物的数/字符以及能输入计算机中并被计算机处理的符号的集合。
  - 数据元素（data element）
    - 数据的基本单位
  - 数据对象（data object）
    - 具有相同性质的数据元素的集合，是数据的一个子集
  - 数据项
    - 具有独立含义的最小标识单位
  - 数据结构的包含的三个方面
    - 逻辑结构

      数据元素之间的逻辑或抽象关系，从逻辑关系上描述数据，与数据的存储结构无关，独立于计算机。可以看作是具体问题抽象处理的数学模型。
      - 线性结构

        数据元素之间存在着一对一的关系，且结构中仅有一个开始节点和一个终端节点，其余节点都是仅有一个直接前趋和一个直接后继。
        例如：链表
      - 非线性结构

        数据元素之间存在着一对多或者多对多的关系。
        例如树结构，网结构，图结构
    - 物理结构

      数据元素及其关系在计算机内从存储方式。
      - 存储结构
        - 顺序存储结构

          按逻辑关系顺序存储
        - 链式存储结构

          在内存中按指针链接存储
      - 存储方法

- 顺序存储方法

  把逻辑上相邻的节点存储到物理结构上也相邻的连续存储单元里，由此得到的存储结构称为顺序存储结构。
  通常借助程序设计语言的数组来描述。

- 链接存储方法

  用一组不一定连续的存储单元存储逻辑上相邻的元素，元素见的逻辑关系是由附加的指针域来表示，由此得到的存储结构称为链式存储结构。

- 索引存储方法

  索引存储方法通常是在存储元素信息的同时，还建立附加的索引表。
  表中的索引项一般形式是：关键字，地址
  关键字是唯一标志一个元素的一个数据项或多个数据项的组合。

- 散列存储方法

  散列存储方法的基本思想是根据元素的关键字直接计算出该元素的存储地址。

- 数据的运算

  即对数据元素施加的操作。

  - 算法的描述

    - 时间复杂度

      假如将算法中基本操作的重复执行次数看成是问题规模n的某个函数f(n)，算法的渐进时间复杂度记作：T(n)=O(f(n))。
      它表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，其中f(n)一般为算法中频度最大的语句频度。
      在分析算法时，往往对算法的时间复杂度和渐进时间复杂度不予区分，而经常是将渐进时间复杂度T(n)=O(f(n))简称为时间复杂度。

    - 空间复杂度

      一个算法空间复杂度S(n)定义为该算法所耗费的存储空间，它是对一个算法在运行过程中临时占用存储空间大小的度量，是问题规模n的函数。

  - 五个准则

    - 输入
    - 输出
    - 有穷性
    - 确定性
    - 可行性

- 表

  - 线性表

    - 定义

      - 1.有且只有一个开头元素，没有前趋，只有一个直接后继
      - 2.有且只有一个终端元素，没有后继，仅有一个直接前趋
      - 3.其余元素称为内部元素，仅有一个直接前趋和一个直接后继

    - 运算

- 常见的基本运算
  - 1.构造空表
  - 2.求表长
  - 3.根据索引查找
  - 4.根据值查找
  - 5.插入
  - 6.删除
- 存储
  - 顺序存储

    线性表的顺序存储指的是将线性表的数据元素按其逻辑次序依次存入一组地址连续的存储单元里，用这种方法存储的线性表称为顺序表。
    - 元素之间的关系

      假设线性表中所有元素的类型都是相同的，且每个元素占用d个存储单元，其中第一个单元的存储地址就是该元素的存储位置
      - a(i+1)跟a(i)关系

        $$LOC(a_{i+1}) = LOC(a_i) + d$$
      - a^i的存储位置

        $$LOC(a_i) = LOC(a_1) + (i-1) * d$$
    - 运算
      - 插入元素
        - 后面的元素需要后移
          - 假设pi是在第i个元素之前插入一个元素的概率，则在长度为n的线性表中插入一个节点是所需要移动元素的期望值（平均数）

            $$E_{is}(n) = \sum_{i=1}^{n+1} p_i(n-i+1)$$
          - 在等概率下插入，需要移动元素的平均次数

            $$E_{is}(n) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$
      - 删除
        - 后面的元素需要前移
          - qi为删除发第i个元素的概率，删除一个元素的平均移动次数

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

- 链式存储

  链式存储结构存储线性表数据元素的存储空间可能是连续的，也可能是不连续的，因而链式表的节点是不可以存取存取的。

  - 单向链表
    - 域
      - 数据域

        存储数据元素的域称为数据域
      - 指针域

        存储直接后继存储地址的域称为指针域
    - 运算
      - 创建
        - 头插法
          - 先生成终点节点，后续的节点插入链表头
        - 尾插法
          - 固定头，新节点接上个的尾部
      - 查找
        - 按结点序号查
        - 按结点值找
      - 插入
      - 删除
  - 循环链表

    循环链表是链式存储结构的另一种形式。
    其特点是单链表中最后一个结点的指针域不为空，而是指向链表头节点，使整个链表构成一个环。
  - 双向链表
    - pre
      - data
        - next

- 栈(Stack)
  - 定义
    - 栈是限定在表的一端进行插入和删除运算的线性表，通常将插入、删除的一端称为栈顶，另一端称为栈第。不含元素的空表称为空栈。
      - 像桶、LIFO(Last In First Out) 后进先出
  - 运算
    - 构造空栈

- InitStack
- 判断栈是否为空
  - StackEmpty
- 判断栈是否已满（满栈）
  - StackFull
- 进栈（入栈-插入）
  - Push
- 退栈（出栈-删除）
  - Pop
- 取栈顶元素
  - GetTop

- 存储结构
  - 顺序存储
    - 顺序栈
      - 数组实现
        - 栈底-数组最低端，下标为0
        - 栈顶-常用整形变量top来值时栈顶位置-通常称top为栈顶指针
      - 运算

```
顺序栈定义：
#define StackSize 100 //栈空间
typedef char DataType;
typedef struct{
      DataType data[StackSize]; //存放结点
      int top;    //栈顶指针
}SeqStack;
SeqStack S;
```

        - 1.置空栈

```
void InitStack(SeqStack * S)
{
      S->top=-1;    //空栈栈顶指针不能是0
}
```

        - 2.判栈空

```
int StackEmpty(SeqStack * S)
{
      return S->==-1;
}
```

        - 3.判栈满

```
int StackFull(SeqStack * S)
{
      return S->top == StackSize - 1;
}
```

- 4.进栈（入栈）

```
void Push(SeqStack * S,DataType x)
{
    if(StackFull(S))
        printf("stack overflow");
    else{
        S->top = S-> + 1;
        S->data[S->top]=x;
    }
}
```

- 5.退栈（弹栈）

```
DataType Pop(SeqStack * S)
{
    if(StackEmpty(S)){
        printf("stack underflow");
        exit(0);
    }else{
        return S->data[S->top--];
    }
}
```

- 6.取栈顶（不改变指针）

```
DataType GetTop(SeqStack * S)
{
    if(SeqStackEmpty(S)){
        printf("stack empty");
    }else{
        return S->data[S->top];
    }
}
```

- 链式存储

  可以解决由顺序存储分配固定空间所产生的一处和空间浪费问题

  - 链栈
    - 链表实现
      - 插入跟删除操作仅限在表头（栈顶）进行
    - 运算

      链栈定义：

```
#define StackSize 100 //栈空间
typedef char DataType;
typedef struct stacknode{
    DataType data; //存放结点
    struct stacknode * next;      //栈顶指针
}StackNode;
typedef StackNode * LinkStack;
LinkStack top;
```

- 1.判栈空

```
int StackEmpty(LinkStack * top)
{
    return top == NULL;
}
```

- 2.进栈（入栈）

```
void Push(LinkStack top,DataType x)
{
    StackNode * p;
    p=(StackNode *)malloc(sizeof(StackNode));
    p->data=x;
    p->next=top;
    top = p;
    return top;
}
```

- 3.退栈（弹栈）

```
void Pop(LinkStack top,DataType x)
{
    StackNode * p = top;
    if(StackEmpty(top))
        printf("stack empty");
    else{
        *x = p->data;
        top=p->next;
        free(p);  //删除p指向的结点
        return top;
    }
}
```

- 4.取栈顶

```
DataType GetTop(LinkStack top)
{
    if(SeqStackEmpty(top)){
        printf("stack empty");
    }else{
        return top->data;
    }
}
```

- 队列（Queue）
  - 定义
    - 队列也是一种操作受限的线性表，它只允许在表的一端进行元素插入，而在另一端进行元素删除。允许插入的一端称为队尾（rear），允许删除的一端称为队头（front）
      - 像排队、先进先出（First In First Out，FIFO）
  - 运算
    - 1.置空队列（构造一个队列）
    - 2.判队空QueueEmpty

- 3.入队列EnQueue
- 4.出队列DeQueue
- 5.取队头GetFront

- 存储结构
  - 顺序存储
    - 顺序队列

      由于队列的队头和队尾的位置是变化的，因此需要设置两个指针front和rear分别指示队头和队尾元素在表中的位置，初始值为0.

      ```
      #define QueueSize 1000
      typedef struct{
            DataType data[QueueSize];
            int front,rear;
      } SeqQueue;
      SeqQueue Q;
      ```

      - 数组实现
        - 入队时，将新元素插入rear所指的位置，再将rear+1
        - 出队时，将front+1，并返回被删除的元素
      - 循环队列

        ```
        #define QueueSize 100
        typedef struct{
              DataType data[QueueSize];
              int front,rear;
        } CirQueue;
        ```

        - 定义
          - 为了解决数组的溢出和浪费问题，可以把数据看出一个环，这样定义的循环队列，数组能被有效利用
        - 运算
          - 1.置空队列（构造一个队列）

            ```
            void InitQueue(CirQueue * Q)
            {
                  Q->front=Q->rear=0;
            }
            ```

          - 2.判队空QueueEmpty

            ```
            int QueueEmpty(CirQueue * Q)
            {
                  return Q->rear==Q->front;
            }
            ```

            - 队满：尾指针追上头指针
            - 对空：头指针追上尾指针

- 3.判队满

```
int QueueFull(CirQueue * Q)
{
    return (Q->rear + 1) % QueueSize == Q->front;
}
```

- 4.入队列EnQueue

```
void EnQueue(CirQueue * Q)
{
    //插入队尾
    if(QueueFull(Q))
        printf("Queue overflow");
    else{
        Q->data[Q->rear]=x;
        Q->rear=(Q->rear+1) % QueueSize;
    }
}
```

- 5.出队列DeQueue

```
DataType DeQueue(CirQueue * Q)
{
    //取队头的值,删除队头元素
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        x = Q->data[Q->front];
        Q->front=(Q->front+1) % QueueSize;
        return x;
    }
}
```

- 6.取队头GetFront

```
DataType GetFront(CirQueue * Q)
{
    //取队头的值
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        return Q->data[Q->front];
    }
}
```

- 链式存储

- 链队列

  限制在表头删除、表尾插入操作的单链表

```
#define QueueSize 1000
typedef struct qnode{
    DataType data[QueueSize];
    int qnode * next;
} QueueNode;
typedef struct{
    QueueNode * front;
    QueueNode * rear;
} LinkQueue;
LinkQueue Q;
```

  - 链表实现
    - 一个队列由一个头指针和一个尾指针唯一确定
  - 运算
    - 1.置空队列（构造一个队列）

```
void InitQueue(LinkQueue * Q)
{
    //头节点
    Q->front=(QueueNode *)malloc(sizeof(QueueNode));
    //尾指针也指向头节点
    Q->rear=Q->front;
    Q->rear->next=NULL;
}
```

    - 2.判队空QueueEmpty

```
int QueueEmpty(LinkQueue Q)
{
    //头尾指针相等队列为空
    return Q->rear=Q->front;
}
```

    - 3.入队列EnQueue

```
void EnQueue(LinkQueue*Q,DataType x)
{
    //将x插入队尾
    QueueNode * p = (QueueNode *)malloc(sizeof(QueueNode));
    p->data = x;
    p->next = NULL;
    Q->rear->next = p;
    Q->rear=p;
}
```

- 4.出队列DeQueue

```
DataType DeQueue(LinkQueue * Q)
{
    //取队头的值,删除队头元素
    QueueNode * p;
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        p = Q->front;
        Q->front=Q->front->next;
        free(p);
        return (Q->front->data);
    }
}
```

- 5.取队头GetFront

```
DataType GetFront(LinkQueue * Q)
{
    //取队头的值
    if(QueueEmpty(Q))
        printf("Queue emtpy");
    else{
        return Q->front->next->data;
    }
}
```

散列

定义

散列（Hash）同顺序、链式或索引存储结构一样，是存储线性表的又一种方法。散列存储的基本思想是：以线性表中的每个元素的关键字key作为自变量，通过一种函数H（key）计算出函数值，把这个函数值解释为一块连续存储空间的单元地址（即下标），将该元素存储到这个单元中。散列存储中使用的函数H（key）称为散列函数或哈希函数函数，它存储关键字到存储地址的映射（或称转换）。H（key）的值称为散列地址或者哈希地址，使用的数组空间是线性表进行散列存储的地址空间，所以被称之为散列表或者哈希表。当在散列表上进行查找时，搜先根据给定的关键字key，用与散列存储时使用的同一散列函数H（key）计算出散列地址，然后按此地址从散列表中取对应的元素。

运算

构造散列

构造散列函数的目标是使散列地址尽可能均匀地分布在散列空间上，同时使计算尽可能简单。

- 直接地址法

直接地址法是以关键字key本身或关键字加上某个常量C作为散列地址的方法。
H(key)=key+C
在使用时，为了使散列地址与存储空间吻合，可以调整C。这种方法计算简单，并且没有冲突。它适合于关键字的分布基本连续的情况，若关键字分布不连续，空号较多，将会造成较大的空间浪费

- 数字分析法

  数字分析法是假设有一组关键字，每个关键字又n位数字组成，如k1,k2...k3。数字分析法是从中提取数字分布比较均匀的若干位作为散列地址。

- 除余数法

  除余数法是选择一个适当的p（p≤散列表长m）去除关键字k，所得余数作为散列地址的方法。
  对应的散列函数H(k)为
  H(k)=k%p

- 平方取中法

  平方取中法是取关键字平方的中间几位作为散列地址的方法，因为一个乘积的中间几位和乘数的每一位都相关，故由此产生的散列地址较为均匀，具体取多少位视实际情况而定。

- 折叠法

  折叠法是首先将关键字分割成位数相同的几段（最后一段的位数可少一些），段的位数取决于散列地址的位数，由实际情况而定，然后将它们叠加和（舍去最高进位）作为散列地址的方法。
  折叠法又分移位叠加和边界叠加。
  移位叠加是将各段的最低为对齐，然后相加；
  边界叠加则是将两边相邻的段沿边界来回折叠，然后对其相加。

- 存储结构

多维数组

定义

- 二维数组矩阵表示

$$D_{m*n} = \begin{bmatrix} a_{00} & a_{11} & \cdots & a_{0,n-1} \\ a_{10} & a_{21} & \cdots & a_{1,n-1} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

存储

顺序存储

按行顺序存储

- $a_{00}, a_{10}, \ldots, a_{0\ n-1}, a_{10}, a_{11}, \ldots, a_{1,n-1}, a_{m-1\ 0}, a_{n-1\ 1}, a_{m-1\ n-}$

按列顺序存储

- $a_{00}, a_{01}, \ldots, a_{m-1\ 0}, a_{01}, a_{11}, \ldots, a_{m-1\ 1}, a_{0\ n-1}, a_{1\ n-1}, a_{m-1\ n-}$

- 运算

  只要知道开始结点的存储地址，维数和每维的上、下界，以及每个元素所占用的单元数，就可以将每个数组元素的存储地址表示为其下标的线性函数。

  - aij的地址计算函数

    $$LOC(a_{ij}) = LOC(a_{00}) + (i * n * j) * d$$

- 矩阵
  - 特殊矩阵
    - 对称矩阵

      $$a_{ij} = a(ji)\ (0 \geq i, j \leq n-1)$$

      - 所以只需存储上三角或下三角的元素即可

        $$\begin{bmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ \cdots & \cdots & \cdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

        - 第i行(0≤i≤n-1)恰好有i+1个元素，元素总数为

          $$\sum_{i=0}^{n-1}(i+1) = n(n+1)/2$$

          - sa[n(n+1)/2]为n阶对称矩阵A的存储结构
          - $$k = \begin{cases} \frac{i*(i+1)}{2} + j & i \geq j, \\ \frac{j*(j+1)}{2} + j & i < j \end{cases} \qquad 0 \leq k \leq n(n+1)/2 - 1 。$$

          - a_ij的存储地址

            $$LOC(a_{ij}) = LOC(sa[k]) = LOC(sa[a]) + k * d$$

    - 三角矩阵

      以主对角线划分，三角矩阵有上三角和下三角两种。
      下三角矩阵正好相反，它的主对角线上方均为常数c或零
      上三角矩阵是指矩阵的下三角（不包括对角线）中的元素均为常数c或零的n阶方阵
      一般情况下，三角矩阵的常数c均为零

      - 存储结构

        $$sa[n(n+1)/2+1]$$

- 上三角矩阵

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ c & a_{11} & a_{12} & \dots & a_{1,n-1} \\ c & c & a_{22} & \dots & a_{2,n-1} \\ \dots & \dots & \dots & \dots & \dots \\ c & c & \dots & c & a_{n-1,n-1} \end{bmatrix}$$

  - 元素总数

$$\sum_{m=0}^{i-1}(n-m) = i*(2*n-i+1)/2$$

  - sa[k]与a_ij存储位置的对应关系

$$k = \begin{cases} i*(2n-i+1)/2+j-i & i \le j, \\ n*(n+1)/2 & i > j \end{cases}$$

- 下三角矩阵

$$\begin{bmatrix} a_{00} & c & c & \dots & c \\ a_{10} & a_{11} & c & \dots & c \\ a_{20} & a_{21} & a_{22} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1,n-1} \end{bmatrix}$$

  -

$$k = \begin{cases} i*(i+1)/2+j & i \ge j, \\ n*(n+1)/2 & i < j \end{cases}$$

- 稀疏矩阵

  有一种特殊矩阵，其中有s个非零元素，而s远远小于矩阵元素的总数，通常把这种矩阵称为稀疏矩阵。

- 三元组表

  如果将表示稀疏矩阵非零元素的三元组（i,j,aij）按行优先的顺序排列，则可得到一个其结点均为三元组的线性表，将这种线性表的顺序存储结构称为三元组表。

- 定义

```
#define MaxSize 1000
typedef struct{
    int i,j;//非零元素的行号，列号（下标）
    DataType v;//非零元素
} TriTupleNode;

typedef struct{
    //存储三元组的数组
    TriTupleNode data[MaxSize];
    //矩阵的行数/列数和非零元素个数
    int m,n,t;
} TSMatrix;//稀疏矩阵类型
```

- 带行表的三元组表

  为了便于随机存取任意一行的非零元素，可在按行优先存储的三元组表中，增加一个存储每一行的第一个非零元素在三元组表中位置的数组。
  这样就得到稀疏矩阵的另一种顺序存储结构，带行表的三元组表，又称为行逻辑链接顺序表。

  - 定义

```
type struct{
    TriTupleNode data[MaxSize];
    //每行第一个非零元素的位置表
    int RowPos[MaxRow];
    int m,n,t;
} RLSMatrix;
```

- 广义表

  广义表是线性表的推广，又称列表。

  - 定义

    广义表是n(n≥0)个元素a1,a2,a3,...,an的有限序列，其中ai可以是原子项，也可以是广义表。
    通常记LS=(a1,a2,a3,...,an)，LS是广义表的名字，n为它的长度，层数为它的深度。
    a1是表头（head），a2....an为表尾（tail）

  - 性质
    - 广义表的元素可以是子表，而子表又可以含有子表
    - 广义表具有递归和共享的性质
  - 运算
    - 构建广义表
      - 遇到左括号时递归构造子表，否则构造原子结点；遇到逗号时递归构造后续广义表，直到表达式字符串输入结束。（假设是输入字符串构造）
    - 输出广义表
      - 遇到tag=1的结点，是一个子表的开始，则先打印一个左括号，如果该子表为空，则输出一个空格符，否则递归调用输出该子表，子表打印完毕后输出右括号；遇到tag=0的结点，则直接输出其数据域的值。若还有后续元素，则递归调用打印后续每个元素，知道遇到link域为NULL。

- 查找广义表
  - 若遇到tag=0的原子结点，如果是要找的结点，则查找成功；否则，若还有后续元素，则递归调用本过程查找后续元素，知道遇到link域为NULL的元素。若遇到tag=1的结点，则递归调用本过程在该子表中查找，若还有后续元素，则递归调用本过程查找后续每个元素，知道遇到link域为NULL的元素 。
- 求广义表表头
  - 广义表的第一个元素
- 求广义表表尾
  - 除了第一个元素外的所有元素
- 求广义表深度
  - 扫描广义表的第一层每个结点，队每个结点递归调用计算出其子表的深度，取最大的子表深度，然后加1即为广义表的最大深度。

- 存储结构

```
//当atom=0表示原子，list=1表示子表
typedef enum {atom,list} NodeTag;
typedef struct GLNode{
    //用以区分原子结点和表结点
    NodeTag tag;
    union{
        //用以存放原子结点的值
        DataType data;
        //指向子表指针
        GLNode * slink;
    };
    //指向下一个表结点
    GLNode * next;
} * Glist; //广仪表类型
```

- 查找
  - 平均查找长度

    由于查找运算的主要操作是关键字的比较，因此，通常把查找过程中的平均比较次数（也称为平均查找长度:Average Search Length,ASL）作为衡量一个查找算法效率优劣的标准。

    - 计算公式

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

    - 计算公式简化版

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

  - 顺序表的查找

- 顺序查找

  顺序查找（Sequential Search）又称线性查找，是一种最简单和最基本的查找方法。
  其基本思想是：从表的一端开始，顺序扫描线性表，依次把扫描到的记录关键字与给定的值*k*相比较，若某个记录的关键字等于k，则表明查找成功，返回该记录所在的下标；若直到所有的记录都比较完，仍未找到关键字与*k*的记录，则表明查找失败，返回空值。

  - 平均查找长度

$$ASL = \sum_{i=1}^{n} P_i C_i = \sum_{i=1}^{n} P_i(n-i+1) = \frac{n+1}{2}$$

- 二分查找

  二分查找（Binary Search）又称折半查找。
  二分查找要求查找对象的线性表必须是顺序存储结构的有序表。
  其基本思想是：首先将待查的*k*值和有序表R[1...n]的中间位置mid上的记录关键字进行比较，若相等，则查找成功，返回该记录的下标mid；否则，若R[mid].key>*k*，*则k*在左子表R[1....mid-1]中，接着再再左子表中进行二分查找即可；否则，若R[mid].key<*k*，则说明待查找记录再右子表R[mid+1..n]中，接着只要再右子表中进行二分查找即可。这样，经过一次的关键字比较，就可以缩小一半的查找空间，如此进行下去，直到找到关键字为k的记录或者当前的查找区间为空时为止。二分查找的过程是递归的，因此可以用递归的方法处理。

  - 平均查找长度

$$ASL = \sum_{j=1}^{n} P_i C_i = \frac{1}{n} \sum_{j=1}^{h} j.2^{j-1} = \frac{n+1}{n} log_2^{(n+1)} - 1$$

- 索引顺序查找

  索引顺序查找又称分块查找。
  它要求按如下的索引方式来存储线性表：将表R[1..n]均分为b快，前B1快中的结点个数为s=[n/b]，第b快的结点数≤s;每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块的最小关键字，即要求表是“分块有序”的；抽取各块中的最大关键字及其起始位置构成一个索引表ID[1..b]，即ID[i](1≤i≤b)中存放着dii快的最大关键字及该块再表R中的起始位置，显然，索引表是按关键字递增有序的。
  分块查找的基本思想：首先查找索引表，可用二分查找或者顺序查找，然后再确定的快中进行顺序查找。由于分块查找实际上是两次查找过程，因此整个查找过程的平均查找长度是两次查找的平均查找长度之和。

  - 平均查找长度

$$ASL_{blk} = (b+1)/2 + (s+1)/2 = (s^2 + 2s + n)/(2s)$$

- 树表的查找

  - 二叉排序树

    二叉排序树（Binary Sort Tree，BST）又称二叉查找树。
    二叉排序树的性质:
    1. 若它的右子树非空，则右子树上所有结点的值均大于根结点的值。
    2. 若它的左子树非空，则左子树上所有的结点的值均小于根结点的值。
    3. 左、右子树本身又各是一颗二叉排序树。

    二叉树查找思想：若二叉树为空，则直接返回空。否则，若给定值key等于根结点的关键字，则表明查找成功，返回当前根结点指针；若给定值key小于根结点关键字，，则继续在根结点的左子树中查找，若给定值key大于根结点的关键字，则继续在根结点的右子树中查找。显然这是各递归查找过程。

- B树

  B树的定义：一颗m(m≥3)阶的B树，或为空树，或为满足下列性质的m叉树：
  1、每个结点至少包含下列信息域：
  （n,p0,k1,p2,k2,...,kn,pn）
  其中，n为关键字的个数；k1（1≤i≤n）为关键字，且k1<ki+1()（1≤i≤n-1）；pi（0≤i≤n）为指向子树根结点的指针，且pi所指向子树中所有结点和关键字均小于ki+1，pn所指向子树中所有结点关键字均大于kn；
  2、树中每个结点至多有m棵子树。
  3、若树非空，则根结点至少有1个关键字，至多有m-1个关键字。因此，若根结点不是叶子，则它至少有两棵子树。
  4、所有的叶结点都在同一层上，并且不带信息，叶子的层数为树的高度h。
  5、每个非根结点中所包含的关键字个数满足：[m/2]-1≤n≤m-1。因为每个内部结点的度数正好是关键字总数加1，所以，除根结点之外的所有非终端结点至少有[m/2]棵子树，至多有m棵子树。

- B+树

  B+树是一种常用于文件组织的B树的变形树。一颗m阶的B+树和m阶的B树的差异在于：
  1、有k个孩子的结点必包含k个关键字
  2、所有的叶结点中包含了关键字的信息及指向相应结点的指针，且叶子结点本身以找关键字的大小自小到大顺序链接。
  3、所有非终端结点可看成是索引部分，结点中仅含有其子树（根结点）中的最大关键字（或最小关键字）。

- 散列表的查找

  散列的查找是通过对记录关键字值进行某种运算直接求出记录的地址，是一种由关键字到地址的直接转换方法，不需要反复比较。

排序

评价排序算法的标注主要有两条：
执行算法需要的时间，以及算法所需要的附加空间。
另外，算法本身的复杂度也是考虑的重要因素之一。

内排序

整个待排序数据都在内存中处理，不涉及数据的内外存交换，则称这种排序为内排序；反之为外排序。

插入

插入排序的主要思想：每次将一个待排序的记录按其关键字的大小插入到前面以及排好序的文件中的适当位置，直到全部记录插入完为止。

- 直接插入排序

  基本操作：假设待排序的记录存储在数组R[1...n]中，在排序过程的某一时刻，R被划分成两个子区间，R[1...i-1]和[i...n]，其中前一个为已排好序的有序区，而后一个为无序区，开始时有序区中只含有一个元素R[1]，无序区为R[2...n]。排序过程中，只需要每次从无序区中取出第一个元素，把它插入到有序区的适当位置，使之成为新的有序区，依次这样经过n-1次插入后，无序区为空，有序区包含了全部n个元素，至此排序完毕。

- 希尔排序

  基本思想：先取定一个小于n的整数d1作为第一个增量，把数组R中的全部元素分成d1个组，所有下标距离为d1的倍数的元素放在同一个组中，即R[1]，R[1+d1]，R[1+2d1]，...为第一组，R[2]，R[2+d1]，R[2+2d1]，...为第二组，.....，接着在各组内进行直接插入排序；然后再取d2（d2<d1）为第二个增量，重复上述分组和排序，直到所取的增量d_t=1(d_{t-1}<...<d2<d1)，把所有的元素放在同一组中进行直接插入排序为止。

- 交换

  交换排序的基本思想:两两比较待排序记录的关键字，如果发现两个记录的次序相反时即进行交换，直到所有记录都没有反序时为止。

  - 冒泡排序

    冒泡排序基本思想：通过相邻元素之间的比较和交换，使关键字较小的元素逐渐从底部移向顶部，就像水底下的气泡一样逐渐向上冒泡，所以使用该方法的排序称为“冒泡”排序，也称“起泡”排序。
    冒泡排序过程描述：首先将R[n].key和R[n-1].key进行比较，若R[n].key<R[n-1].key，则交换R[n]和R[n-1]，使轻者上浮，重者下沉；接着比较R[n-1].key和R[n-2].key，同样使轻者上浮，重者下沉，依次类推，直到比较R[2].key和R[1].key，若反序则交换，第一趟排序结束，此时，记录R[1]的关键字最小。然后再对R[n]~R[2]的记录进行第二趟排序，使次小关键字的元素上浮到R[2]中，重复n-1趟后，整个冒泡排序结束。

  - 快速排序

    快速排序又称为划分排序：是对冒泡排序的一种改进方法，其基本思想是：
    首先再当前无序区R[low...high]中任取一个记录作为排序比较的基准，用此基准将当前无序区划分为两个较小的无序区[row...i-1]和[i+1...high]，并使左边的无序区中所有记录的关键字均小于等于基准的关键字，右边的无序区中所有记录的关键字均大于等于基准的关键字，而基准记录x则位于最终排序的位置i上，即R[low...i-1]中关键字<=基准<=R[i+1...high]中的关键字。这个过程称为一趟快速排序。当R[row...i-1]和R[i+1...high]均非空时，分别对它们进行上述划分，直到所有的无序区中的记录均已排好序为止。

    具体操作:设两个指针i和j，它们的初值分别为low和high，基准记录x=R[i]，首先从j所指位置起向前搜索找到第一个关键字小于基准x.key的记录存入当前i所指向的位置上，i自增1，然后再从i所指位置向后搜索，找到第一个关键字大于x.key的记录存入当前j所指向的位置上，j自减1；重复这两步，直至i=j为止。

- 选择

  选择排序基本思想：每一趟再待排序的记录中选出关键字最小的记录，依次存放再已排好序的记录序列的最后，直到全部记录排序完为止。

  - 直接选择排序

    直接选择排序基本思想：每次从待排序的无序区中选择出关键字值最小的记录，将该记录与该区中的第一个记录交换位置。初始时，R[1...n]为无序区，有序区为空。第一趟排序是在无序区R[1...n]中选出最小的记录，将它与R[1]交换，R[1]为有序区；第二趟排序是在无序区R[2...n]中选出最小的记录与R[2]交换，此时R[1...2]为有序区；以此类推，做n-1趟排序后，区间R[1...n]中的记录按递增有序。

  - 堆排序

    堆排序是对直接排序的一种改进。
    其基本思想是：在排序过程中，将记录数组R[1...n]看成是一棵完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系，在当前无序区中选择关键字最大（或者最小）记录。

- 归并

- 二路归并排序

  基本思想：首先将待排序文件看成n个长度为1的有序子文件，把这些子文件两两归并，得到[n/2]个长度为2的有序子文件；然后再把着[n/2]个有序子文件两两归并，如此反复，直到最后得到一个长度为n的有序文件为止。

- 分配

  不需要使用比较的排序算法

  - 有箱排序

    箱排序又称桶排序，其基本思想是：设置若干个箱子，依次扫描待排序的记录R[0]，R[1]，...，R[n-1]，把关键字等于k的记录全部装入第k个箱子里（分配），然后按序号依次将各非空的箱子首尾连接起来。

  - 基数排序

    基数排序是对箱排序的一种改进，其基本思想是：首先按关键字的最低位k_{i}^{d-1}进行箱排序，然后再按关键字的k_{i}^{d-2}进行箱排序，...，最后按最高位k_i^0进行箱排序。

- 外排序

- 图

  图形结构简称为图（Graph）。
  图G由两个集合V和E组成，定义为
  G=(V, E)
  其中V是顶点的有限非空集合，
  E是由V中顶点偶对表示的边的集合。
  通常，V(G)和E(G)分别表示图G的顶点集合和边集合。
  E(G)可以为空集，即图G只有顶点没有边。

  - 有向图

    对于一个图G，若每条边都是有方向的，则称该图为有向图。

    - 有向边（弧）

      在有向图中，由两个顶点组成的有序对的边称为有向边，通常用尖括号便是。
      <vi,vj>和<vj,vi>是两条不同的有向边。

      有向边又称为弧，边的起点称为狐尾，边的终点称为弧头。

      - 出边

        有向边<vi,vj>,顶点vi的一条出边

      - 入边

        有向边<vi,vj>,顶点vj的一条入边

    - 有向完全图

      具有n(n-1)条边或弧的有向图称为有向完全图

    - 度

      - 入度

        顶点到终点的入边数目，记为ID(v)

      - 出度

        出度是以该顶点为起点的出边数目，记为OD(v)

- 路径

  在图G中，若存在一个顶点序列Vp,V1,V2,...V3,Vq，使得(vp,v1),(v1,v2),...,(v3,vq)均属于E(G)，则称顶点Vp到Vq存在一条路径。

  在有向图中，路径也是有向的。

  - 路径长度

    路径长度是指一条路径上经过的边的数目。

  - 简单路径

    若一条路径上除了起点和终点可以为同一个顶点外，其余顶点均不相同的路径称为简单路径。

    - 回路（环）

      若一条简单路径上的起点和终点为同一个顶点，则称该路径为回路或环。

- 强连通图

  在有向图G中，如果任意两个顶点Vi和Vj都连通，则称图G为强连通图。

- 无向图

  对于一个图G，若每条边都是没有方向的，则称该图为无向图。

  - 无向边

    在一个无向图中，边均是顶点的无序对，通常用圆括号表示。

    因此(vi,vj)和(vj,vi)是表示同一条边。

  - 无向完全图

    具有1/2*n*(n-1)条边的无向图称为无向完全图。

  - 度

    在无向图中，顶点v的度定义为以该顶点为一个端点的边的数目，记为D(v)。

  - 连通

    在无向图G中，如果从顶点Vi到顶点Vj有路径，则称Vi和Vj是连通的。

    - 连通图

      若图G中任意两个顶点Vi和Vj都是连通，则称图G为连通图。

      - 连通分量

        无向图的极大连通图子图称为连通分量。

- 权

  若在一个图中的每条边上标上某种数值，该数值称为该边的权。

  - 带权图

    边上带权的图称为带权图

    - 网络

      带权图的连通图称为网络

- 顶点数n、边数e、度数D(v)的关系

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

- G'是G的子图

$$G =< V, E >$$
$$G\prime =< V\prime, E\prime >$$
$$V\prime \subseteq V$$
$$E\prime \subseteq E$$
记作： $G\prime \subseteq G$
则称 $G\prime$ 是 G 的子图

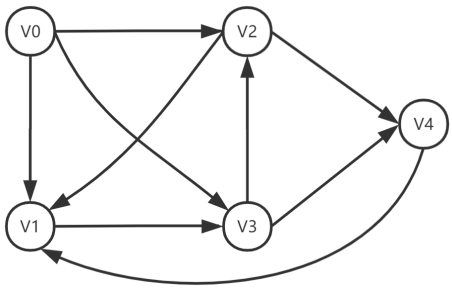
- 存储结构
  - 邻接矩阵
    表示图形中顶点之间相邻关系的矩阵
    - n阶方阵

$$A[i][j] = \begin{cases} 1 & \text{若 } (v_i, v_j) \text{ 或} < v_i, v_j > \text{是 } E(G) \text{ 的边} \\ 0 & \text{若 } (v_i, v_j) \text{ 或} < v_i, v_j > \text{不是 } E(G) \text{ 的边} \end{cases}$$

结构定义：

```
#define MaxVartextNum 50 //最大顶点数
typedef struct{
    //顶点数组
    VertextType vext[MaxVertexNum];
    //邻接矩阵
    Adjmatrix arcs[MaxVertexNum][MaxVertexNum];
}MGraph;
```

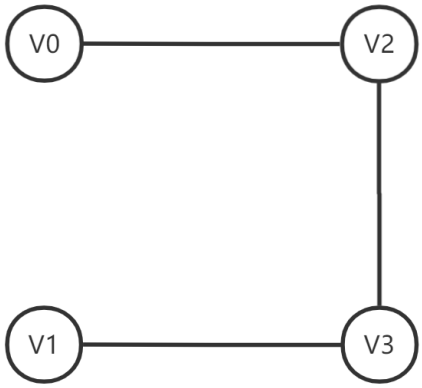
      - 有向图
        - 有向图



- 相邻居然表示法

$$A=\begin{bmatrix}0&1&1&1&0\\0&0&0&1&0\\0&1&0&0&1\\0&0&1&0&1\\0&1&0&0&0\end{bmatrix}$$

- 无向图
  - 无向图



  - 相邻矩阵表示法

$$A=\begin{bmatrix}0&0&1&1\\0&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix}$$

    - 对称，可以采用压缩存储

- 邻接表

  链式存储结构

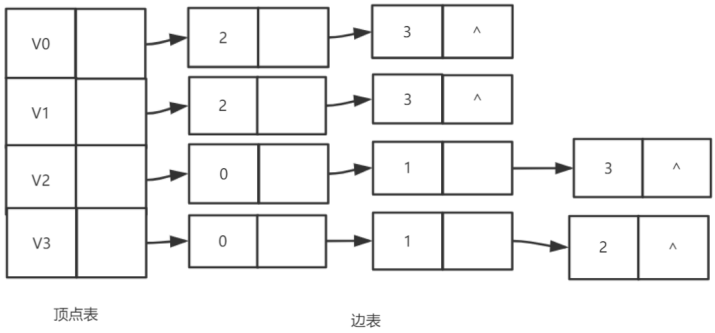
- 存储结构定义

```
#define MaxVertextNum 20
typedef char VertexType;
typedef struct node{ //边表结构类型
      int adjvex;//顶点序号
      //指向下一条边的指针
      struct node * next;
}EdgeNode;

typedef struct vnode{ //顶点表结点
      VertexType vertex; //顶点域
      EdgeNode * link;  //边表指针
}VNode,Adjlist[MaxVertextNum]; //邻接表

//图类型
typedef Adjlist ALGraph;
```

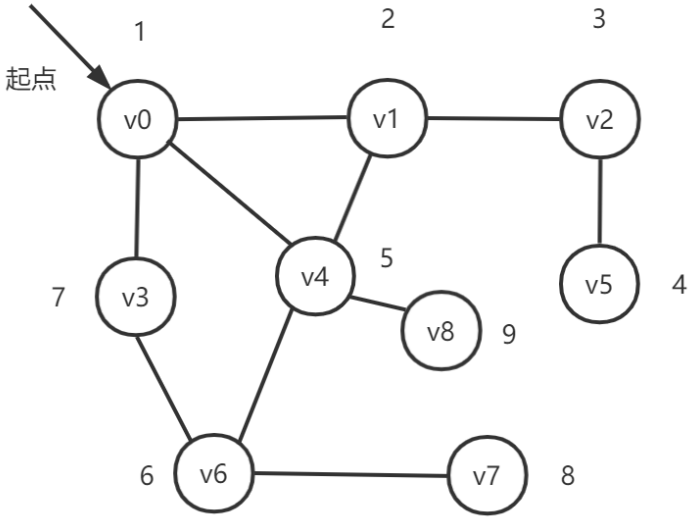
  - 有向图
    - 有向图邻接表
    - 有向图逆邻接表
  - 无向图
    - 无向图邻接表



- 运算
  - 图的遍历
    - 深度优先搜索遍历

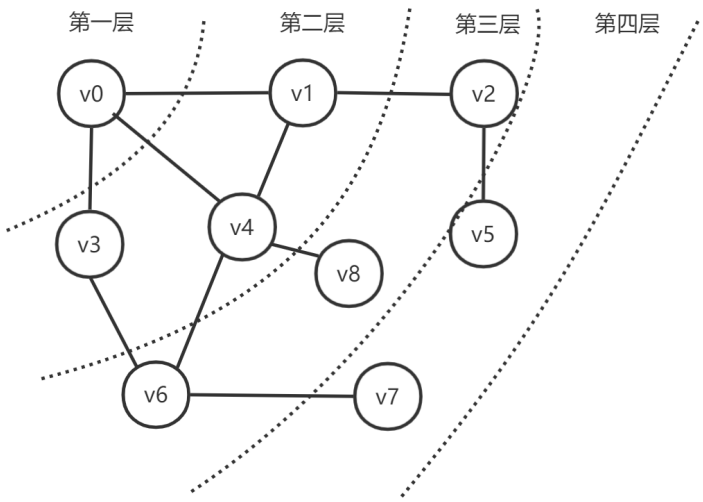
深度优先搜索（Depth First search，DFS）
假设初始状态是图中所有顶点都未曾访问过，则可从图G中任选一顶点v为初始出发点，首先访问出发点v，并将其标记为已访问过；然后依次从v出发搜索v的每个邻接点w，若w未曾访问过，则以w作为新的出发点出发，继续进行深度优先遍历，知道图中所有和v有路径相通的顶点都被访问到；若此时图中仍有顶点未被访问，则另选一个未曾访问的顶点作为起点，重复上述过程，直到图中所有顶点都被访问到为止。

- 图的深度优先遍历序列



对图进行深度优化遍历时，按访问顶点的先后次序得到的顶点序列称为图的深度优先遍历序列，或简称DFS序列。

- 广度优先搜索遍历



广度优先搜索（Breadth First Search，BFS）遍历类似于树的按层次遍历。
其基本思想是：首先访问出发点vi，接着依次访问vi的所有未被访问过的邻接点vi1,vi2.....vit并均标记为已访问过，然后再按照vi1,vi2.....vit的次序，访问每一个顶点的所有未层访问过的顶点并均标记为已访问过，依此类推，直到图中所有和初始出发点vi有路径相通的顶点都被访问过为止。

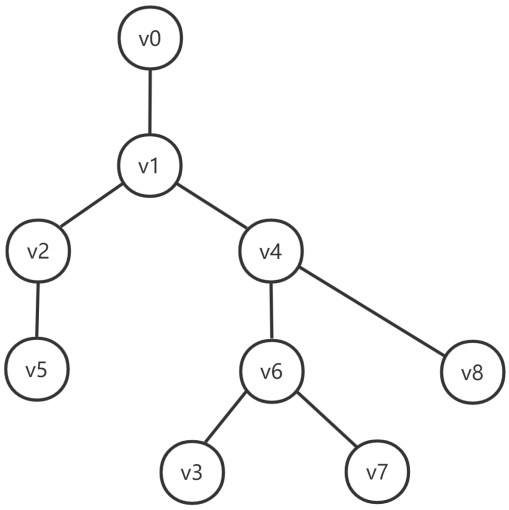
- 图的生成树
  - 连通图
    - 无向图

      若从图的某个顶点出发，可以系统地访问到图的所有顶点，则遍历时经过的边和图的所有顶点所构成的子图，称为该图的生成树。此定义对有向图同样适用。

    - 强连通图

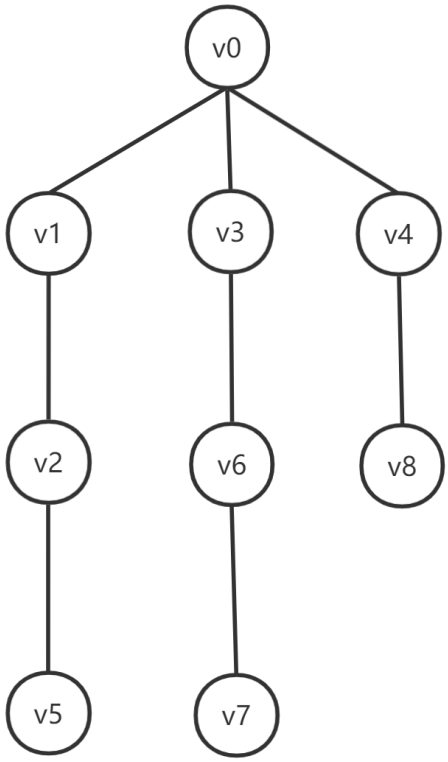
      从图中的其中任意一顶点v出发，都可以访问遍历图中的所有顶点，从而得到以v为根的生成树。若图是有根的有向图，设根为v，则从根v出发也可以完成对图的遍历，因而也能得到以v为根的生成树。

- DFS生成树



由深度优先搜索所得的生成树称之为深度优先生成树，简称DFS生成树。

- BFS生成树



由广度优先搜索所得的生成树称之为广度优先生成树，简称BFS生成树。

- 最小生成树

把生成树各边的权值总和称为该树的权，把权值最小的生成树称为图的最小生成树（Mininum Spanning Tree,MST）。

  - 性质

    假设N=(V,{E})是一个连通网，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属于U，v属于V-U，则必存在一颗包含边（u,v）的最小生成树。

  - 普利姆（Prim）算法
  - 克鲁斯卡尔（Kruskal）算法

- 最短路径
  - 迪杰斯特拉算法

    迪杰斯特拉算法求最短路径的实现的思想：设有向图G=(V,E)，其中，V={1,2,...,n},cost是表示G的邻接矩阵，cost[i][j]表示有向边<i,j>的权。若不存在有向边<i,j>，则cost[i][j]的权无穷大。设S是一个集合，其中的每个元素表示一个顶点，从源点到这些顶点的最短距离已经求出。设顶点v1为源点，集合S的初始只包含顶点v1。数组dist记录从源点到其他各顶点当前的最短距离，其初值为dist[i]=cost[v1][i],i=2,...,n。从S之外的顶点集合V-S中选出一个顶点w，是dist[w]的值最小。从源点到达w只通过S中的顶点，把w加入集合S中并调整dist中记录的从源点到V-S中的每个顶点v的距离，即从原来的dis[v]和dist[w]+cost[w][v]中选择较小的值作为新的dist[v]。重复上述过程，直到S中包含V中其余顶点的最短路径。
- 拓扑排序
  - AOV网

    把顶点表示活动，边表示活动间先后关系的有向无环图（DAG）称为顶点活动网，简称AOV网。
  - 拓扑序列

    在AOV网中，若不存在回路（即环），所有活动可排成一个线性序列，使得每个活动的所有前趋活动都排在该活动的前面，此序列就是拓扑排序。
    - 拓扑排序

      有AOV网构造拓扑序列的过程称为拓扑排序。
      - 1、在有向图中选一个没有前趋（入度为零）的顶点，且输出之。
      - 2、从有向图中删除该顶点及其与该顶点有关的所有边。
      - 3、重复执行上述两个步骤，直到全部顶点都已输出或图中剩余的顶点中没有前趋（入度为零）顶点为止。
      - 4、输出剩余的无前趋结点。

- 树
  - 定义

    树是n(n≥0)个结点的有限集T。
    可空（空树n=0），可非空集。

    对于任意一颗非空树：
    1、有且仅有一个特定的称为根（Root）的结点。
    2、当n>1时，其余的结点可分为m(m>0)个互不相交的有限集T1,T2,...,Tm,其中每个集合本身又是一棵树，并称为根的子树。
  - 表示法
    - 树形图表示法
    - 嵌套集合表示法
    - 凹形表示法
    - 广义表表示法
  - 相关术语
    - 度（Degree）

      一个结点拥有的子树数称为该结点的度。

- 度数等于零
  - 叶子（Leaf）

    度数为零的结点称为叶子。
- 度数不为零
  - 非终端结点或分支结点

    除了根结点之外，分支结点也称为内部结点，而根结点又称为开始结点。

- 层次（Level）

  树中结点的层次是从根开始算起，根为第一层，其余结点的层次等于其父结点的层次+1.

  - 深度（Depth）

    树中结点的最大层次称为树的深度或高度。
- 森林（Forest）

  森林是m（m≥0）棵互不相交的树的集合。

- 二叉树（Binary Tree）
  - 定义

    二叉树是n(n≥0)个结点的有限集合。
    每个结点最多只有两棵子树。
    它或者是空集，或者是由一个根节点及两棵互不相交的分别称作这个根的左子树和右子树的二叉树组成。
  - 性质
    - 在二叉树的第i层上至多有2^{i-1}个结点（i≥1）
    - 深度为k的二叉树至多有2^{k}-1个结点(k≥1)
    - 对任何一棵二叉树T，若其终端结点数为n0，度数为2的结点数为n2，则n0=n2+1
  - 存储结构
    - 顺序存储结构

      从树根开始自上到下，每层从左到右地给该树中每个结点进行编号。然后以各结点的编号为下标，把每个结点的值对应存储到一个一维数组中。
      - 对于完全二叉树
        - 既简单又节省存储空间
      - 对于一般二叉树
        - 需要构造虚拟结点实质称为完全二叉树，浪费存储空间

- 链式存储结构

```
typedef struct node{
      DataType data;
      struct node * lchild, * rchild;
}BinTNode;
typedef BinTNode * binTree;
```

```
-----------------------------
|  lchild  |  data  |  rchild  |
-----------------------------
```

or

```
-------------------------------------
| lchild  |  data  |  parent  | rchild  |
-------------------------------------
```

- 运算
  - 生成
    - 广义表生成法

      (A (B (, D (E ,F)),C))

    - 完全二叉树法

      先对一般的二叉树添加虚结点，使之成为完全二叉树，然后建立结点，如果是第一个结点，则令其为根结点，否则将新结点作为左子节点或右子节点连接到它的父级上。如此重复下去。
  - 遍历
    - 递归遍历法

      访问根结点，遍历左子树和遍历右子树
    - 非递归遍历法

- 利用栈的非递归中序遍历算法

```
void Inorder1(BinTree bt)
{
    SeqStack S;
    BinTNode * p;
    while(!StackEmpty(&S)){
        while(GetTop(&S)){
            //直到左子树空为止
            Push(&S,GetTop(&S)->lchild);
        }
        //空指针退栈
        P = Pop(&S);
        if(!StackEmpty(&S)){
            printf("%c",GetTop(&S)->data);
            p = Pop(&S);
            //右子树进栈
            Push(&S,p->rchild);
        }
    }
}
```

- 指针数组算法

```
void Inorder2(BinTree bt)
{
    BinTNode *ST{100};
    int top = 0;
    ST[top] = bt;
    do{
        while(ST[top]!=NULL){
            top = top + 1;
            ST[top] = ST[top -1]->lchild;
        }
        top = top -1;
        if(top >= 0){
            printf("%c",ST[top]->data);
            ST[top]=ST[top]->rchild;
        }
    }while(top != -1);
}
```

- 满二叉树

一颗深度为k且有2^{k}-1个结点的二叉树称为满二叉树。

每一层的结点数都达到最大值，因此不存在度数为1的结点，且所有叶子结点都在第k层上。

- 全二叉树

若一颗深度为k的二叉树，其前k-1层是一颗满二叉树，而最下面一层，即第k层上的结点都集中在该层最左边的若干位置上，则称此二叉树为完全二叉树。

- 二叉排序树

1、若左子树不空，则左子树上所有结点的值均小于它的根结点的值；
2、若右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3、左、右子树也分别为二叉排序树；
4、没有键值相等的结点。

- 红黑树
- 索引二叉树

```
-----------------------------------
| lchild | ltag | Data | rtag | rchild |
-----------------------------------
```

- 树和森林
  - 存储结构
    - 双亲表示法

```
#define MaxTreeSize 100;
typedef struct{
    DataType data;
    int parent;//父节点位置域
} PTNode;
typedef struct{
    PTNode node[MaxTreeSize];
    int n;//结点数
} Ptree;
```

    - 孩子链表法

```
typedef struct cnode{
    int child;
    struct cnode * next;
}CNode;
typedef struct{
    DataType data;
    CNode * firstchild;
}PANode;
typedef struct{
    PANode nodes[MaxTreeSize];
    int n,r;
}CTree;
```

  - 转二叉树
    - 树转二叉树

      1、所有兄弟结点之间加一道连线
      2、对每个结点保留最左边的连线，去掉该结点与其他孩子的连线。

    - 森林转二叉树

      1、将森林中的每棵树转成二叉树
      2、将二叉树的根结点看作是兄弟连在一起，形成一棵二叉树

    - 二叉树转树

      1、若二叉树中结点X是双亲Y的左孩子，则把X的右孩子、右孩子的右孩子，……，都与Y用连线连起来。
      2、去掉所有双亲到右孩子的连线。

- 哈夫曼树

  哈夫曼树又称最优树，是一类带权路径长度最短的树。

  - 相关术语

- 路径长度

  两个结点构成的路径上的分支数目称为路径长度。
  树根到树中每个结点的路径长度之和称为树的路径长度。
  完全二叉树是路径长度最短的二叉树。

- 权

  将书中的结点赋上一个具有某种意义的实数，我们称此实数为该结点的权。

- 带权路径长度

$$WPL = \sum_{i=1}^{n} w_i l_i$$

  1. 带权路径长度：从树根结点到某个结点之间的路径长度与该结点上权的乘积称为该结点的带权路径长度。
  2. 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称为树的带权路径长度。

- 哈夫曼算法

  1. 根据与n个权值{w1,w2,…,wn}对应的n个结点构成n棵二叉树的森林F={T1,T2,...,Tn},其中每棵二叉树Ti都只有一个权值为wi的根结点，其左、右树均为空。
  2. 在森林F中选出两棵根结点的权值最小的树作为一颗新树的左、右子树，且置新树的附加根结点的权值为其左、右子树上根结点的权值之和。
  3. 从F中删除这两棵树，同时把新树加入到F中。
  4. 重复步骤2和3，直到F中只有一棵树为止，此树便是哈夫曼树。

- 哈夫曼编码

  利用哈夫曼树求得的用于通信的二进制编码称为哈夫曼编码。
  树中从根到每个叶子都有一条路径，对路径上的各分支约定指向左子树的分支表示“0”码，指向右子树的分支表示“1”码，取每条路径上的0或1的序列作为各叶子结点对应的字符编码，这就是哈夫曼编码。