初赛（二）

翁思源

信息学奥赛

wengsiyuan40@gmail.com

2026 年 8 月 26 日

今天只解决三个问题

1. **栈**：为什么只能从一端操作？怎样判断出栈序列是否合法？
2. **队列**：为什么先进先出？为什么 BFS 离不开队列？
3. **链表**：指针到底指向谁？插入、删除时怎样避免“断链”？

本节课的初赛目标

看见状态变化就会画图；看见代码就能在纸上模拟；看见四个选项就知道从哪里排除。

三种结构都在保存“一串数据”

栈：只看栈顶	队列：两端分工	链表：用箭头连接
后进先出 LIFO	先进先出 FIFO	结点可分散 顺序靠指针

不要死记定义

初赛题真正考的是：**数据从哪一端进入、从哪一端离开、下一步能访问谁。**

做状态模拟题：坚持一行一行记账

纸上至少记录三列

步骤	操作	当前状态
1	push 3	$[3]$
2	push 5	$[3,5]$
3	pop	$[3]$

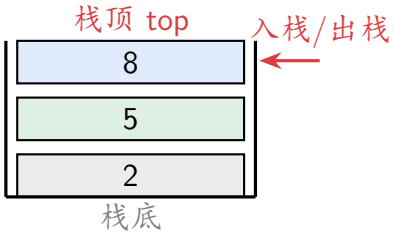
三种常见失分方式

- 凭感觉跳过中间状态；
- 把栈顶、队头画反；
- 修改指针后，仍按旧箭头思考。

先建立一张考点地图

结构	高频知识题	程序题中的信号
栈	LIFO、合法出栈序列、表达式、括号匹配	push/pop/top、递归、DFS
队列	FIFO、队头队尾、循环队列	front/pop/push、BFS、分层扩展
链表	结点结构、插入删除、单向/双向/循环	next/prev、new、指针改向

栈像一摞只能从顶部拿取的盘子



访问规则

- 最后放入的最先取出；
- 只能直接看到栈顶；
- 想拿下面的元素，必须先拿走上面的。

四个基本操作都围绕栈顶

操作	含义	C++ STL
入栈	把 x 放到栈顶	`s.push(x)`
出栈	删除栈顶元素	`s.pop()`
取栈顶	查看但不删除	`s.top()`
判空	是否没有元素	`s.empty()`
大小	当前元素个数	`s.size()`

易错点

`pop()` 只负责删除，不返回被删元素；通常先 `top()`，再 `pop()`。

阅读栈代码：先把操作翻译成中文

```
stack<int> s;
s.push(3);        // push 3
s.push(5);        // push 5
cout << s.top(); // top = 5
s.pop();          // remove 5
cout << s.top(); // top = 3
```

状态必须这样写

空栈 $\to [3] \to [3,5] \to [3]$，其中最右边表示栈顶。

判断出栈序列：不是随便排列

已知入栈顺序固定，每个元素可以在将来的某个时刻出栈。

考场模拟法

1. 目标元素若还没入栈，就继续按给定顺序入栈；
2. 目标元素若正好在栈顶，就出栈；
3. 目标元素已经在栈内、但被别的元素压住，则该序列**非法**。

一句话判死刑

“想出的元素在栈里，却不是栈顶”——无法绕过上面的元素，立即非法。

真题：哪一个出栈序列非法？

有 6 个元素，按照 $6,5,4,3,2,1$ 的顺序进入栈 S。下列哪个出栈序列非法？

A. $5,4,3,6,1,2$

B. $4,5,3,1,2,6$

C. $3,4,6,5,2,1$

D. $2,3,4,1,5,6$

先别凭感觉选

按目标序列逐项操作；只要出现“目标在栈内却被压住”，就找到了非法项。

选项 C 的矛盾出现在哪里？

目标：$3, 4, 6, 5, 2, 1$

目标	操作后栈状态（底 $\rightarrow$ 顶）	结果
3	入 $6, 5, 4, 3$，出 3：$[6, 5, 4]$	可行
4	出 4：$[6, 5]$	可行
6	栈顶是 5，6 被压在下面	**失败**

答案：C

要让 6 出栈，必须先弹出 5；但目标序列要求先 6 后 5，矛盾。

真题：再做一次合法性判断

空栈的入栈顺序为 $1,2,3,4,5,6$，哪种出栈顺序不可能？

A. $6,5,4,3,2,1$

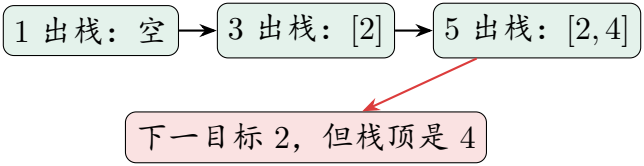
B. $1,6,5,4,3,2$

C. $2,4,6,5,3,1$

D. $1,3,5,2,4,6$

找出第一个“被压住”的目标

检查选项 D：$1, 3, 5, 2, 4, 6$



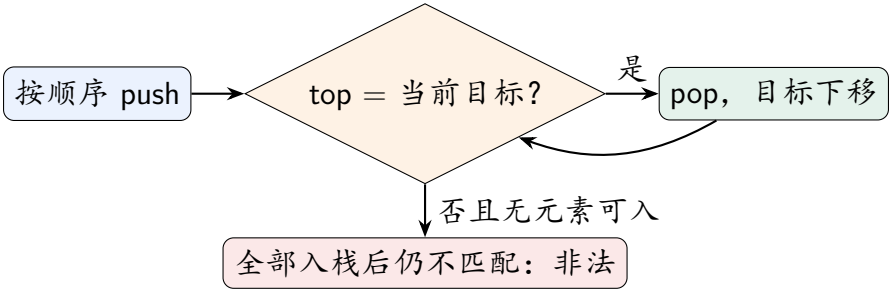
答案：D

2 已经入栈，却被后来入栈的 4 压住，不能越过 4 出栈。

合法出栈序列的通用程序模拟

模拟思想

依次压入每个元素；每压入一次，就尽可能弹出与目标序列开头相同的栈顶。



合法出栈序列：参考代码

```cpp
bool valid(vector<int> &in, vector<int> &out) {
    stack<int> s;
    int j = 0;
    for (int x : in) {
        s.push(x);
        while (j < out.size() && s.top() == out[j]) {
            s.pop();
            j++;
        }
    }
    return j == out.size();
}
```

复杂度

每个元素最多入栈、出栈各一次，因此时间复杂度为 $O(n)$。

拓展：合法序列数量是卡特兰数

当 n 个不同元素按固定顺序入栈时，合法出栈序列数为

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

n	1	2	3	4	5
C_n	1	2	5	14	42

初赛使用方式

若题目问“有多少种可能的出栈顺序”，先确认**入栈顺序固定、每个元素只进出一次**，再考虑卡特兰数。

三种表达式只是在改变运算符的位置

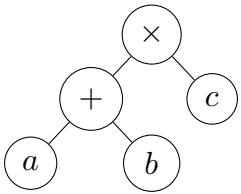
以 $(a+b)\times c$ 为例：

形式	表达式	读法
前缀	$\times\ +\ a\ b\ c$	运算符在两个操作数前
中缀	$(a+b)\times c$	人类最常用，需要优先级/括号
后缀	$a\ b\ +\ c\ \times$	运算符在两个操作数后

为什么前缀、后缀不需要括号？

运算符与操作数的组合顺序已经写进位置中，不再依赖优先级。

用表达式树统一理解三种写法



- 前缀：根、左、右（前序）
- 中缀：左、根、右（中序）
- 后缀：左、右、根（后序）

最稳的转换方法

先根据括号和优先级画出运算结构，再按要求读树。

后缀表达式求值：遇到运算符就取两个数

从左到右扫描：

1. 遇到数字：入栈；
2. 遇到运算符：弹出右操作数 b，再弹出左操作数 a；
3. 计算 $a \text{ op } b$，结果重新入栈；
4. 扫描结束后，栈中唯一的数就是答案。

减法和除法最容易写反

第一次弹出的是**右操作数**，第二次弹出的是**左操作数**。

手算示例：6 2 3 + −

读到的符号	操作	栈（底 → 顶）
6	入栈	$[6]$
2	入栈	$[6, 2]$
3	入栈	$[6, 2, 3]$
+	$2 + 3 = 5$	$[6, 5]$
−	$6 - 5 = 1$	$[1]$

对应中缀表达式

每次把“两个操作数 + 运算符”合并成一个整体：$6 - (2 + 3)$。

真题：后缀表达式对应哪个中缀式？

后缀表达式

$$6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ *\ 2\ \hat{}\ 3\ +$$

对应的中缀表达式是 ()。

A. $((6-(2+3))\times(3+8/2))^2+3$

B. $6-2+3\times3+8/2^2+3$

C. $(6-(2+3))\times((3+8/2)^2)+3$

D. $6-((2+3)\times(3+8/2))^2+3$

把每次合并结果重新压回栈

$$
\begin{aligned}
2\ 3\ + &\Rightarrow (2+3)\\
6\ (2+3)\ - &\Rightarrow 6-(2+3)\\
8\ 2\ / &\Rightarrow 8/2\\
3\ (8/2)\ + &\Rightarrow 3+8/2\\
(6-(2+3))\ (3+8/2)\ * &\Rightarrow (6-(2+3))(3+8/2)\\
\cdots\ 2\ \hat{} &\Rightarrow ((6-(2+3))(3+8/2))^2\\
\cdots\ 3\ + &\Rightarrow ((6-(2+3))(3+8/2))^2+3
\end{aligned}
$$

答案：A

这里的 ^ 表示题目中的乘方运算，不是 C++ 位运算异或。

真题：中缀式转前缀式

表达式 a + (b - c) * d 的前缀表达式为 ()。

A. * + a - b c d

B. + a * - b c d

C. a b c - d * +

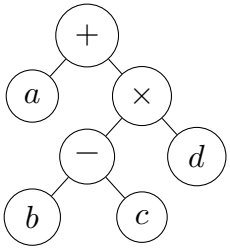
D. a b c - + d

提示

最外层运算是加法，所以前缀表达式的第一个符号一定是 +。

先分成左右两棵子树再读前序

$$a + (b - c) \times d$$



前序读取：

$$+\ a\ *\ -\ b\ c\ d$$

答案：B

写成连续字符即 `+a*-bcd`。

括号匹配也是栈：等待最近的左括号

- 遇到左括号 ([{：入栈；
- 遇到右括号：必须与当前栈顶配对，然后弹出；
- 中途无法配对，或扫描结束栈非空，都不合法。

为什么不能用计数代替？

([)] 中左右括号数量相等，但关闭顺序错误；栈能记住“最近还没关闭的括号”。

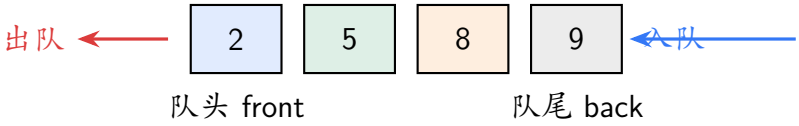
识别关键词

嵌套、撤销、最近一次、递归调用、括号匹配——优先想到栈。

栈部分小结：先找“唯一可操作的一端”

1. LIFO：最后进入的元素最先离开；
2. 合法出栈序列：目标元素被压住时立即非法；
3. 后缀求值：数字入栈，运算符弹两个；
4. 表达式转换：优先画运算结构或表达式树；
5. 栈常见于 DFS、递归、括号匹配和撤销操作。

队列像排队：先来的人先离开



两端分工

新元素从队尾进入，老元素从队头离开，因此遵循 FIFO（先进先出）。

队列的五个基本操作

操作	含义	C++ STL
入队	放到队尾	`q.push(x)`
出队	删除队头	`q.pop()`
取队头	查看最早进入者	`q.front()`
取队尾	查看最后进入者	`q.back()`
判空	是否没有元素	`q.empty()`

与栈对比

二者都有 push/pop/empty；真正区分它们的是**删除和查看发生在哪一端**。

阅读队列代码：最早进入的先输出

```
queue<int> q;
q.push(3);
q.push(5);
q.push(8);
cout << q.front(); // 3
q.pop();           // remove 3
cout << q.front(); // 5
```

状态

空队列 $\to [3] \to [3,5] \to [3,5,8] \to [5,8]$。

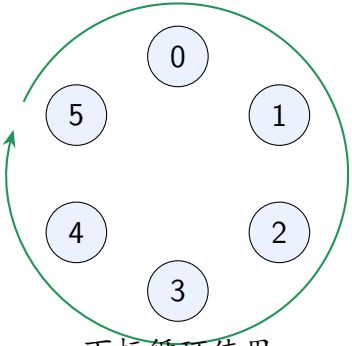
顺序队列会出现“假溢出”

用长度为 6 的数组保存队列，队头不断右移：

问题

右侧没有位置，但左侧明明空着；若只向右走，就会误认为数组已经装满。

循环队列：走到末尾就绕回开头



下标循环使用

长度为 M 时：

$$next(i) = (i + 1) \bmod M.$$

常见约定：空出一个位置

- 队空：$front = rear$
- 队满：$(rear + 1) \bmod M = front$
- 元素数：$(rear - front + M) \bmod M$

循环队列题先确认 front 和 rear 的含义

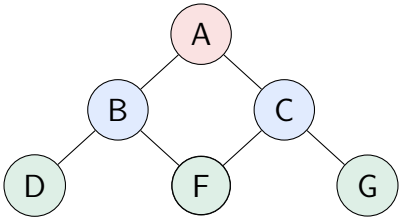
不同教材可能采用不同约定：

- **常见约定 A**：front 指向队头元素，rear 指向下一个可写位置；
- **常见约定 B**：front 指向队头前一个位置，rear 指向队尾元素。

不能只背公式

先从题目给出的入队、出队语句判断两个指针指向哪里，再套队空、队满和长度公式。

BFS 为什么使用队列？因为要按发现顺序扩展



分层顺序

第 0 层：A；第 1 层：B、C；第 2 层：D、E、F、G。先发现的点必须先扩展，正好符合 FIFO。

BFS 的固定四步循环

1. 起点入队，并立即标记已访问；
2. 取队头结点；
3. 枚举它能到达的相邻结点；
4. 对每个未访问邻点：标记并入队；最后弹出/进入下一轮。

为什么“入队时标记”？

如果等到出队才标记，同一个点可能被多个邻居重复放入队列。

真题：下面哪项表述不恰当？

A. 图的深度优先遍历常使用栈；

B. 栈后进先出，队列先进先出；

C. 队列常常用于广度优先搜索；

D. 栈与队列本质不同，因此无法用栈实现队列。

答案：D

A、B、C 都正确；两个栈可以实现一个队列，所以“无法实现”的说法错误。

来源：CSP-J 2022 第一轮，第 10 题（公开试题整理）

两个栈怎样实现队列？

入队栈 S_1

新元素都压入 S_1。

$$1,2,3 \text{ 入队} \Rightarrow S_1 = [1,2,3]$$

出队栈 S_2

S_2 为空时，把 S_1 全部倒入 S_2：

$$S_2 = [3,2,1]$$

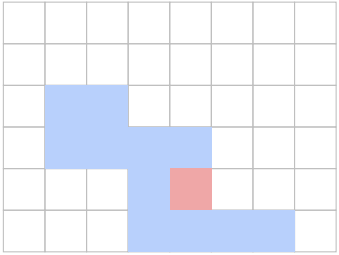
此时栈顶 1 正是最早入队者。

两次倒序得到原顺序

第一次入栈倒序，第二次转移再倒序，因此可以实现 FIFO。

真题程序：洪水填充其实就是网格 BFS

在 8×8 字符图像中，从起点出发，只沿上、下、左、右移动，把与起点颜色相同且连通的像素改成新颜色。



蓝色连通块

起点 $(4, 4)$

$\longrightarrow$ 整块改色

来源：CSP-J 2022 第一轮，完善程序“洪水填充”

先读懂判定函数：候选点必须满足什么？

```
1 bool is_valid(char image[ROWS][COLS], Point pt, int prev_color, int new_color) {
2     int r = pt.r, c = pt.c;
3     return 0 <= r && r < ROWS && 0 <= c && c < COLS &&
4            [1] && image[r][c] != new_color;
5 }
```

条件按层次拆开

坐标不越界；颜色与起点原颜色相同；尚未改成新颜色。

```
image[r][c] == prev_color
```

起点入队后，要立刻改色

```
queue<Point> q;
q.push(cur);
int prev_color = image[cur.r][cur.c];
[2];
```

： image[cur.r][cur.c] = new_color

改色同时承担“已访问标记”的作用，避免起点被邻点再次加入队列。

记忆

BFS 中通常是“**入队时标记**”，不是“出队时才标记”。

四联通邻点必须覆盖上、下、左、右

```
1 Point points[4] = {
2     [3],
3     Point(pt.r - 1, pt.c),
4     Point(pt.r, pt.c + 1),
5     Point(pt.r, pt.c - 1)
6 };
```

已有“上、右、左”，缺少“下”：

```
Point(pt.r + 1, pt.c)
```

答案

```
Point(pt.r + 1, pt.c)
```

发现合法邻点：先标记，再入队

```
for (auto p : points) {
    if (is_valid(image, p, prev_color, new_color)) {
        [4];
        [5];
    }
}
```

标记已访问

```
image[p.r][p.c] = new_color
```

等待后续扩展

```
q.push(p)
```

五个空的答案不是背出来的

空	答案	推理依据
	与原颜色相同	只扩展同一连通块
	起点改为新颜色	入队时标记
	$(r+1, c)$	补齐下方向
	邻点改为新颜色	防止重复访问
	邻点入队	将来继续扩展

原题选项答案

A， B， C， D， A。

BFS 完善程序的考场阅读顺序

1. 先看题意，确认“状态”和“可扩展关系”；
2. 找队列的入队、取队头、出队位置；
3. 找访问标记，检查是在入队时还是出队时设置；
4. 检查邻居枚举是否完整、是否越界；
5. 最后才逐个对照选项。

年份订正

这道洪水填充完善程序来自 **CSP-J 2022 第一轮**；CSP-J 2023 的完善程序不是 BFS。

队列部分小结：先发现的状态先处理

1. FIFO：队尾入队，队头出队；
2. 循环队列：先确认 front/rear 约定，再用取模；
3. BFS：队列保证按发现顺序、按层扩展；
4. 访问标记通常在入队时设置；
5. 洪水填充：边界、同色、未访问、改色、入队。

数组靠相邻位置，链表靠箭头

数组

12	7	20	5

地址连续，可通过下标直接定位。

链表

地址可分散，逻辑顺序由 next 指针决定。

单链表结点只有“数据 + 后继指针”

```
struct Node {
    int data;    // value
    Node* next; // address of next node
};
Node* head;      // first node
```

读箭头的方式

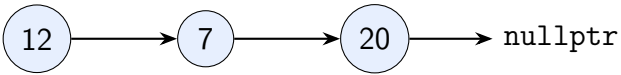
p->next 表示“p 指向的结点中，保存的下一个结点地址”。

尾结点

普通单链表的最后一个结点没有后继，通常令 next = nullptr。

遍历链表：顺着 next 一直走

```cpp
Node* p = head;
while (p != nullptr) {
    cout << p->data << ' ';
    p = p->next;
}
```

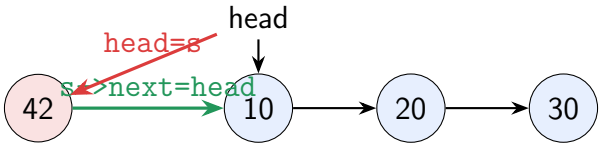


时间复杂度

链表不能直接跳到第 k 个结点；必须从头走过去，访问第 k 个结点需要 $O(k)$。

头插法：新结点必须先抓住旧链表

原链表：head $\rightarrow 10 \rightarrow 20 \rightarrow 30$



顺序不能反

若先令 head=s，就会丢失旧头结点的地址，旧链表无法再找到。

真题：怎样让新结点成为第一个结点？

已知 Node* head 指向链表头部，要插入数据为 42 的新结点并让它成为第一个结点，正确操作是 ()。

A. `newNode->data=42; newNode->next=head; head=newNode;`

B. `head->data=42; newNode->next=head; head=newNode;`

C. `newNode->data=42; head->next=newNode;`

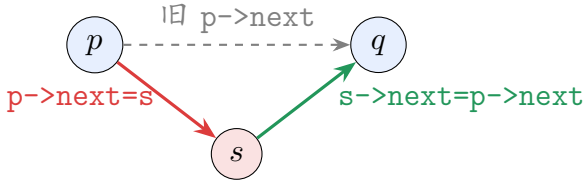
D. `newNode->data=42; newNode->next=head;`（不改 head）

答案：A

新结点先指向旧头，随后 head 再改指向新结点；两条链都不能少。

来源：CSP-J 2023 第一轮，第 4 题（代码按核心操作整理）

在结点 p 后插入 s：只改两条箭头



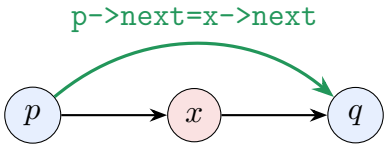
安全顺序

`s->next = p->next;` 再执行 `p->next = s;`。

核心原则

先让新结点接住后半段，再让前半段指向新结点。

删除 p 后面的结点：先保存，再跨过去



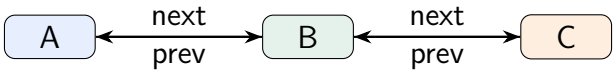
典型代码

```
Node* x = p->next;  p->next = x->next;  delete x;
```

为什么先保存 x？

修改 p->next 后，若没有变量保存被删结点地址，就无法正确释放它。

双向链表让每个结点都能向前、向后走




```
1 struct Node {
2     int data;
3     Node *prev, *next;
4 };
```

代价

多保存一个指针；插入、删除时必须同时维护两个方向的一致性。

在双向链表 p 后插入 s：四条关系都要成立

插入后应满足：$p \leftrightarrow s \leftrightarrow q$，其中 q 原来是 p 的后继。

目标箭头	语句
$s \to q$	`s->next = p->next;`
$q \to s$	`p->next->prev = s;`
$s \to p$	`s->prev = p;`
$p \to s$	`p->next = s;`

推荐顺序

先接新结点与旧后继，再接前驱与新结点；最后改 `p->next`，避免丢失 q。

真题：双向循环链表插入的正确顺序

在双向循环链表结点 p 后插入结点 s，下面哪组操作正确？

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

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

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

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

答案 D：每次修改后都没有丢地址

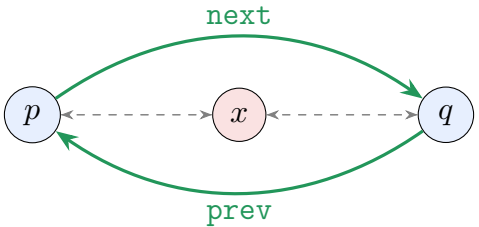
1. `s->next = p->next;`：s 先记住旧后继 q；
2. `p->next->prev = s;`：让 q 的前驱变成 s；
3. `s->prev = p;`：让 s 的前驱变成 p；
4. `p->next = s;`：最后让 p 的后继变成 s。

A、B、C 为什么错？

它们过早修改 `p->next`，后面再访问 `p->next` 时已经不是旧后继 q，会自环或改错前驱。

双向链表删除 x：让左右邻居重新牵手

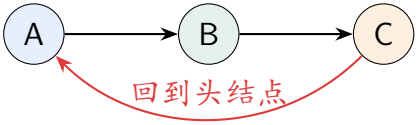
设 $p = x->prev$，$q = x->next$：




```
x->prev->next = x->next;    x->next->prev = x->prev;
```

最后再 `delete x;`。

循环链表：尾结点不再指向空



遍历停止条件改变

普通链表：走到 nullptr；
循环链表：再次回到起点。

易错点

循环链表没有空指针作为天然终点；停止条件写错会陷入死循环。

数组与链表：别把“找到位置”和“修改位置”混在一起

操作	数组	链表
按下标访问第 i 个元素	$O(1)$	$O(i)$
已知结点/位置后的插入删除	常需移动，$O(n)$	改指针，$O(1)$
存储方式	连续内存	结点可分散
额外空间	无指针域	每个结点有指针域
大小调整	普通数组固定	可动态申请结点

重要前提

链表插入删除的 $O(1)$ 是在“**已经拿到相关结点指针**”的前提下；若先查找位置，仍可能需要 $O(n)$。

真题：链表和数组的区别是什么？

链表和数组的区别包括 ()。

A. 数组不能排序，链表可以；
B. 链表比数组能存储更多的信息；
C. 数组大小固定，链表大小可动态调整；
D. 以上均正确。

答案：C

数组当然可以排序；能存多少信息取决于元素类型和内存，不能简单说链表更多。

来源：CSP-J 2022 第一轮，第 4 题（公开试题整理）

链表指针题的“四步检查法”

1. 先画修改前的结点与箭头；
2. 每执行一条语句，就擦掉旧箭头、画上新箭头；
3. 检查是否有结点失去入口、再也找不到；
4. 检查双向链表是否满足：$u->next = v$ 时，$v->prev = u$。

不要在脑中同时执行四条语句

指针语句有先后顺序；第 3 行中的 p->next 可能已经不是第 1 行时的 p->next。

链表部分小结：先接住，再改入口

1. 单链表：每个结点只有 next；
2. 头插：s->next=head，再 head=s；
3. 中间插入：新结点先接后半段；
4. 删除：左右（或前后）结点跨过被删结点；
5. 双向链表：next 与 prev 必须成对维护；
6. 循环链表：终点不是 nullptr，而是回到起点。

综合真题：栈经过队列时至少需要多大容量？

栈 S 和队列 Q 初始为空。每个 e_i 都依次经历：进栈、出栈、进队列、出队列。栈的进栈顺序为

$$e_1, e_2, e_3, e_4, e_5, e_6,$$

队列的出队顺序为

$$e_2, e_4, e_3, e_6, e_5, e_1.$$

栈 S 的容量至少是多少？

A. 2

B. 3

C. 4

D. 6

来源：CSP-J 2022 第一轮，第 5 题（题意压缩整理）

队列不改变出栈后的相对顺序

元素从栈出来后按顺序进入 FIFO 队列，因此队列的出队顺序就是栈可以产生的出栈顺序：

$$e_2, e_4, e_3, e_6, e_5, e_1.$$

目标出栈	必须先入栈	出栈后栈内
e_2	e_1, e_2	$[e_1]$
e_4	e_3, e_4	$[e_1, e_3]$
e_3	无	$[e_1]$
e_6	e_5, e_6	$[e_1, e_5]$
e_5, e_1	无	空

最大同时在栈中的元素数为 3

在准备弹出 e_4 时，栈中会出现

$$[e_1, e_3, e_4],$$

在准备弹出 e_6 时，栈中会出现

$$[e_1, e_5, e_6].$$

答案：B，容量至少为 3

既构造出了容量为 3 的合法过程，又确实出现过 3 个元素同时在栈中，所以最小容量就是 3。

一页对比：看到关键词就选对结构

	栈	队列	链表
顺序	后进先出	先进先出	由指针决定
入口/出口	同一端	两端分工	任意已知结点附近
典型信号	最近、撤销、嵌套	分层、最短步数、BFS	动态、插入删除、next
常见陷阱	被压住、左右数顺序	标记太晚、front/rear	断链、自环、双向不一致

课堂练习 1：判断结构与状态

1. 依次执行 `push(2)`，`push(7)`，`pop()`，`push(5)`，栈顶是什么？
2. 队列依次入队 $2, 7, 5$，出队一次后，队头是什么？
3. 固定入栈顺序 $1, 2, 3, 4$，序列 $3, 1, 4, 2$ 是否可能？

要求

不只写答案，必须画出每一步的栈或队列状态。

课堂练习 1：答案

1. 栈：$[] \to [2] \to [2,7] \to [2] \to [2,5]$，栈顶为 5；
2. 队列：$[] \to [2] \to [2,7] \to [2,7,5] \to [7,5]$，队头为 7；
3. 不可能。弹出 3 后栈为 $[1,2]$，下一目标 1 被 2 压住。

课堂练习 2：表达式与链表

1. 后缀表达式 8 3 2 * - 的值是多少？
2. 中缀表达式 $(a-b)/(c+d)$ 的后缀表达式是什么？
3. 在单链表结点 p 后插入 s，下面顺序是否安全？

```
p->next=s;  s->next=p->next;
```

课堂练习 2：答案

1. $3 \times 2 = 6$，再算 $8 - 6 = 2$；
2. a b - c d + /；
3. 不安全。第一句后 p->next 已经变成 s，第二句会令 s->next=s，形成自环并丢失旧后继。

正确顺序

s->next=p->next； 再执行 p->next=s;。

初赛选择题：四种快速检查

1. **栈序列**：找第一个“目标在栈内但不是栈顶”的位置；
2. **表达式**：先确定最外层运算，再画树或用栈合并；
3. **队列/BFS**：检查是否队头取出、邻点入队、入队即标记；
4. **链表**：逐句更新箭头，检查丢地址、自环和双向不一致。

考前清单：每道题都能回到“状态变化”

必须会

- LIFO 与 FIFO；
- 合法出栈模拟；
- 前中后缀互转与求值；
- BFS 队列模板逻辑；
- 单/双链表插入删除。

考场提醒

- 不跳步；
- 不凭印象判断指针；
- 先看题中结构约定；
- 遇到否定词先圈出；
- 答完用反例复核。

本节课的最终结论

栈

一端操作，处理“最近一次尚未完成”的任务。

队列

两端分工，处理“按发现顺序等待”的任务。

链表

用指针表达顺序，修改前必须先保住旧链接。

真正的通用能力

把抽象结构画成状态，把每条语句变成一次明确的状态变化。

真题来源与复习建议

本课件中的真题页面根据公开试卷整理：

- CSP-J 2022：合法出栈序列、栈与队列、双向链表、洪水填充完善程序；
- CSP-J 2023：链表头插法、后缀表达式；
- CSP-J 2024：合法出栈序列。

课后复习顺序

先闭卷重画本课的状态图，再完成真题；错题必须写出“出错的第一步”，不要只抄正确选项。