1 线性表

顺序表、链表

顺序表和数组的区别

- 一个是大小可变的列表，而一个是大小不变的数组（当程序被编译时，其大小是固定的）
- 列表是动态数据结构（因为它的大小可以改变），而数组是静态数据结构（它具有固定大小）
- 列表和数组相关，可变大小的列表可以在固定大小的数组中实现

2 栈和队列

$$\frac{1}{n+1}C_{2n}^{n}$$

循环队列：

- 插入 `Q.rear = (Q.rear+1) % MaxSize`
- 元素个数：`(Q.rear+MaxSize-Q.front)%MaxSize`
- 判断队空：`Q.rear == Q.front`
- 判断队满：`(Q.rear+1)%MAxSize == Q.front`
- 少用一个存储空间：必须用有**MaxSize+1**个元素的数组才能表示一个长度为**Maxsize**的循环队列

3 字符串

KMP算法进行下一趟匹配时，**i**没有“回溯”**(**即**i**不变**)**，**j**也不一定从**1**再开始——这就是**KMP**算法的改进之处

当模式**p**中的第**j**个字符与正文**t**中的相应的字符**(**假设第**i**个**)** “失配”时 `p[j]≠t[i]`，在模式**p**中需重新和该字符**(t[i])**进行比较的字符的位置 `p[k]`

在改进的模式匹配算法（**KMP**算法）中，**next**数组的作用是在模式串与目标串进行匹配时，当出现不匹配的情况时，根据已经匹配的部分信息，快速确定下一次匹配的起始位置

4 数组

5 图

	邻接表	邻接矩阵
空间复杂度	无向图 O(|V| + 2|E|)；有向图O(|V| + |E|)	$O(\|V\|^2)$
适合用于	存储稀疏图	存储稠密图
表示方式	不唯一	唯一
计算度/出度/入度	计算有向图的度、入度不方便，其余很方便	必须遍历对应行或列
找相邻的边	找有向图的入边不方便，其余很方便	必须遍历对应行或列

生成树：

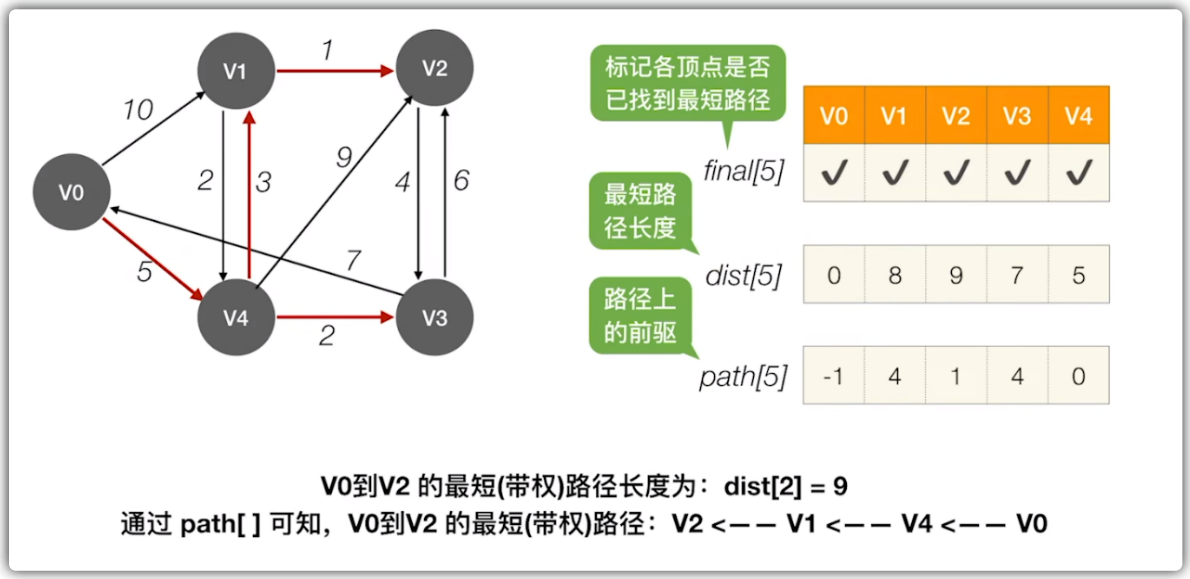
- 连通图的生成树是包含图中全部顶点的一个极小连通子图
- 若图中顶点数为**n**，则它的生成树含有**n-1**条边。对生成树而言，若砍去它的一条边，则会变成非连通图，若加上一条边则会形成一个回路

最小生成树

- **Prim**算法：从某一个顶点开始构建生成树；每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。
- **Kruskal**算法：每次选择一条权值最小的边，使这条边的两头连通（原本已经连通的就不选）；直到所有结点都连通

最短路径问题

- **Dijkstra**算法：检查所有邻接V_i的顶点，若其**final**值为**false**，则更新**dist**和**path**信息



有向无环图：若一个有向图中不存在环，则称为有向无环图，简称***DAG图 (Directed Acyclic Graph)***

6 查找

> 平均查找长度：所有查找过程中进行关键字的比较次数的平均值

顺序查找：$O(n)$

- $ASL_{成功} = \frac{1+2+3+\ldots+n}{n} = \frac{n+1}{2}$

  $ASL_{失败} = n+1$

二分查找：适用于有序的顺序表

$mid = \lfloor (low + high)/2 \rfloor$ 向下取整

- $ASL_{bs} = \sum_{i=1}^{n} P_i \times C_i = \frac{1}{n} \sum_{i=1}^{n} C_i$

  $= \frac{1}{n} \sum_{j=1}^{h} j \times 2^{j-1} = O(\log_2 n)$ 个

- 树高 $h = \lceil \log_2(n+1) \rceil$

- 先求“个数”：
  $ASL_{bs.suss} = (1\times1 + 2\times2 + 4\times3 + 8\times4) / 15 = 3.26$
  “次数” = 2 × “个数” - 1 = 2 × $ASL_{bs.suss}$ - 1 = 5.52
  $ASL_{bs.unsuss} = (4 + 4 + \cdots + 4) / 16 = 4$
  “次数” = 2 × “个数” = 2 × $ASL_{bs.unsuss}$ = 8

成功的二分查找恰好走了一条从判定树的根结点到某个内部结点的路径，其关键字比较个数为该内部结点在树中的层次数

分块查找/索引顺序查找

1. 在索引表中确定待查记录所属的分块（顺序/折半）
2. 在块内顺序查找

用折半查找索引：若索引表中不含目标关键字，则折半查找索引表最终停在**low>high**，要在**low**所指分块中查找

算法分析: 分块查找的平均查找长度 $ASL_{bs} = ASL_b + ASL_s$
其中, ASL_b为查找索引表的ASL, ASL_s为查找块的ASL.
设将长度为n的顺序表平均分成b块, 每块含s个元素, 则 $b=\lceil n/s \rceil$
且设每个元素查找的概率相同.

若查找索引表为二分查找, 则 $ASL_{bs} = \log_2(n/s+1) + s/2$

若查找索引表为顺序查找, 则 $ASL_{bs} = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}(\frac{n}{s} + s) + 1$

分成**b**块，每块**s**个元素

ASL取最小值：**n**个元素分成$\sqrt{n}$ 块，每个块内有$\sqrt{n}$个元素

平衡二叉树

若树高为**h**，则最坏情况下查找一个关键字最多需要对比**h**次，则查找操作的时间复杂度不可能超过$O(h)$

以n_h表示深度为**h**的平衡树中含有的最少结点数

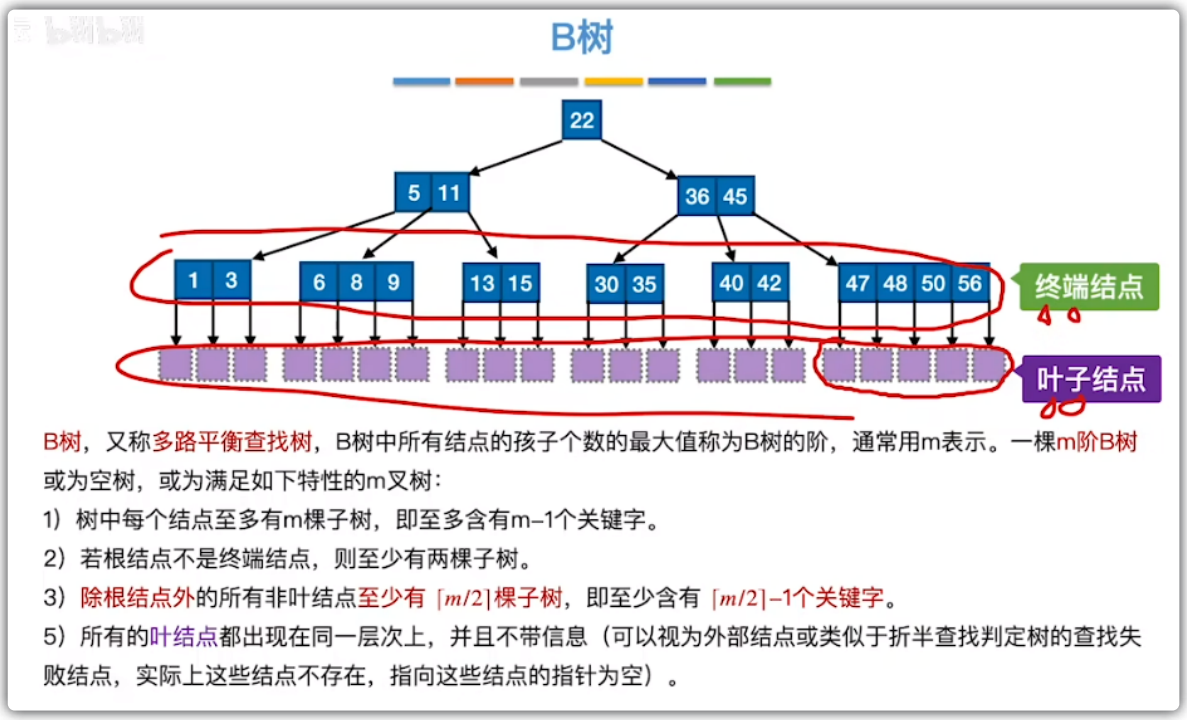
结点数为**n**，$h_{max} = O(log_2 n)$，因此平衡二叉树的平均查找长度为$O(log_2 n)$

B树

如何保证查找效率

- 若每个结点内关键字太少，导致树变高，要查找更多层结点，效率更低

  策略：m叉查找树中，规定除了根节点外，任何结点至少有$\lceil m/2 \rceil$个分叉，即至少含有$\lceil m/2 \rceil - 1$ 个关键字
- 不够“平衡”，树会很高，要查很多层结点
  - 策略：m 叉查找树中，规定对于任何一个结点，其所有子树的高度都要相同，不能有高度差



B树，又称多路平衡查找树，B树中所有结点的孩子个数的最大值称为B树的阶，通常用m表示。一棵m阶B树或为空树，或为满足如下特性的m叉树：

1）树中每个结点至多有m棵子树，即至多含有m-1个关键字。

2）若根结点不是终端结点，则至少有两棵子树。

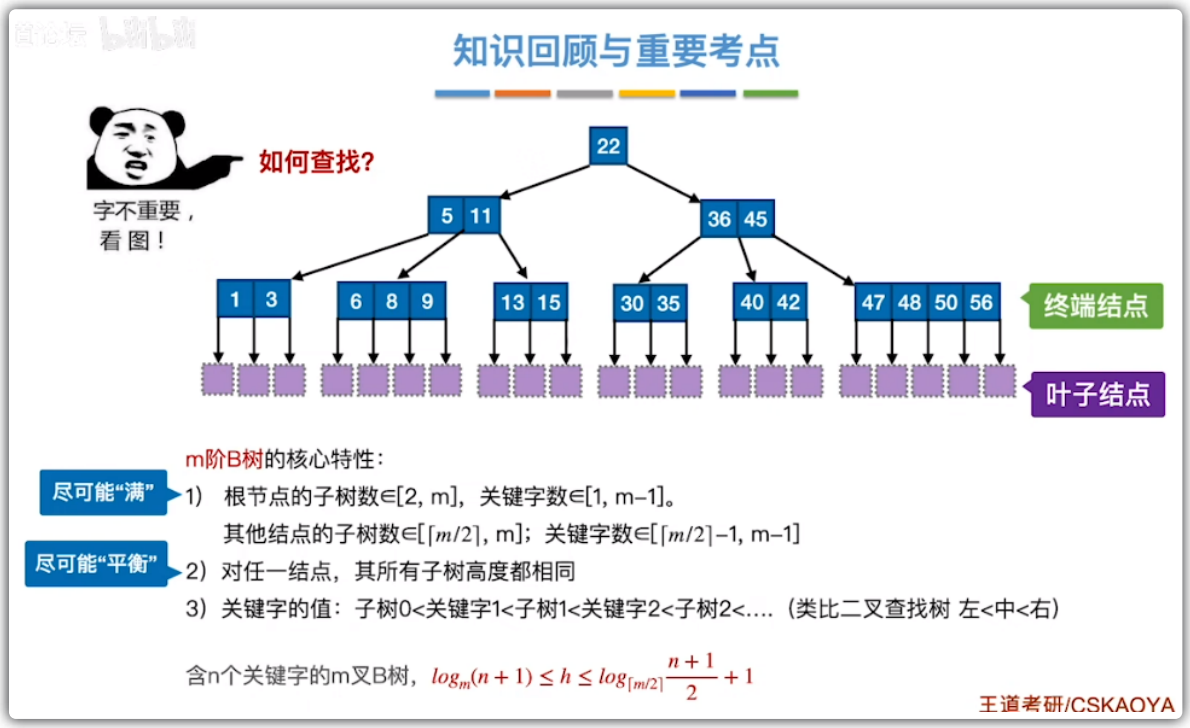
3）除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$棵子树，即至少含有 $\lceil m/2 \rceil$-1个关键字。

5）所有的叶结点都出现在同一层次上，并且不带信息（可以视为外部结点或类似于折半查找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。

4）所有非叶结点的结构如下：

n	P_0	K_1	P_1	K_2	P_2	…	K_n	P_n

其中，Ki（i = 1, 2,…, n）为结点的关键字，且满足K1 < K2 <…< Kn；Pi（i = 0, 1,…, 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，Pi所指子树中所有结点的关键字均大于Ki，n（$\lceil m/2 \rceil - 1 \leq n \leq m-1$）为结点中关键字的个数。



若**B**树中每个非终端结点可以有**2**或**3**个子树，这种**3**阶**B**树便又称为**2-3**树

B树的插入：在插入**key**后，若导致原结点关键字数超过上限，则从中间位置$\lceil m/2 \rceil$将其中的关键字分为两部分，左部分包含的关键字放在原结点中，右部分包含的关键字放到新结点中，中间位置$\lceil m/2 \rceil$的结点插入原结点的父结点。若此时导致其父结点的关键字个数也超过了上限，则继续进行这种分裂操作，直至这个过程传到根结点为止，进而导致**B**树高度增**1**。

B树的删除：

- 若被删除关键字在终端节点， 则直接删除该关键字 **(**要注意节点关键字个数是否低于下限$\lceil m/2 \rceil - 1$**)**
- 若被删除关键字在非终端结点，则用直接前驱或直接后继来替代被删除的关键字
- 若删除后低于下限：
  - **兄弟够借**：若被删除关键字所在结点删除后的关键字个数低于下限，且此结点右（或左）兄弟结点的关键字个数还很宽裕，则需要调整该结点、右（或左）兄弟结点及其双亲结点（父子换位法）

- 当右兄弟很宽裕时，该结点的后继（在父结点）下坠到该结点，后继的后继（在右兄弟结点）上浮到其父结点
- 当左兄弟很宽裕时，该结点的前驱（在父结点）下坠到该结点，前驱的前驱（在左兄弟结点）上浮到其父结点

- **兄弟不够借**：将关键字删除后与左（或右）兄弟结点及双亲结点（下坠）中的关键字进行合并

哈希查找

开放地址法的特点：

- 优：思路清晰，算法简单
- 缺：无法处理“溢出”；**HT**中的元素不易删除；易于产生“聚集”现象。

链地址法的特点

- 优点：节省空间（用于大记录）；解决冲突；没有溢出；方便删除
- 缺点：额外使用空间

> 哈希函数设计的足够好（*设计冲突足够少的哈希函数*）——最理想情况：散列查找时间复杂度达到$O(1)$

同一哈希函数、不同的处理“冲突”的方法，其**ASL**不同

对处理“冲突”方法相同的哈希表，其**ASL**依赖于装填因子

装填因子α**=**表中记录数**/**散列表长度——散列表装的有多满；直接影响散列表的查找效率

装填因子$\alpha = \frac{\text{已装入记录数}}{\text{HT长度HashSize}}$ ⇨ α越小，发生冲突的可能性越小；α越大，发生冲突的可能性越大

装填因子α过小，空间浪费就越多.

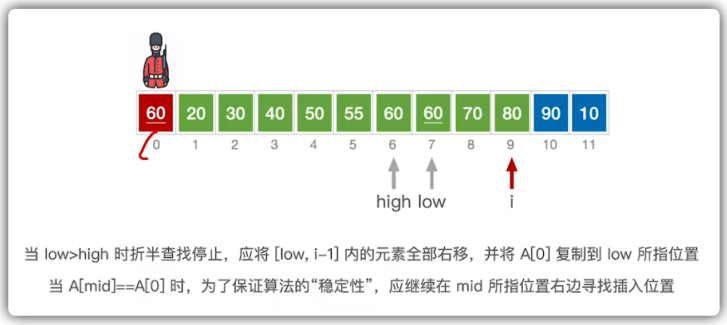
开放地址法(线性探测)：$ASL_{succ} = \frac{1}{2}(1 + \frac{1}{1-\alpha})$　$ASL_{unsucc} = \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$

链地址法：$ASL_{succ} = 1 + \frac{\alpha}{2}$　$ASL_{unsucc} = \alpha + e^{-\alpha}$

7 排序

插入排序——依次加入排序

- 最好时间复杂度：$O(n)$，原本就有序
- 最坏时间复杂度：$O(n^2)$
- 稳定性：稳定

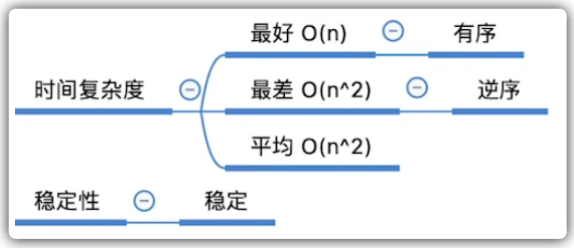
- 改进——二分插入排序：仅减少了关键字的比较次数，而记录移动的次数未变
  - 

  - 整体来看时间复杂度依旧是$O(n^2)$

希尔排序

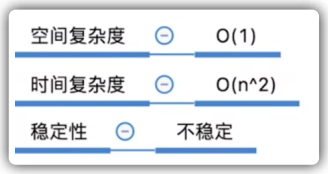
- 空间复杂度：**O(1)**
- 最坏时间复杂度：$O(n^2)$，与增量序列**d**的选择有关
- 稳定性：不稳定

冒泡排序



选择排序——选择最小的加入

选择排序的目的是如何减少记录移动的次数



归并排序

- n个元素进行2路归并排序，归并趟数=⌈log2n⌉
- 每趟归并时间复杂度为O(n)，则算法时间复杂度O(nlog2n)
- 空间复杂度=O(n)，来自于辅助数组B
- 稳定性：稳定

这里的**n**指参加归并排序有**n**个元素

结论**:** 从空间角度，适合外部排序

快速排序

若指针相遇**/i=j**，则该位置**i**就是分区索引值，即 **L.r[i] = tmp**，且返回索引指针**i**的值

- 时间复杂度=O(n*递归层数)
  - 最好时间复杂度=O(nlog2n)
    - 每次选的枢轴元素都能将序列划分成均匀的两部分
  - 最坏时间复杂度=O(n^2)
    - 若序列原本就有序或逆序，则时、空复杂度最高（可优化，尽量选择可以把数据中分的枢轴元素。）
- 空间复杂度=O(递归层数)
  - 最好空间复杂度=O(log2n)
  - 最坏空间复杂度=O(n)

不稳定

堆排序

中的关键字序列不一定满足堆的定义，因此

- 如何将一个无序序列变成一个堆？⇨建初堆(Building the initial heap)
- 如何在输出堆顶记录后，调整剩余记录成为一个新的堆？⇨堆排序

} 堆调整

空间复杂度	O(1)
时间复杂度	建堆 O(n)、排序 O(nlog n)；总的时间复杂度=O(nlog n)
稳定性	不稳定

基于大根堆的堆排序得到“递增序列”，基于小根堆的堆排序得到“递减序列”