

数学基础

等差数列求和公式

$$S_n = \frac{n(a_1 + a_n)}{2} \quad (1)$$

或者

$$S_n = na_1 + \frac{n(n - 1)d}{2} \quad (2)$$

等比数列

$$S_n = \frac{a_1(1 - q^n)}{1 - q} = \frac{a_1(q^n - 1)}{q - 1} \quad (3)$$

树

常见性质和结论

- 树的节点数等于所有节点的度数之和加一
- 度为m的树中第i层上最多有 m^{i-1} 个节点
 - 第一层有1个节点,第二层至多m个节点,第三层至多有 m^2 个节点,以此类推
- 高度为h的m叉树至多有 $S = \sum_{i=1}^h m^{i-1} = \frac{m^h - 1}{m - 1}$ 个节点
- 非空二叉树上的叶节点数等于度为2的节点数加一,即 $n_0 = n_2 + 1$
 - 总节点数 $n=B+1$,B为分支总数, $B=n_1+2*n_2$
 - $n_0+n_1+n_2=n_1+2*n_2$, $n_0 = n_2 + 1$
- 非空二叉树第k层最多有 2^{k-1} 个节点
- 高度为h的非空二叉树最多有 $2^h - 1$ 个节点
- 在含有n个节点的二叉链表中,空链域(空指针)的个数是n+1。
 - n个节点一共有2n个指针域
 - 除了根节点外其他的n-1个节点都有一个指针域非空
 - 故空指针域的个数为 $2n-(n-1)=n+1$
- 二叉树不是树的特殊情况,两者是完全不同的数据结构

完全二叉树的性质

- 完全二叉树的叶子节点只可能在层次最大的两层出现
- 若有度为1的节点,那么这个节点只可能有一个,并且该节点只有左孩子没有右孩子
- 节点数为n的完全二叉树的高度为 $h = \lfloor \log_2 n \rfloor + 1$ 或者 $h = \lceil \log_2(n + 1) \rceil$
- 若总节点数为偶数,则n1的个数为1,否则为0

由遍历序列构造二叉树

前序遍历+中序遍历

前序遍历序列：根结点，左子树的前序遍历序列，右子树的前序遍历序列

中序遍历序列：左子树的中序遍历序列 根结点 右子树的中序遍历序列

由前序遍历的根结点可推出中序遍历的根结点，在中序遍历中，根结点左边的子序列即为左子树的中序序列，同样可得到右边的。

这样递归分解下去即可得到二叉树

例子1

前序遍历序列：ADBCE

中序遍历序列：BDCAE

由前序遍历可得到这个树的根结点即为A，左子树的中序遍历序列即为BDC，长度为3。右子树即为E，长度为1。

在前序遍历序列中根节点后三个序列即为左子树的前序遍历序列，为DBC，可得这个左子树的根节点为D，由此可得到树的结构

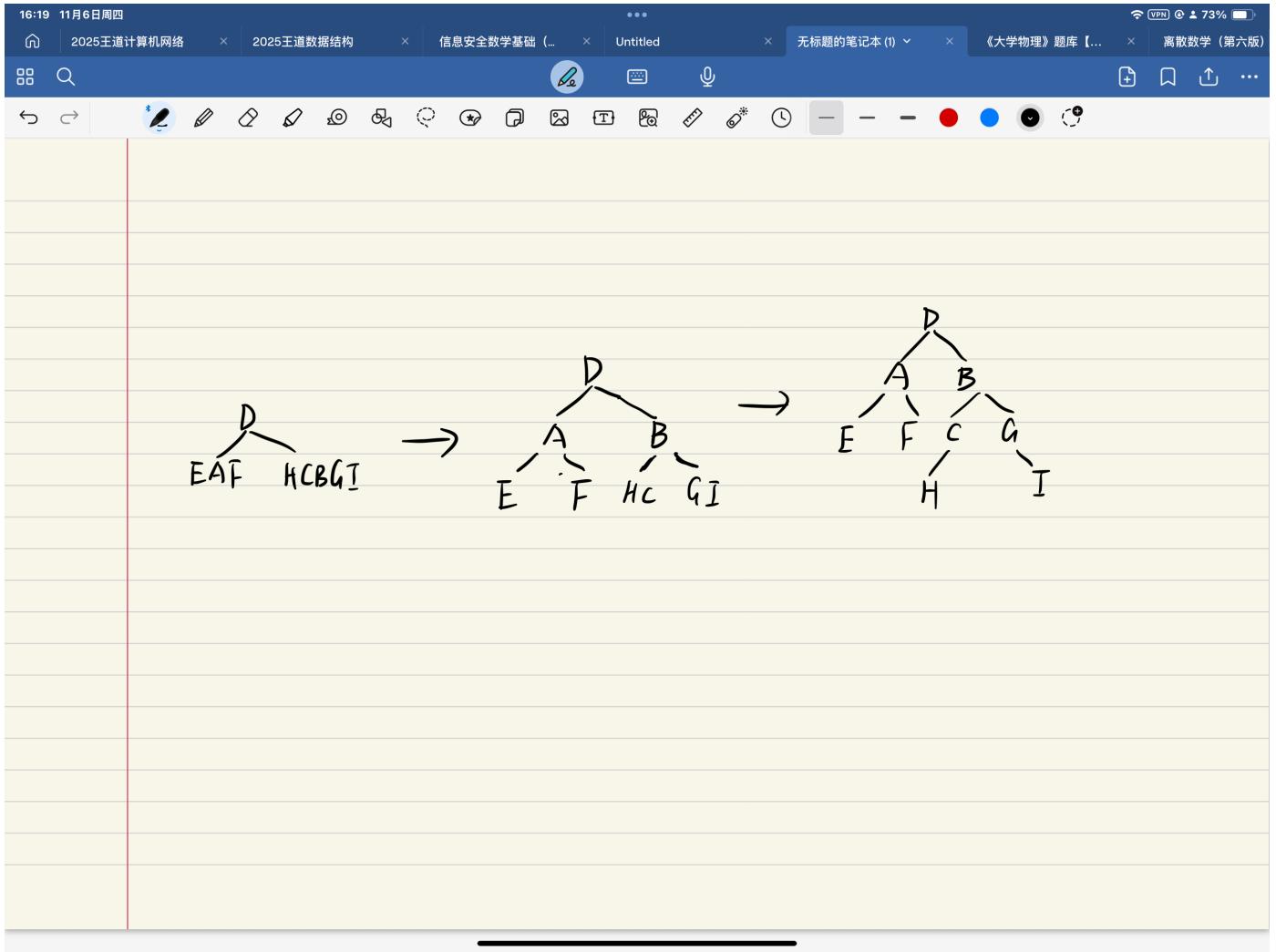


例子2

前序遍历序列：DAEFBCHGI

中序遍历序列：EAFDHCBGI

解答如图：



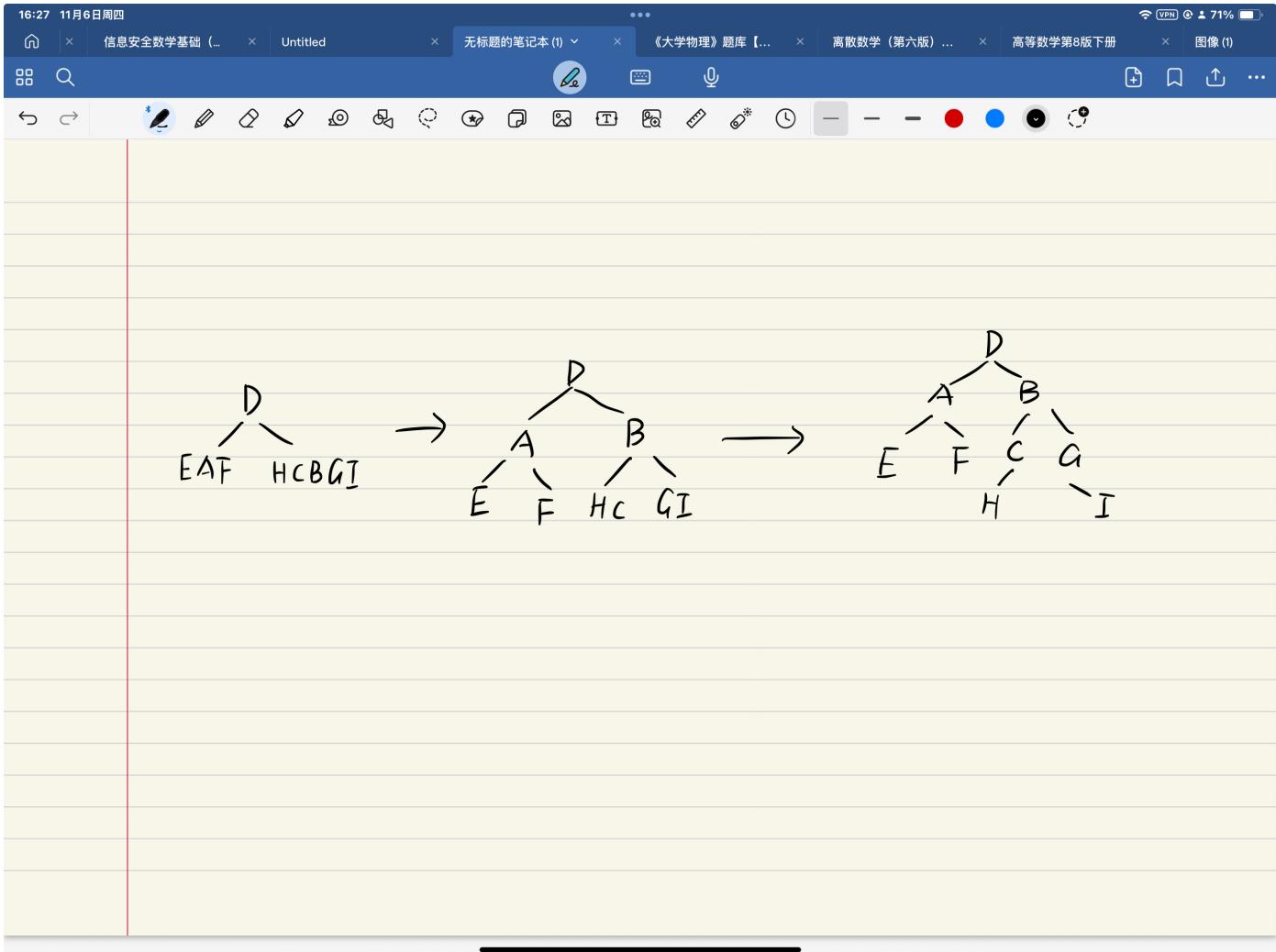
中序+后序

直接来例子

后序遍历序列：EFAHCIGBD

中序遍历序列：EAFDHCGBI

不同的是在后序遍历中，最后面的节点才是根节点，其他的一样



二叉树的线索化

线索二叉树将普通二叉树中的叶子节点的空指针指向其前驱/后继。具体前驱/后继是谁，则要看这个线索二叉树是根据什么样的遍历方式进行遍历的。引入线索二叉树就是为了加快查找节点前驱和后继发明的。

规定：若无左子树，令lchild指向其前驱节点；若无右子树，令rchild指向其后继节点。还需要增加两个标志域，标识指针区域指向左右孩子或前驱/后继

- n个节点的线索二叉树上含有的线索数(空链域)的数量为n+1

其中，标志域的含义如下

$$\begin{aligned} \text{ltag} &= \begin{cases} 0, & \text{lchild域指示结点的左孩子} \\ 1, & \text{lchild域指示结点的前驱} \end{cases} \\ \text{rtag} &= \begin{cases} 0, & \text{rchild域指示结点的右孩子} \\ 1, & \text{rchild域指示结点的后继} \end{cases} \end{aligned}$$

中序线索化一个二叉树的代码如下

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

typedef struct ThreadNode {
    int data;
```

```

    struct ThreadNode *lchild, *rchild;
    int ltag, rtag;
} ThreadNode, *ThreadTree;

ThreadNode *pre = NULL; // 维护前驱节点

// 访问并建立线索
void visit(ThreadNode *q) {
    if (q->lchild == NULL) { // 左指针为空，将其线索化
        q->lchild = pre;
        q->ltag = 1; // 左指针线索化
    }
    if (pre != NULL && pre->rchild == NULL) { // 建立前驱节点的后继联系
        pre->rchild = q;
        pre->rtag = 1;
    }
    pre = q; // 更新前驱节点
}

// 中序遍历并线索化二叉树
void InThread(ThreadTree T) {
    if (T != NULL) {
        InThread(T->lchild); // 遍历左子树
        visit(T); // 访问当前节点
        InThread(T->rchild); // 遍历右子树
    }
}

// 创建线索二叉树
void CreateInthread(ThreadTree T) {
    pre = NULL; // 初始化前驱节点为 NULL
    if (T != NULL) {
        InThread(T); // 中序遍历并线索化
        if (pre != NULL && pre->rchild == NULL) {
            pre->rchild = NULL;
            pre->rtag = 1; // 最后一个节点的右指针线索化
        }
    }
}

```

树的存储结构

孩子兄弟表示法

用孩子兄弟表示法表示一棵树的结构：一个节点的右指针不再是右孩子，变成了右兄弟。左指针还是左孩子。（左连孩子右连兄弟）在一棵子树中的同一层称为兄弟

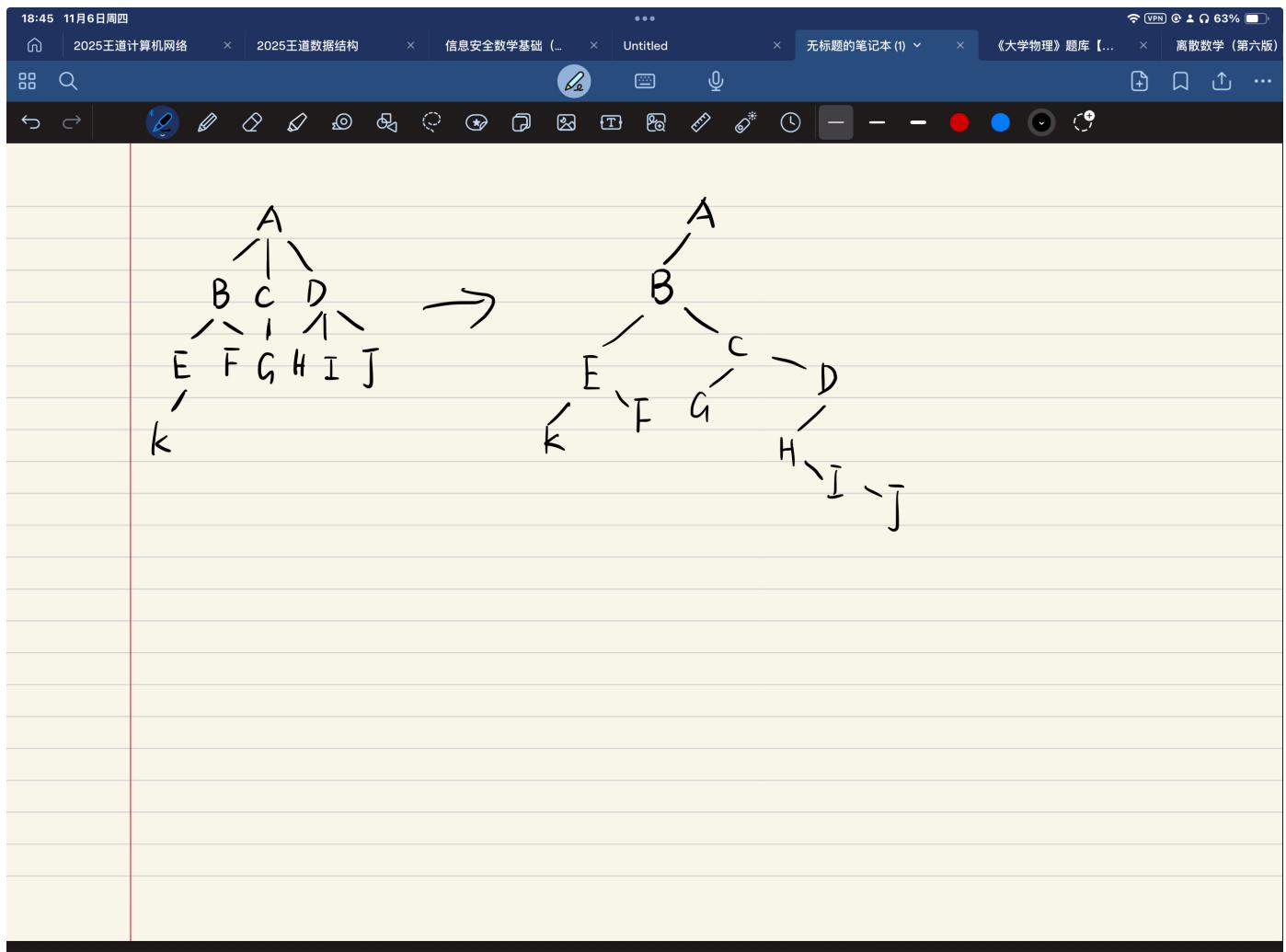
给出一棵树

```

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

```

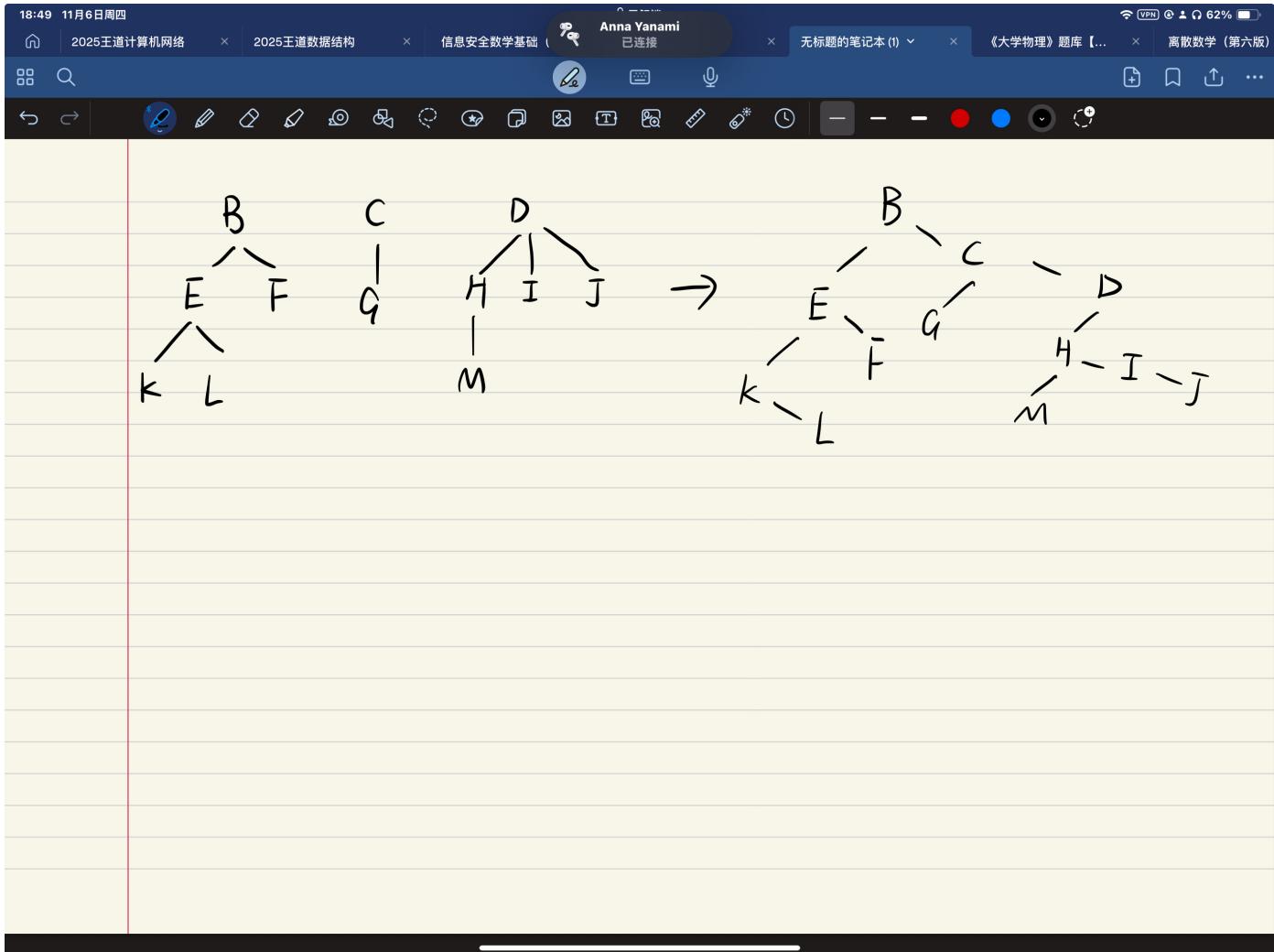
对应的孩子兄弟表示法如图所示



森林的存储结构

孩子兄弟表示法

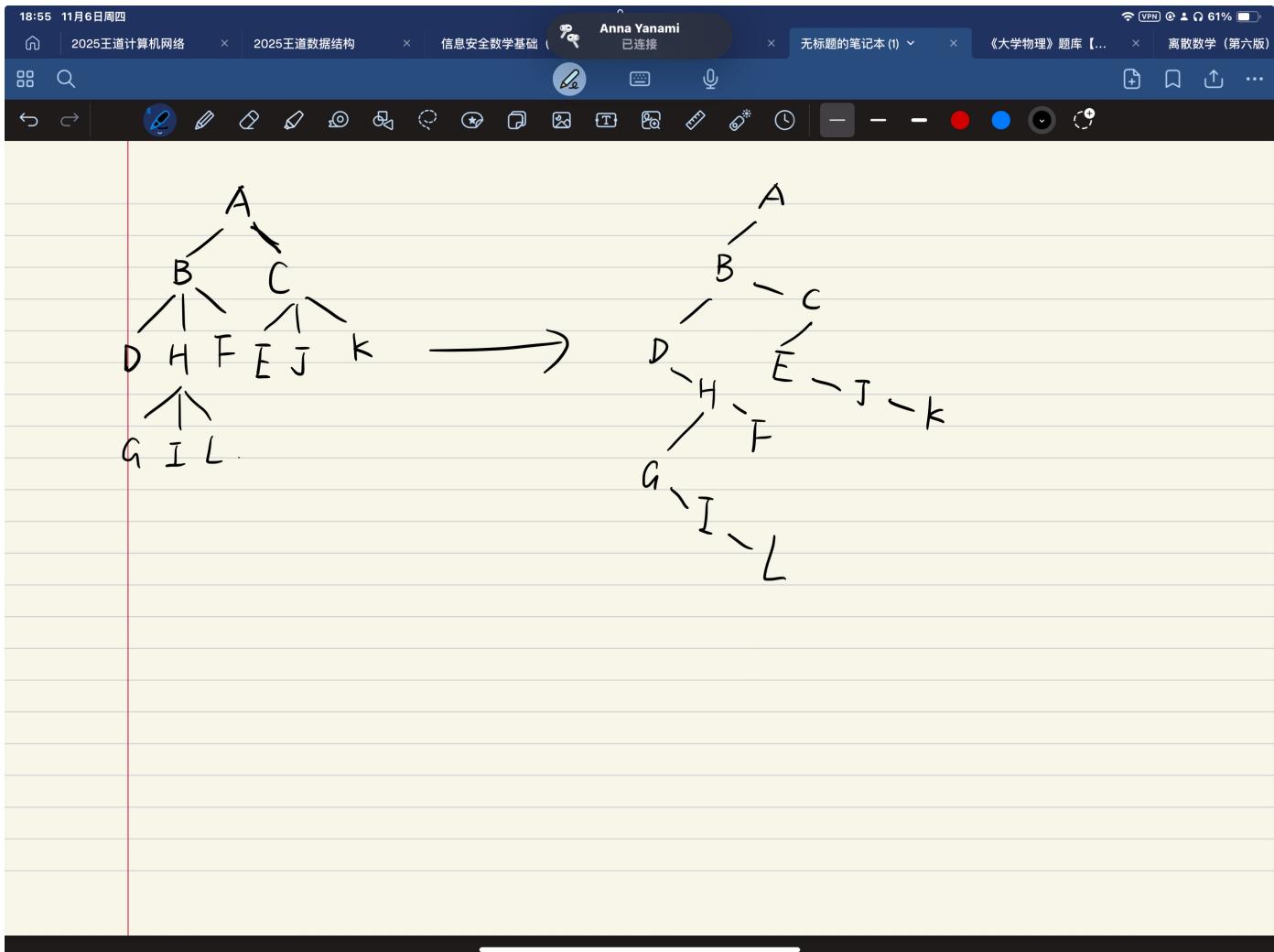
和树的存储略有不同，在森林中可以将每一颗树的树根视为平级的兄弟，如下图所示



树，森林，二叉树的转换

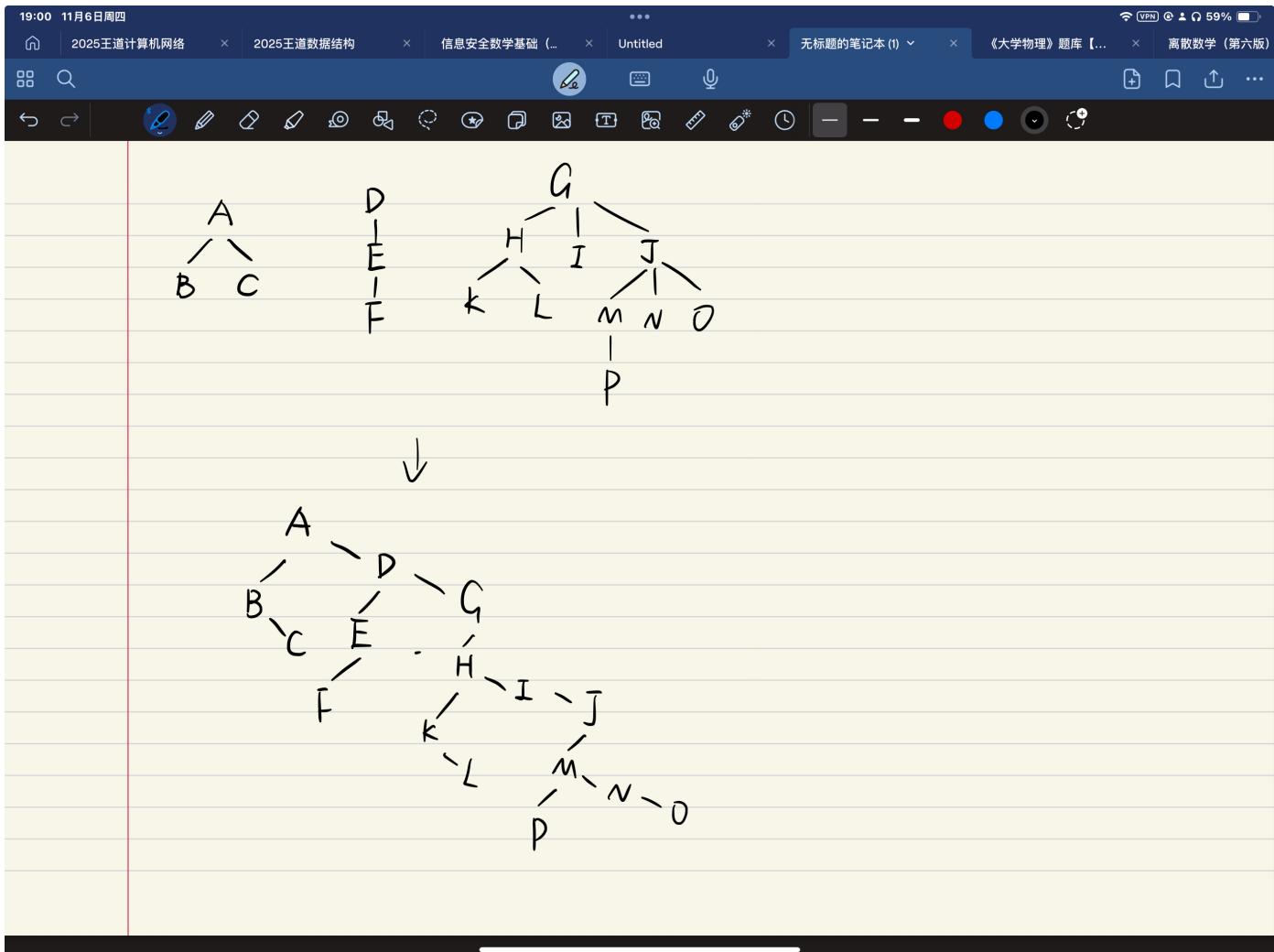
树转二叉树

就是用孩子兄弟表示法将这棵树转换为二叉树。



森林转二叉树

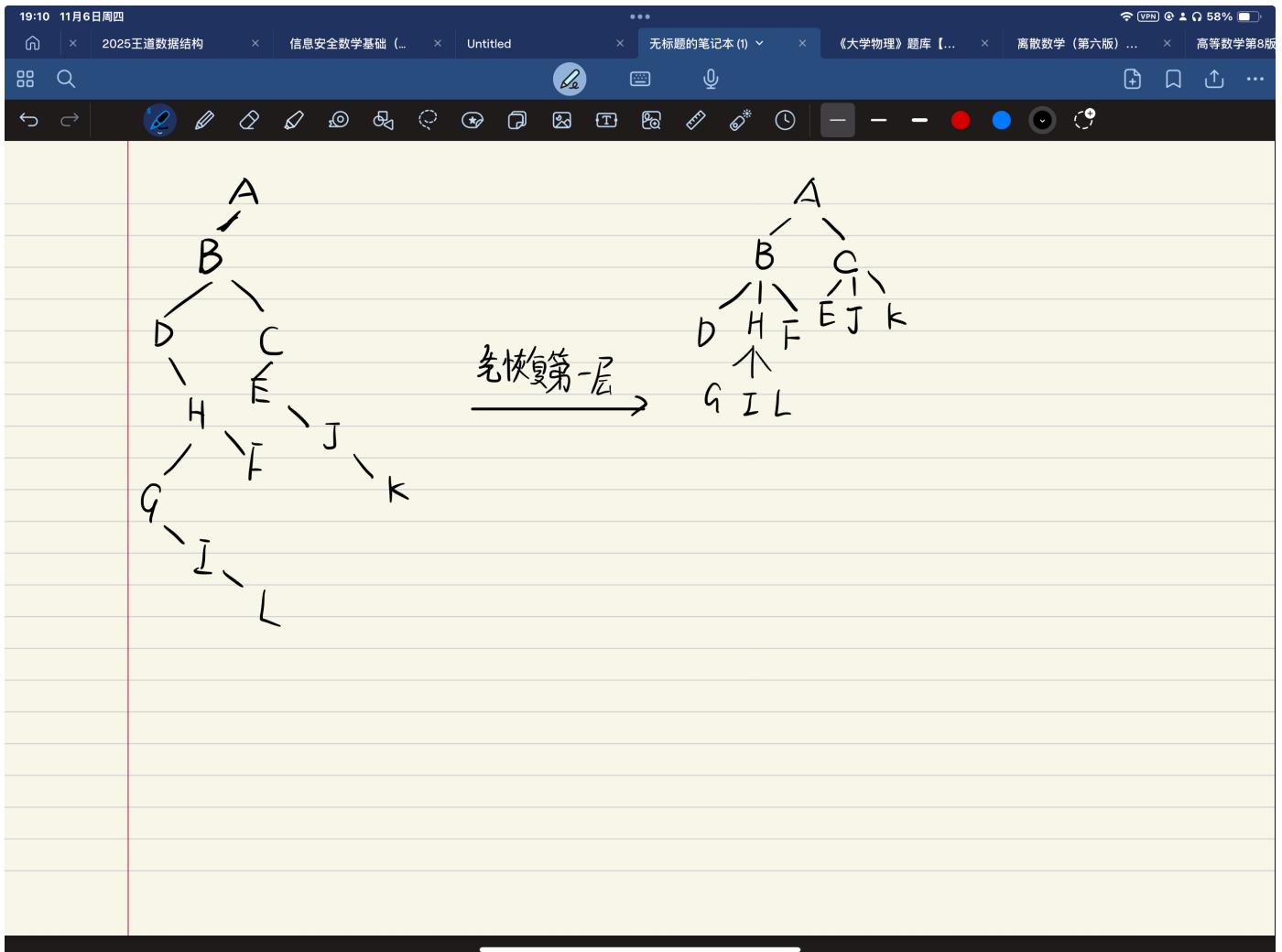
和树转二叉树一致。注意的是森林中各棵树的根节点视为平级的兄弟关系



二叉树转树

1. 先画出树的根节点
2. 从树的根节点开始，按树的层序（注意不是二叉树的层序）恢复每个节点的孩子

如何恢复每个节点的孩子：在二叉树中，如果当前处理的节点有左孩子，就把左孩子和“一整串右指针糖葫芦”拆下来，按顺序挂在当前节点的下方。

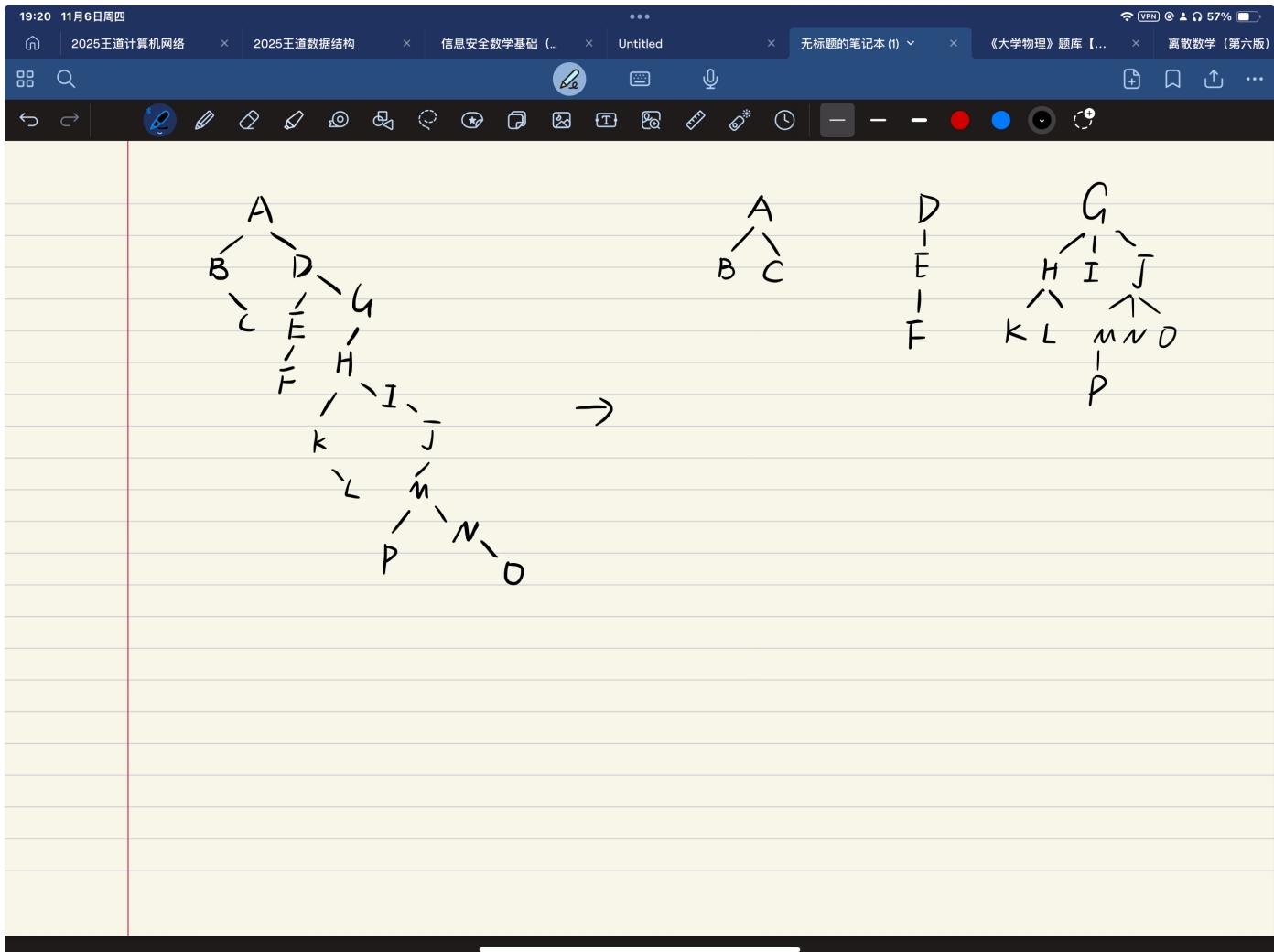


二叉树转森林

注意：森林中各棵树的根节点视为平级的兄弟关系

1. 先把二叉树的根节点和“一整串右指针糖葫芦”拆下来，作为多棵树的根节点
2. 按“森林的层序”恢复每个节点的孩子

如何恢复每个节点的孩子：在二叉树中，如果当前处理的节点有左孩子，就把左孩子和“一整串右指针糖葫芦”拆下来，按顺序挂在当前节点的下方。

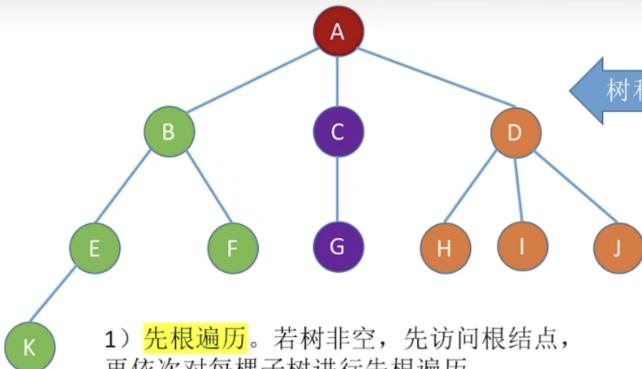


树的遍历

先根遍历

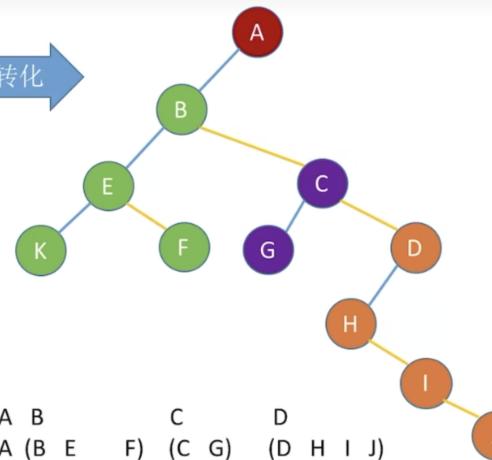
若树非空，先访问根节点，再依次对每棵子树进行先根遍历

树的先根遍历



1) 先根遍历。若树非空，先访问根结点，再依次对每棵子树进行先根遍历。

```
//树的先根遍历
void PreOrder(TreeNode *R){
    if (R!=NULL){
        visit(R); //访问根节点
        while(R还有下一个子树T)
            PreOrder(T); //先根遍历下一棵子树
    }
}
```



A B C D
A (B E F) (C G) (D H I J)
A (B (E K) F) (C G) (D H I J)

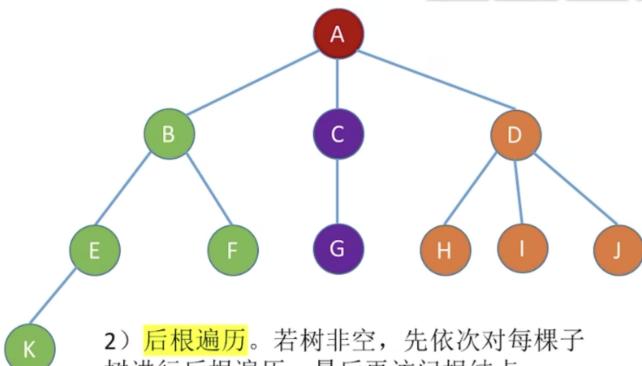
树的先根遍历序列与这棵树相应二叉树的先序序列相同。



后根遍历

和先根遍历大致是一致的。都是依次展开各个子树进行后根遍历

树的后根遍历



2) 后根遍历。若树非空，先依次对每棵子树进行后根遍历，最后再访问根结点。

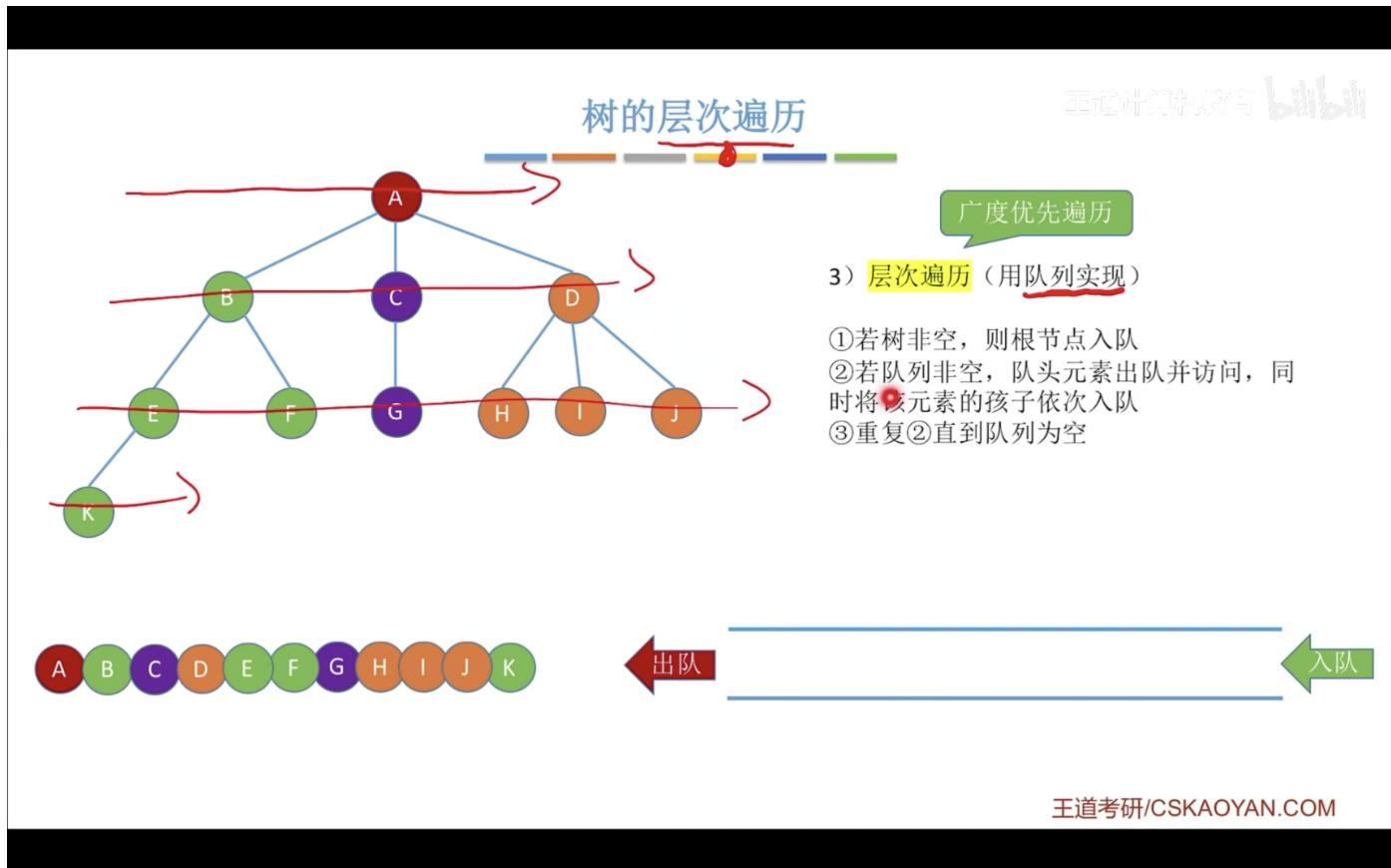
```
//树的后根遍历
void PostOrder(TreeNode *R){
    if (R!=NULL){
        while(R还有下一个子树T)
            PostOrder(T); //后根遍历下一棵子树
        visit(R); //访问根节点
    }
}
```

B C D A
(E F B) (G C) (H I J) D A
((K E) F B) (G C) (H I J) D A



层次遍历

1. 若树非空，则根节点入队
2. 若队列非空，队头元素出队并访问，同时将该元素的孩子依次入队
3. 重复②直到队列空为止



森林的遍历

先序遍历

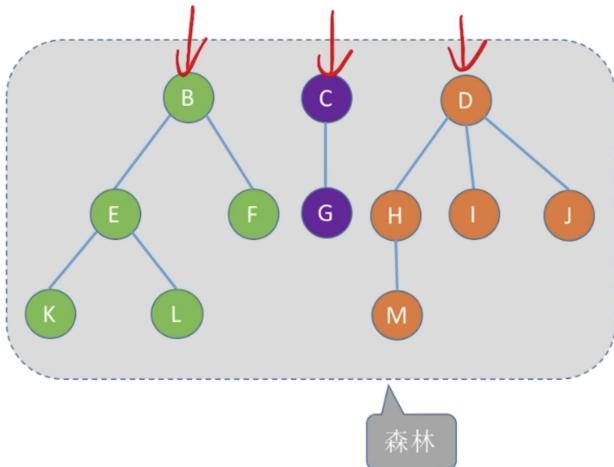
1. 访问森林中第一颗树的根节点
2. 先序遍历第一颗树中根节点的子树森林
3. 先序遍历除去第一颗树之后剩余的树构成的森林

效果等同于对各个子树进行先序遍历

森林的先序遍历

王道考研

森林。森林是 m ($m \geq 0$) 棵互不相交的树的集合。每棵树去掉根节点后，其各个子树又组成森林。



1) 先序遍历森林

若森林非空，则按如下规则进行遍历：

访问森林中第一棵树的根结点。

先序遍历第一棵树中根结点的子树森林。

先序遍历除去第一棵树之后剩余的树构成的森林。

B
(B E F)
→(B (E K L) F)

C
(C G)
(C G)

D
(D H I J)
(D (H M) I J)

效果等同于依次对各个树进行先根遍历



王道考研/CSKAOYAN.COM

中序/后序遍历

中序、后序遍历也是一样的，都是等同于对各个子树进行中序/后序遍历

哈夫曼树

节点的权值：有某种现实意义的值

节点的带权路径长度：树中所有叶节点的带权路径之和（从根节点到当前节点的长度乘当前节点的权值）

树的带权路径长度：树中所有叶子节点的带权路径长度之和

构造哈夫曼树

1. 将给出的权值进行从小到大排序，形成一个有序列表
2. 选出两个权值最小的节点作为左右子树，并且连接成一个新的节点，新节点的权值就是左右孩子权值相加
3. 在原来的有序列表中删除步骤二中选取的两个节点，用新节点替换，并重新排序
4. 若排序完的列表中权值最小的节点不包含新生成的节点，就要新开一棵子树，不用和原有的子树对齐层次，先生成完再进行对齐

例子

题目：在由六个字符组成的字符集S当中，各字符出现的频次分别为3, 4, 5, 6, 8, 10求这棵哈夫曼树

15:02 11月7日周五

2025王道计算机网络 2025王道数据结构 信息安全数学基础 (... Untitled 无标题的笔记本 (1) 《大学物理》题库 (...) 离散数学 (第六版)

VPN 94%

(3, 4, 5, 6, 8, 10)
(5, 6, 7, 8, 10)
(7, 8, 10, 11)
(10, 11, 15)

21

```
graph TD; 36[36] --> 15[15]; 36 --> 21[21]; 15 --> 7[7]; 15 --> 8[8]; 21 --> 11[11]; 21 --> 10[10]; 7 --> 3[3]; 7 --> 4[4]; 11 --> 5[5]; 11 --> 6[6]
```

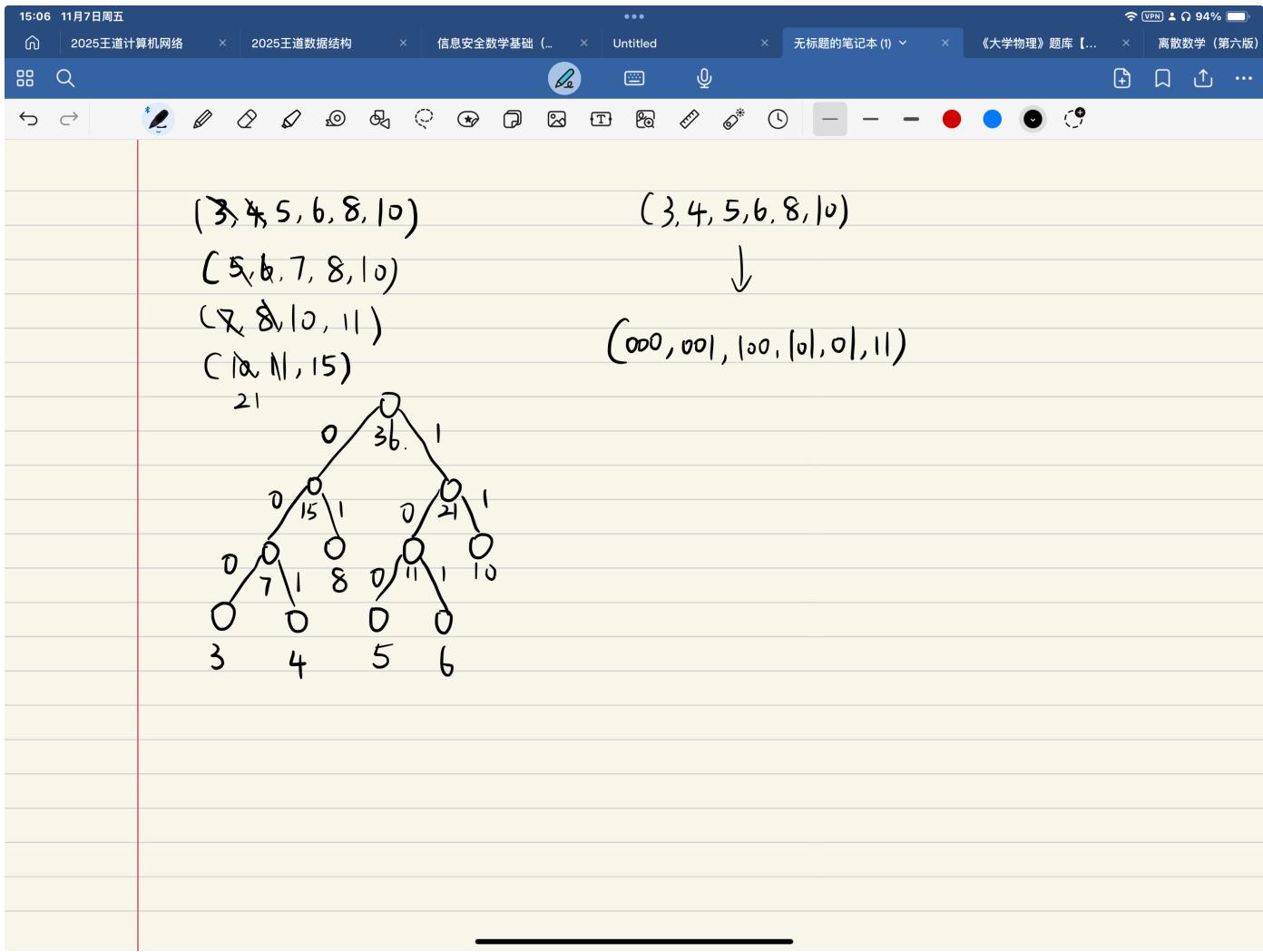
哈夫曼树的性质

1. 每个初始节点最终都成为叶子节点，且权值越小的节点到根节点的路径越长
2. 哈夫曼树的节点总数为 $2n-1$, n 为叶子节点的个数
3. 不存在度数为1的节点

哈夫曼编码

在哈夫曼树中，左分支路径标记为0，右分支路径标记为1，从根节点到每个叶子节点中的路径标记组成的序列即为该叶子节点字符编码

以上面例子中的哈夫曼树为例



可得到唯一的编码

图的基本概念

1. 路径

顶点p到q的路径是顶点序列, v_p, v_i, \dots, v_q

2. 回路

第一个顶点和最后一个顶点相同的路径称为回路或环

3. 简单路径

在路径序列中, 每一个顶点都不重复出现的路径

4. 简单回路

除了第一个和最后一个顶点之外, 其余的顶点都不重复的路径序列

5. 路径长度

路径上边的数目

6.点到点的距离

从顶点u出发到顶点v的最短距离若存在,则此路径的长度就是u到v的距离;若不存在,则距离为无穷

7.连通图(无向图)

图中任意两个顶点都连通

常考点:对于有n个顶点的无向图G

若G是连通图,则最少有 $n-1$ 条边,

若G是非连通图,则最多可能有 $C(2,n-1)$ 条边

8.强连通图(有向图)

图中任何一对顶点都是强联通的

常见考点:对于n个顶点的有向图G

若G是强联通图,则最少有n条边

9.无向图中顶点和边的关系

无向图中全部顶点的度之和等于边数的2倍

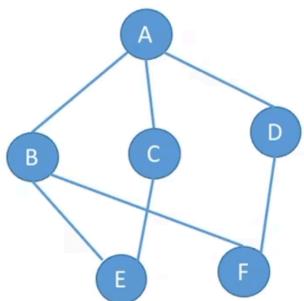
图的存储结构

邻接矩阵

稠密图(即边数较多的图)适合采用邻接矩阵法存储

图的存储——邻接矩阵法

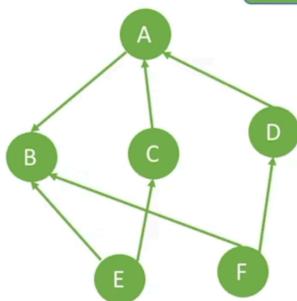
无向图



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	1	0
D	1	0	0	0	0	1
E	0	1	1	0	0	0
F	0	1	0	1	0	0

关闭弹幕

有向图



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	0
C	1	0	0	0	0	0
D	1	0	0	0	0	0
E	0	1	1	0	0	0
F	0	1	0	1	0	0

顶点中可以存
更复杂的信息

```
#define MaxVertexNum 100
typedef struct{
    char Vex[MaxVertexNum];
    int Edge[MaxVertexNum][MaxVertexNum];
    int vexnum,arcnum;
} MGraph;
```

//顶点数目的最大值

//顶点表

//邻接矩阵，边表

//图的当前顶点数和边数/弧数



王道考研/CSKAOYAN.COM

节点数为n的邻接矩阵是n*n的

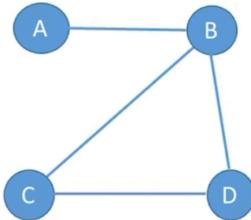
计算一个顶点的出度:看这个顶点所在行的非零元个数

计算一个顶点的入度:看这个顶点所在列的非零元个数

计算一个顶点的度:看这个顶点所在行,列的非零元个数之和

邻接矩阵的性质:

邻接矩阵法的性质



	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

设图G的邻接矩阵为 \mathbf{A} （矩阵元素为0/1），则 \mathbf{A}^n 的元素 $\mathbf{A}^n[i][j]$ 等于由顶点*i*到顶点*j*的长度为*n*的路径的数目

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix} * \begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{matrix}$$

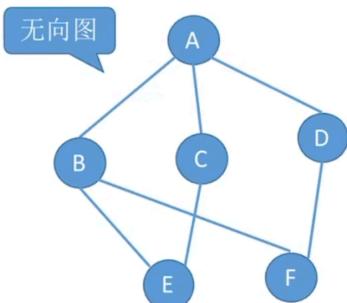
$$\mathbf{A}^2[1][4] = a_{1,1} a_{1,4} + a_{1,2} a_{2,4} + a_{1,3} a_{3,4} + a_{1,4} a_{4,4} = 1$$

$\overbrace{\mathbf{A} \rightarrow \mathbf{B}}^0 \cdot \overbrace{\mathbf{B} \rightarrow \mathbf{D}}^1 = 1$ $\overbrace{\mathbf{A} \rightarrow \mathbf{C}}^0 \cdot \overbrace{\mathbf{C} \rightarrow \mathbf{D}}^0 = 0$
 $\overbrace{\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{D}}^1$ $\mathbf{A} \rightarrow \mathbf{C} \rightarrow \mathbf{D} X$

王道考研/CSKAOYAN.COM

邻接表法

邻接表法（顺序+链式存储）



```
//用邻接表存储的图
typedef struct{
    AdjList vertices;
    int vexnum,arcnum;
} ALGraph;
```

```
/*"边/弧"
typedef struct ArcNode{
    int adjvex; //边/弧指向哪个结点
    struct ArcNode *next; //指向第一条弧的指针
    //InfoType info; //边权值
}ArcNode;
```

```
/*"顶点"
typedef struct VNode{
    VertexType data; //顶点信息
    ArcNode *first; //第一条边/弧
}VNode,AdjList[MaxVertexNum];
```



王道考研/CSKAOYAN.COM

邻接表适合存储稀疏图(即边数较少的图)

十字链表法

只能存储有向图，并且可以非常方便的求解顶点的入度和出度

图的遍历

BFS(广度优先遍历)

思路：以一个节点出发，然后不断查找与之相邻的节点，重复这样下去。

在查找相邻节点的时候，有可能查找到已经遍历过的节点。所以需要对已访问的节点进行标记

代码

```
#define MAX_VERTEX_NUM 10

bool visited[MAX_VERTEX_NUM];

void BFS(Graph G, int v) {
    visit(v); //访问初始顶点
    visited[v] = true; //对v做已访问标记
    EnQueue(Q,v); //顶点v入队
    while (!isEmpty(q)) {
        DeQueue(Q,v);
        for (w = FirstNeighbor(G,v); w >= 0; w = NextNeighbor(G,v,w)) {
            //检测v的所有邻接点
            if (!visited(w)) {
                visit(w);
                visited[w] = true;
                Enqueue(Q,w); //顶点w入队
            }
        }
    }
}
```

DFS

思想：先从图中某一顶点v出发，访问与v邻接且未被访问过的顶点w₁，接着再从w₁出发，访问与w₁邻接且未被访问过的顶点w₂，…以此类推。当不能再向下访问时，依次退回到最近访问的节点，若他还有未被访问的顶点，则从该顶点继续向下搜索

```
bool visited[MAX_VERTEX_NUM];

void DFSTra(Graph G) {
    for (int i = 0; i < G.vexnum; i++) {
        visited[i] = false;
    }
    for (int i = 0; i < G.vexnum; i++) {
        if (!visited[i]) {
            DFS(G,i);
        }
    }
}
```

```

void DFS(MGraph G, int i) {
    visit(i);
    visited[i] == true;
    for (;;) { // 检测i的所有邻接点
        if (!visited(j)) {
            DFS(G, j);
        }
    }
}

```

图的应用

最小生成树

简单来说就是用成本最低的方式将所有点连起来的结构，并且不存在环路

在图论中，给定一个带权重的无向连通图 $G=(V,E)$ ，如果它的一个子图 T 满足以下条件，那么 T 就是 G 的最小生成树：

1. 生成 (Spanning): T 包含了 G 中的所有顶点。
2. 树 (Tree): T 是连通的且没有环（如果有 n 个顶点，它恰好有 $n-1$ 条边）。
3. 最小 (Minimum): T 中所有边的权重之和是所有可能的生成树中最小的。

最小生成树的性质

- 最小生成树的边数等于顶点减一

构造最小生成树

Prim算法

算法思想：贪心算法。维护两个集合：已经选入树的点和边的集合 S_1 ，未被选入树的点和边的集合 S_2 。刚开始的时候 S_2 是整个图。

先随便选一个点，观察整个图，把权值最小的边和相应的点加入 S_1 中。之后在 S_2 中继续寻找权值最小的边，不断放入 S_1 中，直到边数等于顶点数减一。

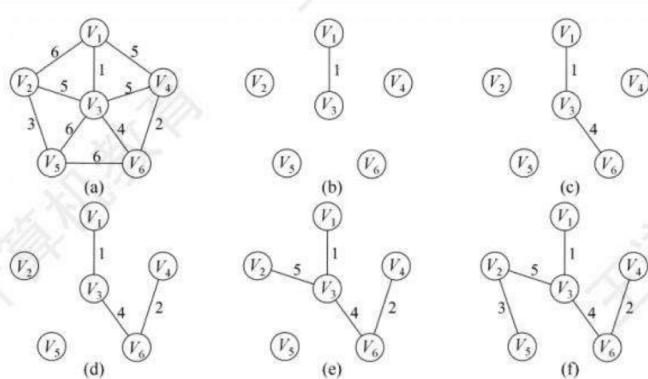


图 6.15 Prim 算法构造最小生成树的过程

克鲁斯卡尔算法

先把图中所有的权值按从小到大排序,不断选取当前未被选取过且权值最小的边.保证所有顶点都被选入且不构成环路,直到边数等于顶点数减一

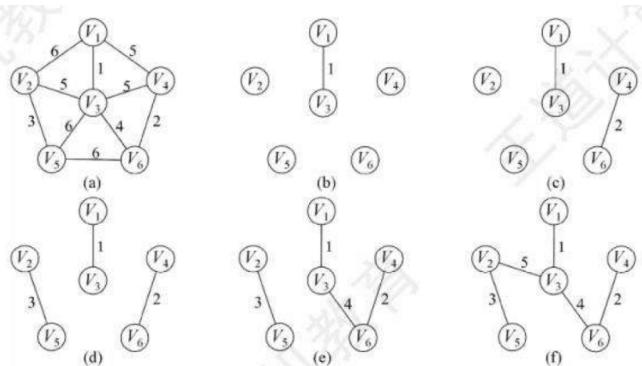


图 6.16 Kruskal 算法构造最小生成树的过程

最短路径

迪杰斯特拉算法

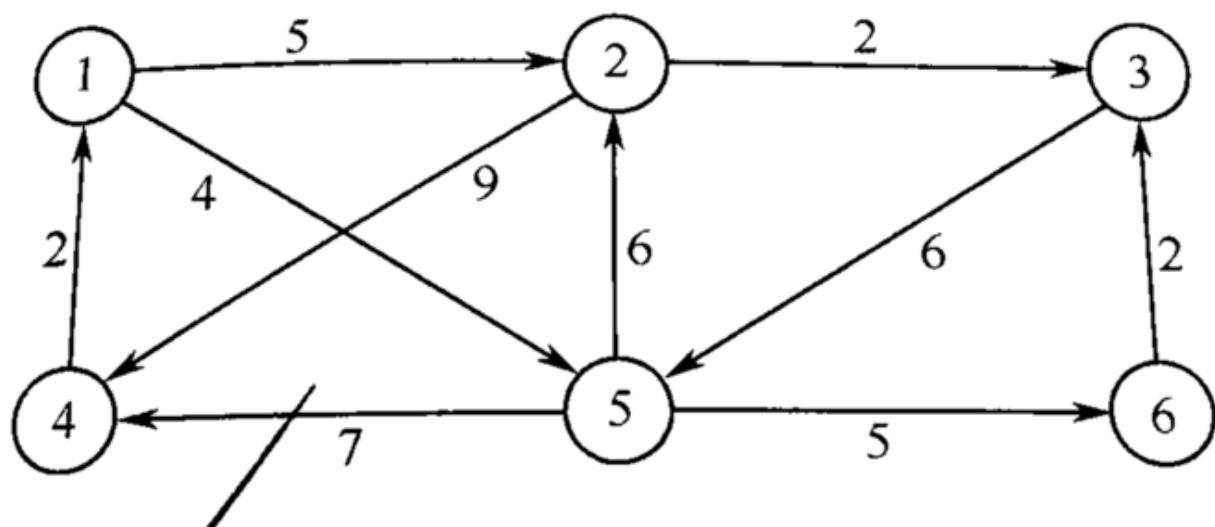
假设我们维护两个集合:

- **S (Sure):** 已经确定最短路的点 (已提起来的球)。
- **U (Unsure):** 还没确定最短路的点 (还在桌上的球)。

步骤循环:

1. 选最小: 扫描 U 中所有点, 找出 `dist` 值最小的那个点 u 。 (这就是贪心)
2. 移入 S : 把 u 移动到 S 集合中。 (盖棺定论, 不再修改 u)
3. 松弛邻居: 遍历 u 的所有邻居 v 。
 - 如果 `dist[u] + weight(u, v) < dist[v]`:
 - 更新 `dist[v] = dist[u] + weight(u, v)`。

例子:



初始状态

- 起点：1
 - 初始距离 (D):
 - $D[1]=0$
 - $D[2]=5$ (由 $1 \rightarrow 2$)
 - $D[5]=4$ (由 $1 \rightarrow 5$)
 - 其余顶点 (3,4,6) 为 ∞
-

第一步：选择当前最近的顶点

- 比较候选点：在未确定集合 {2,3,4,5,6} 中，当前距离 D 最小的是 顶点 5 (距离为 4)。
 - 操作：将 5 加入已确定集合。目标结点顺序：5。
 - 松弛邻居（从 5 出发更新）：
 - $5 \rightarrow 6$ (权重 5): $D[6]=D[5]+5=4+5=9$ 。
 - $5 \rightarrow 4$ (权重 7): $D[4]=D[5]+7=4+7=11$ 。
 - $5 \rightarrow 2$ (权重 6): 路径 $4+6=10$, 大于当前的 $D[2]=5$, 不更新。
 - 当前各点距离：
 - 2:5
 - 3: ∞
 - 4:11
 - 6:9
-

第二步：选择下一最近顶点

- 比较候选点：在 {2,3,4,6} 中，最小的是 顶点 2 (距离为 5)。
 - 操作：将 2 加入已确定集合。目标结点顺序：5, 2。
 - 松弛邻居（从 2 出发更新）：
 - $2 \rightarrow 3$ (权重 2): $D[3]=D[2]+2=5+2=7$ 。
 - $2 \rightarrow 4$ (权重 9): 路径 $5+9=14$, 大于当前的 11, 不更新。
 - 当前各点距离：
 - 3:7
 - 4:11
 - 6:9
-

第三步：选择下一最近顶点

- 比较候选点：在 {3,4,6} 中，最小的是 顶点 3 (距离为 7)。
- 操作：将 3 加入已确定集合。目标结点顺序：5, 2, 3。

- 松弛邻居： $3 \rightarrow 5$ 指向已确定集合，无需更新。

- 当前各点距离：

- 4:11

- 6:9

第四步：选择下一最近顶点

- 比较候选点：在 $\{4,6\}$ 中，最小的是 顶点 6 (距离为 9)。

- 操作：将 6 加入已确定集合。目标结点顺序：5, 2, 3, 6。

- 松弛邻居： $6 \rightarrow 3$ 指向已确定集合，无需更新。

- 当前各点距离：

- 4:11

第五步：选择最后一个顶点

- 比较候选点：只剩 顶点 4 (距离为 11)。

- 操作：将 4 加入已确定集合。

佛洛依德算法

复杂度太高,基本不采用

拓扑排序

简单来说就是找到做这些事情的先后顺序

实现

1. 从AOV网中选出一个没有前驱(入度为0)的顶点并输出

2. 从网中删除该顶点和所有以他为起点的有向边(把这个点和这个点的"手"删除)

3. 重复1,2直到当前的AOV网为空或者当前网中不存在无前驱的顶点为止

可以使用拓扑排序查看图是否有环

排序

二叉排序树

概念：

左子树上的值<根节点上的值

右子树上的值>根节点上的值

对二叉排序树进行中序遍历能得到一个有序升序序列

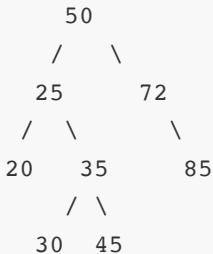
建立方式

1. 第一个元素是根节点
2. 后续元素比根节点小的,放左边,比根节点大的放右边

例子:

50、72、85、25、20、35、45、30

建立后的二叉排序树



查找方式

找出35需要的比较次数

1. 第一次比较: 与根节点 **50** 比较。
 - $35 < 50$, 往左走。
2. 第二次比较: 与左子节点 **25** 比较。
 - $35 > 25$, 往右走。
3. 第三次比较: 与右子节点 **35** 比较。
 - $35 == 35$, 找到了!

快速排序

排序思路:将列表的第一个元素视为枢轴.两个指针分别指向枢轴开始第一个元素和列表最后一个元素,从右到左找出第一个比枢轴小的数,从枢轴开始从左往右找第一个比枢轴大的数,二者交换位置,指针分别加一减一.直到指针相遇.指针相遇后需要将枢轴元素(列表第一个元素)与 `High` 指针指向的元素交换。

判断排序序列是不是第k趟快排结果

1. 先将序列从小到大排一遍
2. 找出已经归位的数字,即这个数字的左边比他小/没有数字,右边比他大/没有数字
3. 数一数这样的数字有几个,满足有k或者k+1个这样的数字即可

例子:

18. 【2019 统考真题】排序过程中, 对尚未确定最终位置的所有元素进行一遍处理称为一“趟”。下列序列中, 不可能是快速排序第二趟结果的是()。
- | | |
|---------------------------------|---------------------------------|
| A. 5, 2, 16, 12, 28, 60, 32, 72 | B. 2, 16, 5, 28, 12, 60, 32, 72 |
| C. 2, 12, 16, 5, 28, 32, 72, 60 | D. 5, 2, 12, 28, 16, 32, 72, 60 |

A,排序后的序列为2,5,12,16,28,32,60,72 找出已归位的数字为28,72,符合特征

B,2,5,12,16,28,32,60,72 找出已归位的数字为2,72,符合特征

C,找出已归位的数字为2,28,32,符合特征

堆排序

堆

堆的结构分为大根堆和小根堆

大根堆 (Max-Heap) : 每个节点的值都 \geq 其左右孩子的值。

小根堆 (Min-Heap) : 每个节点的值都 \leq 其左右孩子的值。

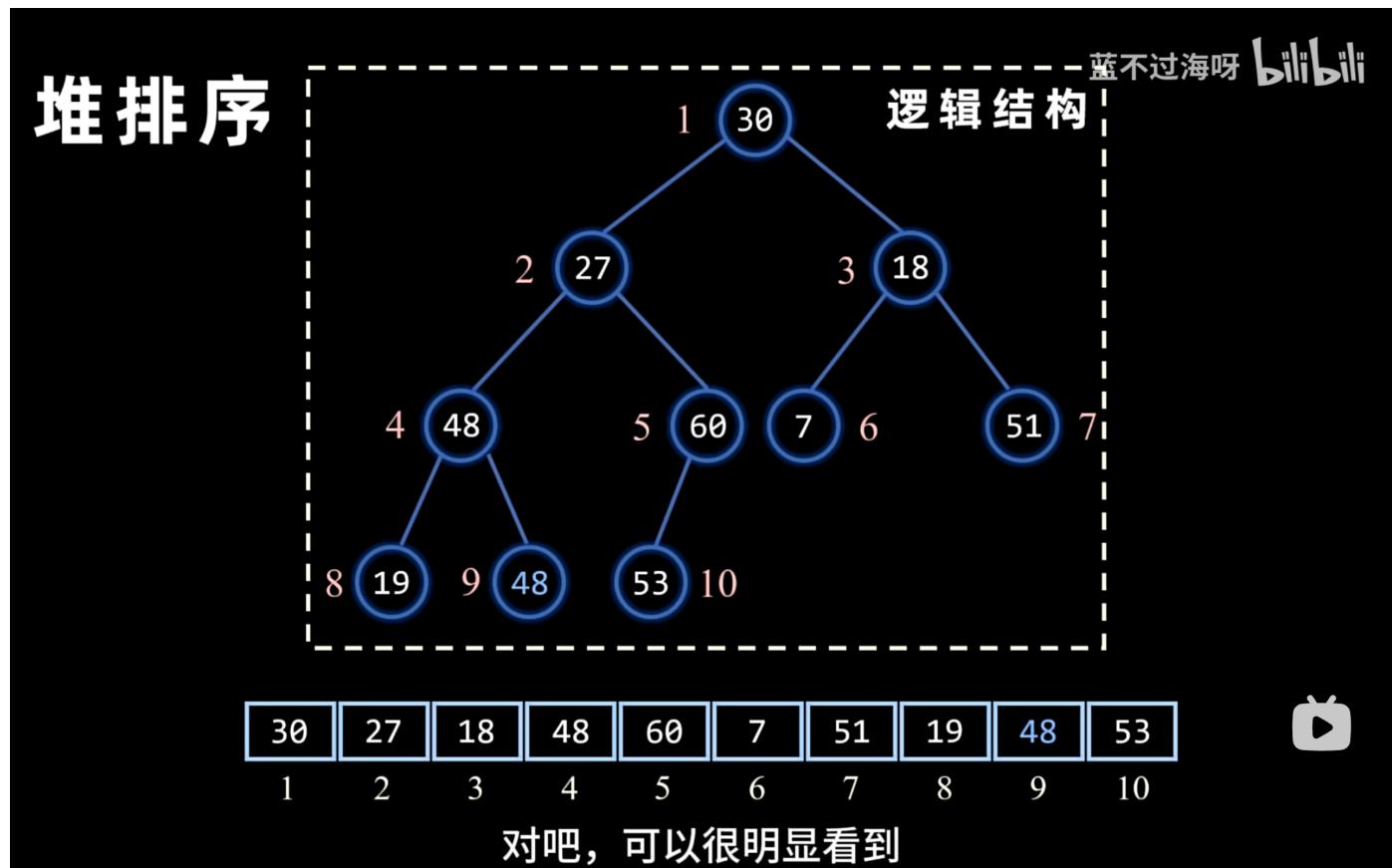
堆排序的过程

1. 先将序列转化为二叉树

数组转二叉树的规则 (假设数组下标从1开始) :

- 根节点: 索引 1
- 左孩子: 索引 $2 \times i$
- 右孩子: 索引 $2 \times i + 1$

例子:



2. 建立堆

这里建立大根堆为例

- 找到最后一个叶子节点,序号为 $n/2$
- 找到其父节点,序号为 $[n/2]$,作为起点
- 从起点开始向前操作,查看当前节点的值是不是比孩子节点的值要大
- 若当前节点的值比孩子节点的值小,那么将当前节点和孩子节点中值较大的节点作交换,直到它比孩子大或者变成叶子为止

冒泡排序

代码:

```
void BubbleSort (int a[],int n) {  
    for (int i = 0; i < n - 1; i++) {  
        bool flag = false;  
        for (int j = n - 1; j > i; j--) {  
            if (a[j-1] > a[j]) {  
                swap(a[j],a[j-1]);  
                flag = true;  
            }  
        }  
        if (flag == true) {  
            return ;  
        }  
    }  
}
```

每一趟排序后,最小的/最大的元素会到开头或者末尾处

冒泡排序最显著的特征是, 经过 k 趟排序后, 序列末尾一定有 k 个元素是已经排好序且处于最终位置的。

查找

二分查找

要求序列必须是递增序列

代码:

```
int binSearch (const vector<int>& l,int target) {  
    int left = 0;  
    int right = l.size() - 1;  
    while (left <= right) {  
        int mid = (left + right) / 2;  
        if (target < l[mid]) {  
            right = mid - 1;  
        }  
        if (target > l[mid]) {
```

```

    left = mid + 1;
}
if (target == l[mid]) {
    return mid;
}
return -1;
}

```

每次以区间中的中点作为mid,比较l[mid]和目标值的大小关系.目标值偏大,说明左区间中不可能有目标值,左指针移动到mid+1,反之说明右区间中不可能有目标值,右指针移动位mid-1.直到目标值和l[mid]相同

最大查找次数

可以用一颗二叉判定树的深度来描述:

$$h = \lfloor \log_2 n \rfloor + 1 \quad (4)$$

n是序列的长度

卡特兰数

共有两处地方需要用到卡特兰数

树

给定节点数n,求能构成多少形态不同的二叉树:

$$C_n = \frac{1}{n+1} C_{2n}^n = \frac{1}{n+1} \times \frac{(2n)!}{n! \times n!} \quad (5)$$

栈

一个栈的入栈序列是1,2,3,...n,求有多少种合法的出栈序列

$$C_n = \frac{1}{n+1} C_{2n}^n = \frac{1}{n+1} \times \frac{(2n)!}{n! \times n!} \quad (6)$$