六、数据结构（链表、栈、队列、树）

（一）考点分析

结构类型	重点考点	考察频率	难度
链表	双向链表删除操作（指针顺序）	25%	★★★
	链式栈/队列插入与删除	20%	★★
	链表指针操作常见错误（断裂/环）	15%	★★★
	静态链表（数组模拟）应用	10%	★★
	链表合并与拆分技巧	5%	★★★
栈	出栈序列合法性判断	30%	★★
	链式栈操作（栈顶指针更新）	20%	★★
	栈在表达式求值中的应用	15%	★★★
	多栈共享空间问题	10%	★★★
	栈的递归调用实现	5%	★★★
队列	循环队列判满/判空条件	25%	★★
	队列假溢出问题解决	20%	★★
	双端队列操作（入队/出队方向）	15%	★★★
	队列在 BFS 中的应用	10%	★★
	优先级队列实现原理	5%	★★★
二叉树	遍历（前序/中序/后序）重建树	30%	★★
	性质计算（叶子数/深度/结点数）	25%	★★
	二叉查找树验证与操作	20%	★★★
	顺序存储结构（数组表示）	15%	★
	哈夫曼树与贪心算法应用	10%	★★★

	满 k 叉树结点与高度计算	5%	★★
表达式	中缀转后缀表达式规则	25%	★★
	中缀转前缀表达式规则	20%	★★
	后缀表达式求值过程	15%	★★★
	表达式语法树构建	10%	★★★
	运算符优先级与结合性处理	5%	★★★
递归与存储	二叉树递归遍历实现	20%	★★
	递归边界条件处理	15%	★★★
	动态内存分配陷阱（泄露/野指针）	10%	★★★
	结构体对齐与空间优化	5%	★★★

（二）核心知识点

1. 双向链表删除操作

核心知识点

双向链表删除需同时更新前驱和后继节点的指针，顺序至关重要：

- 将前驱节点的后继指向当前节点的后继
- 将后继节点的前驱指向当前节点的前驱

黄金法则

先更新邻居指针，再删除当前节点，顺序错误会导致链表断裂

高频公式

```
// 正确删除节点 p（p 非头尾节点）
p->prev->next = p->next;  // ① 前驱指向后继
p->next->prev = p->prev;  // ② 后继指向前驱
delete p;
```

易错点

- 未检查边界条件（删除头/尾节点需特殊处理）
- 错误顺序：若先执行 delete p 会导致内存访问错误
- 未更新头指针（删除头节点时）

C++模板

```
struct Node {
    int data;
    Node* prev;
    Node* next;
};
void deleteNode(Node* &head, Node* p) {
    if (p == nullptr) return;

    // 处理头节点
```

```
        if (p == head) head = p->next;

        // 更新前驱指针
        if (p->prev)
            p->prev->next = p->next;

        // 更新后继指针
        if (p->next)
            p->next->prev = p->prev;

        delete p;
    }
```

2. 栈的出栈序列合法性

核心知识点

验证出栈序列是否满足后进先出(LIFO)原则：

- 模拟入栈/出栈过程
- 当栈顶元素等于出栈序列当前元素时出栈
- 最终栈为空则合法

黄金法则

栈顶元素 == 出栈序列当前元素 时必须出栈

高频公式

```
    // 判断 popV 是否是 pushV 的合法出栈序列
    bool isValidStackSeq(vector<int> pushV, vector<int> popV) {
        stack<int> st;
        int idx = 0;
```

```
        for (int num : pushV) {
            st.push(num);
            // 关键：连续出栈检查
            while (!st.empty() && st.top() == popV[idx]) {
                st.pop();
                idx++;
            }
        }
        return st.empty();
    }
```

易错点

- 忽略连续出栈可能性（需用 while 而非 if）
- 未处理空栈时的.top()调用（导致崩溃）
- 输入序列含重复元素时的处理错误

3. 二叉树遍历重建

核心知识点

利用两种遍历序列唯一确定二叉树：

- 先序+中序：先序首元素为根，中序分割左右子树
- 后序+中序：后序末元素为根，中序分割左右子树

黄金法则

先序/后序定位根节点，中序定位左右子树边界

高频公式

```
// 先序+中序重建二叉树
TreeNode* buildTree(vector<int>& pre, int preStart, int preEnd,
                    vector<int>& in, int inStart, int inEnd) {
```

```
        if (preStart > preEnd) return nullptr;


        int rootVal = pre[preStart];
        int rootIdx = find(in.begin() + inStart, in.begin() + inEnd + 1,
rootVal) - in.begin();
        int leftSize = rootIdx - inStart;


        TreeNode* root = new TreeNode(rootVal);
        root->left = buildTree(pre, preStart+1, preStart+leftSize,
                               in, inStart, rootIdx-1);
        root->right = buildTree(pre, preStart+leftSize+1, preEnd,
                                in, rootIdx+1, inEnd);
        return root;
    }
```

易错点

- 递归边界条件错误（start > end 时应返回空）
- 未计算左子树大小导致索引越界
- 中序查找根节点未处理不存在情况

4. 表达式转换（中缀→后缀）

核心知识点

转换规则：

- 操作数直接输出
- 运算符入栈（栈顶优先级≥当前符时先出栈）
- 左括号入栈，右括号弹出至左括号

黄金法则

栈内优先级：左括号 < 加减 < 乘除 < 右括号（触发弹出）

高频公式

```
string infixToPostfix(string s) {
    stack<char> st;
    string res;
    unordered_map<char, int> pri = {{'+',1}, {'-',1}, {'*',2}, {'/',2}};

    for (char c : s) {
        if (isdigit(c)) res += c;
        else if (c == '(') st.push(c);
        else if (c == ')') {
            while (st.top() != '(') {
                res += st.top(); st.pop();
            }
            st.pop(); // 弹出'('
        } else { // 运算符
            while (!st.empty() && pri[st.top()] >= pri[c]) {
                res += st.top(); st.pop();
            }
            st.push(c);
        }
    }
    // 弹出剩余运算符
    while (!st.empty()) {
        res += st.top(); st.pop();
    }
```

```
        return res;
    }
```

易错点

- 忽略运算符优先级比较
- 右括号处理时未检查空栈
- 最后未弹出栈中剩余运算符

5. 二叉查找树验证

核心知识点

BST 定义：

- 左子树所有节点 < 根节点
- 右子树所有节点 > 根节点
- 左右子树也必须是 BST

黄金法则

递归传递值域边界（避免仅比较父子节点）

高频公式

```
    bool isValidBST(TreeNode* root, long min_val = LONG_MIN, long max_val = LONG_MAX) {
        if (!root) return true;
        if (root->val <= min_val || root->val >= max_val)
            return false;
        return isValidBST(root->left, min_val, root->val) &&
               isValidBST(root->right, root->val, max_val);
    }
```

易错点

- 仅比较父子节点（错误：需整棵子树在值域内）

- 边界值处理不当（如 INT_MIN/INT_MAX）
- 未考虑相等值是否允许（通常 BST 不允许重复值）

6. 循环队列操作

核心知识点

循环队列核心操作：

- 队头指针 front，队尾指针 rear
- 队列空：front == rear
- 队列满：(rear+1)%size == front

黄金法则

指针移动必取模：**ptr = (ptr+1) % capacity**

高频公式

```
class CircularQueue {
    vector<int> data;
    int front, rear, size;
public:
    CircularQueue(int k) : data(k), front(0), rear(0), size(0) {}

    bool enQueue(int val) {
        if (isFull()) return false;
        data[rear] = val;
        rear = (rear + 1) % data.size(); // 关键取模
        size++;
        return true;
    }

    bool deQueue() {
```

```
            if (isEmpty()) return false;
            front = (front + 1) % data.size(); // 关键取模
            size--;
            return true;
        }

        bool isFull() {
            return size == data.size();
        }
    };
```

易错点

- 队满条件误用 rear+1 == front（未考虑循环）
- 指针移动未取模导致越界
- 未维护 size 导致判空/满逻辑复杂

7. 哈夫曼树与贪心算法

核心知识点

哈夫曼编码特性：

- 频率越高的字符编码越短
- 任意字符编码不是其他编码的前缀
- 贪心策略：每次合并频率最小的两棵树

黄金法则

最小堆实现：priority_queue<频率, vector<频率>, greater<>>

高频公式

```
int huffmanCode(vector<int>& freq) {
    priority_queue<int, vector<int>, greater<int>> pq(freq.begin(),
```

```
freq.end());
        int cost = 0;
        while (pq.size() > 1) {
            int a = pq.top(); pq.pop();
            int b = pq.top(); pq.pop();
            cost += a + b;          // 合并代价
            pq.push(a + b);         // 新节点入堆
        }
        return cost;
    }
```

易错点

- 未使用最小堆（默认最大堆）
- 仅计算根节点值（实际需累加合并代价）
- 忽略单节点特殊情况（size=1 时无合并）

8. 二叉树深度计算

核心知识点

关键公式：

- 最小深度：完全二叉树时深度最小
- 最大深度：退化为链表时深度最大
- 深度为 h 的满二叉树节点数：2^h-1

黄金法则

最小深度：满足 2^{h-1} < n ≤ 2^h-1

高频公式

```
// 最小深度计算（完全二叉树）
int minDepth(int n) {
```

```
        int h = 0, cnt = 0;
        while (cnt < n) {
            h++;
            cnt = (1 << h) - 1; // 2^h - 1
        }
        return h;
    }
    // 最大叶子数（完全二叉树）
    int maxLeaves(int n) {
        return (n + 1) / 2;  // 最后一层叶子数
    }
```

易错点

- 混淆深度定义（根深度=0 或 1）
- 最小深度未考虑非完全二叉树情况
- 叶子数计算未用整数除法（(n+1)/2 非 n/2）

9. 链式栈操作

核心知识点

链式栈特点：

- 栈顶指针指向最新节点
- 入栈：新节点→原栈顶，栈顶→新节点
- 出栈：栈顶→下一节点，删除原栈顶

黄金法则

入栈：先连后断（新节点 next=栈顶，再更新栈顶）

高频公式

```
    void push(Node* &top, int val) {
```

```
        Node* newNode = new Node(val);
        newNode->next = top;   // 新节点指向原栈顶
        top = newNode;         // 栈顶更新为新节点
    }
    int pop(Node* &top) {
        if (!top) throw "Empty";
        Node* temp = top;
        int val = top->data;
        top = top->next;       // 栈顶下移
        delete temp;
        return val;
    }
```

易错点

- 未更新栈顶指针（直接操作节点）
- 出栈未检查空栈
- 内存泄漏（未 delete 出栈节点）

10. 递归边界处理

核心知识点

递归三要素：

- 递归终止条件
- 递归参数更新
- 结果合并方式

黄金法则

递归基必须能终止（如节点为空/索引越界）

高频公式

```
// 二叉树遍历递归模板
void traverse(TreeNode* root) {
    if (!root) return;      // 关键终止条件

    // 前序位置
    traverse(root->left);
    // 中序位置
    traverse(root->right);
    // 后序位置
}
```

易错点

- 缺少递归基导致无限递归
- 未正确传递参数（如未+1/-1）
- 递归层数过深导致栈溢出

（三）真题强化

1. 2010 年第 16 题（双向链表）

双向链表中有两个指针域 llink 和 rlink，分别指向该结点的前驱及后继。设 p 指向链表中的一个结点，它的左右结点均非空。现要求删除结点 p，则下面语句序列中错误的是（ ）。

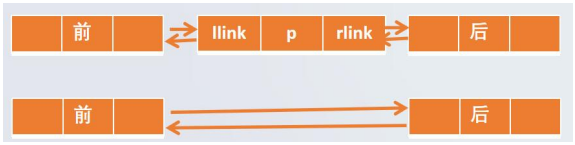
A. p->rlink->llink = p->rlink; p->llink->rlink = p->llink; delete p;

B. p->llink->rlink = p->rlink; p->rlink->llink = p->llink; delete p;

C. p->rlink->llink = p->llink; p->rlink->llink->rlink = p->rlink; delete p;

D. p->llink->rlink = p->rlink; p->llink->rlink->llink = p->llink; delete p;

答案： A



解析：

正确删除双向链表结点 p 的操作：

将 p 的前驱结点的 rlink 指向 p 的后继：p->llink->rlink = p->rlink

将 p 的后继结点的 llink 指向 p 的前驱：p->rlink->llink = p->llink

释放 p：delete p

选项 A 错误：

p->rlink->llink = p->rlink：使后继结点的 llink 指向自身（逻辑错误）

p->llink->rlink = p->llink：使前驱结点的 rlink 指向自身（逻辑错误）

其他选项正确实现删除操作。

2. 2017 年第 13 题（链式栈）

向一个栈顶指针为 hs 的链式栈中插入一个指针 s 指向的结点时，应执行（ ）。

A. hs->next = s;

B. s->next = hs; hs = s;

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

D. s->next = hs; hs = hs->next;

答案： B

解析：

链式栈的栈顶指针 hs 指向栈顶结点。

插入新结点 s 的步骤：

将 s 的 next 指向当前栈顶：s->next = hs

更新栈顶指针为 s：hs = s

选项 B 正确实现插入操作，其他选项会导致栈断裂或指针错误。

3. 2018 年第 23 题（双向链表应用）

完善程序

```
#include <iostream>
using namespace std;
const int N = 100010;
int n;
int L[N], R[N], a[N];
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        int x;
        cin >> x;
        【①】; //
    }

    for (int i = 1; i <= n; ++i) {
        R[i] = 【②】; //
        L[i] = i - 1;
    }

    for (int i = 1; i <= n; ++i) {
        L[【③】] = L[a[i]]; //
```

```
            R[L[a[i]]] = R[【④】]; //
        }

        for (int i = 1; i <= n; ++i) {
            cout << 【⑤】 << " "; //
        }
        cout << endl;
        return 0;
    }
```

答案：

① a[x] = i

② i + 1

③ R[a[i]]

④ a[i]

⑤ R[i]

解析：

功能：求解排列 P 中每个元素右侧第一个比它大的元素位置。

①：存储元素 x 的位置 a[x] = i

②：初始化双向链表（左指针 i-1，右指针 i+1）

③-④：从最小值开始删除结点，更新链表：

L[R[a[i]]] = L[a[i]]：更新后继的左指针

R[L[a[i]]] = R[a[i]]：更新前驱的右指针

⑤：删除后 R[i] 存储右侧第一个更大元素的位置

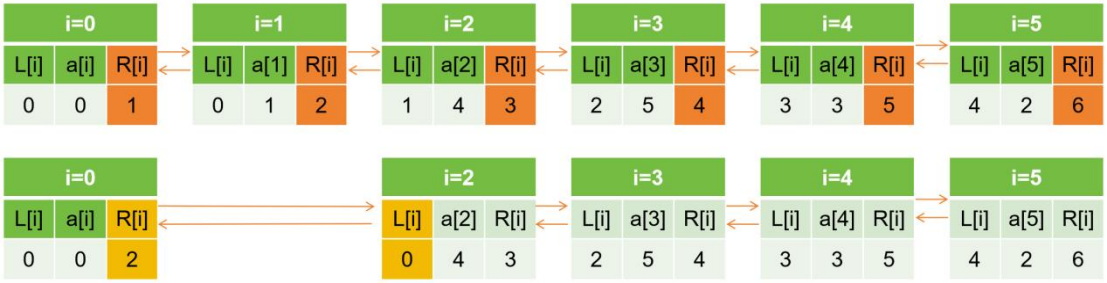
给定一个包含 n 个整数的序列（序列中整数为 1 到 n 的排列），按值**从小到大的顺序（值 1 到 n）依次删除每个值对应的位置节点**。对于每个被删除的位置 i，记录删除时其后继节点的位置（若没有后继，则为 n+1），最终输出每个位置被删除时

的后继位置。

核心算法原理

- 位置映射：将值映射到其位置（a[x] = i 表示值为 **x 的元素在序列中的位置是 i**）。
- 双向链表：用数组模拟双向链表：
  - R[i] 表示位置 i 的**后继位置**
  - L[i] 表示位置 i 的**前驱位置**
- 按值删除：从值 1 到 n 依次处理：
  - 通过映射 a[i] 找到要删除的位置 p
  - 删除节点 p：
    - 将 **p 的前驱节点的后继**指向 **p 的后继**：R[L[p]] = R[p]
    - 将 **p 的后继节点的前驱**指向 **p 的前驱**：L[R[p]] = L[p]
- 记录后继：删除节点时，R[p] 的值即为该位置被删除时的后继位置（最终输出）。

算法本质：使用双向链表求解每个元素右侧首个更大值的位置



执行流程（输入 1 5 4 2 3）：

- 映射：a[1]=1, a[5]=2, a[4]=3, a[2]=4, a[3]=5
- 链表初始化：R=[2,3,4,5,6], L=[0,1,2,3,4]
- 按值升序处理：

  值 1：删除位置 1 → 修改 R[0]=2, L[2]=0

  值 2：删除位置 4 → 修改 R[3]=5, L[5]=3

  值 3：删除位置 5 → 修改 R[4]=6, L[6]=4

  值 4：删除位置 3 → 修改 R[2]=5, L[5]=2

  值 5：删除位置 2 → 修改 R[1]=3, L[3]=1

- 输出 R：[2,6,5,6,6]（对应答案 q=[2,6,6,5,6]）

算法原理：

- 按值从小到大删除节点
- 删除节点时，其**右指针即为右侧首个更大值**的位置
- 链表维护剩余节点（均为更大值）的位置关系

完整版程序

```
#include <iostream>
using namespace std;
const int N = 100010;
int n;
int L[N], R[N], a[N];
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        int x;
        cin >> x;
        a[x] = i; // ①
    }

    for (int i = 1; i <= n; ++i) {
        R[i] = i + 1; // ②
        L[i] = i - 1;
    }

    for (int i = 1; i <= n; ++i) {
        L[R[a[i]]] = L[a[i]]; // ③
```

```
            R[L[a[i]]] = R[a[i]]; // ④
        }

        for (int i = 1; i <= n; ++i) {
            cout << R[i] << " "; // ⑤
        }
        cout << endl;
        return 0;
    }
```

4. 2009 年第 12 题（栈的合法性）

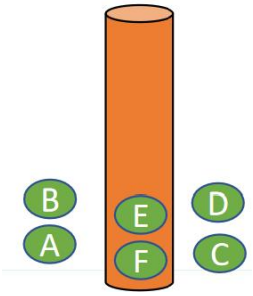
有六个元素 FEDCBA 从左至右依次顺序进栈，在进栈过程中会有元素被弹出栈。问下列哪一个不可能是合法的出栈序列?

A. EDCFAB

B. DECABF

C. CDFEBA

D. BCDAEF



答案： C

解析:

栈规则：后进先出（LIFO），出栈序列需满足任意时刻后出栈元素不能在先出栈元素前压栈。

选项 C 分析（CDFEBA）：

F 进栈后，若 C 出栈（需先弹出 D、E），但序列中 D 在 C 后出栈（矛盾）

其他选项合法：

A:F 进 → E 出 → D 进 → D 出 → C 进 → C 出 → F 出 → A 进 → B 进 → A 出 → B 出

B:F 进 → E 进 → D 进 → D 出 → E 出 → C 进 → C 出 → B 进 → A 进 →

A 出 → B 出 → F 出

D:F 进 → E 进 → D 进 → C 进 → B 进 → B 出 → C 出 → D 出 → A 进 → A 出 → E 出 → F 出

5. 2009 年第 13 题（后缀表达式）

表达式 a*(b+c)-d 的后缀表达式是：

A. abcd*+-

B. abc+*d-

C. abc*+d-

D. -+*abcd

答案： B

解析：

中缀转后缀规则：

操作数顺序不变

运算符按运算顺序后置

转换步骤：

a*(b+c)-d → 括号内优先：(b+c) → bc+ → 乘法：a(bc+)* → abc+* → 减法：abc+*d-

6. 2022 年第 6 题（前缀表达式）

对表达式 a+(b-c)*d 的前缀表达式为（ ）。

A. *+a-bcd

B. +a*-bcd

C. abc-d*+

D. abc-+d

答案： B

解析：

中缀转前缀规则：从右向左扫描，运算符前置。

转换步骤：

a+(b-c)*d → 括号内：(b-c) → -bc → 乘法：*-bcd → 加法：+a*-bcd

7. 2023 年第 8 题（后缀转中缀）

后缀表达式 6 2 3 + - 3 8 2 / + * 2 ^ 3 + 对应的中缀表达式是（ ）。

A. ((6-(2+3))*(3+8/2))^2+3

B. 6-2+3*3+8/2^2+3

C. (6-(2+3))*((3+8/2)^2)+3

D. 6-((2+3)*(3+8/2))^2+3

答案： A

解析：

后缀求值步骤：

2 3 + → 5

6 5 - → 1

8 2 / → 4

3 4 + → 7

1 7 * → 7

7 2 ^ → 49

49 3 + → 52

中缀结构：

((6-(2+3)) × (3+8÷2))² + 3

三、二叉树

8. 2007 年第 20 题（二叉树遍历）

已知 7 个节点的二叉树的先根遍历是 1 2 4 5 6 3 7，中根遍历是 4 2 6 5 1 7 3，则该二叉树的后根遍历是（ ）。

A. 4 6 5 2 7 3 1

B. 4 6 5 2 1 3 7

C. 4 2 3 1 5 4 7

D. 4 6 5 3 1 7 2

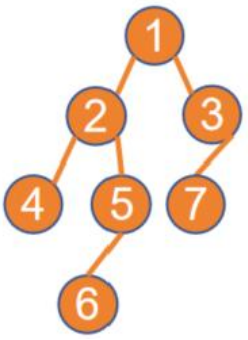
答案： A

解析：

建树步骤：

先序首元素 1 为根。

中序分割：左子树 4 2 6 5，右子树 7 3。



递归构建：

左子树先序 2 4 5 6 → 根 2，中序 4（左）、6 5（右）

右子树先序 3 7 → 根 3，中序 7（左）

二叉树结构：

后序遍历：左子树 → 右子树 → 根 → 4 6 5 2 7 3 1

9. 2011 年第 7 题（二叉树性质）

如果根结点的深度记为 1，则一棵恰有 2011 个叶结点的二叉树的深度最少是（ ）。

A. 10

B. 11

C. 12

D. 13

答案： C

解析：

最小深度条件：完全二叉树时深度最小。

公式：**叶子结点数 L 满足 L< 2^(h-1)，其中 h 为深度。**

在二叉树的第 n 层上，最多有 2^n−1（n≥1）个节点

计算：2^11 = 2048 > 2011，故最小深度 h = 12（2^11 对应深度 12）。

10. 2012 年第 6 题（二叉树遍历）

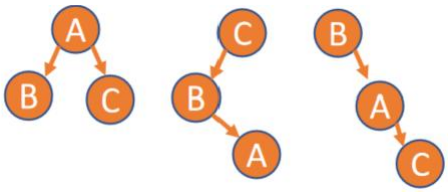
如果一棵二叉树的中序遍历是 BAC，那么它的先序遍历不可能是（ ）。

A. ABC

B. CBA

C. ACB

D. BAC



答案： C

解析：

中序 BAC：B 为左子树，A 为根，C 为右子树。

先序可能性：

A：A（根）→ B（左）→ C（右） → 中序 BAC（合法）

C：A（根）→ C（左）→ B（右） → 中序 CAB（与 BAC 矛盾）

11. 2015 年第 22 题（二叉树性质）

一棵结点数为 2015 的二叉树最多有___个叶子结点。

答案： 1008

解析：

最大叶子数条件：完全二叉树时叶子最多。

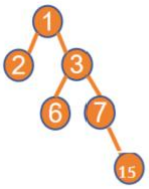
设度为 0 的节点有 n 个，那么度为 2 节点有 n-1，设度为 1 的节点有 a 个，总节点：n-1+a+n=2015，即 2n+a=2016

当 a=0 时，n=1008

对任何二叉树，度为 0 的节点（叶子节点）总是比度为 2 的节点多一个

12. 2016 年第 11 题（二叉树存储）

一棵二叉树采用顺序存储结构（根下标为 1，结点 i 的左孩为 2i，右孩为 2i+1），则图中所有结点的最大下标为（ ）。



A. 6

B. 10

C. 12

D. 15

答案： D

解析：

完全二叉树性质：深度为 h 的树最大下标为 2^h - 1。

示例：深度 4 的满二叉树最大下标 2^4 - 1 = 15（如结点按层编号为 1 至 15）。

13. 2018 年第 7 题（满 k 叉树）

根节点深度为 0，一棵深度为 h 的满 k (k>1) 叉树共有（ ）个结点。

A. (k^(h+1) - 1)/(k - 1)

B. k^h - 1

C. k^h

D. (k^h - 1)/(k - 1)

答案： A

解析：

示例：k=2,h=2 时 S = (2^3-1)/(2-1) = 7（符合满二叉树）。

14. 2021 年第 11 题（哈夫曼树）

在数据压缩编码中的哈夫曼编码方法，在本质上是一种（ ）的策略。

A. 枚举

B. 贪心

C. 递归

D. 动态规划

答案： B

解析：

哈夫曼编码是一种经典的数据压缩算法，其核心在于构建最优前缀码。算法的执行过程如下：

每次从频率优先队列中**选取两个频率最小的**节点进行合并，形成一个新的父节点，其频率为子节点频率之和。

重复此过程，直到所有节点合并为一棵哈夫曼树。

此方法体现了贪心算法的核心思想：每一步都采取局部最优选择（合并当前最小频率节点），通过局部最优累积达到全局最优（整体编码长度最小）。

为什么不是其他选项?

A. 枚举：涉及穷举所有可能解，哈夫曼编码不遍历所有组合。

C. 递归：哈夫曼编码可用递归实现，但本质策略是贪心，递归只是实现方式。

D. 动态规划：需解决重叠子问题并保存中间状态，哈夫曼编码无此特征。

因此，正确答案为 B（贪心）。

15. 2023 年第 5 题（三叉树）

根节点的高度为 1，一棵拥有 2023 个节点的三叉树高度至少为（ ）。

A. 6

B. 7

C. 8

D. 9

答案： C

解析：

三叉树的最小高度发生在树为完全三叉树时（即每层尽量填满节点）。

高度至少为 8，因为 h=7 时最大节点数 1093 小于 2023，而 h=8 时最大节点数 3280 大于等于 2023。

因此，正确答案为 C（8）。

16. 2019 年第 14 题（二叉树遍历）

假设一棵二叉树的后序遍历序列为 DGJHEBIFCA，中序遍历序列为 DBGEHJACIF，则其前序遍历序列为（ ）。

A. ABCDEFGHIJ

B. ABDEGHJCFI

C. ABDEGJHCFI

D. ABDEGHJFIC

答案： B

解析：

通过后序和中序遍历序列重建二叉树，再输出前序遍历序列。重建步骤如下：

步骤 1：确定根节点

后序序列末元素为根（A）。中序序列中 A 的位置分割左/右子树：

中序左子树：A 左侧为 DBGEHJ（节点序列）。

中序右子树：A 右侧为 CIF（节点序列）。

后序序列中，左子树部分对应 A 前的子序列（DGJHEB），右子树部分对应 B 到 A 前的子序列（IFC，但需注意后序为左右根顺序）。

步骤 2：递归构建左子树（后序：DGJHEB，中序：DBGEHJ）

后序末元素 B 为左子树根。中序中 B 的位置：左为 D，右为 GEHJ。

B 的左子树：后序 D（B 前的单个元素），中序 D（B 左侧），故 D 为 B 的左子节点。

B 的右子树：后序 GJHE，中序 GEHJ。后序末 E 为根。中序中 E 的位置：左为 G，右为 HJ。

E 的左子树：后序 G（E 前的元素），中序 G（E 左侧），故 G 为 E 的左子节点。

E 的右子树：后序 JH，中序 HJ。后序末 H 为根。中序中 H 的位置：左空，右为 J。

H 的右子树：后序 J（H 前的元素），中序 J（H 右侧），故 J 为 H 的右子节点。

步骤 3：递归构建右子树（后序：IFCA，但 A 为根已用，实际右子树后序：IFC，中序：CIF）

后序末 C 为右子树根。中序中 C 的位置：左空，右为 IF。

C 的右子树：后序 IF，中序 IF。后序末 F 为根。中序中 F 的位置：左为 I，右空。

F 的左子树：后序 I（F 前的元素），中序 I（F 左侧），故 I 为 F 的左子节点。

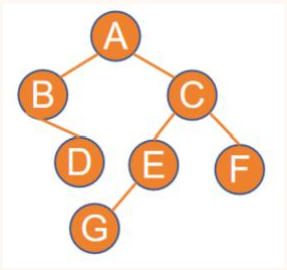
前序遍历（根→左→右）： A → B → D → E → G → H → J → C → F → I，即序列 ABDEGHJCFI。

比较选项：B（ABDEGHJCFI）匹配。

因此，正确答案为 B。

17. 2008 年第 26 题（二叉树遍历）

阅读程序写结果




```
#include<iostream>
#include<cstring>
using namespace std;
#define MAX 100
void solve(char first[], int spos_f, int epos_f, char mid[], int spos_m, int epos_m) {
    int i, root_m;
    if (spos_f > epos_f) return;
    for (i = spos_m; i <= epos_m; i++)
        if (first[spos_f] == mid[i]) {
            root_m = i;
            break;
        }
    solve(first, spos_f + 1, spos_f + (root_m - spos_m), mid, spos_m, root_m - 1);
    solve(first, spos_f + (root_m - spos_m) + 1, epos_f, mid, root_m + 1, epos_m);
    cout << first[spos_f];
}
int main() {
    char first[MAX], mid[MAX];
    int len;
    cin >> len;
```

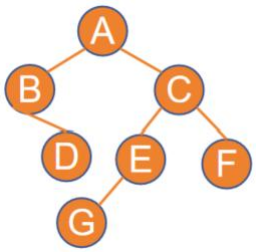
```
        cin >> first >> mid;
        solve(first, 0, len - 1, mid, 0, len - 1);
        cout << endl;
        return 0;
    }
```

输入：

7

ABDCEGF

BDAGECF



输出：

答案： **DBGEFCA**

解析：

程序功能：根据前序序列（first）和中序序列（mid）输出后序遍历序列（通过递归实现后序遍历输出）。

- 输入说明：
  - len = 7：节点数。
  - 前序序列：ABDCEGF → 顺序 A、B、D、C、E、G、F。
  - 中序序列：BDAGECF → 顺序 B、D、A、G、E、C、F。
- 建树过程：
  - 根节点：前序首元素 A。中序中 A 的位置（索引 2）分割左/右子树：
    - 中序左子树：A 左侧 BD（索引 0-1）。
    - 中序右子树：A 右侧 GECF（索引 3-6）。
  - 递归左子树：前序子序列 BD（起始索引 spos_f+1 到 spos_f+(root_m-spos_m) = 1 到 2），中序子序列 BD（索引 0-1）。
    - 根 B（前序首），中序中 B 左侧空，右侧 D → B 右子为 D。
  - 递归右子树：前序子序列 CEGF (起始索引 spos_f+(root_m-spos_m)+1 = 3

到 6），中序子序列 GECF（索引 3-6）。

- 根 C（前序首），中序中 C 左侧 GE，右侧 F → C 左子树 GE，右子树 F。
  - 递归 C 左子树：前序 EG，中序 GE → 根 E，中序中 E 左侧 G，右侧空 → E 左子为 G。
  - 递归 C 右子树：前序 F，中序 F → 根 F。

- 后序遍历输出顺序（左右根）：
  - 左子树（B 的子树）：B 的左空，B 的右 D → 后序 D → B。
  - 右子树（C 的子树）：C 的左子树（E 和 G）→ G → E；C 的右子树 F → F；最后 C。
  - 根 A。
    - 完整序列：D → B → G → E → F → C → A → DBGEFCA（输出无空格）。
    - 因此，输出结果为 DBGEFCA。

18. 2013 年第 28 题（二叉查找树）

完善程序（二叉查找树）

二叉查找树具有如下性质：每个节点的值都大于其左子树上所有节点的值，小于其右子树上所有节点的值。试判断一棵树是否为二叉查找树。

输入说明：

第一行包含一个整数 n，表示这棵树有 n 个顶点，编号分别为 1, 2, ..., n，其中编号为 1 的为根结点。

之后的第 i 行有三个整数 value, left_child, right_child，分别表示该节点关键字的值、左子节点的编号、右子节点的编号；

如果不存在左子节点或右子节点，则对应位置为 0。

输出说明：

输出 1 表示这棵树是二叉查找树，输出 0 表示不是。

```
#include <iostream>
using namespace std;
```

```
const int SIZE = 100;
const int INFINITE = 1000000;
struct node {
    int left_child, right_child, value;
};
node a[SIZE];
int is_bst(int root, int lower_bound, int upper_bound) {
    int cur;
    if (root == 0) return 1;
    cur = a[root].value;
    if ((cur > lower_bound) && (【①】) &&
        (is_bst(a[root].left_child, lower_bound, cur) == 1) &&
        (is_bst(【②】, 【③】, 【④】) == 1))
        return 1;
    return 0;
}
int main() {
    int i, n; cin >> n;
    for (i = 1; i <= n; i++)
        cin >> a[i].value >> a[i].left_child >> a[i].right_child;
    cout << is_bst(【⑤】, -INFINITE, INFINITE) << endl;
    return 0;
}
```

答案:

① cur < upper_bound

② **a[root].right_child**

③ **cur**

④ **upper_bound**

⑤ **1**

解析：

程序功能：递归验证二叉查找树（BST），利用上下界约束节点值范围。

- BST 验证原理：
  - 当前节点值必须在开区间（lower_bound, upper_bound）内。
  - 左子树所有节点值小于当前值 → 递归左子树时，上界更新为当前值。
  - 右子树所有节点值大于当前值 → 递归右子树时，下界更新为当前值。
- 填空详解：
  - ① cur < upper_bound：当前节点值需小于上界（与 cur > lower_bound 结合确保值在区间内）。
  - ② a[root].right_child：递归检查右子树，传入右子节点编号。
  - ③ cur：右子树的下界更新为当前节点值（右子树值需大于当前值）。
  - ④ upper_bound：右子树的上界继承父调用（保持不变）。
  - ⑤ 1：初始调用根节点编号为 1（题目指定根节点编号为 1）。
- 递归逻辑：
  - 终止条件：root == 0（空树是 BST）。
  - 当前节点检查：值在（lower_bound, upper_bound）内。
  - 左子树递归：下界不变，上界设为当前值。
  - 右子树递归：下界设为当前值，上界不变。
- 示例分析：

  假设输入：

  3

  2 2 3

1 0 0

3 0 0

- 节点 1（根）：value=2，left_child=2，right_child=3。
- 节点 2：value=1，left_child=0，right_child=0。
- 节点 3：value=3，left_child=0，right_child=0。
- 递归调用 is_bst(1, -∞, ∞)：
- 节点 1：值 2 ∈ (-∞, ∞)，左子树递归 is_bst(2, -∞, 2)，右子树递归 is_bst(3, 2, ∞)。
- 节点 2：值 1 ∈ (-∞, 2) → 有效；左右子树空，返回 1。
- 节点 3：值 3 ∈ (2, ∞) → 有效；左右子树空，返回 1。
- 最终返回 1（是 BST）。

完整版程序

```
#include <iostream>
using namespace std;
const int SIZE = 100;
const int INFINITE = 1000000;
struct node {
    int left_child, right_child, value;
};
node a[SIZE];
int is_bst(int root, int lower_bound, int upper_bound) {
    int cur;
    if (root == 0) return 1;
    cur = a[root].value;
    if ((cur > lower_bound) && (cur < upper_bound) &&
        (is_bst(a[root].left_child, lower_bound, cur) == 1) &&
```

```
            (is_bst(a[root].right_child, cur, upper_bound) == 1))
                return 1;
        return 0;
}
int main() {
        int i, n; cin >> n;
        for (i = 1; i <= n; i++)
                cin >> a[i].value >> a[i].left_child >> a[i].right_child;
        cout << is_bst(1, -INFINITE, INFINITE) << endl;
        return 0;
}
```