浅谈大型互联网的算法和架构(一)

邱丹 qiudan@youku.com

2015.11 北京(合一学院大讲堂)

关于本人

邱丹

优酷土豆开发副总裁(2007~)

程序员(1989~)

现场小测验：绝症的可能性?

- 医学统计，某绝症得病的概率是千分之一
- A同学体检报告呈阳性
- A不信，医生回复：可能误报，不过我们的准确率是95%，且不会漏报。
- A得该病的可能性有多大？

得病可能性？

- (A) > 95% (极可能)
- (B) 50~95% (可能性较大)
- (C) 5~50% (可能性较小)
- (D) < 5% (极不可能)

行为偏差

- **答案：2% 。千人里有50个疑似阳性，只有1个患病**
- **医学专家仅1/5给出正确答案。**

- **行为偏差，最终的事实是多个因素的综合影响，人习惯依赖直觉一个因素，从而容易产生偏差**

- **专家也不例外。**
  - 偏差1 ：面试, mysql数据太多查询慢，答拆成1000个小表，或者做内存表期望提高查询速度。
  - 偏差2：开发网站很简单
  - 偏差3：IT技术发展很快

- 《算法 + 数据结构 = 程序》（1976, Pascal之父沃斯)
- 一句话获得1984图灵奖
- 所谓技术进步，就是不断翻新历史。
- 算法 + 架构 = 互联网程序 (20xx)

Part Ⅰ -- 查找算法(单机)

一个无序数组(Array)

数组

72	8	26	2	17	35	80

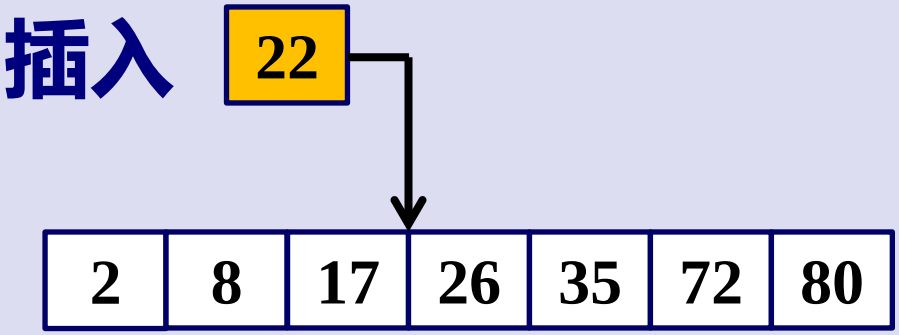
找7到20之间的数

数组

1.排序

2.二分查找快($< log_2 n$)

数组中插入数据

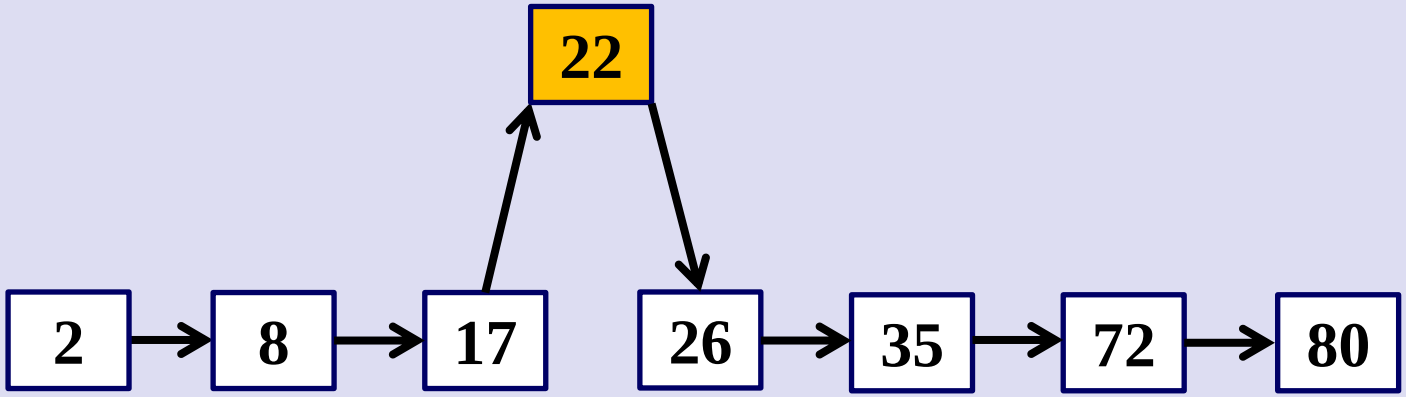


数组问题：插入太慢? 得挪数据

试试链表(Linked list)

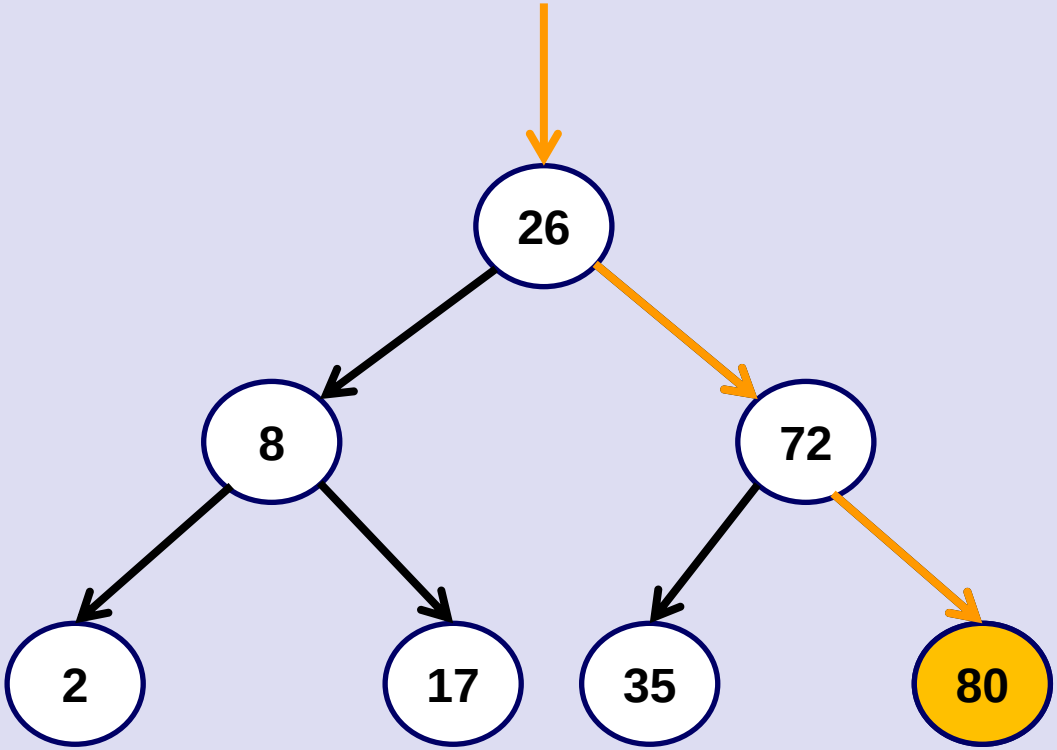
链表

链表插入数据



链表：插入快、查找慢?

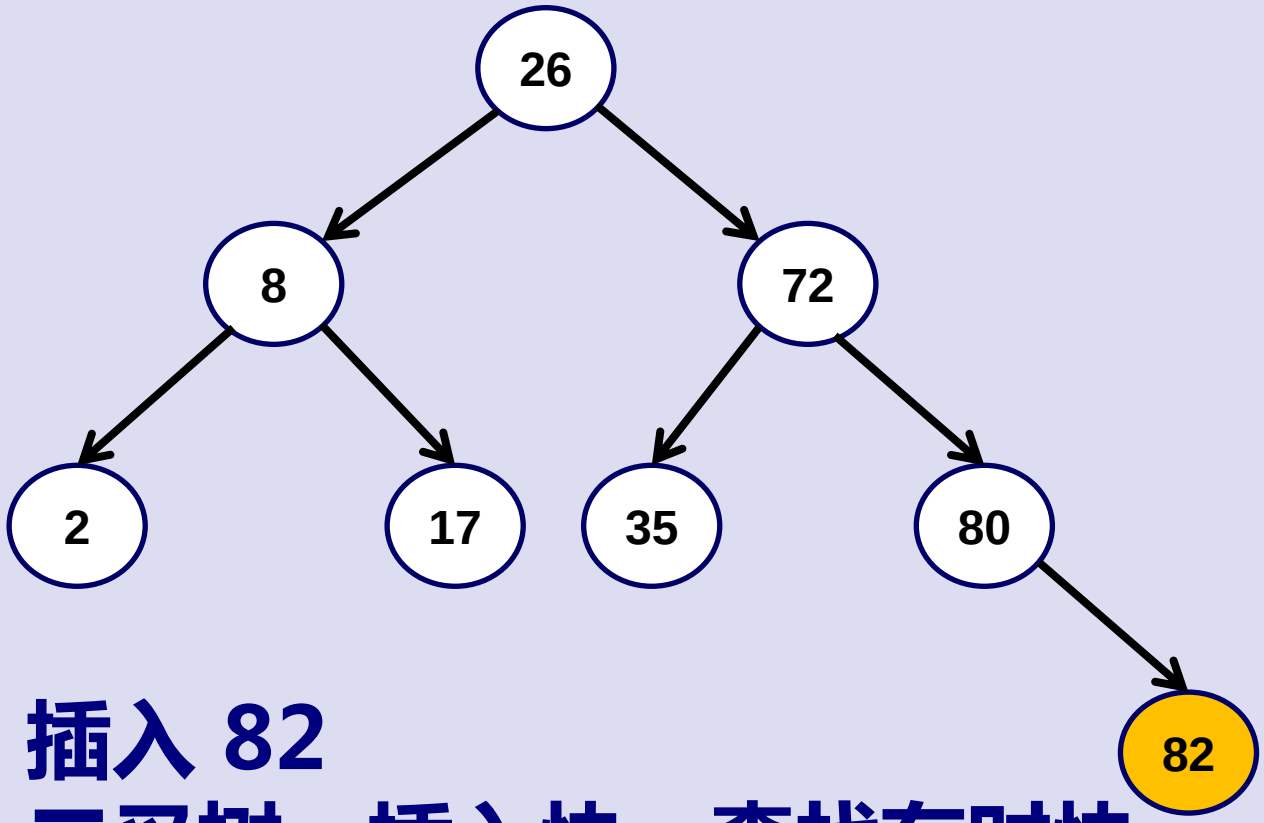
于是有了二叉树(Binary Tree)



小的在左边，大的在右边

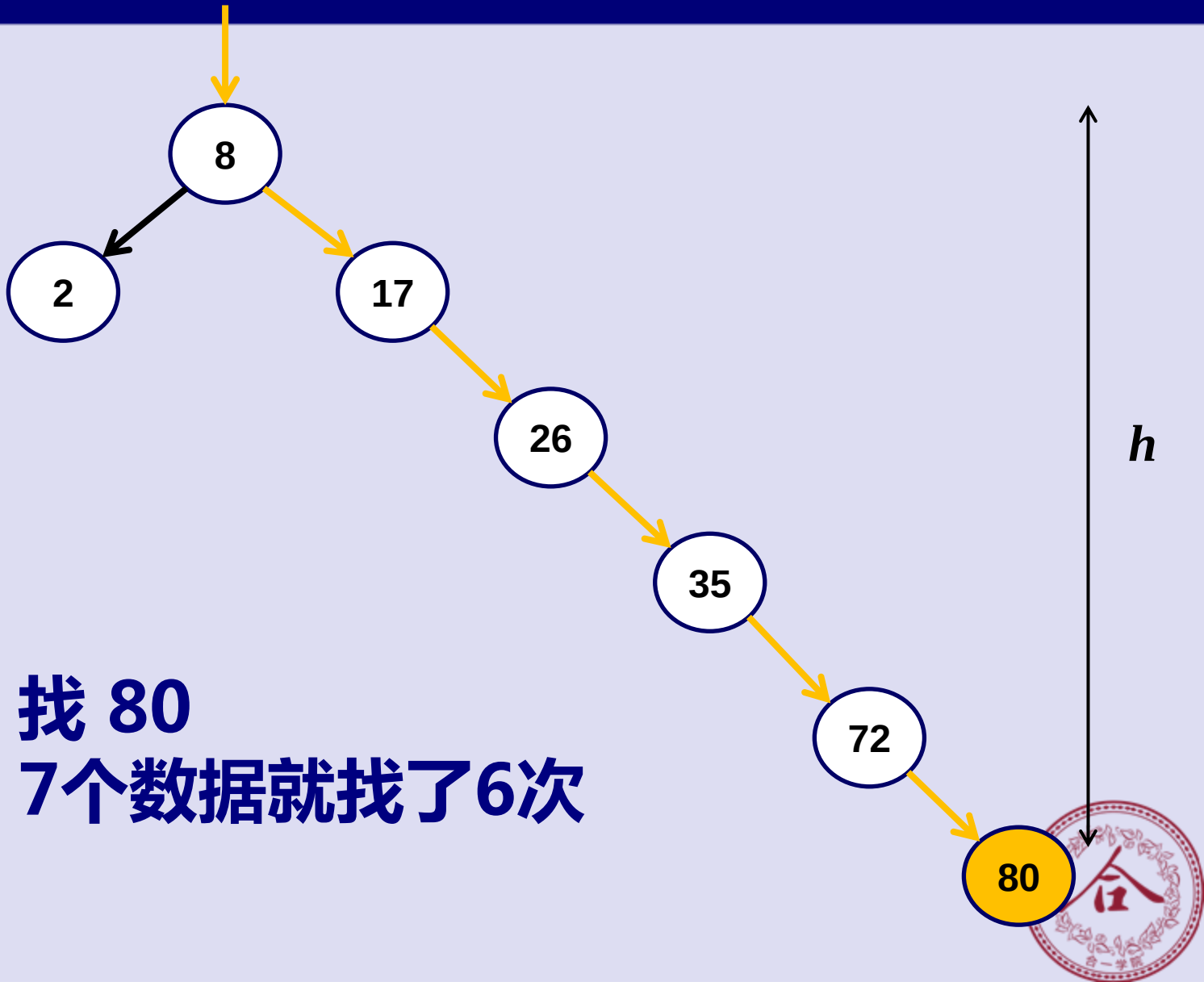
找 80

二叉树:插入数据



- 插入 82
- 二叉树：插入快、查找有时快

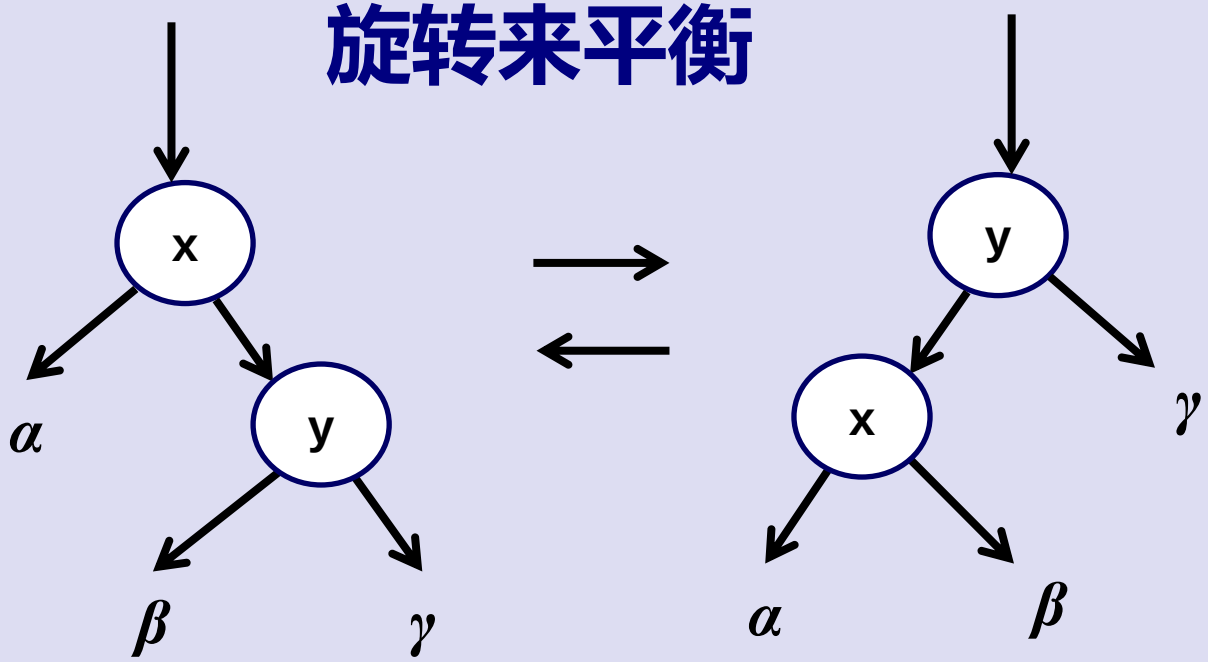
二叉树的烦恼：出现极端情况？



找 80
7个数据就找了6次

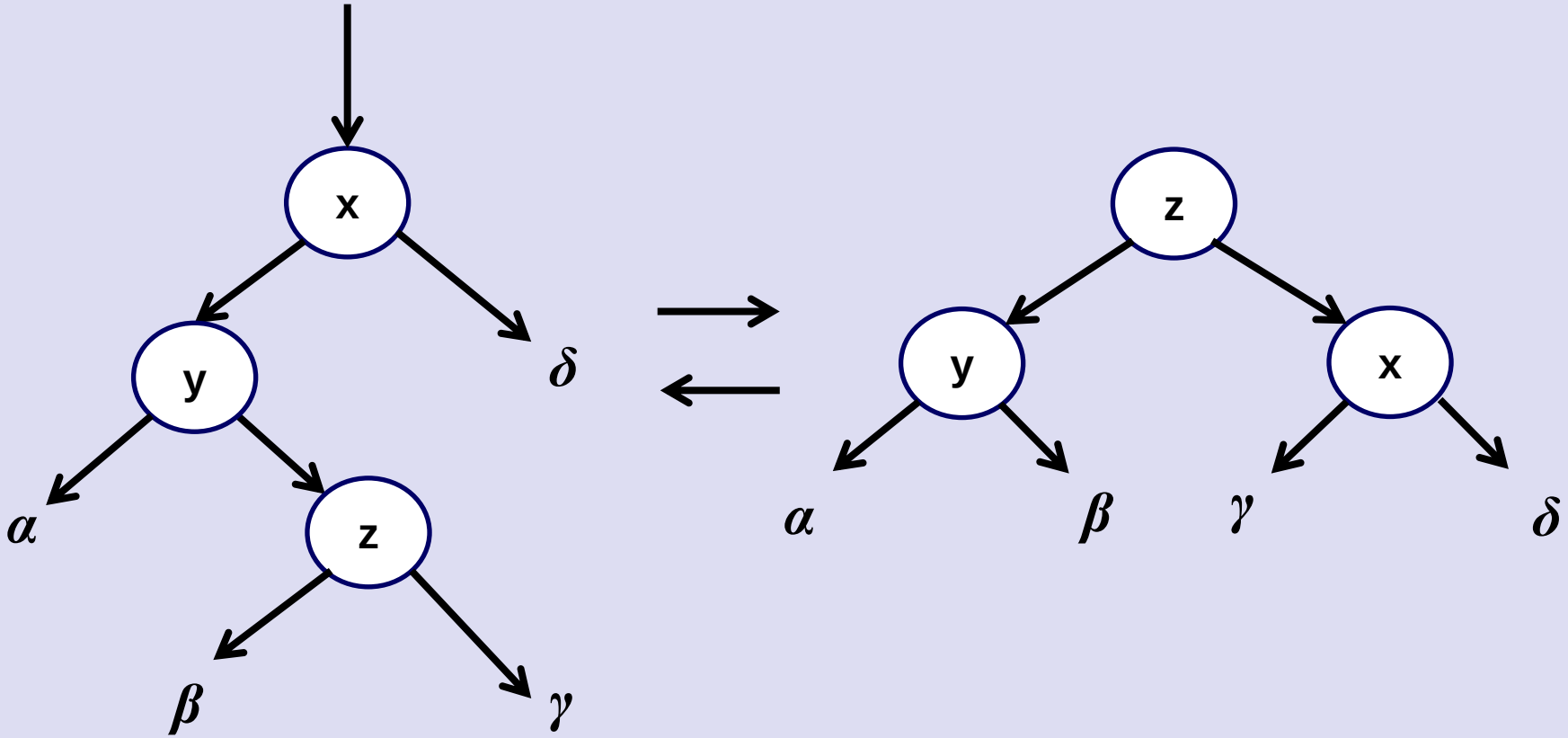
平衡二叉树:AVL Tree(1962)

旋转来平衡



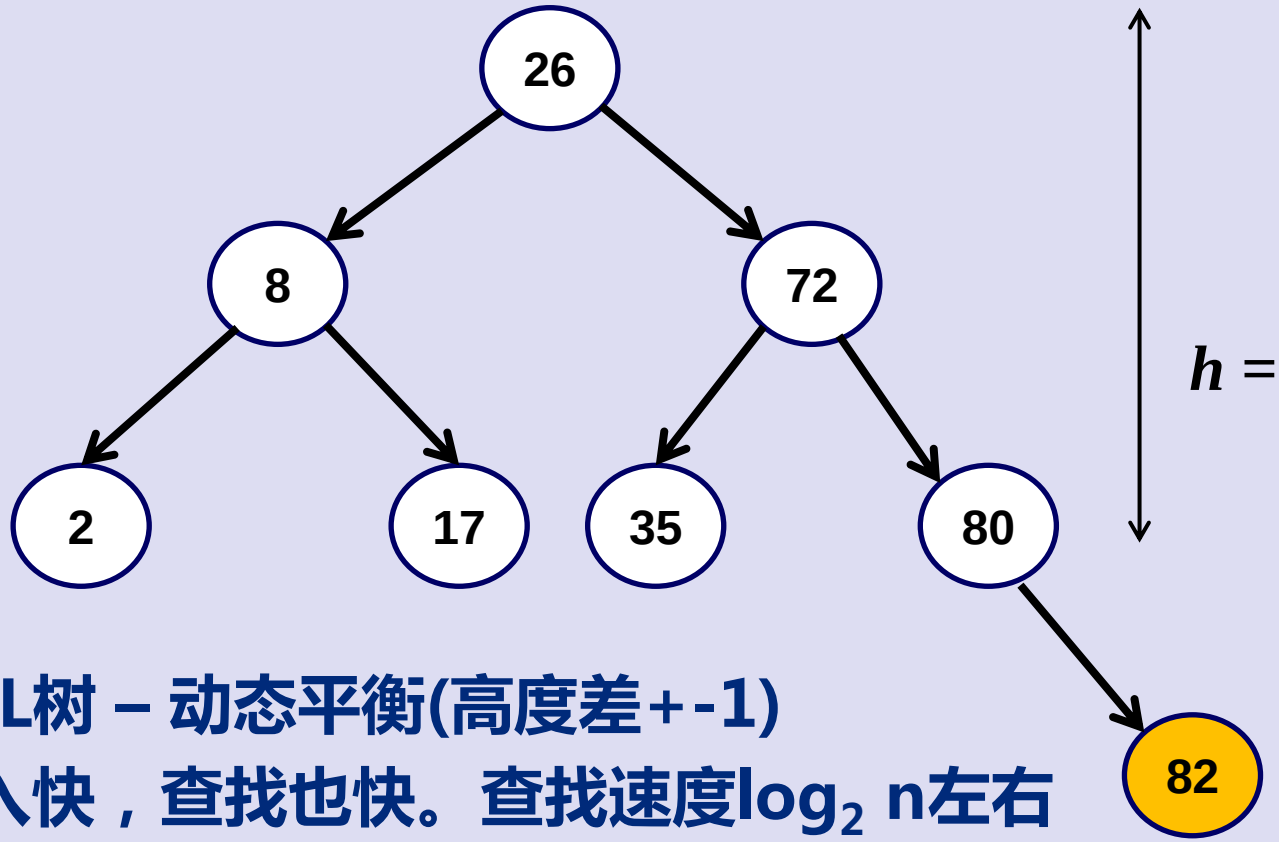
Case 1: 单旋

平衡二叉树:AVL Tree



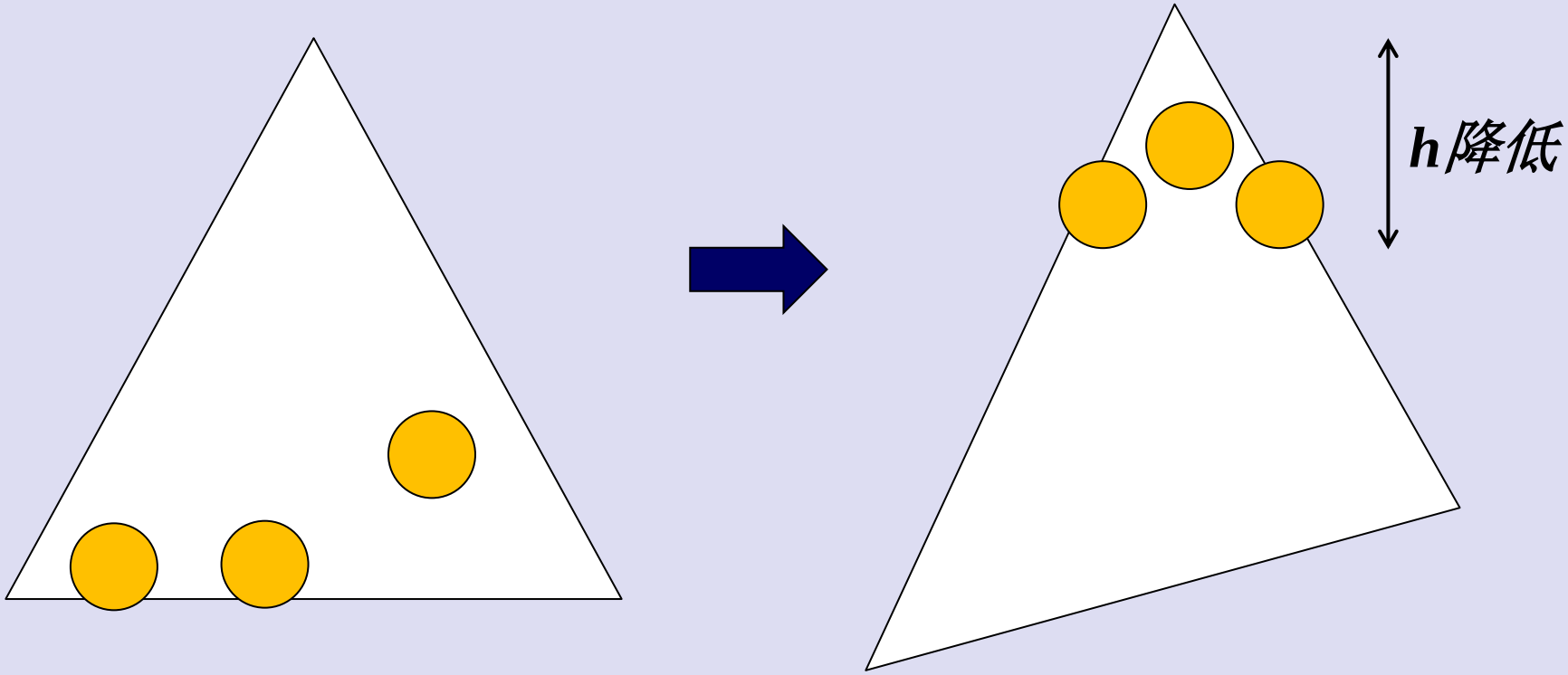
Case 2: 双旋

AVL树平衡后



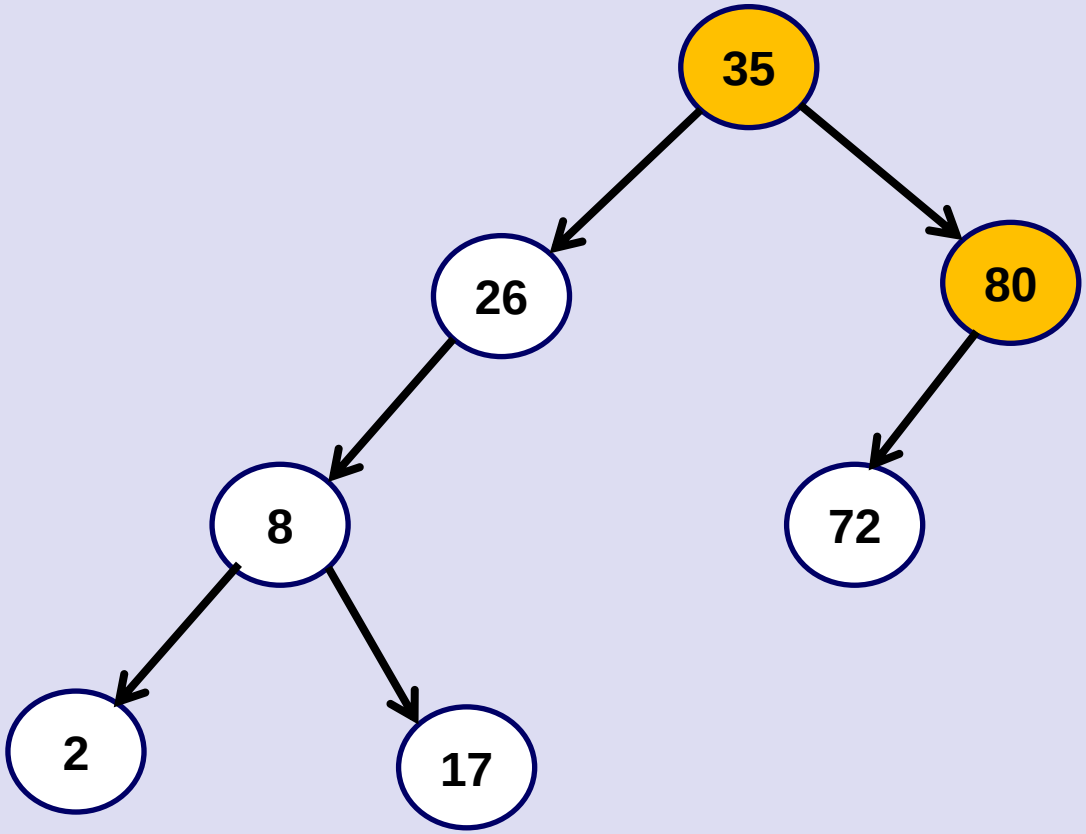
- AVL树 – 动态平衡(高度差+-1)
- 插入快，查找也快。查找速度log_2 n左右
- 完美! 还能更快吗？

伸展树(Splay Tree,1985)



- 热节点尽量挪近根节点(zig-zig,zig-zag,zig)
- 自平衡，有时会牺牲平衡
- 多用于Cache(squid3 ...)

一棵Splay Tree



经常查找的35和80靠近树根

还能更快吗？

数据不断增长 ……
内存放不下了。怎么办？

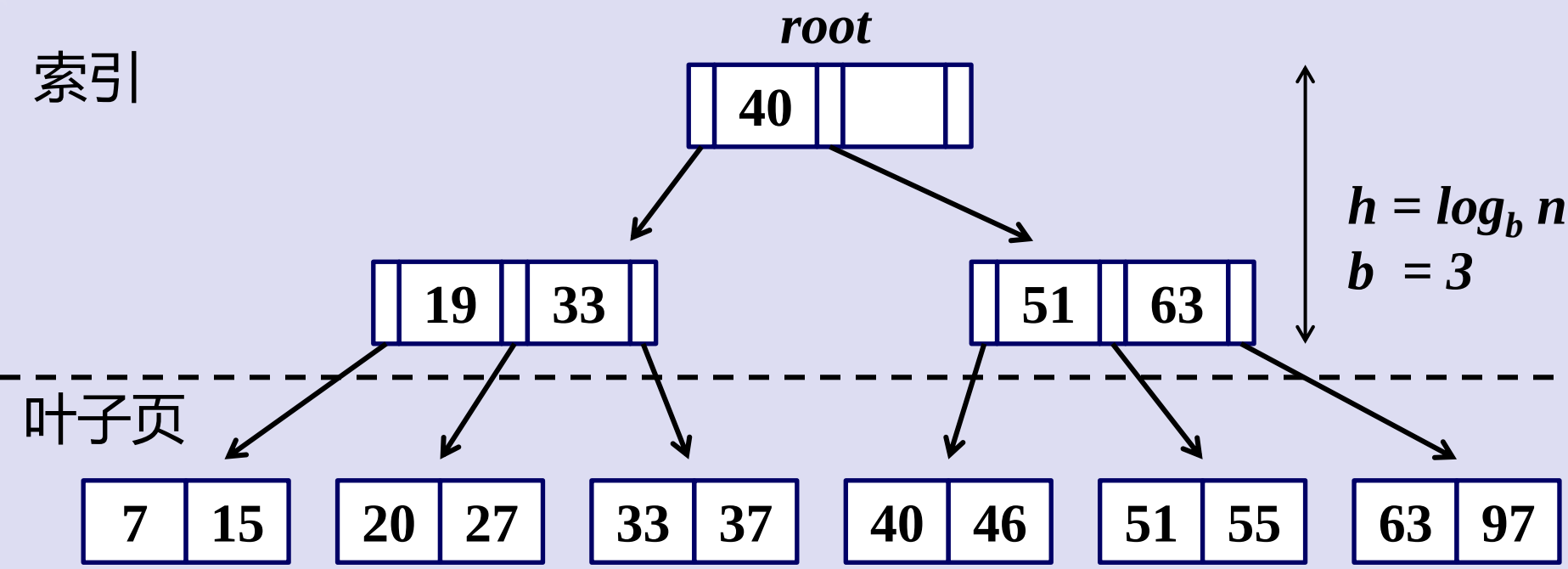
硬盘 … AVL/Splay Tree能放吗？
不能，怎么办？

硬盘问题

- I/O次数有限。需尽量减少I/O次数
- 顺序读取 >> 随机读取。 一次需多取数据

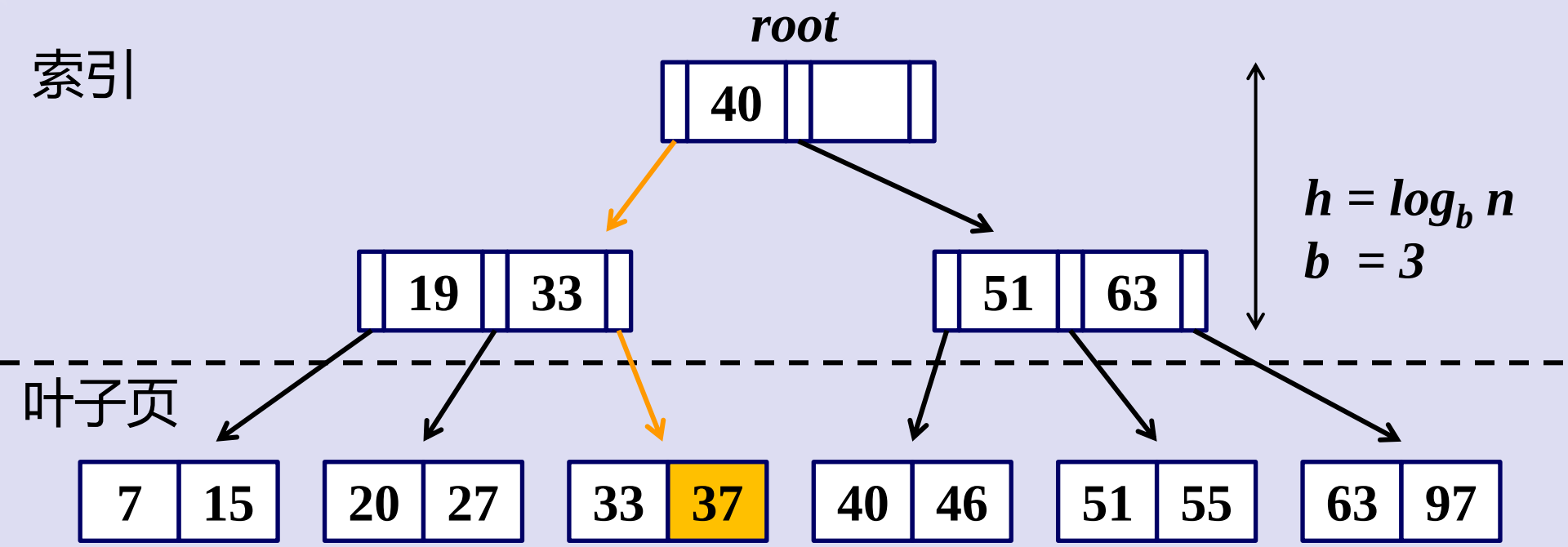
ISAM树(1964 IBM)

Indexed Sequential Access Method

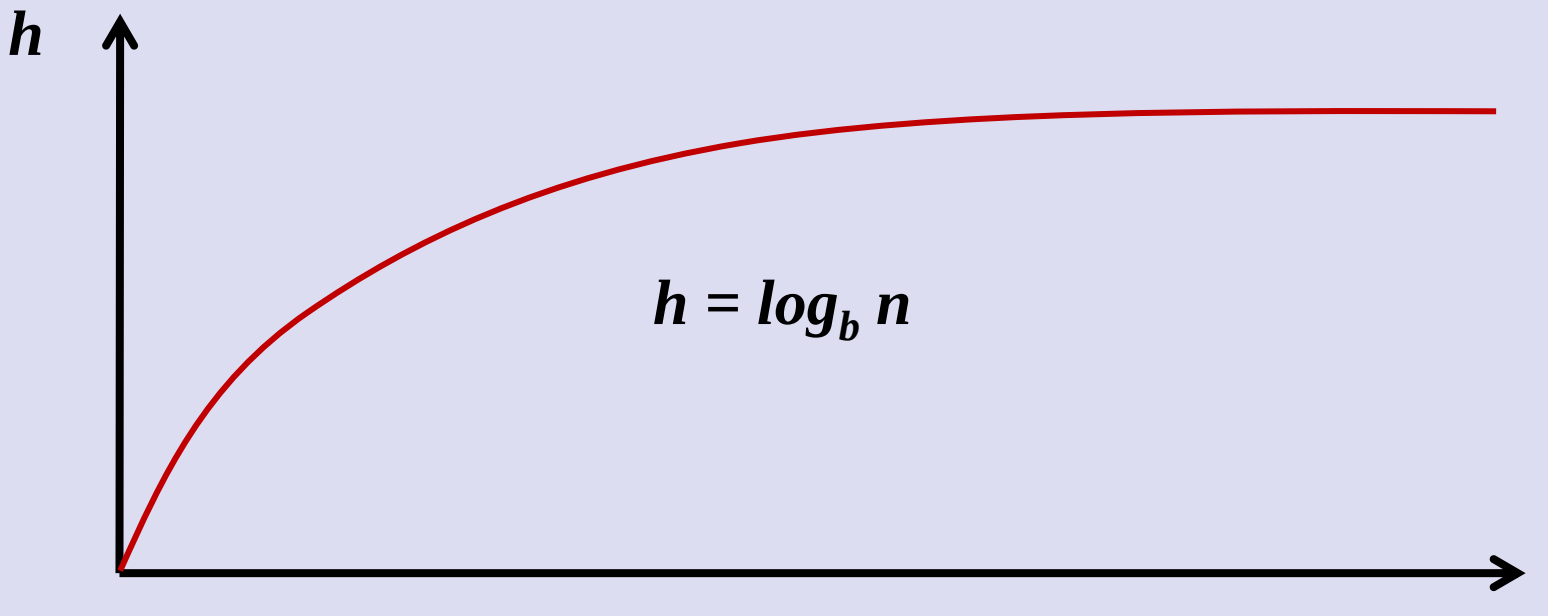


- 相当于多叉平衡树
- 树矮（能减少硬盘I/O次数）
- 节点记录多(一次性读取更多数据)

ISAM树的搜索


root
索引
40
19
33
51
63
叶子页
7
15
20
27
33
37
40
46
51
55
63
97
$h = log_b\ n$
$b = 3$

ISAM搜索效率



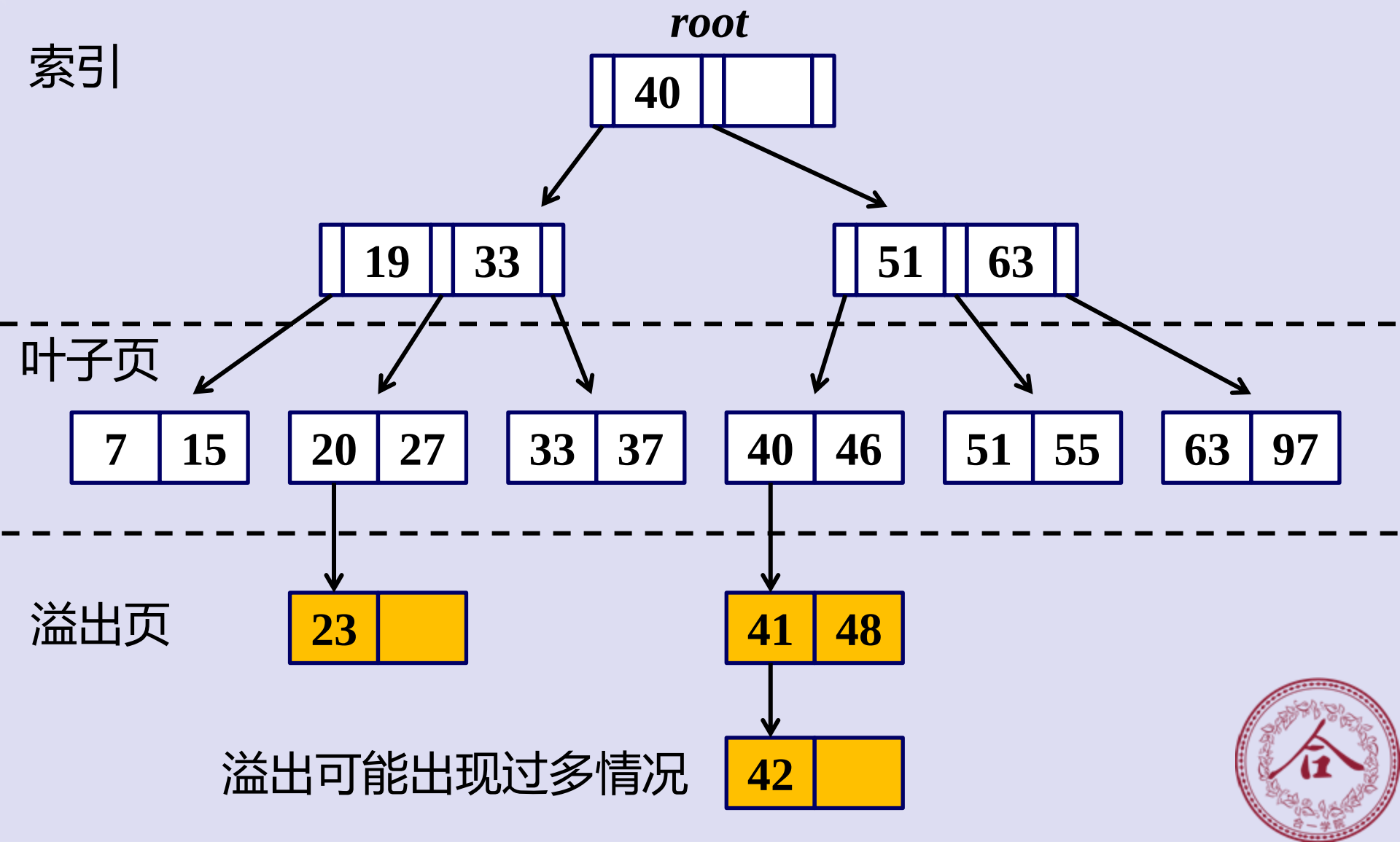
if 每页1k,每数据8 bytes ➔ 每页b ≈ 100个数据

h ≈ 2, n=10,000

h ≈ 3, n=1,000,000

h ≈ 4, n=100,000,000

ISAM索引
插入数据


索引
root
40
19
33
51
63
叶子页
7
15
20
27
33
37
40
46
51
55
63
97
溢出页
23
41
48
溢出可能出现过多情况
42

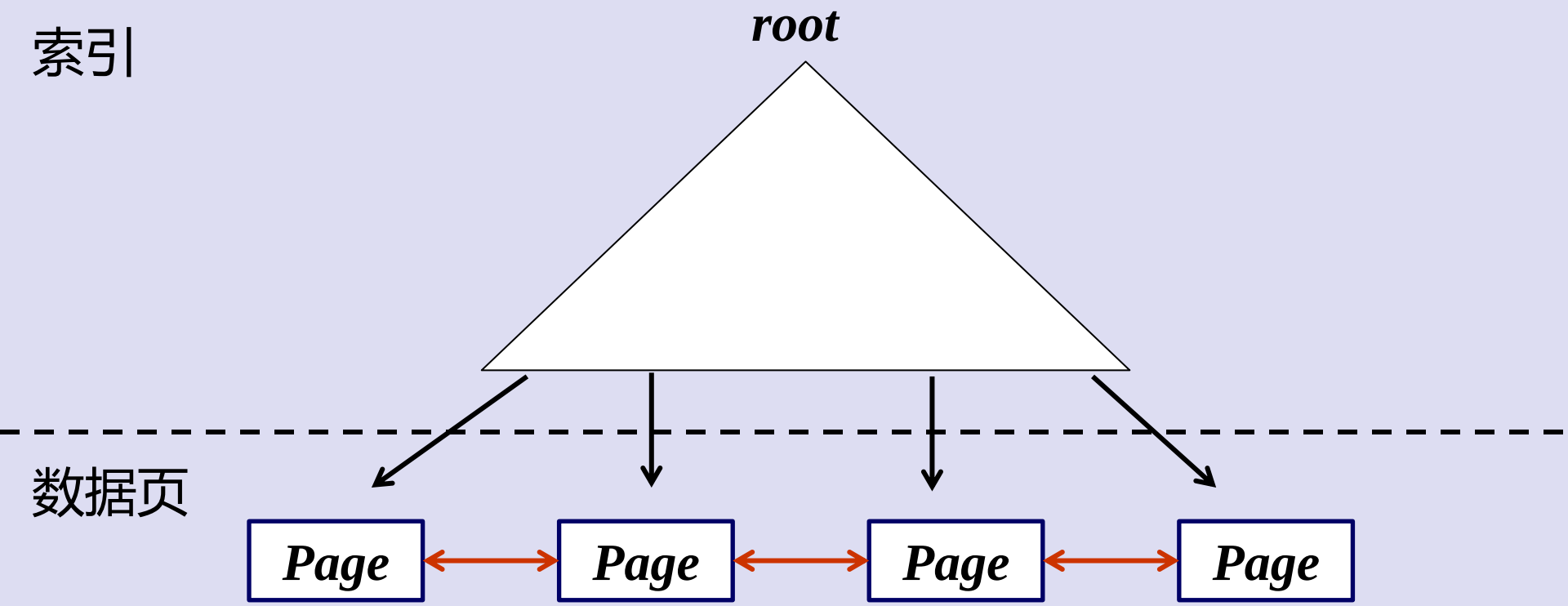
ISAM应用

- **Foxpro/dBase**
- **MS Access**
- **MySQL 3.23前的唯一引擎(ISAM Engine)**
  - 后来被MyISAM取代(并不是ISAM树)
- **Berkeley DB**

- **优点：简单，易实现**
- **问题：溢出页不平衡问题, rebuild index?**

B+Tree(B+树)

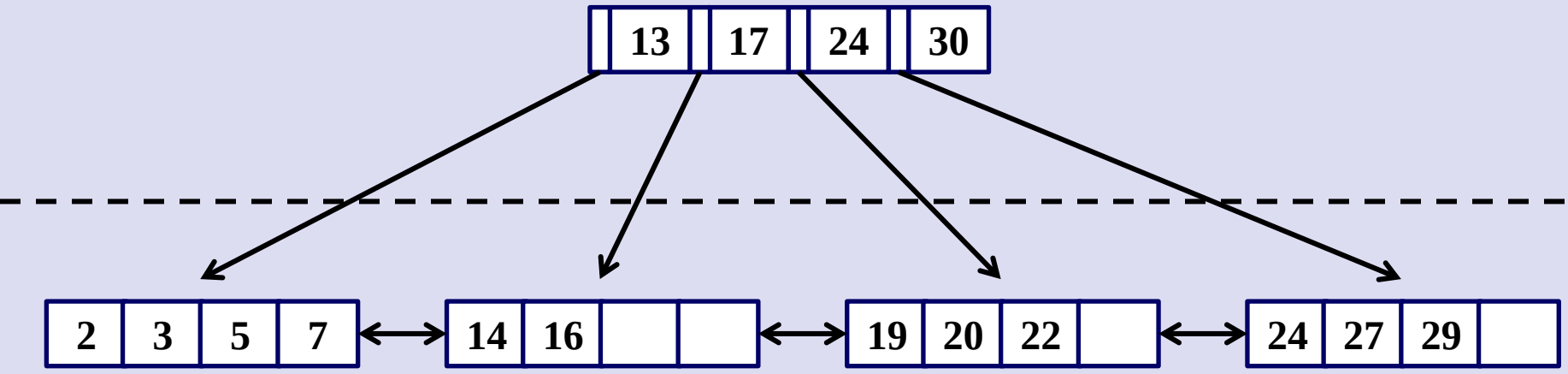
B-tree 1970，B+Tree 1979 Douglas Comer



- Btree相当于能自平衡的ISAM
- 叶数据页相互链接, Range查询更有效率

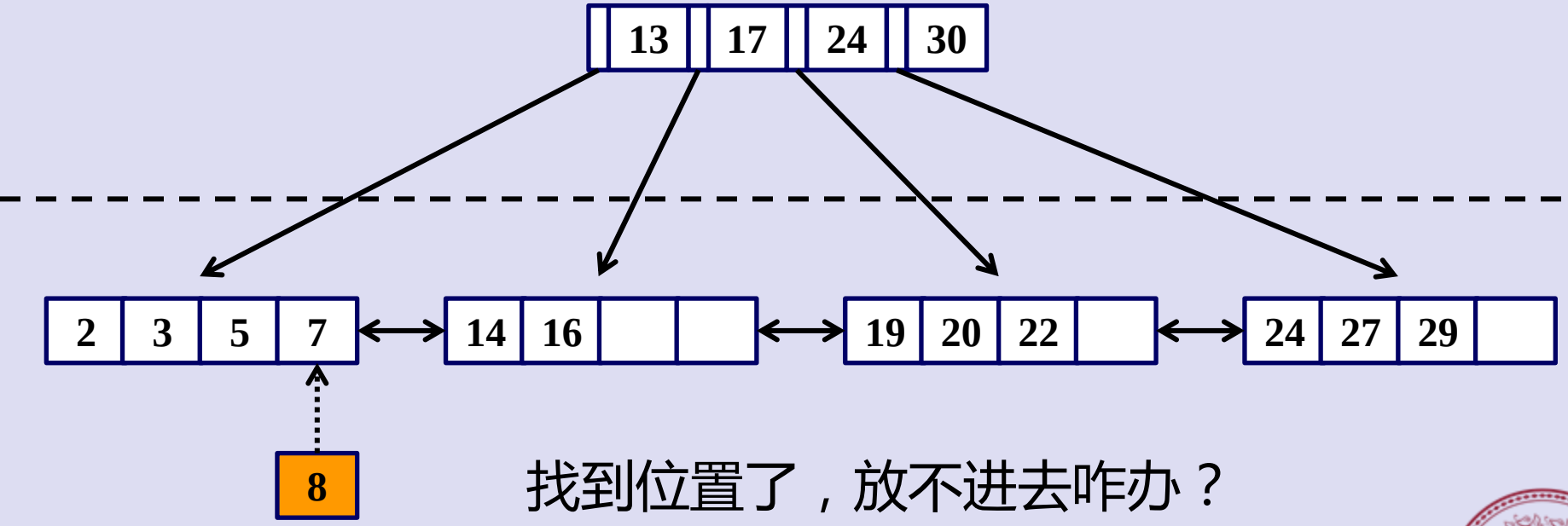
B+树的插入

插入 8
如何不溢出、保平衡?



B+树的插入(Cont.)

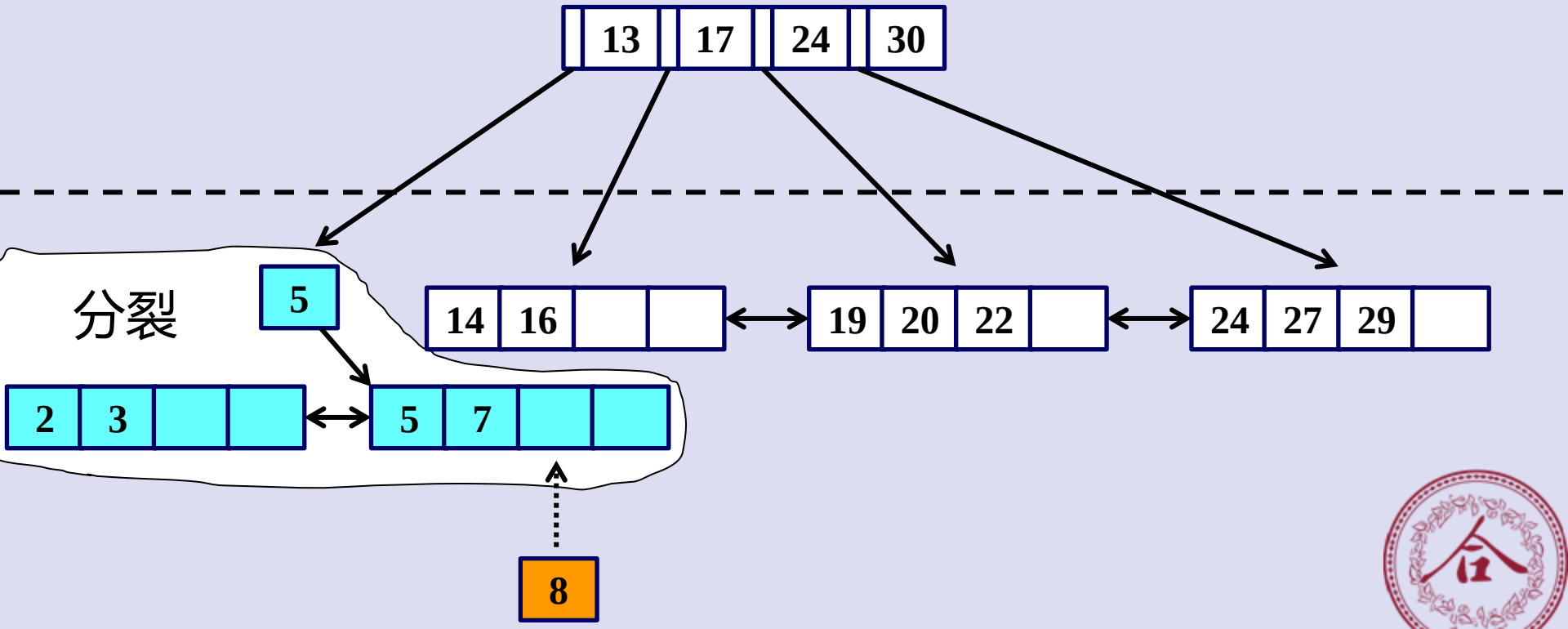
插入 8



找到位置了，放不进去咋办？

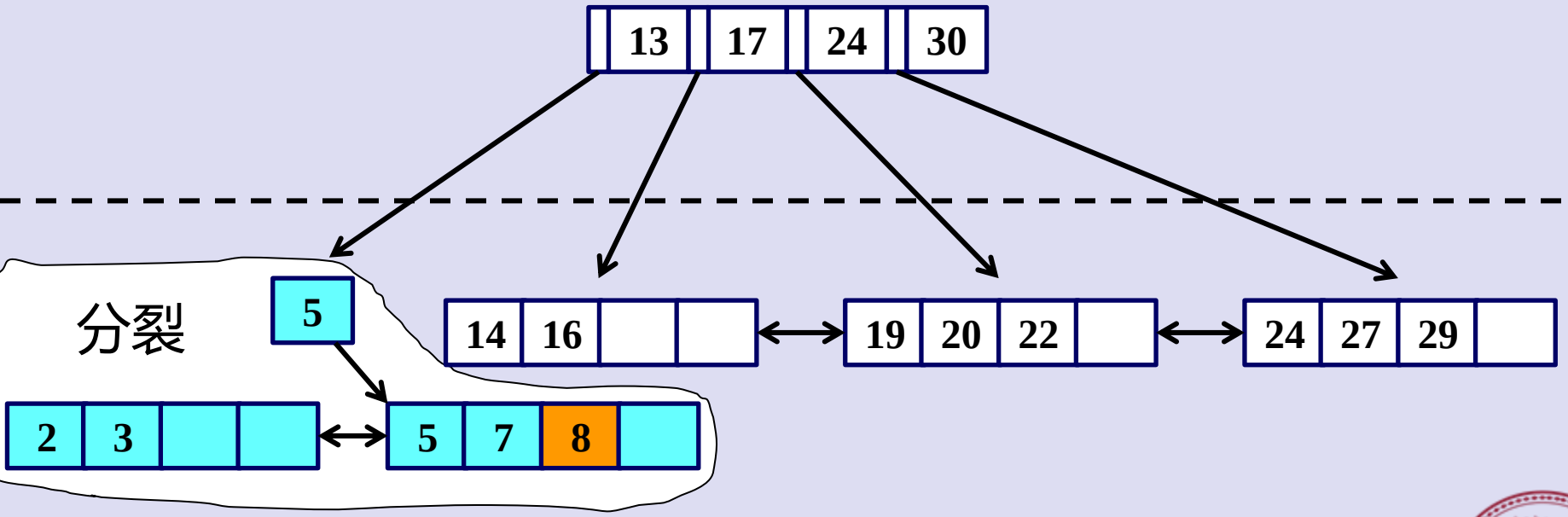
B+树的插入(Split)

插入 8



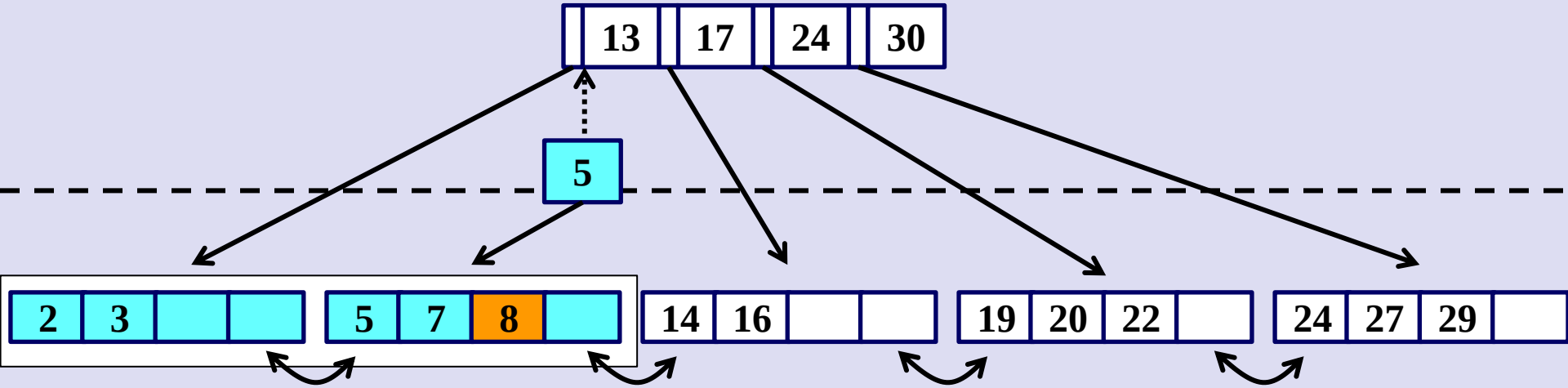
B+树的插入(Cont.)

插入 8



B+树的插入(Cont.)

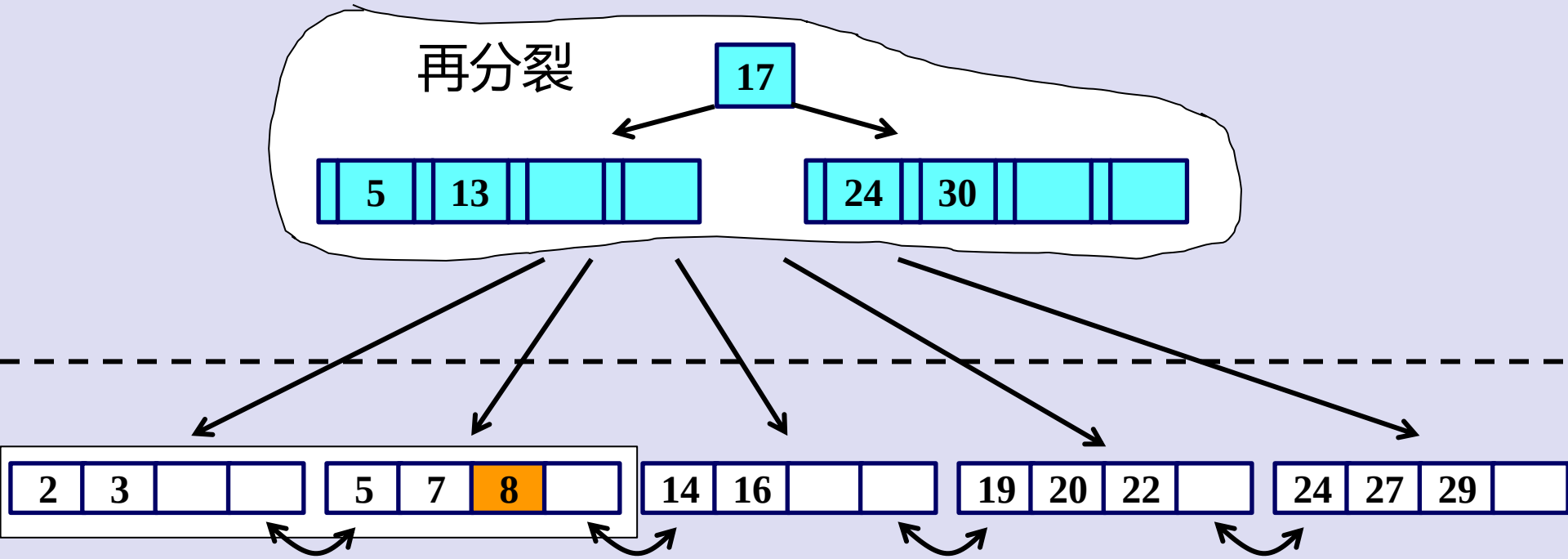
插入 8



5放不进去咋办，难道又分裂？

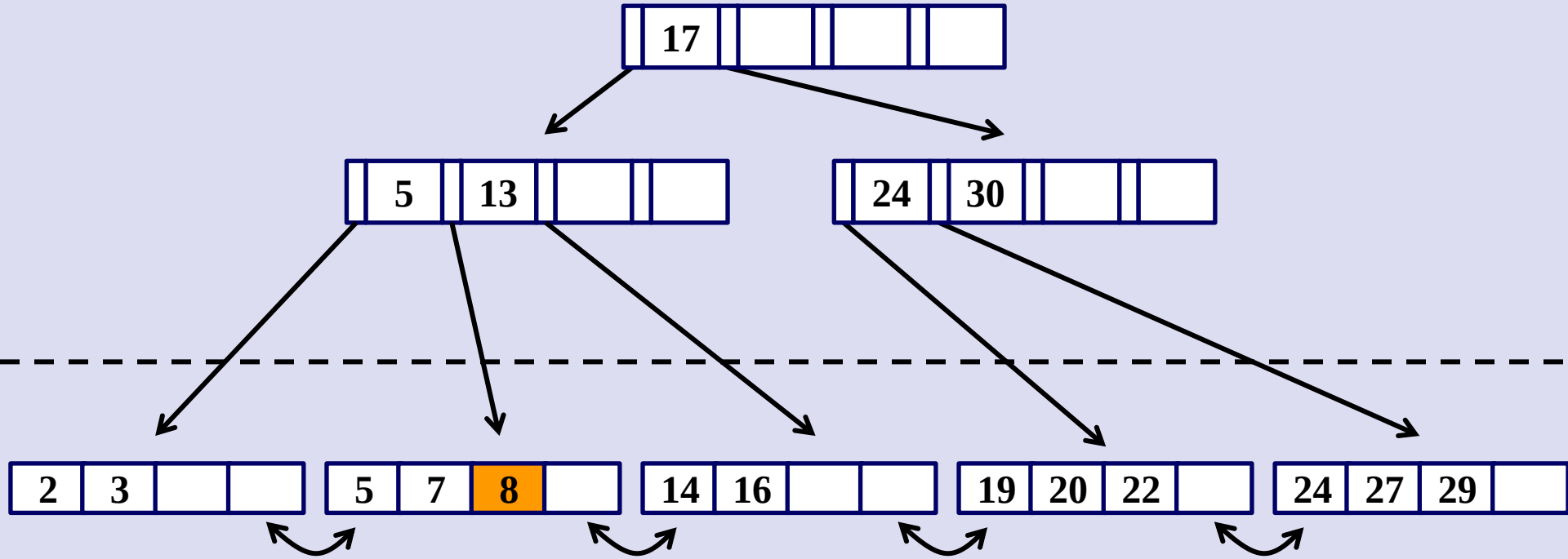
B+树的插入(Cont.)

插入 8



B+树的插入(完成)

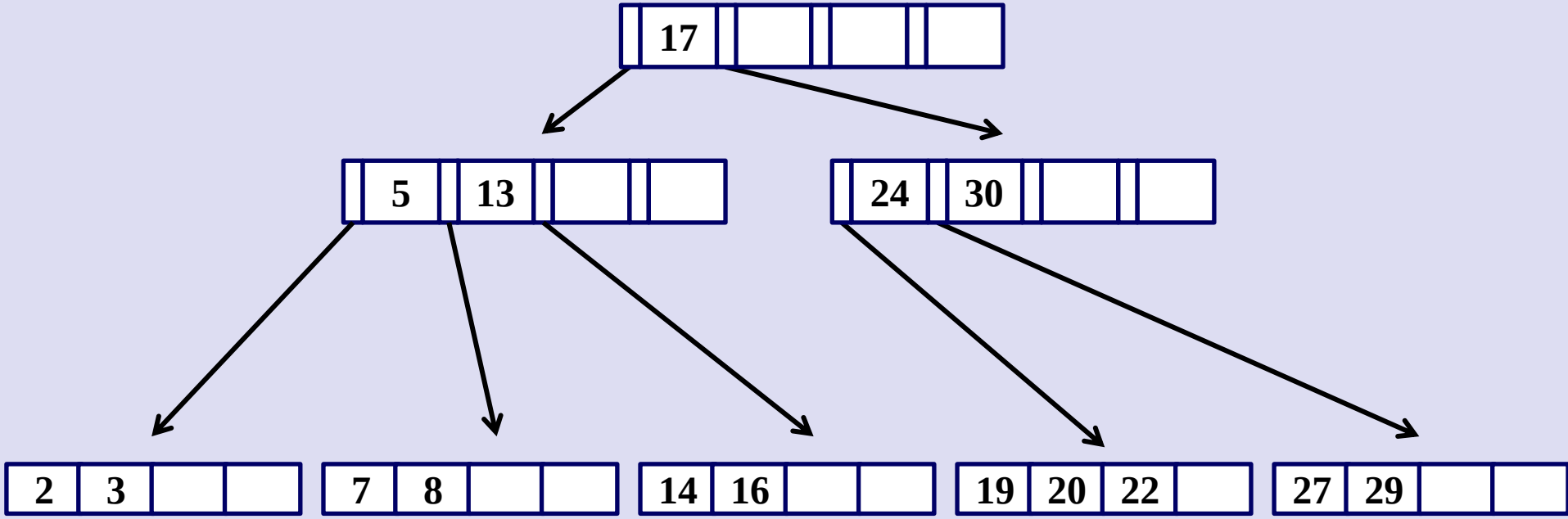
插入 8



完成！
两次分裂，多了一层
没有溢出，完成平衡

B-Tree、B+Tree、B*Tree

- **关注B+Tree就行了。**



一颗B-Tree例子

- 索引即数据
- 叶节点无法Link(Range慢)

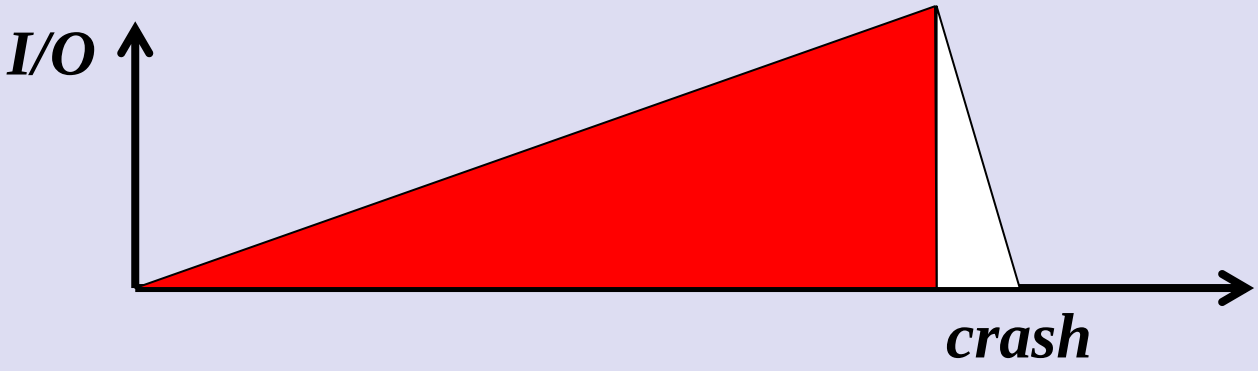
B+Tree统治力(1979~)

- MySQL / MyISAM/Innodb
- Oracle / SQL Server
- Berkeley DB / TokyoCabinet/QDBM
- Mongodb / Hbase
- XFS
- … …

- 天天同B+Tree打交道，你了解它吗？
- 30年历史，谁能打破？

B+Tree的严重问题

- 插入效率低。特别是离散的插入。
- 每插入1条记录，涉及2次I/O操作。1 block的写入(1k?)
- 随着记录的增长，I/O负担越来越重。



B+Tree这么强，要不内存也用AVL?

- **AVL Tree内存的优势 > B+ Tree**
  - B Tree节点内部可能存在Scan
  - AVL查询优势最明显(log n, 也最平衡)
- **如果修改比查询重要。可以考虑4阶B-Tree(2-3-4 Tree) / Red-Black Tree(红黑树)**
  - 红黑树查询比AVL慢，但修改操作比AVL快。
  - 红黑树太复杂。
- **什么都想快，还有没有更好的算法。**

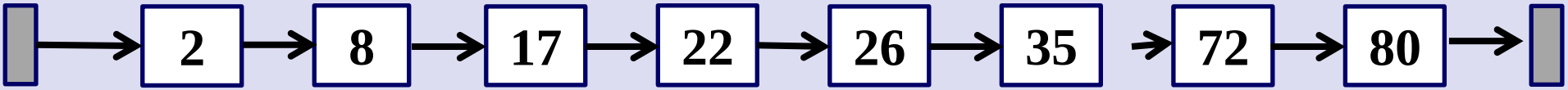
平衡树的问题

- **平衡树的插入问题：**
  - 自平衡需要大量开销：旋转、分裂等。
- **什么算法插入快？**
- **行为偏差**

对，就是被遗忘的

链 表

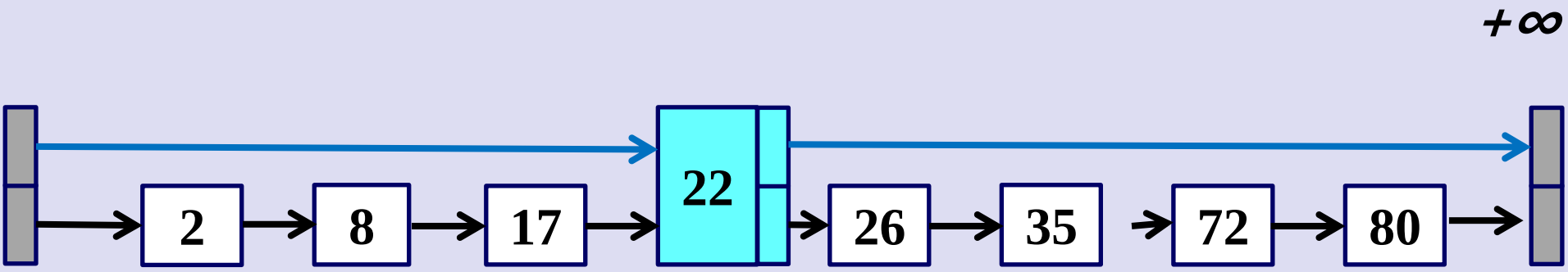
$+\infty$



链表：插入快、查找慢?

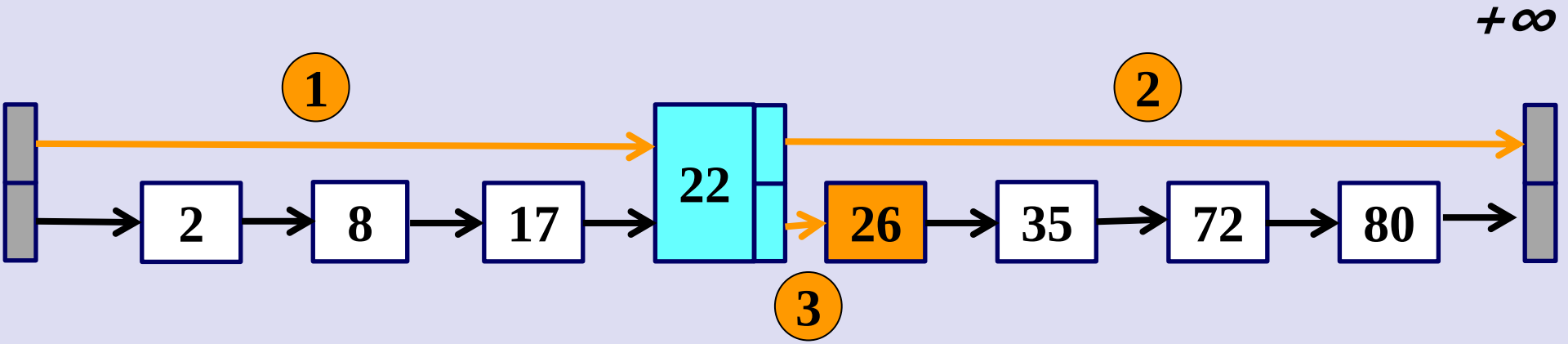
跳表(Skip List, 1989)

多了层链接到路牌，帮助跳过无用节点



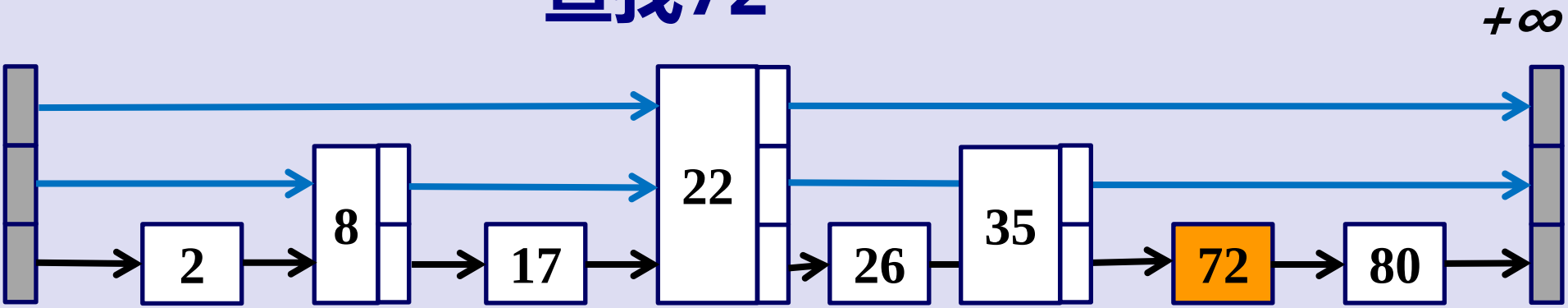
Skip List(查找)

- 查找26
- 3步



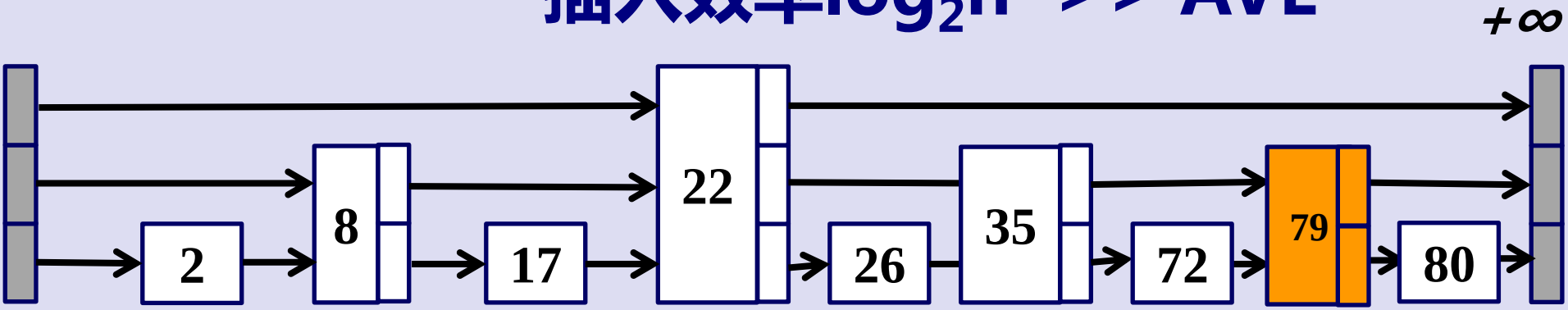
Skip List (查找)

- 2层（或多层)链接情况
- 查找效率$\log_2 n$ = AVL
- 查找72



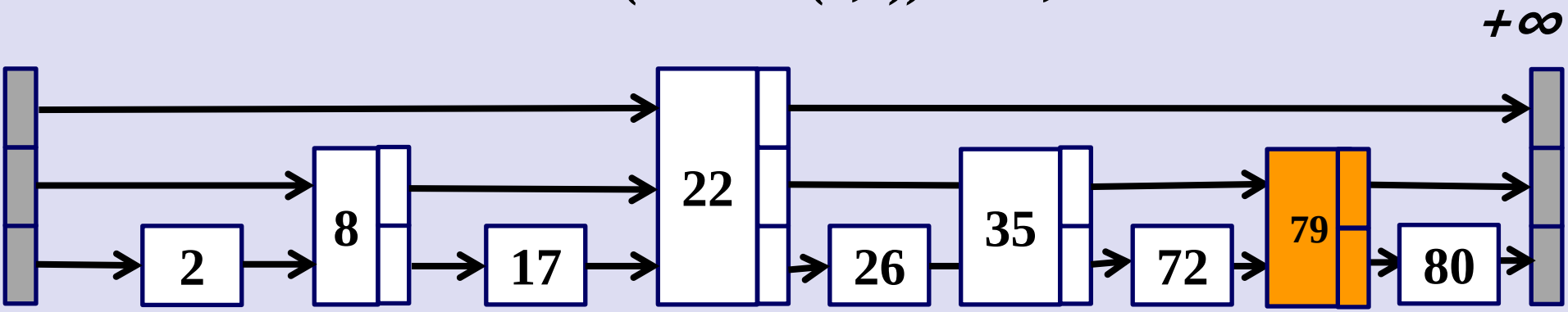
Skip list(插入)

- 插入79
- h = 2
- 插入效率$\log_2 n$ >> AVL

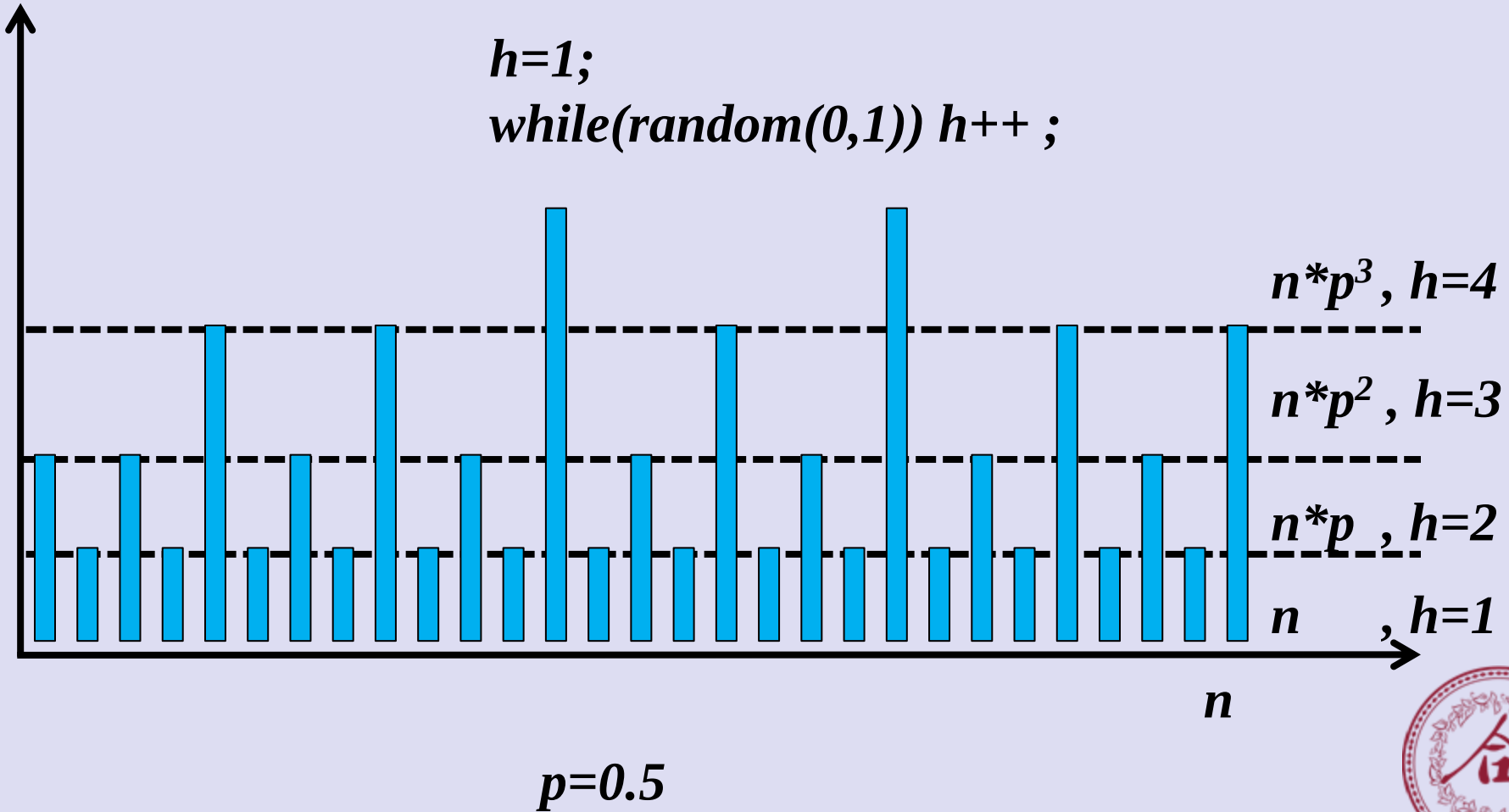


Skip list(插入节点的高度确定)

h=1;
while(random(0,1)) h++ ;



Skip list节点高度的概率分布



Skip List(应用)

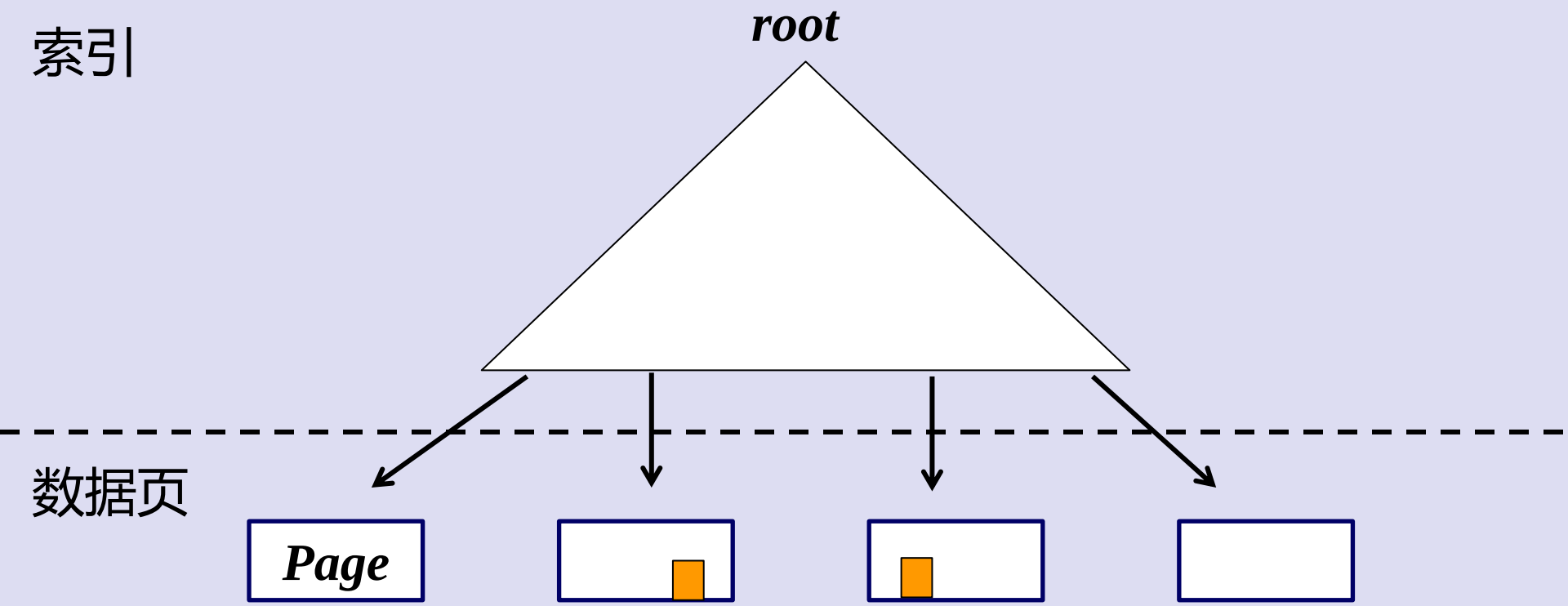
- **Redis**
  - zset(排序set)是个Hash Table + Skip List
- **LevelDB/BigTable**
  - By Jeffrey Dean(Google Fellows, 搞MapReduce和BigTable)
  - BigTable精华
  - 用Skip List做memtable表，取代AVL Tree
- **memSQL**
  - Facebook一帮人出来搞的，号称史上最快RDBMS
  - 全内存数据库，Skip List取代B+Tree和AVL Tree

Skip List(1989)

海量数据时代内存表！

硬盘算法还能改进吗？

回顾B+Tree问题



- 插入效率低。特别是离散的插入。
- 每插入1条记录，涉及2次I/O操作(R/W)。
  1 Page的写入(1k?)。插入100条，I/O=200
- 记录的增长，恶化明显。

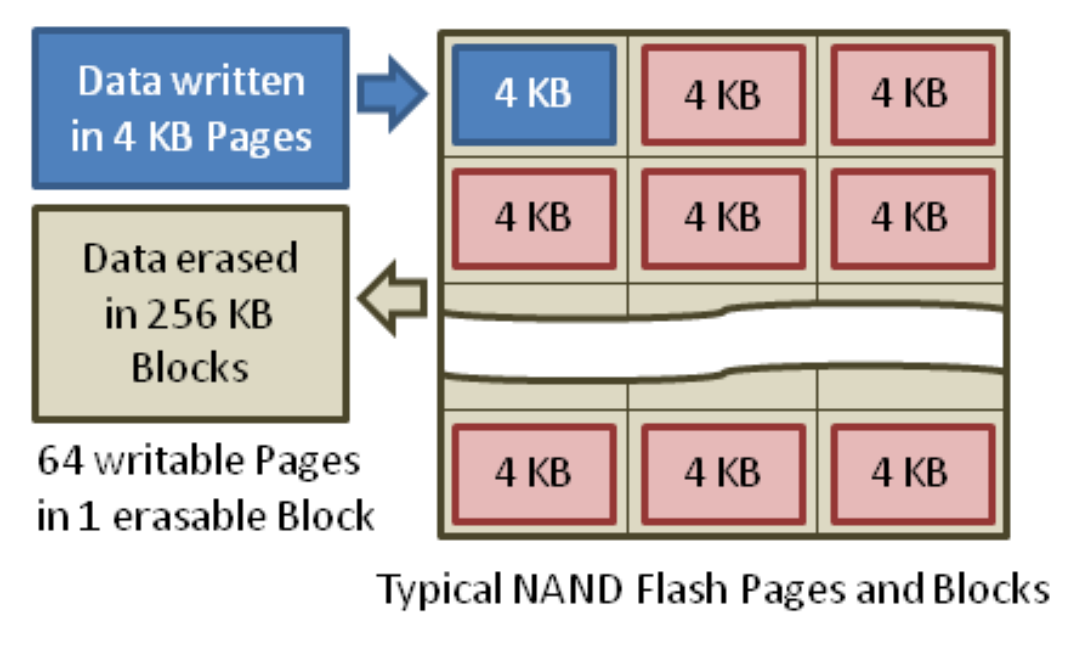
从机械硬盘到SSD

- **机械硬盘 – 200次随机IO/秒 ，IO次数是瓶颈**
- **SSD – 100k次随机IO/秒，IO次数不是瓶颈**

- **SSD**
  - SSD吞吐量 = IOps * 每次IO处理数据
  - 降低IOps并不能提高性能
  - 每次IO处理数据 是瓶颈

SSD写放大(WA)

- **WA (Write Amplification)，写放大**

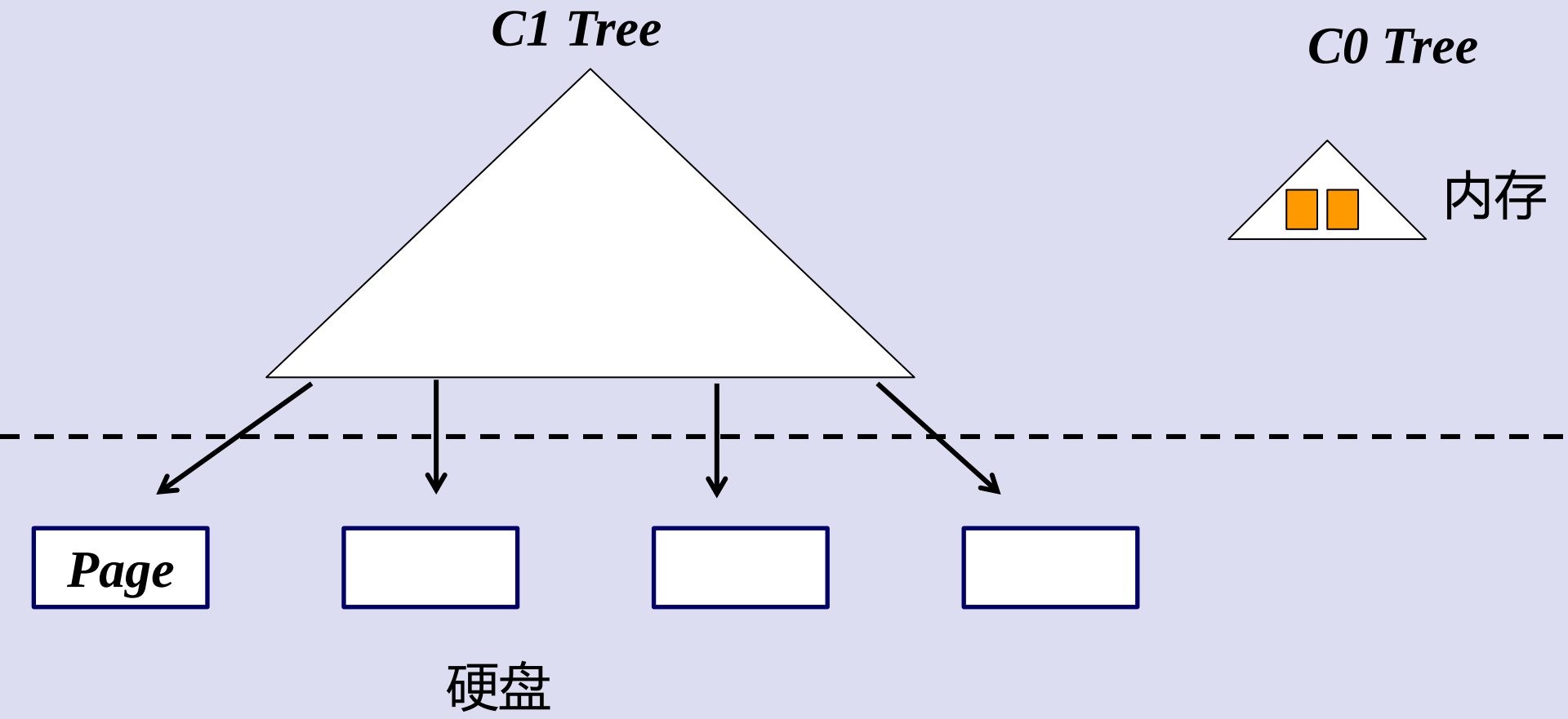


为了写4k的数据，必须整块256k擦除和写入
消耗大量带宽

SSD存储的核心问题

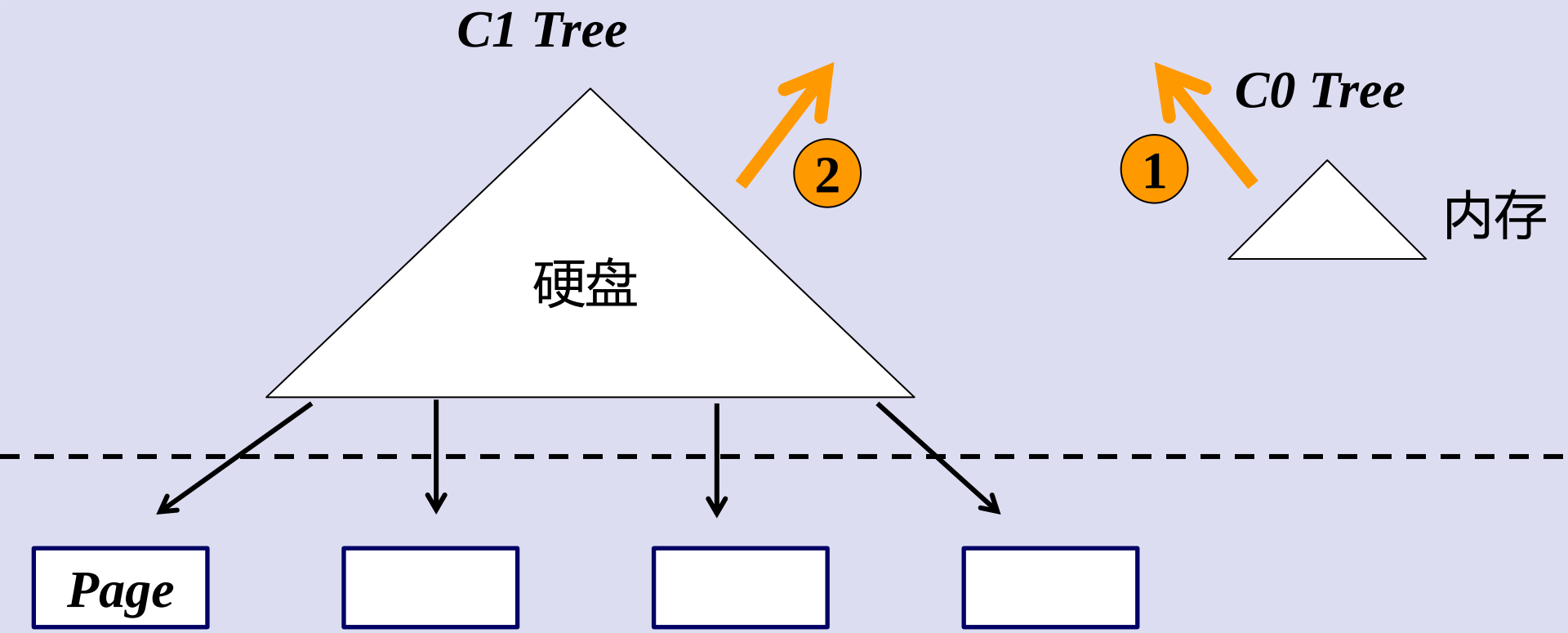
- **WA(Write Amplification)，写放大**
- **RA(Read Amplification)，读放大**
- **SA(Space Amplification)，存储放大**

LSM-Tree(1996)
(Log–Structured Merge-Tree)



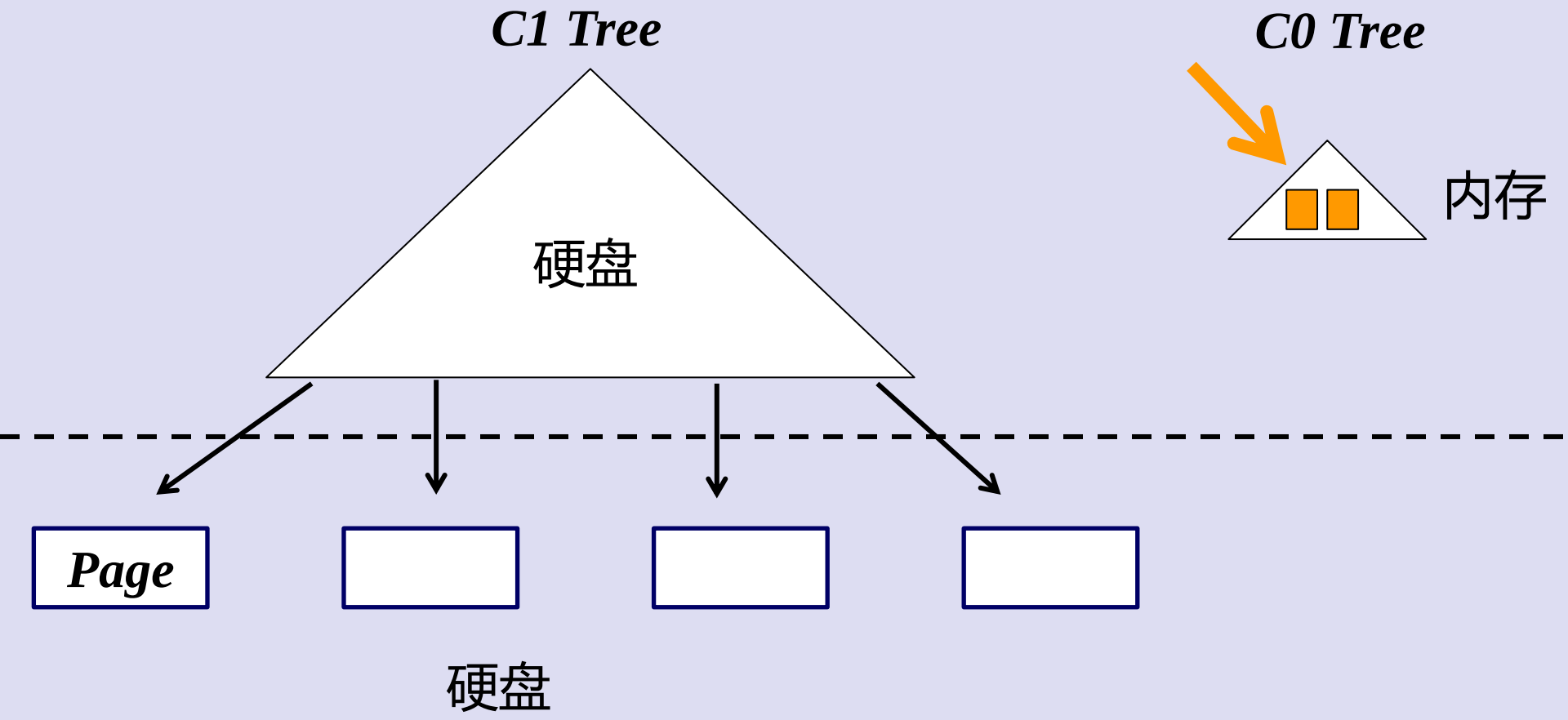
- **LSM-Tree: 是个写入优化的B+Tree**
- **先写入一个临时的小缓存C0**
- **再合并到硬盘C1中**

LSM-Tree(查询)



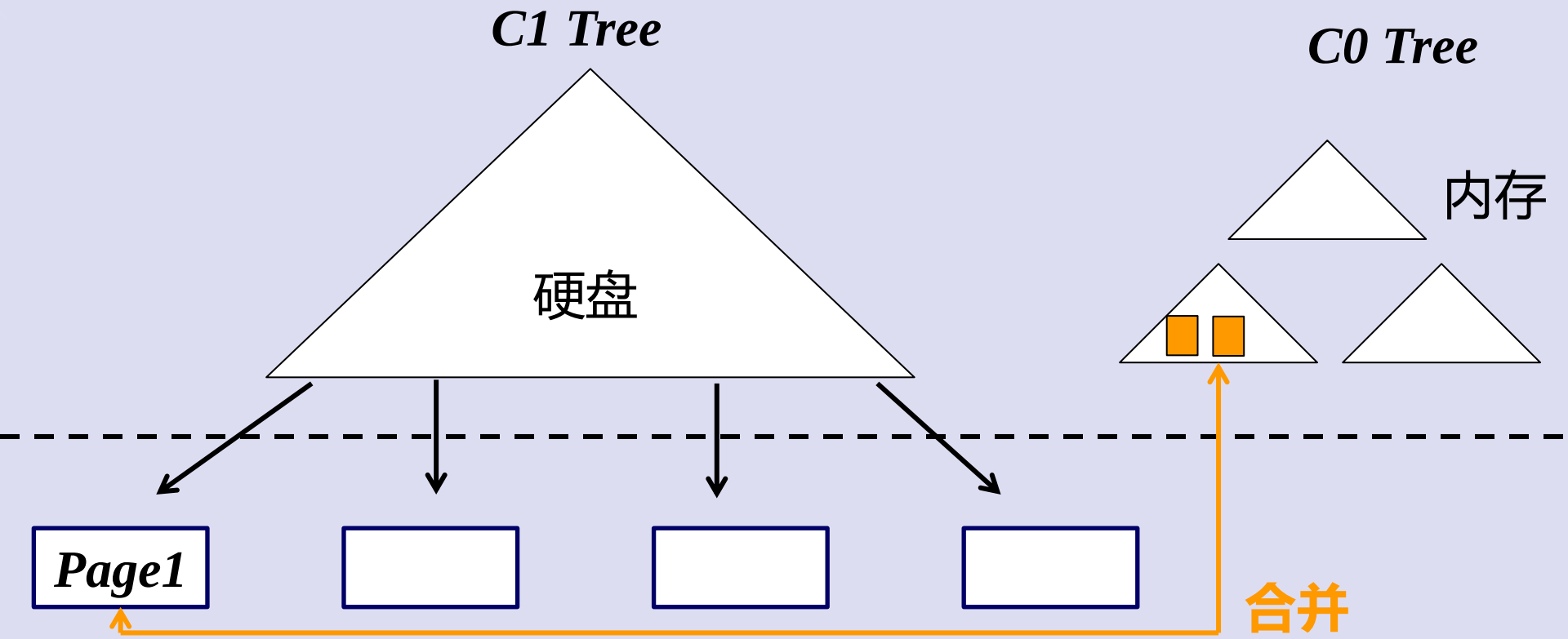
- 1. 查询C0
- 2. 如C0查不到，查询C1
- 新热通常在C0, C0相当于缓存

LSM-Tree(插入/修改)



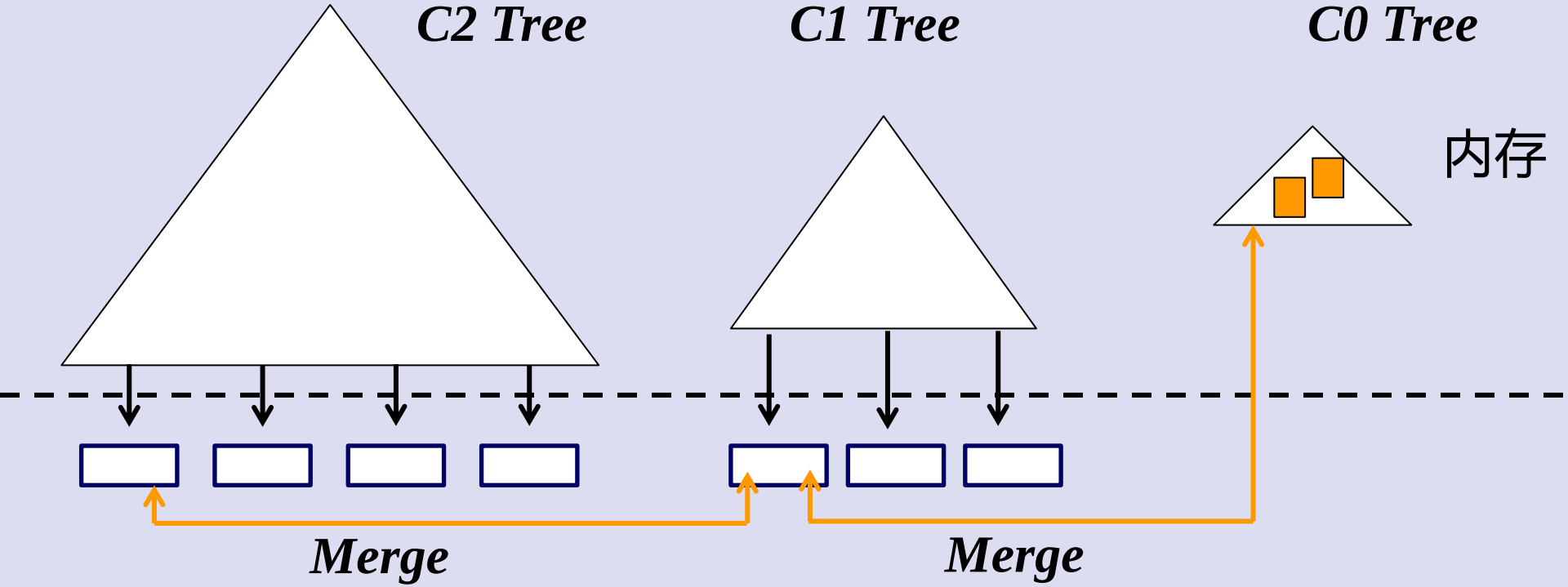
- **直接更新到C0 tree中**
- **0次磁盘I/O**

LSM-Tree(Merge)



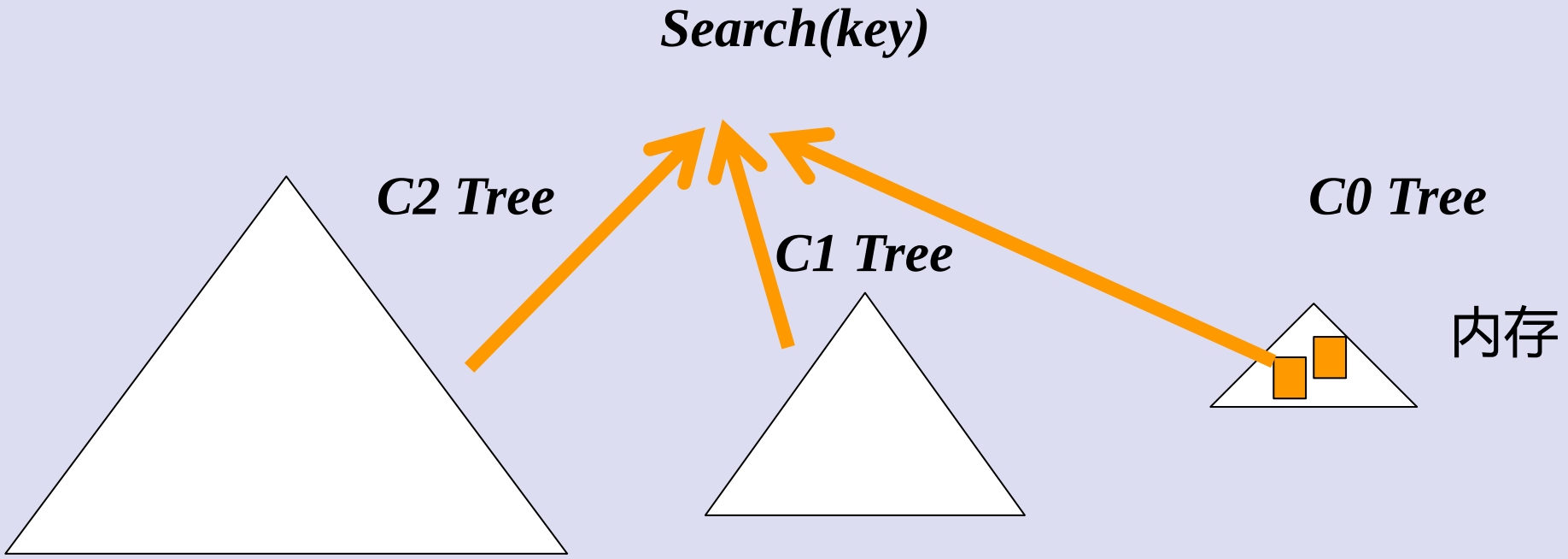
- C0部分同C1进行Page合并(Merge)
  - 如果C0 = C1 / 25 (C1是C0的25倍大),
  - 如果C1 Page有250条记录
  - 则每次合并一个C1 Page，可能包含250/25=10条C0记录
  - I/O次数减少20倍(10 x 2RW)
- 如果C1比C0大太多，合并命中的可能性会严重降低，于是：

LSM-Tree(多组件情况)



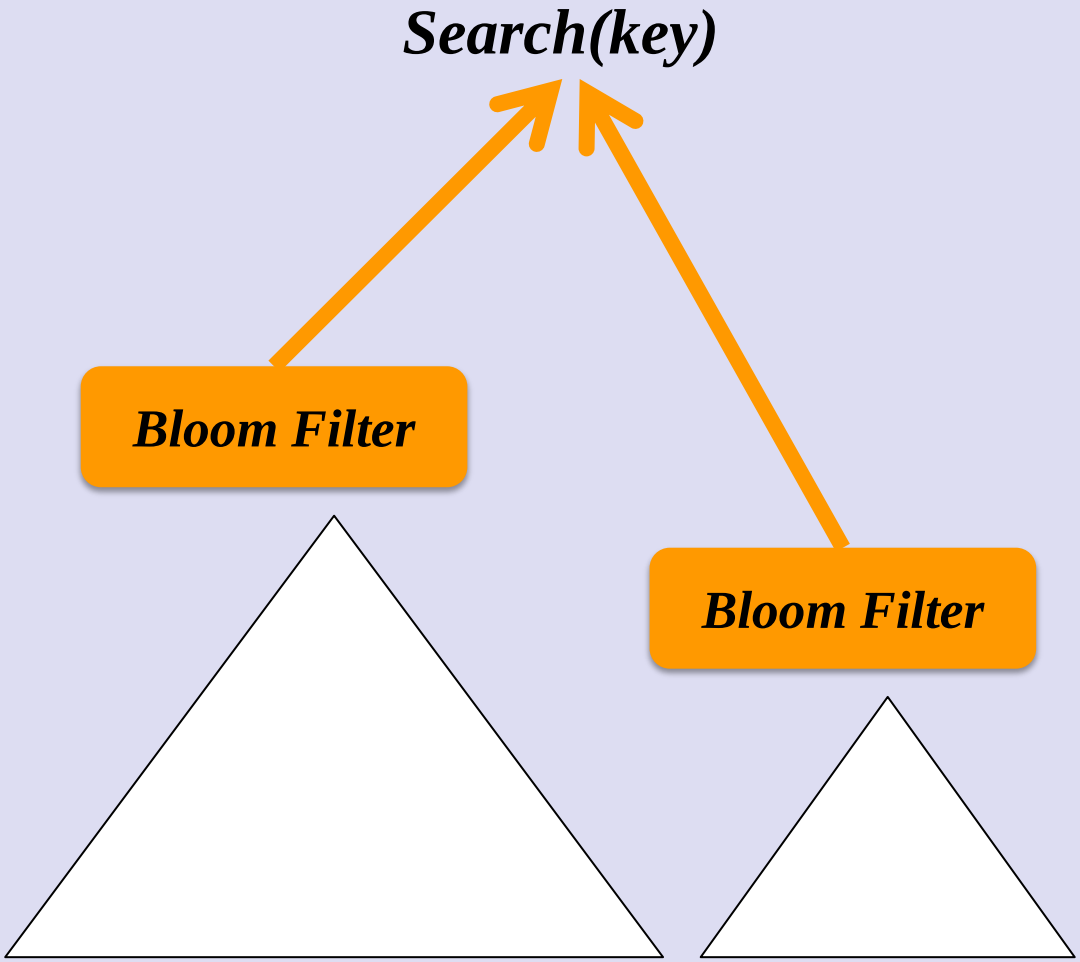
- 中间增加一组(或多组)大小适中的Tree
- C0 比 C1小， C1比C2小
- C0满了同C1合并
- C1满了同C1合并
- ……
- 查询降低，Merge效率提高

LSM-Tree(RA读放大)



- RA(Read Amplification)，
  哪怕读取1byte也涉及硬盘多个Block读取。
- LSM需要从多个Tree中查找数据，
  存在RA(读放大)问题
- 读的效率比Btree低

LSM用Bloom Filters降低RA



- Bloom Filter减少命中硬盘
- 仅能优化点查询

LSM-Tree典型案例

- **LevelDB**
  - C0用Skip List
  - LSM-Tree
- **RedisDB**
  - 优酷开发的kv持久化引擎，用于播放云记录
  - RedisDB = Redis + LevelDB
- **RocksDB – Facebook核心引擎(优化ssd),MySQL Engine:MyRocks, Mongo:MongoRocks**
- **MongoDB 3.0(WiredTiger)**
- **BigTable**
- **HBase**
- **Cassandra**
- **SQLite**

未来希望看到的项目

- **技术进步其实很缓慢(LSM-Tree 1996)**
- **MySQL支持LSM-Tree的开源引擎**
  - 高性能，no事务。(like MyISAM)
  - LevelDB for MySQL ? – MyRocks
- **高性能kv数据库**
  - RedisDB(优酷持久化kv引擎) ?- RocksDB
- **用skip list高性能分布式内存开源数据库 (memSQL收费)**

什么最快? IT技术进步了吗？

数组(Array)		
链表(Linked list)		
二叉树(Binary Tree)		
平衡树(AVL Tree)	1962	
伸展树(Splay Tree)	1985	Squid3
ISAM	1964	Foxpro/dBase,MS Access, MySQL 3.23前
红黑树(Red-Black Tree) 2-3-4 Tree	1972	
B+树(B+Tree)	1979	MyISAM, Innodb, Oracle, DB2, SQL Server, Berkeley DB, Mongodb, HBase, TokyoCabinet/QDBM
跳表Skip List	1989	Redis,LevelDB/BigTable,memSQL,nessdb
LSM树(LSM-Tree)	1996	LevelDB, RedisDB, BigTable, HBase, Cassandra, nessdb

还能更快吗？

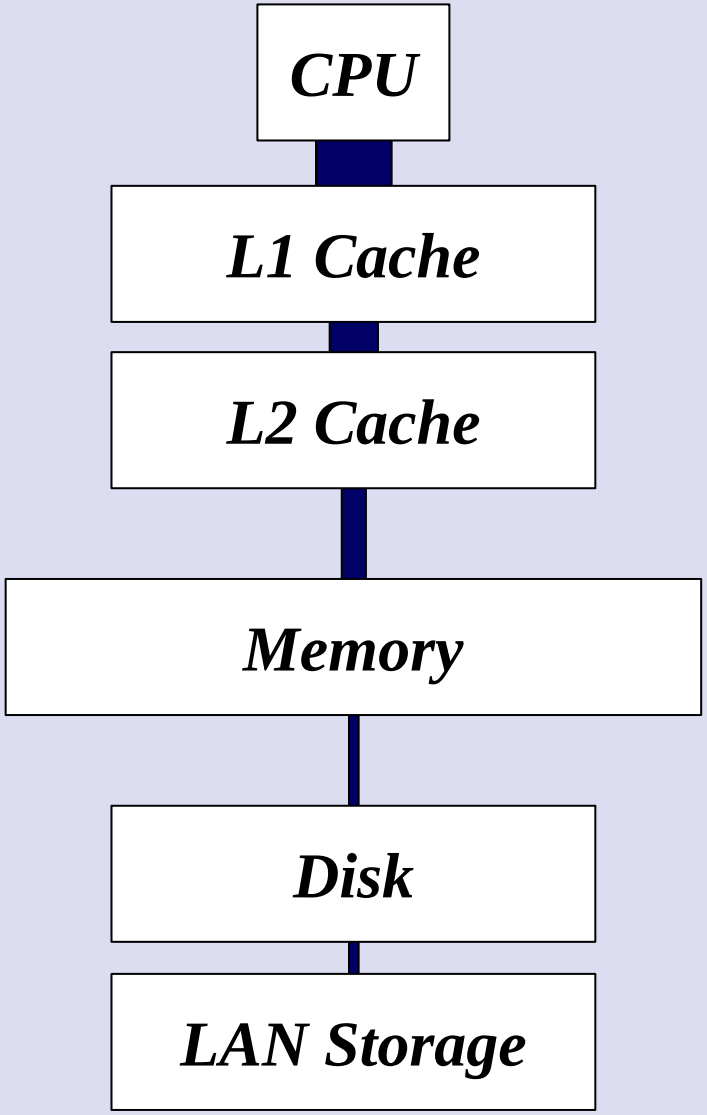
问题？

- 200个数的查找，上面给出的几个算法哪个最快？
- 下面两段代码，结果一样，哪个快？

```
for i in 0..n
  for j in 0..m
    for k in 0..p
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

```
for i in 0..n
  for k in 0..p
    for j in 0..m
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

存储层次结构



- 越靠近CPU越快(1)
- L1 - 桌子上(3)
- L2 - 厅里(10)
- Mem - 小区口(100)
- Disk - 公司(10M)

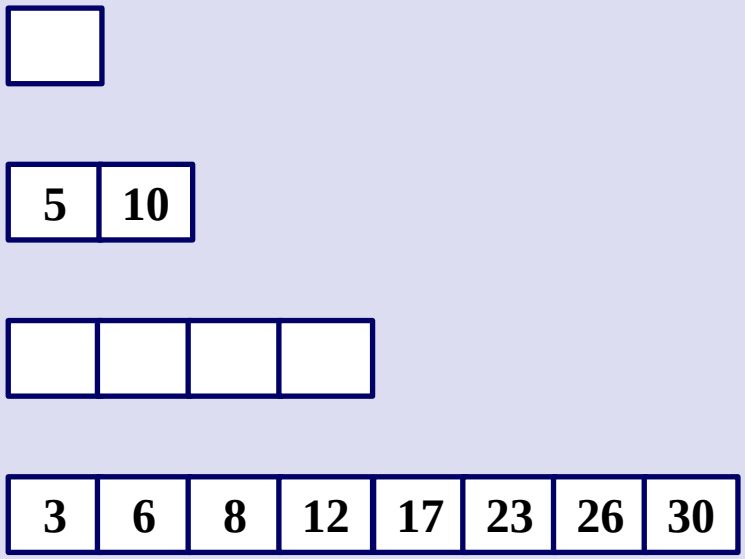
- 想想家里如何有效倒垃圾
  - 先丢在桌上(insert)
  - 满了再一层层移走(merge)
    书桌 →厅→小区→ 社区→ …→北京垃圾处理中心
  - 不用考虑每层垃圾桶的大小

Cache-Oblivious Algorithms
(缓存参数无关算法, 1999)

- 1. 存在多级缓存(L1, L2, Mem, FlashMem, Disk....)
- 2. 无需知道各级缓存大小
- 根据以上条件设计算法，能够自动适应各级缓存，尽可能优化计算。

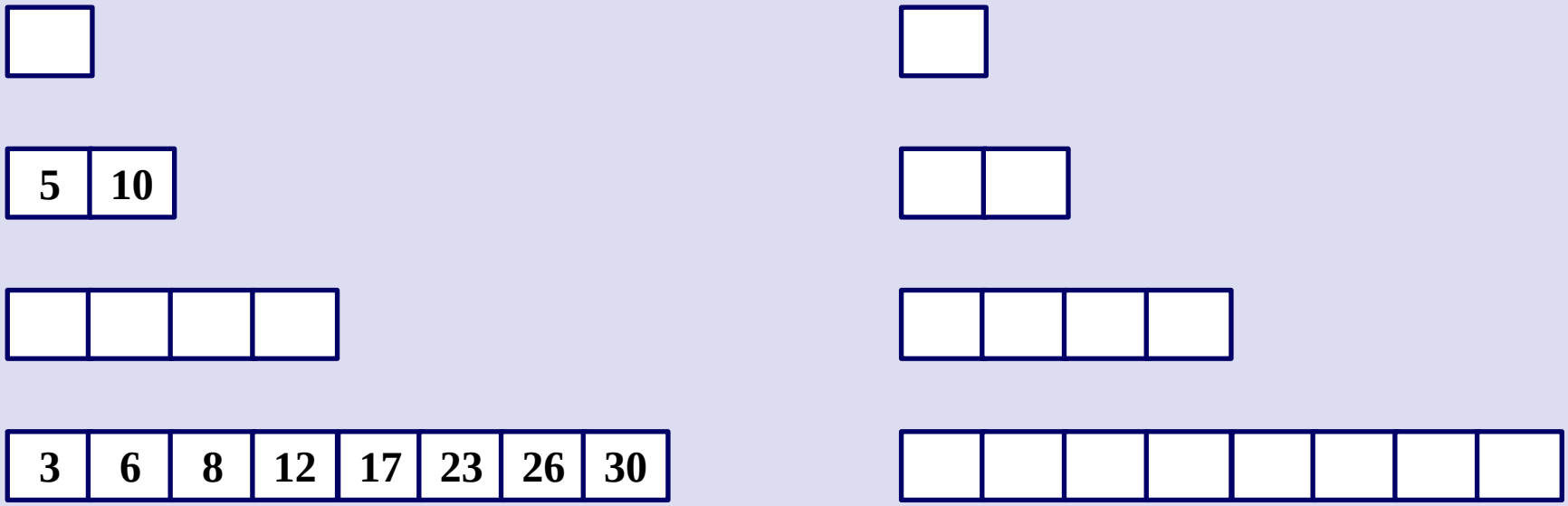
- 想想看当前的LSM Tree/B+Tree等算法，固定page block大小，白白浪费了L1, L2等资源。相当于每次丢垃圾都跑到小区楼下的垃圾桶。

Fractal Tree 分形树(2007)
又叫COLA(Cache-Oblivious Lookahead Array)



- Log N个数组
- 2倍增大
- 每个数组要满，要么空
- 每个数组排好序

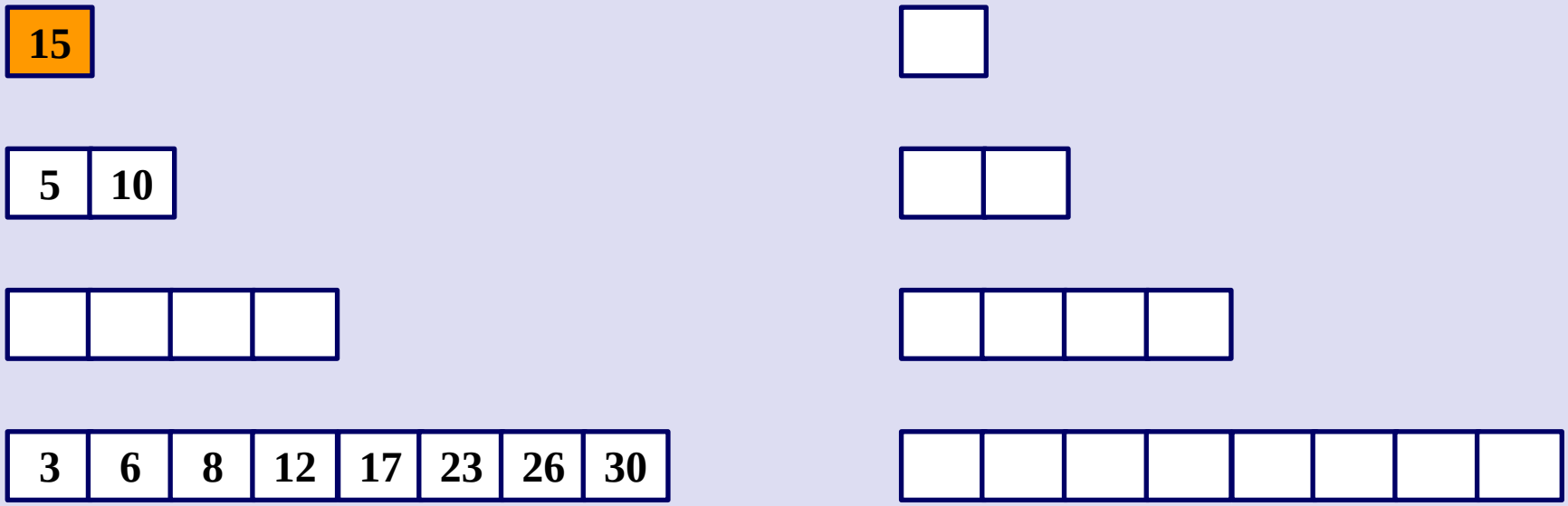
Fractal Tree(插入)



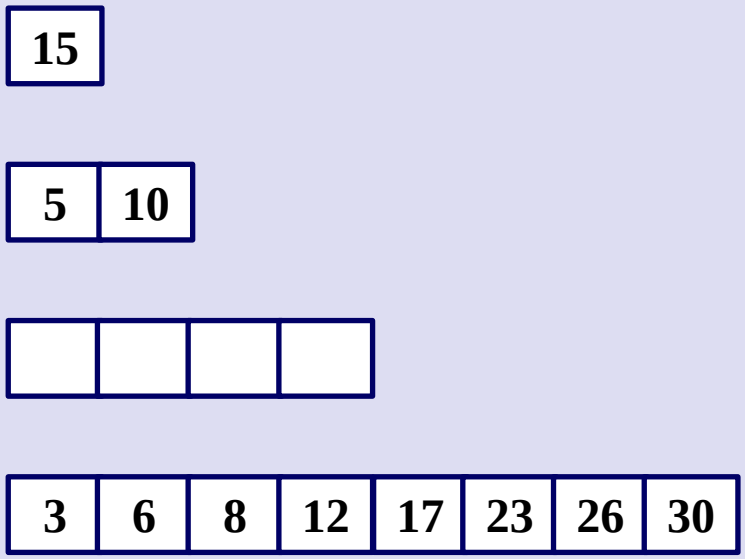
数组

临时数组

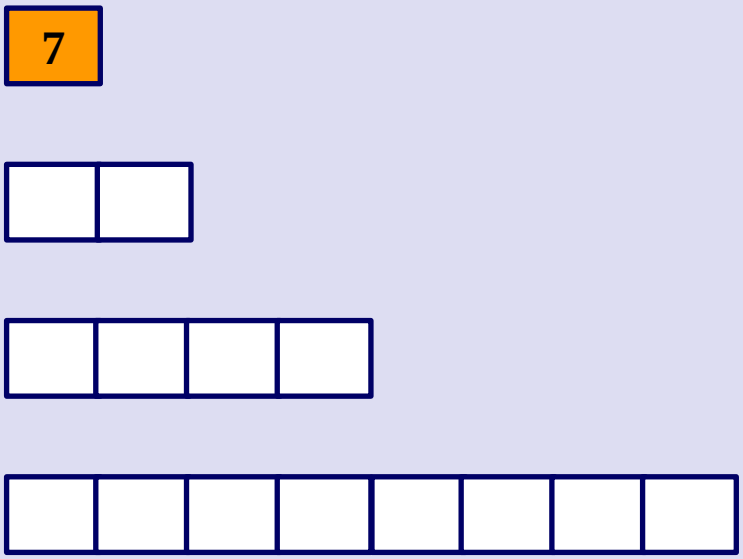
Fractal Tree(插入15)



Fractal Tree(插入7)



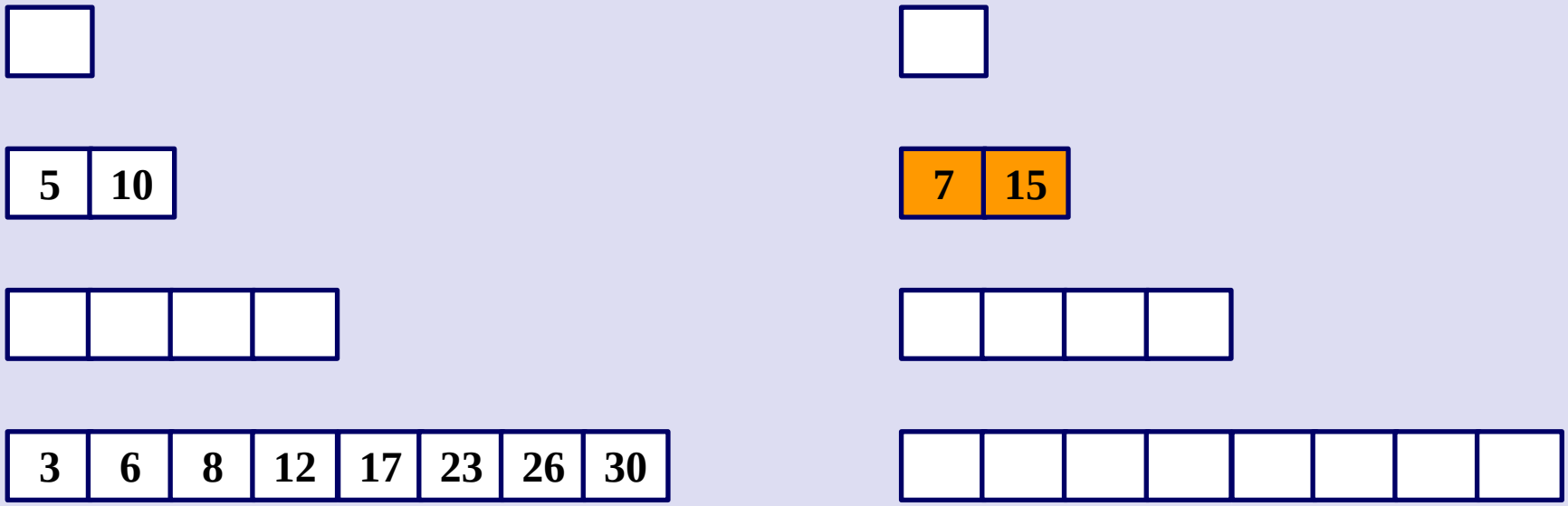
数组



临时数组

Fractal Tree(插入7)

合并[7], [15]

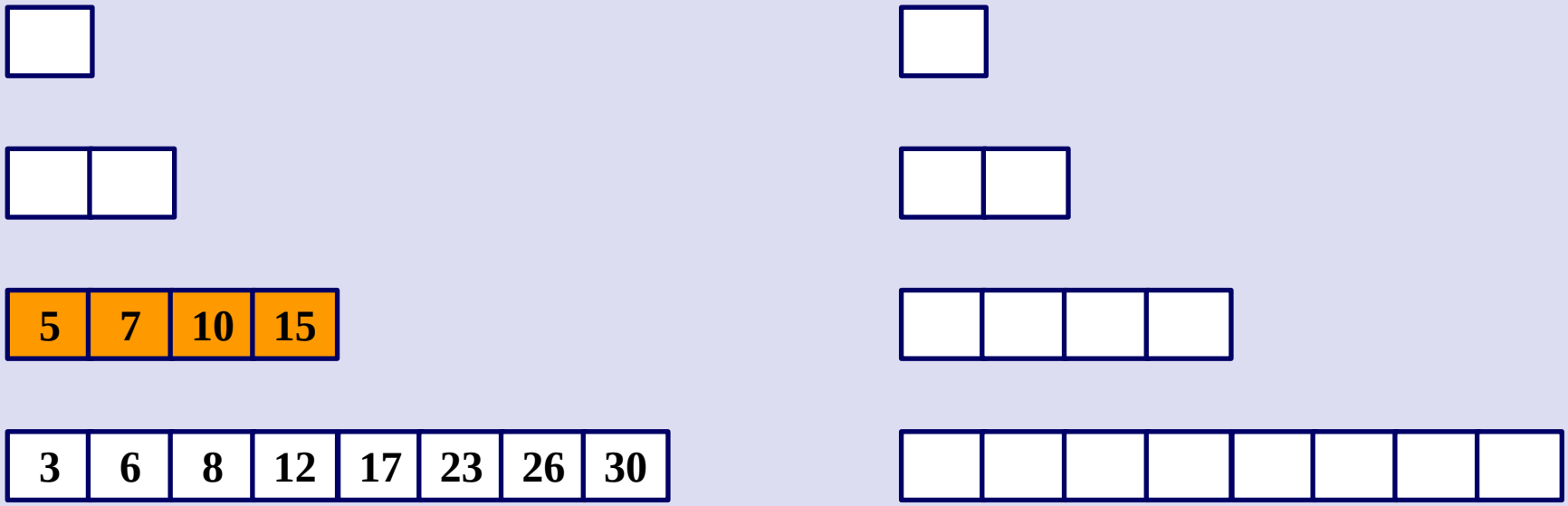


数组

临时数组

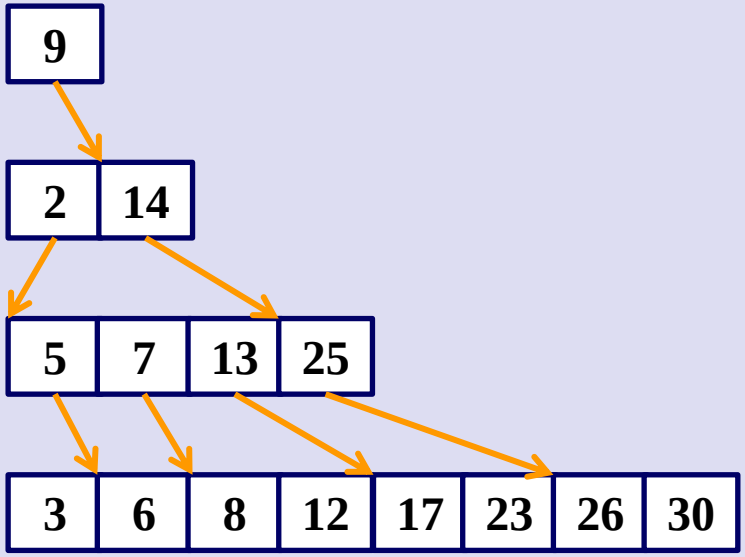
Fractal Tree(插入7 – 完成)

合并 [5,10], [7,15]



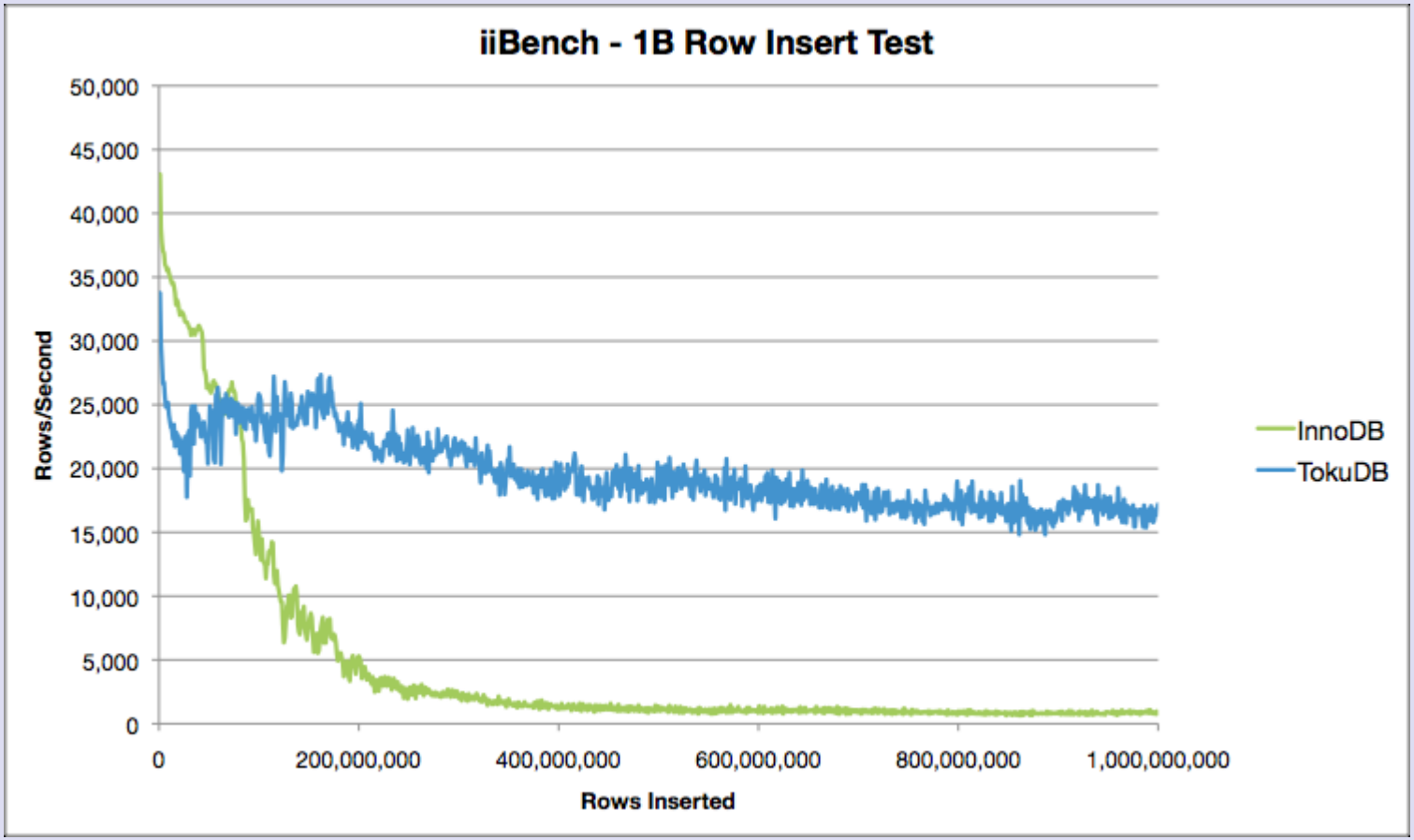
- **小块内存中合并(充分利用L1, L2)**
- **大块硬盘中合并**

Fractal Tree(查询优化)



查询时间Log_2 N ＜ LSM Tree

TokuDB Engine的性能对比(Fractal Tree)



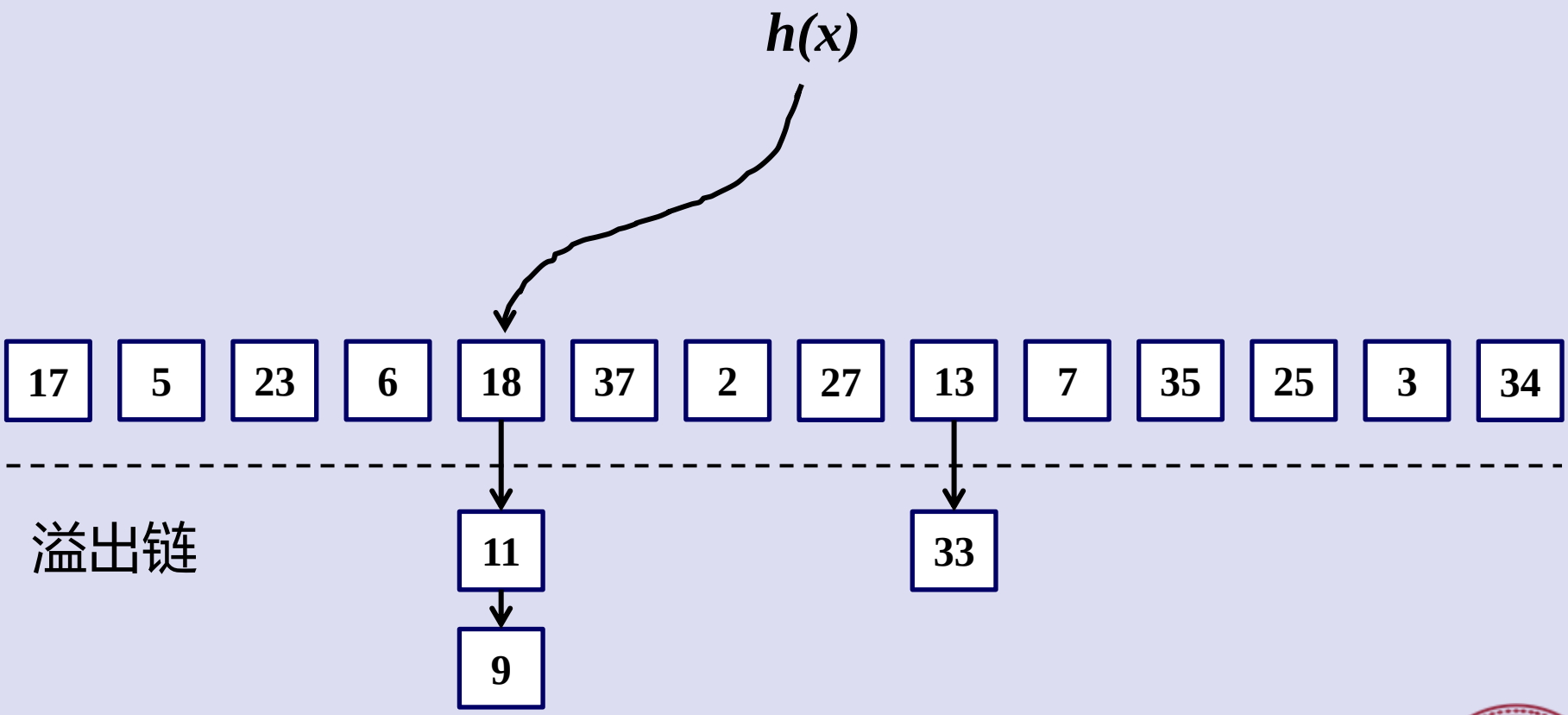
10x ~ 100x on insert, FT vs Btree

Btree vs LSM vs FT

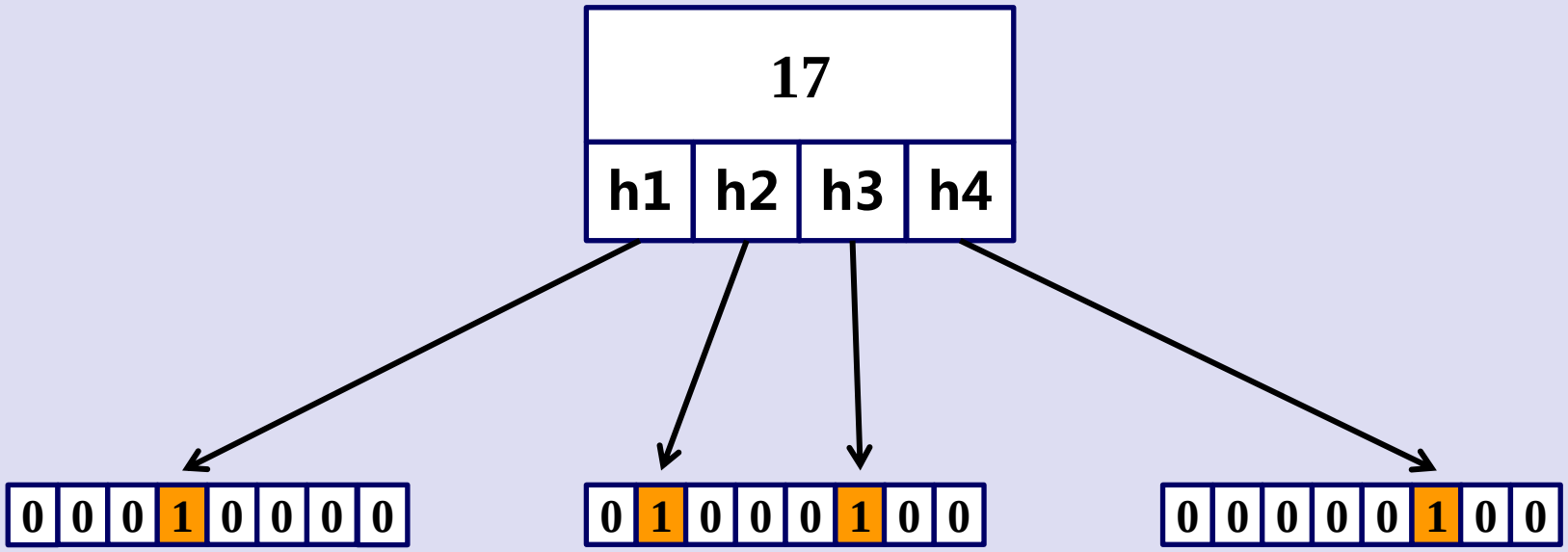
	Read	Write	产品
Btree	好	中	InnoDB ...
FT Index	好	好	TokuDB
LSM Tree	中 (可BloomFilter优化point查找)	好	LevelDB, RocksDB, MongoDB 3.0, BigTable, HBase, Cassandra

点查询(Point Lookup)

Hash查找



Bloom Filter(布隆过滤器, 1970)



- 可以极大节约空间
- 用多个Hash函数散列到不同位上
- 几个Hash函数全部命中的概率极低
- Hash函数越多，概率越低
- 应用：∃函数。email是否存在等

Hash函数选用

- crc32
- md5
- djb2 ,djb3
- sdbm
- fnv-1, fnv-1a
- Murmur(效率高, 2010, hadoop/nginx)
- CityHash(2011, SSE64优化过，长串比murmur快，短比murmur慢)

各Hash函数速度和冲突

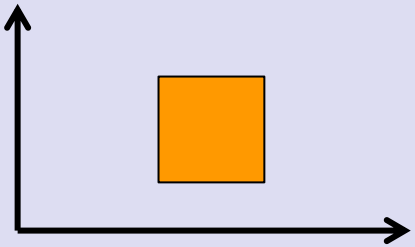
Hash	Lowercase	Random UUID	Numbers
Murmur	145 ns	259 ns	92 ns
	6 collis	5 collis	0 collis
FNV-1a	152 ns	504 ns	86 ns
	4 collis	4 collis	0 collis
FNV-1	184 ns	730 ns	92 ns
	1 collis	5 collis	0 collis*
DBJ2a	158 ns	443 ns	91 ns
	5 collis	6 collis	0 collis***
DJB2	156 ns	437 ns	93 ns
	7 collis	6 collis	0 collis***
SDBM	148 ns	484 ns	90 ns
	4 collis	6 collis	0 collis**
CRC32	250 ns	946 ns	130 ns
	2 collis	0 collis	0 collis

三类查询问题

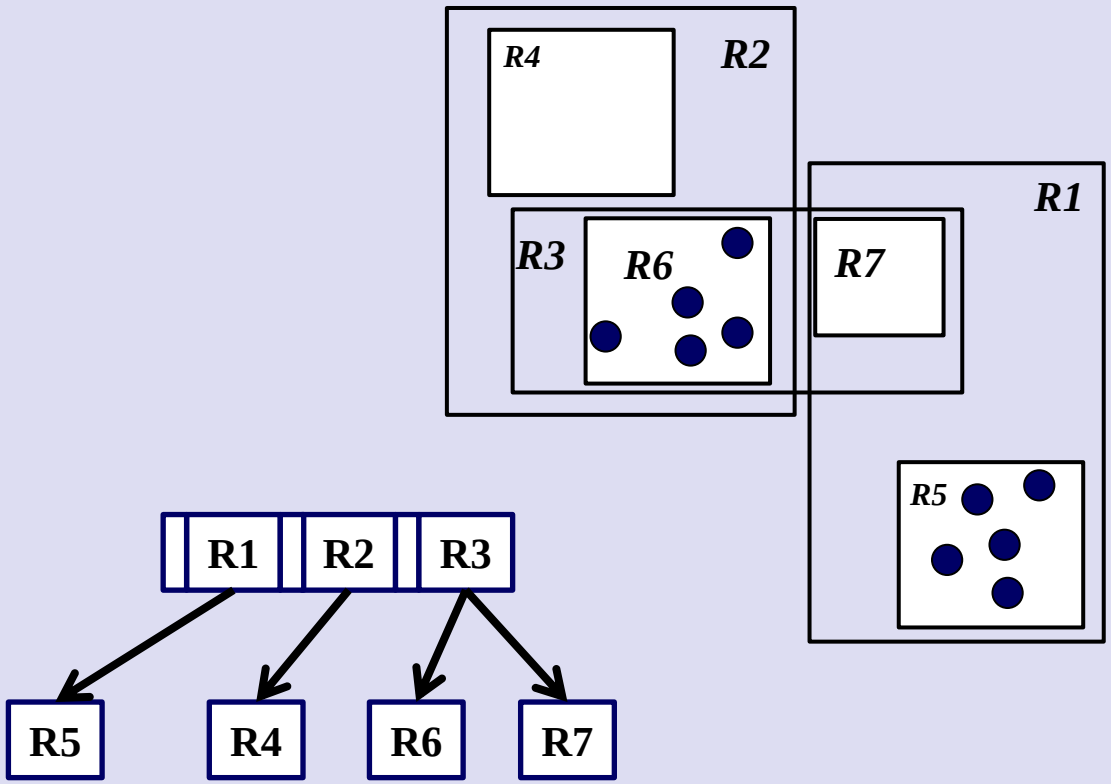
- **0维查询 - 点查询(Point Query)**

- **1维查询 - 范围查询(Range Query)**
  - (1000~2000)

- **多维查询 - 空间查询(Multi-Dimensional Query)**
  - 2个或多个组合
  - x=100~300 and y=4000~5000
  - 附近300米的便利店
  - 离地球最近的三个恒星

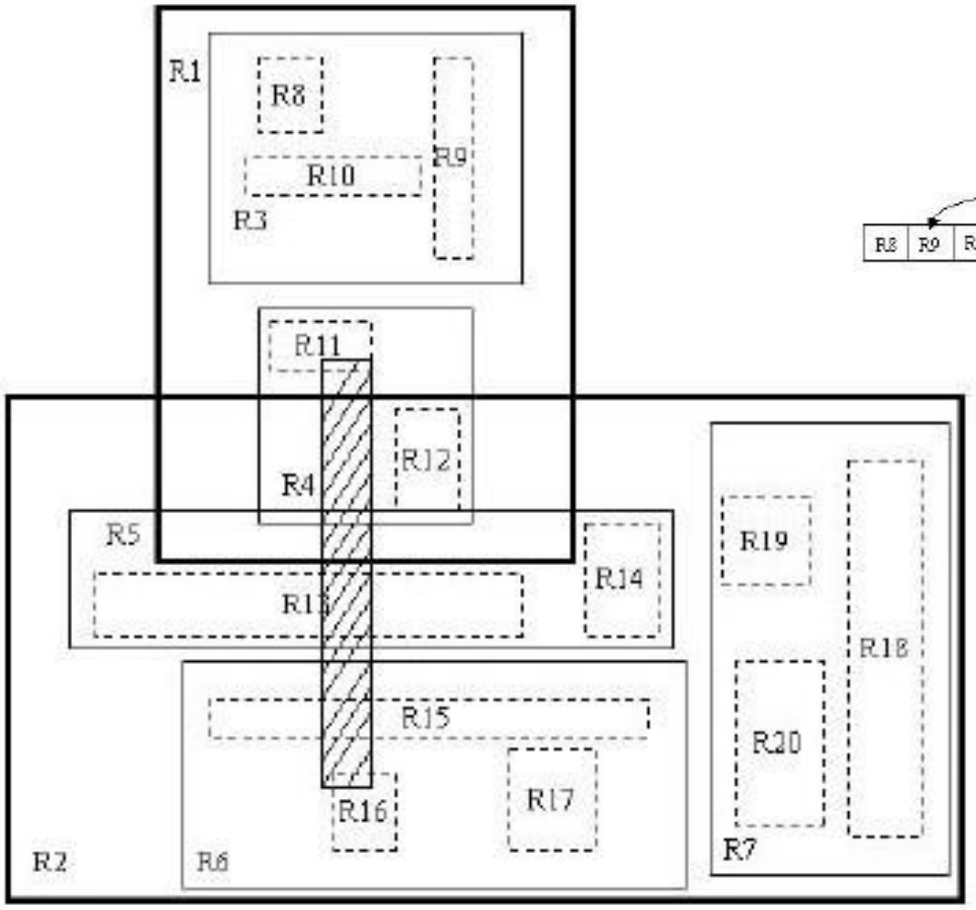


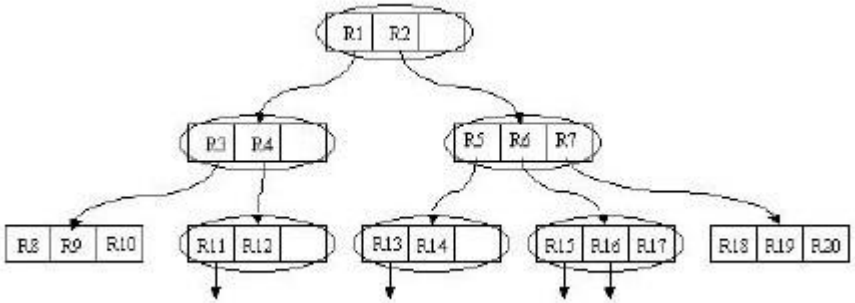
多维查询之R-Tree(R树, 1984)



- 如同世界地图
- 划分好区域
- 索引高层区域越大

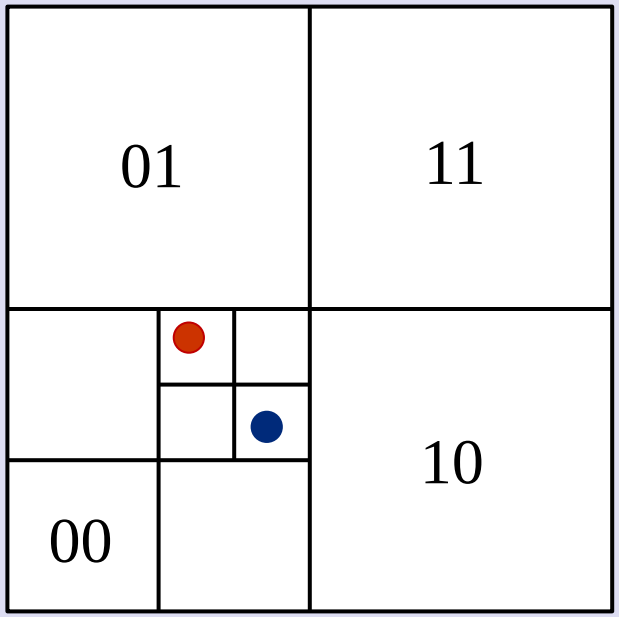
R-Tree





计算阴影部分的内容?
R11, R13, R15, R16

Mongodb空间索引
(Mongodb GeoSpatial Index)

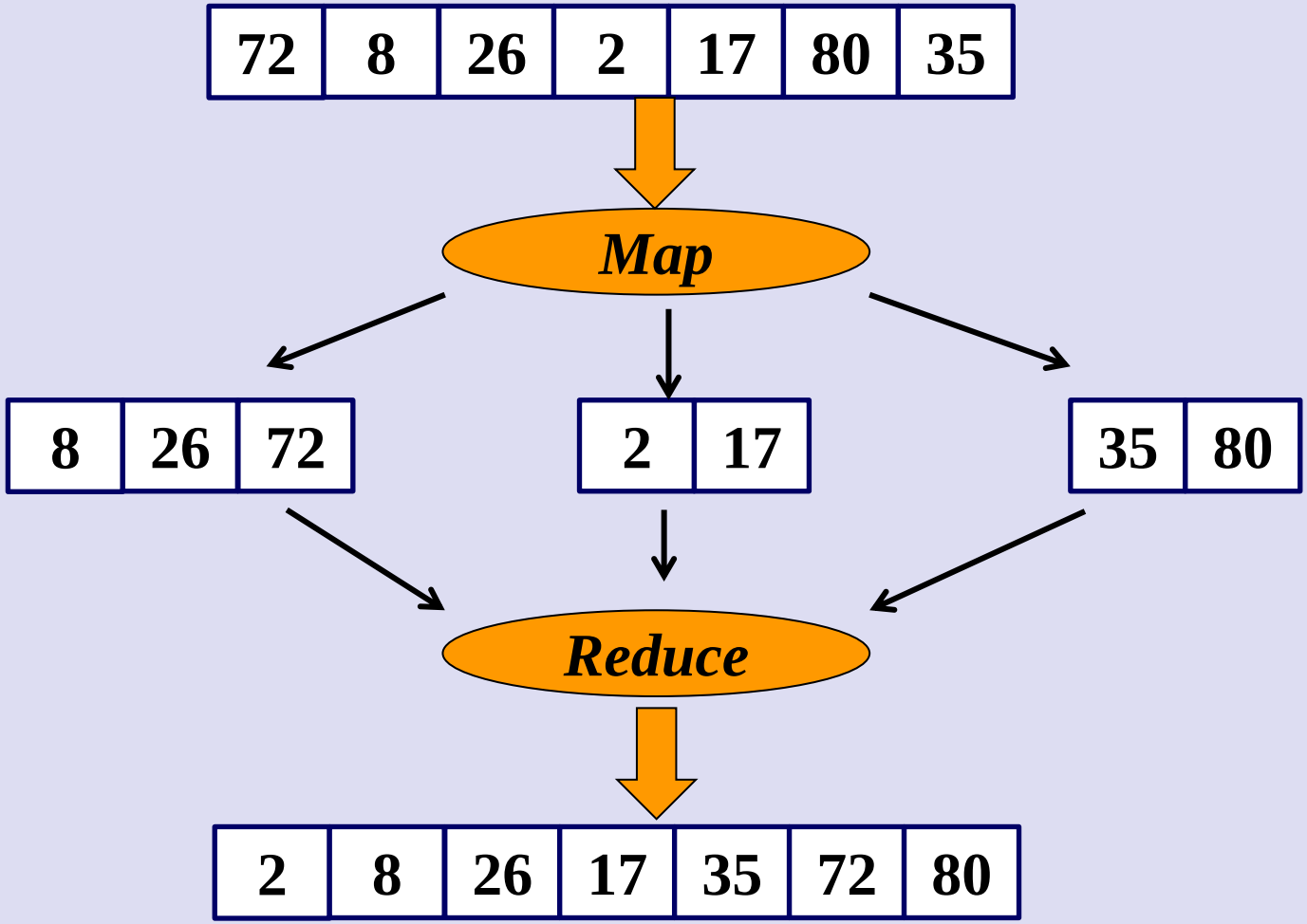


- **规则的空间划分**
- **每层4个象限**
  01 11
  00 10

- ***001101***
- ***001110***

相近点前缀相同

MapReduce



MapReduce的问题

- 蛮力代替索引
- Range类查询对比索引方案代价过高
- 一次查询全部参与计算，代价过高
- 适合索引无法达到的复杂查询

总结：点查询(Point Lookup – 0维)

- **哈希表Hash Table**
- **各种哈希函数**
- **布隆过滤器Bloom Filter**

总结：范围查询(Range Query – 1维)

数组(Array)		
链表(Linked list)		
二叉树(Binary Tree)		
平衡树(AVL Tree)	1962	
伸展树(Splay Tree)	1985	Squid3
ISAM	1964	Foxpro/dBase,MS Access, MySQL 3.23前
红黑树(Red-Black Tree) **2-3-4 Tree**	1972	
B+树(B+Tree) **B树/B*树**	1979	MyISAM, Innodb, Oracle, DB2, SQL Server, Berkeley DB, Mongodb, HBase, TokyoCabinet/QDBM, XFS
跳表Skip List	1989	Redis,LevelDB/BigTable,memSQL
跳舞树(Dancing Tree)	200x	Reiser4
LSM树(LSM-Tree)	1996	LevelDB, RedisDB, BigTable, HBase, Cassandra, MongoDB 3, RocksDB
分形树 Fractal Tree(COLA)	2000, 2007	TokuDB

总结：多维查询(2D/3D...)

- **R-Tree(1984)**
- **K-d Tree (1975)**
- **Mongodb GeoSpatial Index/GeoHash**

Q & A

谢谢！