第5章 树与二叉树

#数据结构与算法

5.1 树的定义与基本术语

5.1.1 定义

> 树T是包含n(n>=0)个结点的有限集合
> 在任意一棵非空树，有：
>
> 1. 有且只有一个根结点
> 2. 当n>1时，其余结点可分为m(m>0)个互不相交的有限集T1, T2, ..., Tm，其中每个集合又是一棵树，并且被称为根的子树

特点：递归定义；结点不均匀；层次结构不交叉；由若干子树构成

5.1.2 基本术语

1. **结点的度**：结点拥有的子树个数。
2. **树的度**：树内各结点的度的最大值。
3. **叶子结点** (终端结点)：度为0的结点。
4. **分支结点** (非终端结点)：度不为0的结点。
5. **孩子**：结点的子树的根。
6. **双亲**：孩子结点的上层结点。
7. **兄弟**：同一个双亲的孩子之间互称兄弟。
8. **结点的层次**：从根开始定义，根为第1层，根的孩子为第2层，以此类推。
9. **树的高度(深度)**：树中结点的最大层次。
10. **有序树**：树中结点的各子树从左至右是有次序的，不能互换。
11. **无序树**：树中结点的各子树从左至右是无次序的，可以互换。
12. **森林**：m(m>=0)棵互不相交的树的集合。

5.2 二叉树的定义，性质和存储结构

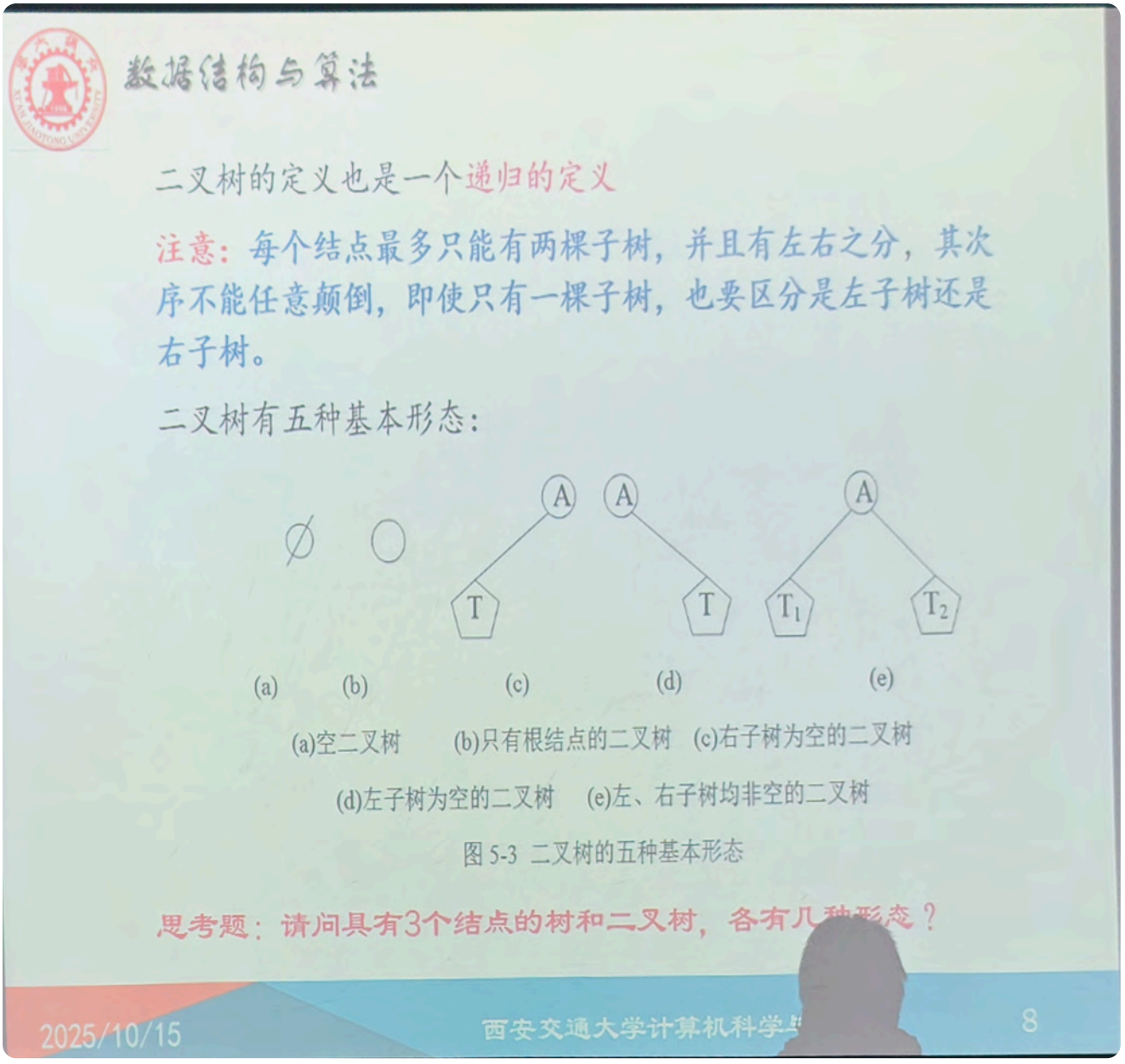
5.2.1 二叉树的定义

> 二叉树T是包含n(n>=0)个结点的有限集合
> 在任何一棵非空二叉树，有：
>
> 1. 有且只有一个根结点
> 2. 当n>1时，其余结点可分为两个互不相交的有限集L和R，其中每个集合又是一棵二叉树，并且分别称为根的左子树和右子树。

特点：

1. 每个结点最多有两棵子树
2. 左子树和右子树是有序的，次序不能颠倒
3. 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树

二叉树有五种形态



5.2.2 二叉树的性质

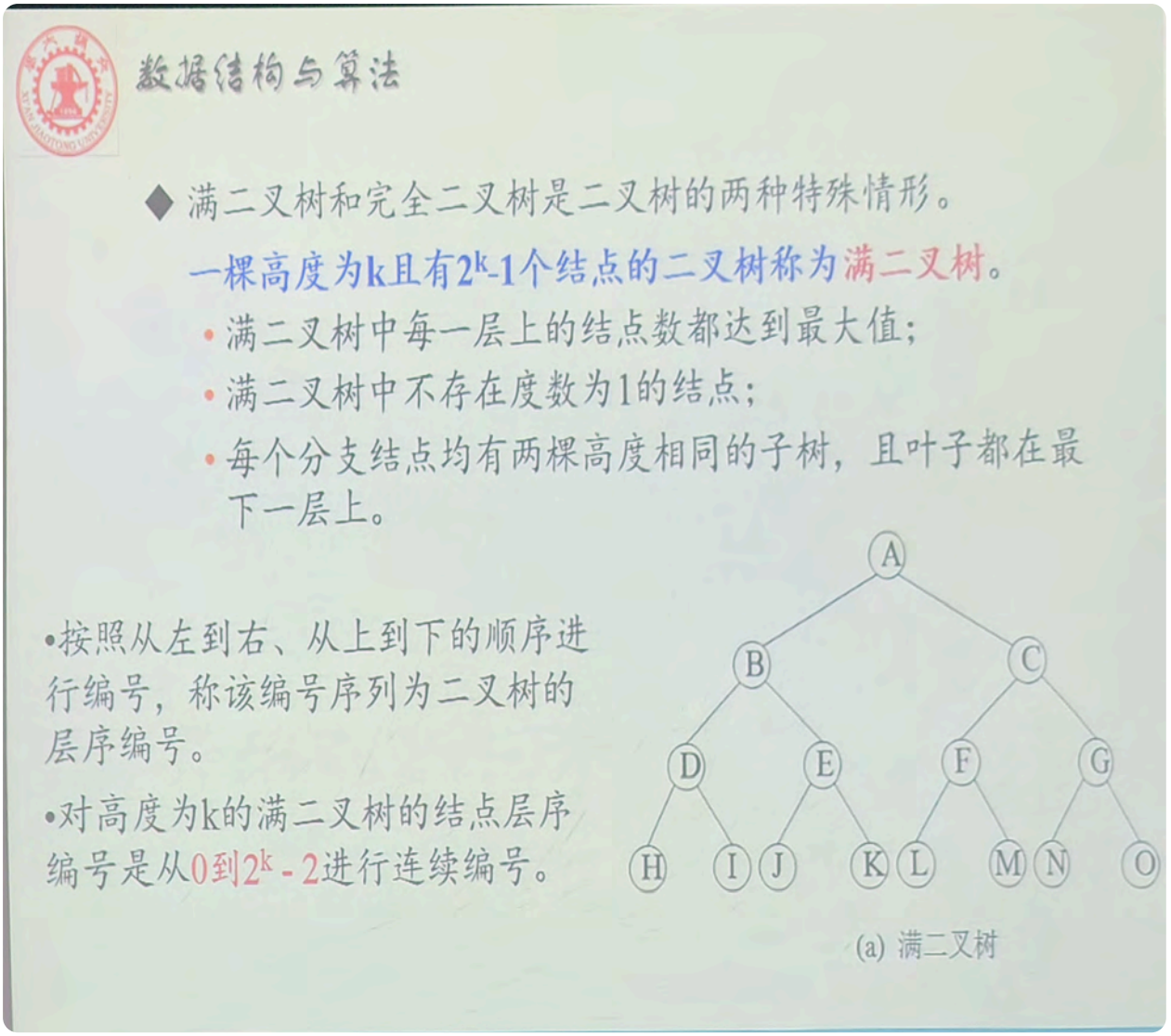
1. 在二叉树的第i层最多有2^{i-1} 个结点
2. 高度为k的二叉树最多有2^k-1个结点
3. 对于任意非空一棵二叉树T，如果叶子结点的个数为n_0,度为2的结点个数为n_2,则有

$$n_0 = n_2 + 1$$

> **证明：** $n = n_0 + n_1 + n_2$
> $n = e + 1$
> $e = n_1 + 2n_2$

定义满二叉树与完全二叉树

- **满二叉树**：一棵高度为h，且含有 $2^h - 1$ 个结点的二叉树。特点是每一层都含有最多的结点。
- **完全二叉树**：高度为h，有n个结点的二叉树，当且仅当其每一个结点都与高度为h的满二叉树中编号为1至n的结点一一对应时，称为完全二叉树。特点是叶子结点只可能在层次最大的两层上出现，且最大层次上的叶子结点都依次排列在该层最左边的位置上。



完全二叉树的性质

对于一棵有n个结点的完全二叉树，对结点从1到n进行连续编号（层序遍历），则对于编号为i(1<=i<=n)的结点：

1. 若 i=1，则结点i是根，无父结点；若 i>1，则其父结点是 $\lfloor i/2 \rfloor$。
2. 若 2i>n，则结点i无左孩子；否则其左孩子是 2i。
3. 若 2i+1>n，则结点i无右孩子；否则其右孩子是 2i+1。

> **证明** (数学归纳法):
> 我们用数学归纳法证明：对于任意结点 i，其左孩子是 $2i$，右孩子是 $2i+1$（如果存在)。
>
> 1. **基本情况**:
>   - 当 $i = 1$ 时，根结点的左孩子是第2个被编号的结点，右孩子是第3个。此时 $2i=2$，$2i+1=3$。性质成立。
> 2. **归纳假设**:
>   - 假设对于所有 $1 <= k < i$ 的结点，性质成立，即结点 k 的左、右孩子分别是 $2k$ 和 $2k+1$。
> 3. **归纳步骤**:
>   - 我们需要确定结点 i 的孩子的编号。根据层序遍历的定义，结点 i 的孩子会在所有编号小于 i 的结点的孩子都被编号之后，才会被编号。
>   - 具体来说，结点 i 的左孩子是在结点 $i-1$ 的所有孩子被编号之后，序列中的下一个。根据归纳假设，结点 $i-1$ 的右孩子（即其最后一个孩子）的编号是 $2(i-1) + 1 = 2i - 1$。
>   - 因为编号是连续的，所以紧随其后的结点，也就是 i 的左孩子，其编号必然是 $(2i - 1) + 1 = 2i$。i 的右孩子则顺理成章地被编号为 $2i + 1$。
>   - 因此，该性质对于结点 i 也成立。

因此，该性质常用于**完全二叉树**的**顺序存储**（数组实现)。

5.2.3 二叉树的存储结构

二叉树一般采用两种存储结构：顺序存储和链式存储。

需要实现的操作:

1. Initial(BT)
2. Build(BT)
3. Parent(VT,p)
4. L/R_Child(BT,p)
5. Empty(BT) //判断二叉树是否为空

进阶操作:

1. Insert(BT,p)
2. Delete(BT,p)
3. Leaf(BT) //求度为1或2的结点数
4. Traversal(BT) //遍历二叉树
   多种访问顺序:
   - preorder
   - inorder
   - postorder

- levelorder //层次遍历

1. 顺序存储

> 采用一组地址连续的存储单元（一维数组）来存放二叉树中的结点。

适用于**完全二叉树**。按照从上到下、从左到右的顺序将结点存入数组中。结点间的逻辑关系可以通过数组下标的计算来体现，如第5章 树与二叉树中所述。

优点：节省存储空间（不需要指针域），可以方便地找到任一结点的父结点和孩子结点。 **缺点**：对于非完全二叉树，会浪费大量存储空间。在进行插入、删除等操作时，可能需要移动大量元素，效率较低。

2. 链式存储

> 采用链表来存放二叉树中的结点，每个结点由数据域和指针域组成。

最常见的链式存储结构是**二叉链表**，每个结点包含三个字段：

- `data`：存放结点的数据。
- `lchild`：指向左子树根结点的指针。
- `rchild`：指向右子树根结点的指针。

```
template<class T>
struct BiTNode {
   T data;
   BiTNode<T> *lchild, *rchild;
};

template<class T>
using BiTree = BiTNode<T>*;
```

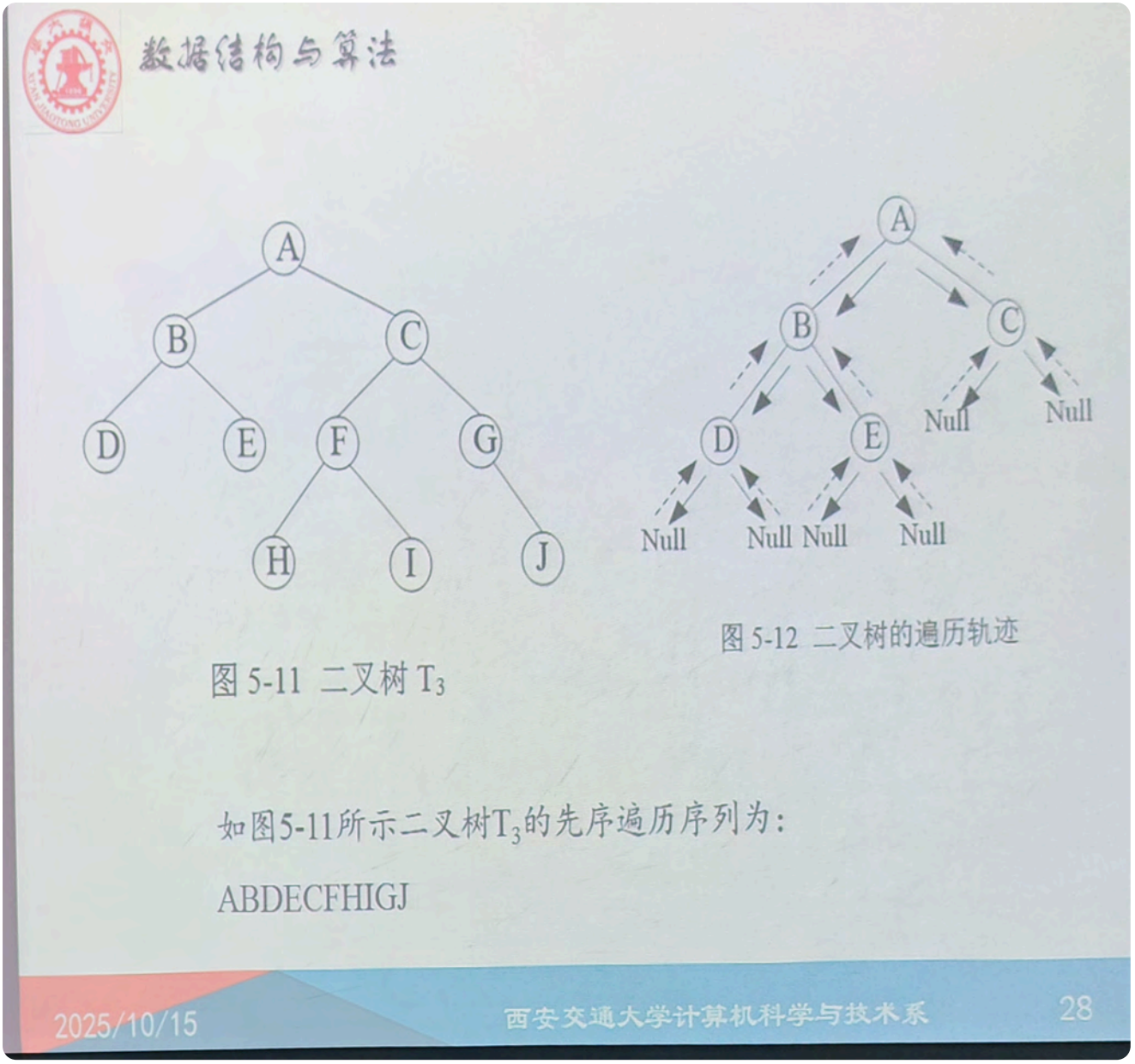
优点：插入、删除操作灵活方便，不会浪费存储空间。
缺点：需要额外的指针域，存储密度低。查找结点的父结点要从根结点开始遍历。

有时为了方便找到父结点，也会使用**三叉链表**（第5章 树与二叉树），在结点中增加一个指向父结点的指针 `parent`。

5.3 二叉树的遍历

遍历是二叉树最基本的操作，指按某种次序访问树中的每个结点，且每个结点仅被访问一次。

5.3.1 先序遍历



规则：若二叉树为空，则空操作；否则：

1. 访问根结点
2. 先序遍历左子树
3. 先序遍历右子树

递归实现

```
template<class T>
void PreOrder(BiTree<T> T) {
    if (T != nullptr) {
        visit(T); // 访问根结点
        PreOrder(T->lchild); // 递归遍历左子树
        PreOrder(T->rchild); // 递归遍历右子树
    }
}
```

非递归实现

> 使用栈来辅助

实现思路一

```
#include <stack>

template<class T>
void PreOrder_NonRecursive(BiTree<T> T) {
    if (T == nullptr) return;
    std::stack<BiTree<T>> S;
    BiTree<T> p = T;
    while (p || !S.empty()) {
        if (p) {
            visit(p);
            S.push(p);
            p = p->lchild;
        } else {
            p = S.top();
            S.pop();
            p = p->rchild;
        }
    }
}
```

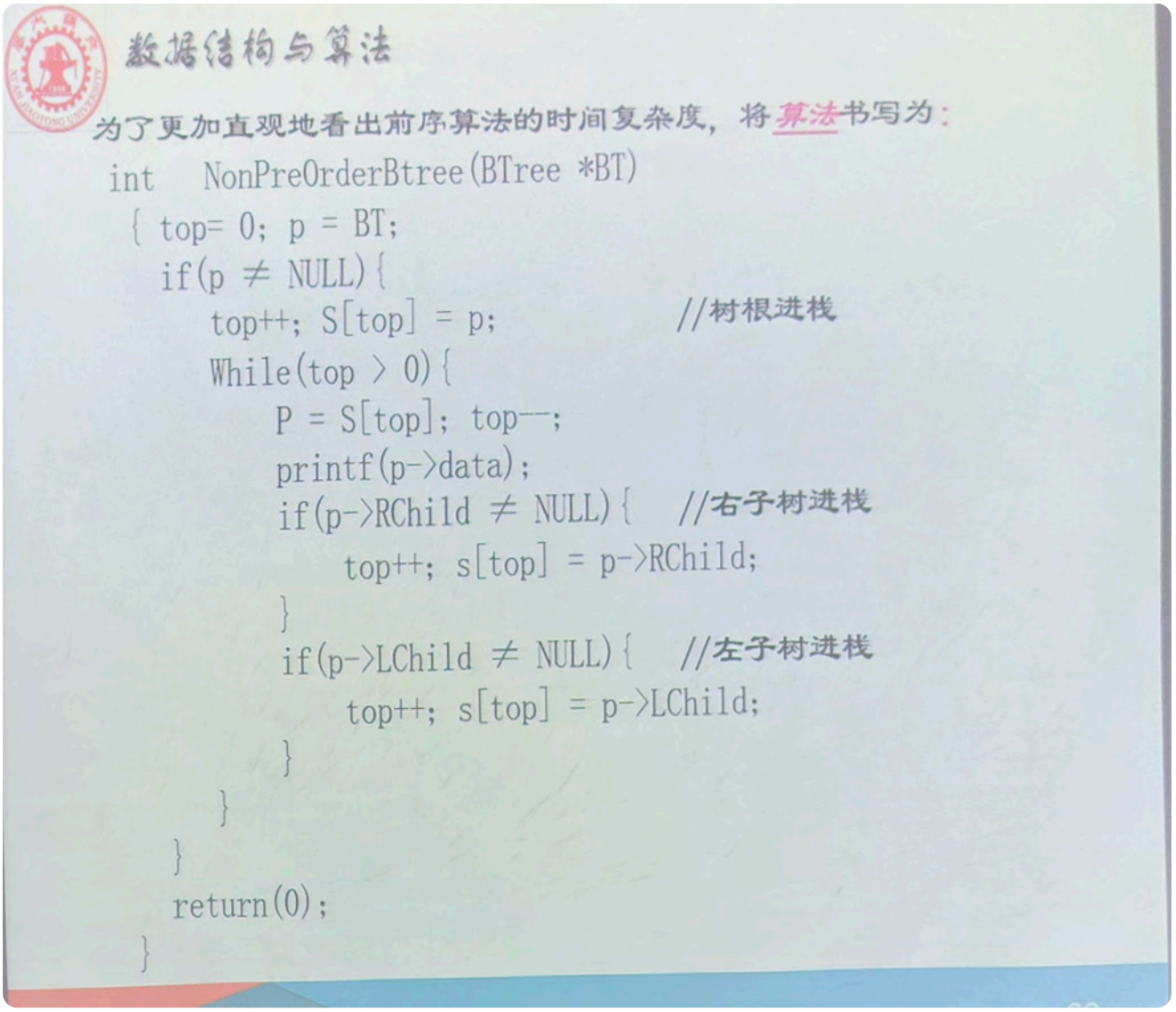
复杂度分析

时间复杂度: O(n)
循环的条件是 `p` 指针不为空或栈不为空。树中的每个结点 `p` 都会被访问(`visit`)一次、入栈一次、出栈一次，所有操作都是O(1)的。因此，总时间复杂度为 O(n)。
空间复杂度: O(h)
空间复杂度取决于栈S的最大深度。栈中存放的是从根到当前访问结点路径上的结点。栈的最大深度等于树的高度 `h` 。

最坏情况：当树退化为单链时， `h` 约等于 `n` ，空间复杂度为O(n)。
最佳情况：当树为完全二叉树时， `h` 约为 `log n` ，空间复杂度为O(log n)。

实现思路二（不能算标准先序算法）

如果优化一下算法，充分利用栈的性质：



详细复杂度分析

该算法使用一个显式栈来实现二叉树的先序遍历。
时间复杂度: O(n) 时间复杂度取决于对每个结点的操作次数。设树中结点总数为 n。

1. 入栈与出栈：在整个循环中，每个结点仅入栈一次，也仅出栈一次。根结点在循环开始前入栈，其他结点在其父结点被访问后入栈。
2. 访问结点：每个结点在从栈中弹出时被访问一次。
   由于每个结点的入栈、出栈和访问操作都是常数时间 $O(1)$，因此 n 个结点的总时间复杂度为 $n * O(1) = O(n)$。
   空间复杂度: O(h) 空间复杂度主要由辅助栈 s 的最大容量决定。

5.3.2 中序遍历

若二叉树为空，则空操作；否则：

1. 中序遍历左子树
2. 访问根结点
3. 中序遍历右子树

> 中序遍历的结果其实就是对树的投影从左往右的输出。

递归实现

```
template<class T>
void InOrder(BiTree<T> T) {
    if (T != nullptr) {
        InOrder(T->lchild); // 递归遍历左子树
        visit(T); // 访问根结点
        InOrder(T->rchild); // 递归遍历右子树
    }
}
```

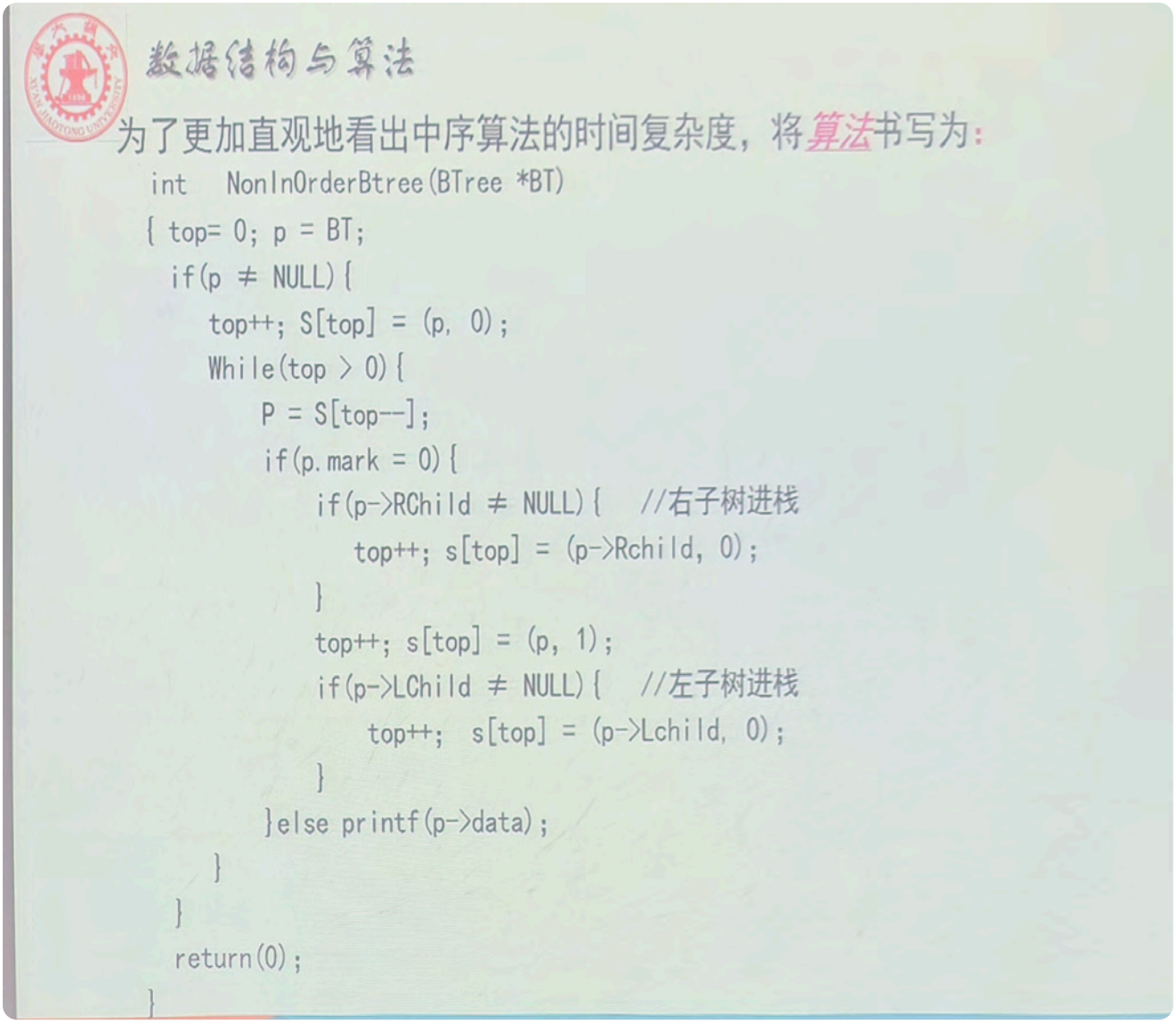
非递归实现

```
#include <stack>

template<class T>
void InOrder_NonRecursive(BiTree<T> T) {
    if (T == nullptr) return;
    std::stack<BiTree<T>> S;
    BiTree<T> p = T;
    while (p || !S.empty()) {
        if (p) {
            S.push(p);
            p = p->lchild; // 一直向左
        } else {
            p = S.top();
            S.pop();
            visit(p); // 访问出栈结点
            p = p->rchild; // 转向右子树
        }
    }
}
```

复杂度分析

为了更直观地分析，我们可以考虑另一种非递归实现，它通过给栈中元素添加一个标记(mark)来模拟递归的调用过程。



思路：

- 结点第一次入栈时，标记为0。
- 当一个标记为0的结点p出栈时，按 **右子 -> p(标记改为1) -> 左子** 的顺序将它们重新入栈。
- 当一个标记为1的结点出栈时，访问该结点。

```
#include <stack>

template<class T>
void InOrder_NonRecursive_Mark(BiTree<T> T) {
    if (T == nullptr) return;

    // 栈中元素为 (结点指针, 标记)
    std::stack<std::pair<BiTree<T>, int>> S;
    S.push({T, 0});

    while (!S.empty()) {
```

```
        auto [p, mark] = S.top();
        S.pop();

        if (p == nullptr) continue;

        if (mark == 0) {
            // 按右、中、左的顺序入栈
            if (p->rchild != nullptr) {
                S.push({p->rchild, 0});
            }
            S.push({p, 1}); // 再次入栈，标记为1
            if (p->lchild != nullptr) {
                S.push({p->lchild, 0});
            }
        } else { // mark == 1
            visit(p); // 第二次遇到，访问结点
        }
    }
}
```

时间复杂度分析:

- 每个结点都会入栈两次（一次标记为0，一次标记为1），相应地也会出栈两次。对于树中的 `n` 个结点，总的入栈和出栈操作次数是 `2n + 2n = 4n`。
- `visit` 操作对每个结点执行一次。
- 所有操作都是常数时间的，因此总时间复杂度为 **O(n)**。

空间复杂度分析:
空间复杂度取决于栈的最大深度。在最坏情况下（树退化为单链），栈需要存储从根到叶子的整条路径上的所有结点，空间复杂度为 **O(n)**。在最好情况下（完全二叉树），树的高度为 `log n`，空间复杂度为 **O(log n)**。通常表示为 **O(h)**，其中h是树的高度。

5.3.3 后序遍历

规则：若二叉树为空，则空操作；否则：

1. 后序遍历左子树
2. 后序遍历右子树
3. 访问根结点

递归实现

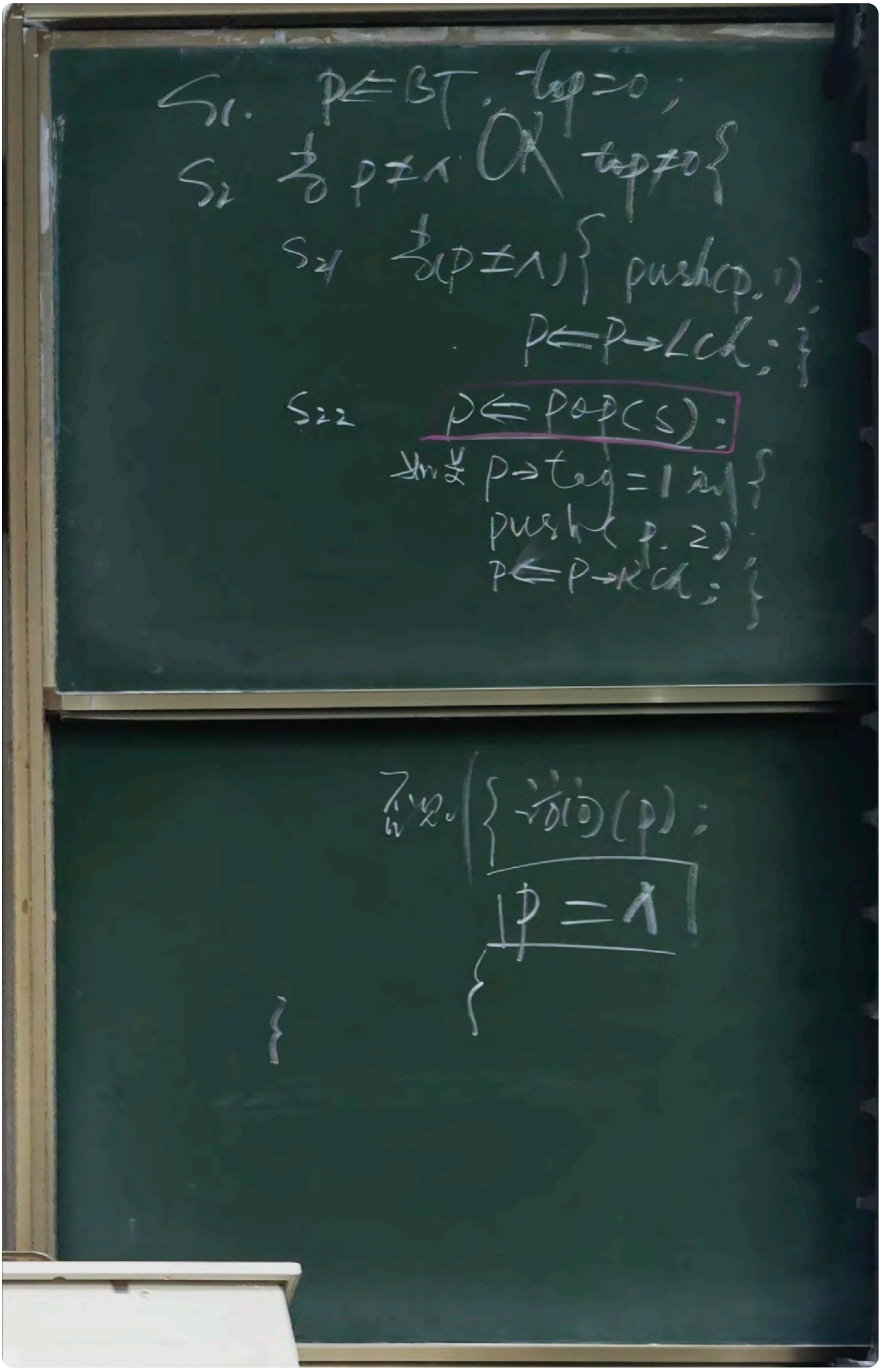
```
template<class T>
void PostOrder(BiTree<T> T) {
```

```
    if (T != nullptr) {
        PostOrder(T->lchild); // 递归遍历左子树
        PostOrder(T->rchild); // 递归遍历右子树
        visit(T); // 访问根结点
    }
}
```

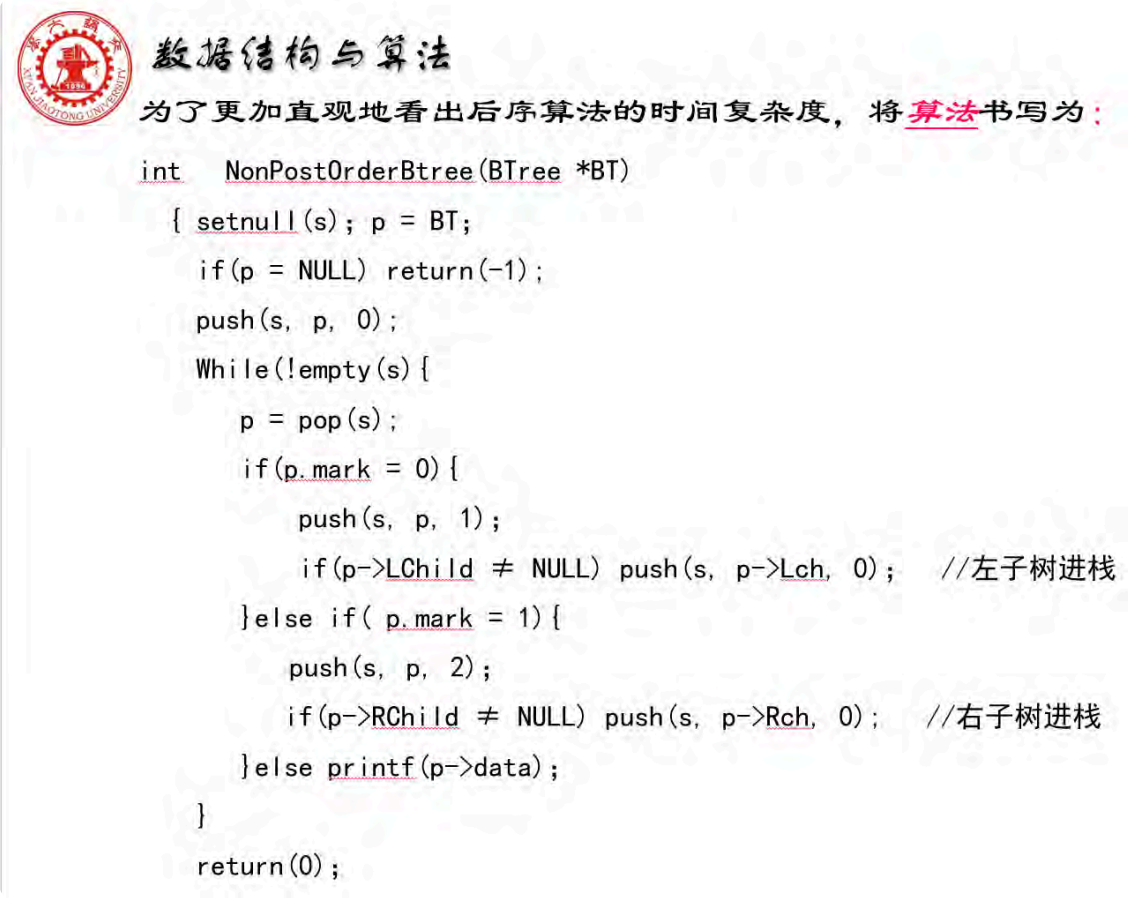
非递归实现

后序遍历的非递归实现较复杂，需要判断结点是第几次出现在栈顶。

算法分析：


S1. P←BT. top=0;
S2 若 P≠Λ OR top≠0 {
S21 若(P≠Λ){ push(P,1);
P←P→Lch; }
S22 P←POP(S);
如果 P→tag=1 则{
push(P,2);
P←P→Rch; }
}
否则{ 访问(P);
P=Λ
{

为了更直观地分析时间复杂度，可以采用一种使用三个标记状态（0, 1, 2）的非递归算法来模拟后序遍历的访问过程。这种方法中，每个结点会根据其处理阶段入栈三次。



算法思路：

- `mark = 0`：结点第一次被处理。此时应先遍历其左子树。算法会将该结点以 `mark=1` 重新入栈，然后将其左孩子（如果存在）以 `mark=0` 入栈。
- `mark = 1`：结点第二次被处理，说明其左子树已遍历完毕。此时应遍历其右子树。算法会将该结点以 `mark=2` 重新入栈，然后将其右孩子（如果存在）以 `mark=0` 入栈。
- `mark = 2`：结点第三次被处理，说明其左右子树均已遍历完毕。此时，可以访问该结点。

```
#include <stack>
#include <utility> // for std::pair

template<class T>
void PostOrder_NonRecursive_Mark(BiTree<T> T) {
    if (T == nullptr) return;

    std::stack<std::pair<BiTree<T>, int>> S;
    S.push({T, 0}); // 根结点第一次入栈，标记为0

    while (!S.empty()) {
        auto [p, mark] = S.top();
```

```
        S.pop();

        if (p == nullptr) continue;

        if (mark == 0) {        // 第一次遇到，处理左子树
            S.push({p, 1});
            if (p->lchild != nullptr) {
                S.push({p->lchild, 0});
            }
        } else if (mark == 1) { // 第二次遇到，处理右子树
            S.push({p, 2});
            if (p->rchild != nullptr) {
                S.push({p->rchild, 0});
            }
        } else {            // mark == 2，第三次遇到，访问结点
            visit(p);
        }
    }
}
```

复杂度分析

时间复杂度：**O(n)** 对于树中的每个结点，都会以标记0、1、2分别入栈和出栈一次。因此，每个结点总共入栈3次，出栈3次。
空间复杂度：**O(h)**

5.3.4 层次遍历

规则：从根结点开始，按从上到下、从左到右的层次顺序访问树中的每个结点。

实现思路：

> 使用一个队列来辅助实现。
>
> 1. 将根结点入队。
> 2. 当队列不为空时，循环执行以下操作：
>   a. 队头结点出队并访问。
>   b. 若该结点的左孩子不为空，则将其左孩子入队。
>   c. 若该结点的右孩子不为空，则将其右孩子入队。

C++ 实现：

```
#include <queue>

template<class T>
```

```
void LevelOrder(BiTree<T> T) {
    if (T == nullptr) return;
    std::queue<BiTree<T>> Q;
    Q.push(T);
    while (!Q.empty()) {
        BiTree<T> p = Q.front();
        Q.pop();
        visit(p); // 访问出队结点
        if (p->lchild != nullptr) {
            Q.push(p->lchild);
        }
        if (p->rchild != nullptr) {
            Q.push(p->rchild);
        }
    }
}
```

复杂度分析：

时间复杂度: O(n)

每个结点都会入队一次、出队一次。因此，总时间复杂度为 O(n)。

空间复杂度: O(w)

空间复杂度取决于队列的最大长度，即树的最大宽度 `w`。

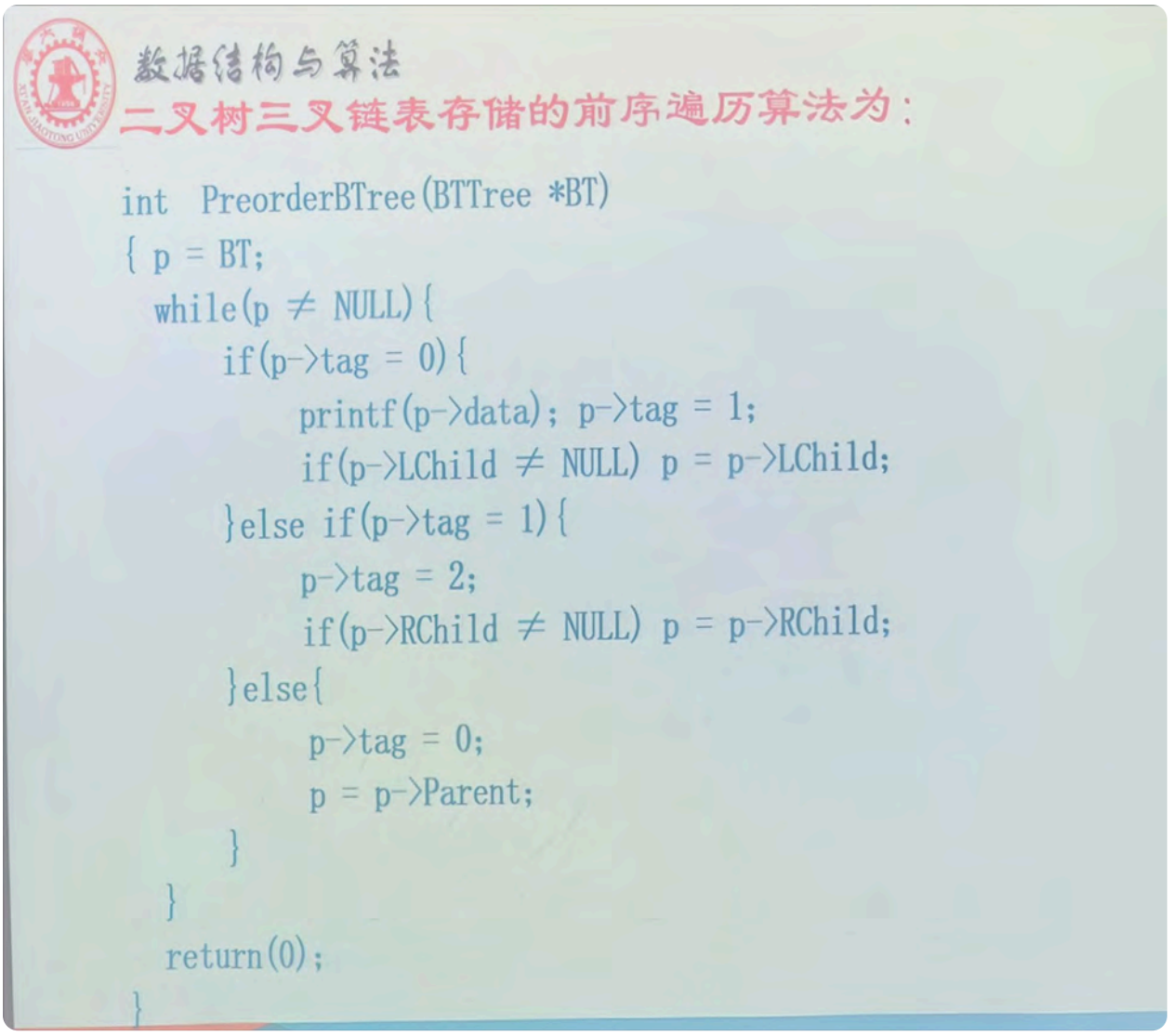
最坏情况：对于一棵完全二叉树，最后一层结点数最多，约为 `n/2`。此时为 O(n)。

最坏情况(退化)：当树退化为单链时，队列中最多只有一个元素，空间复杂度为 O(1)。

5.3.5 二叉树的三叉链表存储遍历（张亚明推荐）

在三叉链表（即结点中包含 `parent` 指针）的存储结构下，标准的先序、中序、后序和层次遍历算法逻辑本身没有变化，因为它们主要依赖 `lchild` 和 `rchild` 指针。

- 例：改进的前序算法



三叉链表以牺牲少量空间为代价（增加一个指针域），换取了向上导航的灵活性，简化了部分复杂算法的实现。

二叉树性质：

- 给出前序序列和中序序列，就能构造完整的二叉树。
- 给出后序序列和中序序列，就能构造完整的二叉树。
- 给出前序序列和后序序列，不能构造唯一的二叉树。
- 一棵二叉树的前序，中序，后序遍历序列中，叶子结点的相对位置不会发生变化。
- n+1个指针为空，n-1个指针有用。

5.3.6 线索二叉树 (Threaded Binary Tree)

在n个结点的二叉链表中，有n+1个空指针域。利用这些空指针域存放指向结点在某种遍历次序下的前驱和后继结点的指针，这些指针称为"线索"，加上线索的二叉树称为线索二叉树(TBT)。

结构定义

为了区分指针指向的是孩子还是线索，需要增加两个标志位 `LTag` 和 `RTag`：

- `LTag = 0`：`lchild` 指向左孩子
- `LTag = 1`：`lchild` 指向中序前驱 (线索)
- `RTag = 0`：`rchild` 指向右孩子
- `RTag = 1`：`rchild` 指向中序后继 (线索)

```
template<class T>
struct ThreadNode {
    T data;
    ThreadNode<T> *lchild, *rchild;
    int LTag, RTag; // 左右标志
};

template<class T>
using ThreadTree = ThreadNode<T>*;
```

5.4 二叉树应用：哈夫曼树

> **哈夫曼树**又称**最优二叉树**，由Huffman提出，是带权路径长度最短的树。（不唯一）

定义：假设有n个结点，每个结点的权值分别为$\omega_1, \omega_2, \dots \omega_n$,构造一棵以这$n$个结点为叶子结点的二叉树，则其中带权路径长度最短WPL的树称为哈夫曼树。

5.4.1 相关术语

1. **路径**：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。
2. **路径长度**：路径上的分支数目。
3. **结点的权 (Weight)**：赋予某个结点的数值，通常代表其重要性或出现的频率。
4. **结点的带权路径长度**：从根结点到该结点的路径长度与该结点权值的乘积。
5. **树的带权路径长度 (WPL)**：树中所有叶子结点的带权路径长度之和。通常记为：

$$WPL = \sum_{i=1}^{n} w_i l_i$$

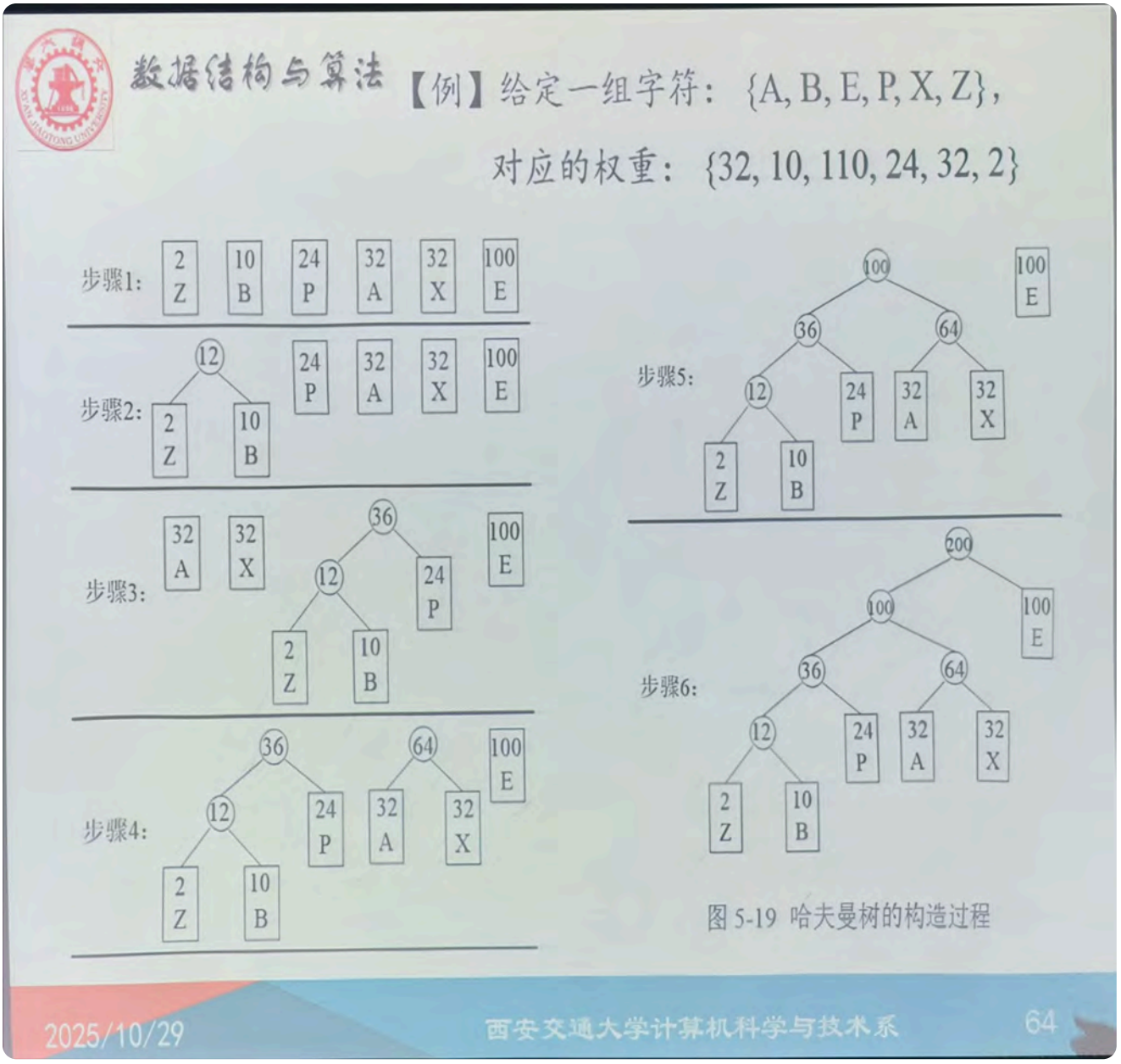
   其中，n是叶子结点的数目，w_i是第i个叶子结点的权值，l_i是第i个叶子结点到根结点的路径长度。

5.4.2 哈夫曼树的构造过程

哈夫曼树的构造遵循贪心策略，每次选择权值最小的结点进行合并。

算法步骤：

1. **初始化**：根据给定的n个权值$\{w_1, w_2, \ldots, w_n\}$，构造n棵二叉树的集合$F = \{T_1, T_2, \ldots, T_n\}$，其中每棵树$T_i$只有一个带权为$w_i$的根结点。
2. **合并**：当集合F中还有多于两棵树时，重复以下操作：
   a. 从F中选取两棵根结点权值最小的树。
   b. 创建一个新的根结点，其权值为这两棵树根结点权值之和。
   c. 将这两棵树分别作为新结点的左、右子树。
   d. 从F中删除被选取的两棵树，并将新生成的树加入F中。
3. **完成**：当F中只剩下一棵树时，这棵树就是哈夫曼树。



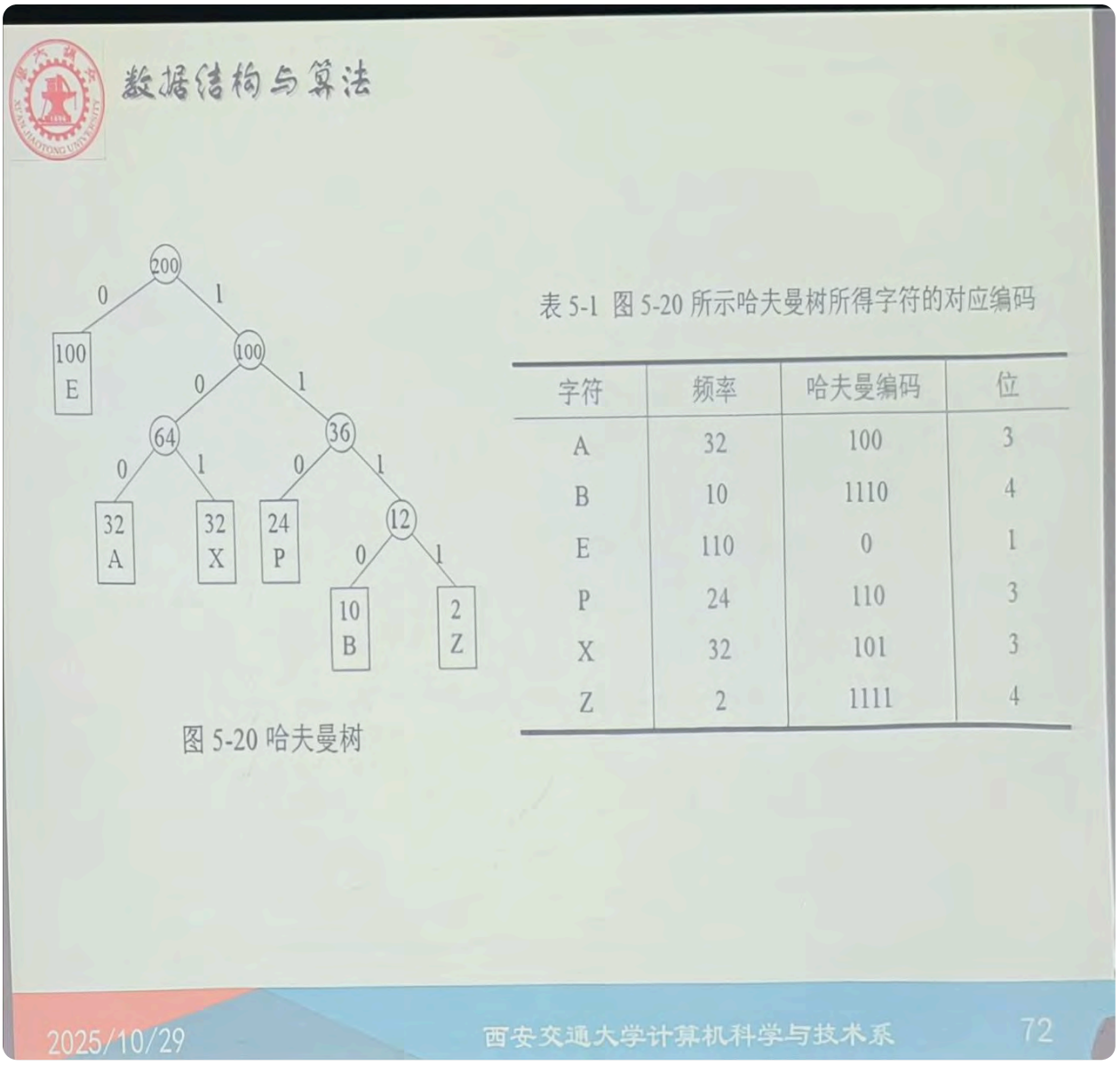
应用时，为了让哈夫曼树的构造唯一，方便逆推，可以在原来基础上加上限制：

1. 权值小的左子树，权值大的右子树
2. 权值相等，先出现的为左子树
3. 高度低的为左子树

5.4.3 哈夫曼树的应用

1. **编码/译码**

   哈夫曼编码是一种用于无损数据压缩的前缀编码。其核心思想是为出现频率高的字符分配较短的编码，为出现频率低的字符分配较长的编码。



   - **编码**：根据字符频率构造哈夫曼树，约定左分支为'0'，右分支为'1'，从根到各叶子结点的路径就构成了对应字符的编码。
   - **译码**：由于是前缀编码，解码时不会有歧义。从根结点开始，根据二进制码流的'0'或'1'遍历树，直到到达叶子结点，即完成一个字符的解码。

2. **加密**

   哈夫曼编码可以作为一种简单的加密方法。如果不知道字符与编码的对应关系（即哈夫曼树的结构），就无法解码。

3. **压缩**

   哈夫曼编码通过缩短高频字符的表示来减小文件体积。

4. **判定**

   在某些判定问题中，可以构造最优判定树，其结构和思想与哈夫曼树类似，用于最小化平均判定次数。

5.5 二叉树应用：BST（二叉查找树）

> 快速访问数据，建立索引

5.5.1 二叉查找树的定义

二叉查找树（Binary Search Tree, BST），又称二叉排序树，满足以下性质：

1. 若左子树非空，则左子树上所有结点的关键字值均小于根结点的关键字值。
2. 若右子树非空，则右子树上所有结点的关键字值均大于根结点的关键字值。
3. 左右子树本身也分别是二叉查找树。

特性：对二叉查找树进行中序遍历，可以得到一个递增的有序序列。

5.5.2 BST的查找

在BST中查找值为 `key` 的结点。

递归实现

```
BiTNode* BST_Search_Recursive(BiTree T, ElemType key) {
    if (!T) {
        return nullptr; // 查找失败
    }
    if (key == T->data) {
        return T; // 查找成功
    } else if (key < T->data) {
        return BST_Search_Recursive(T->lchild, key); // 在左子树中查找
    } else {
        return BST_Search_Recursive(T->rchild, key); // 在右子树中查找
    }
}
```

非递归实现

```
BiTNode* BST_Search_NonRecursive(BiTree T, ElemType key) {
    BiTree p = T;
    while (p) {
        if (key == p->data) {
            return p; // 查找成功
        } else if (key < p->data) {
            p = p->lchild;
        } else {
            p = p->rchild;
        }
    }
```

```
    return nullptr; // 查找失败
}
```

5.5.3 BST的插入

插入操作的核心是先查找到合适的插入位置，然后将新结点链接上去。

递归实现

```
bool BST_Insert_Recursive(BiTree& T, ElemType key) {
    if (!T) {
        T = new BiTNode(key); // 找到插入位置，创建新结点
        return true;
    }
    if (key == T->data) {
        return false; // 结点已存在，插入失败
    } else if (key < T->data) {
        return BST_Insert_Recursive(T->lchild, key);
    } else {
        return BST_Insert_Recursive(T->rchild, key);
    }
}
```

非递归实现

```
bool BST_Insert_NonRecursive(BiTree& T, ElemType key) {
    if (!T) {
        T = new BiTNode(key);
        return true;
    }
    BiTree p = T, parent = nullptr;
    while (p) {
        parent = p;
        if (key == p->data) {
            return false; // 结点已存在
        } else if (key < p->data) {
            p = p->lchild;
        } else {
            p = p->rchild;
        }
    }
    // p为NULL, parent为待插入结点的父结点
    BiTNode* newNode = new BiTNode(key);
```

```
    if (key < parent->data) {
        parent->lchild = newNode;
    } else {
        parent->rchild = newNode;
    }
    return true;
}
```

5.5.4 BST的构造

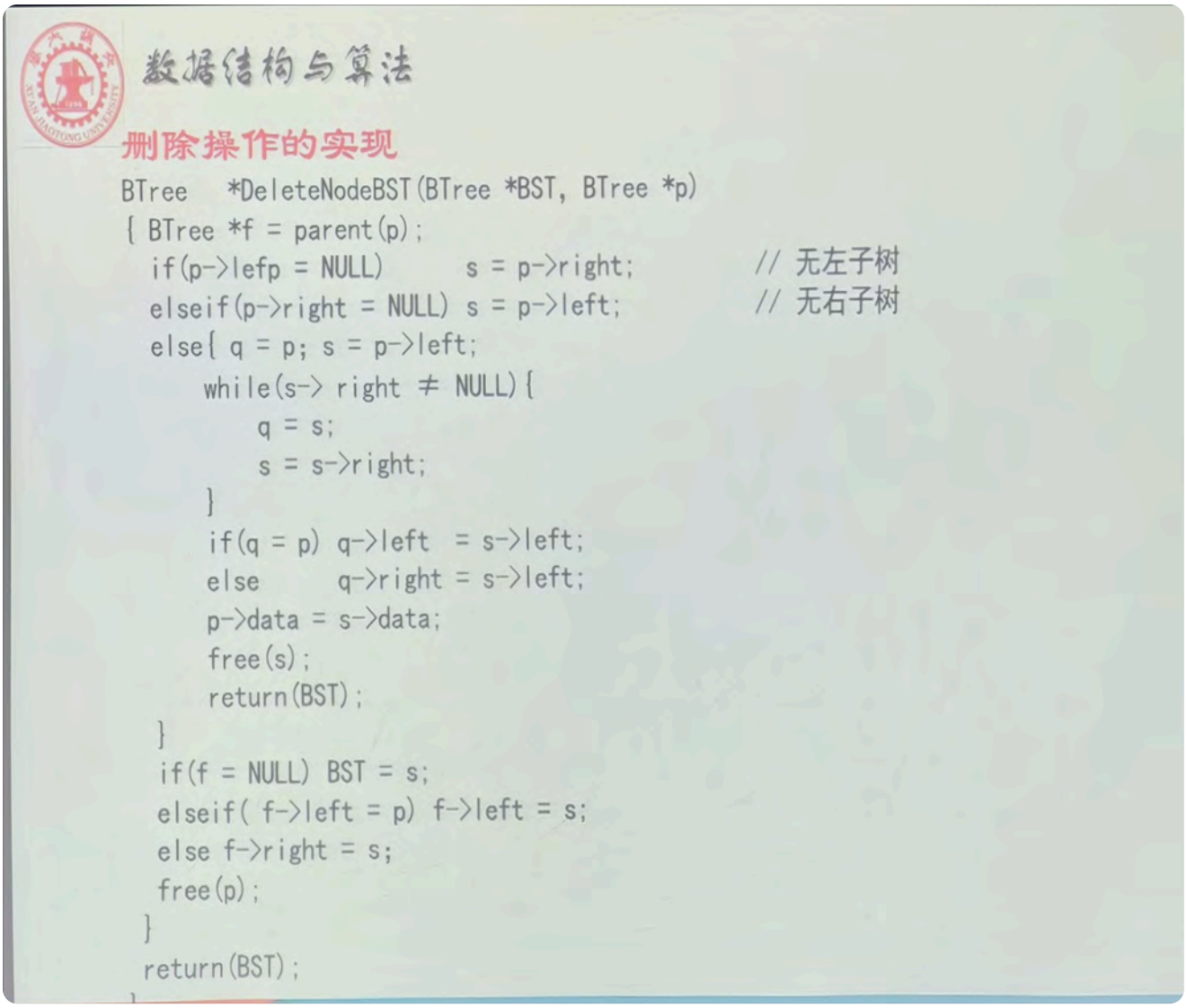
构造一棵二叉查找树的过程，就是将一个元素序列逐个插入到一棵初始为空的BST中。

```
void Create_BST(BiTree& T, ElemType arr[], int n) {
    T = nullptr;
    for (int i = 0; i < n; i++) {
        BST_Insert_NonRecursive(T, arr[i]);
    }
}
```

5.5.5 BST的删除

删除操作是BST中最复杂的操作，需要分情况讨论：

1. **被删除结点p是叶子结点**：直接删除，并将其父结点的相应指针置空。
2. **被删除结点p只有左子树或右子树**：让p的子树直接成为p父结点的子树。
3. **被删除结点p既有左子树又有右子树**：
   - 找到p的**直接前驱**（左子树中最大的结点）或**直接后继**（右子树中最小的结点）。
   - 用其直接前驱（或后继）的值替换p的值。
   - 将被替换的直接前驱（或后继）结点删除（该结点的删除必然属于情况1或2）。



附加空间为O(0)

非递归实现

```
bool BST_Delete(BiTree& T, ElemType key) {
    BiTNode *p = T, *parent = nullptr;
    // 1. 查找值为key的结点p及其父结点parent
    while (p && p->data != key) {
        parent = p;
        if (key < p->data) {
            p = p->lchild;
        } else {
            p = p->rchild;
        }
    }

    if (!p) {
        return false; // 树中无此结点，删除失败
    }
```

```
    // 2. 根据p的子树情况进行删除
    // 情况3: p有左右两棵子树
    if (p->lchild && p->rchild) {
        // 寻找p的直接后继s (右子树中最小的结点) 及其父结点sp
        BiTNode *s = p->rchild, *sp = p;
        while (s->lchild) {
            sp = s;
            s = s->lchild;
        }
        // 用后继s的值覆盖p的值
        p->data = s->data;
        // 现在问题转化为删除结点s (s必然是叶子或只有右孩子)
        p = s;
        parent = sp;
    }

    // 情况1 & 2: p是叶子结点或只有一个孩子
    BiTNode* child = nullptr;
    if (p->lchild) {
        child = p->lchild;
    } else {
        child = p->rchild;
    }

    if (!parent) { // 删除的是根结点
        T = child;
    } else if (parent->lchild == p) {
        parent->lchild = child;
    } else {
        parent->rchild = child;
    }

    delete p;
    return true;
}
```

> 除了替换方法完成删除操作之外，还可采用截枝操作实现删除操作。但这种方法可能会大幅改变树的形状，从而影响查找速度。

5.6 二叉树应用：平衡二叉树

5.6.1 定义

平衡二叉树（Balanced Binary Tree 或 AVL Tree）是一种自平衡的二叉查找树。它或者是一棵空树，或者其左右两棵子树的高度差的绝对值不超过1，并且左右两棵子树也都是一棵平衡二叉树。

为了衡量结点的平衡程度，引入了**平衡因子BF**的概念：

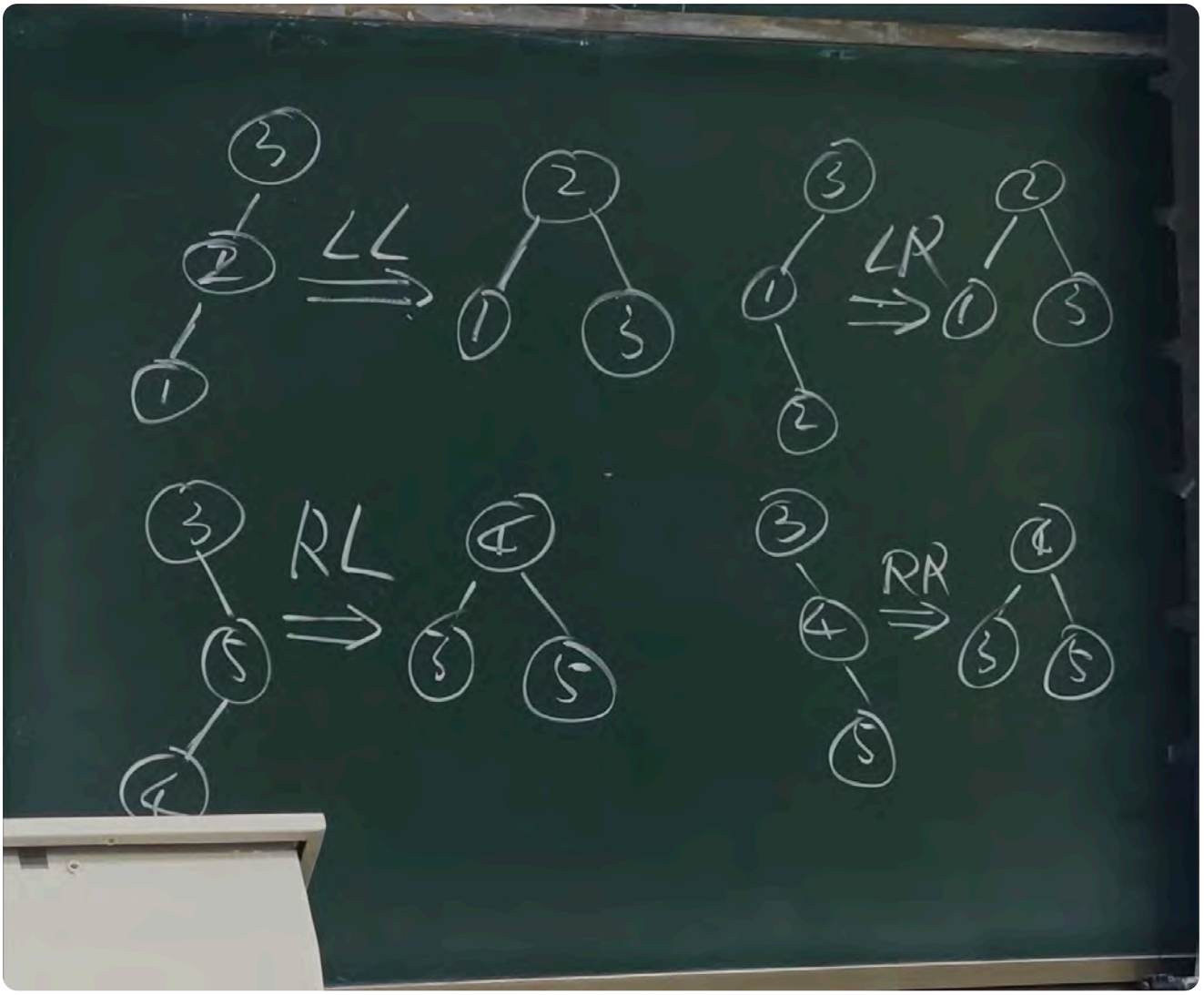
> $$BF(T) = h_L - h_R$$
>
> 其中，h_L 和 h_R 分别为结点T的左、右子树的高度。

对于一棵AVL树，其所有结点的平衡因子只能是 **-1、0 或 1**。当插入或删除一个结点导致某个结点的平衡因子绝对值大于1时，就需要通过**旋转**操作来恢复树的平衡。

> 结论：具有n个结点的AVL树，其深度最大值为$1.44\log_2(n)$，通常接近$\log_2(n)$。

5.6.2 四种平衡旋转法

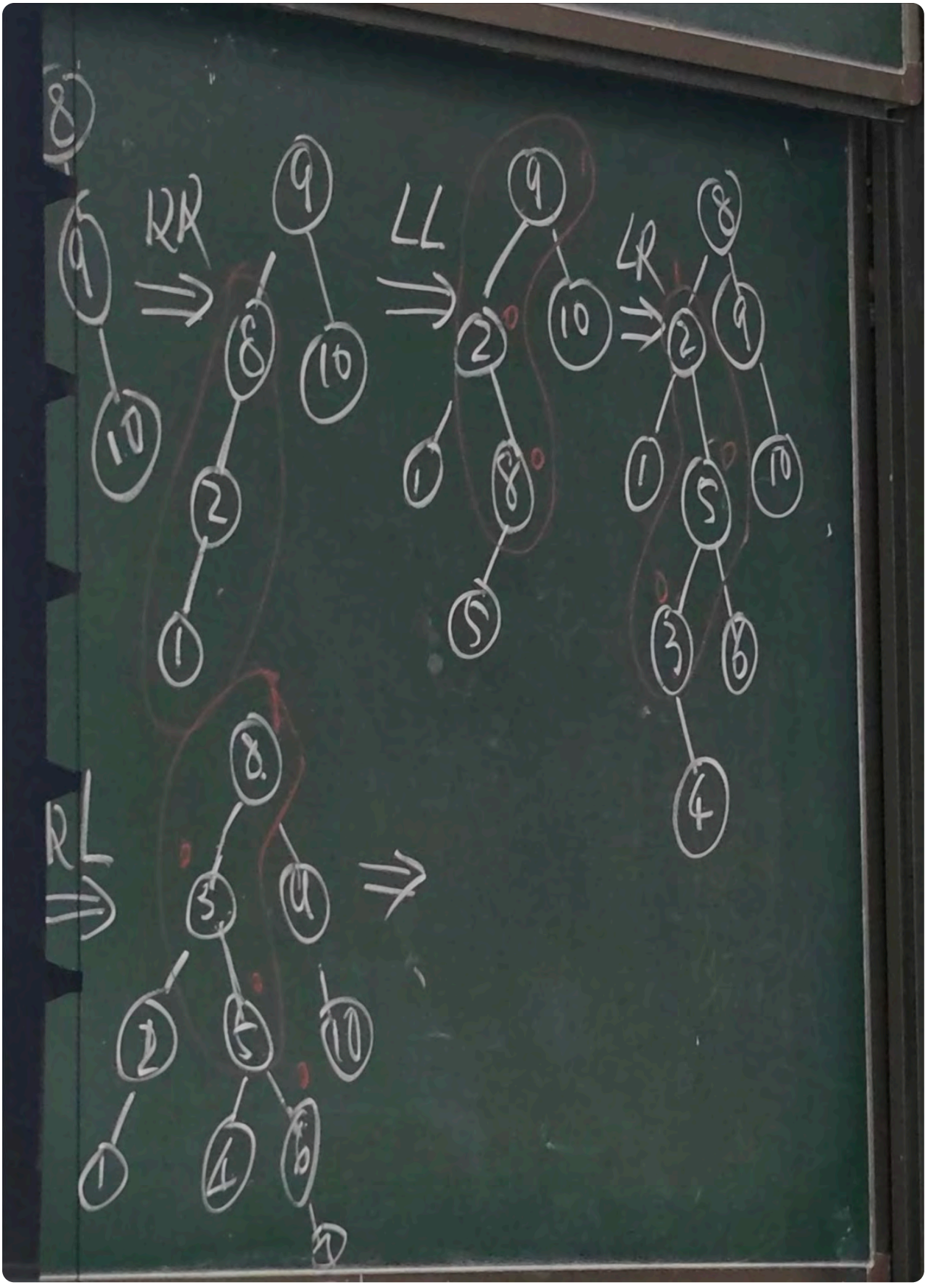
当向AVL树中插入或删除结点后，可能会破坏其平衡性。此时，需要从插入/删除位置开始，向上回溯检查路径上各结点的平衡因子。第一个平衡因子绝对值大于1的结点，就是**最小不平衡子树**的根。针对该子树，需要进行相应的旋转操作来恢复平衡。



- 按照结点关系可以分为LL型，LR型，RL型，RR型

5.6.3 AVL树的插入

1. 将S以BST树的插入方式插入到AVL中
2. 判断AVL是否平衡，如果是，则插入成功
3. 如果否，找到失去平衡的最小子树，判断旋转类型，做相应旋转



5.7 二叉树应用：堆与优先队列

5.7.1 优先队列

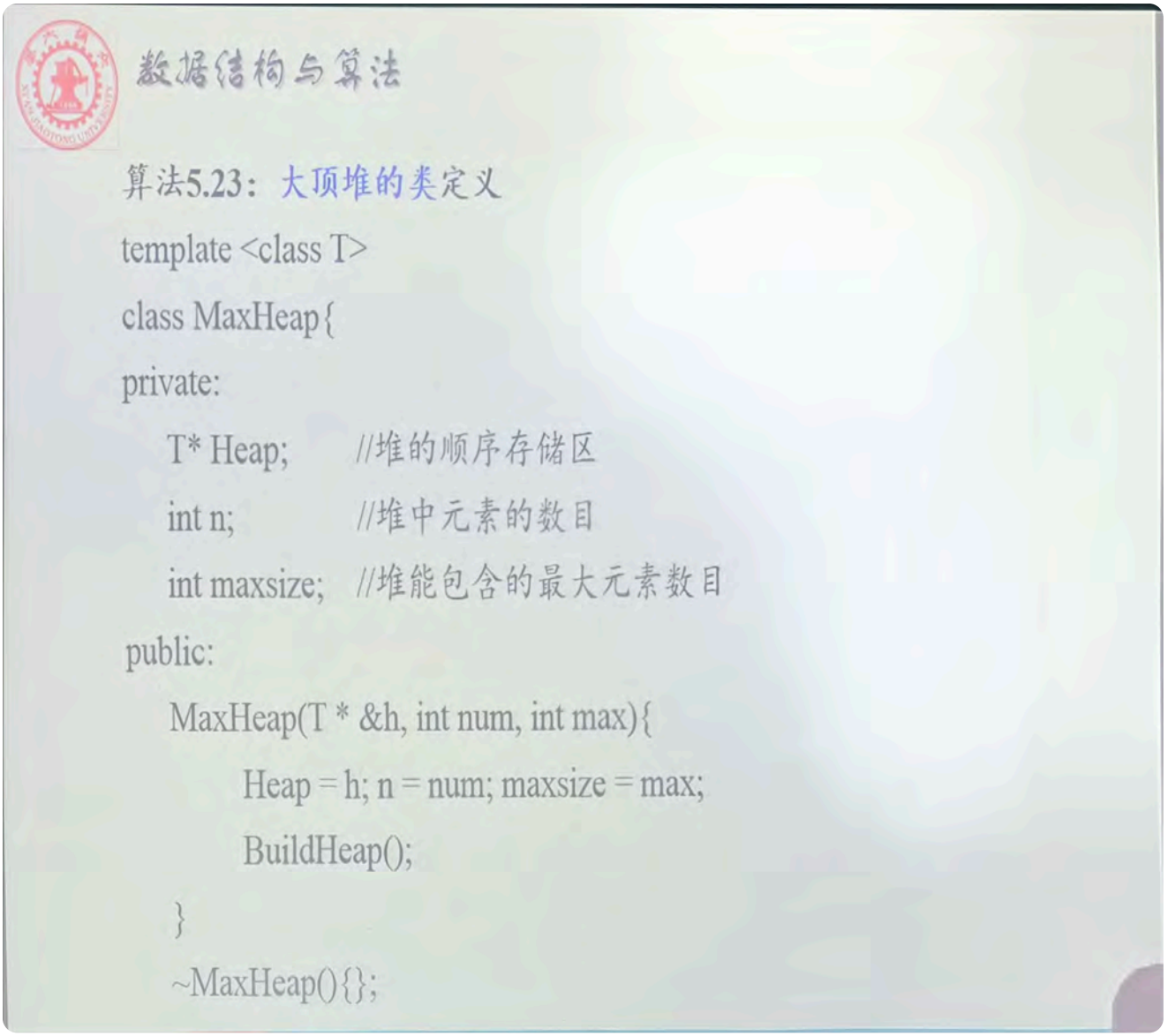
优先队列是一种特殊的抽象数据类型，类似于队列，但每个元素都有一个与之关联的"优先级"。元素出队（被处理）的顺序是根据它们的优先级，而不是它们入队的顺序。

优先队列可以用多种数据结构实现（如无序数组、有序数组、链表等），但使用**堆**来实现时效率最高。

5.7.2 堆的定义与性质

堆是一种特殊的基于树的数据结构，通常被看作一棵**完全二叉树**，并满足堆序性：

> **堆序性**：
>
> - **大顶堆**：任意结点的值都 **不小于** 其子结点的值。堆顶元素是整个堆的最大值。
> - **小顶堆**：任意结点的值都 **不大于** 其子结点的值。堆顶元素是整个堆的最小值。



性质：

1. 结构性：堆总是一棵完全二叉树，这使得它非常适合用**数组**进行顺序存储，可以节省空间并方便地进行父子结点定位。

2. 有序性：任何一个以结点为根的子树，其本身也是一个堆。
3. 存储：对于数组中下标为 `i` 的结点：
   - 其父结点下标为 $\lfloor (i-1)/2 \rfloor$ (当 i>0 时)。
   - 其左孩子下标为 $2i+1$。
   - 其右孩子下标为 $2i+2$。

5.7.3 堆的基本操作

以大顶堆为例：

1. 堆的插入

思路：将新元素放在堆末尾（数组末尾），然后通过与其父结点比较和交换，不断向上调整，直到满足堆序性。

步骤：

1. 将新元素添加到数组的末尾。
2. 比较新元素与其父结点，若新元素大于父结点，则交换两者位置。
3. 重复步骤2，直到新元素小于或等于其父结点，或已到达堆顶。

2. 堆的删除（删除堆顶）

思路：删除堆顶元素后，为保持完全二叉树的结构，将堆末尾的元素移到堆顶，然后通过与其子结点比较和交换，不断向下调整，直到满足堆序性。

步骤：

1. 用数组最后一个元素替换堆顶元素。
2. 将数组长度减一。
3. 从堆顶开始，比较当前结点与其左右孩子。
4. 若当前结点小于其孩子中的较大者，则与之交换。
5. 重复步骤4，直到当前结点大于其所有子结点，或已成为叶子结点。

3. 堆的建立（筛选法）

思路：将一个无序序列构建成一个堆。
最高效的方法是从最后一个非叶子结点开始，自下而上，自右向左，对每个结点执行向下调整操作。

步骤：

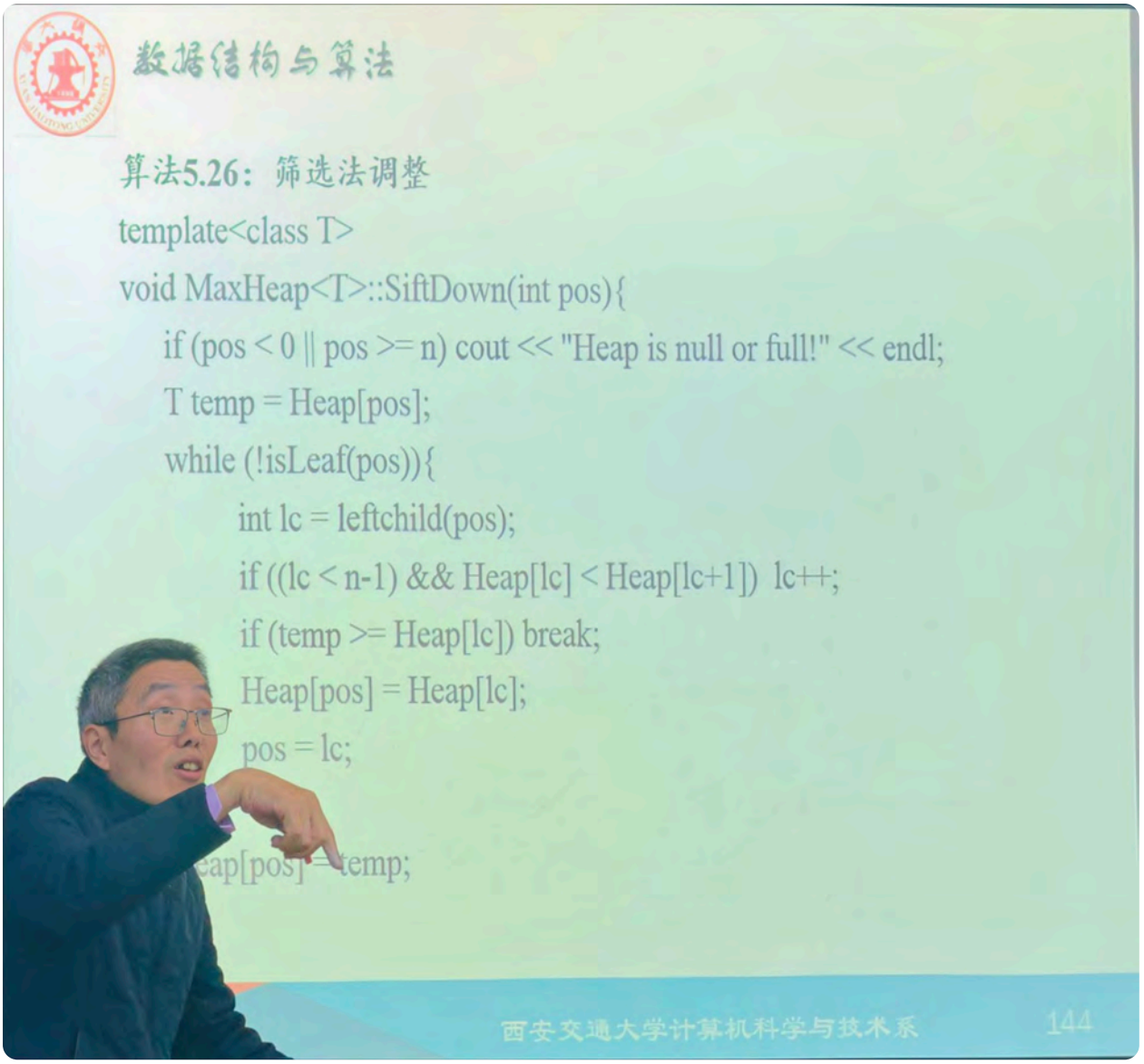
1. 将给定序列看作一个完全二叉树。
2. 找到最后一个非叶子结点，其下标为 $\lfloor n/2 \rfloor - 1$ (n为元素个数)。
3. 从该结点开始，直到根结点（下标为0），依次对每个结点执行“向下调整”操作。

5.7.4 堆的应用

1. 堆排序

堆排序是一种高效的原地排序算法，利用堆的性质进行排序。
过程：

1. **建堆**：将待排序序列构建成一个大顶堆。此时，堆顶元素为序列的最大值。
2. **排序**：
   a. 将堆顶元素与堆末尾元素交换。此时，最大值已"归位"到数组末尾。
   b. 将堆的规模减1，并对新的堆顶元素执行"向下调整"，以恢复大顶堆。
   c. 重复上述操作，直到堆中只剩一个元素。



- 留意这里break的 用法

2. 优先队列的实现

堆是实现优先队列的最优数据结构之一，操作对应关系如下：

- **插入元素**：对应堆的插入操作，时间复杂度 $O(\log n)$。

- **获取最大/小值**：直接获取堆顶元素，时间复杂度 O(1)。
- **删除最大/小值**：对应堆的删除操作，时间复杂度 O($\log n$)。

5.8 树与森林

5.8.1 树的存储结构

双亲表示法

- 在树中，除了根结点没有双亲结点外，其余的结点都有**唯一**的双亲结点。
- 采用一组连续空间（数组）存储每个结点，同时在每个结点中附设一个指示器，指示其双亲结点在数组中的位置。根结点的双亲位置通常设为-1。

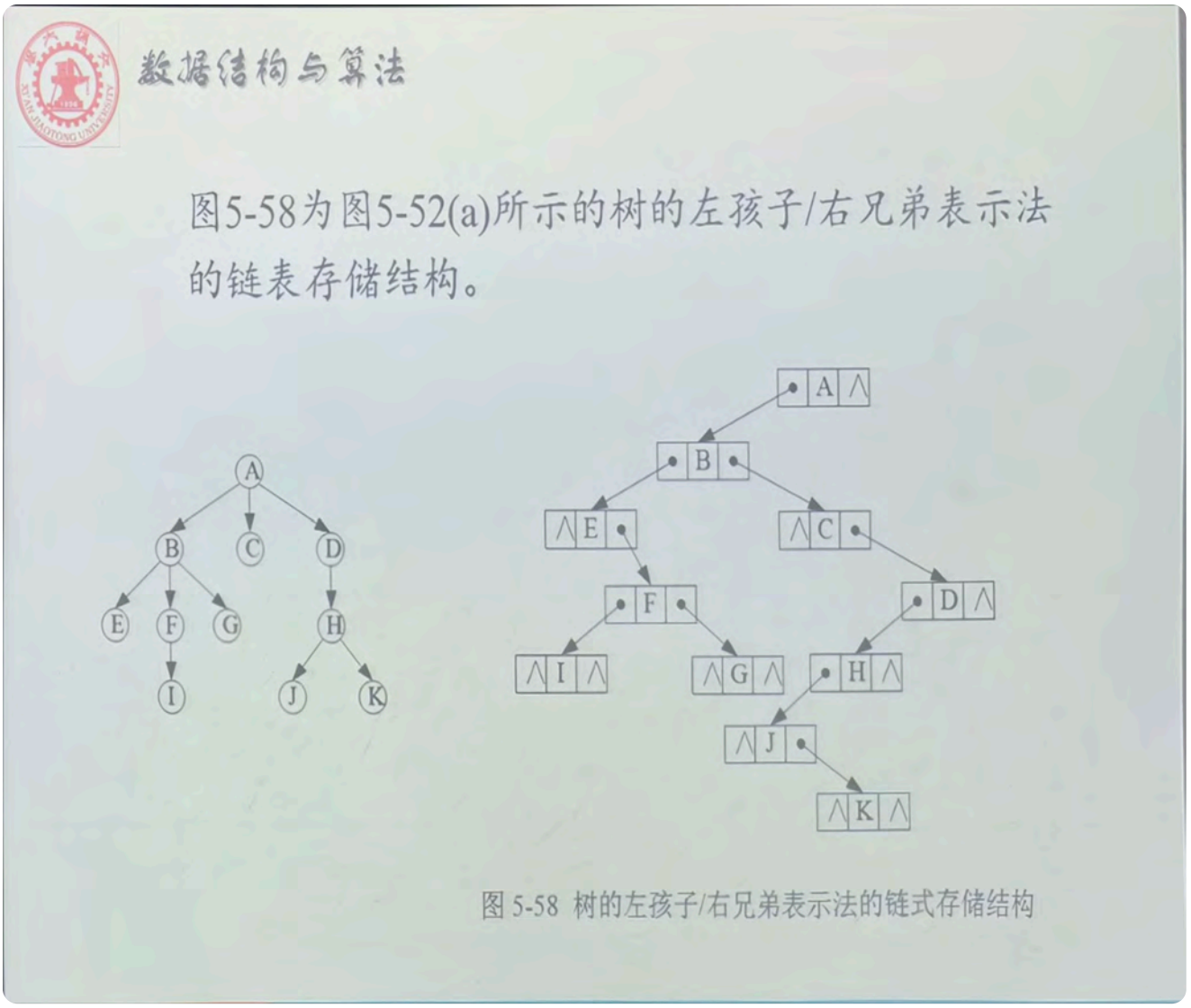
data	parent

- **优点**：查找指定结点的双亲非常方便，时间复杂度为O(1)。
- **缺点**：查找指定结点的孩子时，需要遍历整个结构。

孩子表示法

- 将每个结点的孩子结点排列起来，看成一个线性表，用链表存储。对于n个结点的树，就有n个孩子链表。每个结点在数组中占一个位置，该位置存放结点的数据和一个指向其孩子链表头结点的指针。

孩子兄弟表示法



5.8.2 树、森林与二叉树的转换

树、森林和二叉树之间存在着紧密的联系，可以进行相互转换。这种转换的核心是**孩子兄弟表示法**。

树转换为二叉树

任何一棵树都可以通过"左孩子右兄弟"的规则转换为唯一的一棵二叉树。

转换规则：

1. **连接孩子**：将树中每个结点的第一个孩子作为其在二叉树中的左孩子。
2. **连接兄弟**：将树中每个结点的右邻兄弟作为其在二叉树中的右孩子。

- 转换后，原树中结点的第一个孩子成为二叉树中的左孩子。
- 原树中结点的兄弟成为二叉树中的右孩子。
- 转换后得到的二叉树，其**根结点一定没有右子树**。

森林转换为二叉树

任何森林也可以转换为唯一的一棵二叉树。

转换步骤：

1. **树转二叉树**：将森林中的每一棵树分别转换为二叉树。
2. **连接森林**：将第一棵二叉树的根作为转换后二叉树的根。然后，将第二棵二叉树的根作为第一棵二叉树根的右孩子；将第三棵二叉树的根作为第二棵二叉树根的右孩子，以此类推，直到所有树都连接起来。

二叉树转换为森林或树

这是上述过程的逆操作。

转换规则：

1. **识别根**：若一个结点是其父结点的左孩子，则它和它的所有右兄弟（沿着 `rchild` 指针链）是其父结点在树中的孩子。
2. **拆分**：从根结点开始，沿其右指针链访问到的所有结点，都是森林中各棵树的根。
3. **重建**：对每个根结点，其左子树递归地转换为其子树森林，重复此过程，直到所有结点都被处理。

5.8.3 树与森林的遍历

树的遍历

树主要有两种深度优先的遍历方式：

1. **先根遍历** ：先访问根结点，然后依次从左到右先根遍历根的各棵子树。
2. **后根遍历**：先依次从左到右后根遍历根的各棵子树，然后访问根结点。

森林的遍历

森林的遍历方式与树类似：

1. **先序遍历**：
   a. 访问森林中第一棵树的根结点。
   b. 先序遍历第一棵树的子树森林。
   c. 先序遍历除去第一棵树后剩余的森林。
2. **中序遍历**：
   a. 中序遍历第一棵树的子树森林。
   b. 访问森林中第一棵树的根结点。
   c. 中序遍历除去第一棵树后剩余的森林。

遍历的等价关系

树/森林的遍历与其转换后的二叉树遍历存在着重要的等价关系：

- 树/森林的**先序遍历** <==> 对应二叉树的**先序遍历**。

- 树的**后根遍历** <==> 对应二叉树的**中序遍历**。
- 森林的**中序遍历** <==> 对应二叉树的**中序遍历**。
- 树/森林的**层次遍历** <==> 对应二叉树的**层次遍历**。

上一章 第4章 扩展线性表—数组与广义表
下一章 第6章 图

附录：

课后需要做的

1. 定义
2. floyd算法
3. Dijstna
4. 最小生成树

C++ 实现

这里提供了笔记中讨论的二叉树主要操作的C++实现。

```
#include <iostream>
#include <queue>

// 定义数据元素类型
using ElemType = int;

// 1. 定义二叉树结点结构 (三叉链表)
struct BiTNode {
    ElemType data;
    BiTNode *lchild, *rchild;
    BiTNode *parent; // 指向父结点的指针

    BiTNode(ElemType val) : data(val), lchild(nullptr), rchild(nullptr), parent(nullptr) {}
};

using BiTree = BiTNode*;

// 声明辅助函数
void DestroyTree(BiTree& T);

/**
 * @brief 初始化一棵空二叉树
 * Initial(BT)
 */
```

```
void Initial(BiTree& T) {
    T = nullptr;
}

/**
 * @brief 判断二叉树是否为空
 * Empty(BT)
 */
bool Empty(BiTree T) {
    return T == nullptr;
}

/**
 * @brief 构造二叉树 (根据带-1作为空标记的先序序列)
 * Build(BT)
 * 例如，输入: 1 2 4 -1 -1 5 -1 -1 3 -1 -1
 */
void Build(BiTree& T, BiTNode* p = nullptr) {
    ElemType val;
    std::cin >> val;
    if (val == -1) {
        T = nullptr;
    } else {
        T = new BiTNode(val);
        T->parent = p;
        Build(T->lchild, T);
        Build(T->rchild, T);
    }
}

/**
 * @brief 获取结点的父结点
 * Parent(BT, p)
 */
BiTNode* Parent(BiTNode* p) {
    return p ? p->parent : nullptr;
}

/**
 * @brief 获取结点的左孩子
 * L_Child(BT, p)
 */
BiTNode* L_Child(BiTNode* p) {
```

```
    return p ? p->lchild : nullptr;
}

/**
 * @brief 获取结点的右孩子
 * R_Child(BT, p)
 */
BiTNode* R_Child(BiTNode* p) {
    return p ? p->rchild : nullptr;
}

/**
 * @brief 插入新结点作为p的左孩子
 * Insert(BT, p)
 * @return 成功返回true, 失败(p为空或已有左孩子)返回false
 */
bool InsertLeftChild(BiTNode* p, ElemType val) {
    if (!p || p->lchild) {
        return false;
    }
    BiTNode* newNode = new BiTNode(val);
    newNode->parent = p;
    p->lchild = newNode;
    return true;
}

/**
 * @brief 删除p的左子树
 * Delete(BT, p)
 */
void DeleteLeftSubtree(BiTNode* p) {
    if (p) {
        DestroyTree(p->lchild);
    }
}

/**
 * @brief 计算度为1或2的结点数 (即非叶子结点数)
 * Leaf(BT) -> 对应"求度为1或2的结点数"
 */
int CountInternalNodes(BiTree T) {
    if (!T || (!T->lchild && !T->rchild)) {
        return 0; // 空树或叶子结点
```

```
    }
    // 是内部结点，计数1，并递归计算左右子树
    return 1 + CountInternalNodes(T->lchild) + CountInternalNodes(T->rchild);
}

// 辅助函数：递归释放树的内存
void DestroyTree(BiTree& T) {
    if (T) {
        DestroyTree(T->lchild);
        DestroyTree(T->rchild);
        delete T;
        T = nullptr;
    }
}
```