数据结构课程习题总览

——24 级信管 郭鹏飞

前言

本文档加注【习题】的题目，为订正过确认无误的；未标注的（5.3、5.6、5.11、5.13、7.3）为 gpf 同学没学过的；（3.4、6.21）缺失题干；（6.23）个人感觉仅凭题目中的先序后序无法确定唯一的树。

本文档相比原来的“习题答案”，进行了一遍人工**勘误**，将所有内容进行了订正（不保证完全正确，至少目前 gpf 同学看不出来错误了），通过 DeepSeek 补充了部分习题的**讲解。**同时对**排版**进行了些许改变以增加易读性。

参考文档：

- 习题答案-第1至3章.docx
- 习题答案-第4至6章-更正-1228-1.docx
- 习题答案-第7章.docx
- 习题答案-第9章-更正-1224-1.docx

第 1~3 章

【习题】1.1 基本概念

1. **数据** 是对客观事物的符号表示，是能输入到计算机中并被计算机程序处理的符号总称。
2. **数据元素** 是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。一个数据元素可由若干**数据项**组成。
3. **数据对象** 是性质相同的数据元素的集合，是数据的一个子集。
4. **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。通常包括以下三方面：
    - **逻辑结构**：数据元素之间的逻辑关系。
    - **存储结构（物理结构）**：数据结构在计算机中的表示（映像）。
    - **运算**：施加在该数据上的操作。
5. **存储结构** 是数据结构在计算机中的存储方式，包括数据元素的表示和关系的表示。常见的有顺序存储、链式存储、索引存储和散列存储等。
6. **数据类型** 是一个值的集合和定义在这个值集上的一组操作的总称。例如，C 语言中的 int 类型包括整数集合和加、减、乘、除等运算。
7. **抽象数据类型（ADT）** 是指一个数学模型以及定义在该模型上的一组操作。ADT 只定义逻辑特性和操作，不涉及具体实现。例如，“栈”是一个抽象数据类型，可定义 InitStack、Push、Pop 等操作。

【习题】1.2 数据结构和抽象数据类型的概念

- **数据结构** 强调的是数据元素之间的**逻辑关系、存储结构及相关操作**，是计算机存储、组织数据的方式，既包括逻辑层面，也包括物理实现层面。
- **抽象数据类型（ADT）** 是数据结构的**抽象描述**，仅关注“**做什么**”，而不关心“**如何做**”。它通过数据抽象和数据封装，定义了一组操作接口，隐藏了内部实现细节。
- **程序设计语言中的数据类型** 是语言系统内置或用户定义的**具体类型**，规定了变量的**取值范围、存储形式以及允许的操作**，如 int、float、struct 等。它是 ADT 在某种语言中的**具体实现**。

程序设计语言中**数据类型概念**的**区别**

- **数据类型**（如 C 语言的 int）是**语言层面的实现**，包括存储方式和操作；
- **数据结构**关注数据的**逻辑与物理组织**；
- ADT 是更高层次的**抽象模型**，强调操作的规范与封装，独立于具体语言。

根据《数据结构题集（C 语言版）》（严蔚敏、吴伟民、米宁 编著）第二章的内容，以下是 2.1、2.2、2.3 题的标准答案：

【习题】2.1 描述以下三个概念的区别：头指针，头结点，首元结点（第一个元素结点）

- **头指针** 指向链表中**第一个结点**的指针。若链表有头结点，则头指针指向头结点；否则指向首元结点。头指针是链表的必要标识，没有头指针则无法访问链表。
- **头结点** 是附加在链表**首元结点之前**的一个结点。其**数据域通常不存储实际信息**（或存储如链表长度等附加信息），指针域指向首元结点。引入头结点可以**统一空表和非空表的操作**，简化插入、删除等算法的实现。
- **首元结点** 是链表中存储**第一个实际数据元素**的结点。如果链表有头结点，则首元结点是头结点的后继结点；如果没有头结点，则首元结点就是链表的第一个结点。

简单总结： 头指针 → 指向链表的入口（可能是头结点或首元结点） 头结点 → 附加的结点，不存实际数据，便于操作统一 首元结点 → 第一个存实际数据的结点

【习题】2.2 填空题

(1) 在顺序表中插入或删除一个元素，需要平均移动 <u>一半（或 n/2）</u> 元素，具体移动的元素个数与 <u>插入或删除的位置</u> 有关。
(2) 顺序表中逻辑上相邻的元素的物理位置 <u>一定</u> 紧邻。单链表中逻辑上相邻的元素的物理位置 <u>不一定</u> 紧邻。
(3) 在单链表中，除了首元结点外，任一结点的存储位置由 **<u>其前驱结点的指针域（或 next 域）</u>** 指示。
(4) 在单链表中设置头结点的作用是 **<u>简化插入、删除操作，统一空表和非空表的处理</u>**（*或类似表述：使空链表和非空链表的操作一致；方便首元结点的插入和删除*）。

【习题】2.3 在什么情况下用顺序表比链表好？

顺序表的优点在于**随机访问效率高**（按索引直接访问），**存储密度高**（无需额外指针空间）。在以下情况下更适合使用顺序表：

1. 需要**频繁进行**按序号（下标）**访问元素**的操作。

2. 表中元素总量大致已知，**插入、删除操作较少**（或只在尾部进行）。
3. **对存储空间要求较高**，希望尽量减少存储开销（链表需要额外指针域）。

【习题】2.6 题答案

已知 L 是无表头结点的单链表，且 P 结点既不是首元结点，也不是尾元结点。试从下列提供的答案中选择合适的语句序列。

a) **在 P 结点后插入 S 结点的语句序列是** (4) S->next = P->next; (1) P->next = S;

说明：先让 S 指向 P 的后继（next），再让 P 指向 S。

b) **在 P 结点前插入 S 结点的语句序列是** (7) Q = P; (11) P = L; (8) while (P->next != Q) P = P->next;（假设 Q 记录原 P 结点位置） (4) S->next = Q;（或 S->next = P->next） (1) P->next = S;

说明：（需先找到 P 的前驱，但选项有限，常用 (11)从头遍历）首先用 Q 记录原 P 结点的位置，然后将 P 指针重置为头指针 L 开始遍历，循环**找到 P 的前驱结点**（即 P->next == Q 的结点），接着**让 S 指向原 P 结点(Q)**，最后**让前驱结点(P)指向 S**，完成在 P 前的插入操作。

c) **在表首插入 S 结点的语句序列是** (5) S->next = L; (12) L = S;

说明：先让 S 指向原首元结点，再让头指针 L 指向 S。

d) **在表尾插入 S 结点的语句序列是** (11) P = L; (9) while (P->next != NULL) P = P->next; (1) P->next = S; (6) S->next = NULL;
说明：P 遍历到尾结点，然后插入 S，最后让 S 的 next 为 NULL。

【习题】2.7 题答案

已知 L 是带表头结点的非空单链表，且 P 结点既不是首元结点，也不是尾元结点。 试从下列提供的答案中选择合适的语句序列。

a) **删除 P 的直接后继**： (11) Q = P->next;（或 P->next = P->next->next 需先保存） (3) P->next = P->next->next; (14) free(Q);

说明：先用 Q 保存 P 的后继结点，然后修改 P 的 next 指针跳过该结点，最后释放

Q。

b) **删除 P 的直接前驱**: (10) Q = P; (从头遍历找前驱的前驱) (12) P = L; (8) while (P->next->next != Q) P = P->next; (11) Q = P->next (Q 为要删除的结点) (3) P->next=P->next->next; (14) free(Q);

说明：用 Q 保存原 P 结点的位置，P 从头结点开始遍历，找到前驱的前驱结点，Q 重新赋值为要删除的结点（P 的直接前驱），前驱的前驱跳过要删除的结点，释放要删除的结点。

c) **删除 P 结点**: (10) Q = P; (12) P = L; (7) while (P->next != Q) P = P->next; (3) P->next=P->next->next; (14) free(Q);

说明：保存要删除的 P 结点，然后从头遍历找到 P 的前驱结点，修改前驱的 next 指针跳过 P，最后释放 P。

d) **删除首元结点**: (12) P = L; (11) Q=P->next; (3) P->next = P->next->next; (14) free(Q);

说明：头指针 L 指向头结点，保存首元结点，修改头结点的 next 指针跳过首元结点，释放首元结点。

e) **删除尾元结点**: (9) while (P->next->next != NULL) P = P->next; (11) Q=P->next; (3) P->next=P->next->next; (14) free(Q);

说明：首先遍历到倒数第二个结点（P->next->next 为 NULL），保存尾元结点，将倒数第二个结点的 next 置为 NULL，释放尾元结点。

【习题】2.8 题答案

已知 P 结点是某双向链表的中间结点。试从下列提供的答案中选择合适的语句序列。

a) **在 P 结点后插入 S 结点的语句序列是** (7) S->next = P->next; (6) S->priou = P; (3) P->next = S; (12) S->next->priou = S; (即 P->next->priou = S)

常见答案: (7), (6), (3), (12)或(12), (7), (3), (6)

说明：

先让 S 的 next 指向 P 的后继，S 的 priou 指向 P

然后让 P 的 next 指向 S

最后让原 P 的后继（现在是 S 的后继）的 priou 指向 S

顺序很重要：必须先处理 S 与原 P 后继的关系，再修改 P 的 next

b) **在 P 结点前插入 S 结点的语句序列是** (8) S->priou = P->priou; (5) S->next = P; (13) P->priou->next = S; (4) P->priou = S;

常见答案: (8), (5), (13), (4) 或 (8) (4) (5) (13)

说明：
先让 S 的 priou 指向 P 的前驱，S 的 next 指向 P
然后让 P 的前驱的 next 指向 S
最后让 P 的 priou 指向 S
顺序很重要：必须先让 S 与原 P 前驱建立关系，再修改 P 的 priou

c) **删除 P 结点的直接后继结点的语句序列是** (15) Q = P->next; (1) P->next = Q->next;或 P->next = P->next->next; (11) P->next->priou = P; (18) free(Q);

常见答案: (15), (1), (11), (18)

说明：
先用 Q 保存要删除的后继结点
让 P 的 next 指向后继的后继（跳过要删除的结点）
让新后继的 priou 指向 P
释放要删除的结点
注意：步骤(11)中的 P->next 已经是新的后继了

d) **删除 P 结点的直接前驱结点的语句序列是** (16) Q = P->priou; (2) P->priou = Q->priou; 或 P->priou = P->priou->priou; (10) P->priou->next = P; (即 Q->priou->next = P) (18) free(Q);

常见答案: (16), (2), (10), (18)

说明：
先用 Q 保存要删除的前驱结点
让 P 的 priou 指向前驱的前驱（跳过要删除的结点）
让新前驱的 next 指向 P
释放要删除的结点
注意：步骤(10)中的 P->priou 已经是新的前驱了

e) **删除 P 结点的语句序列是** (9) P->priou->next = P->next; (14) P->next->priou = P->priou; (17) free(P);

常见答案: (9), (14), (17)

说明：

让 P 的前驱的 next 指向 P 的后继（跳过 P）
让 P 的后继的 priou 指向 P 的前驱（跳过 P）
释放 P 结点
两步指针修改可以交换顺序，但必须在 free 之前完成

【习题】3.2 简述栈和线性表的差别

栈是**操作受限的线性表**，是线性表的一个子集。它们的差别主要体现在逻辑特性和操作上：

1. **逻辑结构**：
    - 线性表的元素之间是“一对一”的线性关系，每个元素有且仅有一个**前驱**和一个**后继**（首尾元素除外）。
    - 栈同样具有这种线性关系，但其特殊性在于，元素的插入和删除只能在表的**同一端**（栈顶）进行。
2. **数据操作**：
    - 线性表允许在任意位置进行**插入**、**删除**和**存取**操作，操作更为灵活。
    - 栈只允许在**栈顶**进行插入（Push，入栈）和删除（Pop，出栈）操作，遵循**后进先出**的原则。存取（GetTop）操作也只能作用于栈顶元素。
3. **存储结构**：
    - 两者都可以采用**顺序存储结构**（如顺序栈、顺序表）和**链式存储结构**（如链栈、链表）实现。
4. **应用场景**：
    - 线性表多用于表示一般性的线性数据集合。
    - 栈多用于需要“后进先出”逻辑的场景，如函数调用、表达式求值、括号匹配、递归调用等。

核心区别：栈是逻辑上的一种特殊线性表，其操作是线性表操作的一个**子集**，但增加了一个“**仅在栈顶操作**”的限制规则。

【习题】3.3 写出下列栈程序段的输出结果

程序功能：对字符栈执行一系列入栈、出栈操作，最后输出。 **逐步执行与分析**：

1. InitStack(S);// 初始化栈 S 为空栈
2. x='c'; y='k'; //将 x、y 赋值为 c、k
3. Push(S, x);// 入栈 c。*栈内(自底向顶)：c*
4. Push(S, 'a');// 入栈 a。*栈内：c, a*

5. Push(S, y);// 入栈 k。*栈内：c, a, k*
6. Pop(S, x);// 弹出栈顶到 x，x = 'k'。*栈内：c, a*
7. Push(S, 't');// 入栈 t。*栈内：c, a, t*
8. Push(S, x);// 入栈 x（此时 x='k'）。*栈内：c, a, t, k*
9. Pop(S, x);// 弹出栈顶到 x，x = 'k'。*栈内：c, a, t*
10. Push(S, 's');// 入栈 s。*栈内：c, a, t, s*
11. while(!StackEmpty(S)) { Pop(S, y); printf(y); }// 依次弹出栈中所有元素并打印
    - 第一次：弹出 s，y='s'，打印 s。*栈内：c, a, t*
    - 第二次：弹出 t，y='t'，打印 t。*栈内：c, a*
    - 第三次：弹出 a，y='a'，打印 a。*栈内：c*
    - 第四次：弹出 c，y='c'，打印 c。*栈内：空*
12. printf(x);// 打印变量 x 的值（注意此时 x 在第 9 步后已变为'k'）

最终输出： stac（while 循环输出） + k（最后打印 x） = **stack**

3.4 简述以下算法的功能（栈的元素类型 SElemType int）

/* 缺失原题 */

(1) 算法 **Status algo1(Stack S)**的功能

正确答案：该算法的功能是**将栈 S 中所有元素的顺序进行逆置（反转）**。

过程分析：

1. while 循环：将栈 S 中的所有元素依次**弹出**，并按弹出顺序存入数组 A 中。由于栈是“后进先出”的，这个过程中，数组 A 记录的是原栈从**栈顶到栈底的逆序**。
2. for 循环：将数组 A 中的元素从下标 1 到 n 依次**压入**栈 S。由于数组 A 保存的是原栈的逆序，按正序压入栈后，新栈的顺序就变成了原栈的**完全颠倒**。
    - **效果示例**：若原栈从底到顶为 [1, 2, 3]（3 为栈顶），执行后栈变为 [3, 2, 1]（1 为栈顶）。

(2) 算法 Status algo2(Stack S, int e)的功能

正确答案: 该算法的功能是**删除栈 S 中所有值等于 e 的元素，并保持其余元素的原有相对顺序**。

过程分析:

1. 第一个 while 循环: 将栈 S 中的元素逐个弹出，检查其值 d。若 d != e，则将其压入**辅助栈 T**。此操作后，栈 S 为空，栈 T 中保存了所有不等于 e 的元素，但其顺序相对于原栈是**反的**。
2. 第二个 while 循环: 将辅助栈 T 中的元素逐个弹出，并压回栈 S。经过这次"倒手"，元素顺序再次反转，从而**恢复了不等于 e 的元素在原栈 S 中的原始相对顺序**。
    - **效果示例**: 若原栈从底到顶为 [1, 2, 3, 2, 4]，e=2，执行后栈变为 [1, 3, 4]，且 1、3、4 的相对顺序保持不变。

总结对比:

- algo1: **反转**整个栈的序列。
- algo2: **过滤**掉特定值的元素，并**保持**剩余元素的顺序。

【习题】3.9③ 试将下列递推过程改写为递归过程

原递推过程：

```
void ditui(int n) {
    int i;
    i = n;
    while(i > 1)
        printf(i--);
}
```

功能说明：此过程从 n 开始递减打印，直到 i 为 1 时停止（但原代码不会打印 1，因为循环条件是 i > 1）。即，输入 n，输出 n, n-1, ..., 2。 **改写后的递归过程：**

```
void digui(int n) {
    if (n > 1) {          // 递归终止条件对应原循环条件 i > 1
        printf(n);        // 先打印当前 n
        digui(n - 1);     // 递归调用，参数减 1
    }
    // 当 n <= 1 时，递归结束，直接返回
}
```

说明：

1. 递归终止条件 n > 1 对应原 while 循环的条件 i > 1。
2. 递归调用前执行 printf(n)，保证了打印顺序与原过程一致（从 n 到 2）。
3. 递归调用 digui(n - 1)实现了原过程中 i--的递减逻辑。

【习题】3.10 试将下列递归过程改写为非递归过程

原递归过程

```
void test(int &sum) {
    int x;
    scanf(x);
    if(x == 0)
        sum = 0;
    else {
        test(sum);
        sum += x;
    }
    printf(sum);
}
```

功能说明：此过程递归读取整数，直到遇到 0 为止。遇到 0 时，将 sum 清零，然后**在递归返回过程中**，将之前读入的非零数**逆序累加**到 sum 中，并逐层打印当前的 sum 值。

改写后的非递归过程：

```
void test_nonrecursive(int &sum) {
    Stack S;          // 使用一个栈来模拟递归的调用栈，保存读取的 x 值
    InitStack(S);
    int x;

    sum = 0; // 初始化 sum

    // 第一阶段：读取数据，直到遇到 0，并将非零值入栈
    scanf(x);
```

```
    while(x != 0) {
        Push(S, x);    // 将非零值压入栈中
        scanf(x);
    }
    // 循环结束时 x 为 0，此时对应递归中的 sum = 0

    // 第二阶段：出栈并累加，同时打印（模拟递归返回过程）
    printf(sum);        // 打印初始的 0（对应递归最深一层遇到 0 时打印的
sum）
    while(!StackEmpty(S)) {
        Pop(S, x);     // 按读取的逆序弹出数值
        sum += x;
        printf(sum);   // 打印每次累加后的结果
    }
}
```

说明：

1. 递归的“递去”过程是不断读入非零值并深入调用。这里用 while 循环和栈 S 模拟：读入非零值就压栈，遇到 0 则停止。
2. 递归的“归来”过程是在每一层将 x 累加到 sum 并打印。这里通过将栈中元素依次弹出、累加并打印来模拟。弹出的顺序正好是读入顺序的逆序，与原递归的返回过程一致。

根据您提供的图片内容，以下是《数据结构题集（C 语言版）》第三章习题 3.9、3.10 和 3.11 的参考答案。 /* 这是谁来 AIGC 的 ((*/

【习题】3.11 简述队列和栈这两种数据类型的相同点和差异处

相同点：

1. **逻辑结构：**两者都是**操作受限的线性表**，数据元素之间具有“一对一”的线性关系。
2. **存储结构：**都可以采用**顺序存储**（顺序栈、循环队列/顺序队列）或**链式存储**（链栈、链队列）实现。
3. **基本操作：**都包含**初始化、判空、插入（Push/EnQueue）、删除（Pop/DeQueue）、读取队头/栈顶元素**等基本操作。

主要差异：

特征	栈（Stack）	队列（Queue）
操作规则	后进先出（LIFO）	先进先出（FIFO）

插入/删除端	仅限**同一端** （栈顶）	在**两端**进行 （队尾插入，队头删除）
典型操作名	入栈(Push) 出栈(Pop)	入队(EnQueue) 出队(DeQueue)
读取元素	只能读栈顶(GetTop)	只能读队头(GetHead)
典型应用	函数调用、表达式求值、递归、括号匹配	任务调度、消息缓冲、广度优先搜索

核心区别总结： 栈的插入和删除都限制在**栈顶**，形成 LIFO 的特性；而队列的插入在**队尾**，删除在**队头**，形成 FIFO 的特性。这是二者最本质的逻辑差异。

第 4~6 章

【习题】4.5 给出函数输出结果

函数执行过程：

StrAssign(s, 'THIS IS A BOOK'); // s = "THIS IS A BOOK"

Replace(s, SubString(s, 3, 7), 'ESE ARE'); // 将 s 中第 3 个字符起长度为 7 的子串 "IS IS A" 替换为 "ESE ARE"，得到 s = "THESE ARE BOOK"

StrAssign(t, Concat(s, 'S')); // t = s + 'S' = "THESE ARE BOOKS"

StrAssign(u, 'XYXYXYXYXYXY'); // u = "XYXYXYXYXYXY"

StrAssign(v, SubString(u, 6, 3)); // 从 u 第 6 个字符起取长度为 3 的子串，v = "YXY"

StrAssign(w, 'W'); // w = 'W'

printf('t=', t, 'v=', v, 'u=', Replace(u, v, w)); // 输出

其中 Replace(u, v, w)是将 u 中所有子串 v 替换为 w。u 中 v 出现的位置：

- 第 2 个字符开始："YXY"→ 替换为 'W'
- 第 5 个字符开始："YXY"→ 替换为 'W'
- 第 8 个字符开始："YXY"→ 替换为 'W'

t=THESE ARE BOOKS
v=YXY
u=XWXWXW

【习题】4.6 将字符串 s 转化为 t

已知： s = '(XYZ)+*' t = '(X+Z)*Y' **转化步骤**（利用 Concat、SubString 操作，假设字符串下标从 1 开始）：

1. 取 s 的部分字符：
   - '('：SubString(s, 1, 1)
   - 'X'：SubString(s, 2, 1)
   - 'Y'：SubString(s, 3, 1)
   - 'Z'：SubString(s, 4, 1)
   - ')'：SubString(s, 5, 1)
   - '+'：SubString(s, 6, 1)
   - '*'：SubString(s, 7, 1)
2. 依次连接得到 t：

```
t = Concat( Concat( Concat( Concat( Concat( Concat( Concat(
        SubString(s, 1, 1),                  // '('
        SubString(s, 2, 1)),                 // 'X'
```

SubString(s, 6, 1)),   // '+'
SubString(s, 4, 1)),   // 'Z'
SubString(s, 5, 1)),   // ')'
SubString(s, 7, 1)),   // '*'
SubString(s, 3, 1));   // 'Y'

简化表示：

t = '(' + 'X' + '+' + 'Z' + ')' + '*' + 'Y'

【习题】4.7 求 next 和 nextval 函数值

定义：next[1] = 0，next[j] = 前 j-1 个字符的最长相等前后缀长度 + 1；nextval 在 next 基础上优化。

(1) s = 'aaab'（长度 4）

j	1 2 3 4
模式串	a a a b
next[j]	0 1 2 3
nextval[j]	0 0 0 3

计算过程：

- j=1：next[1]=0，nextval[1]=0
- j=2：next[2]=1；比较 s[2]=a 与 s[1]=a，相等 → nextval[2]=nextval[1]=0
- j=3：next[3]=2；比较 s[3]=a 与 s[2]=a，相等 → nextval[3]=nextval[2]=0
- j=4：next[4]=3；比较 s[4]=b 与 s[3]=a，不等 → nextval[4]=next[4]=3

(2) t = 'abcabaa'（长度 7）

j	1 2 3 4 5 6 7
模式串	a b c a b a a
next[j]	0 1 1 1 2 3 2

计算过程：

模式串 t = 'abcabaa'

已知：

t = a b c a b a a

长度 m = 7

下标从 1 开始。

第一步：计算 next[j]

`next[1] = 0`（规定）。

从 `j=2` 开始，`next[j]` 等于 **模式串前 `j-1` 个字符组成的子串的最长相等真前后缀长度 + 1**。

- **j = 2**：子串 `a`，长度 1，最长相等真前后缀长度 = 0，`next[2] = 0+1 = 1`。
- **j = 3**：子串 `ab`，真前后缀比较：
    - 前缀集合 `{a}`，后缀集合 `{b}`，无相同 → 长度 0，`next[3] = 1`。
- **j = 4**：子串 `abc`：
    - 前缀 `{a, ab}`，后缀 `{c, bc}`，无相同 → 长度 0，`next[4] = 1`。
- **j = 5**：子串 `abca`：
    - 前缀 `{a, ab, abc}`，后缀 `{a, ca, bca}`，相同的只有 `a` 长度 1 → `next[5] = 1+1 = 2`。
- **j = 6**：子串 `abcab`：
    - 前缀 `{a, ab, abc, abca}`，后缀 `{b, ab, cab, bcab}`，相同的 `ab` 长度 2 → `next[6] = 2+1 = 3`。
- **j = 7**：子串 `abcaba`：
    - 前缀 `{a, ab, abc, abca, abcab}`，后缀 `{a, ba, aba, caba, bcaba}`，相同的只有 `a` 长度 1 → `next[7] = 1+1 = 2`。

得到 next 数组：

```text
j      1 2 3 4 5 6 7
t[j]   a b c a b a a
next   0 1 1 1 2 3 2
```

第二步：计算 `nextval[j]`

规则：

1. `nextval[1] = 0`。
2. 对于 j ≥ 2：
   若 `t[j] == t[next[j]]`，则 `nextval[j] = nextval[next[j]]`；
   否则 `nextval[j] = next[j]`。

- **j=1**: `nextval[1] = 0`
- **j=2**: `next[2]=1`，比较 t[2]='b' 与 t[1]='a'，不同 → `nextval[2] = next[2] = 1`
- **j=3**: `next[3]=1`，比较 t[3]='c' 与 t[1]='a'，不同 → `nextval[3] = 1`

- **j=4**: `next[4]=1`，比较 t[4]='a' 与 t[1]='a'，相同 → `nextval[4] = nextval[1] = 0`
- **j=5**: `next[5]=2`，比较 t[5]='b' 与 t[2]='b'，相同 → `nextval[5] = nextval[2] = 1`
- **j=6**: `next[6]=3`，比较 t[6]='a' 与 t[3]='c'，不同 → `nextval[6] = 3`
- **j=7**: `next[7]=2`，比较 t[7]='a' 与 t[2]='b'，不同 → `nextval[7] = 2`

得到 nextval 数组：

```
nextval  0 1 1 0 1 3 2
```

(3) u = 'abcababbabcabaacbacba'（长度 20）结果（下标从 1 开始）：

- next： [0, 1, 1, 1, 2, 3, 2, 3, 1, 2, 3, 4, 5, 6, 7, 2, 1, 1, 2, 1]
- nextval：[0, 1, 1, 0, 1, 3, 1, 3, 0, 1, 1, 0, 1, 3, 7, 2, 1, 0, 2, 1]

计算过程：

计算 nextval 数组

规则：

1. `nextval[1] = 0`
2. 对于 `j ≥ 2`：
   - 如果 `u[j] == u[next[j]]`，则 `nextval[j] = nextval[next[j]]`
   - 否则 `nextval[j] = next[j]`

逐步计算：

1. **j=1**：`nextval[1] = 0`
2. **j=2**：`next[2]=1`
   `u[2]='b'`，`u[1]='a'` → 不同
   `nextval[2] = next[2] = 1`
3. **j=3**：`next[3]=1`
   `u[3]='c'`，`u[1]='a'` → 不同
   `nextval[3] = 1`
4. **j=4**：`next[4]=1`
   `u[4]='a'`，`u[1]='a'` → 相同
   `nextval[4] = nextval[1] = 0`

5. **j=5**：`next[5]=2`
   `u[5]='b'`，`u[2]='b'` → 相同
   `nextval[5] = nextval[2] = 1`
6. **j=6**：`next[6]=3`
   `u[6]='a'`，`u[3]='c'` → 不同
   `nextval[6] = 3`
7. **j=7**：`next[7]=2`
   `u[7]='b'`，`u[2]='b'` → 相同
   `nextval[7] = nextval[2] = 1`
8. **j=8**：`next[8]=3`
   `u[8]='b'`，`u[3]='c'` → 不同
   `nextval[8] = 3`
9. **j=9**：`next[9]=1`
   `u[9]='a'`，`u[1]='a'` → 相同
   `nextval[9] = nextval[1] = 0`
10. **j=10**：`next[10]=2`
    `u[10]='b'`，`u[2]='b'` → 相同
    `nextval[10] = nextval[2] = 1`
11. **j=11**：`next[11]=3`
    `u[11]='c'`，`u[3]='c'` → 相同
    `nextval[11] = nextval[3] = 1`
12. **j=12**：`next[12]=4`
    `u[12]='a'`，`u[4]='a'` → 相同
    `nextval[12] = nextval[4] = 0`
13. **j=13**：`next[13]=5`
    `u[13]='b'`，`u[5]='b'` → 相同
    `nextval[13] = nextval[5] = 1`
14. **j=14**：`next[14]=6`
    `u[14]='a'`，`u[6]='a'` → 相同
    `nextval[14] = nextval[6] = 3`
15. **j=15**：`next[15]=7`
    `u[15]='a'`，`u[7]='b'` → 不同
    `nextval[15] = 7`
16. **j=16**：`next[16]=2`
    `u[16]='c'`，`u[2]='b'` → 不同
    `nextval[16] = 2`
17. **j=17**：`next[17]=1`
    `u[17]='b'`，`u[1]='a'` → 不同
    `nextval[17] = 1`
18. **j=18**：`next[18]=1`
    `u[18]='a'`，`u[1]='a'` → 相同
    `nextval[18] = nextval[1] = 0`

19. **j=19**：`next[19]=2`
    `u[19]='c'`，`u[2]='b'` → 不同
    `nextval[19] = 2`
20. **j=20**：`next[20]=1`
    `u[20]='b'`，`u[1]='a'` → 不同
    `nextval[20] = 1`

最终 nextval 数组：

```
0 1 1 0 1 3 1 3 0 1 1 0 1 3 7 2 1 0 2 1
```

【习题】4.8 KMP 算法匹配全过程

主串 s：'ADBADABBAABADABBADADA' **模式串 pat**：'ADABBADADA'（长度 10）
(1) 求模式串 pat 的 nextval 函数值（下标从 1 开始）：

j	1 2 3 4 5 6 7 8 9 10
模式串	A D A B B A D A D A
next[j]	0 1 1 2 1 1 2 3 4 3
nextval[j]	0 1 0 2 1 0 1 0 4 0

计算过程：

- j=3: next[3]=1，比较 pat[3]=A 与 pat[1]=A，相等 → nextval[3]=nextval[1]=0
- j=6: next[6]=1，比较 pat[6]=A 与 pat[1]=A，相等 → nextval[6]=nextval[1]=0
- j=8: next[8]=3，比较 pat[8]=A 与 pat[3]=A，相等 → nextval[8]=nextval[3]=0
- j=10: next[10]=3，比较 pat[10]=A 与 pat[3]=A，相等 → nextval[10]=nextval[3]=0
- 其余情况 nextval[j] = next[j]

KMP 算法匹配全过程（主串 s='ADBADABBAABADABBADADA'，使用 nextval 进行匹配）： 初始化：主串指针 i=1，模式串指针 j=1。

1. i=1,j=1: s[1]=A, p[1]=A，匹配，i=2,j=2
2. i=2,j=2: s[2]=D, p[2]=D，匹配，i=3,j=3
3. i=3,j=3: s[3]=B, p[3]=A，不匹配，j=nextval[3]=0 → i=4,j=1
4. i=4,j=1: s[4]=A, p[1]=A，匹配，i=5,j=2

5. i=5,j=2: s[5]=D, p[2]=D，匹配，i=6,j=3
6. i=6,j=3: s[6]=A, p[3]=A，匹配，i=7,j=4
7. i=7,j=4: s[7]=B, p[4]=B，匹配，i=8,j=5
8. i=8,j=5: s[8]=B, p[5]=B，匹配，i=9,j=6
9. i=9,j=6: s[9]=A, p[6]=A，匹配，i=10,j=7
10. i=10,j=7: s[10]=A, p[7]=D，不匹配，j=nextval[7]=1
    - 继续比较 s[10] 与 p[1]：s[10]=A, p[1]=A，匹配，i=11,j=2
11. i=11,j=2: s[11]=B, p[2]=D，不匹配，j=nextval[2]=1
    - 继续比较 s[11] 与 p[1]：s[11]=B, p[1]=A，不匹配，j=nextval[1]=0 → i=12,j=1
12. i=12,j=1: s[12]=A, p[1]=A，匹配，i=13,j=2
13. i=13,j=2: s[13]=D, p[2]=D，匹配，i=14,j=3
14. i=14,j=3: s[14]=A, p[3]=A，匹配，i=15,j=4
15. i=15,j=4: s[15]=B, p[4]=B，匹配，i=16,j=5
16. i=16,j=5: s[16]=B, p[5]=B，匹配，i=17,j=6
17. i=17,j=6: s[17]=A, p[6]=A，匹配，i=18,j=7
18. i=18,j=7: s[18]=D, p[7]=D，匹配，i=19,j=8
19. i=19,j=8: s[19]=A, p[8]=A，匹配，i=20,j=9
20. i=20,j=9: s[20]=D, p[9]=D，匹配，i=21,j=10
21. i=21,j=10: s[21]=A, p[10]=A，匹配，i=22,j=11

此时 j=11 > 模式串长度 10，匹配成功。匹配起始位置为 i - 10 = 12。

自检：

1. 检查 next 数组

模式串 `ADABBADADA`（长度 10，下标从 1 开始）
计算正确 next 数组（`next[1]=0`，`next[j] = 前 j-1 个字符的最长相等真前后缀长度 + 1`）。

j	pat[j]	前缀子串(1..j-1)	最长相等真前后缀长度	next[j]
1	A	-	-	0
2	D	A	0	1
3	A	AD	0	1
4	B	ADA	1（`A`）	2
5	B	ADAB	0	1
6	A	ADABB	0	1
7	D	ADABBA	1（`A`）	2
8	A	ADABBAD	2（`AD`）	3
9	D	ADABBAD A（注意空格为分隔符）	3（`ADA`）	4
10	A	ADABBADAD	2（`AD`）	3

所以正确 next 为：

`[0, 1, 1, 2, 1, 1, 2, 3, 4, 3]`

与题目给出的 next 一致 ✓

---

2. 检查 nextval 数组

规则：`nextval[1] = 0`，
若 `pat[j] == pat[next[j]]`，则 `nextval[j] = nextval[next[j]]`，否则 `nextval[j] = next[j]`。

1. j=1: 0
2. j=2: next=1, pat[2]=D ≠ pat[1]=A → 1 ✓
3. j=3: next=1, pat[3]=A == pat[1]=A → `nextval[3] = nextval[1] = 0` ✓
4. j=4: next=2, pat[4]=B ≠ pat[2]=D → 2 ✓
5. j=5: next=1, pat[5]=B ≠ pat[1]=A → 1 ✓
6. j=6: next=1, pat[6]=A == pat[1]=A → `nextval[6] = nextval[1] = 0` ✓
7. j=7: next=2, pat[7]=D == pat[2]=D → `nextval[7] = nextval[2] = 1`
   题目给 `nextval[7]=1` ✓
8. j=8: next=3, pat[8]=A == pat[3]=A → `nextval[8] = nextval[3] = 0` ✓
9. j=9: next=4, pat[9]=D ≠ pat[4]=B → `nextval[9] = next[9] = 4`
   题目给 `nextval[9]=4` ✓
10. j=10: next=3, pat[10]=A == pat[3]=A → `nextval[10] = nextval[3] = 0` ✓

所以题目 nextval 数组正确 ✓

---

3. 检查 KMP 匹配过程

主串 `s = 'ADBADABBAABADABBADADA'` 长度 21，下标从 1 开始。
题目匹配步骤中，关键检查几个跳转位置：

- 第 3 步：i=3,j=3 不匹配，j=nextval[3]=0，此时 i 应 +1，j=1，i=4 ✓

- 第 10 步：i=10,j=7 不匹配，j=nextval[7]=1，继续与 s[10] 比较 ✓
- 第 11 步：i=11,j=2 不匹配，j=nextval[2]=1，比较 s[11] 与 pat[1] 不匹配，j=nextval[1]=0，i=12,j=1 ✓

- 后面完全匹配，最终 i=22,j=11 结束，起始位置 22-10=12 ✓

注意，主串从 1 开始，匹配起始位置为 `i - pat.length = 22-10 = 12`，表示模式串在主串的第 12 个位置开始匹配成功。
检查主串从 12 到 21 的字符：

`s[12..21] = 'ADABBADADA'` 与模式串完全一致 ✓

结论： 这道题给出的 nextval 数组 和 KMP 匹配过程 都是正确的。

5.3 按高下标优先存储方式，顺序列出数组 $A_{2\times2\times3\times3}$ 中所有元素 a_{ijkl}

“高下标优先” 又称“**以最右的下标为主序**”，即在存储和访问时，**最右边（最低位）的下标变化最**慢，最左边（最高位）的下标变化最**快**。 对于四维数组 A[2][2][3][3]，其下标变化顺序为：i(第一维) → j(第二维) → k(第三维) →l(第四维)。

5.6 三对角矩阵 $(a_{ij})_{n\times n}$ 存储于数组 B[3][n]中，使得 B[u][v]=a_{ij}，推导从 (i,j)到 (u,v)的下标变换公式

1. 分析矩阵与存储结构：

- 三对角矩阵中，非零元素 aij 满足 ｜i－j｜≤1，即位于主对角线及其上、下各一条对角线上。
- 数组 B[3][n]的第一维大小 3 对应三条对角线，第二维大小 n 对应每条对角线上最多有 n 个元素。

2. **确定 u(属于哪条对角线)：**

- 当 i－j=1 时，元素位于**下对角线**（i>j）。令其对应 u=0。
- 当 i－j=0 时，元素位于**主对角线**。令其对应 u=1。
- 当 i－j=－1 时，元素位于**上对角线**（i<j）。令其对应 u=2。
- 即：u=(i－j)+1，其中 (i－j)∈{－1,0,1}。

3. **确定 v(元素在该对角线上的位置)：**

v =j-1

5.11 利用广义表的GetHead和GetTail操作写出函数表达式，把原子banana从给定的广义表中分离出来。广义表有L1到L7。

根据广义表的 **GetHead** 和 **GetTail** 操作，将原子 **banana** 从各广义表中分离出来的函数表达式如下：

1. **L_1 = (apple, pear, banana, orange)**
   GetHead(GetTail(GetTail(L_1)))
2. **L_2 = ((apple, pear), (banana, orange))**
   GetHead(GetHead(GetTail(L_2)))
3. **L_3 = (((apple)), (pear), (banana), (orange))**

   GetHead(GetHead(GetTail(GetTail(GetHead(L_3)))))

4. **L_4 = (apple, (pear), ((banana)), ((orange)))**
   GetHead(GetHead(GetHead(GetTail(GetTail(L_4)))))
5. **L_5 = (((apple)), ((pear)), (banana), orange)**
   GetHead(GetHead(GetTail(GetTail(L_5))))
6. **L_6 = (((apple, pear), banana), orange)**
   GetHead(GetTail(GetHead(L_6)))
7. **L_7 = (apple, (pear, (banana), orange))**
   GetHead(GetHead(GetTail(GetHead(GetTail(L7)))))

对于 L7 = (apple, (pear, (banana), orange))，分离出 banana 的操作序列如下：

1. **GetTail(L7)**：取出 L7 第一个元素 apple 之后剩余的部分。
   - **结果**：((pear, (banana), orange))

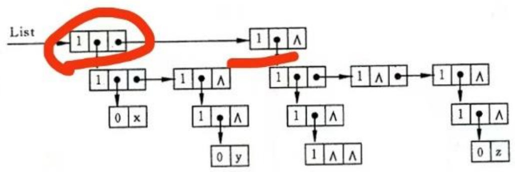
- *说明：注意结果是一个子表，所以外层有括号。*

2. GetHead(GetTail(L7))：取上述结果子表的第一个元素。
   - **结果：** (pear， (banana)， orange)
   - *说明：这是 L7 的第二个元素，即一个子表。*
3. GetTail(GetHead(GetTail(L7)))：取出子表 (pear， (banana)， orange)中第一个元素 pear 之后剩余的部分。
   - **结果：** ((banana)， orange)
   - *说明：现在得到了一个包含两个元素的新子表。*
4. GetHead(GetTail(GetHead(GetTail(L7))))：取上述结果子表的第一个元素。
   - **结果：** (banana)
   - *说明：banana 本身是一个原子，但作为子表的元素，它被括号包围。*
5. GetHead(GetHead(GetTail(GetHead(GetTail(L7)))))：最后，取出子表 (banana)中的第一个元素，也就是原子 banana 本身。
   - **最终结果：** banana

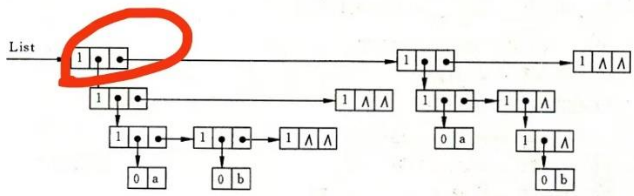
5.13 根据广义表的存储结构图写出广义表。

已知以下各图为广义表的存储结构图，其结点结构和 5.12 题相同。写出各图表示的广义表。

(1)



(2)



(1) ((x,(y)), ((())), (),(z)) 注意

1	^	^

为 (())

(2) (((a,b,()),()),(a,(b)),())

对于给出的例子：

(1) ((x,(y)),（((())), (),(z)))

分析结构：

1. 最外层：两个元素的表
2. 第一个元素：`(x,(y))`
    - x 是原子
    - `(y)`是只有一个原子 y 的表
3. 第二个元素：`（((())), (),(z))`
    - `((()))`：三层嵌套的空表
    - `()`：空表
    - `(z)`：只有一个原子 z 的表

(2) (((a,b,()),()),(a,(b)),())

分析结构：

1. 最外层：三个元素的表
2. 第一个元素：`((a,b,()),())`
    - 内层：`(a,b,())` 和 `()`
3. 第二个元素：`(a,(b))`
4. 第三个元素：`()`

解题技巧：

1. **从根部开始**：找到广义表的根结点
2. **深度优先遍历**：按照存储结构遍历每个分支
3. **注意括号匹配**：每个子表都需要用括号括起来

4. **区分原子和子表**：原子直接写，子表要加括号

5. **处理空表**：空表表示为`()`

【习题】6.1 根据给定的边集合 <I,M>, <I,N>, <E,I>, <B,E>, <B,D>, <A,B>, <G,J>, <G,K>, <C,G>, <C,F>, <H,L>, <C,H>, <A,C>，可构建出如下树结构：

```
        A
      /    \
     B      C
    / \     /|\
   D  E   G F H
       |  / \   \
       I K  J    L
      / \
    M    N
```

问题解答：

1. **根结点**：A
2. **叶子结点**：D、M、N、J、K、F、L （说明：叶子结点即没有子结点的结点）
3. **结点 G 的双亲**：C
4. **结点 G 的祖先**：A、C （说明：祖先指从根到该结点路径上的所有结点）
5. **结点 G 的孩子**：J、K
6. **结点 E 的子孙**：I、M、N
7. **结点 E 的兄弟**：D
8. **结点 F 的兄弟**：G、H （说明：兄弟指具有相同双亲的结点）
9. **结点 B 的层次号**：2
10. **结点 N 的层次号**：5 （说明：根结点 A 的层次为 1，向下逐层递增）
11. **树的深度**：5 （说明：树中结点的最大层次为树的深度）
12. **以结点 C 为根的子树的深度**：3 （说明：在该子树中，C 为第 1 层，叶子结点 J、K、L 位于第 3 层）

【习题】6.3 分别画出具有 3 个结点的树 和 3 个结点的二叉树 的所有不同形态。

一、具有 3 个结点的树的所有不同形态

树（普通树）中，结点的子树**没有左右顺序**之分，同一双亲下的孩子之间是无序的。 3 个结点的树共有 **2 种** 不同形态：

1. **根结点有两个孩子**（两个叶子结点）

```
  0
 / \
0   0
```

2. **根结点有一个孩子，该孩子又有一个孩子**（形成一条链）

```
    0
   /
  0
 /
0
```

注：链的形状只有一种，因为子树无序，不能区分“左链”或“右链”。

二、具有 3 个结点的二叉树的所有不同形态

二叉树中，结点的子树**有严格的左右顺序**，即使只有一个子树也必须指明是左子树还是右子树。 3 个结点的二叉树共有 **5 种** 不同形态：

1. **左斜链**（所有结点只有左孩子）

```
    O
   /
  O
 /
O
```

2. 根有左孩子，左孩子有右孩子

```
  O
 /
O
 \
  O
```

3. 根有左、右两个孩子（满二叉树形态）

```
        O
       / \
      O   O
```

4. 根有右孩子，右孩子有左孩子

```
        O
          \
           O
          /
        O
```

5. 右斜链（所有结点只有右孩子）

```
        O
          \
           O
             \
              O
```

总结

3 个结点的树：2 种 形态。

3 个结点的二叉树：5 种 形态。

关键区别：二叉树强调左右子树的有序性，因此形态更多；而普通树中子树无序，形态更少。

【习题】6.4 对于深度为 H 的满 k 叉树（第 H 层为叶子结点，其余各层每个结点都有 k 棵非空子树），按层次顺序从 1 开始对全部结点编号，有以下结论：

1) 各层的结点数目是多少？
   设根结点在第 1 层，则第 h 层（$1 \leqslant h \leqslant H$）的结点数目为：**第 h 层结点数=$k^{h-1}$**
2) 编号为 p 的结点的父结点（若存在）的编号是多少？

   编号为 p 的结点（$p > 1$）的父结点编号为：**父结点编号=$\lceil (p-1)/k \rceil$**

   其中 $\lceil\cdot\rceil$ 表示向上取整。当 $p = 1$（根结点）时，无父结点。
3) 编号为 p 的结点的第 i 个儿子结点（若存在）的编号是多少？

   编号为 p 的结点的第 i 个儿子结点（$1 \leqslant i \leqslant k$）的编号为：**第 i 个儿子编号=$(p-1)\cdot k+i+1$**
4) 编号为 p 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？

   条件：**结点 p** 不是其父结点的第 k 个孩子（即不是**最右侧孩子**），等价于 $(p-1) \bmod k$ 不等于 0

   右兄弟编号：若满足上述条件，则**右兄弟的编号为 $p+1$**。

【习题】6.5 已知一棵度为 k 的树中有 n1 个度为 1 的结点， n2 个度为 2 的结点，...， nk 个度为 k 的结点。设该树中叶子结点（度为 0 的结点）个数为 n0。

求解过程如下：

1. **总结点数（n）：**

   n=n0+n1+n2+⋯+nk

2. **总分支数（边数）：** 在树结构中，除根结点外，每个结点都对应一条从其双亲指向它的边，因此树的总边数为 n−1。 总边数也等于所有结点的**度之和**（因为一个度为 d 的结点会贡献 d 条边）。所以：

   n−1=0⋅n0+1⋅n1+2⋅n2+⋯+k⋅nk

3. **建立方程并求解 n0：** 将步骤 1 中的 n=n0+n1+n2+⋯+nk 代入步骤 2 的等式：

   (n0+n1+n2+⋯+nk)−1=n1+2n2+⋯+knk

   整理等式，将度之和项移到一边：

   n0+n1+n2+⋯+nk−1=n1+2n2+⋯+knk

   n0−1=(n1+2n2+⋯+knk)−(n1+n2+⋯+nk)

   n0−1=(1−1)n1+(2−1)n2+(3−1)n3+⋯+(k−1)nk

   n0=1+0⋅n1+1⋅n2+2⋅n3+⋯+(k−1)nk

最终答案： 该树中叶子结点的个数 n0 为： $\boldsymbol{n_0 = 1 + \sum_2^k (i-1) n_i}$

或等价地写作： n0=1+n2+2n3+3n4+⋯+(k−1)nk

【习题】6.19 与各树对应的二叉树

将普通树转换为二叉树的标准方法是 **"左孩子右兄弟"表示法**（又称二叉链表表示法）。转换规则如下：

- 二叉树的每个结点有两个指针：lchild 和 rchild。
- 原树中结点的 **第一个孩子** 转换为二叉树中该结点的 **左孩子 (**lchild**)**。
- 原树中结点的 **下一个兄弟**（同一双亲的下一个子结点）转换为二叉树中该结点的 **右孩子 (**rchild**)**。

按照此规则，图 (a) 至 (d) 对应的二叉树如下：

a) 单结点树

原树：

```
  A
```

对应二叉树：

```
    A
   / \
NULL NULL
```

文字描述：二叉树仅包含一个结点 A，其 lchild 与 rchild 均为空。

b) 根结点有两个孩子的树

原树：

```
   A
  / \
 B   C
```

对应二叉树：

```
   A
  /
 B
  \
   C
```

文字描述：

- A 的左孩子（lchild）指向其第一个孩子 B，右孩子（rchild）为空（根无双亲）。
- B 的左孩子为空（无子结点），右孩子（rchild）指向其兄弟 C。
- C 的左右孩子均为空。

c)　结构与（b）相同的树
对应二叉树与（b）完全相同。

```
    A
   /
  B
   \
    C
```

d)　多层多分支的树

原树（根据描述）：

```
          A
       /  |  \
      B   C   D
         / / \ \
        E F  G H        //这里四个都是 C 的子节点
      / \     |
     I   J    K
```

对应二叉树：

```
        A
       /
      B
       \
        C
       / \
      E   D
     / \
    I   F
    \    \
    J     G
         / \
        K   H
```

这种转换方法的核心思想是：将树的多分支结构转换为二叉树的左右分支结构，通过左指针表示"父子关系"，右指针表示"兄弟关系"。

- 原树中结点的第一个孩子 → 二叉树中该结点的左孩子(lchild)
- 原树中结点的下一个兄弟 → 二叉树中该结点的右孩子(rchild)

关键性质：

1. **根结点没有右孩子**
    - 因为根结点没有兄弟
    - 二叉树的根对应原树的根

2. **叶子结点的特征**
    - 原树的叶子结点 → 二叉树中**左孩子为 NULL 的结点**
    - 因为叶子结点没有孩子，所以左指针为 NULL
3. **二叉树的高度**
    - 二叉树的高度 = 原树的深度（层数）
    - 因为最深的路径是沿着"第一个孩子链"向下
4. **二叉树的宽度**
    - 最宽的一层对应原树中某层的所有兄弟链

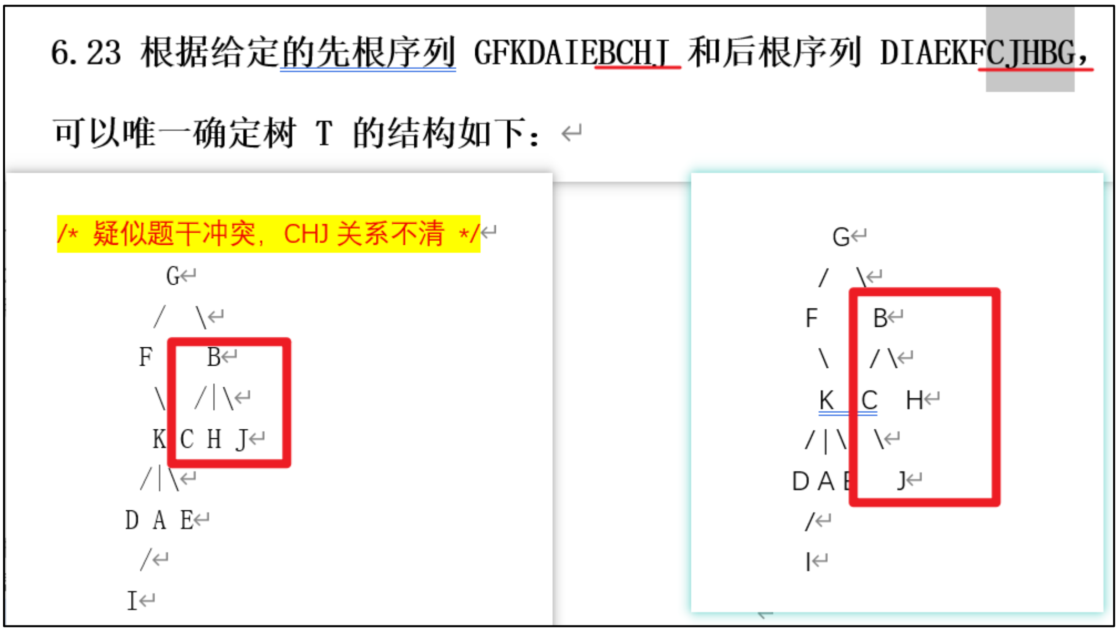
6.21 见书上答案。

/* 题是? */

6.23 根据给定的先根序列 GFKDAIEBCHJ 和后根序列 DIAEKFCJHBG，可以唯一确定树 T 的结构如下：

```
      G
     / \
    F   B
     \ /|\
     K C H J
    /|\
   D A E
    /
   I
```

```
     G
    /  \
   F     B
    \    / \
     K   C   H
   / | \   \
  D A E    J
   /
   I
```



6.23 根据给定的先根序列 GFKDAIEBCHJ 和后根序列 DIAEKFCJHBG，可以唯一确定树 T 的结构如下：

/* 疑似题干冲突，CHJ 关系不清 */

```
      G
    /   \
   F     B
    \   /|\
     K C H J
   /|\
  D A E
   /
  I
```

```
      G
    /  \
   F     B
    \   / \
     K  C   H
   / | \    \
  D A E      J
   /
   I
```

遍历验证：

- 先根序列：G → F → K → D → A → I → E → B → C → H → J，即 GFKDAIEBCHJ。
- 后根序列：D → I → A → E → K → F → C → J → H → B → G，即 DIAEKFCJHBG。

【习题】6.26 根据哈夫曼算法，为给定频率的字母构造哈夫曼树及编码如下。编码不唯一，但加权路径长度（WPL）相同。

频率 假设字母

0.07 a

0.19 b

0.02 c

0.06 d

频率	假设字母
0.32	e
0.03	f
0.21	g
0.10	h

哈夫曼树构建步骤（合并过程）：

1. 初始节点：c(0.02), f(0.03), d(0.06), a(0.07), h(0.10), b(0.19), g(0.21), e(0.32)。
2. 合并最小两个 0.02(c) 与 0.03(f) → 节点 N1(0.05)。
3. 合并最小两个 0.05(N1) 与 0.06(d) → 节点 N2(0.11)。
4. 合并最小两个 0.07(a) 与 0.10(h) → 节点 N3(0.17)。
5. 合并最小两个 0.11(N2) 与 0.17(N3) → 节点 N4(0.28)。
6. 合并最小两个 0.19(b) 与 0.21(g) → 节点 N5(0.40)。
7. 合并最小两个 0.28(N4) 与 0.32(e) → 节点 N6(0.60)。
8. 最后合并 0.40(N5) 与 0.60(N6) → 根节点 Root(1.00)。

哈夫曼编码（约定：合并时左子树赋 0，右子树赋 1）：

字母	频率	编码
e	0.32	11
g	0.21	01
b	0.19	00
h	0.10	1011
a	0.07	1010
d	0.06	1001
f	0.03	10001
c	0.02	10000

加权路径长度（WPL）计算： WPL = 0.32×2 + 0.21×2 + 0.19×2 + 0.10×4 + 0.07×4 + 0.06×4 + 0.03×5 + 0.02×5 = **2.61** **说明：**编码结果与合并顺序有关（例如将 0.19 与 0.21 合并，或与 0.28 合并，会得到不同的编码分布），但 WPL 均为最小值 2.61。

对比分析哈夫曼编码方案和使用 0-7 的二进制编码（即**等长编码**）方案各自的优缺点。

固定长度编码（0-7 二进制编码）

- 由于实例中有 8 个字母，每个字母需用 3 位二进制表示（因 23=8）。
- **平均码长** = 3 位/符号。

根据 6.26 题的计算，哈夫曼编码的加权路径长度（平均码长）为 **2.61 位/符号**。

压缩率：比固定长度编码节省约 **13%** 的存储/传输空间（因 1−2.61/3≈13%）。哈夫曼编码通过为高频字符（如 e:0.32）分配短码（如"00"），为低频字符（如 c:0.02）分配长码（如"1110"），实现了更高的压缩效率

题目核对：

1. 核对哈夫曼树构建过程

给出的频率（按数值升序排列）：

text

c = 0.02

f = 0.03

d = 0.06

a = 0.07

h = 0.10

b = 0.19

g = 0.21

e = 0.32

第一步

合并 c(0.02) 与 f(0.03) → N1(0.05)

剩余：N1(0.05), d(0.06), a(0.07), h(0.10), b(0.19), g(0.21), e(0.32)

第二步

最小的两个是 N1(0.05) 与 d(0.06) → N2(0.11)

剩余：a(0.07), h(0.10), N2(0.11), b(0.19), g(0.21), e(0.32)

第三步

最小的两个是 a(0.07) 与 h(0.10) → N3(0.17)

剩余：N2(0.11), N3(0.17), b(0.19), g(0.21), e(0.32)

第四步

最小的两个是 N2(0.11) 与 N3(0.17) → N4(0.28)

剩余：b(0.19), g(0.21), N4(0.28), e(0.32)

第五步

最小的两个是 b(0.19) 与 g(0.21) → N5(0.40)

剩余：N4(0.28), e(0.32), N5(0.40)

第六步

最小的两个是 N4(0.28) 与 e(0.32) → N6(0.60)

剩余：N5(0.40), N6(0.60)

第七步

合并 N5(0.40) 与 N6(0.60) → 根(1.00)

题目给出的合并顺序和这个完全一致。

---

2. 核对哈夫曼编码

按题中约定：合并时左子树赋 0，右子树赋 1。我们来复原树的形态。

从最后一步反推：

第 7 步：

根

├── N5(0.40)

└── N6(0.60)

第 6 步：

N6 是 N4(0.28) + e(0.32)：

假设 N4 是左(0)，e 是右(1)

→e 的编码前缀是 1（现在看最终编码表里 e 是 11，需要符合树结构，可能是树根的右(N6)再往右(e)就是 11 吗？稍后检查）

我们先正向推导编码路径：

正向编码推导

构建时设定左 0，右 1：

1. N5 = b 与 g 合并：
   - b(0.19) 在 N5 中的位置？
     从第 5 步看 b 与 g 合并成 N5，设 b 左(0)→0…，g 右(1)→1…。
     最终 b 的编码是 00，说明后面没有其他位，即 N5 在最后的根里是左子树(0)，所以 b 最终 00，g 是 01。
2. N6 = N4 与 e 合并：
   根：左 N5 (0)，右 N6 (1)
   N6：左 N4 (0)，右 e (1)
   所以 e 的路径 = N6 的 1 + N6 内部的右 1→11 ✅
3. N4 = N2 与 N3 合并：
   在 N6 里 N4 是左 (0)，所以 N4 路径前缀 = 10（根右 N6=1，再左 N4=0）
4. N2 与 N3 合并时，谁左谁右？
   N2(0.11) 与 N3(0.17)：题目假设 N2 左(0)，N3 右(1)
   所以：
   - N2 的路径前缀 = N4 前缀 10 加 N4 内的左 0→100
   - N3 的路径前缀 = N4 前缀 10 加 N4 内的右 1→101
5. N3 = a 与 h 合并：
   在 N3 内，a(0.07) 左(0)，h(0.10) 右(1)
   所以：
   - a = 1010 ✅
   - h = 1011 ✅
6. N2 = N1 与 d 合并：
   N2 内：N1 左(0)，d 右(1)
   所以：
   - d = N2 前缀 100 + 1→1001 ✅
   - N1 前缀 = 1000
7. N1 = c 与 f 合并：
   在 N1 内：c(0.02) 左(0)，f(0.03) 右(1)
   所以：
   - c = 10000 ✅
   - f = 10001 ✅

全部字母编码与题目给出的表一致。

3. 计算 WPL

WPL = 各符号频率 × 码长（位数）:

- e(0.32) 码长 2 → 0.64
- g(0.21) 码长 2 → 0.42
- b(0.19) 码长 2 → 0.38
- h(0.10) 码长 4 → 0.40
- a(0.07) 码长 4 → 0.28
- d(0.06) 码长 4 → 0.24
- f(0.03) 码长 5 → 0.15
- c(0.02) 码长 5 → 0.10

总和 = 0.64+0.42=1.06，+0.38=1.44，+0.40=1.84，+0.28=2.12，+0.24=2.36，+0.15=2.51，+0.10=2.61 ✅

题目 WPL=2.61 正确。

---

4. 比较固定长度编码

8 个字母用 3 位编码，平均长度 = 3。

压缩率计算

(1 − 2.61/3) × 100% ≈ (1 − 0.87) × 100% ≈ 13% 节省空间，题中这个数字正确。

第 7 章

【习题】7.1 已知如下图所示的有向图，请给出该图的

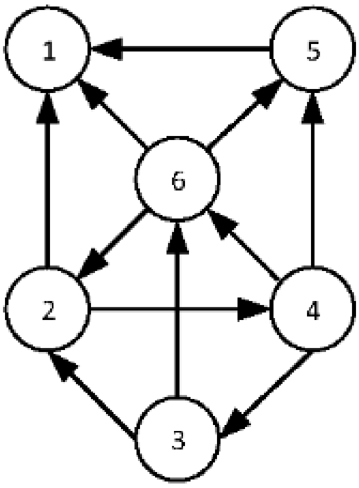
(1)每个顶点的入/出度;

(2)邻接矩阵;

(3)邻接表;

(4)逆邻接表;

(5)强连通分量



答:

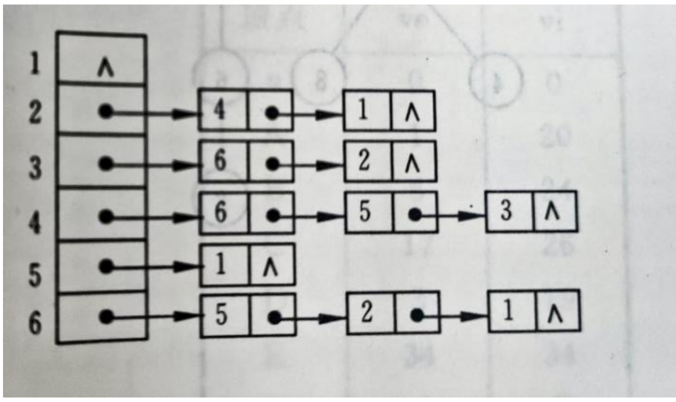
（1）每个顶点的入/出度

顶点	1	2	3	4	5	6
入度	3	2	1	1	2	2
出度	0	2	2	3	1	3

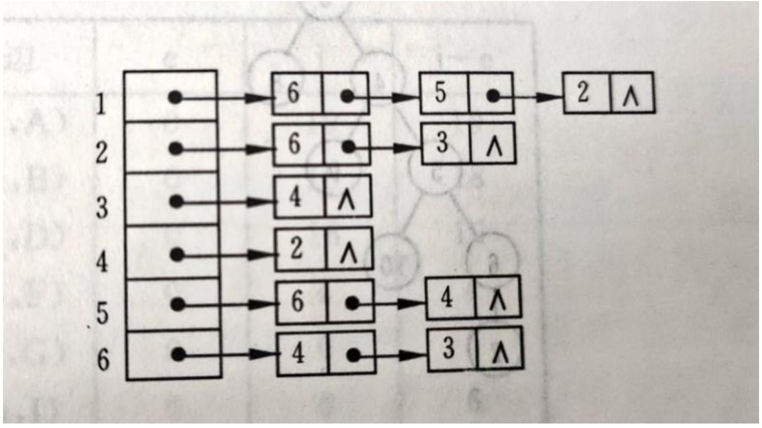
（2）邻接矩阵

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

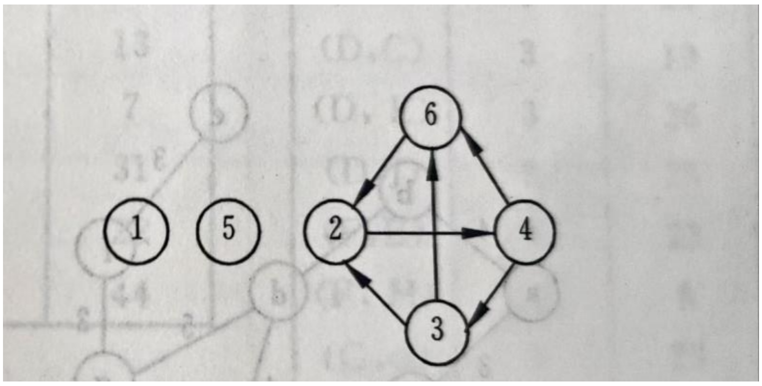
（3）邻接表



（4）逆邻接表



（5）有 3 个强连通分量

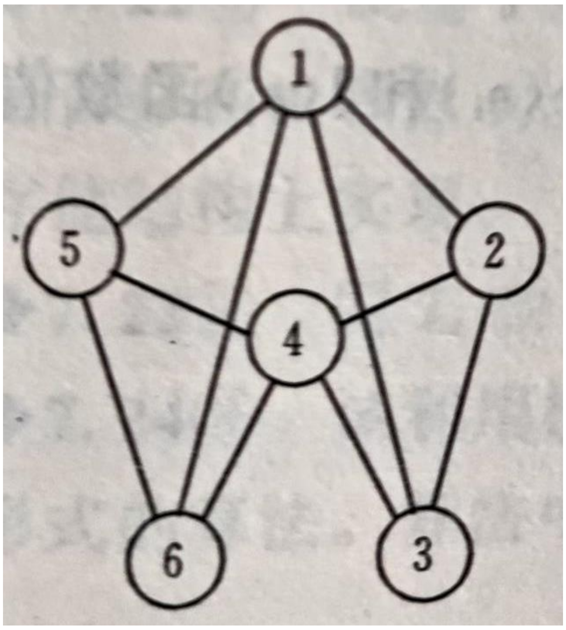


【习题】7.2 已知有向图的邻接矩阵为$A_{n\times n}$，试问每一个$A_{n\times n}^{k}$，(k=1，2，・・・，n)，各具有何种实际含义？

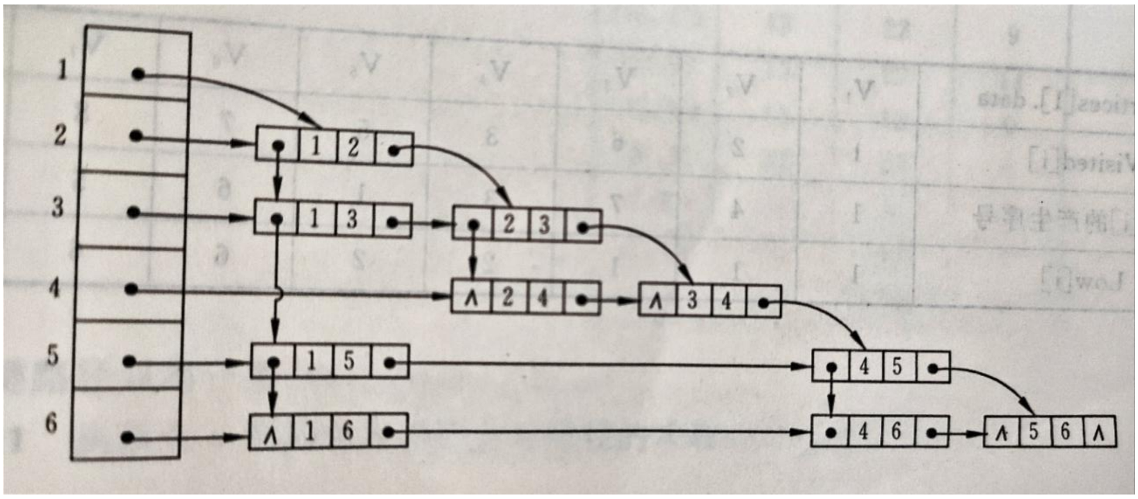
答：数 $a_{ij}^{(k)} = A_{n\times n}^{k}$ ，则 $a_{ij}^{(k)}$ 为由 i 到 j 的长度为 k 的路径数。注意，不能理解为简单路径。

7.3 题答案

画出下图所示的无向图的**邻接多重表**，*使得其中每个无向边结点中第一个顶点号小于第二个顶点号，且每个顶点的各邻接边的链接顺序，为它所邻接到的顶点序号由小到大的顺序*。列出**深度优先和广度优先**搜索遍历该图所得顶点序列和边的序列。



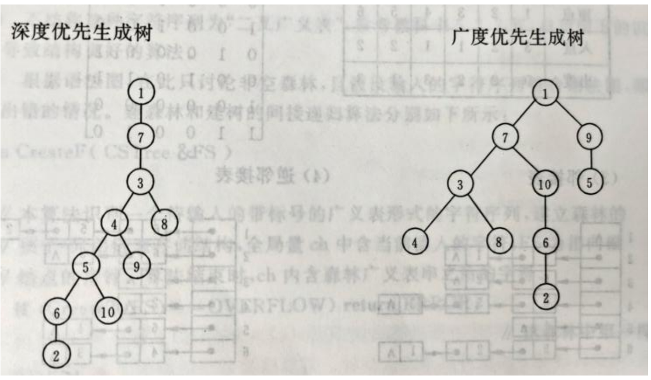
答：



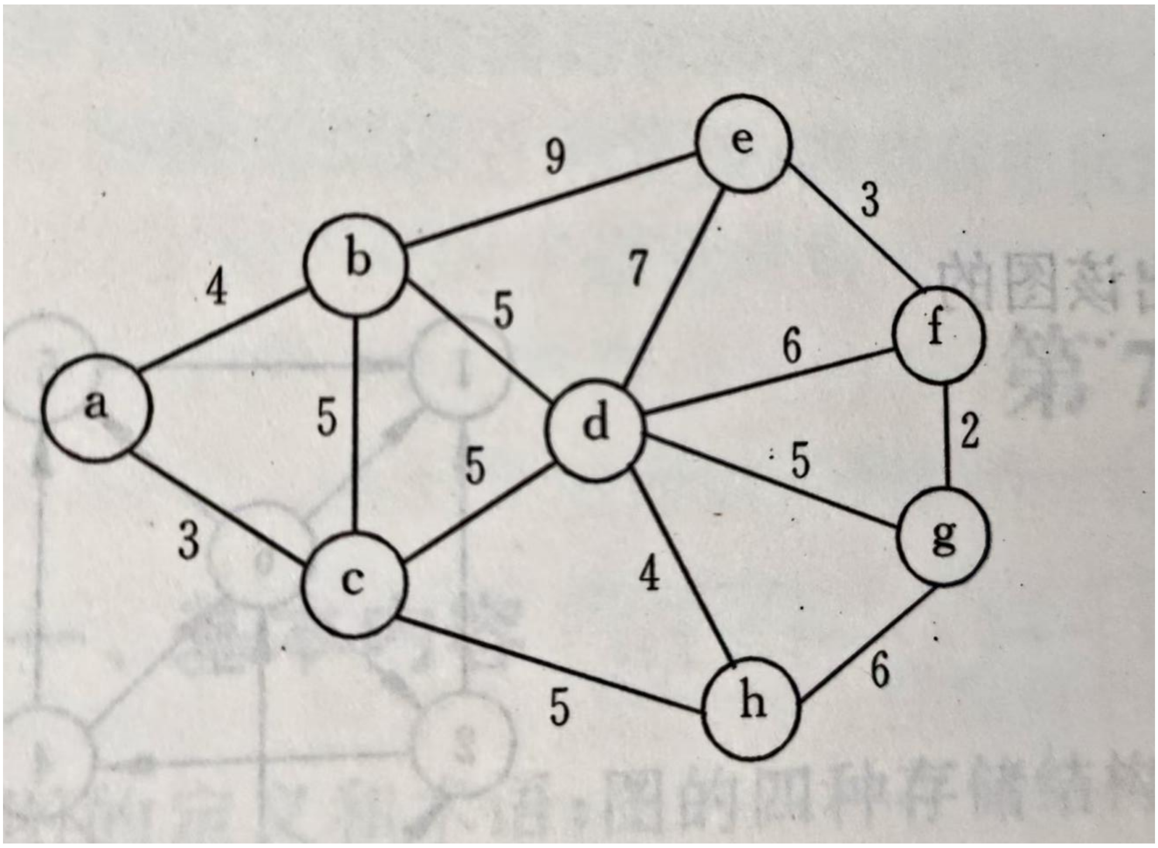
【习题】7.5 已知以二维数组表示的图的邻接矩阵如下图所示。试分别画出自顶点 1 出发进行遍历所得的深度优先生成树和广度优先生成树。

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	0	1	0
2	0	0	1	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	1	0	0	0	1	0
5	0	0	0	0	0	1	0	0	0	1
6	1	1	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	1
8	1	0	0	1	0	0	0	0	1	0
9	0	0	0	0	1	0	1	0	0	1
10	1	0	0	0	0	1	0	0	0	0

答：



【习题】7.7 请对下图的无向带权图，



（1） 写出它的邻接矩阵，并按普里姆(Prim)算法求其最小生成树；

（2） 写出它的邻接表，并按克鲁斯卡尔(Kruskal)算法求其最小生成树。

答：

（1）

邻接矩阵

$$\begin{bmatrix} 0 & 4 & 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 5 & 9 & 0 & 0 & 0 \\ 3 & 5 & 0 & 5 & 0 & 0 & 0 & 5 \\ 0 & 5 & 5 & 0 & 7 & 6 & 5 & 4 \\ 0 & 9 & 0 & 7 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 6 & 3 & 0 & 2 & 0 \\ 0 & 0 & 0 & 5 & 0 & 2 & 0 & 6 \\ 0 & 0 & 5 & 4 & 0 & 0 & 6 & 0 \end{bmatrix}$$

一、Prim 算法求最小生成树

从顶点 a 开始：

步骤记录：

1. **初始化**：U = {a}
    - 候选边：a-b=4, a-c=3, 其他∞
2. **第 1 轮**：选最小边 a-c=3
    - 加入顶点 c，U = {a,c}
    - 更新候选边：
        - c-b=5 > 原来的 4，保持 a-b=4
        - c-d=5（新）
        - c-h=5（新）
3. **第 2 轮**：选最小边 a-b=4
    - 加入顶点 b，U = {a,c,b}
    - 更新候选边：
        - b-d=5 = 原来的 5，不更新
        - b-e=9（新）
4. **第 3 轮**：选最小边 c-d=5（**与 b-d=5 相同，任选**）
    - 加入顶点 d，U = {a,c,b,d}
    - 更新候选边：
        - d-e=7 < 原来的 9，更新为 d-e=7
        - d-f=6（新）
        - d-g=5（新）
        - d-h=4 < 原来的 5，更新为 d-h=4
5. **第 4 轮**：选最小边 d-h=4
    - 加入顶点 h，U = {a,c,b,d,h}
    - 更新候选边：
        - h-g=6 > 原来的 5，保持 d-g=5
6. **第 5 轮**：选最小边 d-g=5
    - 加入顶点 g，U = {a,c,b,d,h,g}

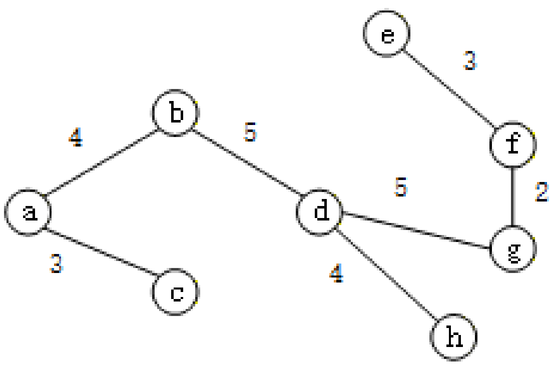
- 更新候选边：
    - g-f=2 < 原来的 6，更新为 g-f=2

7. **第 6 轮**：选最小边 g-f=2
    - 加入顶点 f，U = {a, c, b, d, h, g, f}
    - 更新候选边：
        - f-e=3 < 原来的 7，更新为 f-e=3
8. **第 7 轮**：选最小边 f-e=3
    - 加入顶点 e，U = {a, c, b, d, h, g, f, e}
    - 完成

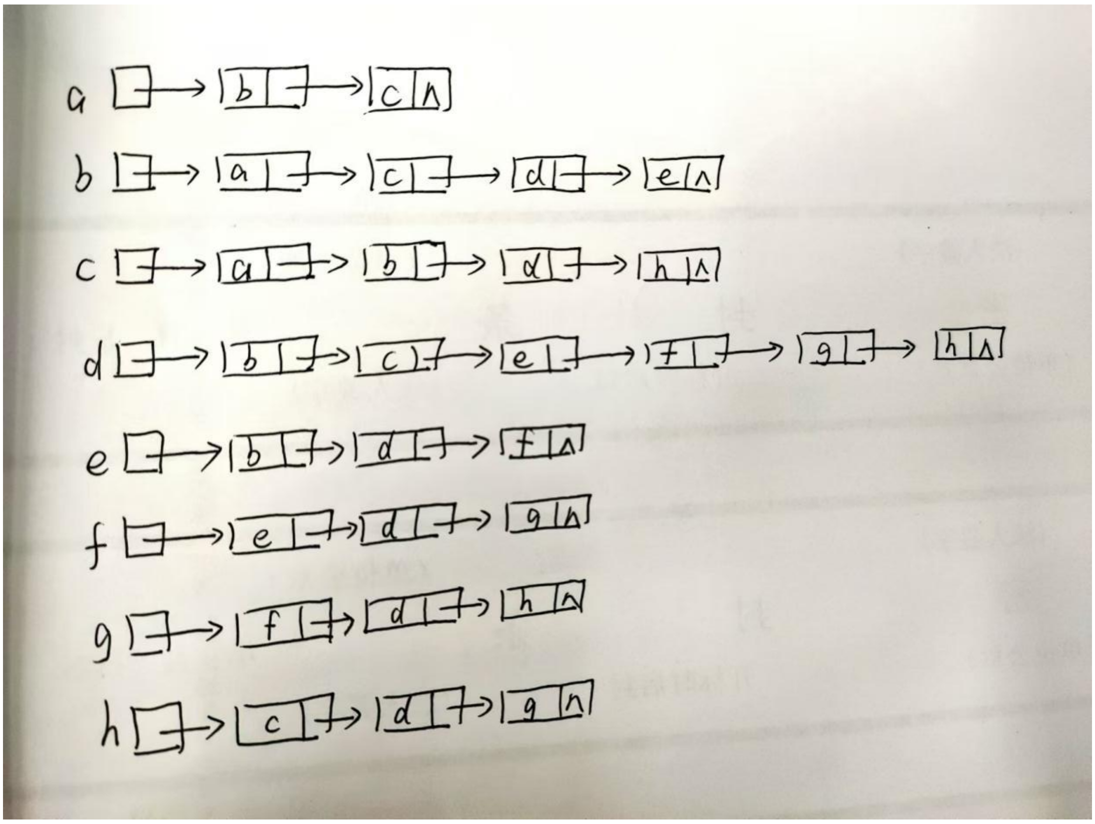
Prim 算法结果：

边	权值
a-c	3
a-b	4
c-d	5（这里 b-d 也可以）
d-h	4
d-g	5
g-f	2
f-e	3

总权值：26



（2）



二、Kruskal 算法求最小生成树

步骤记录：

1. **所有边按权值排序：**
   - g-f=2
   - a-c=3
   - f-e=3
   - a-b=4， d-h=4
   - b-c=5， c-d=5， b-d=5， c-h=5， d-g=5
   - d-f=6， g-h=6
   - d-e=7
   - b-e=9
2. **依次选择边**（不形成环）：
   - 选择 g-f=2（加入 g,f）
   - 选择 a-c=3（加入 a,c）
   - 选择 f-e=3（加入 e，现在{g,f,e}连通）

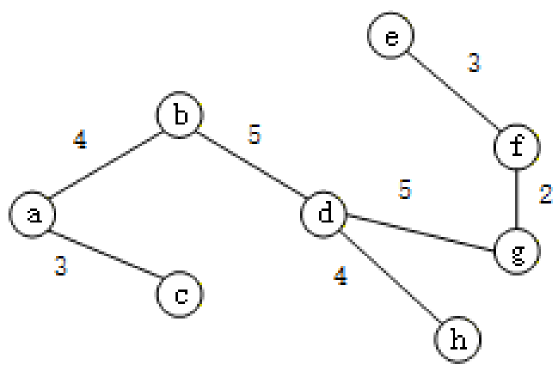
- 选择 a-b=4（加入 b，现在{a, c, b}连通）
- 选择 d-h=4（加入 d, h，新连通分量{d, h}）
- 选择 c-d=5（连接{a, c, b}和{d, h}，**注意 b-d=5 也可，任选**）
- 选择 d-g=5（连接{a, b, c, d, h}和{g, f, e}）

3. **已选边检查：**
    - 已选 7 条边，n-1=7，完成

Kruskal 算法结果：

```text
边      权值
g-f      2
a-c      3
f-e      3
a-b      4
d-h      4
c-d      5（这里 b-d 也可以）
d-g      5

总权值：26
```



第 9 章

【习题】9.2 折半查找过程

有序线性表：(a, b, c, d, e, f, g) 假设下标从 1 开始，关键字与下标对应：1→a, 2→b, 3→c, 4→d, 5→e, 6→f, 7→g。 **查找关键字 e**（下标 5）：

1. low=1, high=7, mid=⌊(1+7)/2⌋=4，关键字为 d。

   ∵ e ＞ d，∴ low=mid+1=5。

2. low=5, high=7, mid=⌊(5+7)/2⌋=6，关键字为 f。

   ∵ e ＜ f，∴ high=mid-1=5。

3. low=5, high=5, mid=5，关键字为 e，找到。

查找关键字 f（下标 6）：

1. low=1, high=7, mid=⌊(1+7)/2⌋=4，关键字 d。

   ∵ f ＞ d，∴ low=mid+1=5。

2. low=5, high=7, mid=⌊(5+7)/2⌋=6，关键字 f，找到。

查找关键字 g（下标 7）：

1. low=1, high=7, mid=⌊(1+7)/2⌋=4，关键字 d。

   ∵ g ＞ d，∴ low=mid+1=5。

2. low=5, high=7, mid=⌊(5+7)/2⌋=6，关键字 f。

   ∵ g ＞ f，∴ low=mid+1=7。

3. low=7, high=7, mid=7，关键字 g，找到。

【习题】9.3 长度为 10 的有序表折半查找判定树与平均查找长度

假设有序表关键字为（1,2,3,...,10）。 **判定树**（平衡二叉树结构）：

```
            5
          /    \
        2        8
      /   \     /  \
    1      4   6     9
          /     \     \
        3         7    10
```

层数与结点分布：

- 第 1 层：结点 5（比较 1 次）
- 第 2 层：结点 2、8（比较 2 次）
- 第 3 层：结点 1、4、6、9（比较 3 次）
- 第 4 层：结点 3、7、10（比较 4 次）

等概率查找成功的平均查找长度：
ASL=（1×1+2×2+4×3+3×4）/10 =(1+4+12+12)/10=29/10=2.9

【习题】9.4 混合查找方法（表长 n=50）

查找规则：若表长 ≤ 10，则顺序查找；否则折半查找。**判定树构建**（关键字假设为 1,2,...,50）：

- 整个表长 50 > 10，采用折半查找，取中间值 mid=25 为根结点。
- 递归左右子表，直到子表长度 ≤ 10 时停止划分，该子表作为叶子结点（内部采用顺序查找）。

具体划分：

1. 根结点：25（区间 [1,50]）
   - 左子树：[1,24]（长度 24>10），mid=12
     - 左子树：[1,11]（长度 11>10），mid=6
       - [1,5]（长度 5≤10）→ 叶子
       - [7,11]（长度 5≤10）→ 叶子
     - 右子树：[13,24]（长度 12>10），mid=18
       - [13,17]（长度 5≤10）→ 叶子
       - [19,24]（长度 6≤10）→ 叶子
   - 右子树：[26,50]（长度 25>10），mid=38
     - 左子树：[26,37]（长度 12>10），mid=31
       - [26,30]（长度 5≤10）→ 叶子

        - [32,37]（长度 6≤10）→ 叶子
    - 右子树：[39,50]（长度 12>10），mid=44
        - [39,43]（长度 5≤10）→ 叶子
        - [45,50]（长度 6≤10）→ 叶子

判定树结构示意图（简化）:

```
                    25
              /                \
            12                    38
       /         \           /          \
     6            18         31           44
   /   \        /   \      /   \        /   \
[1,5] [7,11] [13,17][19,24] [26,30][32,37][39,43][45,50]
```

（注：[ ] 表示叶子子表，内部顺序查找；数字为内部结点，直接比较找到。） **平均查找长度计算：**

- **内部结点关键字**（7 个）：查找长度 = 结点深度（比较次数）
    - 25：1
    - 12，38：2
    - 6，18，31，44：3
    - 总和：1 + 2×2 + 4×3 = 1 + 4 + 12 = 17
- **叶子子表关键字**（43 个）：查找长度 = 路径比较次数 d + 子表内顺序查找比较次数
    - 路径比较次数 d：从根到叶子子表经过的内部结点数（均为 3）
    - 子表内平均查找长度：(m+1)/2（m 为子表大小）

叶子子表	m d	子表内总比较次数 m(m+1)/2	该子表总查找长度 m·d + m(m+1)/2
[1,5]	5 3	15	5×3 + 15 = 30
[7,11]	5 3	15	30
[13,17]	5 3	15	30
[19,24]	6 3	21	6×3 + 21 = 39
[26,30]	5 3	15	30
[32,37]	6 3	21	39
[39,43]	5 3	15	30
[45,50]	6 3	21	39

叶子子表总查找长度 = 30×4 + 39×4 = 120 + 156 = 276 **全表总查找长度** = 17 + 276 = 293 **平均查找长度** ASL = 293 / 50 = **5.86**

【习题】9.8 解答

已知有序表及其权值如下:

关键字 A B C D E F G H I J K L

权值 8 2 3 4 9 3 2 6 7 1 1 4

PH 值定义：查找树中所有结点的"深度 × 权值"之和，即带权路径长度，其值越小表示在该权值分布下查找效率越高。

(1) 次优查找树的构造与 PH 值

构造过程（采用次优查找树算法，选择根结点时使左右子树的累计权值差最小）

1. **确定整棵树根结点** 计算每个位置左右累计权值差的绝对值，选最小者：

i	关键字	左边累计权值	右边累计权值	差值绝对值
6	F	26	21	**5**（最小）

2. 故根结点为 **F**。
3. **递归构造左右子树**
    - **F 的左子树**（关键字 A-E，权值和 26） 计算得位置 3（C）的差值最小（|10-13|=3），故左子树根为 **C**。
        - C 的左子树（A,B）选 A 为根，A 的右孩子为 B。
        - C 的右子树（D,E）选 E 为根，E 的左孩子为 D。
    - **F 的右子树**（关键字 G-L，权值和 21） 计算得位置 9（I）的差值最小（|8-6|=2），故右子树根为 **I**。
        - I 的左子树（G,H）选 H 为根，H 的左孩子为 G。
        - I 的右子树（J,K,L）选 L 为根，L 的左子树（J,K）选 J 为根，J 的右孩子为 K（这里 J、K 都可以作为根，权值均为 1）。

最终次优查找树结构（根节点深度为 1）（小括号内为权值，中括号内为深度）：

```
                  F(3)[1]
              /            \
          C(3)[2]          I(7)[2]
        /      \          /         \
  A(8)[3]   E(9)[3] H(6)[3]   L(4)[3]
      \         /         /          /
     B(2)[4] D(4)[4] G(2)[4] J(1)[4]
                                  \
                                   K(1)[5]
```

PH 值计算：

- 深度 1：F → 1×3 =3

- 深度 2：C → 2×3 =6；I → 2×7 = 14
- 深度 3：A → 3×8 = 24；E → 3×9 = 27；H → 3×6 = 18；L → 3×4 = 12
- 深度 4：B → 4×2 = 8；D → 4×4 = 16；G → 4×2 = 8；J → 4×1 =4
- 深度 5：K → 5×1 =5

PH=3+6+14+24+27+18+12+8+16+8+4+5=145

(2) 折半查找判定树与 PH 值

判定树构造（假设向下取整，取 mid = ⌊(low+high)/2⌋）：

1. 根结点：mid = ⌊(1+12)/2⌋ = 6 → **F**
2. 左子树（1-5）：mid = ⌊(1+5)/2⌋ = 3 → **C**
    - C 的左子树（1-2）：mid = ⌊(1+2)/2⌋ = 1 → **A**，A 的右孩子为 **B**（位置 2）
    - C 的右子树（4-5）：mid = ⌊(4+5)/2⌋ = 4 → **D**，D 的右孩子为 **E**（位置 5）
3. 右子树（7-12）：mid = ⌊(7+12)/2⌋ = 9 → **I**
    - I 的左子树（7-8）：mid = ⌊(7+8)/2⌋ = 7 → **G**，G 的右孩子为 **H**（位置 8）
    - I 的右子树（10-12）：mid = ⌊(10+12)/2⌋ = 11 → **K**
        - K 的左子树（10-10）：mid = 10 → **J**
        - K 的右子树（12-12）：mid = 12 → **L**

树结构：

```
                F(3)[1]
              /          \
        C(3)[2]          I(7)[2]
       /      \         /        \
   A(8)[3]   D(4)[3] G(2)[3]   K(1)[3]
     \          \        \       /     \
    B(2)[4]   E(9)[4] H(6)[4] J(1)[4] L(4)[4]
```

PH 值计算：

- 深度 1：F → 1×3 = 3
- 深度 2：C → 2×3 = 6；I → 2×7 = 14
- 深度 3：A → 3×8 = 24；D → 3×4 = 12；G → 3×2 = 6；K → 3×1 = 3
- 深度 4：B → 4×2 = 8；E → 4×9 = 36；H → 4×6 = 24；J → 4×1 = 4；L → 4×4 = 16

PH=3+6+14+24+12+6+3+8+36+24+4+16=156

结论

- 次优查找树的 PH 值（145）小于折半查找树的 PH 值（156），说明在给定权值分布下，**次优查找树的平均查找效率更高**，因为它通过调整结点位置，将权值大的结点（如 A、E、I）放在了较浅的深度。

继续递归部分展开讲解

当前状态：

text

根：F(6) [区间 1-12]

左子树：C(3) [区间 1-5]

右子树：I(9) [区间 7-12]

1. 处理 C 的左子树（区间[1,2]：A,B）

计算区间[1,2]:

- k=1(A)：Δ = |(SW[0]-SW[0]) - (SW[2]-SW[1])| = |0 - (10-8)| = |0-2| = 2
- k=2(B)：Δ = |(SW[1]-SW[0]) - (SW[2]-SW[2])| = |8 - 0| = 8

最小 Δ=2，所以根结点是 **A**

A 的左子树：区间[空]，返回 NULL
A 的右子树：区间[2,2]，只有一个结点 B

结果:

text

A(1)

 \

 B(2)

2. 处理 C 的右子树（区间[4,5]：D,E）

计算区间[4,5]:

- k=4(D)：Δ = |(SW[3]-SW[3]) - (SW[5]-SW[4])| = |0 - (26-17)| = |0-9| = 9
- k=5(E)：Δ = |(SW[4]-SW[3]) - (SW[5]-SW[5])| = |(17-13) - 0| = |4-0| = 4

最小 Δ=4，所以根结点是 **E**

E 的左子树：区间[4,4]，只有一个结点 D
E 的右子树：区间[空]，返回 NULL

结果：

text

E(5)

 /

D(4)

3. 处理 I 的左子树（区间[7,8]：G,H）

计算区间[7,8]：

- k=7(G)：Δ = |(SW[6]-SW[6]) - (SW[8]-SW[7])| = |0 - (37-31)| = |0-6| = 6
- k=8(H)：Δ = |(SW[7]-SW[6]) - (SW[8]-SW[8])| = |(31-29) - 0| = |2-0| = 2

最小 Δ=2，所以根结点是 **H**

H 的左子树：区间[7,7]，只有一个结点 G
H 的右子树：区间[空]，返回 NULL

结果：

text

H(8)

 /

G(7)

4. 处理 I 的右子树（区间[10,12]：J,K,L）

计算区间[10,12]：

- k=10(J)：Δ = |(SW[9]-SW[9]) - (SW[12]-SW[10])| = |0 - (50-45)| = |0-5| = 5
- k=11(K)：Δ = |(SW[10]-SW[9]) - (SW[12]-SW[11])| = |(45-44) - (50-46)| = |1-4| = 3

- k=12(L)：Δ = |(SW[11]-SW[9]) - (SW[12]-SW[12])| = |(46-44) - 0| = |2-0| = 2

最小 Δ=2，所以根结点是 **L**

L 的左子树：区间[10,11]：J,K
L 的右子树：区间[空]，返回 NULL

5. 处理 L 的左子树（区间[10,11]：J,K）

计算区间[10,11]：

- k=10(J)：Δ = |(SW[9]-SW[9]) - (SW[11]-SW[10])| = |0 - (46-45)| = |0-1| = 1
- k=11(K)：Δ = |(SW[10]-SW[9]) - (SW[11]-SW[11])| = |(45-44) - 0| = |1-0| = 1

两者 Δ 相同=1，可以任选，这里选 **J**（按题目结果）

J 的左子树：区间[空]，返回 NULL
J 的右子树：区间[11,11]，只有一个结点 K

结果：

text

J(10)

 \

 K(11)

完整的递归构造过程总结：

text

第 1 层：整个区间[1,12] → 根：F(6)

第 2 层：

 - F 左子树区间[1,5] → 根：C(3)

 - F 右子树区间[7,12] → 根：I(9)

第 3 层：

- C 左子树区间[1,2] → 根：A(1)

- C 右子树区间[4,5] → 根：E(5)

- I 左子树区间[7,8] → 根：H(8)

- I 右子树区间[10,12] → 根：L(12)

第 4 层：

- A 右子树区间[2,2] → 只有 B(2)

- E 左子树区间[4,4] → 只有 D(4)

- H 左子树区间[7,7] → 只有 G(7)

- L 左子树区间[10,11] → 根：J(10)

第 5 层：

- J 右子树区间[11,11] → 只有 K(11)

叶结点确定的关键点：

1. 区间长度为 1 时：

当一个区间只有一个结点时，它自然成为叶结点（或只有一个孩子的结点）

- 如区间[2,2]：只有 B，作为 A 的右孩子
- 如区间[4,4]：只有 D，作为 E 的左孩子

2. 区间长度为 0 时：

返回 NULL，表示空子树

- 如 A 的左子树区间[空]
- 如 E 的右子树区间[空]

3. 递归终止条件：

c

```
if (区间开始 > 区间结束)
    return NULL;   // 空子树
if (区间开始 == 区间结束)
    return 创建结点(关键字[开始]);   // 叶结点
```

4. 构建叶结点时的左右子树：

对于叶结点（区间长度为 1）：

- **左子树**：区间[i,i-1] → 空
- **右子树**：区间[i+1,i] → 空
- 所以**真正的叶结点没有孩子**

但在题目中：

- B 是 A 的右孩子，但 B 自己可能还有子树吗？没有，因为区间[2,2]处理完毕
- 实际上，在最终的树中，B、D、G、K 都是叶结点（没有孩子）

【习题】9.11 解答

推导过程

设 Nh 是高度为 h 的平衡二叉树（AVL 树）的最少结点数，则有递推关系：

$N_0=0, N_1=1, N_h=1+N_{h-1}+N_{h-2} (h\geqslant 2)$

计算：

- N2=1+1+0=2
- N3=1+2+1=4
- N4=1+4+2=7
- N5=1+7+4=12
- N6=1+12+7=20

给定结点数 n=12，满足 N5=12，而 N6=20>12。因此，含 12 个结点的平衡二叉树的最大深度（高度）为 **5**（深度定义为从根到叶子的最长路径上的结点数，**根结点深度为 1**）。

树形示例

以下是一棵深度为 5 的平衡二叉树，结点用大写字母表示（可对应关键字，树满足平衡二

叉树性质，每个结点的左右子树高度差不超过 1)：

```
            A
           / \
         B       C
        / \       / \
      D    E    L    F
     / \     \          \
   G      H      I       J
  /
K
```

【习题】9.21 解答

已知关键字序列：Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec。 哈希函数：$H(\mathrm{x}) = \lfloor i/2 \rfloor$，其中 i 为关键字首字母在英文字母表中的序号（A=1，B=2，···，Z=26）。 地址空间：0 ~ 16，表长为 17。

1. 计算各关键字哈希地址

关键字	首字母	序号 I	$H(\mathrm{x}) = \lfloor i/2 \rfloor$
Jan	J	10	5
Feb	F	6	3
Mar	M	13	6
Apr	A	1	0
May	M	13	6
June	J	10	5
July	J	10	5
Aug	A	1	0
Sep	S	19	9
Oct	O	15	7
Nov	N	14	7
Dec	D	4	2

2. 用线性探测开放定址法处理冲突

按给定顺序插入关键字，冲突时探查下一位置（模 17）。插入过程如下：

插入顺序	关键字	H(x)	探查地址（最终位置）
1	Jan	5	5（成功）
2	Feb	3	3（成功）

3	Mar	6	6（成功）
4	Apr	0	0（成功）
5	May	6	6 冲突 →7（成功）
6	June	5	5 冲突 →6→7→8（成功）
7	July	5	5→6→7→8→9（成功）
8	Aug	0	0 冲突 →1（成功）
9	Sep	9	9 冲突 →10（成功）
10	Oct	7	7 冲突 →8→9→10→11（成功）
11	Nov	7	7→8→9→10→11→12（成功）
12	Dec	2	2（成功）

最终哈希表（0 ~ 16 单元）存储情况如下：

地址	关键字
0	Apr
1	Aug
2	Dec
3	Feb
4	空
5	Jan
6	Mar
7	May
8	June
9	July
10	Sep
11	Oct
12	Nov
13	空
14	空
15	空
16	空

3. 用链地址法处理冲突

每个地址建立一个单链表，冲突关键字插入链表尾部（也可头部，此处按插入顺序链接）。链表结构如下：

- 地址 0: Apr → Aug
- 地址 1: （空）
- 地址 2: Dec
- 地址 3: Feb
- 地址 4: （空）
- 地址 5: Jan → June → July
- 地址 6: Mar → May

- 地址 7: Oct → Nov
- 地址 8: （空）
- 地址 9: Sep
- 地址 10: （空）
- 地址 11: （空）
- 地址 12: （空）
- 地址 13: （空）
- 地址 14: （空）
- 地址 15: （空）
- 地址 16: （空）

根据给定的关键字序列和哈希函数 $H(x)=\lfloor i/2 \rfloor$（i 为首字母序号），在地址空间 0—16 内，分别采用线性探测开放定址法和链地址法处理冲突，构造哈希表并计算平均查找长度（ASL）如下。

(一)线性探测开放定址法

哈希表构造结果（表长 17）：

- 地址与关键字对应：

0: Apr, 1: Aug, 2: Dec, 3: Feb, 4: 空, 5: Jan, 6: Mar, 7: May, 8: June, 9: July, 10: Sep, 11: Oct, 12: Nov, 13~16: 空

查找成功的平均查找长度 各关键字查找时所需比较次数（探查次数）：
Jan(1), Feb(1), Mar(1), Apr(1), May(2), June(4), July(5), Aug(2), Sep(2), Oct(5), Nov(6), Dec(1) 总比较次数 = 31，关键字数 = 12
ASL 成功=31/12≈2.583

查找不成功的平均查找长度 对每个地址（0~16），计算直至遇到空位的比较次数： 0:5, 1:4, 2:3, 3:2, 4:1, 5:9, 6:8, 7:7, 8:6, 9:5, 10:4, 11:3, 12:2, 13:1, 14:1, 15:1, 16:1 总比较次数 = 63，地址数 = 17
ASL 不成功=63/17≈3.706

(二)链地址法

哈希表构造结果（每个地址对应一个单链表）：

- 地址 0: Apr → Aug
- 地址 2: Dec
- 地址 3: Feb
- 地址 5: Jan → June → July
- 地址 6: Mar → May
- 地址 7: Oct → Nov

- 地址 9: Sep
- 其余地址为空链表

查找成功的平均查找长度 各关键字查找时所需比较次数（在链表中的位置）：Apr(1), Aug(2), Dec(1), Feb(1), Jan(1), June(2), July(3), Mar(1), May(2), Oct(1), Nov(2), Sep(1) 总比较次数 = 18，关键字数 = 12

ASL 成功=18/12=1.5

查找不成功的平均查找长度 对每个地址，比较次数等于链表长度（探查至链表末尾）： 0:2, 1:0, 2:1, 3:1, 4:0, 5:3, 6:2, 7:2, 8:0, 9:1, 10~16:0 总比较次数 = 12，地址数 = 17

ASL 不成功=12/17≈0.706

（注：若定义包括最后一次与空指针的比较，则总比较次数为 29，ASL 不成功 =29/17≈1.706，但通常采用链表长度计算。）

结果汇总

处理方法	查找成功的 ASL	查找不成功的 ASL
线性探测开放定址	2.583	3.706
链地址法	1.5	0.706

以上计算基于等概率假设，即查找每个关键字的概率相同，且查找每个地址的概率相同。

【习题】9.23 两种给定策略的可行性分析

策略一：将待删表项的关键字置为 -1（与空表项相同）

不可行。 原因：开放定址法中，查找操作遇到空表项（值为 -1）时，会认为目标关键字不存在而停止探查。如果删除时直接将关键字置为 -1，会导致原本位于该位置之后的、因冲突而填入其他地址的同义词关键字“断链”，**后续查找时会在该空位提前终止**，**从而错误地判定为“不存在”**。

举例说明 设哈希函数 H(key)=key % 7，表长 7，初始空表值均为 -1。 依次插入关键字：14（地址 0）、21（地址 0 冲突，线性探测到地址 1）、8（地址 1 冲突，探测到地址 2）。 表状态：[14, 21, 8, −1, −1, −1, −1] 查找 8：H(8)=1 → 地址 1 为 21（冲突） → 地址 2 为 8（找到）。 若删除 21（地址 1），按策略一置为 -1，表变为：[14, −1, 8, −1, −1, −1, −1] 再次查找 8：H(8)=1 → 地址 1 为 -1（空），查找提前终止，错误地认为 8 不存在。

策略二：顺序递补（用探测序列上下一个关键字覆盖待删位置，并将

该下一位置置为 -1）

不可行。 原因：虽然避免了“断链”，但**破坏了关键字的探测序列关系**。原被移动的关键字本应处于其**原始哈希地址**对应的探测序列上，移动后其当前位置可能不再处于其原始哈希地址的探测序列中，导致后续查找时无法正确找到它。

举例说明 沿用上例初始表：[14，21，8，-1，-1，-1，-1] 删除 21（地址 1）：

- “探测序列上下一个关键字”是地址 2 的 8。
- 将地址 1 覆盖为 8，地址 2 置为 -1，表变为：[14，8，-1，-1，-1，-1，-1]
  查找 8：H(8)=1 → 地址 1 为 8（找到），看似没问题。 但查找原本在地址 0 的 14 的同义词（如另一个关键字 28，H(28)=0）时，若 28 之前因冲突被安置在地址 1 之后（例如地址 3），现在地址 1 被 8 占用，破坏了 28 的探测路径。更直观地，考虑查找刚被移动的 8 的其他同义词（如 15，H(15)=1）： 插入 15：H(15)=1 → 地址 1 已有 8（冲突）→ 地址 2 为 -1，插入。表：[14，8，15，-1，-1，-1，-1] 现在删除 8（地址 1）：
- 下一关键字是地址 2 的 15，将地址 1 覆盖为 15，地址 2 置为 -1，表：[14，15，-1，-1，-1，-1，-1] 查找 15：H(15)=1 → 地址 1 为 15（找到），但**此时 15 并不在它原始哈希地址（H(15)=1）的探测序列上**，因为它本应被线性探测到地址 2，现在却因递补到了地址 1。这破坏了哈希表的逻辑一致性，当表继续填充时可能导致更严重的混乱。

一种可行的方法：懒惰删除（标记删除法）

方法：

- 为每个表项增加一个状态标记位（或利用特殊值表示不同状态），通常有三种状态：
  1. **空（Empty）**：初始状态，从未插入过关键字。
  2. **占用（Occupied）**：当前存放有效关键字。
  3. **删除（Deleted）**：该位置曾被占用，但关键字已被删除。
- 删除操作：仅将表项状态从 **占用** 改为 **删除**，关键字值可以保留或清空（但不能置为 -1，以免与“空”混淆）。
- 查找操作：探查时遇到 **占用** 且关键字匹配则成功；遇到 **空** 则失败（说明目标不存在）；遇到 **删除** 则继续探查（因为目标关键字可能还在后面）。
- 插入操作：探查时遇到 **空** 或 **删除** 都可以插入新关键字（通常优先插入第一个可用的 **删除** 或 **空** 位置）。

对查找和插入算法的影响

1. **查找算法**：
    - 需要增加对“删除”状态的判断逻辑。
    - 查找失败的条件变为：遇到 **空** 状态（而不是遇到 -1）。
    - 遇到 **删除** 状态时，继续沿探测序列向后查找。
    - **优点**：不会因删除导致同义词链断裂。
    - **缺点**：查找可能需要更多次探查，因为要跳过“删除”标记。
2. **插入算法**：
    - 探查时，第一个遇到的 **空** 或 **删除** 位置都可作为插入点。
    - 通常优先使用 **删除** 位置，可以提高空间利用率，避免“删除”位置堆积。
    - **优点**：新关键字可以重用被删除的位置，保持了哈希表的装载因子有效。
    - **缺点**：需要额外空间存储状态标记，插入逻辑稍复杂。

举例说明（用状态标记）

设状态：E（空），O（占用），D（删除）。 初始表：[E, E, E, E, E, E, E] 插入 14（H=0）：[O(14)，E, E, E, E, E, E] 插入 21（H=0 冲突，探测到地址 1）：[O(14)，O(21)，E, E, E, E, E] 插入 8（H=1 冲突，探测到地址 2）：[O(14)，O(21)，O(8)，E, E, E, E] 删除 21（地址 1）：状态改为 D，[O(14)，D(21)，O(8)，E, E, E, E] 查找 8（H=1）：

- 地址 1：D，继续；
- 地址 2：O(8)，找到。 插入 15（H=1）：
- 地址 1：D，可插入，[O(14), O(15), O(8), E, E, E, E]（重用删除位）。