逻辑结构与物理结构（低频）

要点

一. 逻辑结构

- 元素之间的逻辑关系称为**逻辑结构**。数据元素之间的逻辑结构有以下 4 种基本类型。

1. **集合结构**：数据元素之间没有任何特殊关系（顺序/层次或连接），它们仅仅是“同属于一个集合”。
2. **线性结构**：数据元素按照一定的顺序排列，每个元素（除了第一个和最后一个）都有一个前驱和后继。
3. **树形结构**：数据元素之间存在一对多的层次关系。有一个唯一的根结点，其他结点分为若干层级。每个结点可以有多个子结点，但只有一个父结点。
4. **图形结构或网状结构**：数据元素之间存在多对多的关系。数据元素可以用结点表示，关系可以用边表示。图可以是有向的（有向图）或无向的（无向图）。

类型	关系类型	典型数据结构	应用场景
集合结构	无关系	集合	数据分类、松散数据的管理
线性结构	一对一	数组、链表、栈、队列	顺序处理、有序存储的数据
树形结构	一对多	树、二叉树、B 树	层次结构、快速查找
图形结构	多对多	图、邻接矩阵、邻接表	网络关系、路径规划

二. 物理结构

- 物理结构是逻辑结构在计算机内存中的具体存储和实现方式。
- 物理结构的设计重点在于：如何存储数据元素，以及如何表示数据元素之间的逻辑关系。
- 物理结构需要体现逻辑结构的逻辑关系。
- 根据逻辑结构在计算机中的表示和实现方式的不同，物理结构可以分为以下四种类型：

顺序存储结构	定义	将数据元素存放在地址连续的存储单元中。
	特点	数据之间的逻辑关系和物理关系是一致的。每个数据元素的存储位置可以通过下标直接计算得到。操作速度快，但可能存在存储空间浪费或扩展困难的问题。
	典型结构	数组、顺序表
	应用场景	数据大小固定，随机访问需求高
链式存储结构	定义	将数据元素存放在任意存储单元中，存储单元可以是连续的，也可以是不连续的。
	特点	数据元素的存储位置不能直接反映逻辑关系。通过指针（存放关联数据元素的地址）来表示数据之间的逻辑关系。插入、删除操作效率高，但随机访问效率低。
	典型结构	单链表、双链表、循环链表。
	应用场景	动态数据存储，插入删除频繁
索引存储结构	定义	在存储结点信息的同时，还建立附加的索引表来标识结点的存储地址。
	特点	索引表由若干索引项组成，通过索引号确定结点的存储地址索引存储增加了额外的存储空间开销，但查找效率较高。
	典型结构	数据库索引
	应用场景	快速定位数据，查找频繁的场景
散列存储结构	定义	通过哈希函数将数据元素的关键码映射到存储地址，建立数据元素存储位置与关键码之间的对应关系。
	特点	查找速度快，操作效率高。可能会出现冲突，需要采用冲突解决策略。
	典型结构	哈希表、散列表
	应用场景	快速查找，缓存设计，字典存储

时间/空间复杂度（高频）

要点

一. 时间复杂度

将算法中基本操作的执行次数作为算法时间复杂度的度量。

1. 概念

- 时间复杂度：算法中所有语句的频度（执行次数）之和。
- 一般使用最深层循环内的语句中原操作的执行频度（重复执行次数）来表示。
- **最好/最坏时间复杂度**：考虑输入数据“最好”/“最坏”的情况。
- **平均时间复杂度**：考虑所有输入数据等概率出现的情况。

2. 规则

I. 可以只考虑阶数高的部分。

II. 问题规模足够大时，常数项系数可以忽略。

III. 加法规则：多项相加，只保留最高阶的项，且系数变为 1。

IV. 乘法规则：多项相乘，都保留。

V. $O(1)$：常量时间阶；$O(n)$：线性时间阶；$O(log_2 n)$：对数时间阶；$O(nlog_2 n)$：线性对数时间阶。

VI. $O(1) < O(log_2 n) < O(n) < O(nlog_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

任意底数的对数复杂度都是同阶的，例如$O(log_2 n) = O(log_3 n)$。

二. 空间复杂度

- 空间复杂度：计算机中运行时所需存储空间大小的量度。
- 存储空间包括：指令/常数/变量占用空间；输入数据占用空间；辅助存储空间。
- 算法的空间复杂度一般指辅助空间大小。
- 当空间复杂度是常量时，空间复杂度为 $O(1)$，表示原地工作。

例：一维数组 $a[n]$：空间复杂度 $O(n)$；二维数组 $a[n][m]$：空间复杂度 $O(n \times m)$

例子

`int n = 3;  //①`
`for(int i = 0; i < n; i++){`
`  cout << 1;  //②`
`  for(int j = 0; j < n; j++){`
`  cout << 2;  //③`
`  }`
`}`
	左边代码中，
	语句①的执行次数为 1 次，时间复杂度为$O(1)$。
	语句②的执行次数为 n 次，时间复杂度为$O(n)$。
	语句③的执行次数为 n^2 次，时间复杂度为$O(n^2)$。

`void Func(int N, int M){`
`  for(int k = 0; k < M; ++k)`
`  ++count; //①`
`  for(int k = 0; k < N ; ++k)`
`  ++count; //②`
`}`
	左边代码中，
	语句①的执行次数为 M 次，时间复杂度为$O(M)$。
	语句②的执行次数为 N 次，时间复杂度为$O(N)$。
	时间复杂度为 $O(N+M)$。
	若 N 远大于 M，则为 $O(N)$。

<table>
<tr>
<td>

```
int m = 0, i, j;
for(i=1; i <= n; i *= 2)
   for(j=1; j <= i; j++)
       m++;
```

</td>
<td>
分析外层循环：
①找问题规模 n：在条件 $i <= n$ 中，则问题规模为 n。
②找循环控制变量 i：i 初始化为 1，循环次数为 x，则第 x 次时，$i = 2^{x-1}$。
③建立循环次数与问题规模的联系：因循环条件为 $i <= n$，所以建立关系为 $i = n$，即 $2^{x-1} = n$，解得 $x = log_2 n + 1$。
所以时间复杂度为 $O(log_2 n)$。
分析内层循环：
同理，问题规模为 i，$i = 2^{x-1}$。循环控制变量为 j，增量为 1，则第 y 次时，$y = \sum_{j=1}^{2^i} 1$。
综合分析取数量级：
$$\sum_{i=0}^{log_2 n} \sum_{j=1}^{2^i} 1 = \sum_{i=0}^{log_2 n} 2^i = 1 + 2 + 4 + \ldots + 2^{log_2 n} = 2n - 1$$
综上，时间复杂度为 $O(n)$。
</td>
</tr>
<tr>
<td>

```
int func(int n){
    int i = 0, sum = 0;
    while(sum < n)
        sum += ++i;
    return i;
}
```

</td>
<td>
简单算一下 i 与 sum 的变化过程：
<table>
<tr><td>i</td><td>1</td><td>2</td><td>…</td><td>t</td></tr>
<tr><td>sum</td><td>$0+1$</td><td>$0+1+2$</td><td>…</td><td>$0+1+2+\ldots+t$</td></tr>
</table>
$sum < n$，有
$$0 + 1 + 2 + \ldots + t = \frac{t(1+t)}{2} < n \rightarrow \frac{t^2}{2} < n \rightarrow t < \sqrt{2n}$$
解得时间复杂度为 $O(n^{1/2})$
</td>
</tr>
</table>

考法

历年：选择题 9 次、算法题 11 次

考法一：选择题给定一个程序，分析程序的时间复杂度。

考法二：算法题中分析我们自己写的算法的时间/空间复杂度。

选择题或者算法一小问（2 分），考试中主要包括循环、递归等形式来考察时间复杂度。
考频比较高，但难度不大，可以找几个例子自行练习体会。

线性表的顺序存储

要点

一. 顺序表的定义

- 用一组地址连续的存储单元依次存储线性表中的具有相同特性的数据元素，使得**逻辑上相邻的两个元素在物理位置上也相邻**。可以根据首元素地址和元素序号随机存取表中任一元素。
- 除首尾元素外，顺序表中的每个元素都有一个唯一的前驱和一个唯一的后继。
- 注意：线性表是一种逻辑结构，表示元素之间一对一的相邻关系。顺序表和链表是指存储结构。

二. 结构体定义（必会，算法题写出来一般 2 分）

```
#define MAX_SIZE 10            //会静态数组即可，动态数组一般不需要
typedef struct {
    ElemType data[MAX_SIZE];  //存放顺序表元素的数组
    int length;               //存放顺序表的长度
} SqList;                     //顺序表类型的定义
```

三. 基本操作

1. 插入操作（平均时间复杂度$O(N)$）

第 i 个位置插入新元素 e。先判断 i 位置是否合法，将第 i 个元素及其后所有元素依次往后移动一个位置，空位置插入新元素 e，顺序表长度增加 1。**先后移，再放入，从后往前（否则会覆盖），平均时间复杂度为 O(n)。**

```
bool ListInsert(SqList *L, int i, int e) {
    if(i<1||i>L->length+1) return false;
    if(L->length >= MAX_SIZE) return false;
    for (int j = L->length;j>= i; j--)
            L->data[j] = L->data[j-1];
        L->data[i-1] = e;
    L->length++;
    return true;
}
```

2. 删除操作（平均时间复杂度$O(N)$）

删除第 i 个位置的元素，用引用变量 e 返回。判断 i 的位置合法性，将被删元素赋给引用变量 e，并将第 $i+1$ 个元素及其后的所有元素依次往前移动一个位置，表长减 1。**平均时间复杂度为 O(n)。**

```
bool ListDelete(SqList *L, int i, int *e) {
    if(L->length == 0) return false;
    if( i < 1 || i> L->length) return false;
    e = L->data[i-1];
    for (int j = i; j< L->length; j++)
        L->data[j-1]= L->data[j];
    L->length--;
    return true;
}
```

3. 按位/按值查找

按位：顺序表支持**随机访问**，只需给出下标，即可读取该元素的值，时间复杂度为 $O(1)$。

按值：从头遍历，找到则返回对应数组下标加 1（若下标从 0 开始），按值 e 查找，主要运算是比较操作，比较的次数与值 e 在表中的位置有关，也与表长有关，平均比较次数为 $(n+1)/2$，平均时间复杂度 $O(n)$。

```
int LocateElem(SqList L, int e) {
    //在顺序表 L 中查找值为 e 的数据元素，返回其序号。
    for(int i=0; i<L.length;i++)
        if(L.data[i]==e) return i+1;  //查找成功
    return 0;       //查找失败，返回 0。
}
```

考法

近十年（2016-43 算法题、2018-41 算法题、2020-41 算法题）历年共考过 7 次算法题

考法：考察顺序表时，一般是对数组操作（手写代码、时间/空间复杂度）

查找——顺序遍历查找、**折半查找**

排序——**快速排序**、**归并排序**

题型为选择题和算法题。利用数组进行随机访问计算地址、线性表算法题的设计和实现

顺序表和链表对比

比较	顺序表	链表
存储分配方式	一次性分配	多次分配
存储密度	等于 1	小于 1
存取方式	随机存取	只能顺序存取
插入删除	$O(n)$	$O(1)$
按位查找	$O(1)$	$O(n)$

链表总结

	优点	缺点
单链表	插入删除方便	存储密度小，无法随机存取
双链表	相比单链表更方便访问前驱结点	存储密度更小，无法随机存取
循环链表	可以循环访问	没有明确的链表尾，单次遍历循环时需小心设置结束条件
静态链表	可以在没有指针的语言中表示链表，同时有链表的优点	静态链表虽然使用数组实现（物理上连续），但逻辑上仍是链式结构，不支持$O(1)$的随机访问，访问元素仍需遍历。

线性表的链式存储

要点

一. 单链表的定义

- 使相邻的元素不必存储在相邻的内存空间中，**逻辑上相邻元素在物理位置不要求相邻。**
- 每个结点只包含一个指针域，指向其直接后继。
- 头结点：可以不存储任何数据信息，或者存储单链表的长度等附加信息。

二. 结构体定义（必会，算法题写出来一般 2 分）

```
typedef struct LNode {
    ElemType data;          //data 中存放结点数据域
    struct LNode *next;       //指向后继结点的指针
}LNode, *LinkList;          //定义单链表结点类型
```

三. 基本操作

1. 结点的创建

$//$创建新结点$LNode * s = (LNode *)malloc(sizeof(LNode));$

说明：用户分配了一片 $LNode$ 型空间（构造了一个 $LNode$ 型的结点），并定义一个名字为 s 的指针来指向这个结点，同时我们把 s 也当作这个结点的名字。这里 s 命名了两个东西：一个是结点，另一个是指向这个结点的指针。

对于此类描述：“p 指向 q”，此时 p 指代指针（因为 p 既是指针名又是结点名，但是结点不能指向结点）。又如“用函数 $free()$ 释放 p 的空间”，此时 p 指代结点（因为 p 既是指针名又是结点名，但指针变量自身所需的存储空间是系统分配的，不需要用户调用函数 $free()$ 释放，只有用户分配的存储空间才需要用户自己来释放，所以 p 指代结点）。

2. 单链表的创建（动态结构—需要时分配内存，不需要时回收）

1）头插法

先来的结点在单链表的后面，后到的结点在前面。整个单链表是**逆序**的。

每个结点插入的时间为 $O(1)$，设单链表长为 n，总的时间复杂度为 $O(n)$。

```
void createlistF (LNode *&C, int a[], int n){
    LNode *s;
    C = (LNode*)malloc(sizeof(LNode));
    C -> next = NULL;
    for (int i = 0; i < n; i++){
        s = (LNode*)malloc(sizeof(LNode));
        s -> data = a[i];
        s -> next = C -> next;  //s 所指新结点的指针域 next 指向 C 中的开始结点
        C -> next = s;   //头结点指针域 next 指向 s 结点，使 s 成为新开始结点
    }
}
```

2）尾插法

- 每次新到的结点插入到单链表的末尾。为便于插入，可设置尾指针指向末尾结点。
- 头指针指向头结点，头结点不是数据结点；尾指针指向尾结点，是数据结点。
- 注意：插入新结点后，需要把新结点作为新的尾结点。

```
void createlistR (LNode *&C, int a[], int n) {
    //s 用来指向新申请的结点，r 始终指向 C 的终端结点
    LNode *s, *r;
    //申请 C 的头结点空间
    C = (LNode*)malloc(sizeof(LNode));
    C -> next = NULL;
    r = C; //r 指向头结点，因为此时头结点就是终端结点
    //循环申请 n 个结点来接收数组 a 中元素
    for (int i = 0; i < n; ++i) {
        //s 指向新申请的结点
        s = (LNode*)malloc(sizeof(LNode));
        //用新申请的结点来接收 a 中的一个元素
        s -> data = a[i];
        r -> next = s;//用 r 来接纳新结点
        //r 指向终端结点，以便接纳下一个到来的结点
        r = r -> next;
    }
    r -> next = NULL;
}
```

3. 插入结点（先查找，p的后面插入 q，并赋值为 x）

插入操作是将值为 x 的新结点插入到单链表的第 i 个位置。先检查插入位置的合法性，然后找到待插入位置的前驱结点，即第 $i-1$ 个结点，再在其后插入新结点。

```
q->data = x;  q->next = p->next;  p->next = q;
```

4. 删除结点

删除操作是将单链表的第 i 个结点删除。先检查删除位置的合法性，然后查找表中第 $i-1$ 个结点，即被删结点的前驱结点，再将其删除。

```
q = p->next;              //令 q 指向被删除结点
p->next = q->next;        //将*q 结点从链中“断开”
free(q);                  //释放结点的存储空间
```

5. 求链长

```
while(p) { i++; p=p->next; } //每访问一个结点，计数器加 1，直到访问空结点。
```

6. 按序查找

- 单链表的存储单元是不连续的，因此无论是查找第 i 个结点，还是查找值为 e 的结点，都只能从表中第一个结点（头结点）出发，顺着指针 $next$ 域逐个往下搜索，直到找到满足的结点为止，时间复杂度为 $O(n)$。

```
while(p && j<i){ p=p->next; ++j; }
```

7. 按值查找

- 第一个结点开始，依次比较表中各结点数据域的值，若某结点数据域的值等于给定值,则返回结点指针；否则返回 NULL。

```
while(p != NULL && p->data != e) {p = p->next; }
```

四. 双链表的定义和结构体定义

双链表是链表的一种，指的是构成链表的每个结点中设立两个指针域：一个指向其直接前驱的指针域 *prior*；一个指向其直接后继的指针域 *next*。这样形成的链表中有两个方向不同的链，故称为双向链表，简称双链表。

```
/*双向链表的结构 */
typedef struct DNode {
   ElementType data;          // 数据域
   struct DNode *next;        // 指向后继结点的指针
   struct DNode *prior;       // 指向前驱结点的指针
} DNode, *DLinkList;
```

五. 双链表的基本操作

1. 双链表的插入

插入结点时，只需指定其直接前驱结点。例：将指针 S 指向的新结点插入到 p 结点之后。
关键点： 钩链时必须按“先右后左”的顺序操作，以防止断链。

```
S=(DNode *)malloc(sizeof(DNode));     // 分配新结点空间
S->data = e;                          // 新结点数据赋值
S->next = p->next;                    // 新结点指向后继
p->next->prior = S;                       // 后继结点的前驱指向新结点
p->next =S;                           // 前驱结点指向新结点
S->prior = p;                         // 新结点的前驱指向前驱结点
```

2. 双链表的删除

删除某结点时，只需指定其直接前驱结点。例：将 p 结点的后继结点删除。

```
q = p->next;              // 获取待删除的结点
p->next = q->next;        // 前驱结点直接指向后继结点
q->next->prior = p;       // 后继结点的前驱指向前驱结点
free(q);                  // 释放待删除结点空间
```

考法

近十年（2015-41、2016-1/2、2019-41、2021-1、2023-2）
历年：小题 6 次、算法题 4 次、应用题 1 次

1. 要会用 C 语言定义链表结点结构体
2. 插入删除某个结点（选择常考）
3. 灵活使用头插法、尾插法（算法/选择）。如逆置单链表

栈和队列的基本概念

要点

栈和队列都属于操作受限的线性表（在线性表的基础上增加限制）。

一. 栈的基本概念

只能在同一端进行插入、删除操作的线性表。**“先进后出”** 或 **“后进先出”**

●**栈顶：** 允许插入/删除的一端，用栈顶指针来指示栈顶的位置。

●**栈底：** 固定不变的一端。

某个元素出栈时，只要是在其后出栈且早于它进栈的元素，那么这些元素(包括该元素)的出栈顺序与它的进栈顺序相反。

二. 队列的基本概念

只允许在一端进行插入，在另一端进行删除的线性表。**“先进先出”**

●**队头：** 允许删除的一端。一般用指针 $front$ 来指示。

●**队尾：** 允许插入的一端。一般用指针 $rear$ 来指示。

考法

近十年（2016-3、2017-2、2018-2、2019-42、2020-2、2022-2）

历年：小题 9 次、大题 1 次

●**考法一：** 给出栈的输入（输出）序列，找不可能的输出（输入）序列。（序列的合法性）

虽然进栈顺序是一定的，但是出栈顺序不确定，因为可以随时进行出栈操作。

●**考法二：** 队列和栈结合，如给出出队顺序，求栈的容量。

●**考法三：** 卡特兰数—对于 n 个不同元素进栈，出栈序列的个数为 $\frac{1}{n+1}C_{2n}^{n}$。

- 考频很高，多为选择题，主要考查进栈和出栈的顺序，可以自己模拟一下。
- 难度不大，送分题，灵活运用。

栈和队列的存储结构

要点

一. 栈的顺序存储（用一维数组，静态顺序栈）

1. 定义

```
#define MaxSize 50              //定义栈中元素的最大个数
typedef struct {
    Elemtype data[MaxSize];        //存放栈中元素
    int top;                    //栈顶指针
}Sqstack;
```

2. 性质

（以下默认初始化 $bottom = top = -1$） n 个空间，下标 $0 \sim n-1$。

●栈空：$bottom = top = -1$；

●进栈：先移动 $top++$，再放入元素。

（与初始化 $bottom = top = 0$；顺序相反）

$S.top++;\ S.data[S.top] = e;$ 等价于 $S.data[++S.top] = e;$

●出栈：先弹出元素，再移动 $top--$；

（出栈后元素仍保留在原存储单元，但逻辑上无法访问）

$e = S.data[S.top];\ S.top--;$ 等价于 $e = S.data[S.top--];$

●栈满：$top = n-1$；或 $top - bottom >= n$。

●栈中元素个数：$top - bottom$

注：进栈要先判断是否栈满（上溢）；出栈先判断是否栈空（下溢）。

注：$top++$与$++top$的区别：$top++$是先将 top 赋值给其他变量，top 再自增 1；$++top$ 是 top 先自增 1，再将 top 赋值给其他变量。

eg：若$a = 1$，则执行“$b = a++;$”结果为“$b = 1,\ a = 2$”；执行“$b = ++a;$”结果为“$b = 2,\ a = 2$”。

3. 共享栈：

两个栈的栈底分别设置在共享空间的两端，两栈栈顶向共享空间的中间延伸。

●栈满：$|top1 - top2| == 1$

二. 栈的链式存储

1. 定义（基于链表实现）

```
typedef struct LNode {
    int data;                  //数据域
    struct LNode *next;        //指针域
} LNode;                     //链栈结点定义
```

2. 性质/操作

●**本质**：“头进头出”的单链表，入栈为头插法，插入到头结点后面。出栈为头出法，将头结点后面的删除。

●**判空**：带头结点的链栈，其判空的条件是“$top->next == NULL;$”，不带头结点的链栈其判空的条件是“$top == NULL;$”（同单链表）

●**进栈**：（单链表头插法）申请结点 p，

$$p->data = e;\ p->next = top->next;\ top->next = p;$$

●**出栈**：（将头结点后面的结点删除）把要删的结点赋值给 p，

$$p = top->next;\ e = p->data;\ top->next = p->next;\ free(p);$$

●**进栈**：（不带头结点）$p->data = e;\ p->next = top;\ top = p;$

●**出栈**：（不带头结点）$p = top;\ top = top->next;$

三. 队列的顺序存储（一维数组，两个指针 front / rear）

1. 定义结构体

```
#define MaxSize 50              //定义队列中元素的最大个数
typedef struct {
    ElemType data[MaxSize];        //存放队列元素
    int front,rear;              //队头指针和队尾指针
}SqQueue;
```

2. 性质/操作

队头指针指向队头元素、队尾指针指向队尾元素（或者队尾元素的下一个位置）。

●**初始化**：$front == rear == 0;$

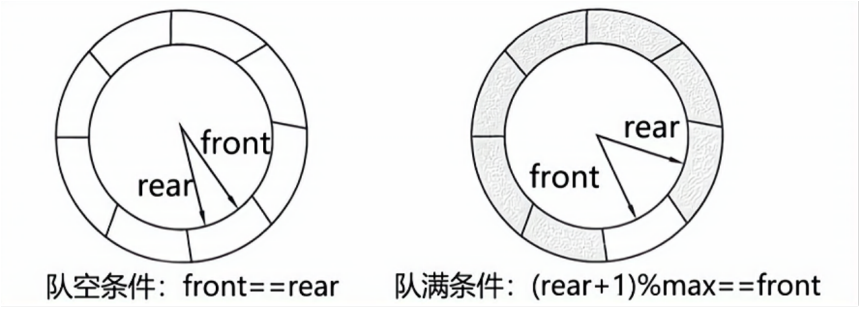
●**入队**：先放入元素，再$rear++$; **出队**：先取出元素，再$front++$;

●**队空**：$front == rear;$

●**假溢出**：因为 rear 的值永远是增加的，被使用过的下标不会被重新使用，导致队列中有空闲空间，但是不能入队。

3. 循环队列

为解决“假溢出”，把队列分配的空间看作一个首尾相接的圆环。



①$front$ 指针指向队首元素；②$rear$ 所指的单元始终为空。

③循环**队列为空**：$front == rear$。

④循环**队列满**：$(rear + 1)\%\ MAX\_SIZE == front$。

⑤**入队**操作：$rear = (rear + 1)\%\ MAX\_SIZE$

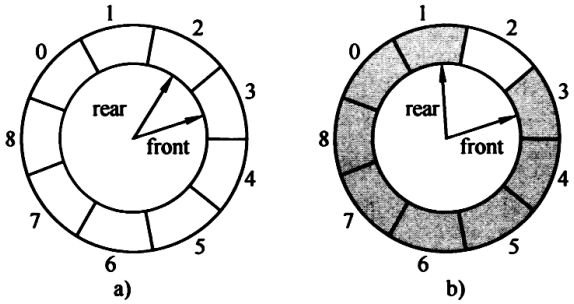
⑥**出队**操作：$front = (front + 1)\%\ MAX\_SIZE$

⑦队列中**元素个数**：$len = (rear - front + MAX\_SIZE)\%\ MAX\_SIZE$

注：关于存取元素和移动指针的次序，不同教辅写法可能不同，但本质一样。

4. 队列的非正常配置问题（408 出现过，选自 TQ）

队空：$(rear+1)\%\ MAX\_SIZE == front$
队满：$(rear+2)\%\ MAX\_SIZE == front$
入队：$rear = (rear+1)\%\ MAX\_SIZE;\ queue[rear] = x;$
出队：$x = queue[front];\ front = (front+1)\%\ MAX\_SIZE;$
元素个数：$len = (rear - front + 1 + MAX\_SIZE)\%\ MAX\_SIZE$



队空与队满示意图

四. 队列的链式存储

1. 定义

```
typedef struct{                          //链式队列结点
    ElemType data;                      //数据域
    struct LinkNode *next;              //指针域
} LinkNode;
typedef struct {                         //链式队列
    LinkNode *front,*rear;              //队列的队头和队尾指针
} LinkQueue;
```

2. 性质/操作

本质：“头出尾插”的单链表，表头进行删除操作，表尾进行插入操作。

●入队：尾插法（插入，改尾，不改头）

$$p->data = x;\ p->next = NULL;\ rear->next = p;\ rear = p;$$

●出队：（删除，改头，不改尾）

$$p = front->next;\ front->next = p->next;\ free(p);$$

●出队：（若删除最后一个元素，头尾都要改）

$$p = front->next;\ front->next = NULL;\ rear = front;\ free(p);$$

五. 双端队列

输出受限的双端队列：只允许从一端删除、两端插入的线性表。
输入受限的双端队列：只允许从一端插入、两端删除的线性表。

考法

近十年（2019-42、2021-2）历年：小题 4 次、大题 1 次（其中双端队列 选择题 3 次）

考法一：顺序栈和链栈的性质——栈空栈满入栈出栈如何操作。
考法二：链栈的基本操作，如删除/插入。**考法三**：循环队列与双端队列的操作。
多为**选择题**，考查顺序栈/队列的**性质**，**操作**及共享栈的应用。
这部分内容很多，但是考频不高，考查**循环队列**的概率比较大。

栈、队列的应用

要点

一. 栈的应用

1. 括号匹配

从左至右扫描表达式，每个右括号将与最近遇到的左括号相匹配。

把遇到的左括号放入栈中，遇到右括号，弹栈相匹配。

匹配**成功**的条件：每个检测到的括号与已检测到的优先级最高的括号都匹配。

匹配**失败**的条件：检测到与已检测到的优先级最高的括号不匹配的括号。或者扫描完整个表达式，还是有已检测到的括号没有完成匹配。

2. 表达式求值（十年 2 次）

①转化前建立两个栈，命名为 $s1$ 和 $s2$。栈 $s1$ 存储表达式中运算数，栈 $s2$ 存储符号。

②转化中，需从左到右扫描中缀表达式每个运算数和符号，若是运算数则压入栈 $s1$ 中。

③若扫描到的是符号，设符号是 C，则可能出现以下情况

a.若 $s2$ 为空，则 C 压入栈 $s2$ 中。

b.若 C 为左括号，则 C 压入栈 $s2$ 中。

c.若 $s2$ 的栈顶符号为左括号，则 C 压入栈 $s2$ 中。

d.若 $s2$ 的栈顶符号的优先级小于 C，则 C 压入栈 $s2$ 中(右括号的优先级最低)。

e.否则，就要将 $s2$ 的栈顶符号弹出，从 $s1$ 中弹出两个(也可能是 1 个)运算数，进行运算，并把运算的结果压回 $s1$。

④扫描到表达式结束，若最后栈 $s2$ 中还有符号，则将其依次出栈，并从 $s1$ 中弹出两个(也可能是 1 个)运算数，进行运算，并把运算的结果压回 $s1$。

⑤最终完成转化过程。

3. 函数和递归调用

将递归→非递归（采用非递归方式重写递归程序），借助**栈**来实现。

遵循基本准则：先调用后执行。

例，首先从$main$函数开始，在$main$函数中调用$fun1()$，在$fun1()$中调用$fun2()$，真正的执行完成的顺序是$fun2()$执行完成返回$fun1()$，$fun1()$执行完成返回$main()$。

二. 队列的应用

1. 树的层次遍历　　2. 图的广度优先遍历　　3. 排队/打印服务

考法

近十年（2015-1、2017-2、2018-1、2019-42、2024-2）

历年：小题 7 次、大题 1 次

考点一：表达式求值（2012-2 | 2014-2 | 2018-1）

考点二：队列的应用，打印/排队（2009-1，送分）

考点三：递归调用（2015-1）

考频不高，偶尔会考中缀表达式转换为后缀表达式的过程中，栈的元素变化。

近十年（2014-2、2015-1、2017-2、2018-1）

树的定义与性质

要点

一. 概念

1. **结点的度**：结点所拥有的子树的个数。
2. **树的度**：树中结点的度的最大值。
3. **叶子结点**：树中度为 0 的结点。
4. **非叶结点**：树中度不为 0 的结点。
5. **树的深度（高度）**：树中结点的最大层次值。根为第一层。
6. **路径**：树中两个结点之间所经过的结点序列。
7. **路径长度**：路径上所经过的边的条数。

二. 性质

1. 树中的结点个数 n 等于所有结点的度之和加 1。

- 除根结点外，每个节点都有与双亲唯一的边连线，因此结点数 n 等于边数之和加 1，即所有结点的度之和加 1。

2. 对于 m 度树，定义叶子结点个数为 n_0，度为 1 的结点个数为 n_1，…，度为 m 的结点个数为 n_m，于是有 $n_0 = n_2 + 2 \times n_3 + 3 \times n_4 + \cdots + (m-1) \times n_m + 1$。
3. 在非空 m 度树中，第 i 层上至多有 m^{i-1} 个结点$(i \geq 1)$。
4. **高度为 h 的 m $(m > 1)$度树，最多有 $(m^h - 1)/(m-1)$ 个结点。**
5. 具有 n 个结点的 m 度树的最小高度是 $\lceil log_m[n(m-1)+1] \rceil$。

- 为使高度最小，则前 $h-1$ 层结点数要最大，则 $(m^{h-1} - 1)/(m-1) < n \leq (m^h - 1)/(m-1)$，即 $h - 1 < log_m[n(m-1)+1] \leq h$，可以求出。

6. 具有 n 个结点的 m 度树，最大高度 h 为 $n - m + 1$。

- 树的度为 m，至少有一个结点有 m 个孩子，为使树的高度最大，其他层可仅有一个结点，则最大高度(层数)为 $n - m + 1$。也可逆推出高度为 h、度为 m 的树至少有 $h + m - 1$ 个结点。

考法

近十年（2016-5/42、2022-4）
历年：小题 3 次、大题 1 次

一、 **考法一**：树/森林边与结点的关系

- 除根结点外，每个结点都被一条边指向，一个度对应一条边，所以总结点数=总度数+1。

二、 **考法二**：树/森林中，高度/叶子结点/分支结点/总结点/度之间的关系

- 题型为**选择题**，考查相关的计算，直接掌握其中的关系，难度不大，解法灵活，实在不行列方程式求解。

二叉树的定义与性质

要点

一. 定义

1. 二叉树分为左右子树，和二度树的本质区别在于**它是有序树。**
2. 二叉树可能没有结点，称为空二叉树；也可能只有一个根结点，称为单结点二叉树。
3. 满二叉树：高度为 h 且含有 2^h-1 个结点的二叉树，每一层的结点数都达到其最大值。
4. 完全二叉树：高度为 h，且有 n 个结点的二叉树，当且仅当其每一个结点都与高度为 h 的满二叉树中编号为 $1\sim n$ 的结点一一对应。对应位置、对应编号、不跳号。

二. 二叉树性质（满足树的所有性质）

1. **对于任何一棵二叉树，若其叶子结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$**
2. **二叉树中的结点个数等于所有结点的度之和加 1。**
3. $Catalan()$ 函数：给定 n 个结点，能构成 $\frac{C_{2n}^{n}}{n+1}$ 种不同的二叉树。

三. 完全二叉树性质（满足二叉树的所有性质）

1. n 个结点的完全二叉树深度为$\lfloor log_2 n\rfloor+1$或$\lceil log_2(n+1)\rceil$。
2. 若完全二叉树的深度为 k，则所有的叶子结点都出现在第 k 层或第 $k-1$ 层。
3. 对于任一结点，若其右子树的最大层次为 j，则其左子树的最大层次为 j 或 $j+1$。
4. **对于一棵完全二叉树，度为 1 的结点个数是 1 或 0。（只有左孩子无右孩子）**
5. 若对一棵有 n 个结点的完全二叉树(深度为$\lfloor log_2 n\rfloor+1$)的结点按层(从第 1 层到第$\lfloor log_2 n\rfloor+1$ 层)序自左至右进行编号，则对于编号为 i $(1\le i\le n)$的结点，有以下情形：
   I. 若 $i=1$，则结点 i 是二叉树的根，无双亲结点；若 $i>1$，则其双亲结点编号是$\lfloor i/2\rfloor$
   II. 若 $2i>n$，则结点 i 为叶子结点，无左孩子；否则，其左孩子结点编号是 $2i$。
   III. 若 $2i+1>n$，则结点 i 无右孩子；否则，其右孩子结点编号是 $2i+1$。

考法

近十年（2017-4、2018-4、2020-3/42、2022-41）
历年：小题 5 次、大题 2 次

考法一：已知完全二叉树的结点总数，求叶结点数/分支结点数/层数/编号。
考法二：已知完全二叉树的编号，求其他信息。
考法三：已知二叉树的深度，求其他信息。

主要考完全二叉树的性质，计算题/选择题的形式。
难度不大，要求对性质做到会应用。

二叉树的遍历

要点

一. 概念

1. **核心：先序遍历：根左右，中序遍历：左根右，后序遍历：左右根。**

- 左右的顺序是不变的，都是先左后右。先中后指的是根结点何时被访问 。

2. 只通过一棵二叉树的 前序/中序/后序/层次 遍历序列，无法唯一地确定一棵二叉树。
3. 前(后)序序列和中序序列可以唯一确定一棵二叉树。
4. 前序序列和后序序列不能唯一确定一棵二叉树，但可以确定二叉树中结点的祖先关系。
5. 递归算法与非递归转换利用栈实现。

- 中序为例：从树根开始依次向左走，见到一个结点就入栈一个，直到踏空，然后栈顶结点弹栈，弹栈结点立即访问，之后沿着该结点向右走一步，然后重复前面的动作一路向左。

6. 层次遍历“自上而下，从左至右”访问各个结点，利用队列实现。

二. 二叉树遍历非递归算法

以中序遍历非递归算法为例过程：

①开始根结点入栈。

②循环执行如下操作：如果栈顶结点左孩子存在，则左孩子入栈；如果左孩子不存在，则出栈并输出栈顶结点，然后检查其右孩子是否存在，如果存在，则右孩子入栈。

③当栈空时算法结束。

```
void inorderNonrecursion (BTNode *bt) {
    if (bt != NULL){
        BTNode *stack[maxSize]; int top = -1;
        BTNode *p; p = bt;
        while (top != -1 || p != NULL) {//栈不空或 p 不空时循环
            while (p != NULL){ //左孩子存在，则左孩子入栈
                Stack[++top] = p;
                p = p->lchild;//左孩子不空，一直向左走
            }
            if(top != -1){//在栈不空的情况下出栈并输出出栈结点
                p = stack[top--];
                Visit(p);//visit()是访问 p 的函数，在这里执行打印结点值的操作
                p = p->rchild;
            }
        }
    }
}
```

考法

考法 1：由遍历序列来构造二叉树。

●中序遍历 + 层次遍历：层次遍历确定根，中序遍历确定左右子树。

●中序遍历 + 先序遍历：先序遍历确定根，中序遍历确定左右子树。

例：给出先序序列“$DAEFBCHGI$”和中序序列“$EAFDHCBGI$”，画出这棵树。

步骤：

①从先序，确定根；

②从中序，结合根，确定左右子树；

③重复①②来划分子树。

先序：根左右。中序：左根右。从先序序列可知根为 D。用 D 把后序序列分为 EAF / D / $HCBGI$，重复这一步骤来划分子数，例如 D 的左子树中包含结点 EAF，从先序序列中“AEF”可知，A 为根，E 为左结点，F 为右结点，“$HCBGI$”同理。

●中序遍历 + 后序遍历：后序遍历确定根，中序遍历确定左右子树。

●给出 树形 + 先序/后序遍历序列，求其他的遍历序列。

- 代表例题：2017-5、2023-4。先根据序列找出根，填进去，根据树形左右子树的结点数来划分左右子树即可。

考法 2：考察遍历序列的性质（下面是一些结论，灵活运用，难度不大）

1. 中序遍历中，根结点的左边有左子树的所有结点，根结点的右边有右子树的所有结点。
2. 当两个相邻结点的前序序列与后序序列顺序不同时为父子；相同为兄弟。
3. 若 a 是 b 的祖先，则后序遍历中一定先遍历 b，后遍历 a。
4. 如果二叉树的先序序列和后序序列正好相反，则其高度等于其结点树。

- 超级核心内容，选择题和大题都喜欢考。选择题主要考查遍历和二叉树的构造，大题主要考察遍历的应用，如借助遍历实现某些应用，如求解二叉树的高度、二叉树的路径。

哈夫曼树与哈夫曼编码

要点

一. 概念

(1) 结点路径：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。
(2) 路径长度：结点路径上的分支数目。
(3) 树的路径长度：从树根到每个结点的路径长度之和。
(4) 结点的带权路径长度：从该结点到树的根结点之间的路径长度与结点的权值的乘积。
(5) 树的带权路径长度：树中所有叶子结点的带权路径长度(WPL)之和

二. 哈夫曼树的构建

1. 方法

①将 n 个权值看作只有根的 n 棵二叉树；
②选出两棵权值最小树作为左右子树，构新树，权值为左右权值之和；
③重复，直到剩下一棵树为止。
潜规则：权值较小的子树放在左边。

2. 性质

(1) 为了规范，取两个最小权值树的时候，小的在左边，大的在右边。
(2) 每个初始结点最终都成为叶结点。权值越大的结点，距离根结点越近。哈夫曼树所有的元素都在叶子结点上。
(3) 每次构造都选择 2 棵树作为新结点的孩子，哈夫曼树的结点一定有兄弟。树中没有度为 1 的结点(满二叉树？)。
(4) 初始时有 n 个结点，构造过程中共新建 $n-1$ 个结点，哈夫曼树的结点总数为 $2n-1$。
(5) 哈夫曼树不唯一，左右子树可以交换，但 WPL 唯一。

三. 哈夫曼编码

1. 步骤：

把频度和次数作为权值来构建哈夫曼树，然后从根到叶子结点，每一个分支处，一般把左边标为“0”，右边标为“1”。从根到叶子的 0/1 编号就是它的哈夫曼编码

2. 性质：

(1) 一个字符的哈夫曼编码不可能是另一个字符哈夫曼编码的前缀。
(2) 根通往任一子结点的路径都不可能是通向其余叶子结点路径的子路径。
(3) 哈夫曼编码不唯一，但是是最优前缀编码。

四. m 叉哈夫曼

每个结点最多可以有 m 个子结点。（二叉哈夫曼树为特殊情况，即 $m=2$）

由 n_0 个初始结点，构造 m 叉哈夫曼树的算法描述如下：

a) 首先计算 $t=(n_0-1)\ mod\ (m-1)$。若 t 不为 0，则补充 $m-1-t$ 个权为 0 的结点到初始结点中；若 t 为 0，则无需添加结点。

b) 每次构造 1 个新的结点，并从全部结点中取出 m 个权值最小的结点，作为新结点的孩子结点，新结点的权为各孩子结点的权之和。再将新结点加入总的结点中。这一过程移除了 m 个结点，新加 1 个结点，故结点减少 $m-1$ 个。

c) 重复上一操作，直到只剩下一个结点，此时 m 叉哈夫曼树构造完毕。

考法

1. 考查哈夫曼树和哈夫曼编码的性质

(1) 哈夫曼树每个结点的度只能为 0 或 2。

(2) 每个非叶结点的权值等于其两个孩子结点的权值之和。

(3) 哈夫曼树是最优二叉树。

2. 构建哈夫曼树

略，参考上面步骤

3. 性质的计算题

结点总数为 $2n-1$（2019-3 考查，直接当结论）

求最小带权路径长度（2022-5，构造后计算）

考频很高，难度低，必会题。一般考选择题和解答题，选择题考查哈夫曼树和性质，以及计算题。解答题一般考哈夫曼树的构造和设计哈夫曼编码。

图的基本概念

一. 基本概念/术语

(1) 图的定义：图 G 由顶点集 V 和边集 E 组成，记为 $G=(V,E)$
(2) 边：顶点的无序对，记为（v，w）；弧：顶点的有序对，记为 $<v$，$w>$
(3) 度：以顶点为一个端点的边的数目；

二. 无向图

当 E 是无向边(边)的有限集合时。顶点 v 的度为依附于该顶点的边的条数。
无向完全图：任意两个顶点间都存在边，共有 $n(n-1)/2$ 条边。

三. 有向图

当 E 是有向边(弧)的有限集合时。
顶点 v 的度分为入度/出度，入度是以 v 为终点的有向边的数目，出度以 v 为起点。
有向完全图：任意两个顶点间都存在方向相反的两条弧。共 $n(n-1)$ 条有向边。

四. 子图/生成子图

图 $G=(V,E)$，$G'=(V',E')$，若 $V'\subset V$ 且 $E'\subset E$，称 G' 为 G 的子图。若 $V'=V$ 且 $E'\subset E$，称 G' 为 G 的生成子图。生成子图必须包含全部顶点，但只包含部分边。

五. 生成树/森林

连通图的生成树是包含图中全部顶点的一个极小连通子图。
在非连通图中，连通分量的生成树构成了非连通图的生成森林。

六. 路径/路径长度/回路

路径为从顶点到另一顶点的一条顶点序列。路径上边的数目称为路径长度。
第一个顶点和最后一个顶点相同的路径称为回路或环。
简单路径：在路径序列中，顶点不重复出现的路径称为简单路径。
简单回路：除第一个顶点和最后一个顶点外，其余顶点不重复出现的回路。

七. 一些结论

(1) 在无向图中，所有顶点的度之和等于边数的 2 倍，必为偶数。
(2) 在有向图中，所有顶点入度之和等于所有顶点的出度之和。
(3) 对于 n 个结点的连通图，边数最少时为 $\boldsymbol{n-1}$，最多时为 $n(n-1)/2$。
(4) 无向图共 n 个顶点，保证图在任何情况下都连通，需要最少 $(n-1)(n-2)/2+1$ 条边。
(5) 极小连通子图中的“极小”表示添加一条边就会有环。
(6) 极大连通子图：从一个顶点开始作为一个子图，逐个添加和这个子图有边相连的顶点，直到所有相连的顶点都被纳入图中，所生成的子图就是一个极大连通子图。

考法

历年：选择题 9 次、应用题 1 次、算法题 1 次
考法一：无向图中边、顶点、度之间的关系。（2017-7）；**考法二：**连通图的特性/性质（2010-7）
题型为**选择题**。难度不大，需要把相关概念理解。
主要考察基本性质和概念：度、完全图、连通分量、子图、生成子图。

图的存储及基本操作

要点

一. 邻接矩阵（顺序存储结构）—— n 个顶点

●两个数组：一维数组 $vexs[n]$ 存储顶点信息，二维数组 $A[n][n]$ 存储顶点之间的关系信息。
●以顶点在 $vexs$ 数组中的下标代表顶点，元素 $A[i][j]$ 存放顶点 i 到顶点 j 之间的关系信息。

1. 无向图（无权/有权）

$$A[i][j]=\begin{cases}1，顶点v_i到顶点v_j有边\\0，顶点v_i到顶点v_j无边\end{cases}\qquad A[i][j]=\begin{cases}W_{i,j}，顶点v_i到顶点v_j的权值\\\infty，顶点v_i到顶点v_j无边\end{cases}$$

I. 无向图的邻接矩阵是对称矩阵。只需上(下)三角矩阵即可，压缩矩阵。
II. 对于特定顶点每行或每列的非 0(或非∞)零元素的个数表示顶点的度。
III. 邻接矩阵是唯一的。

2. 有向图（无权/有权）

$$A[i][j]=\begin{cases}1，顶点v_i到顶点v_j有弧\\0，顶点v_i到顶点v_j无弧\end{cases}\qquad A[i][j]=\begin{cases}W_{i,j}，顶点v_i到顶点v_j的权值\\\infty，顶点v_i到顶点v_j无弧\end{cases}$$

I. 对于顶点 v_i，第 i 行的非 0(或非∞)元素的个数是其出度。
II. 对于顶点 v_i，第 i 列的非 0(或非∞)元素的个数是其入度。
III. 邻接矩阵中非 0(或非∞)元素的个数就是图的弧的数目
IV. 有向图的邻接矩阵也有可能是对称矩阵。
V. 邻接矩阵是唯一的。

3. 存储结构形式定义

注：如果考试中题目明确说明采用邻接矩阵表示，并要求写出定义，则以下需要全部写出；否则无需写出全部代码。

```
typedef struct {
    int no;                         //顶点编号
    char info;                      //顶点其他信息，一般是 char
}VertexType;                        //顶点类型

typedef struct {                    //图的定义
    int edges[maxsize][maxsize];    //邻接矩阵定义
    int n,e;                        //分别为顶点数和边数
    VertexType vex[maxsize];        //存放结点信息
}MGraph;                            //图的邻接矩阵类型
```

4. 性能分析

- BFS 和 DFS 使用邻接矩阵，时间复杂度 $O(|V|^2)$

- 邻接矩阵法求顶点的度出度入度的时间复杂度为 $O(|V|)$。
- 空间复杂度为 $O(|V|^2)$，只和顶点数相关，和实际边数无关。

二. 邻接表（链式存储结构）

以顶点数组中的顶点为头结点，构建一个单链表，其中单链表的数据域是指和这个顶点链接的顶点的序号，每个数组元素都构建一个单链表，称为邻接表。

1. 无向图

建立顶点数组来表示各个顶点，构成表头结点数组，并且作为每个单链表的头结点。将每个头结点与其邻接的顶点的下标依次构成一个单链表。

2. 有向图

一般构建正向邻接表：边结点表示出度信息。

3. 性质

I. 表头向量中每个分量就是一个单链表的头结点，分量个数就是图中的顶点数。

II. 在边或弧稀疏的情况下，用邻接表表示用邻接矩阵表示节省存储空间，其中无向图所占用的空间是 $n+2e$，有向图所占用的空间是 $n+e$。

III. 对于无向图，顶点 v_i 的度是第 i 个链表的结点数。

IV. 有向图第 i 个链表中的结点数是顶点 v_i 出(入)度，求入(出)度，需遍历整个邻接表

V. 邻接表不是唯一的。

4. 存储结构形式定义

```
typedef struct ArcNode{
    int adjvex;                 //该边所指向的结点的位置
    struct ArcNode *nextarc;    //指向下一条边的指针
    int info;                   //该边的相关信息(权值)
}ArcNode;

typedef struct{
    VNode adjlist[maxsize];          //邻接表
    int n,e;                    //顶点数和边数
}AGraph;                        //图的邻接表类型

typedef struct{
    char data;                  //顶点信息
    ArcNode *firstarc;          //指向第一条边的指针
}VNode;
```

三. 图的存储结构归纳

存储结构 / 比较项目		邻接矩阵		邻接表	
		无向图	有向图	无向图	有向图
空间		对称，可压缩存储至 $n(n-1)/2$ 个单元	不对称，存储 n^2 个单元	存储 $n+2e$ 个单元	存储 $n+e$ 个单元
时间	求某个顶点 v_i 的度	扫描邻接矩阵中序号 i 对应的一行，$O(1)$	求出度：扫描矩阵的一行，$O(n)$；求入度：扫描矩阵的一列，$O(n)$	扫描 v_i 的边表，最坏情况 $O(n)$	求出度：扫描 v_i的边表，最坏情况 $O(n)$；求人度：按顶点表顺序扫描所有边表，$O(n+e)$
	求边的数目	扫描邻接矩阵，$O(n^2)$		按顶点表顺序扫描所有边表，$O(n+2e)$	按顶点表顺序扫描所有边表，$O(n+e)$
	判定边 (v_i,v_j)是否存在	直接检查邻接矩阵 $A[i][j]$ 元素的值，$O(1)$		扫描 v_i 的边表，最坏情况 $O(n)$	
适用情况		稠密图		稀疏图	

	邻接矩阵	邻接表	十字链表	邻接多重表
找相邻边	遍历对应的行或列的时间复杂度为$O(\|V\|)$	找有向图的入度必须遍历整个邻接表	方便	方便
删除边 or 顶点	删除边方便；删除顶点需要大量移动数据	无向图中删除边或顶点都不方便	方便	方便
空间复杂度	$O(\|V\|^2)$	无向图$O(\|V\|+2\|E\|)$ 有向图$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$	$O(\|V\|+\|E\|)$
适用	稠密图	稀疏图和其他	有向图	无向图
表示方式	唯一	不唯一	不唯一	不唯一

考法

历年：选择题 5 次、应用题 2 次、算法题 3 次

考法一：不同存储结构某算法的时间复杂度

考法二：给出邻接矩阵/邻接表，求度

考法三：大题，画邻接表/邻接矩阵，或者构造图

题型为**选择题**和**解答题**。历年真题中本部分主要作为解答题一部分。

经常与遍历一起考，结构体的定义是必须要会的。

图的遍历

从图某一顶点出发，按照某种方法沿图中的边对图中**所有顶点访问且仅访问一次**。

一. 广度优先搜索遍历（BFS）

类似于二叉树的层序遍历，尽可能“广”地搜索一个图。

1. 步骤

任取图中一个顶点访问，入队，并将这个顶点标记为已访问；当队列不空时循环执行出队，依次检查出队顶点的所有邻接顶点，访问没有被访问过的邻接顶点并入队；当队列为空时跳出循环，广度优先搜索即完成。

2. 性能分析

BFS 算法需要**借助一个辅助队列，空间复杂度为 $O(|V|)$。**

邻接矩阵表示，时间复杂度为 $O(|V|^2)$；邻接表表示，时间复杂度为 $O(|V|+|E|)$。

二. 深度优先搜索遍历（DFS）

类似于树的先根遍历。尽可能“深”地搜索一个图。

1. 步骤

这是一个递归过程。先设计一个从某个顶点(编号)为 v_0 开始深度优先搜索的函数，便于调用。在遍历整个图时，可以对图中的每个未访问的顶点执行定义的函数。

2. 性能分析

DFS 算法是一个递归算法，要**借助一个递归栈，平均空间复杂度为 $O(|V|)$**：最好的情况下与其余结点都直接相连，空间复杂度 O(1)；最坏情况下，为一条线，空间复杂度为 $O(|V|)$

邻接矩阵表示，时间复杂度为 $O(|V|^2)$；邻接表表示，时间复杂度为 $O(|V|+|E|)$

三. 结论

I. 深度优先遍历使用栈，广度优先遍历使用队列。

II. 每个连通分量调用一次遍历算法，那么调用遍历算法的次数等于连通分量的数量。

III. **使用邻接矩阵的遍历结果是唯一的；**由于邻接表不是唯一的，**使用邻接表时遍历结果不是唯一的**，但是一旦给定邻接表，遍历结果就唯一，即必须按照邻接表的顺序遍历

IV. 使用邻接矩阵的时间复杂度是 $O(n^2)$，使用邻接表的时间复杂度是 $O(n+e)$。

考法

历年：选择题 5 次、算法题 1 次。（2012-5、2013-8、2015-5、2016-6、2020-6）

考法一：给出图，写遍历序列；

考法二：考察深度/广度优先遍历算法的时间复杂度

多为选择题和解答题。能正确分清深度/广度优先遍历的算法思想即可。时间复杂度需要会分析。

最小(代价)生成树

要点

一. 概念

生成树的代价：带权连通图的生成树的所有边的权值之和。
最小生成树：带权连通图中代价最小的生成树。
原则：尽可能选取最小的边，但不能构成回路，n 个顶点的生成树有且仅有 $n-1$ 条边。

二. 性质

最小生成树不是唯一的，即可能有多个最小生成树。
当图 G 中的各边权值互不相等时，G 的最小生成树是唯一的。
最小生成树所对应的边的权值之和总是唯一的，而且是最小的。

三. 普里姆算法

1. 算法思路

每次都选代价最小的**点**，并且子图不构成环，直到连接所有的顶点为止。

2. 算法实现

设用邻接矩阵表示图，若两个顶点之间不存在边，则其权值为无穷大。设置一个一维数组 $closedge[MAX\_EDGE]$，用来保存为选择的顶点之间权值最小的边。在这个结构体中，$lowcost$ 表示在构建最小生成树的过程中，未加入的点中顶点权值的最小值；$adivex$ 表示在构建最小生成树的过程中，未选中的点与哪个已被选中的顶点相连接。

3. 算法分析

设带权连通图有 n 个顶点，则算法的主要执行是二重循环：一是求 $closedge$ 中权值最小的边，频度为 $n-1$；二是修改 $closedge$ 数组，频度为 n。**整个算法的时间复杂度是 $O(n^2)$**，（若用邻接表存储加堆优化可达到 $O(elog_2n)$） 。运算时间与边的数目无关，所以普里姆算法**适用于稠密图**。

四. 克鲁斯卡尔算法

1. 算法思路

每次都选权值最小的**边**，使这条边的两头连通，直到所有的结点都连通。

2. 算法实现

先将各条边按权值排序，从第一条开始选，若该边加入后形成回路，则舍弃，否则选取该边。直到一共选够 $n-1$ 条为止。

3. 算法分析

将边表按权值排序，若采用堆排序或快速排序，则**时间复杂度是 $O(elog_2e)$**。时间复杂度与图的顶点个数 n 无关，克鲁斯卡尔算法适用于稀疏图，而不适用于稠密图。

五. 一些结论

当图的权值都不相等时，不同算法构造的最小生成树是唯一的，但构造过程不唯一。
当图的权值相等时，不同/同义算法最小生成树构造过程不唯一，最小生成树不唯一。
最小生成树的树形可能不同，但是最小生成树后的权值是唯一的。

考法

历年：选择题 3 次、算法题 2 次。（2012-8、2015-6、2020-7、2017-42、2018-42）

考法一：考察两个算法选边的次序（构建最小生成树的过程）

考法二：算法相关的应用题

选择和解答题都可能考。选择主要考使用普里姆算法和克鲁斯卡尔算法构造的步骤和最小生成树的性质。解答题可能考查完整的构造过程。但是代码基本不会考，大家不要怕。

简答题：什么样的图其最小生成树是唯一的？

分析：在构造最小生成树的过程中，要从剩余边中选择权值最小的并入当前生成树中，如果有多条边权值相同且同为最小值，则可任选其中一条边并入，这样就会产生多种最小生成树。要产生唯一一棵最小生成树，所有边的权值都不相同的图才能满足条件。

答：在构造最小生成树时，如果图中所有边的权值均不相等，则其最小生成树是唯一的。

思考：一定要求图中所有边权值各不相同才能使生成树唯一吗？不一定，反例：假如图 G 是一棵树（有 $n-1$ 条边的连通图），它本身就是最小生成树，即使此图中所有边权值都相等，其最小生成树也是唯一的。

对于此题，更为严谨的答案为：所有权值均不相等，或者有相等的边，但是在构造最小生成树的过程中权值相等的边都被并入生成树的图，其最小生成树唯一。

最短路径

要点

●在带权图中，边上的权值表示路径长度；从一个顶点到另一个顶点的路径长度是路径上各边的权值之和。

●**单源**最短路径问题：求从给定源点到其他各个顶点的最短路径长度。——$Dijkstra$

●**点对点**最短路径问题：求每一对顶点之间的最短路径长度。——$Floyd$

一. Dijkstra 算法

1. 算法思想

每次在集合 S 中新加入一个顶点，此顶点满足同现有顶点相连权值最小，判断从源顶点到各顶点的路径长度。

2. 算法实现

用邻接矩阵 $arcs$ 表示带权有向图

$$arcs[i][j]=\begin{cases} e_{ij} & e_{ij}\text{为有向边}<\mathrm{i,j}>\text{的权值} \\ \infty & \text{不存在有向边}<\mathrm{i,j}> \end{cases}$$

设置三个辅助数组：

●数组 $dist[\,]$ 记录了从源点 v_0 到其他各顶点当前的最短路径长度。

●数组 $pre[n]$ 保存最短路径中前一个顶点。

●数组 $final[n]$ 表示一个顶点是否已经加入 S 中。

I. 设从顶点v_0出发，集合S记录已求得的最短路径的顶点，初始为$\{0\}$，$dist[i]$初值为 $arcs[0][i]$。

II. 从顶点集合 $V-S$ 中选出 v_j，满足 $dist[j]=Min\{dist[i]\mid v_i\in V-S\}$，$v_j$ 就是当前求得的一条从 v_0 出发的最短路径的终点，令 $S=S\cup\{j\}$。

III. 修改从 v_0 出发到集合 $V-S$ 上任一顶点 v_k 可达的最短路径长度:如果 $dist[j]+arcs[j][k]<dist[k]$，则令 $dist[k]=dist[j]+arcs[j][k]$。

IV. 重复 II 和 III 操作共 $n-1$ 次，直到所有的顶点都包含在 S 中。

3. 算法分析

I. 数组变量的初始化：时间复杂度是 $O(n)$。

II. 求最短路径的二重循环，时间复杂度是 $O(n^2)$，整个算法的时间复杂度是 $O(n^2)$

4. 性质

I. 迪杰斯特拉算法不适用于有负权值的带权图，也不适用于有负回路的带权图。

II. 普里姆算法解决的问题是找到 $n-1$ 条边，使这 $n-1$ 条边的权值之和最小；而迪杰斯特拉算法是找顶点(源点)到其他所有点的最短的距离，始终要保证其他点到顶点(源点)的距离最短。

二. Floyd 算法

1. 算法思想

I. 定义一个 n 阶方阵序列：$A^{(-1)}$，$A^{(0)}$,…,$A^{(n-1)}$；

II. $A^{(-1)}[i][j] = arcs[i][j]$（$arcs[i][j]$与迪杰斯特拉算法中的定义相同）

III. 递推 $A^{k}[i][j] = Min\{A^{(k-1)}[i][j]$，$A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}$，$k = 0$，$1$，‥，$n-1$

●$A^{(k)}[i][j]$是从顶点 v_i 到 v_j、中间顶点的序号不大于 k 的最短路径的长度

●经过 n 次迭代后所得到的 $A^{(n-1)}[i][j]$ 就是 v_i 到 v_j 的最短路径长度

2. 算法分析

Floyd 算法的时间复杂度为$O(|V|^3)$。

Floyd 算法允许图中有带负权值的边，但不允许有包含带负权值的边组成的回路。

考法

历年：选择题 4 次、应用题 2 次（2009-41、2012-7、2014-42、2016-8、2021-8、2023-6）

考法一：迪杰斯特拉算法最短路径的顶点

考法二：结合计网链路查考（2014 年）

题型为**选择题**和**解答题**。主要考查**迪杰斯特拉 ($Djkstra$)** 算法求解最短路径的步骤，**弗洛伊德 ($Floyd$)** 算法的考查主要聚焦于概念和时间复杂度；解答题方面大家需要重点掌握好迪杰斯特拉 ($Dijkstra$) 算法的实例过程。

拓扑排序

要点

一. 概念

AOV 网：顶点表示活动的网，顶点表示一个活动，边表示顶点之间的前后约束关系。

有向无环图的一个顶点组成的序列称为**拓扑序列**，当且仅当满足下列条件时

I. 每个顶点出现且只出现一次。

II. 若顶点 A 在序列中排在顶点 B 的前面，则图中不存在从顶点 B 到顶点 A 的路径。

每个有向无环图都有一个或多个拓扑序列。

二. 算法

I. 从有向无环图中选择一个**没有前驱的顶点**输出

II. 从图中**删除该顶点**和所有以它为起点的有向边。

III. 重复 I 和 II 直到当前的图为空或当前图中不存在无前驱的顶点为止。而后一种情况则说明有向图中必然存在环。

三. 性能

I. 若基于***邻接表***，则拓扑排序时间复杂度为 $\boldsymbol{O}(|\boldsymbol{V}|+|\boldsymbol{E}|)$

II. 若基于***邻接矩阵***，则拓扑排序时间复杂度为 $\boldsymbol{O}(|\boldsymbol{V}|^{\boldsymbol{2}})$

四. 补充

可以用拓扑排序来检查 AOV 网中是否有环。

拓扑序列往往不唯一。

考法

历年：选择题 8 次（2010-8、2011-8、2012-6、2014-7、2016-7、2018-7、2020-6、2021-7）

考法：给出图，求可能的拓扑序列

题型为**选择题**和解答题（没考过）。

选择题主要考查拓扑排序的实例；**解答题**考查给定图，构造拓扑排序序列。

关键路径

要点

一. 概念

●**AOE 网：**边表示活动的网。
●**顶点**表示事件，每个事件表示在其前面的所有活动已经完成，其后的活动才可以开始。
●**弧**表示活动，弧上的权值表示相应活动所需的时间 or 费用。
●**关键路径：**从源点到汇点的所有路径中，具有最大路径长度的路径。
●**关键活动：**关键路径上的活动。

I. **事件的最早发生时间$ve(i)$**，即顶点v_i的最早发生时间。设v_0是起点，从v_0到v_i的最长路径长度称为事件v_i的最早发生时间，即以v_i为尾的所有活动的最早完成时间。
II. **事件的最晚发生时间$vl(i)$**，即顶点v_i的最晚发生时间，也就是每个顶点对应的事件最晚需要开始的时间，超出此时间将会延误整个工期
III. **活动的最早开始时间$e(i)$**，即弧a_i活动的最早发生时间
IV. **活动的最晚开始时间$l(i)$**，即弧a_i活动的最晚发生时间，是不推迟工期的最晚开工时间

二. 求关键路径的算法：

I. 求AOE网中所有事件的**最早发生时间$ve()$**—按**拓扑排序**序列依次确定
II. 求AOE网中所有事件的**最迟发生时间$vl()$**—按**逆拓扑排序**序列依次确定
III. 求AOE网中所有活动的**最早开始时间$e()$**—等于**弧尾事件**的**最早**发生时间
IV. 求AOE网中所有活动的**最迟开始时间$l()$**—等于**弧头事件**的**最晚**发生时间−活动时间
V. 求AOE网中所有活动的**差额$d() = l() - e()$**，找出所有$d() = 0$的活动构成关键路径。

三. 关键路径的性质

●只有在某顶点所代表事件发生后，从该顶点出发的各有向边所代表的活动才能开始；
●只有在进入某顶点各有向边所代表活动都已结束时，该顶点所代表事件才能发生。
●可以通过加快关键活动，来缩短整个工程的工期。并非关键活动缩短多少工期就缩短多少因为缩短到一定的程度，该关键活动可能变成非关键活动了。
●**关键路径并不唯一**。对于有几条关键路径的网，只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期，只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。

考法

历年：选择题 4 次、应用题 1 次（2011-41、2013-9、2019-5、2020-8、2022-7）
考法：给出 AOE 网，求其中的一些其他的属性。

大概率考选择题。

主要考察求解关键路径、关键活动、缩短工期等。难度不大，找几个例题完整的手算。

二叉排序树

要点

一. 定义

●满足以下性质的二叉树：

I. 若左子树不为空，则左子树上所有结点的值都小于根结点的值；

II. 若右子树不为空，则右子树上所有结点的值都大于根结点的值；

III. 左、右子树都分别是二叉排序树。

●中序遍历二叉排序树，可以得到一个递增序列。

二. 查找

1. 思想

●对二叉排序树进行查找也是从根结点开始。

I. 将给定的 K 值与二叉排序树的根结点的关键字进行比较：若相等，则查找成功。

II. 给定的 K 值小于 BST 的根结点的关键字：继续在该结点的左子树上进行查找。

III. 给定的 K 值大于 BST 的根结点的关键字：继续在该结点的右子树上进行查找。

2. 算法实现（递归）

```
BSTNode * BST_Search(BSTNode *T,int key) {
    if(T == NULL) return NULL; //查找失败
    else {
        if(T->key == key)
            return T;
        else if (key < T->key)
            return BST_Search(T->Lchild, key) :
        else
            return BST_Search(T 一>Rchild, key);
    }
}
```

三. 插入

在 BST 树中插入一个新结点 x 时，若 BST 树为空，则令新结点 x 为插入后 BST 树的根结点；否则，将结点 x 的关键字与根结点 T 的关键字进行比较。

I. 若相等，则不需要插入。

II. 若 $x < T->key$，则结点 x 插入到 T 的左子树中

III. 若 $x > T->key$，则结点 x 插人到 T 的右子树中。

●在二叉排序树中删除后又插入同一结点，得到的二叉排序树与原来的不一定相同，但中序序列一定相同，因为在二叉排序树中插入的新结点均作为叶子结点。

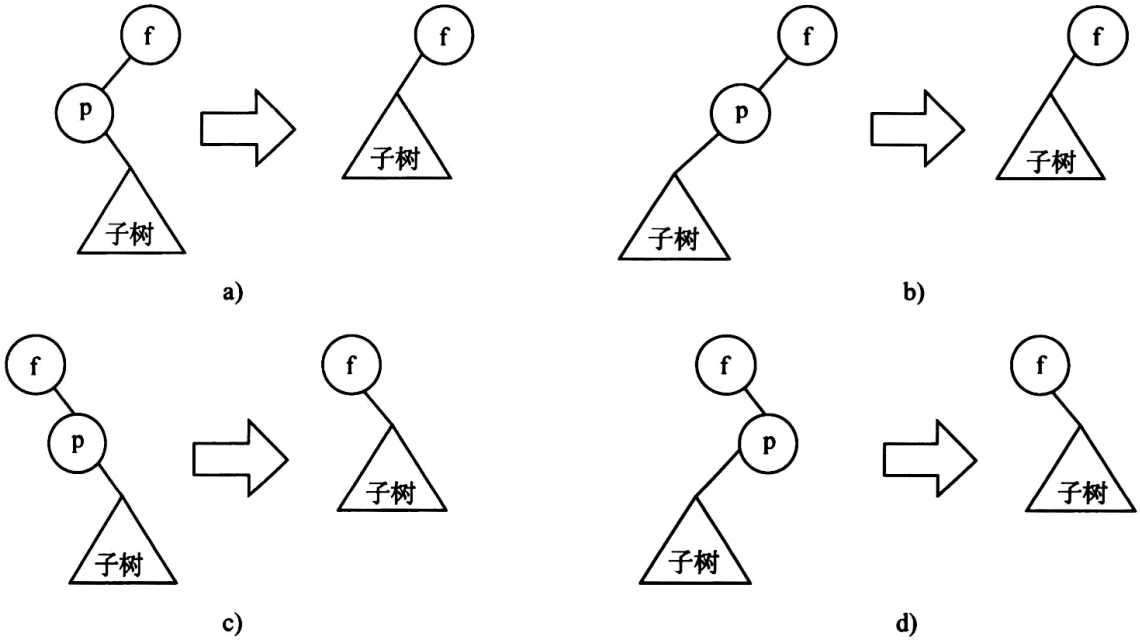
●若删除再插入的为叶子结点，则与原来的相同

四. 删除

●设被删除结点为 p，其父结点为 f

I. 若 p 是叶子结点，则直接删除 p

II. 若 p 只有一棵子树，则使用 p 的子树代替 p，作为 f 的子树。



被删结点 p 只有一棵子树的情况

III. 若 p 既有左子树又有右子树。

●用 p 的前驱/后驱结点 s 代替 p，然后删除 s，然后按照 I 和 II 来处理。

五. 查找性能分析

二叉排序树的平均查找长度和树的深度是等数量级的。

最好的情况：当二叉排序树的左、右子树的高度接近时，n 个结点的二叉树最小高度为 $log_2(n+1)$，平均查找长度为 $log_2 n$。

最坏的情况：当二叉排序树每个结点只有左子树或者右子树，n 个结点的二叉树的最大高度是 n，平均查找长度是 n。

BST 是一种查找效率比较高的组织形式，但其平均查找长度受树的形态影响较当形态比较均匀时，查找效率很好；当形态明显偏向某一方向时，其查找效率就大大降低。

考法

历年：选择题 6 次、算法题 1 次（2011-7、2013-3/6、2015-4、2018-6、2020-5、2022-41）

考法一：根据二叉排序树的性质来选择序列

考法二：二叉排序树的插入、删除、构建

考法三：给出插入序列，构造二叉排序树（大题）

考法四：基于构造的二叉排序树，计算查找成功、失败的 ASL

题型为选择题。需要掌握查找、插入、删除、构造的操作。

主要考查 BST 树和 AVL 树的插入/删除。代码部分不重要，理解为主。

平衡二叉树

一. 定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树

●左子树和右子树的深度之差的绝对值不超过 1

●左子树和右子树也都是平衡二叉树

二叉树上结点的左子树的深度减去其右子树的深度称为该结点的平衡因子。**平衡二叉树上每个结点的平衡因子只可能是 -1、0 或 1；**否则，只要有一个结点的平衡因子的绝对值大于 1，该二叉树就不是平衡二叉树。

二. 实现

I. LL 不平衡（单次右旋转）

若 a 无右子树，左子树 b 的左结点为 c，调整 b 为根结点，c 为左子树、a 为右子树。

II. RR 不平衡（单次左旋转）

若 a 无左子树，右子树 b 的右结点为 c，调整 b 为根结点，a 为左子树、c 为右子树。

III. LR 不平衡（左旋转后右旋转，共两次）

若 a 无右子树，左子树 b 的右子树结点为 c，调整 c 为根结点，b 为左子树，a 为右子树。（若 c 左子树为 d、右子树为 e，调整后，d 作为 b 的右子树，e 作为 a 的左子树）

IV. RL 不平衡（右旋转后左旋转，共两次）

若 a 无左子树，右子树 b 的左子树结点为 c，调整 c 为根结点，a 为左子树，b 为右子树。（若 c 左子树为 d、右子树为 e，调整后，d 作为 a 的右子树，e 作为 b 的左子树）

三. 查找和性质

平衡二叉排序树的查找和 BST 树一样，而平衡二叉排序树的插入和删除分两种情况

I. 如果不整的问题，需要按照调整策略进行调整涉及调整的问题，和 BST 树一致

II. 如果遇到涉及调整的问题，需要按照调整策略进行调整

平衡二叉排序树所有左、右子树的高度基本接近，含有 n 个结点的平衡二叉排序树的最大深度为 $O(log_2 n)$，平衡二叉排序树的平均查找长度为 $O(log_2 n)$

设深度为 h 的平衡二叉排序树具有的最少结点数为 N_h，由平衡二叉排序树性质知：

$$N_0 = 0，N_1 = 1，N_2 = 2，\cdots\cdots，N_h = N_{h-1} + N_{h-2} + 1$$

考法

历年：选择题 7 次（2009-4、2010-4、2012-4、2013-3、2015-4、2019-4、2021-6）

考法一：根据性质来求平衡因子、分支结点数

考法二：平衡二叉树的插入、删除

考法三：给出插入序列，画出平衡二叉树

题型为选择题。需要掌握查找、插入、删除、构造的操作。

主要考查 BST 树和 AVL 树的插入/删除。代码部分不重要，理解为主。

插入排序

要点

一. 直接插入排序

1. 算法思想

每次将一个待排序的记录按其关键字大小插入到前面有序的子序列中，直到全部插入完成，整体有序。

2. 算法代码

```
void InsertSort(int R[], int n){ // 待排数组存在 R[]中，元素个数为 n
    int i, j, temp; // 定义循环变量 i, j 以及暂存变量 temp
    for (i = 1; i < n; i++){ // 从数组的第二个元素开始，依次进行插入排序
        temp = R[i]; // 将当前待插入的元素暂存到 temp 中
        j = i - 1; // 从当前元素的前一个位置开始扫描
        // 从后向前扫描，如果 temp 比前面的元素小，则将前面的元素向后移动一位
        while (j >= 0 && temp < R[j]){
            R[j + 1] = R[j]; // 将元素向后移动一位
            --j; // 向前移动一位，继续比较
        }
        // 找到插入位置，将 temp 插入到正确的位置
        R[j + 1] = temp;
    }
}
```

3. 性能

I. **空间复杂度：**常数个辅助空间，$O(1)$。
II. **时间复杂度：**主要来自对比关键字、移动元素。若 n 个元素，需处理 $n-1$ 趟。
●**最好情况：**共 $n-1$ 趟，没趟只需对比一次，不需要移动元素，$O(n)$。
●**最坏情况：**逆序，第 i 趟，对比 $i+1$ 次，移动 $i+2$ 次，$O(n^2)$。
●**平均时间复杂度：**$O(n^2)$。
III. **稳定性：**稳定。
IV. **适用性：**顺序表、链表。

二. 折半插入排序

1. 算法思想

因前部分序列已经有序，可以使用折半查找法找出插入位置，再统一移动待插入元素后的所有元素，查找过程比较次数为 $O(nlog_2n)$。

2. 算法代码

```
void binaryInsertSort(int arr[], int n) {
    int i, j, low, high, mid;
    int temp; // 暂存待插入元素
    for (i = 1; i < n; i++) { // 第一个元素有序，从第二个元素开始循环
        temp = arr[i]; // 将 arr[i] 暂存到 temp
        low = 0;
        high = i - 1; // 设置查找区间初值
        // 折半查找插入位置
        while (low <= high) {
            mid = (low + high) / 2; // 取中间点
            if (arr[mid] > temp)
                high = mid - 1; // 插入点在左半子表
            else
                low = mid + 1; // 插入点在右半子表
        }
        // low == high + 1 时跳出循环，low 即指向第一个大于 temp 的位置
        for (j = i - 1; j >= low; --j) { // low 即时目标插入位置
            arr[j + 1] = arr[j]; // 插入位置后的数据统一后移
        }
        arr[low] = temp; // 插入正确位置
    }
}
```

3. 性能

I. **空间复杂度：**常数个辅助空间，$O(1)$。
II. **时间复杂度：**相对之下，**仅减少关键字的比较次数，没有减少记录的移动次数。最好情况：**$O(nlog_2n)$；最坏情况：$O(n^2)$；平均情况：$O(n^2)$。
III. **稳定性：**稳定。
IV. **适用性：**顺序表（随机存取）。

三. 希尔排序

1. 算法思想

先将整个待排序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。缩小增量 d，再次分别排序，待整个序列中元素基本有序，对全体元素进行一次直接插入排序。

2. 性能

I. **空间复杂度：**$O(1)$。**时间复杂度：**与增量序列有关，最坏为 $O(n^2)$，在某范围内，可达 $O(n^{1.3})$。
II. **稳定性：**不稳定。**适用性：**仅适用于顺序表。

3. 特点

- 采用希尔排序时会对数据进行分组，且多数情况下同一组内的元素不相邻，故希尔排序只能用于顺序存储结构；
- 当数据量较大时，希尔排序明显比直接插入排序快。当采用一个较好的增量序列对较少的数据进行排序时，使用希尔排序会比快速排序和堆排序更快，但是在涉及大量数据时，希尔排序相比快速排序更慢;
- 在希尔排序的最后一趟排序前，所有元素有可能都不在最终位置。

交换排序

要点

一. 冒泡排序

1. 算法思想

从后往前两两比较相邻元素的值，若为逆序，则交换它们，直到序列比较完。共需要进行 n-1 趟冒泡。若某一趟没有发生交换，则说明此时整体已经有序，所以可以使用一个变量来记录这一趟有没有进行交换。

2. 算法代码

```
void BubbleSort(int R[], int n) {
    int i, j, swap, flag;
    // 外层循环控制冒泡的次数
    for (i = 0; i < n; i++) {
        flag = 0; // flag 标记本趟是否发生了交换
        // 内层循环控制每一趟比较和交换的过程
        for (j = 1; j < n - i; j++) {
            // 如果前一个元素大于后一个元素，则交换两者
            if (R[j - 1] > R[j]) {
                swap = R[j - 1]; // 临时变量 swap 存储 R[j - 1]的值
                R[j - 1] = R[j]; // 将 R[j]的值赋给 R[j - 1]
                R[j] = swap; // 将临时变量 swap 的值赋给 R[j]
                flag = 1; // 发生交换，则 flag 改为 1
            }
        }
        // 如果本趟没有发生交换，说明数组已经有序，提前结束排序
        if (flag == 0) return;
    }
}
```

3. 性能

I. **空间复杂度：** $O(1)$

II. **时间复杂度：** 平均 $O(n^2)$

最好情况： 有序，比较 $n-1$ 次；交换 0 次。时间复杂度为 $O(n)$

最坏情况： 逆序，每次都要交换，交换次数递减，时间复杂度为 $O(n^2)$

III. **稳定性：** 稳定

4. 特点

①适用于顺序存储和链式存储的线性表。
②每趟排序会将一个数据元素放到最终位置。
③序列的初始状态会影响排序趟数、比较次数和移动次数。

二. 快速排序

通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，再分别对这两部分记录进行下一趟排序，从而达到整个序列有序。每次都选一个基准元素（一般为每段第一个元素），通过这个基准元素将待排序的元素分成两个部分。

每完成一趟排序，都会至少有一个新的元素出现在它的最终该出现的位置上。

1. 代码实现：写法一

```
int quick_one_pass(Sqlist* L, int low, int high) {
    int i = low, j = high;
    int pivotkey = L[i]; //L[0]作为临时单元和哨兵
    while(i<j) {         // i=j 时退出扫描
        while(pivotkey < L[j])&&(j>i)
            j--;
        if (j>i) { L[i]=L[j]; i++; }
        while(L[i]< pivotkey) && (j > i)
            i++;
        if (j>i) { L[j]=L[i]; j--; }
    }
    L[i] = pivotkey;
    return(i) ;
}

void quick_Sort( Sqlist* L, int low, int high) {
    int k ;
    if(low < high) {
        //序列分为两部分后,分别对每个子序列排序
        k = quick_one_pass(L, low, high) ;
        quick_Sort(L, low, k-1) ;
        quick_Sort(L, k+1, high) ;
    }
}
```

2. 代码实现：写法二

```
void quickSort(int arr[], int low, int high) {// 快速排序函数
    if (low < high) {
        int pivotloc = partition(arr, low, high);   // 对当前子数组进行划分
        quickSort(arr, low, pivotloc - 1);   // 递归地对左侧子数组进行排序
        quickSort(arr, pivotloc + 1, high);   // 递归地对右侧子数组进行排序
    }
}
int partition(int arr[], int low, int high) {// 划分函数
    int pivot = arr[low];    // 选择第一个元素作为枢轴
    while (low < high) {     // 循环执行元素交换，直到 low 和 high 指针相遇
        while (low < high && arr[high] >= pivot)//从右侧找到一个小于枢轴的元素
            --high;
        arr[low] = arr[high]; // 将较小元素移到枢轴的左侧
        while (low < high && arr[low] <= pivot) //从左侧找到一个大于枢轴的元素
            ++low;
        arr[high] = arr[low]; // 将较大元素移到枢轴的右侧
    }
    arr[low] = pivot;    // 将枢轴元素放置到最终位置
    return low;    // 返回枢轴的最终位置
}
```

3. 性能

I. **空间复杂度：**与递归层数有关。$O(log_2 n)$～$O(n)$

●**最好情况：**递归树分布平均，为 $O(log_2 n)$；最坏情况：高度 n，为 $O(n)$

II. **时间复杂度：**每一层只需处理剩余的待排序元素，复杂度小于 n。平均：$O(nlog_2 n)$

●**最好情况：**每次划分都很均匀，则递归深度最小。$O(nlog_2 n)$

●**最坏情况：**当本身有序，每次划分都不均匀，递归深度增加。$O(n^2)$

III. **稳定性：**不稳定

IV. **适用性：**顺序表

4. 特点

●每趟排序至少会将一个枢轴元素放到最终位置。

●在数据元素较多时，快速排序的平均性能更接近最好情况，因此快速排序的效率比其他排序算法更高。

●适用于顺序存储结构和元素个数较多的关键字序列。

考法

考法一：快排每一趟的顺序
考法二：快速排序的性质和特点
考法三：快排代码要会写（可能写法不同，但是一定要会一个）

选择排序、归并排序、基数排序

要点

一. 简单选择排序

1. 算法思想

每一趟在待排序元素中选取关键字最大(最小)的元素加入到有序子序列。n 个元素的简单选择排序需要 $n-1$ 趟处理。

简单选择排序的阶段性特征与冒泡排序一样，都是每轮结束都会有至少一个新的元素出现在它最终应该出现的位置上，且这些元素的位置都是从序列的一端向另一端蔓延。

2. 代码

```
void SelectSort(int R[], int n) {
    int i = 0, j, k, temp; // 定义循环变量 i、j、k 和临时变量 temp
    for (i = 0; i < n; i++) { // 外层循环，从数组第一个元素到倒数第二个元素
        k = i; // 将当前下标 i 作为最小元素的下标
        for (j = i + 1; j < n; j++){//内层循环，从下一个元素开始寻找最小值
            if (R[k] > R[j]) // 如果找到比当前最小值还小的元素
                k = j; // 更新最小值的下标为 j
        }
        temp = R[i]; // 临时存储当前位置的元素值
        R[i] = R[k]; // 将找到的最小值与当前位置的元素进行交换
        R[k] = temp; // 将临时存储的元素值放到找到的最小值的位置上
    }
}
```

2. 性能

I. **空间复杂度**：$O(1)$

II. **时间复杂度**：不管有序还是乱序，一定是需要进行 $n-1$ 趟处理，共对比 $n(n-1)/2$ 次，平均时间复杂度 $O(n^2)$

III. **稳定性**：不稳定

3. 特点

①适用于顺序存储结构和链式结构：可用于数组（顺序存储）和链表（链式存储）。

②序列的初始状态不影响比较次数，会影响移动次数：无论输入数据的初始状态如何（有序或无序），简单选择排序的比较次数总是固定的。虽然初始状态会影响数据移动的次数，但总体而言，初始序列的有序程度对简单选择排序的时间效率几乎没有影响。

③排序过程中形成有序子序列，并且每次至少都将一个数据元素放到了最终位置：在排序过程中，简单选择排序逐步形成有序子序列。每一趟排序都会将当前最小的元素放到最终位置，逐步缩小未排序部分的范围。

二. 二路归并排序

1. 算法思想

初始时，将每个记录看成一个单独的有序序列，则 n 个待排序记录就是 n 个长度为 1 的有序子序列。然后两两归并，得到 $\lceil n/2 \rceil$ 个长度为 2 或 1 的有序表；再两两归并。如此重复，直到合并成一个长度为 n 的有序表为止。

2. 代码

```
ElemType *B = (ElemType )malloc((n + 1) sizeof(ElemType)); // 辅助数组 B
void Merge(ElemType A[], int low, int mid, int high){
// 表 A 的两段 A[low...mid]和 A[mid+1...high]各自有序，将它们合并成一个有序表
    for (int k = low; k <= high; k++)
        B[k] = A[k]; // 将 A 中所有元素复制到 B 中
    for (int i = low, int j = mid + 1, int k = i; i <= mid && j <= high; k++){
        if (B[i] <= B[j]) // 比较 B 的左右两段中的元素
            A[k] = B[i++]; // 将较小值复制到 A 中
        else
            A[k] = B[j++];
    }
    while (i <= mid)
        A[k++] = B[i++]; // 若第一个表未检测完，复制
    while (j <= high)
        A[k++] = B[j++]; // 若第二个表未检测完，复制
}
```

```
void MergeSort(ElemType A[ ], int low, int high){
    if (low < high){
        int mid = (low + high) / 2;          // 从中间划分两个子序列
        MergeSort(A, low, mid);              // 对左侧子序列进行递归排序
        MergeSort(A, mid + 1, high);         // 对右侧子序列进行递归排序
        Merge(A, low, mid, high);            // 归并
    }
}
```

3. 性能

I. **空间复杂度**：$O(n)$，辅助数组。在排序过程中，使用了辅助向量，大小与待排序记录空间相同，则空间复杂度为 $O(n)$。

II. **时间复杂度**：$O(nlog_2n)$，归并趟数为$O(log_2n)$，每趟归并时间复杂度$O(n)$

III. 具有 n 个待排序记录的归并次数是 log_2n，在归并过程中，构造了棵二叉树，其树高是log_2n。而一趟归并的时间复杂度为 $O(n)$，则整个归并排序的时间复杂度无论是最好还是最坏情况均为 $O(nlog_2n)$。

IV. **稳定性**：稳定

V. **适用性**：顺序表

4. 特点

I. 时间复杂度：归并排序的时间复杂度为$O(nlog_2n)$，在最坏、最好和平均情况下都一样，这与堆排序类似。
II. 空间复杂度较高：归并排序的空间复杂度为$O(n)$，因为在归并过程中需要额外的存储空间来保存临时子序列。
III. 排序过程中元素位置的变化：在归并排序的过程中，直到最后一次合并之前，所有元素都可能不在其最终位置。这意味着在排序过程中，元素的位置可能会多次变化。
IV. 合并多个有序序列的优势：归并排序在合并多个有序序列时表现最佳。这是因为归并排序的核心操作就是合并两个有序序列，它能够高效地处理这种任务。

5. 适用性

- 归并排序适用于两种存储结构：顺序存储和链式存储的线性表。
- 顺序存储：对于顺序存储（即数组），归并排序可以直接在数组上进行操作。由于数组元素可以通过索引直接访问，因此归并排序中的分割和合并操作都可以高效地进行。
- 链式存储：对于链式存储（即链表），归并排序同样适用。归并排序在链表上的实现甚至比在数组上更为自然，因为链表的节点可以通过修改指针来实现分割和合并，不需要额外的空间来存储临时数据。归并排序对链表的优势还在于不需要随机访问，只需要顺序访问，这与链表的特性完全契合。

三. 基数排序（桶排序/数字排序）

1. 算法思想

分为最高位优先、最低位优先。设 n 个待排序记录，关键字由 d 位组成，每位 r 种取值，共需要 r 个桶，进行 d 趟排序。

2. 算法实现

I. 首先以静态链表存储 n 个待排序记录，头结点指针指向第一个记录结点。
II. 分配：按 K^d 值的升序顺序，改变记录指针，将链表中的记录结点分配到 r 个链表(桶)中，每个链表中所有记录的关键字的最低位(K^d)的值都相等，用$f[i]$、$e[i]$作为第 i 个链表的头结点和尾结点。
III. 收集：改变所有非空链表的尾结点指针，使其指向下一个非空链表的第一个结点，从而将 r 个链表中的记录重新链接成一个链表。
IV. 如此依次按K^{d-1},K^{d-2},…,K^1分别进行，共进行 d 趟排序后排序完成。

3. 性能

●**空间复杂度**：在排序过程中使用的辅助空间是 $2r$ 个链表指针，n 个指针域空间，则空间复杂度为 $O(r)$。

●**时间复杂度**：$O(d(n+r))$

设有 n 个待排序记录，关键字位数为 d，每位有 r 种取值，则排序的趟数是 d。在每趟中：链表初始化的时间复杂度为 $O(r)$；分配的时间复杂度为 (n)；分配后收集的时间复杂度为 $O(r)$，则链式基数排序的时间复杂度为 $O(d(n+r))$。

●**稳定性**：稳定

●**适用性**：既适用于顺序结构，也适用于链式结构。适用于 n 较大，但 d（数字位数）和 r（基数）较小的序列。

数据结构：互相之间存在一种或多种特定关系的数据元素集合	逻辑结构	线性结构	一般线性表
			受限线性表——栈、队列、串
			线性表推广——数组、广义表
		非线性结构	集合结构、树形结构、图状结构
	存储结构——物理结构	顺序存储	逻辑上相邻的元素存储在物理位置上也相邻的存储单元中
		链式存储	借助指示元素存储地址的指针来表示元素之间的逻辑关系
		索引存储	存储元素信息的同时，建立附加的索引表
		散列存储	根据元素的关键字直接计算出该元素的存储地址（哈希存储）

- 常见复杂度排序
  $O(1) < O(log_2 n) < O(n) < O(nlog_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$
- 分析的时候一定要注意是几层循环，是 1+2+3+4 那种，还是 1、2、3、4 那种，小心有坑！

线性表顺序表示实现

静态分配

```
typedef struct {
    ElemType data[MaxSize];
    int length;
} SqList; //顺序表类型
```

动态分配

```
typedef struct {
    ElemType *data;
    int length;
} SqList;//顺序表类型
```

```
//定义变量
SqList L;
//实现
L.data=(ElemType*)malloc(sizeof(ElemType)*MaxSize);//C 语言
L.data=new ElemType[MaxSize];//C++
```

C 语言

malloc(m)函数：开辟 m 字节长度的地址空间，并返回这段空间的首地址
sizeof(x)运算：计算变量 x 的长度
free(p)函数：释放指针 p 所指变量的存储空间

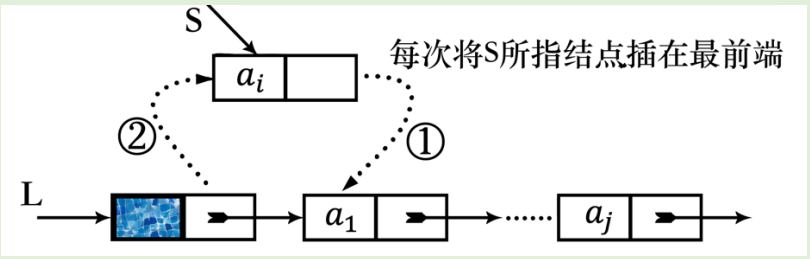
C++

new 类型名 T (初值列表)：申请存放 T 类型对象的内存空间，并初始化。
delete 指针 P：释放指针 P 所指向的内存。P 必须是 new 操作的返回值

- 顺序表插入：在顺序表的指定位置插入新元素，由于表中元素的物理位置是相邻的，所以当插入新元素的时候就需要对表中的元素进行整体移动。（先后移，再插入）
-
- 顺序表删除：删除指定位置元素，将后续元素依次向前移动一个位置（先删除，再前移）

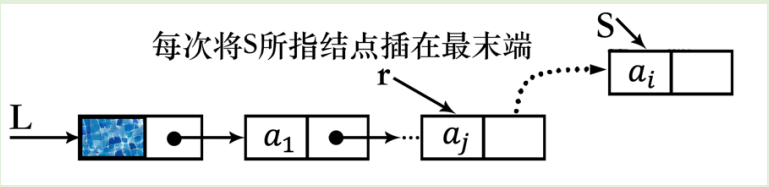
单链表

头插法




```
LNode *s = (LNode*)malloc(sizeof(LNode));    //创建新结点
s->data = x;
s->next = L->next;                  //图中①
L->next = s;                        //图中②，将新结点插入表中，L 为头指针
```

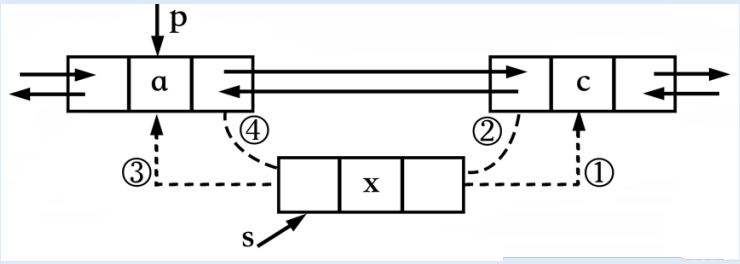
尾插法




```
LNode s = (LNode*)malloc(sizeof(LNode));
s->data = x;
r->next = s;
r = s;                              //r 指向新的表尾结点
```

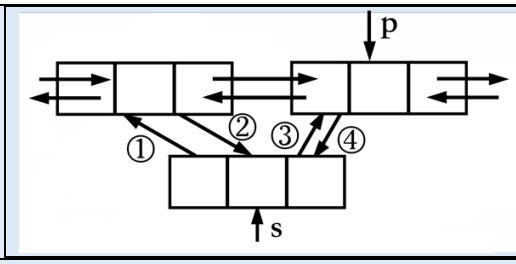
双链表

后插



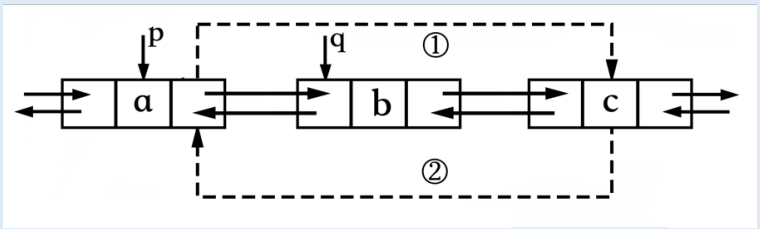

```
s->next = p->next;                  //图①，将结点*s 插入到结点*p 之后
if (p->next != NULL)
    p->next->prior = s;       //图②
s->prior = p;                       //图③
p->next = s;                        //图④
```

前插




```
s->prior = p->prior;      //图①
p->prior->next = s;       //图②
s->next = p;              //图③
p->prior = s;             //图④
```

删除




```
p->next = q->next;                  //步骤①
if(q->next != NULL)
    q->next->prior = p;           //步骤②
free(q);                            //释放结点空间
```

不同链表的时间效率比较

	查找表头结点	查找表头结点	查找结点*P 的前驱结点
带头结点的单链表 L	L->next， 时间复杂度 O(1)	从 L->next 依次向后遍历，时间复杂度 O(n)	通过 p->next 无法找到其前驱（双指针可以）
带头结点仅设头指针 L 的循环单链表	L->next， 时间复杂度 O(1)	从 L->next 依次向后遍历，时间复杂度 O(n)	通过 p->next 可以找到其前驱，时间复杂度 O(n)
带头结点仅设尾指针 R 的循环单链表	R->next， 时间复杂度 O(1)	R，时间复杂度 O(1)	通过 p->next 可以找到其前驱，时间复杂度 O(n)
带头结点的双向循环链表 L	L->next， 时间复杂度 O(1)	L->prior，时间复杂度 O(1)	p->prior，时间复杂度 O(1)

共享栈

- 让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间的中间延伸。
- 栈空判断：$top0 = -1$时 0 号栈为空，$top1 = MaxSize$时 1 号栈为空
- 栈满判断：两个栈顶指针相邻（即$top1 - top0 == 1$）时
- 其他操作和顺序栈类似

循环队列

- 初始时：$Q.front = Q.rear = 0$
- 队头指针+1：$Q.front = (Q.front + 1)\ \%\ MaxSize$
- 队尾指针+1：$Q.rear = (Q.rear + 1)\ \%\ MaxSize$
- 队列长度：$(Q.rear + MaxSize - Q.front)\ \%\ MaxSize$
- 队空：$Q.front == Q.rear$

树性质

(1)树中的结点数等于所有结点的度数之和加 1。(根入度为 0)
(2)度为 m 的树中第 i 层上至多有 m^{i-1}个结点 $(i \geq 1)$。
(3)高度为 h 的 m 叉树至多有 $(m^h - 1)/(m - 1)$个结点。(m叉树≠度m的树)
(4)具有 n 个结点的 m 叉树最小高度为 $\lceil log_m(n(m-1)+1) \rceil$。
(5)高度为 h：m 叉树至少为 h个结点；度为m的树至少有$h + m - 1$个结点(至少一个度为 m)

二叉树性质

(1)非空二叉树上的叶子结点数等于度为 2 的结点数加 1，即 $N_0 = N_2 + 1$。

(2)非空二叉树上第 k 层上至多有 2^{k-1}个结点（k >= 1）。

(3)高度为 h 的二叉树至多有 $2^h - 1$个结点（h >= 1）。

(5)有 n 个结点的完全二叉树高度为 $\lceil log_2(n+1) \rceil$或 $\lfloor log_2 n \rfloor + 1$。

(6)给定 n 个结点，能构成 $\frac{C_{2n}^{n}}{n+1}$ 种不同的二叉树。

线索二叉树

- 将结点空的指针指向其前驱和后继，这种改变指向的指针称为“线索”，加上线索的二叉树
- 标志域：用两个标志域 $ltag$、$rtag$ 来表明指向 前驱后继 or 左右结点。
- ①$ltag$：“0”，$lchild$ 指向左孩子；“1”，$lchild$ 指向前驱。
- ②$rtag$：“0”，$rchild$ 指向右孩子；“1”，$rchild$ 指向后继。

树/森林/二叉树的转换

树转换成二叉树	一棵树可以找到唯一一棵二叉树与之对应，转换以后根没有右子树。左孩子右兄弟。
二叉树转换成树	森林中每棵树 →二叉树；第一棵树 →二叉树根；第二棵树 →二叉树右子树。
森林转换成二叉树	将各棵树分别转换成二叉树；将每棵树的根结点用线相连；以第一课树根结点为二叉树的根，顺时针旋转即可
二叉树转换成森林	将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉；将孤立的二叉树还原成树

哈夫曼树

- 结点的带权路径长度 从根结点到该结点之间的路径长度与该结点的权的乘积
- 树的带权路径长度 树中所有叶子结点的带权路径长度之和
- 构造：每次从合成后存在的结点中选两个权最小的进行构造二叉树，直到所有结点均在树中
- 多叉哈夫曼树（类比外部排序的知识点），注意要补 0 结点（虚叶结点）

哈夫曼树性质

①每个初始结点最终都成为叶结点。权值越大的结点，距离根结点越近。哈夫曼树所有的元素都在叶子结点上。
②每次构造都选择 2 棵树作为新结点的孩子，哈夫曼树的结点一定有兄弟。树中没有度为 1 的结点。又叫做正则（严格）二叉树。
③树的带权路径长度最短。
④假设初始时有 n 个结点，构造过程中共新建了$n-1$个结点，哈夫曼树的结点总数为$2n-1$。
⑤在哈夫曼树中，左右孩子权值之和为父结点权值。子结点小于父结点。

B 树

①树中每个结点至多有 m 棵子树，即至多含有 m − 1个关键字，按递增排列。
②若根结点不是终端结点，则至少有两棵子树。
③除根结点外所有非叶结点至少有 $[m/2]$ 棵子树，至少含有 $[m/2]-1$ 个关键字。

B+树

①每个分支结点最多有 m 棵子树。
②非叶根结点至少有两棵子树，其他每个分支结点至少有 $[m/2]$ 棵子树。
③结点的子树个数与关键字个数相等。

④所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。
⑤为所有叶子结点增加了一个链指针。所有分支结点（可视为索引的索引）中仅包含它的各个子结点（即下一级的索引块）中关键字的最大值及指向其子结点的指针。

散列表

①**查找成功时的平均查找长度：**找到表中已有表项的平均比较次数。
②**查找不成功平均查找长度：**在表中找不到待查表项，但找到插入位置的平均比较次数。
③装填因子是关键字个数和表长度的比值，反映哈希表空间装满程度及表空间利用效率。如关键字序列 6 个，装填因子1/2，表长为12。
④散列表的平均查找长度依赖于散列表的装填因子 α，而不直接依赖于 n。α 越大，表示装填的记录越“满”，发生冲突的可能性越大，反之发生冲突的可能性越小。
⑤失败与成功的查找长度的分母意义不同，失败时，分母是哈希表长度模的值；成功时，分母是元素个数。

红黑树性质

①每个结点是红色 or 黑色的。
②根结点是黑色的，叶结点(NIL结点)都是黑色的。
③不存在两个相邻的红结点（红结点父和孩子是黑色)。
④对每个结点，从该结点到任一叶结点的简单路径上，所含黑结点的数量相同（简称黑高）
⑤从根到叶结点的最长路径不大于最短路径的 2 倍。

快排代码

```
void QuickSort(int R[], int low, int high){
    int i = low, j = high, temp;
    if (low < high){
        temp = R[low];
        while (i < j){ // 完成一躺，把小于 temp 的放左边，大于 temp 的放右边
            while (j > i && R[j] > temp)
                --j; // 空在左边，先从右往左扫描，找到一个小于 temp 的关键字
            if (j > i){
                R[i] = R[j]; // 放 temp 左边
                ++i;         // i 右移一位
            }
            while (i < j && R[i] < temp)++i;
               //从左往右扫描，找到一个大于 temp 的关键字
            if (i < j){
                R[j] = R[i];      // 放 temp 右边
                j--; // j 左移一位
            }
        }
        R[i] = R[low];              // 将 temp 放最终位置
        QuickSort(R, low, i - 1);  // 递归对 temp 左边关键字排序
        QuickSort(R, i + 1, high); // 递归对 temp 右边关键字排序
    }
}
```

类别	排序方法	时间复杂度			空间复杂度	稳定性
		最好	平均	最坏		
插入类	直接插入	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	稳定
	折半插入	$O(N \cdot log_2 N)$	$O(N^2)$	$O(N^2)$	$O(1)$	稳定
	希尔排序	$O(N)$	$O(N^{1.3})$	$O(N^2)$	$O(1)$	不稳定
选择类	直接选择	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	
	堆排序	$O(N \cdot log_2 N)$	$O(N \cdot log_2 N)$	$O(N \cdot log_2 N)$	$O(1)$	
交换类	冒泡排序	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	稳定
	快速排序	$O(N \cdot log_2 N)$	$O(N \cdot log_2 N)$	$O(N^2)$	$O(log_2 N)$~ $O(N)$	不稳定
归并排序		$O(N \cdot log_2 N)$	$O(N \cdot log_2 N)$	$O(N \cdot log_2 N)$	$O(N)$	稳定
基数排序		$O(d(r+n))$	$O(d(r+n))$	$O(d(r+n))$	$O(rd+n))$	稳定

- 当记录本身信息量较大时，为避免耗费大量时间移动记录，可用链式存储结构；**希尔排序和堆排序都利用了顺序存储的随机访问特性，而链式存储不支持。**
- **经过一趟排序，能够保证一个关键字到达最终位置，这样的排序是交换类（起泡、快速）和选择类（简单选择、堆）。**
- **算法关键字比较次数和原始序列无关——简单选择、折半插入排序、基数、归并排序。**
- **算法的排序趟数和原始序列有关——冒泡排序、快速排序、堆排序。**