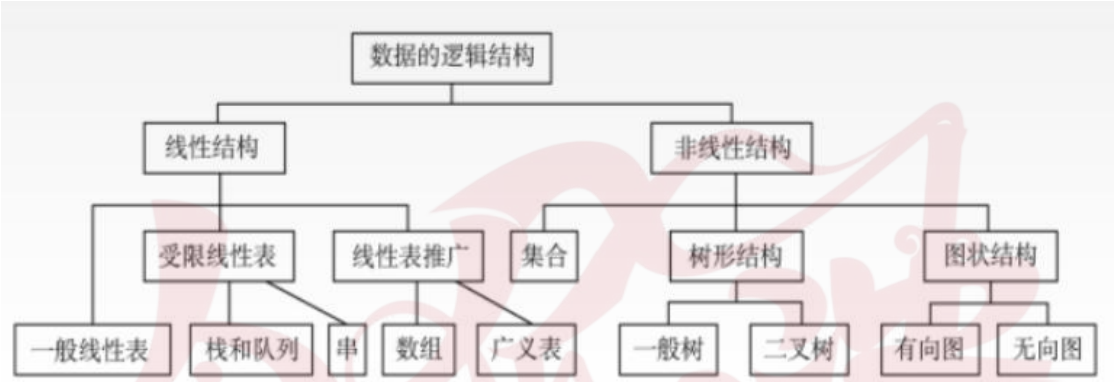
数据结构的定义

在任何问题中，数据元素都不是孤立存在的，而是在它们之间存在着某种关系，这种数据元素相互之间的关系称为结构（structure）。数据结构是相互之间存在一种或多种特定关系的数据元素的集合。数据结构包括三方面的内容：逻辑结构、存储结构和数据的运算。数据的逻辑结构和存储结构是密不可分的两个方面，一个算法的设计取决于所选定的逻辑结构，而算法的实现依赖于所采用的存储结构。

逻辑结构

数据元素之间的逻辑关系，即从逻辑关系上描述数据。它与数据的存储无关，是独立于计算机的



集合：结构中的数据元素之间除了“同属于一个集合”的关系外，别无其他关系。
类似于数学上的集合
线性结构 结构中的数据元素之间只存在一对一的关系。比如排队
树形结构 结构中的数据元素之间存在一对多的关系。比如家族族谱
图状结构或网状结构 结构中的数据元素之间存在多对多的关系。 比如地图

物理结构

数据结构在计算机中的表示（又称映像），也称物理结构。它包括数据元素的表示和关系的表示。数据的存储结构是逻辑结构用计算机语言的实现，它依赖于计算机语言。数据的存储结构主要有：顺序存储、链式存储、索引存储和散列存储。
顺序存储：存储的物理位置相邻。
链接存储：存储的物理位置未必相邻，通过记录相邻元素的物理位置来找到相邻元素。
索引存储：类似于目录。
散列存储：通过关键字直接计算出元素的物理地址。

算法的特征

有穷性：有限步之后结束

确定性：不存在二义性，即没有歧义

可行性：比如受限于计算机的计算能力，有些算法虽然理论上可行，但实际上无法完成。

输入：能被计算机处理的各种类型数据，如数字，音频，图像等等。

输出：一至多个程序输出结果。

算法的复杂度

时间复杂度：

- 衡量算法随着问题规模增大，算法执行时间增长的快慢；
- 是问题规模的函数：T(n)是时间规模函数 时间复杂度主要分析T(n)的数量级
- T(n)=O(f(n)) f(n)是算法中基本运算的频度 一般我们考虑最坏情况下的时间复杂度

空间复杂度：

- 衡量算法随着问题规模增大，算法所需空间增长的快慢；
- 是问题规模的函数：s(n)=O(g(n)) ；算法所需空间的增长率和g(n)的增长率相同。

常用的时间复杂度大小关系：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$$

历年较高考频：复杂度的计算

```
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += i;
}
```

共执行语句：3n + 4

假设每个语句执行时间一致，则总用时T = (3n + 4)t

则时间复杂度T(n) = O(3n + 4) = O(n)

去除常数项，去除常系数，只保留最高次项

[2011年] 设n 是描述问题规模的非负整数，下列程序段的时间复杂度是(　).

```
x = 2;
while(x < n/2)
    x = 2*x;
```

A. $O(\log_2 n)$

B. $O(n)$

C. $O(n\log_2 n)$

D. $O(n^2)$

[2012年] 求整数 n（$n \geqslant 0$）的阶乘的算法如下，其时间复杂度是（　）.

```
int fact( int n) {
    if (n<=1) return 1;
    return n*fact(n-1);
}
```

A. $O(\log n)$

B. $O(n)$

C. $O(n\log n)$

D. $O(n^2)$

[2022年] 下列程序段的时间复杂度是（　）.

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

A. $O(\log_2 n)$

B. $O(n)$

C. $O(n\log_2 n)$

D. $O(n)$

线性表

逻辑结构

定义：线性表是具有相同数据类型的 n（n≥0）个数据元素的有限序列。其中 n 为表长。当 n=0 时 线性表是一个空表

特点：线性表中第一个元素称为表头元素；最后一个元素称为表尾元素。除第一个元素外，每个元素有且仅有一个直接前驱。 除最后一个元素外，每个元素有且仅有一个直接后继。

顺序表

线性表的顺序存储又称为顺序表。 它是用一组地址连续的存储单元，依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻
建立顺序表:

1. 存储空间的起始位置（数组名 data）
2. 顺序表当前的长度（length）

特点：

1. 随机访问，即通过首地址和元素序号可以在 0(1)的时间内找到指定的元素。
2. 存储密度高，每个结点只存储数据元素。无需给表中元素花费空间建立它们之间的逻辑关系
3. 插入和删除操作需要移动大量元素。

顺序表插入

1. 判断 i 的值是否正确
2. 判断表长是否超过数组长度
3. 从后向前到第 i 个位置，分别将这些元素都向后移动一位
4. 将该元素插入位置 i 并修改表长

最好情况：在表尾插入（即 i=n+1），元素后移语句将不执行，时间复杂度为 0(1)。

最坏情况：在表头插入（即 i=1），元素后移语句将执行 n 次，时间复杂度为 0(n)。

平均情况：假设 pi（pi=1/(n+1)）是在第 i 个位置上插入 一个结点的概率，则在长度为 n 的线性表中插入一个结 点时所需移动结点的平均次数为 n/2

顺序表删除

1. 判断 i 的值是否正确
2. 取删除的元素
3. 将被删元素后面的所有元素都依次向前移动一位
4. 修改表长

最好情况：删除表尾元素（即 i=n），无须移动元素，时间复杂度为 0(1)。

最坏情况：删除表头元素（即 i=1），需要移动除第一个元素外的所有元素，时间复杂度为 0(n)。

平均情况：假设 pi(pi=1/n)是删除第 i 个位置上结点的概率，则在长度为 n 的

线性表中删除一个结点时所需移动结点的平均次数为

在下列对顺序存储的有序表（长度为 n ）实现给定操作的算法中，平均时间复杂度为 O(1) 的是（　）.

A. 查找包含指定值元素的算法

B. 插入包含指定值元素的算法

C. 删除第 i(1⩽ i⩽ n) 个元素的算法

D. 获取第 i(1⩽ i⩽ n) 个元素的算法

[2010 年] 设将 n(n>1)个整数存放到一维数组 R 中。设计一个在时间和空间两方面都尽可能高效的算法。将 R 中保存的序列循环左移 p(0<i<n)个位置，即将 R 中的数据由(X0, X1, ... Xn-1)变换为(Xp, Xp+1, ... Xn-1, X0, X1, ... Xp-1)。要求：

1) 给出算法的基本设计思想。

2) 根据设计思想，采用 C 或 C++或 Java 语言描述算法，关键之处给出注释。

3) 说明你所设计算法的时间复杂度和空间复杂度。

链表

线性表的链式存储是指通过一组任意的存储单元来存储线性表中的数据元素。

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

头指针：始终指向链表的第一个结点

头结点（哨兵结点）：是带头结点链表中的第一个结点，结点内通常不存储信息

为什么要设置头结点？

1. 插入删除更方便

2. 无论链表是否为空，其头指针是指向头结点的非空指针，因此空表和非空表的处理也就统一了。

链表的操作

1. 头插法建立单链表：将新结点插入到当前链表的表头

2. 尾插法建立单链表：将新结点插入到当前链表的表尾

3. 按序号查找结点

4. 按值查找结点

5. 插入：将值为 x 的新结点插入到单链表的第 i 个位置上

6. 删除：将单链表的第 i 个结点删除

双链表

```
typedef struct Node {
    Node *prev, *next;
    int data;
} Node;
```

1. 插入
2. 删除

循环链表&&静态链表

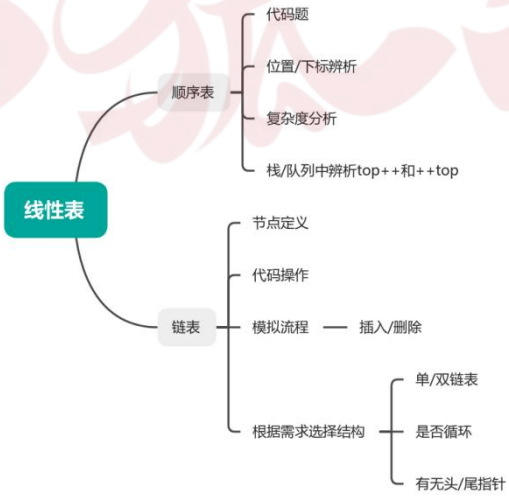
循环单链表：表中最后一个结点指向头结点，整个链表形成一个环

循环双链表：首尾结点构成环

当循环双链表为空表时，其头结点的 prior 域和 next 域都等于 Head

静态链表：用数组来描述线性表的链式存储结构

数组第一个元素不存储数据，存储第一个元素所在的数组下标，最后一个元素的指针域值为-1



已知一个带有表头结点的双向循环链表 L，结点结构为 prev | data |next ，其中 prev 和 next 分别是指向其直接前驱和直接后继结点的指针。现要删除指针 p 所指的结点，正确的语句序列是（ ）.

A. p->next->prev=p->prev; p->prev->next=p->prev; free(p);

B. p->next->prev=p->next; p->prev->next=p->next; free(p);

C. p->next->prev=p->next; p->prev->next=p->prev; free(p);

D. p->next->prev=p->prev; p->prev->next=p->next; free(p);

已知头指针 h 指向一个带头结点的非空单循环链表，结点结构为 data | next，其中 next 是指向直接后继结点的指针，p 是尾指针，q 是临时指针。现要删除该链表的第一个元素，正确的语句序列是（　）.

A. h->next = h->next->next; q = h->next; free (q);

B. q = h->next;h->next = h->next->next; free (q);

C. q = h->next; h->next = q->next; if (p!=q) p=h; free (q);

D. q = h->next; h->next = q->next; if (p == q) p = h; free (q);

现有非空双向链表 L，其结点结构为 prev | data| next，prev 是指向直接前驱结点的指针，next 是指向直接后继结点的指针。若要在 L 中指针 p 所指向的结点(非尾结点)之后插入指针　　指向的新结点，则在执行语句序列 “s->next = p->next; p->next = s;”后，下列语句序列中还需要执行的是（　）.

A. s->next->prev = p; s->prev = p;

B. p->next->prev = s; s->prev = p;

C. s->prev = s->next->prev; s->next->prev = s;

D. p->next->prev = s->prev; s->next->prev = p;

栈

只允许在一端进行插入或删除操作的线性表。

* 栈顶（Top）：线性表允许进行插入和删除的那一端。

* 栈底（Bottom）：固定的，不允许进行插入和删除的另一端

特点：

1. 栈是受限的线性表，所以自然具有线性关系。

2. 栈中元素后进去的必然先出来，即后进先出 LIFO（Last In First Out）

3. n 个不同元素进栈，出栈元素不同排序个数为（n 个元素能构成多少种不同的二叉树）

$$\frac{1}{n+1}C_{2n}^{n}$$

顺序栈

```
typedef struct {
    int data[MaxSize];
    int top;//栈顶指针
} Stack;
```

注意 top 的初值为-1 还是 0

栈的顺序存储结构也叫作顺序栈。

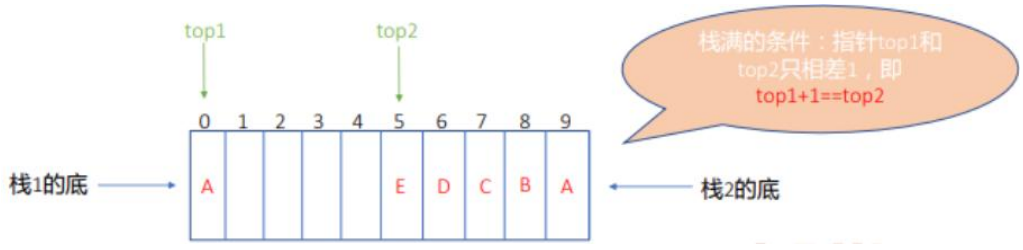
顺序栈的操作

1. 判空：empty()
2. 进栈：push(int data)
3. 出栈：pop()
4. 读取栈顶元素：top()

共享栈

顺序栈的存储空间大小需要事先开辟好，很多时候对每个栈各自单独开辟存储空间的利用率不如将各个栈的存储空间共享

入栈：push(Stack& s, int data, int stack_no)



链式栈

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

typedef struct {
    Node* top;
    int size;
} Stack;
```

用链表的方式来实现栈。栈的链式存储结构也叫作链栈。

1. 链栈一般不存在栈满的情况。

2. 空栈的判定条件通常定为 top==NULL；

链式栈的操作

1. 进栈 push(int data)
2. 出栈 pop()
3. 获取栈顶 top()
4. 判空 empty()

设栈 S 和队列 Q 的初始状态均为空，元素 abcdefg 依次进入栈 S。若每个元素出栈后立即进入队列 Q，且 7 个元素出队的顺序是 bdcfeag，则栈 S 的容量至少是（ ）.

A. 1

B. 2

C. 3

D. 4

一个栈的入栈序列为 1，2，3，...，n，出栈序列是 p1，p2，p3，... pn 若 p2 = 3，则 p3 可能取值的个数是（ ）.

A. n - 3

B. n - 2

C. n - 1

D. 无法确定

队列

队列是只允许在一端进行插入，而在另一端进行删除的线性表

* 队头（Front）：允许删除的一端，又称为队首。

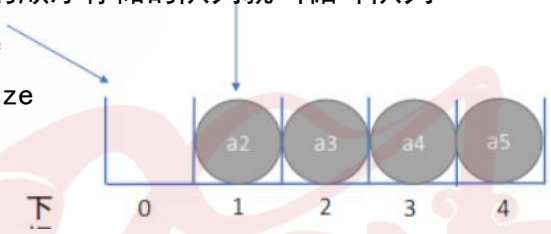
* 队尾（Rear）： 允许插入的一端。
* 先进入队列的元素必然先离开队列，即先进先出（First In First Out）简称FIFO

顺序队列：用数组来实现队列，可以将队首放在数组下标为 0 的位置。

```
typedef struct {
    int data[MaxSize];
    int front, rear;
} Queue;
```

循环队列

把数组“掰弯”，形成一个环。Rear 指针到了下标为 4 的位置还能继续指回到下标为 0 的地方。这样首尾相连的顺序存储的队列就叫循环队列



* 入队：rear=(rear+1)%MaxSize
* 出队：front=(front+1)%MaxSize
* 循环队列的操作
* 1. 入队：
* 2. 出队：

队满判断

牺牲一个单元来区分队空和队满，即队头指针在队尾指针的下一位置作为队满标志：队满条件：(Q.rear + 1) % MaxSize == Q.front

类型中增设表示元素个数的数据成员（int size）

类型中增设 tag 数据成员，以区分是队满还是队空：tag == 0 时，若因删除导致 Q.front == Q.rear，则队空，tag == 1 时，若因插入导致 Q.front == Q.rear，则队满

链式队列

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

typedef struct {
    Node *front, *rear;
} Queue;
```

链式队列的操作

1. 入队：在队尾指针进行插入结点操作。即单链表尾插

2. 出队：头结点的后继结点出队，即带头结点单链表删除第一个结点

关于选择单/双、是否循环、有无头/尾指针的链表做为链式队列的实现：能否 O(1)获取到头尾指针
例如：只有为尾指针的循环单链表就可以

双端队列

允许两端都可以进行入队和出队操作的队列，其逻辑结构仍是线性表
输出受限的双端队列：允许在一端进行插入和删除，但在另一端只允许插入的双端队列
输入受限的双端队列：允许在一端进行插入和删除，但在另一端只允许删除的双端队列
常考输出序列的分析
[2010 年] 某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作。若元素 abcde 依次入此队列后再进行出队操作，则不可能得到的出队序列是（ ）
A. bacde
B. dbace
C. dbcae
D. ecbad

[2011 年] 已知循环队列存储在一维数组 A[0…n-1] 中，且队列非空时 front 和 rear 分别指向队头元素和队尾元素. 若初始时队列为空，且要求第一个进入队列的元素存储在 A[0] 处，则初始时 front 和 rear 的值分别是（ ）.
A. 0，0
B. 0，n - 1
C. n - 1，n
D. n - 1，n - 1

[2014 年] 循环队列放在一维数组 A[0…M - 1] 中，end1 指向队头元素，end2 指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作，队列中最多能容纳 M - 1 个元素。初始时为空。下列判断队空和队满的条件中，正确的是（ ）
A. 队空： end1 == end2； 队满： end 1 == (end2 + 1) mod M
B. 队空： end1 == end2； 队满： end 2 ==(end1 + 1) mod (M-1)
C. 队空： end2 == (end1 + 1) mod M； 队满： end1 == (end2 + 1) mod M

D. 队空： end1 == (end2 + 1) mod M； 队满： end 2 == (end1 + 1) mod (M - 1)

[2019 年] 请设计一个队列，要求满足：(1) 初始时队列为空；(2) 入队时，允许增加队列占用空间；(3) 出队后，出队元素所占用的空间可重复使用，即整个队列所占用的空间只增不减；(4) 入队操作和出队操作的时间复杂度始终保持为 (1)。

请回答：

1) 该队列是应选择链式存储结构，还是应选择顺序存储结构?

2) 画出队列的初始状态，并给出判断队空和队满的条件。

3) 画出第一个元素入队后的队列状态。

4) 给出入队操作和出队操作的基本过程。

栈和队列应用——括号匹配

若是左括号，入栈；

若是右括号，出栈一个左括号判断是否与之匹配；

检验到字符串尾，还要检查栈是否为空。

只有栈空，整个字符串才是括号匹配的。

栈和队列应用——递归

如果在一个函数、过程或数据结构的定义中又应用了它自身，那么这个函数、过程或数据结构称为是递归定义的，简称递归。递归最重要的是递归式和递归边界。

```
void f(int n) {
    if (n == 1 || n == 0) // 递归出口
        return 0;
    return n * f(n - 1); // 递归式
}
```

队列的应用

脱机打印输出、解决多用户引起的资源竞争问题、广度优先遍历、网络电文传输按时间先后顺序依次进行等等

[2009 年] 为解决计算机主机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是(　　).

A. 栈

B. 队列

C. 树

D. 图

[2012 年] 已知操作符包括 “+” “-” “*” “/” “(” 和 “)”。将中缀表达式

a + b - a * ((c + d) / e - f) + g 转换为等价的后缀表达式 ab+acd+e/f-* 时，用栈来存放暂时还不能确定运算次序的操作符.栈初始时为空时，转换过程中同时保存在栈中的操作符的最大个数是（ ）.

A. 5

B. 7

C. 8

D. 11

[2015 年] 已知程序如下:

```
int S(int n) { return (n<=0) ? 0 : S(n-1) + n; }
void main() { cout << S(1); }
```

程序运行时使用栈来保存调用过程的信息,自栈底到栈顶保存的信息依次对应的是（ ）

A. main() → S(1) → S(0)

C. main() → S(0) → S(1)

B. S(0) → S(1) → main()

D. S(1) → S(0) → main()

[2017 年] 下列关于栈的叙述中，错误的是（ ）.

I. 采用非递归方式重写递归程序时必须使用栈

II. 函数调用时，系统要用栈保存必要的信息

III. 只要确定了入栈次序，即可确定出栈次序

IV. 栈是一种受限的线性表，允许在其两端进行操作

A. 仅 I

B. 仅 I、II、III

C. 仅 I、III、IV

D. 仅 II、III、IV

数组和压缩矩阵

特殊矩阵：指具有许多相同矩阵元素或 0 元素，并且这些相同矩阵元素或 0 元素的分布有一定规律

压缩矩阵：找出特殊矩阵中值相同的矩阵元素的分布规律，把那些呈现分布规律的、多个值相同的元素只分配一个存储空间，对 0 元素不分配空间

* 对称矩阵：元素关于主对角线对称
* 三角矩阵：（上/下）三角区的所有元素均为同一常量，其可压缩为存储完（下/上）三角区和主对角线上元素 + 存储常量一次
* 三对角矩阵：又称带状矩阵，除以对角线为中心的 3 条对角线区域外都为 0
* 稀疏矩阵：矩阵中非 0 元素的个数 t，相对矩阵元素的个数 s 来说非常少，即 s>>t

[2018 年] 设有一个 12×12 阶对称矩阵 M， 将其上三角部分的元素 mi, j(1 ⩽ i ⩽ j ⩽ 12)按行优先存入 C 语言的一维数组 N 中，元素 m6, 6 在 N 中的下标是（ ）.

A. 50

B. 51

C. 55

D. 66

串

串：零个或多个任意字符组成的有限序列（内容受限的线性表）

子串：一个串中任意个连续字符组成的子序列（含空串）

存储结构：顺序、链式

模式匹配：子串的定位操作通常称为串的模式匹配，它求的是子串（模式串）在主串中的位置

串的模式匹配

朴素模式：将主串中与模式串长度相同的子串逐个与模式串匹配，暴力法

时间复杂度为 O(mn)

KMP 算法：主串指针不必回溯，模式串指针根据 next 数组部分回溯（如果已匹配相等的前缀序列中有某个后缀正好是模式的前缀，可以将模式串向后滑动到与这些相等字符对齐的位置）

时间复杂度为 O(m+n)

前缀：除最后一个字符外，字符串的所有头部子串

后缀：除第一个字符外，字符串的所有尾部子串

[2015 年] 已知字符串 S 为 'abaabaabacacaabaabcc'，模式串 t 为 ' abaabc '。采用 KMP 算法进行匹配，第一次出现 “失配” （ s[i] ≠ t[j] ）时，i

= j =5，则下次开始匹配时，i 和 j 的值分别是（　）.

A. i = 1, j = 0

B. i = 5, j = 0

C. i = 5, j = 2

D. i = 6, j = 2

编号	0	1	2	3	4	5
t	a	b	a	a	b	c
pm						

编号	0	1	2	3	4	5
t	a	b	a	a	b	c
next						