第8章 查找算法

#数据结构与算法

本章要求：

1. 理解查找的基本概念。
2. 掌握**静态查找表**的常见算法（顺序查找、折半查找）。
3. 熟悉并能运用**散列技术**（哈希表）解决实际问题。
4. 熟悉**线性索引**以及**树形索引**（如B树）的原理。

8.1 查找的基本概念

> **查找又称为检索**，确定一个已经给出的数据是否出现在某个数据元素集合中。

相关概念：

1. key：数据元素的标识，也称为**关键字**或**键值**，用于唯一确定数据元素。
2. 查找表：指同一类型的数据元素（或记录）组成的集合。以关键字确定数据元素。
   - **静态查找表**：数据元素相对固定，不涉及插入和删除操作（如顺序表、散列表）。
   - **动态查找表**：数据元素动态变化，涉及插入和删除操作（如二叉排序树、B树）。
3. 查找：在查找表中确定某个关键字与待查记录相同的数据元素的过程。
4. 平均查找长度（ASL）：即平均比较次数，用来衡量查找算法的优劣。

$$ASL = \sum_{i=1}^{n} P_i C_i$$

其中，P_i 为第 i 个记录的出现概率，C_i 为查找该记录的平均比较次数。

8.2 静态查找表

静态查找表大多以顺序存储的形式实现，本节主要讨论顺序查找和折半查找。

8.2.1 顺序表的查找（顺序查找）

> 顺序表的查找又称**顺序查找**，待查元素存放在数组或者链表中。

顺序查找的思想很简单，就是按照元素存放顺序来遍历顺序表，注意要考虑越界的情况。

性能分析：假设顺序表中有 n 个元素，查找第 i 个元素平均需要比较 i 次（通常从数组下标1开始算起）。可以看出，这种查找方法效率非常低下。

$$ASL = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

8.2.2 折半查找

> 折半查找又称二分法查找，只适用于对**有序顺序表**进行查找的情况（链式结构不适合）

折半查找每次与下标为 $mid = \frac{low+high}{2}$ 的元素值比较，运用递归实现。

折半查找可以描述为二叉树的形式，称为**折半查找判定树**，则最大比较次数为判定树的深度。

性能分析：假定每个记录的查找概率相等，则第 j 层最多有 2^{j-1} 个结点，每个结点的比较次数为 j，设树高 $h = \lceil \log_2(n+1) \rceil$。

$$ASL = \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1} \approx \log_2 n$$

折半查找的平均时间复杂度为 $O(\log_2 n)$。

（拓展）NEXT函数：

next 函数（或 next 数组）是**KMP字符串匹配算法**的导航。当匹配中途出错时，它能告诉我们模式串应该"智能"地向右移动多远，而不是傻傻地只移动一位，从而大大提高匹配效率。

它的核心是计算模式串自身的最长公共前后缀长度。

- **前缀**：指不包含最后一个字符的所有头部子串。
- **后缀**：指不包含第一个字符的所有尾部子串。

以模式串 $p =$"ababa" 为例，我们可以这样计算它的 `next` 数组：

j	子串 `p[]`	前后缀中最长的公共部分	长度 `next[j]`
0	"a"	（无）	0
1	"ab"	（无）	0
2	"aba"	"a"	1
3	"abab"	"ab"	2
4	"ababa"	"aba"	3

因此，模式串 `"ababa"` 的 `next` 数组就是 `[0, 0, 1, 2, 3]`。当匹配到 `p[j]` 失败时，我们就可以参考 `next[j-1]` 的值来决定下一步怎么移动。

更深入地了解 KMP 算法

> 从头到尾彻底理解KMP - v_JULY_v (CSDN)
> 帮你把KMP算法学个通透 - 代码随想录
> 如何更好地理解和掌握 KMP 算法？ - 知乎

8.3 Hash表

> **Hash表**是一种非关键字的查找方法，可以像数组检索一样实现随机查找。将**关键字映射到表中的位置**来访问记录的过程称为**哈希**（散列）。

哈希表使用哈希函数将关键字映射到非负整数，哈希函数既是建表的函数也是哈希查找的函数。

$$Address = H(key)$$

8.3.1 哈希函数的常用构建方法

几点要求：

1. 尽可能分布均匀，减少冲突
2. 计算要简单和快速
3. 定义域要覆盖需要存储的全部关键字
   实际建立哈希表需要的**考虑因素**：

- 计算哈希函数所需的时间；
- 关键字的长度；
- 哈希表空间的大小；
- 关键字的分布情况；
- 记录的查找频率。

8.3.1.1 直接定址法

> 取关键字的某个线性函数值作为哈希地址。
>
> $$Hash(key) = a \times key + b$$
>
> **缺点**：空间复杂度高：对于稀疏图，会浪费大量存储空间，空间复杂度为O(n^2)。（这是顺序存储的典型特征）

- 一对一的映射不会冲突，但是消耗空间太多，一般不使用。

8.3.1.2 除留余数法

> 取关键字被某个不大于哈希表表长 m 的数 p 除后所得的余数作为哈希地址。

$$Hash(key) = key \pmod{p}, \quad (p \leq m)$$

p 的选择很关键，一般取**小于等于哈希表长的最大素数**。

8.3.1.3 平方取中法

> 通过计算关键字的**平方值**来扩大差异，然后取中间的几位数作为哈希表的索引。

8.3.1.4 随机数法

> 运用Random伪随机函数，得到0~(m-1)的值

8.3.1.5 截断法

> 取关键字的一部分作为哈希地址。

例如，手机号后四位。
这种方法简单，但如果关键字的某一部分**分布不均匀**，容易产生冲突。

8.3.1.6 折叠法

> 将关键字分割成**位数相同**的几部分（最后一部分位数可以不同），然后将这几部分相加（可舍去进位）作为哈希地址。
> ①**移位法**：把各部分的最后一位对齐相加；
> ②**分界法**：各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果作为哈希地址。

适合长关键字的情况。例如，关键字为 `9876543210`，哈希表长为三位数，可以分为 `987`、`654`、`321`、`0`，叠加和为 `987+654+321+0 = 1962`，舍去进位后得到 `962`。

8.3.1.7 查字典法

> 预先建立一张“字典”，将关键字和对应的哈希地址存储起来。查找时直接从字典中获取地址。
> 这种方法速度快，不会产生冲突，但只适用于关键字**集合较小且固定**的情况。

8.3.2 处理冲突的方法

冲突解决技术可以分为两类：
开放哈希法：冲突元素不在哈希表中存放，而是存放到哈希表之外的空间中去（如链地址法）；
封闭哈希法（又称**开放定址法**）：允许冲突元素存放到哈希表空间中，即放到哈希表的其他空闲区域中（如线性探测法）。

8.3.2.1 链地址法(拉链法)

将所有哈希地址相同的记录链接在同一条单链表中。哈希表本身只存放每个链表的头指针。

优点：实现简单，非同义词不会冲突，删除结点容易，装填因子 α 可以大于1。
缺点：需要额外的存储空间来存放指针。

- **性能**：查找成功的平均查找长度 $ASL_{成功} \approx 1 + \frac{\alpha}{2}$，其中 α 是装填因子（表中记录数/哈希表长）。

8.3.2.2 公共溢出区法

建立一个公共的溢出区，所有发生冲突的记录都按发生时间顺序存放在这个区域。查找时，若哈希地址对应位置的元素不是目标，则去溢出区进行顺序查找。

优点：处理冲突的过程与开放定址法相比，更为简单。
缺点：查找溢出区时相当于顺序查找，如果冲突记录多，效率会很低。

8.3.2.3 线性探测法

当发生冲突时，从冲突位置开始，依次探测下一个地址，直到找到一个空闲单元。

$$H_i = (H(key) + i) \pmod{m}, \quad (i = 1, 2, \ldots, m-1)$$

缺点：容易产生**淤积**现象，即冲突的记录会连成一片，影响查找效率。

8.3.2.4 平方探测法

当发生冲突时，探测序列的地址增量为 $1^2, -1^2, 2^2, -2^2, \ldots$。

$$H_i = (H(key) \pm i^2) \pmod{m}, \quad (i = 1, 2, \ldots, (m-1)/2)$$

优点：可以有效避免淤积现象。
缺点：不能探测到哈希表上的所有单元，但至少能探测到一半单元。

8.3.2.5 双重哈希法

使用两个哈希函数，当第一个哈希函数 H_1 发生冲突时，使用第二个哈希函数 H_2 来计算步长。

$$H_i = (H_1(key) + i \cdot H_2(key)) \pmod{m}, \quad (i = 1, 2, \ldots, m-1)$$

优点：是解决冲突的较好方法之一，能有效避免聚集现象。
要求：$H_2(key)$ 的值必须与 m 互质，才能保证探测到所有单元。

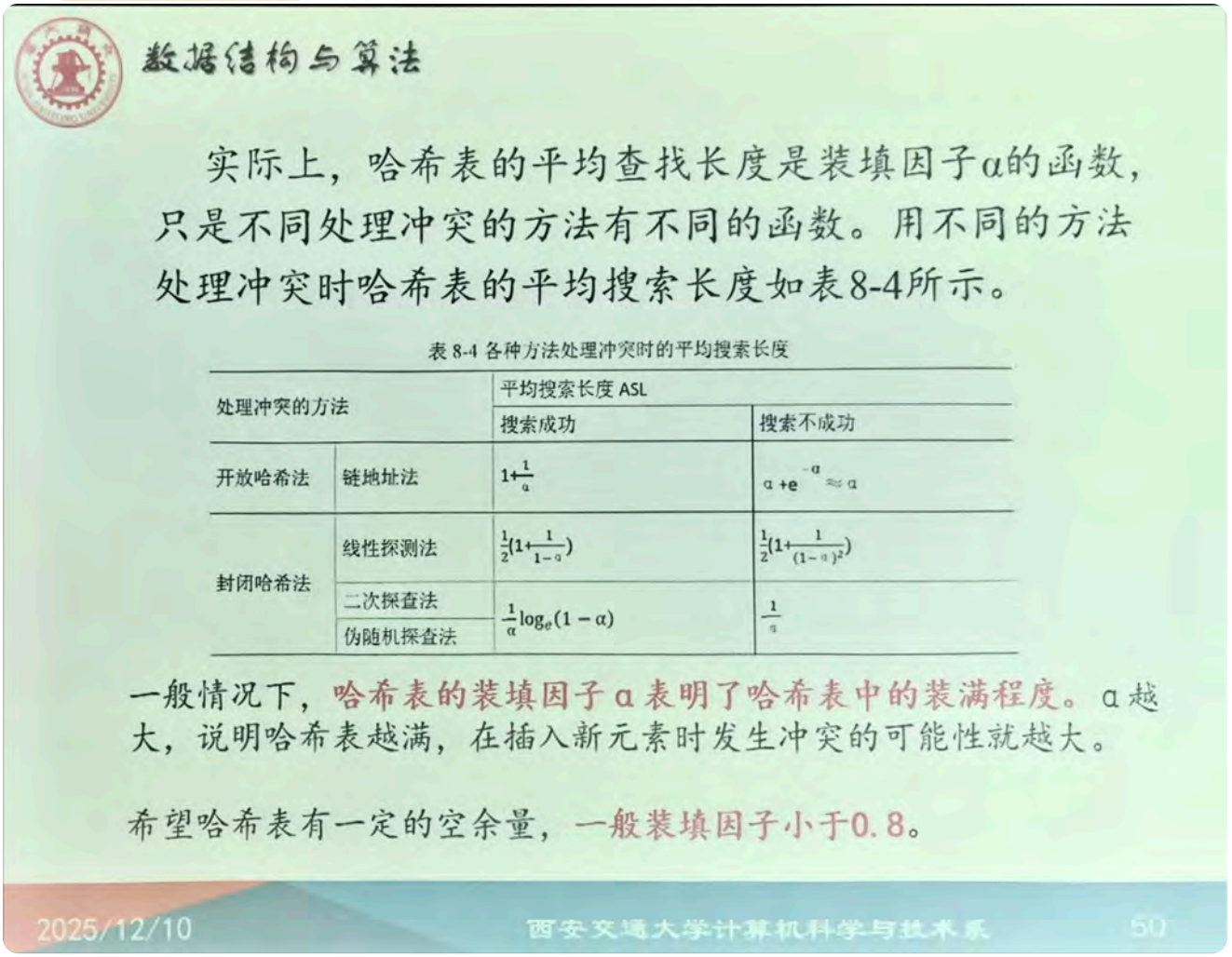
8.3.2.6 树形查找法

树形查找法是对顺序查找法的改进，依靠BST或AVL实现。

- 在这种方法中，每个哈希桶不再简单地存储一个链表，而是存储一个**自平衡二叉查找树**（如AVL树或红黑树）。当多个关键字映射到同一个哈希地址时，它们会被插入到该地址对应的二叉查找树中。
- 当哈希冲突严重导致某个桶内的元素非常多时，将链表转换为树形结构可以有效将查找、插入、删除操作的时间复杂度从 $O(k)$ 降低到 $O(\log k)$，其中 k 是该桶中的元素数量。

哈希表性能分析

使用平均搜索长度ASL来衡量哈希方法的搜索性能（**考试必会内容**）



（注：这里链地址法的搜索成功ASL应该为$1+\frac{\alpha}{2}$）

如何提高查找的效率？

1. 优化查找循环的条件
2. 利用前面的运算结果（如KMP）
3. 数据预处理，检查关键字的出现概率
4. 解决问题的方法

8.3.3 哈希表的实现

数据结构与算法

8.3.3 哈希表的实现

算法8.6：哈希表的类的声明以及部分函数实现

```
struct Element {            //记录的定义
    KeyType key;            //关键字
    field otherdata;        //其他域
};
class HashList{             //用线性探查法处理冲突时哈希表类型的定义
private:
    Element *HT;            //哈希表存储数组
    int  CurrentSize, MaxSize;      //当前哈希地址数及最大地址数
public:
    HashList(int MaxSz=DefaultSize); //构造函数
    ~HashList(){ delete [] HT; }  //析构函数
    void ClearHashList();          //清空哈希表的函数
    bool Insert(Element item);    //哈希表HT中插入一个元素item
    int Search(KeyType x);     //从哈希表HT中查找元素，返回该元素的下标位置
    bool Delete(KeyType x);     //从哈希表HT中删除元素
    void Create(int num);          //建立哈希表HT
};
```

8.4 线性索引

索引结构通常由**数据表**和**索引表**两部分组成。

- 数据表用于存储数据元素信息
- 索引表用来记录关键字与记录存储地址之间的对照表。索引表中的每个元素称为**索引项**，索引项的一般形式为<关键字，指针>，其中指针指向数据表中的包含该关键字的完整记录。
- 记录的存储可以是无序的，但索引项是有序的。
- 以索引结构组织的查找表称为**索引查找表**。

8.4.1 线性索引

线性索引的**索引文件**被组织成一组简单的<关键字，指针>对的序列。在索引文件中，按照**关键字的顺序**进行排序，指针指向存储在磁盘上的文件记录的起始位置或者主索引中关键字的起始位置。可用**顺序查找**或**折半查找**实现记录的定位。

缺点：假如存在大量的数据记录，那么可能由于索引文件太大而不能存储到内存当中。这样在检索过程中，需要多次访问磁盘，降低检索效率。

注：可以通过建立二级索引来改善该问题

8.4.2分块查找

步骤：

1. 将ST表分成若干块
2. 将分块中的**最大**或者**最小**取出
3. 取出的元素组织成另一个顺序表Index
4. 那么Index中的key是有序的

8.5 树形索引

B树是目前使用最为广泛的索引方法，树结构更新之后仍能自动保持平衡，适用于按块存储。

8.5.1 2-3树

> 2-3树是B树的特殊情况，即3阶B树

定义：

1. 每个结点上的子树个数可以是2-3个。每个结点包含一个或两个关键字的值；
2. 每个内结点或有一个关键字两个子树，或有两个关键字三个子树；
3. 所有的叶子结点在2-3树的同一层出现。即高度是一样的。

数据结构与算法

2-3树更新代价更小。具有N个关键字的2-3树的高度比相应的二叉查找树的高度要低很多。

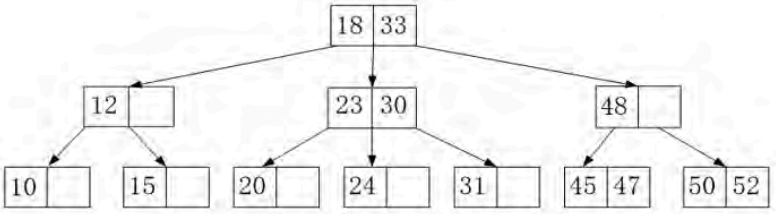


图 8-10 2-3 树

在2-3树中：

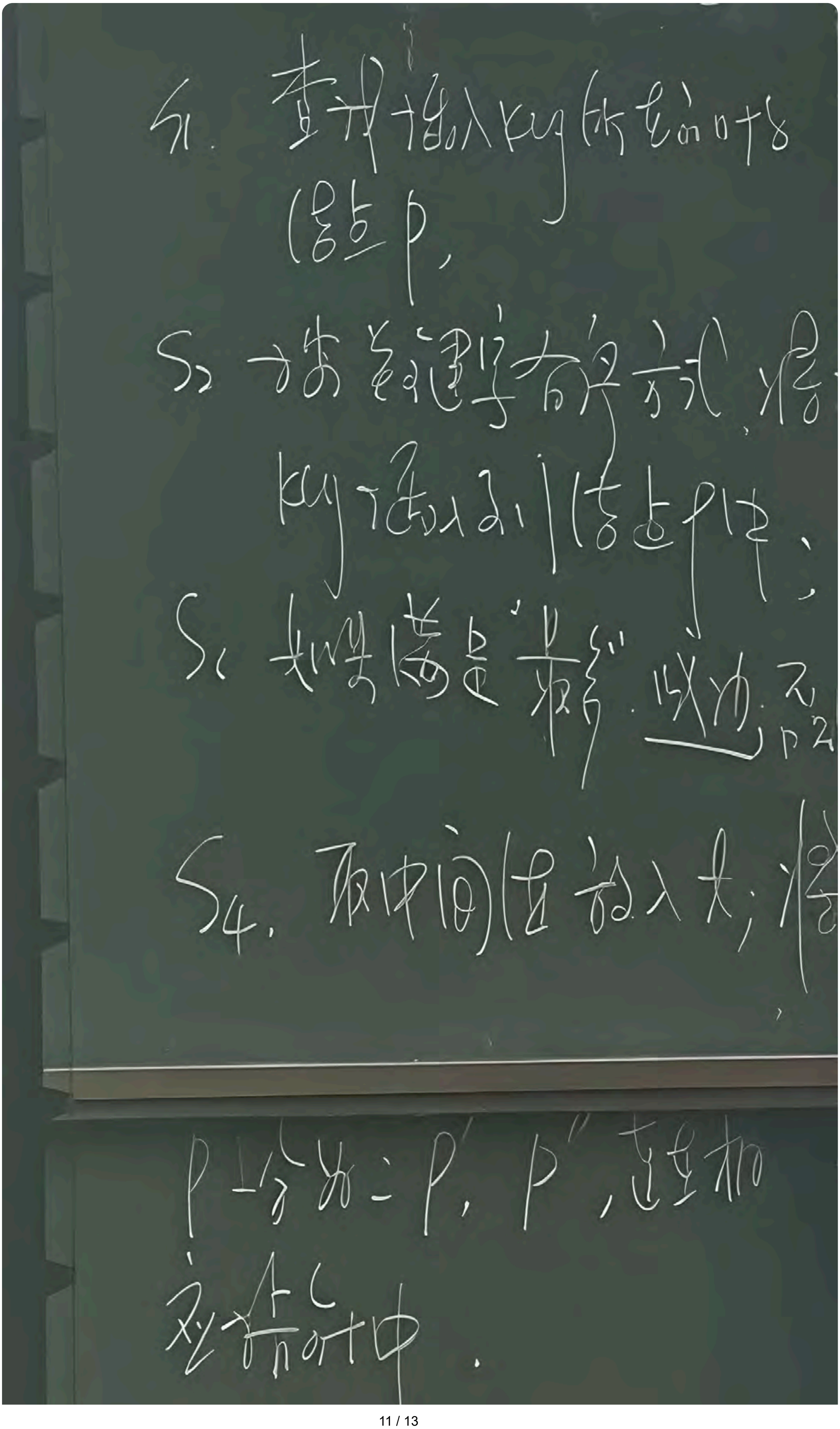
1）左边孩子中的结点值小于父结点中第一个关键字的值；
2）中间孩子中的结点值大于父结点中第一个关键字的值，且小于第二个关键字的值；
3）右边孩子中的结点值大于父结点中第二个关键字的值。

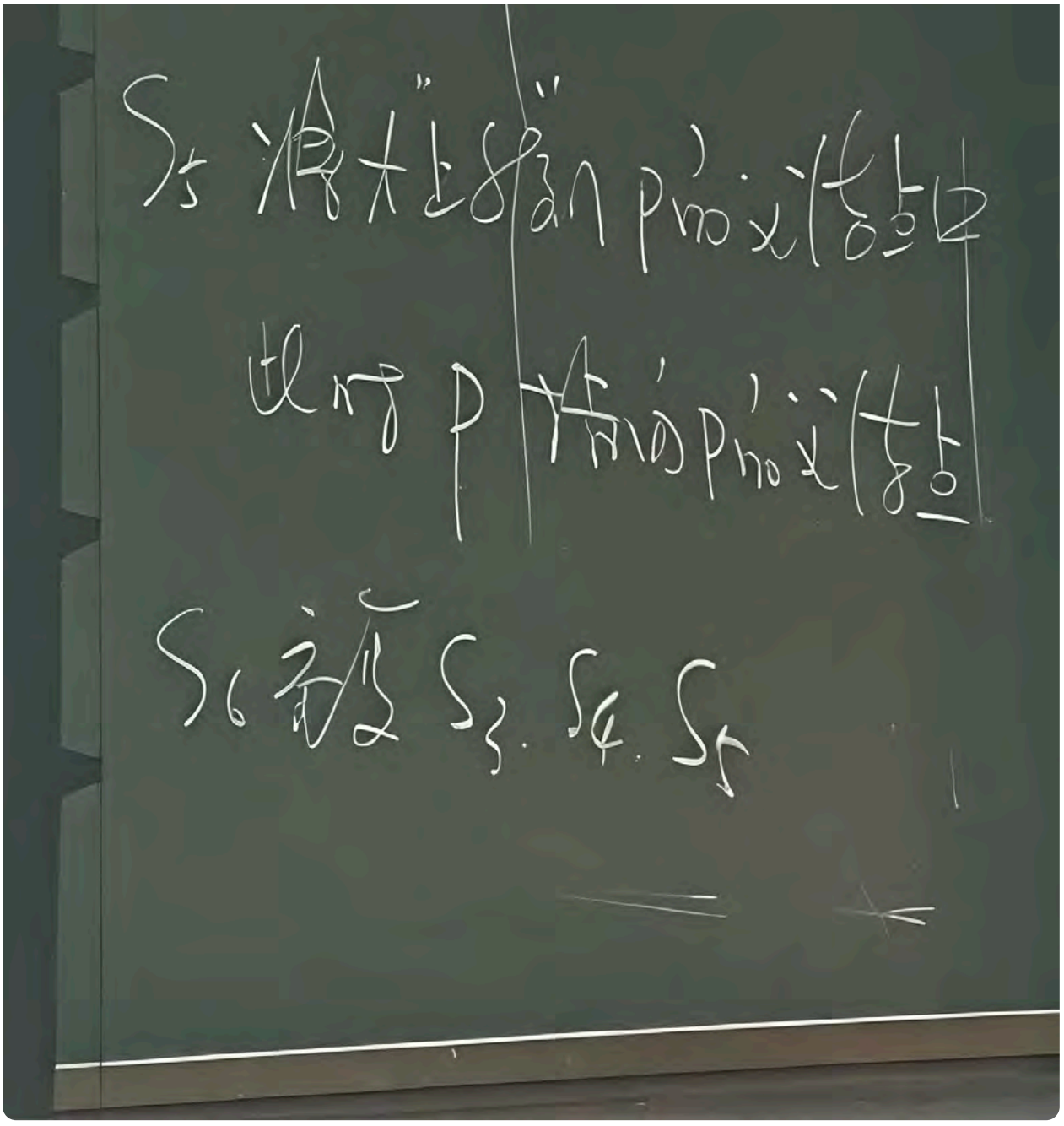
主要操作：插入(Split)，删除(Merge)，查找

比较复杂，建议看一些专栏学习：

- 知乎专栏 - 数据结构与算法之2-3树

- 2-3树定义和查找算法


S1. 查找插入key所在的叶子结点p,
S2. 按关键字有序方式，将key插入到结点p中;
S3. 如果满足"最多"，成功，
S4. 取中间值插入父；将p一分为二：p′，p″，连接到父结点中。



8.5.2 B树（这部分上课没有细讲，建议自行看ppt）

m阶B树的**定义**：

（1）树中每个结点**至多**有m棵子树，即每个结点中**至多**含有m-1个关键字；

（2）如果根结点不是叶子结点，则根结点至少含有两棵子树；

（3）除根结点和叶子结点之外的所有非终端结点，每个结点至少有[m/2]棵**子树**，有[m/2]-1个**关键字**；

（4）所有的非终结点中包含下列信息数据：

（n,P0,K1,P1,K2,P2,...,Kn,Pn)

其中：Ki(i=1,...,n)为关键字，并且Ki<Ki+1(i=1,...,n-1)；Pi(i=0,...,n)为指向子树根结点的指针，且指针Pi-1所指子树中**所有结点的关键字**均小于Ki(i=1,...n)，Pn所指子树中所有结点的关键字均大于Kn，n([m/2]-1≤n≤m-1)为关键字的个数（或n+1为子树的个数）；

（5）所有的叶子结点都出现在**同一层次**上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

注：**在B-树中，每个结点内的关键字从小到大有序排列。**

需要掌握B树（尤其是2-3树）的查找，插入，删除逻辑，不需要掌握具体代码实现。

上一章 第7章 排序算法