大数据计算基础

BIG DATA COMPUTING

刘显敏 海量数据
liuxianmin@hit.edu.cn

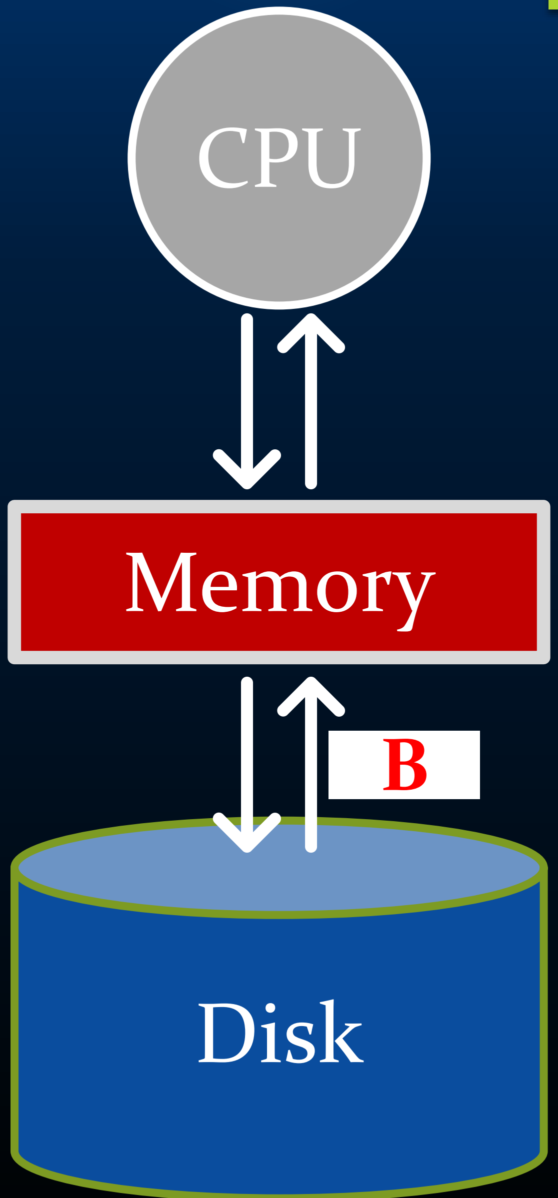
大数据算法

2025年秋

外存算法

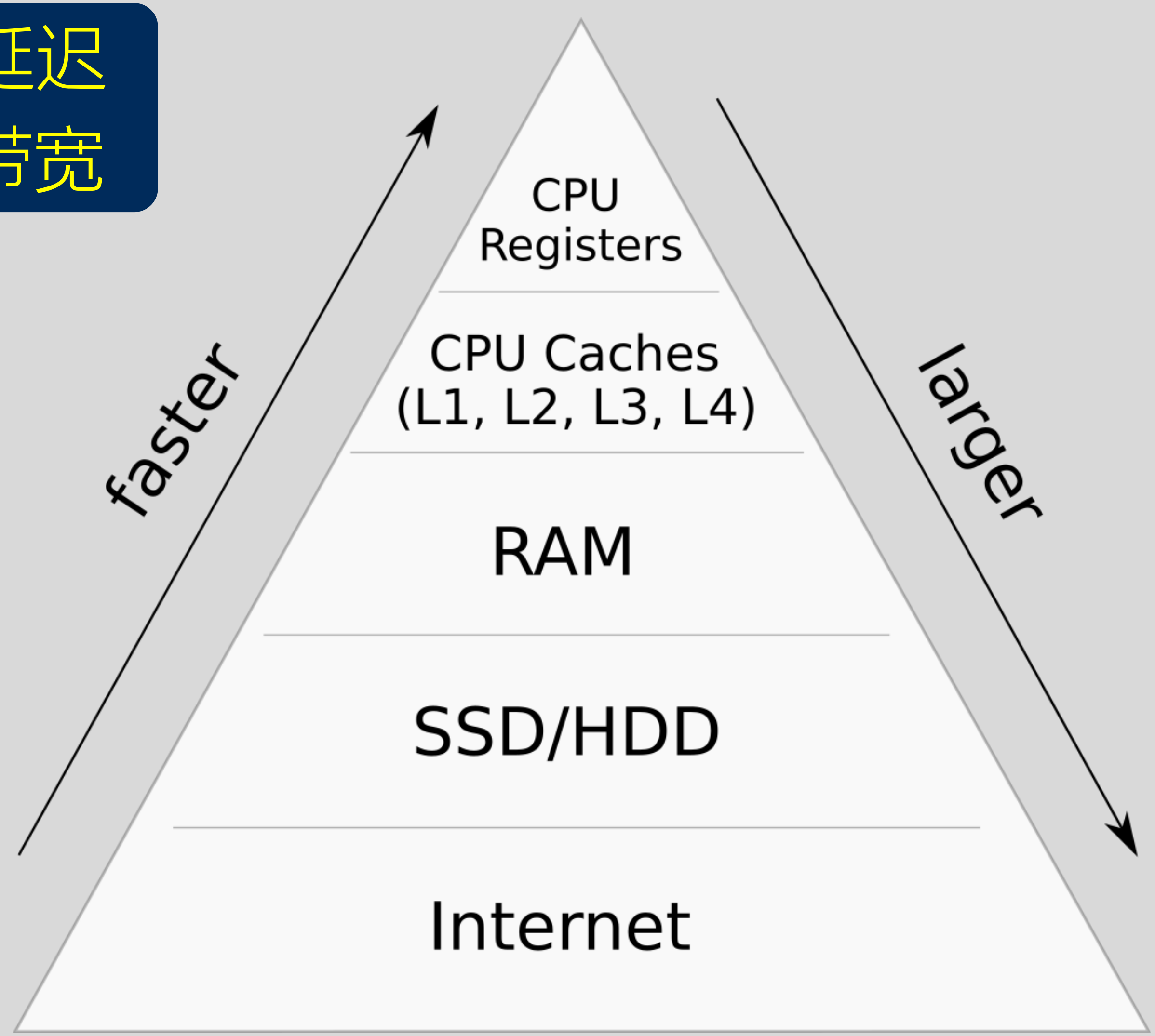
External Memory ||

I/O Algorithm



存储的层次化结构

M B 延迟
价格 带宽



存储的层次化结构

存储器	M	B	延迟	带宽	价格
L1	10K	64B	2ns	80G/s	-
L2	100K	64B	5ns	40G/s	-
L3	1M/core	64B	20ns	20G/s	-
RAM	GBs	64B	100ns	10G/s	1.5
SSD	TBs	4K	0.1ms	5G/s	0.17
HDD	TBs	-	10ms	1G/s	0.04
S3	∞	-	150ms	∞	0.02

存储的层次化结构

- 考虑一个简单情况：*c=*a+*b;

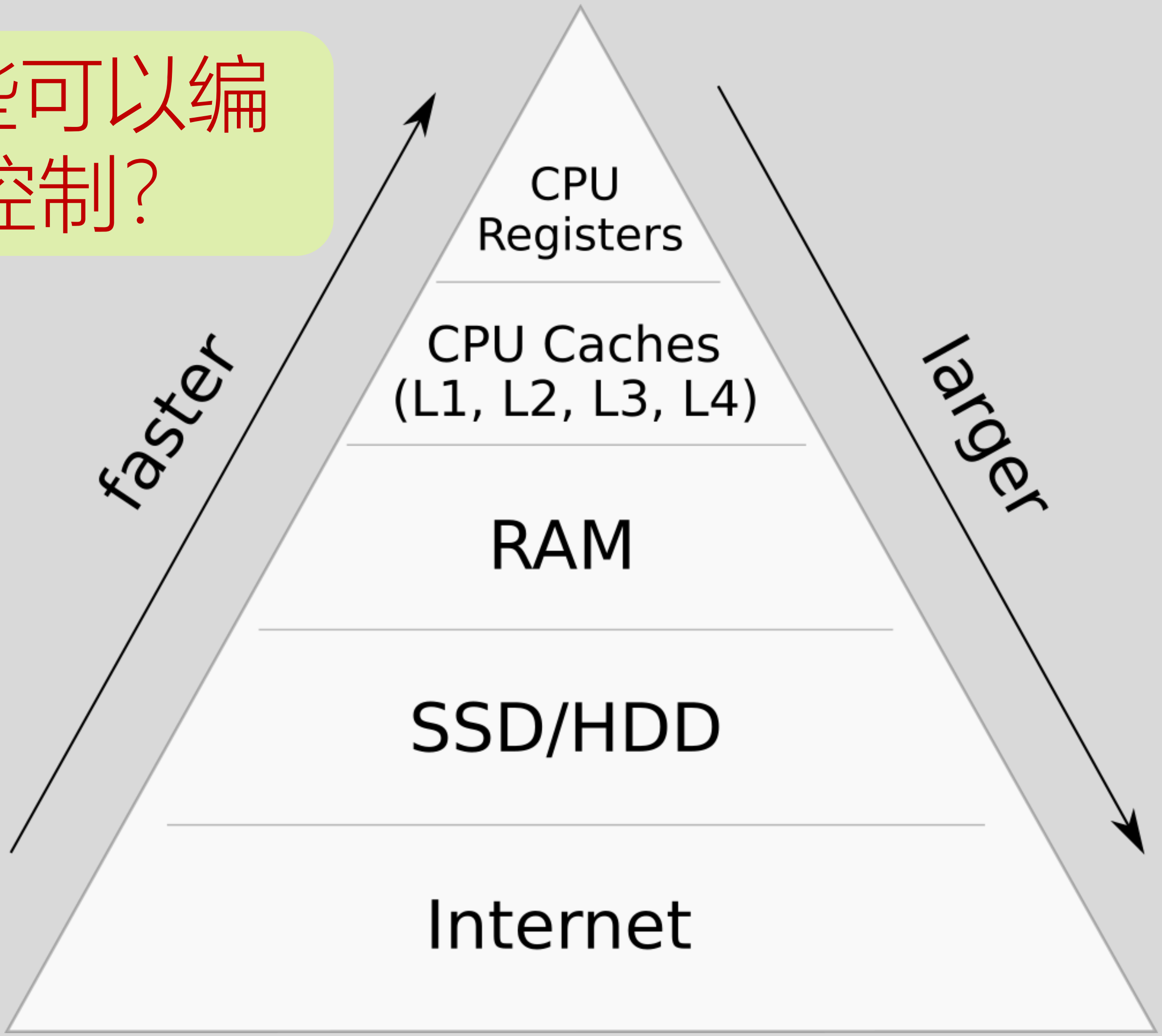
```
mov  eax, DWORD PTR [rsi]
add  eax, DWORD PTR [rdi]
mov  DWORD PTR [rdx], eax
```

	多长时间?	时钟周期?
RAM	≈100ns	200
cache	≈2ns-20ns	5-50
HDD	≈10ms	10^7

10万倍

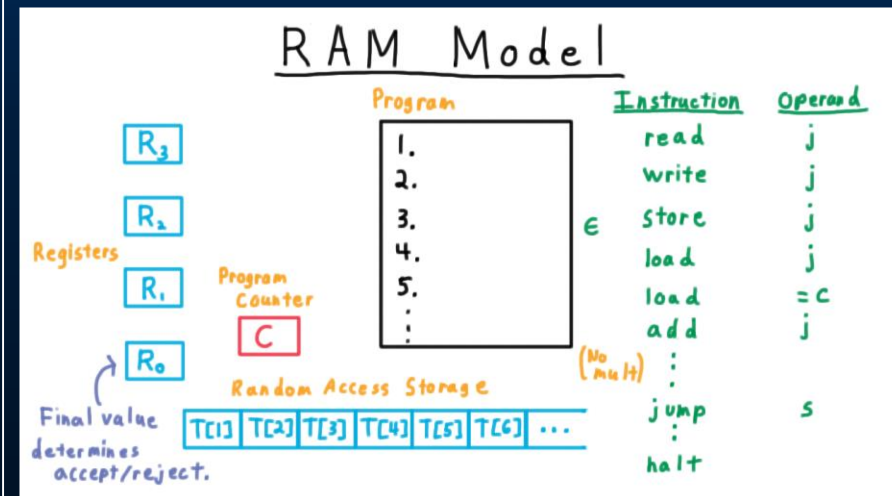
如何设计高效大数据算法

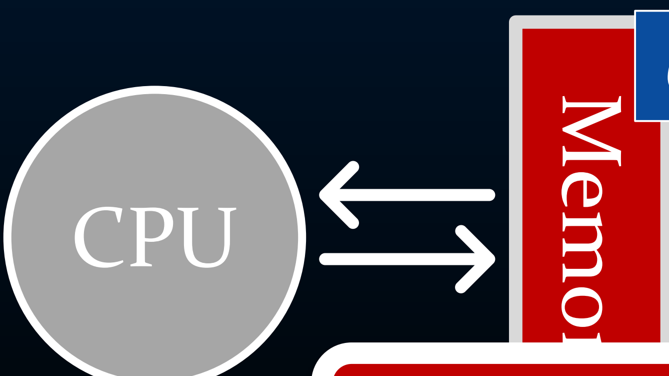
哪些可以编程控制?



计算模型

RAM计算模型

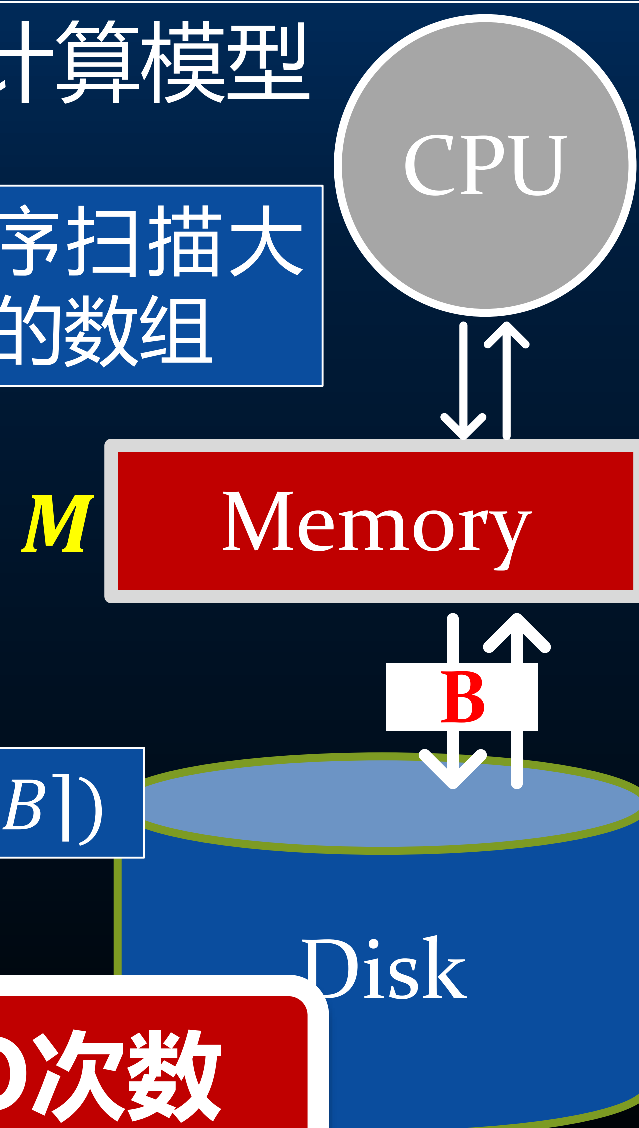




$O(N)$

I/O计算模型

例-顺序扫描大小为N的数组

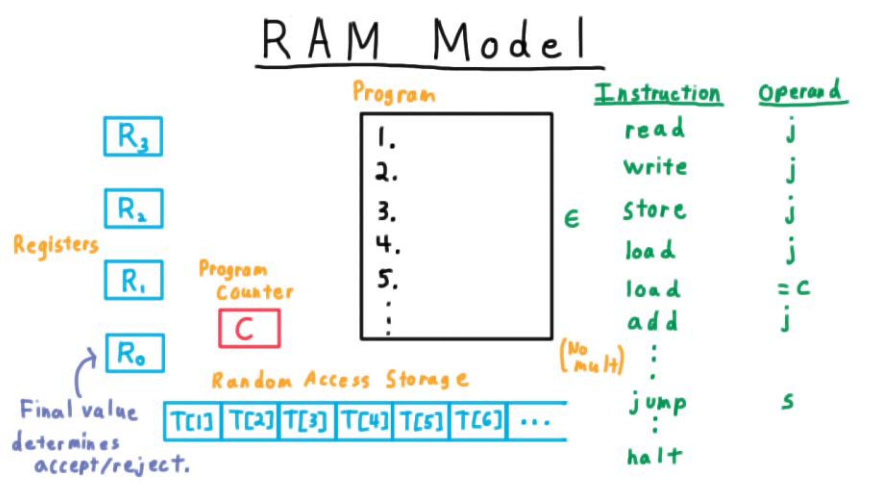


$O(\lceil N/B \rceil)$

指令数目 v.s. I/O次数

计算模型

RAM计算模型



CPU

Memory

$O(N)$

I/O计算模型

例-顺序扫描大小为N的数组

CPU

M Memory

B

$O(\lceil N/B \rceil)$

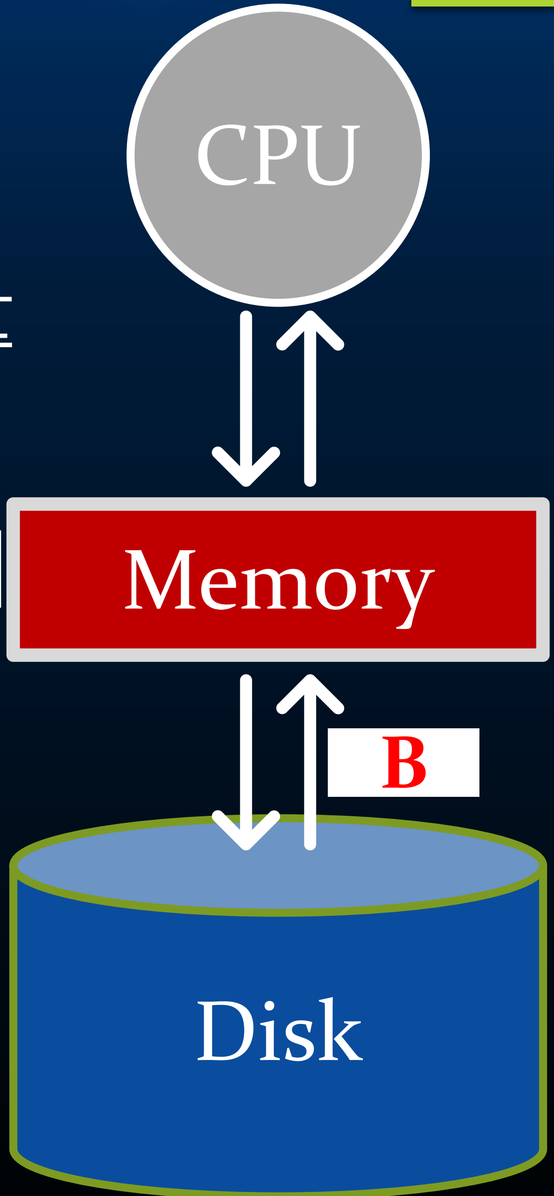
时间 (CPU周期)

指令数目 v.s. I/O次数

计算模型

- 显示指定**读/写**外存(external memory)的指令
- 设计目标：最优化数据传输量

- 即，内外存交换数据块的数目
- 或，读写外存块的数目



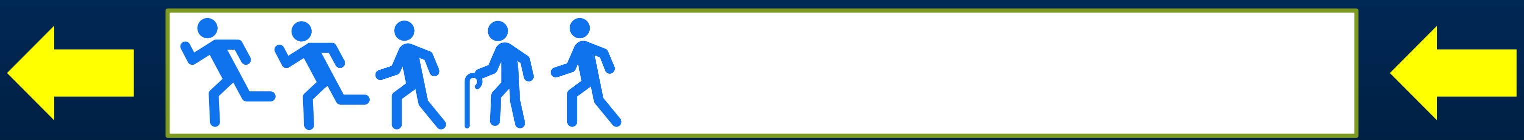
问题1：外存栈(stack)

- 栈
  - 内存维护大小为$2B$的数组，实现内存栈结构
  - 内存中存储k个栈顶数据，$k \le 2B$
  - 外存中存储其余数据
- Push
  - 如果内存中数据少于$2B$，直接内存压栈
  - 否则，向外存写入B个数据，然后，内存压栈
- Pop
  - 如果内存不为空，直接内存弹栈
  - 否则，从外存读入最新写入的B个数据，内存弹栈

最坏情况I/O代价--$O(1)$

均摊情况I/O代价--$O(1/B)$

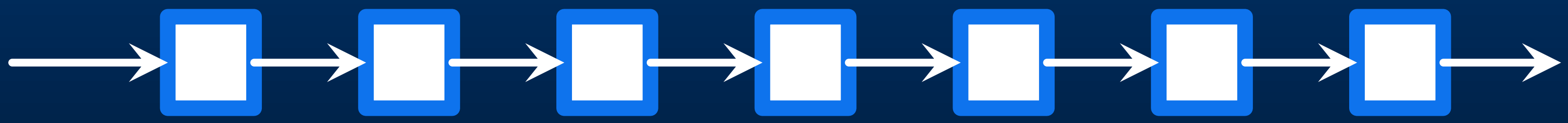
问题2：外存队列(queue)



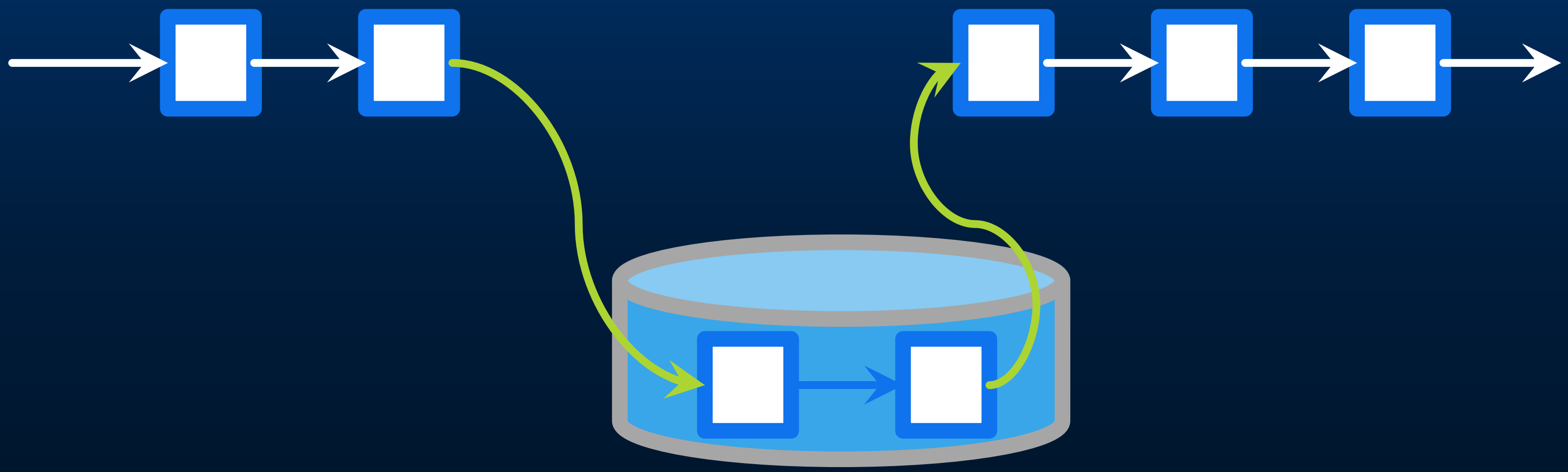
出队　　入队

- 内存维护2个大小为B的数组X和Y，一个用于出队，一个用于入队
- 具体操作?
- I/O代价分析
  - 最坏情况：$O(1)$
  - 均摊情况：$O(1/B)$

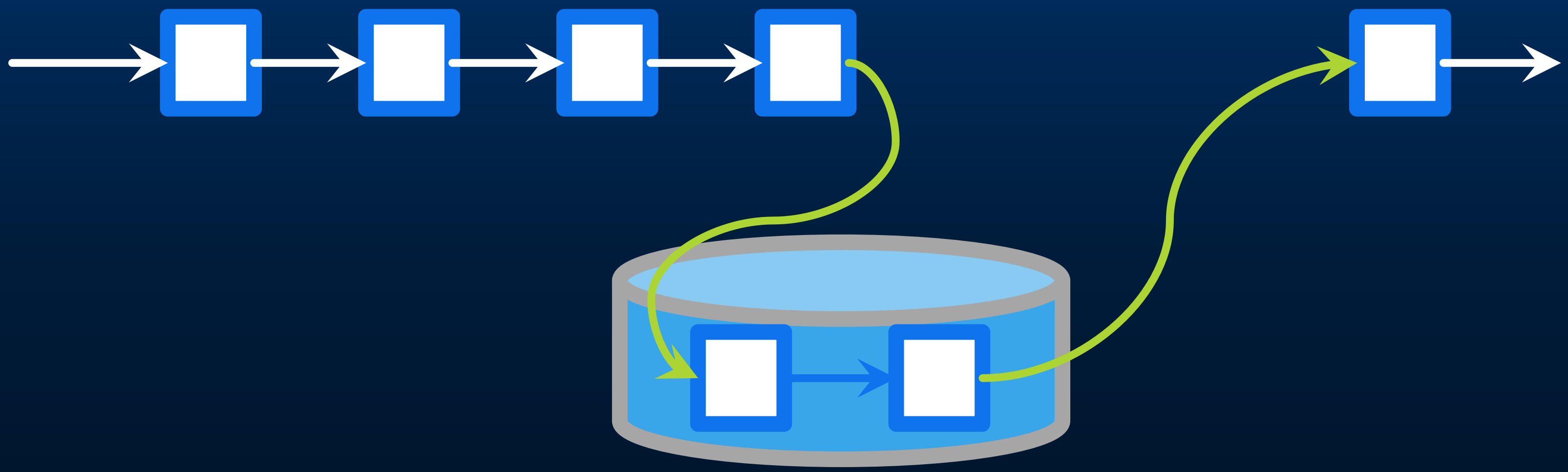
问题3：外存链表



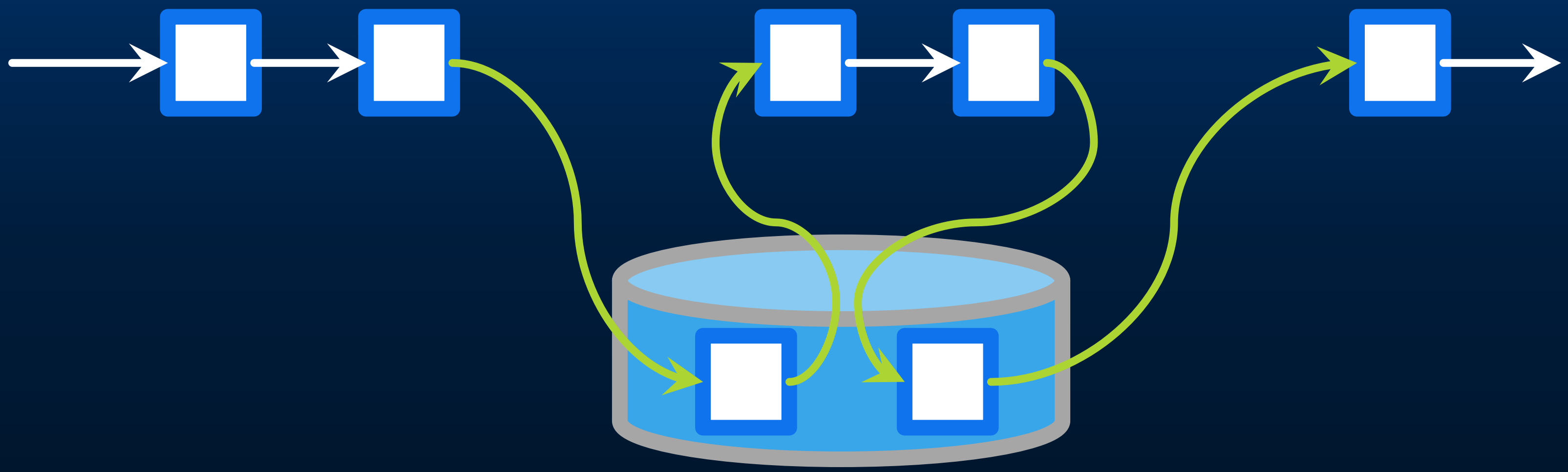
问题3：外存链表



问题3：外存链表



问题3：外存链表

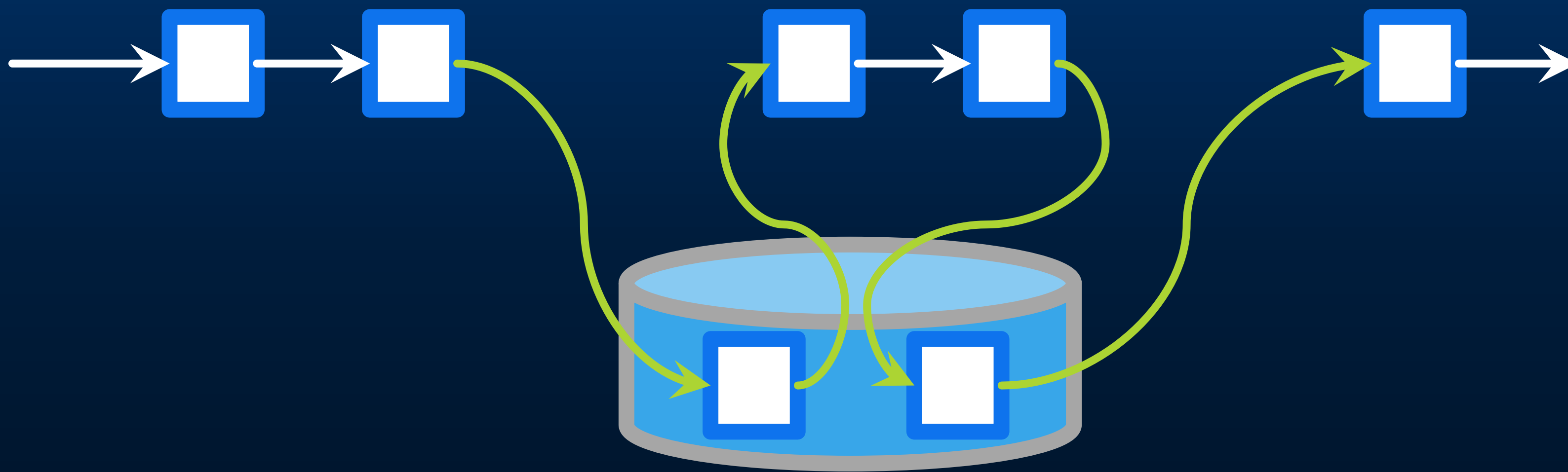


- 三种操作：insert(x,p), remove(p), traverse(p,k)

想法1

- 将链表中元素连续存放，尽可能装满每个块
- 更新?

问题3：外存链表



- 三种操作：insert(x,p), remove(p), traverse(p,k)

想法2 均摊代价 N次连续insert $O(2N/B)$

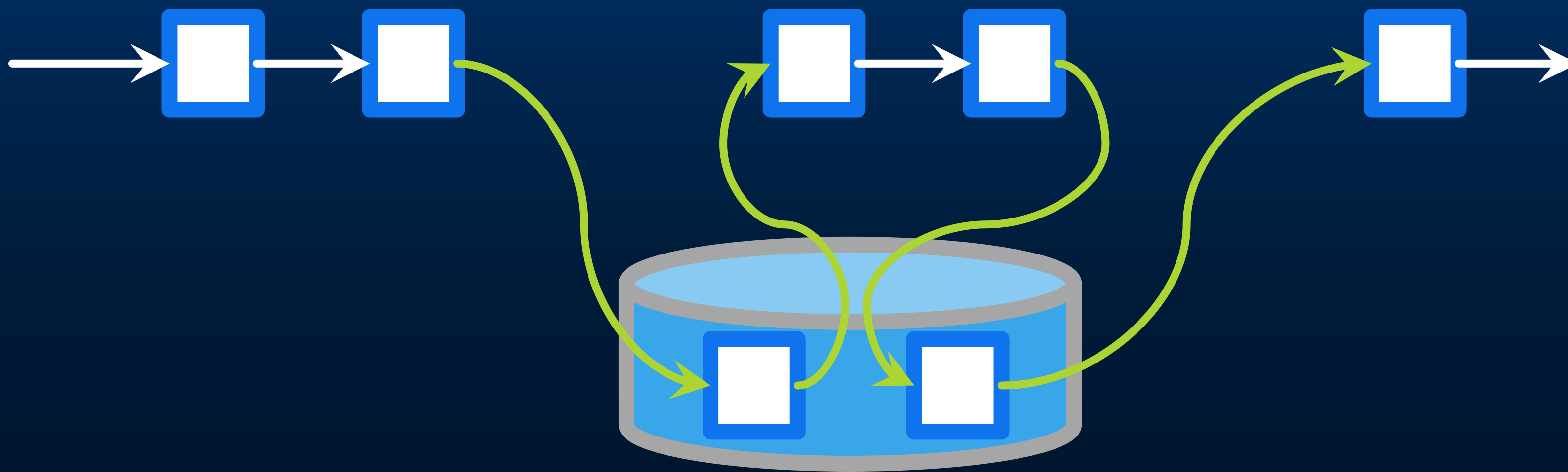
- 每个块装半满，至少包含$B/2$个数据

insert ⇒ 拆分块

traverse $O(2k/B)$

remove ⇒ 合并块；合并后再拆分

问题3：外存链表



- 三种操作：insert(x,p), remove(p), traverse(p,k)

想法2 均摊代价 N次连续insert $O(2N/B)$

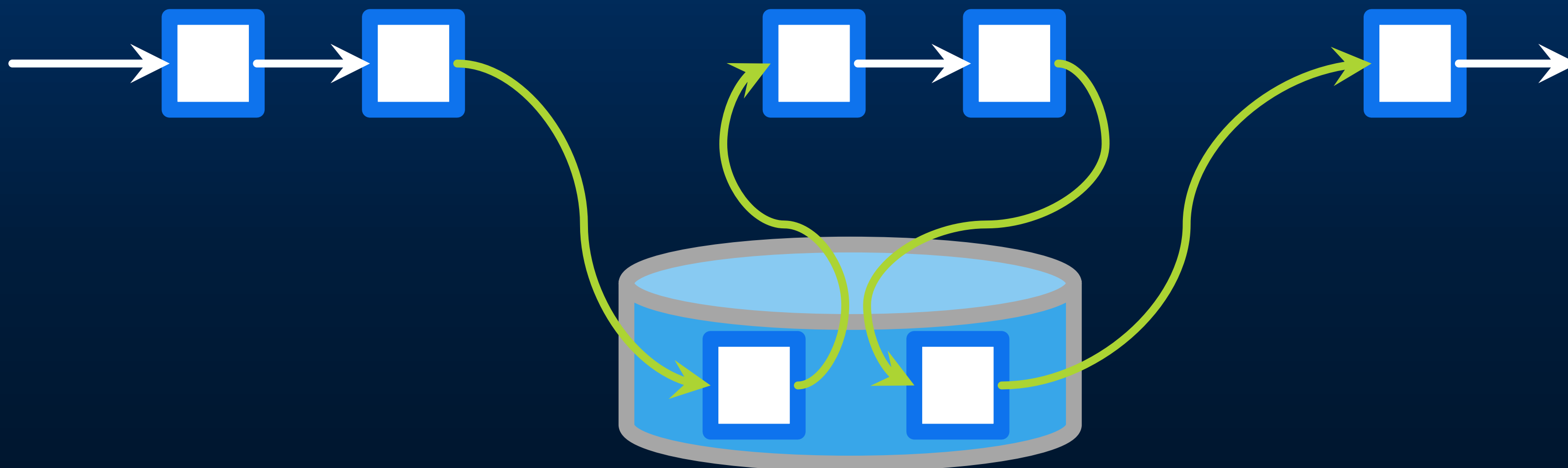
- 每个块装半满，至少包含$B/2$个数据

insert ⇒ 拆分块 最坏 $O(1)$

traverse $O(2k/B)$

remove ⇒ 合并块，合并后再拆分

问题3：外存链表



- 三种操作：insert(x,p), remove(p), traverse(p,k)

想法2

均摊代价 N次连续remove　　$O(2N/B)$

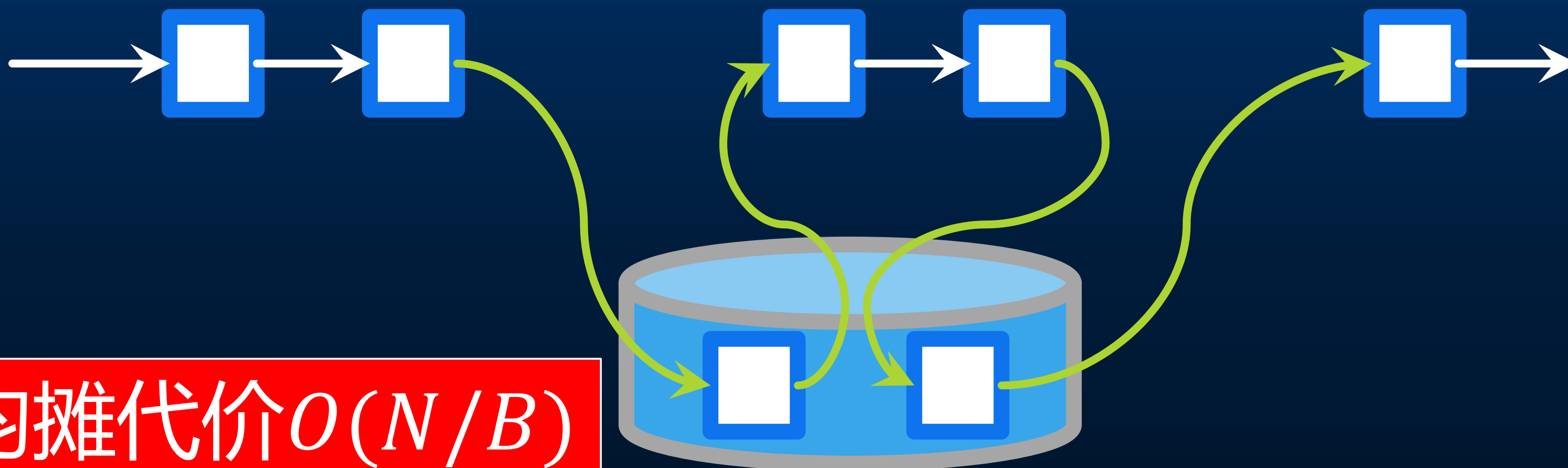
- 每个块装半满，至少包含$B/2$　$O(\log_2 B \cdot N/B)$

insert ⇒ 拆分块　　最坏 $O(1)$

traverse　$O(2k/B)$

remove ⇒ 合并块，合并后再拆分

问题3：外存链表



均摊代价$O(N/B)$

- 三种操作：insert(x,p), remove(p), traverse(p,k)

想法3

traverse	$O(3k/B)$

- 连续的两个块至少包含2B/3个数据；

insert ⇒ 块满则向邻居插入，邻居均满则拆分

最坏 $O(1)$

remove ⇒ 与邻居块合并是否$\le 2B/3$，是则合并

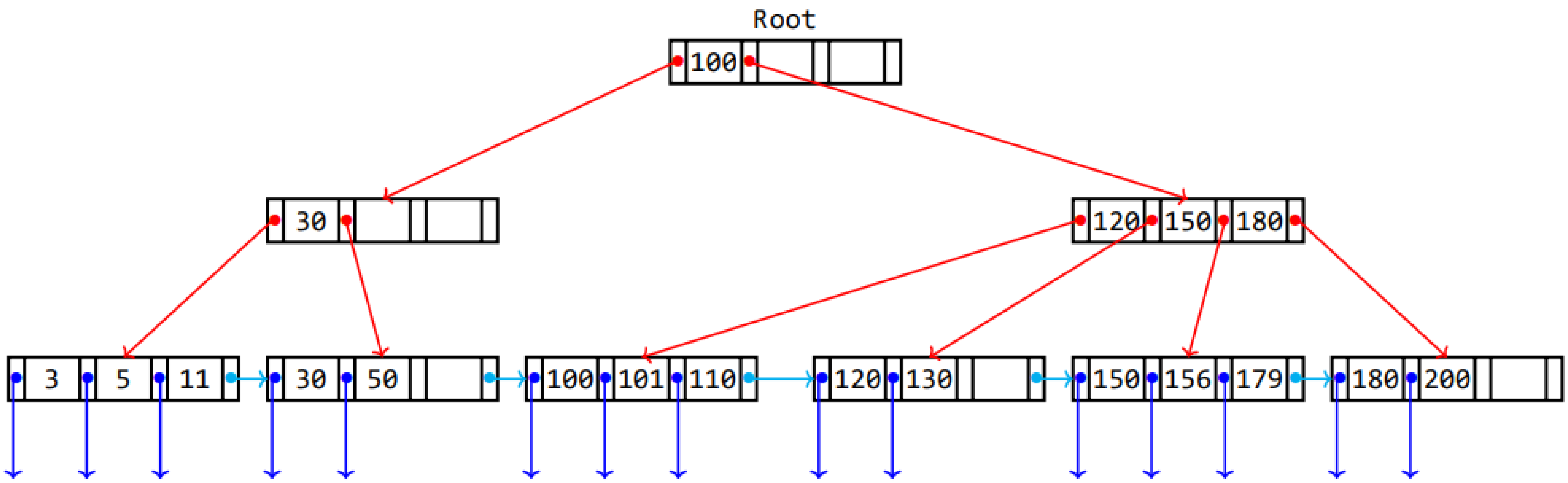
问题4：搜索问题—B+树

搜索问题: 维护数据, 支持insert(x), remove(x), query(x)

- ① 二分查找树
- ② (a, b)-tree: 2 ≤ a ≤ (b + 1)/2
  - 根节点有0或≥2个孩子，其它非叶子节点的孩子数目∈[a, b]
  - insert: 找到对应叶子节点，插入后若大于b，均分，递归上一层插入
  - remove: 找到对应叶子节点，删除后若小于a，与邻接块合并，合并后若大于b，平均划分，进而在上一层递归删除节点或者调整键值
  - query: $O\left(\log_a\left(\frac{N}{a}\right)\right)$
- ③ B-tree： $a = \frac{B}{2}, b = B, O(\log_B N)$
- ④ B+tree：自动平衡、数据按键值有序存储、$O(\log n)$代价
  - 为以块（block）为基本单位读写数据而设计

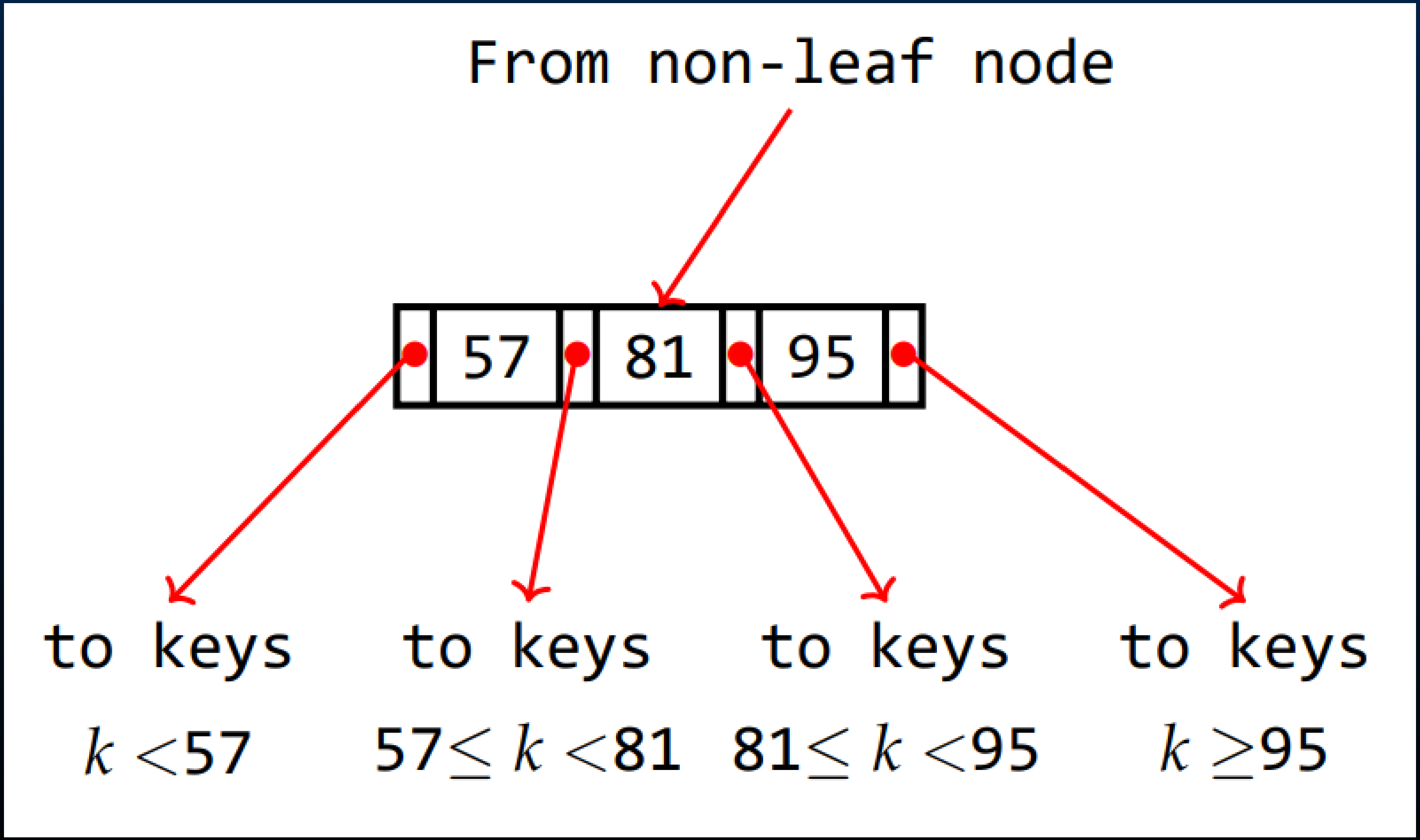
问题4：搜索问题——B+树

- B+Tree: n = 3



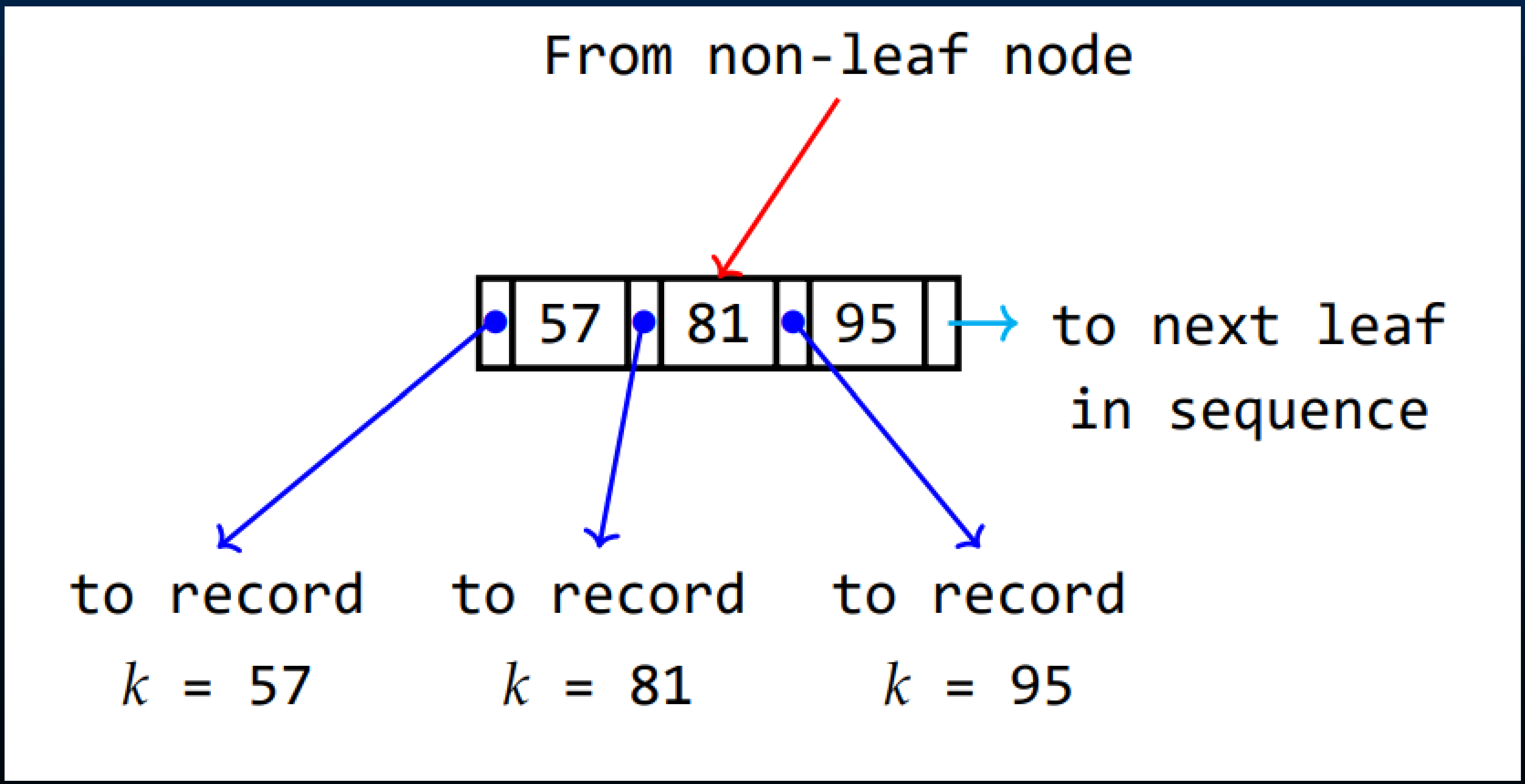
问题4：搜索问题—B+树

- B+Tree: n = 3
- 非叶子节点（non-leaf)



问题4：搜索问题—B+树

- B+Tree: n = 3
- 叶子节点（leaf）



问题4：搜索问题—B+树

- B+Tree中节点的大小
  - n + 1 个指针
  - n 个键值（keys）
- **主要思想：避免节点闲置空间过多**
- 节点至少使用
  - $\lceil (n+1)/2 \rceil$个指针 (Non-leaf Nodes)
  - $\lfloor (n+1)/2 \rfloor$个指针 (Leaf Nodes)

问题4：搜索问题—B+树

给定参数n，B+Tree的规则

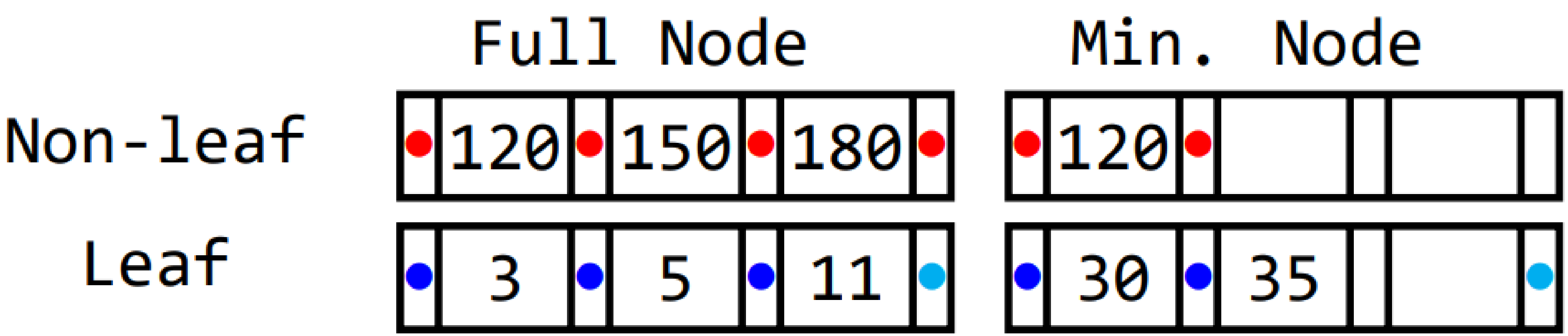
- 平衡树
- 叶子节点都在同一层（最底层）
- 除了指向下一个叶子节点的指针，其余指针指向数据
- pointers/keys的约束如下表所示

	Max Pointers	Max Keys	Min Ptrs (To Data)	Min Keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

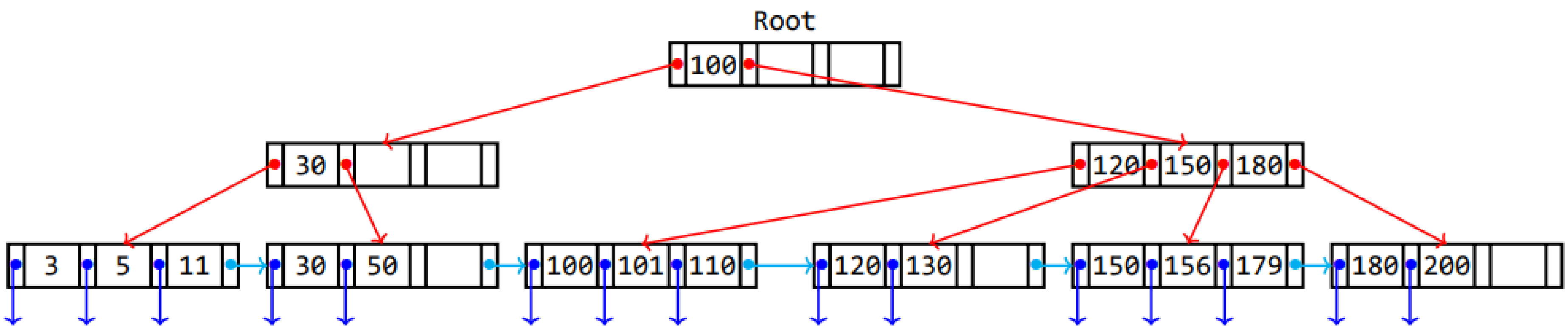
问题4：搜索问题——B+树

	Max Pointers	Max Keys	Min Ptrs (To Data)	Min Keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

- n = 3



问题4：搜索问题—B+树



B+Tree中的查找操作（假设查找键值为K），调用query(root,K)

函数：query(v,K)

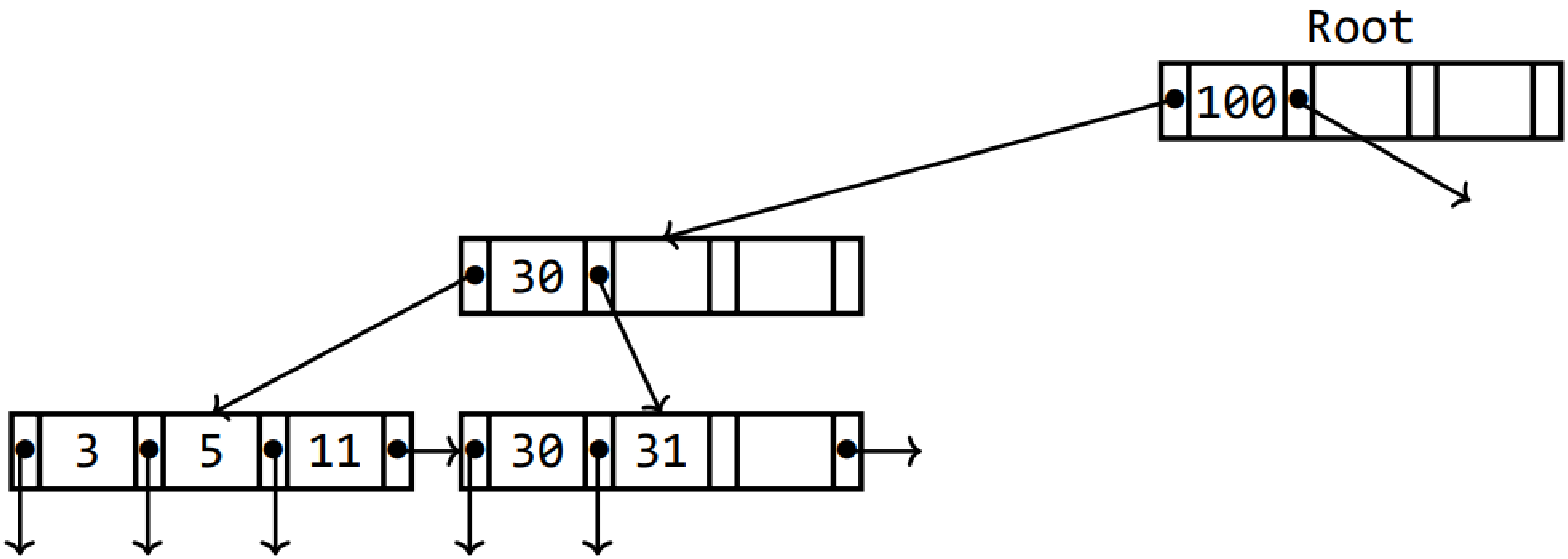
$$O\left(\log_{\left\lceil\frac{n+1}{2}\right\rceil} N\right)$$

- 如果v是叶子结点，在该结点的键值中查找，若有K，贝
- 如果v是非叶子结点，依次考虑v中的键值$K_1, K_2, \dots, K_n$：
  - $K_i \le K < K_{i+1}$，令u为第$i+1$个指针指向结点，递归查找query(u,K)；
  - $K < K_1$，令u为第1个指针指向结点，递归查找query(u,K)；
  - $K_n \le K$，令u为第$n+1$个指针指向结点，递归查找query(u,K)。

问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

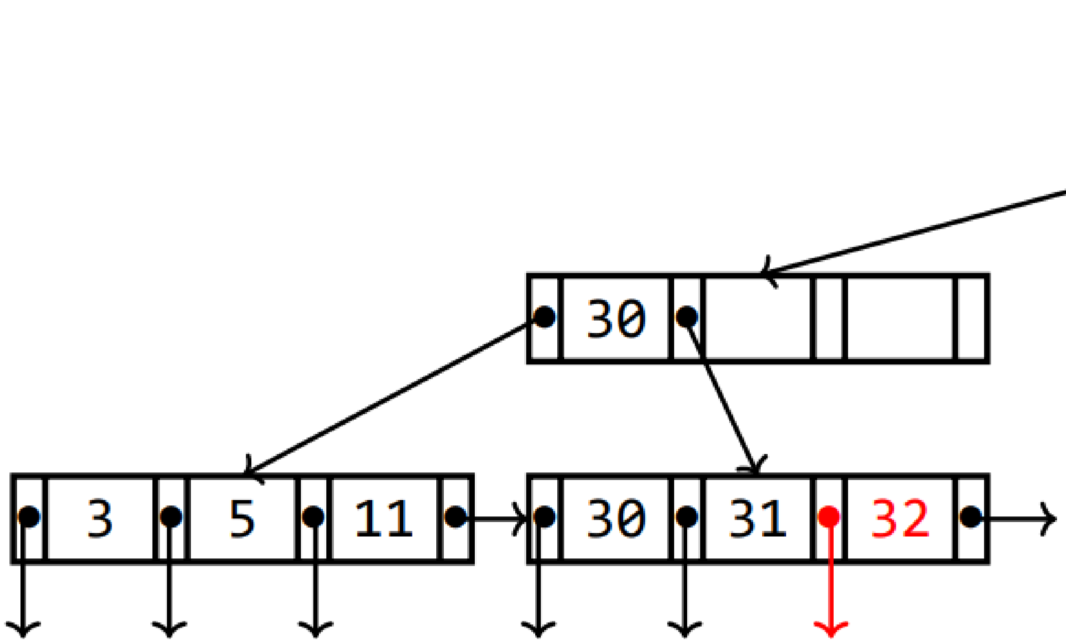
Insert key = 32



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

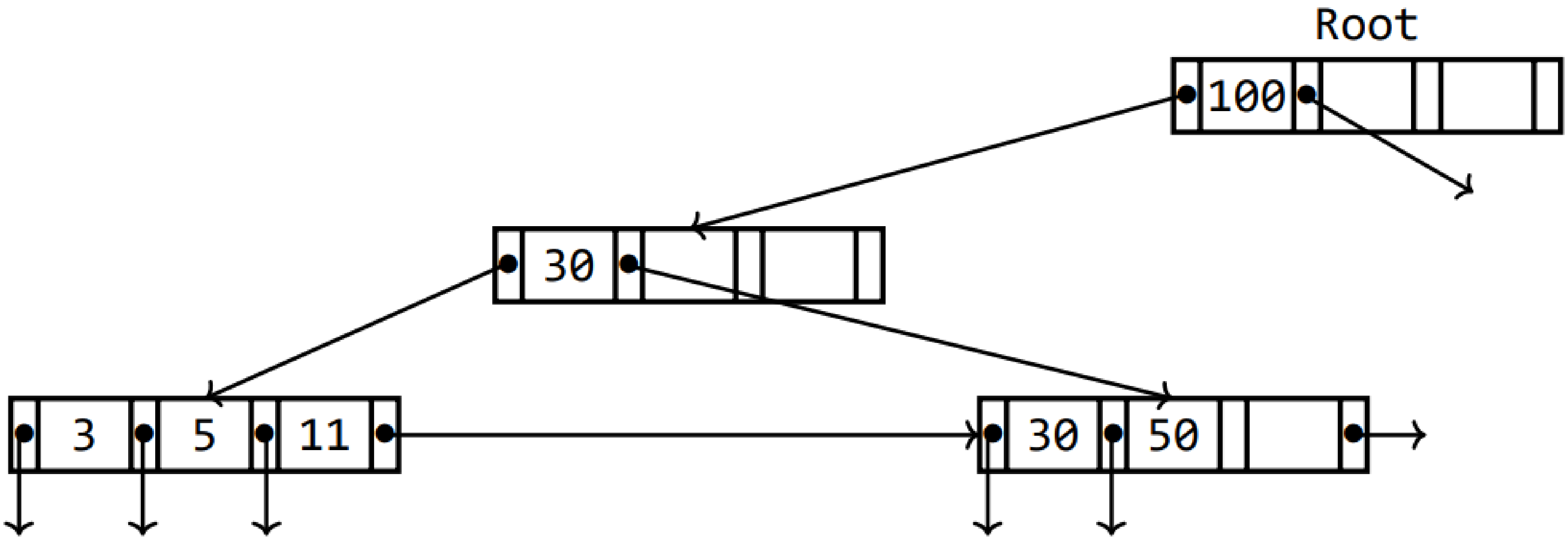
Insert key = 32



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

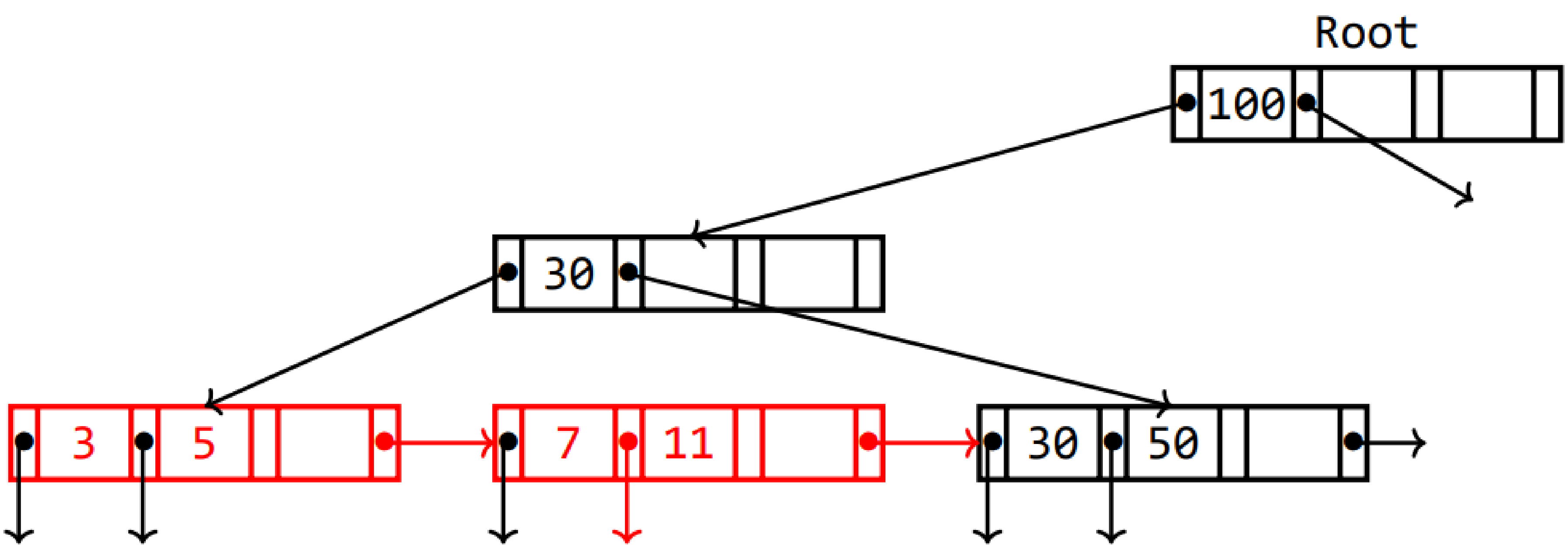
Insert key = 7



问题4：搜索问题—B+树-插入

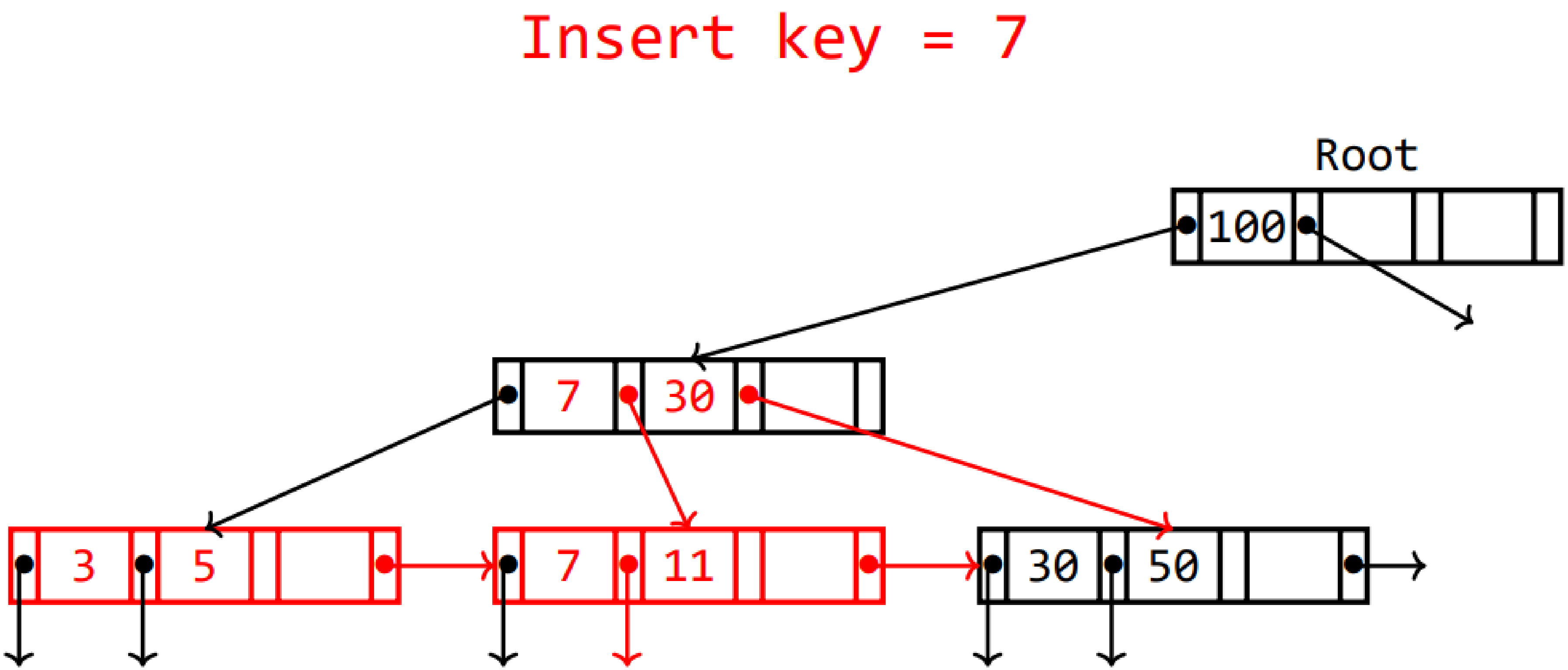
- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

Insert key = 7



问题4：搜索问题—B+树-插入

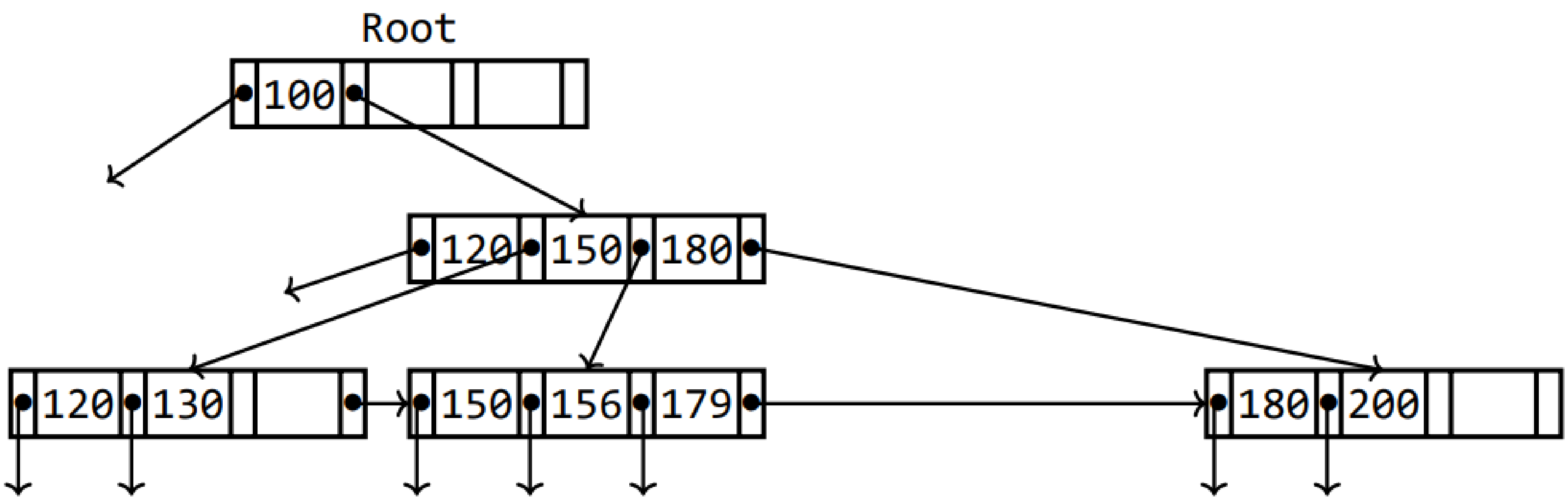
- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root



问题4：搜索问题——B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

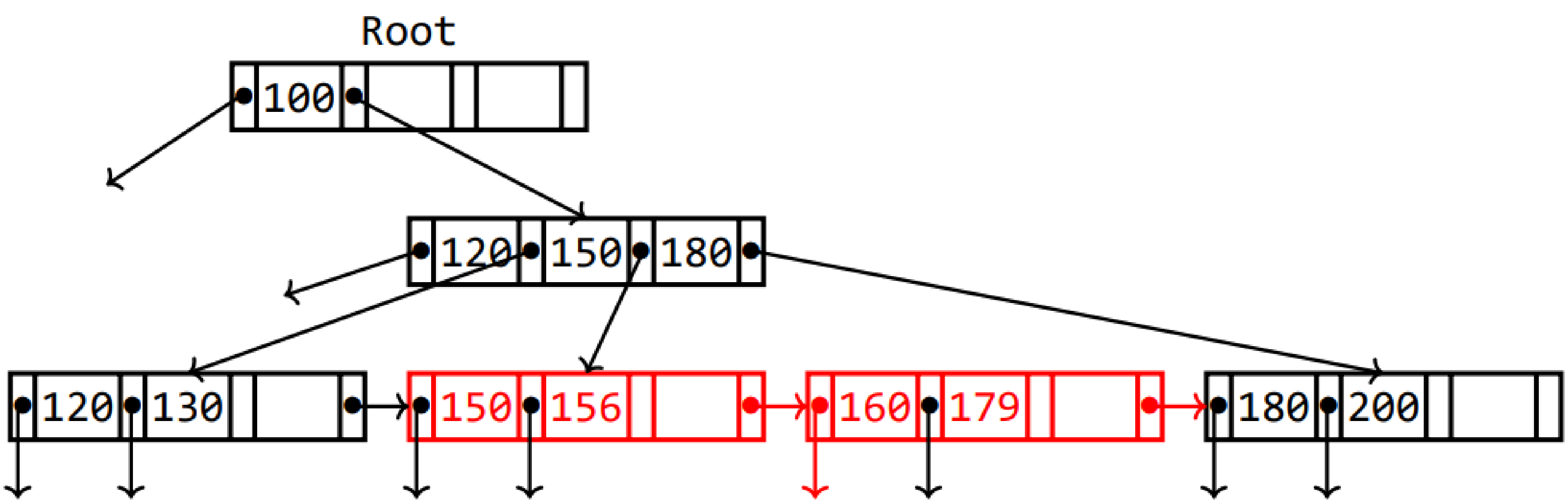
Insert key = 160



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

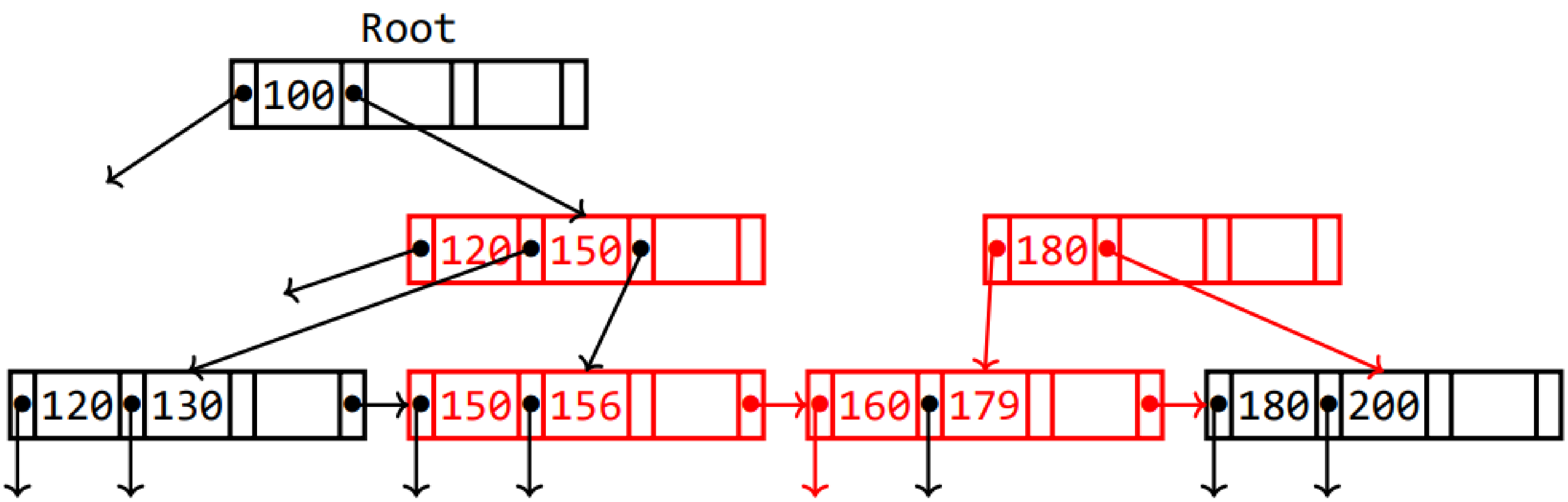
Insert key = 160



问题4：搜索问题—B+树-插入

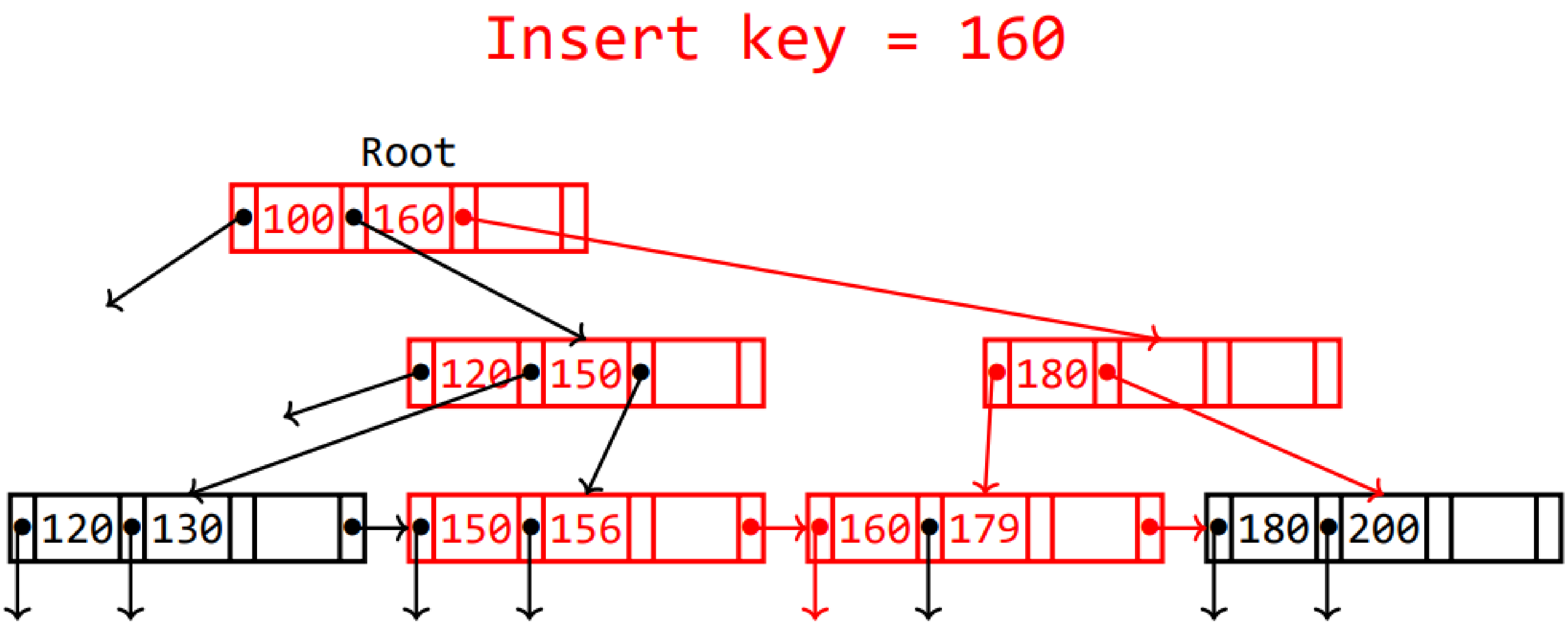
- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

Insert key = 160



问题4：搜索问题—B+树-插入

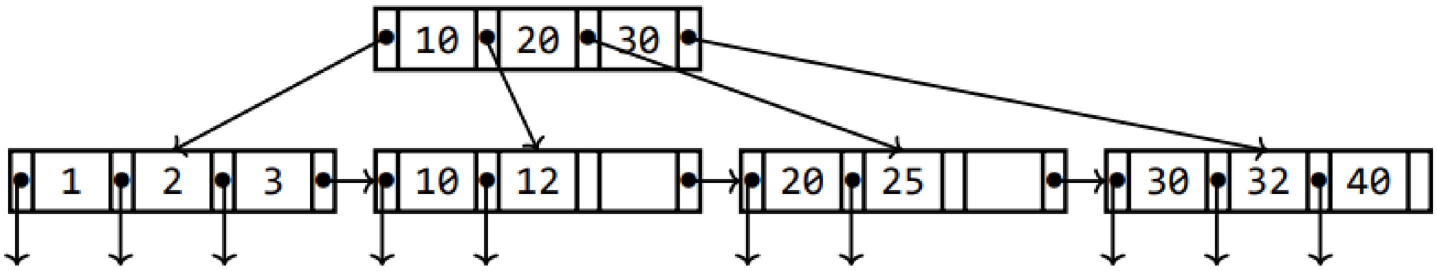
- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

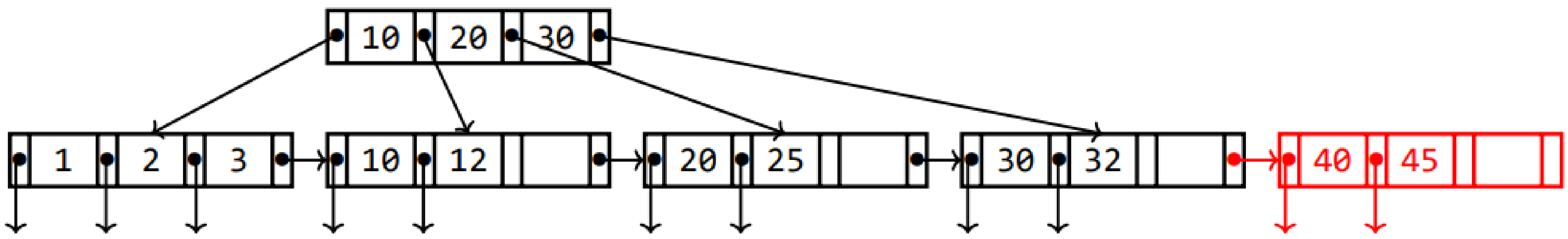
Insert key = 45



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

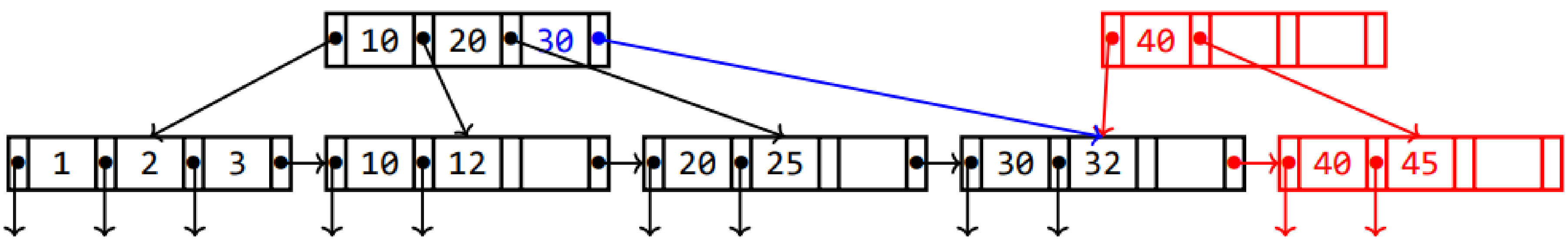
Insert key = 45



问题4：搜索问题—B+树-插入

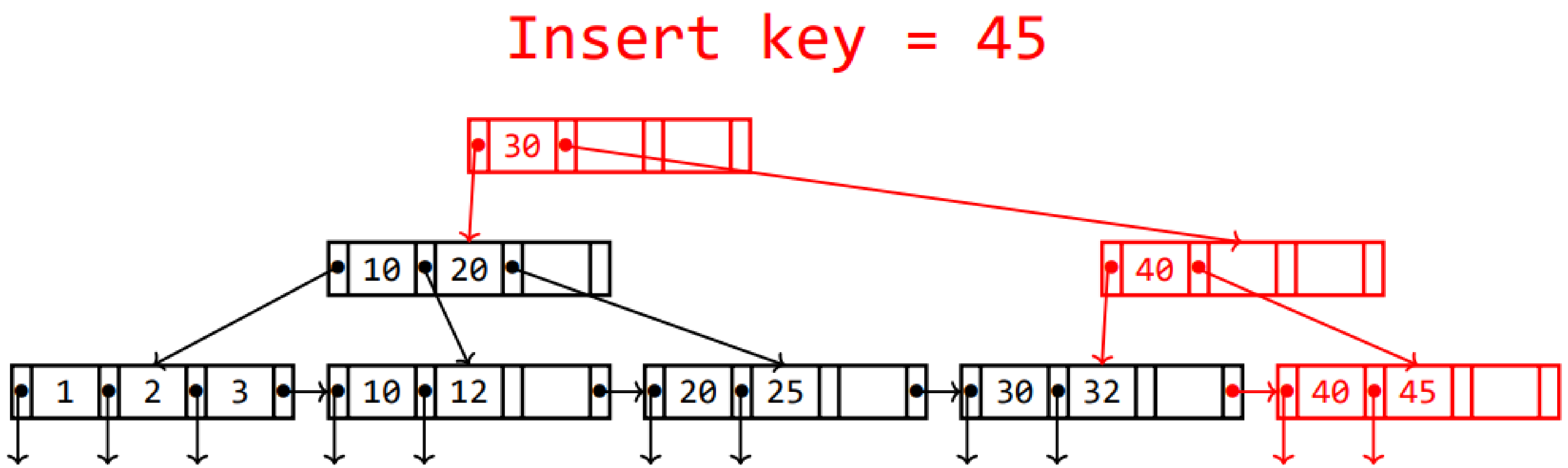
- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root

Insert key = 45



问题4：搜索问题—B+树-插入

- (a) Simple Case
- (b) Leaf Overflow
- (c) Non-leaf Overflow
- (d) New Root



问题4：搜索问题—B+树

B+Tree的插入操作

- 假设插入的键值为K，首先找到对应的叶结点L
- 如果L中有空闲空间，那么直接插入，结束；否则将叶结点分裂为两个结点，并且把其中的键分到这两个新结点中，使得键个数满足最小要求；
- 某一层的结点分裂在上一层看来相当于在上层插入新的键-指针，因此，在上层可以继续递归插入，如有结点分裂，则向上一层推进；
- 如果要在根结点中插入键-指针，并且导致根结点分裂，那么就在上一层新建根结点，该根结点指向刚分裂的两个新结点，键由这两个结点确定；

当分裂叶结点N时

- 假设N是一个容量为n个键的叶结点，插入第 n+1 个键-指针。
- 创建一个新结点M，让M为N的右侧兄弟，将键排序，前 $\lceil (n+1)/2 \rceil$个留在N中，其他键-指针放入M中。
- 上一层新加入的键是M中最小的键值。

问题4：搜索问题——B+树

B+Tree的插入操作

- 假设插入的键值为K，首先找到对应的叶结点L
- 如果L中有空闲空间，那么直接插入，结束；否则将叶结点分裂为两个结点，并且把其中的键分到这两个新结点中，使得键个数满足最小要求；
- 某一层的结点分裂在上一层看来相当于在上层插入新的键-指针，因此，在上层可以继续递归插入，如有结点分裂，则向上一层推进；
- 如果要在根结点中插入键-指针，并且导致根结点分裂，那么就在上一层新建根结点，该根结点指向刚分裂的两个新结点，键由这两个结点确定；

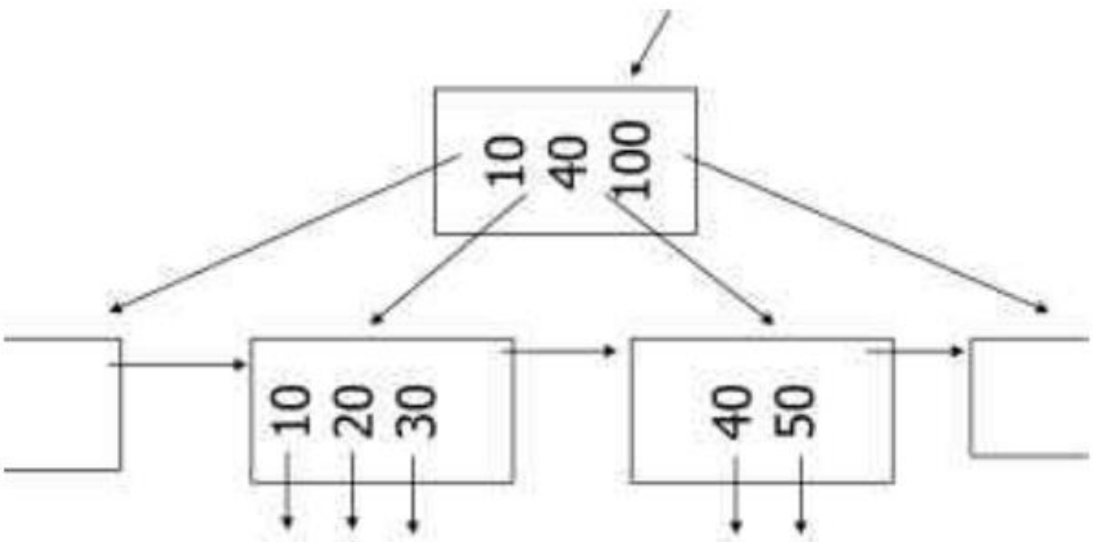
当分裂非叶结点N时

- N包含n个键、n+1个指针，正要插入第n+2个指针、第n+1个键。
- 创建新结点M，M为N的右侧兄弟，将键-指针排序，前$\lceil (n+2)/2 \rceil$个指针留在N中，剩下指针放入M中。
- 前$\lceil n/2 \rceil$个键留在N中，后$\lfloor n/2 \rfloor$个键放入M中，中间的键留出来，插入到上一层，该键指针指向M。

问题4：搜索问题——B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

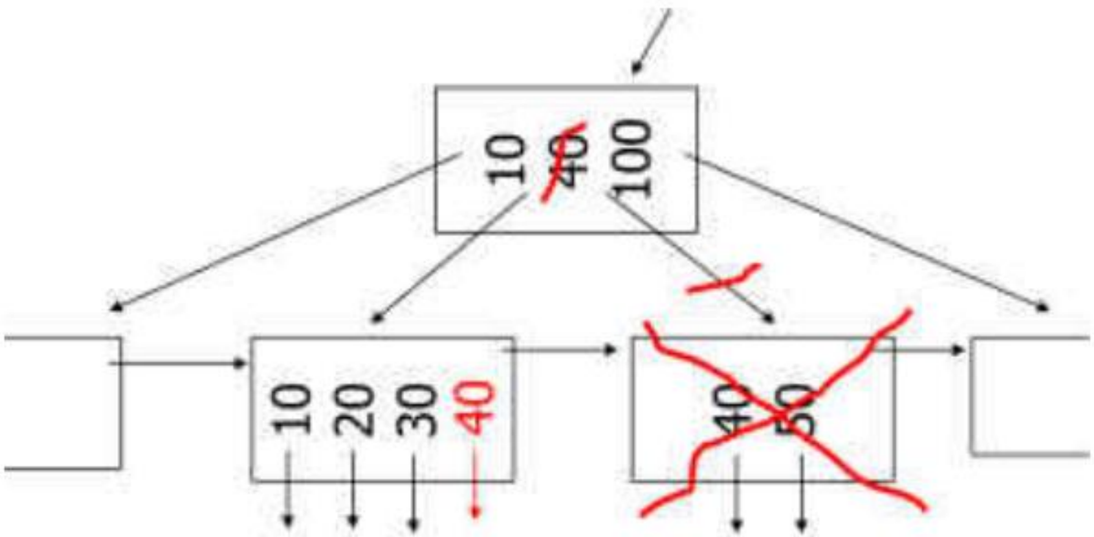
Delete key = 50, n = 4



问题4：搜索问题—B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

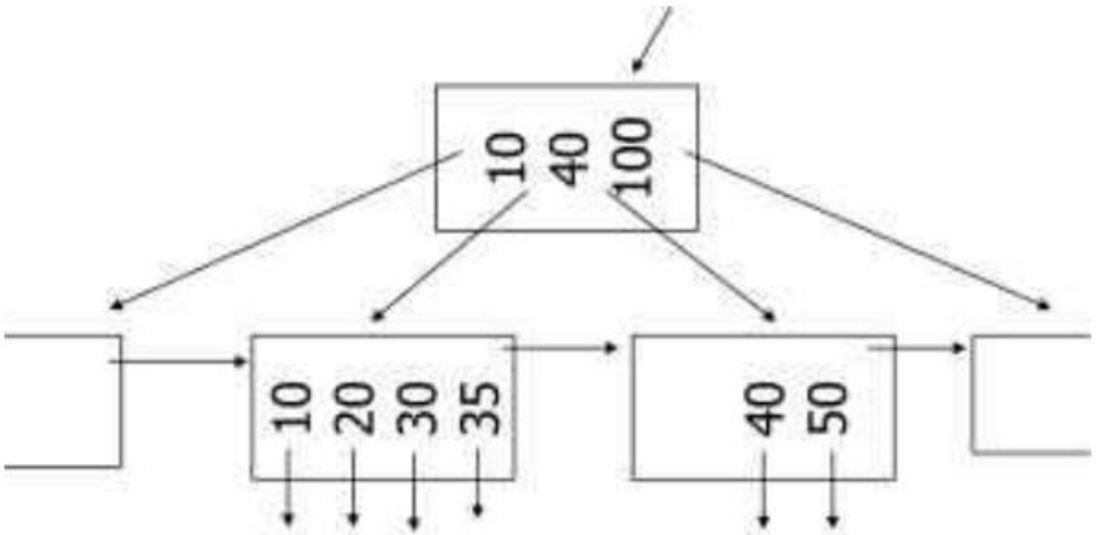
Delete key = 50, n = 4



问题4：搜索问题——B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

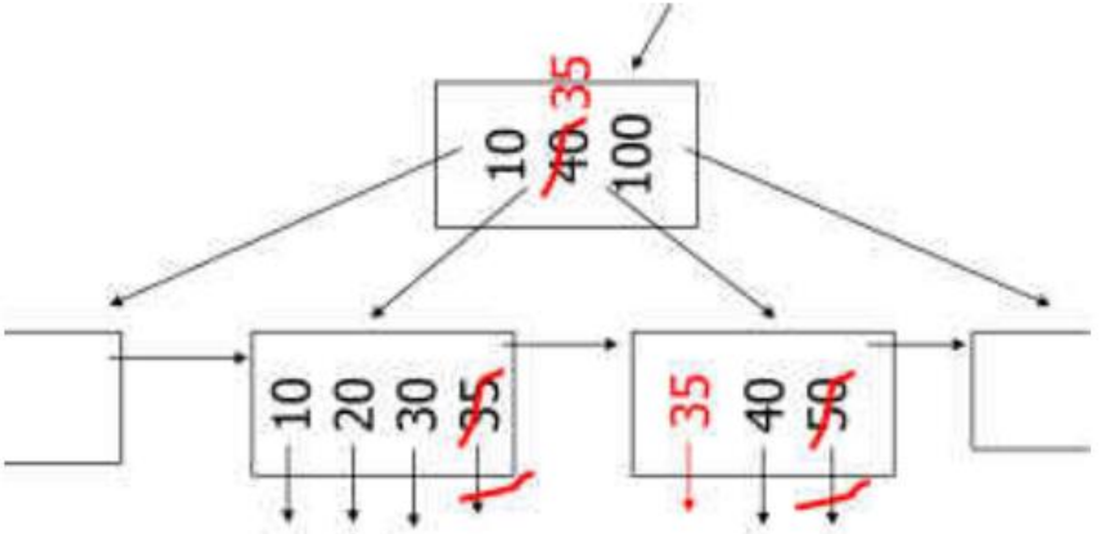
Delete key = 50, n = 4



问题4：搜索问题—B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

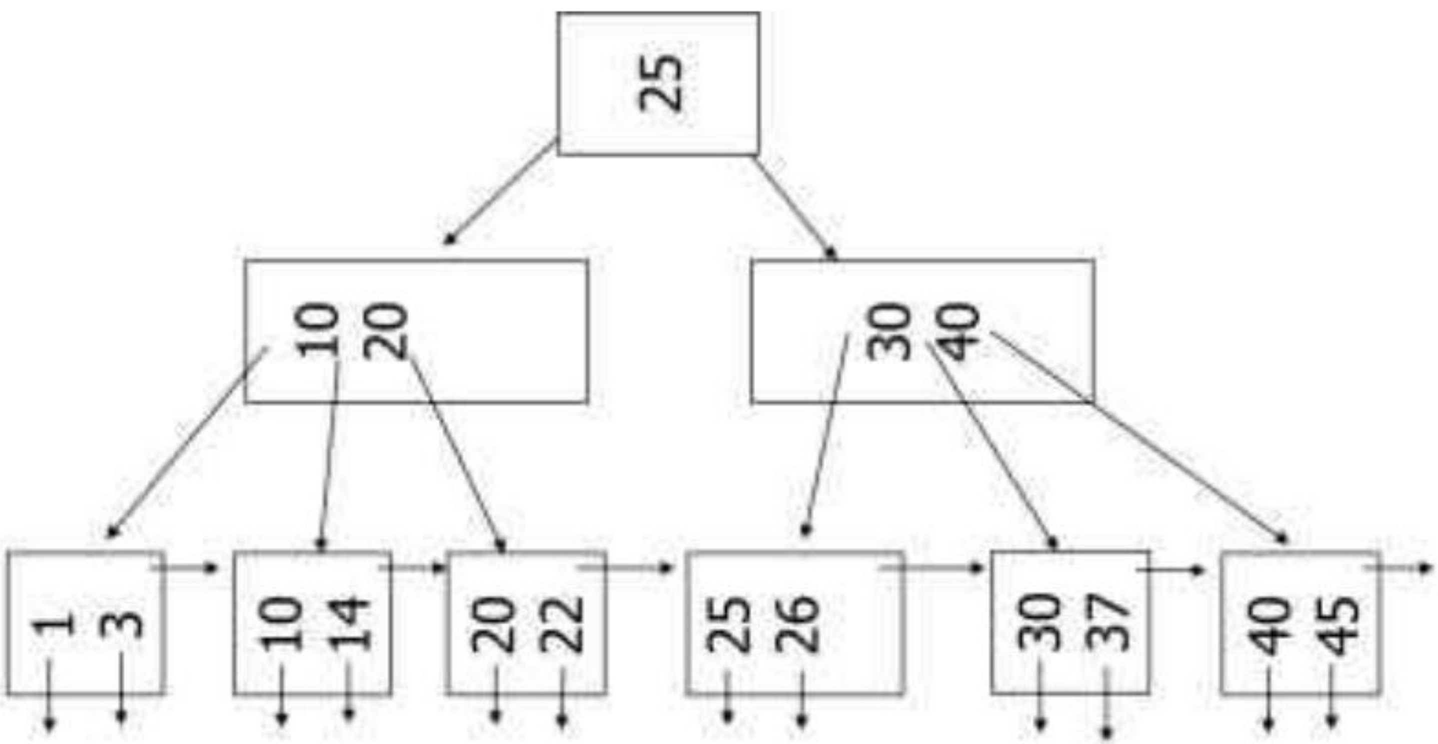
Delete key = 50, n = 4



问题4：搜索问题—B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

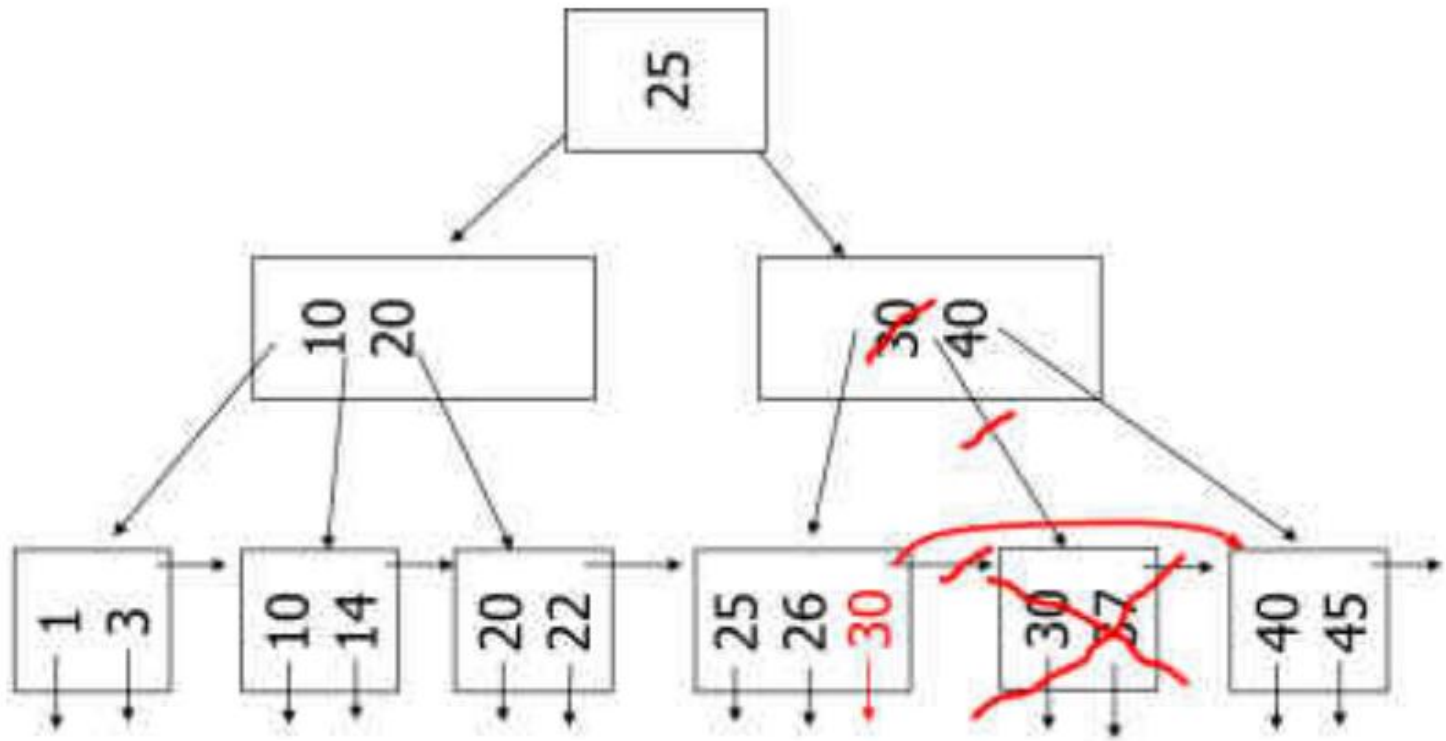
Delete key = 37, n = 4



问题4：搜索问题—B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

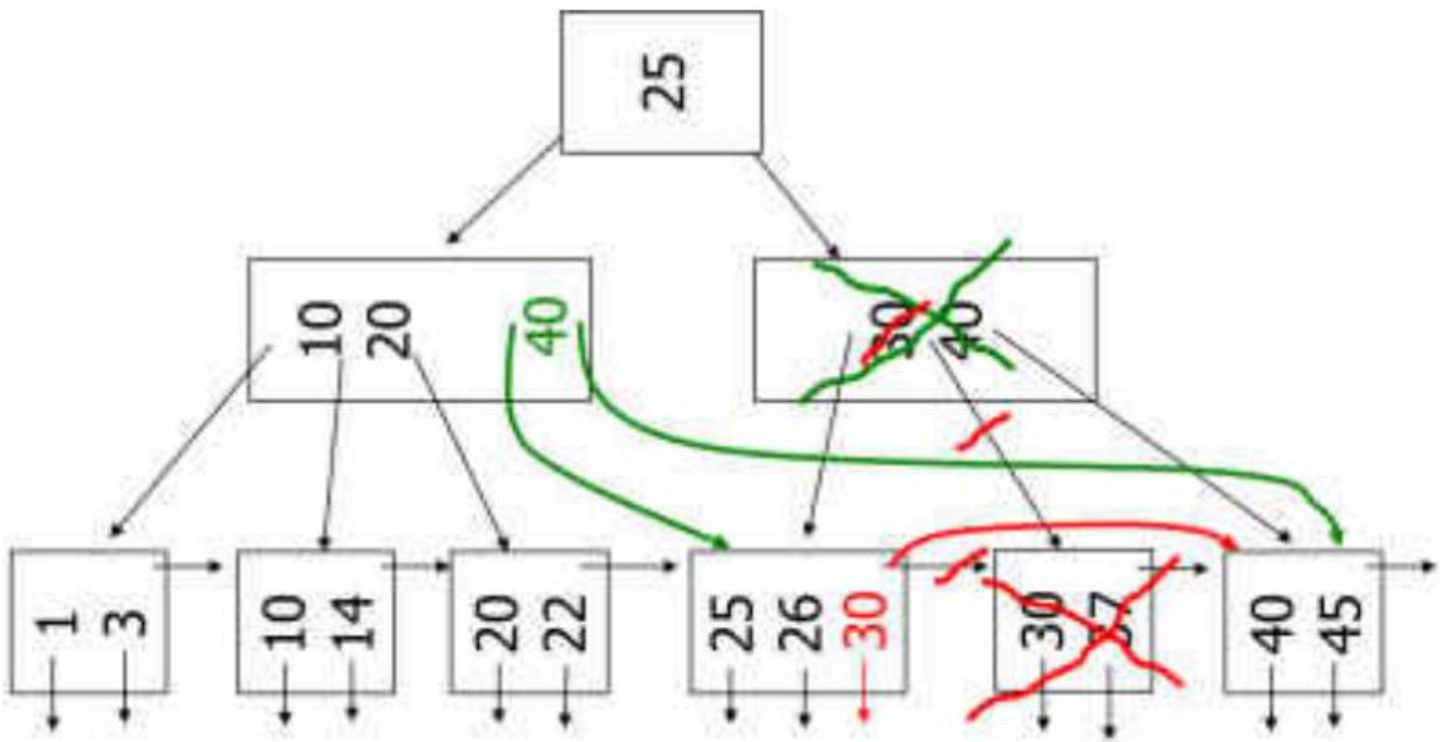
Delete key = 37, n = 4



问题4：搜索问题—B+树

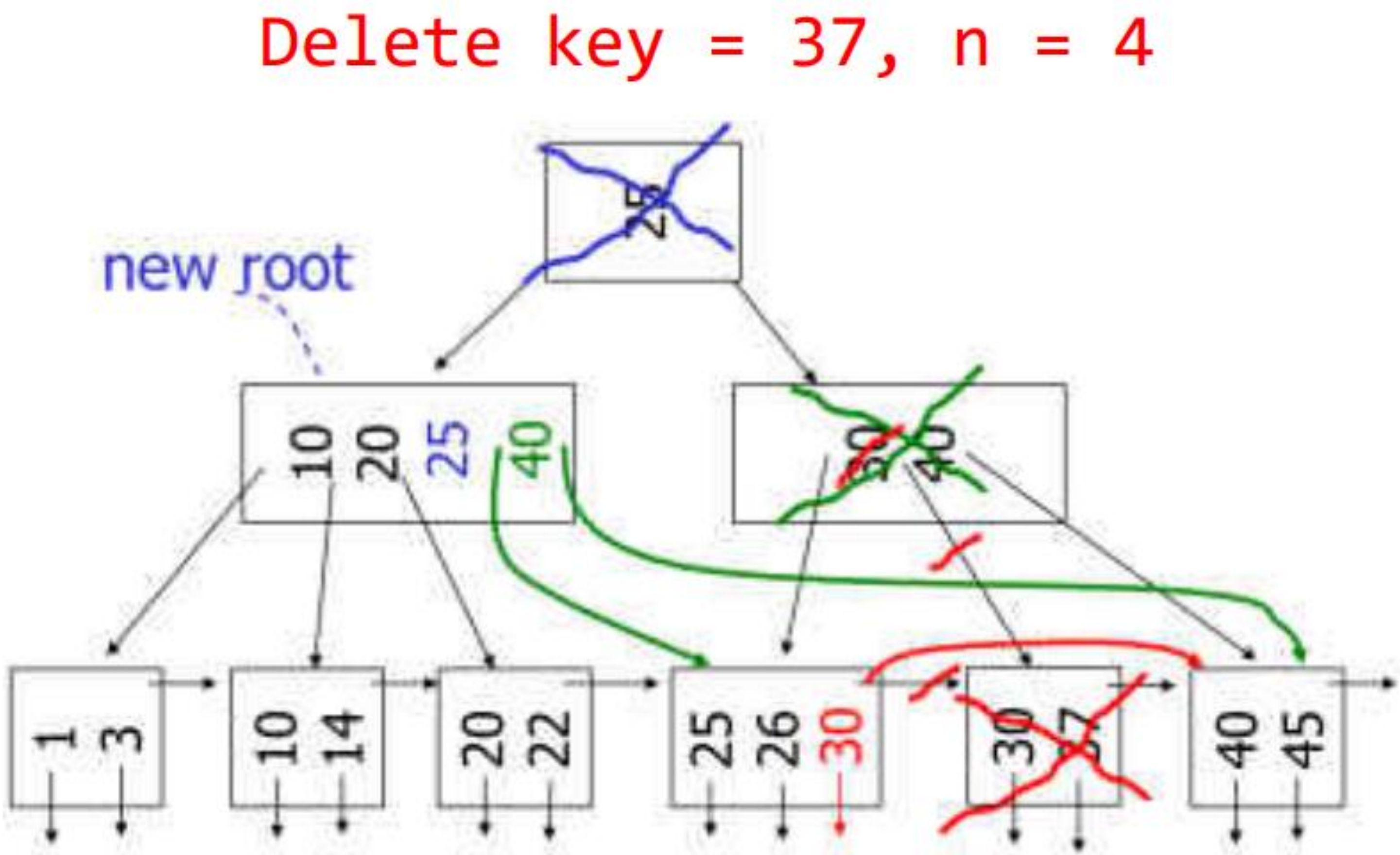
- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf

Delete key = 37, n = 4



问题4：搜索问题—B+树

- (a) Simple Case
- (b) Coalesce with neighbor
- (c) Re-distribute
- (d) Cases b or c non-leaf



问题4：搜索问题—B+树

B+Tree的删除操作

- 假设删除的键值为K，首先找到对应的叶结点L。
- 如果L中删除K后仍然具有满足最小要求的键个数，停止，否则做如下处理：
  - 尝试与L的相邻兄弟节点之一合并（合并后仍能放入同一节点），合并后，相当于在上层结点删除了一个键值，那么递归处理；
  - 否则，考虑L的相邻兄弟，假设其中一个能够提供L一个键-指针，并且去除该键-指针后该兄弟结点仍然满足键数的最小要求，那么L从该兄弟处借得一对键-指针，并更新父节点的对应键值；
  - 如果两个兄弟都无法提供一个键-指针，那么必然是如下情况：
    - ✓ ① L的键数少于最小数；
    - ✓ ② L兄弟M的键数恰好为最小数
  - 那么两个结点可以合并。

问题4：搜索问题——B+树

删除作结点中指针数目的最大值：n+1，记录条数：N

B+Tree的插入操作：$O\left(\log_{\lceil\frac{n+1}{2}\rceil}(N)\right)$；B+Tree的删除操作：$O\left(\log_{\lceil\frac{n+1}{2}\rceil}(N)\right)$

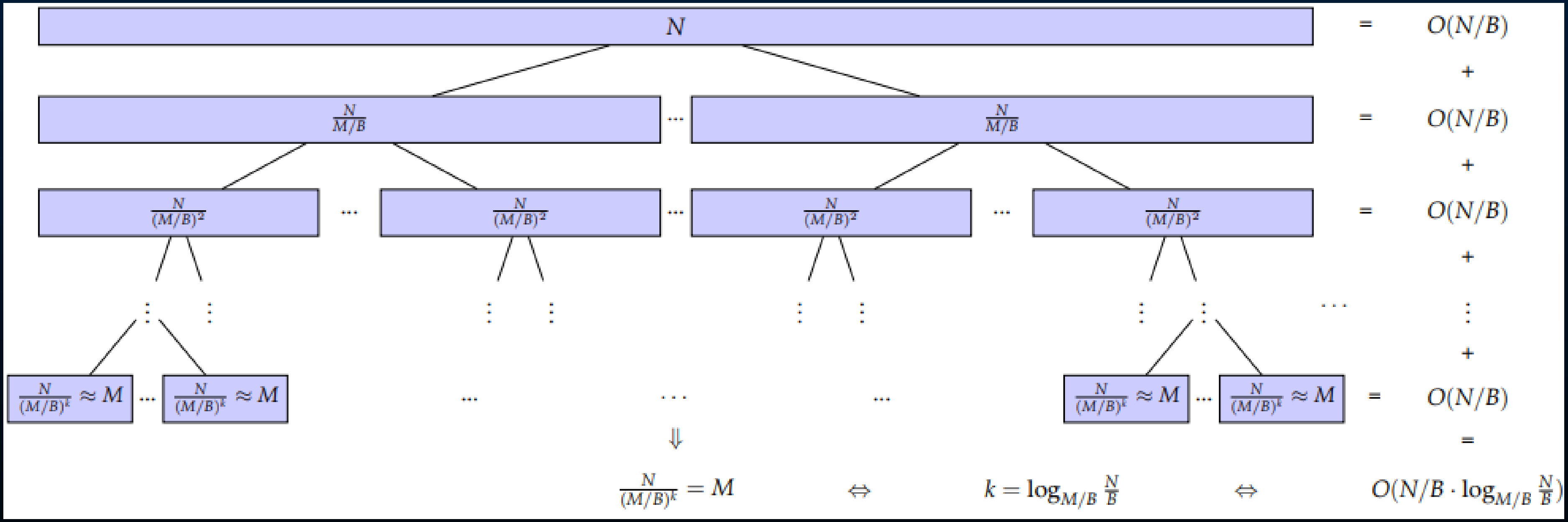
B+Tree的实际性能：

- Typical Fill-Factor：67%
  - 假设最大扇出（fanout）为320，平均fanout = 320×0.67 = 214
- Typical Capacities：
  - Height 4: 214^4 =2,097,273,616个数据
  - Height 3: 214^3 = 9,800,344个数据
- Pages per level：
  - Level 1 = 1 page = 32 KB
  - Level 2 = 214 pages = 6.7 MB
  - Level 3 = 45,796 pages =1431 MB
  - Level 4 ≈ 300 GB

问题6：外存排序问题

外存排序问题（External Sort）：N个数据

- 分成大小为 O(M) 的组，每组可在内存排序，需要 O(M/B) 次I/O
- 排好序的分组，进行归并（Merge），每次可以归并 O(M/B) 个分组
- I/O代价：$O\left(\frac{N}{B}\cdot log_{\frac{M}{B}}\left(\frac{N}{B}\right)\right)$

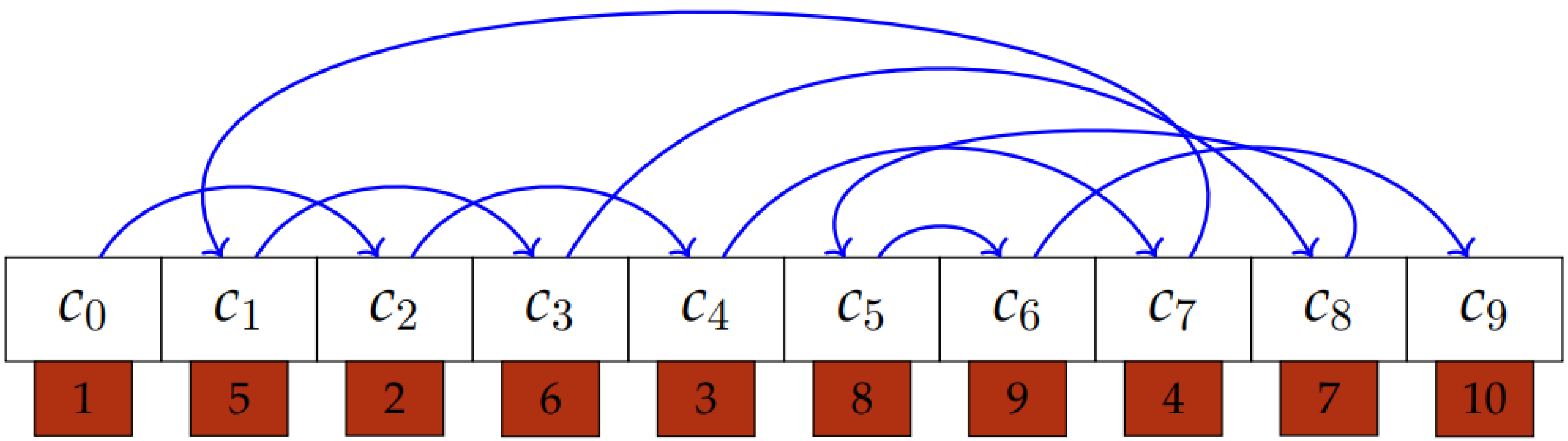


问题7：List Ranking问题

什么是List Ranking问题?

- 一个简单的例子：遍历外存上的链表
- 直观解法的最坏情况：O(N)次I/O

假设N = 10, B = 2, M = 4

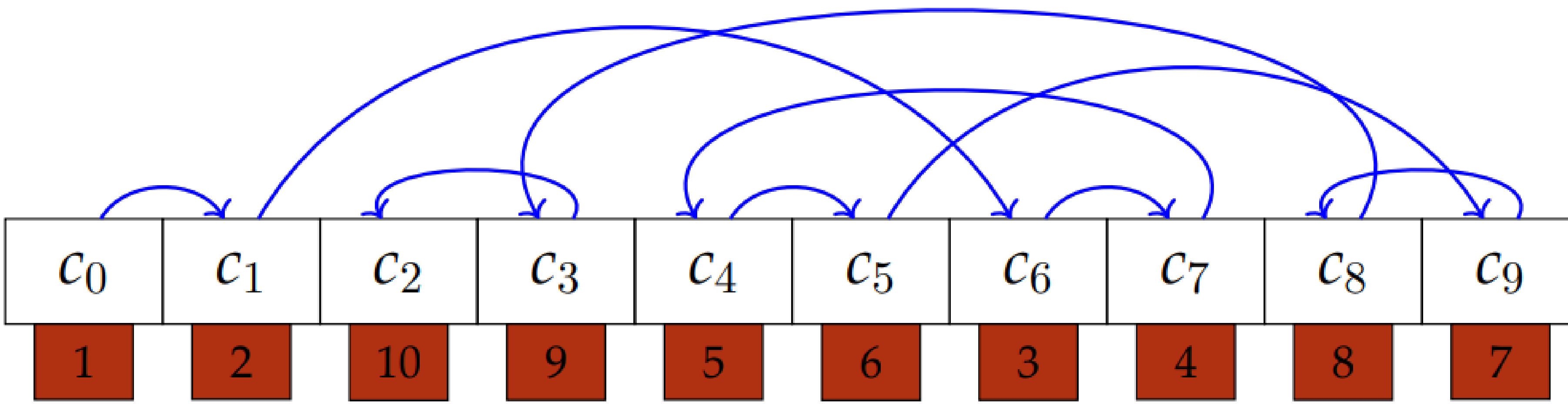


问题7：List Ranking问题

什么是List Ranking问题?

- 一个简单的例子：遍历外存上的链表
- 直观解法的最坏情况：O(N)次I/O

假设N = 10, B = 2, M = 4



问题7：List Ranking问题

定义[List Ranking Problem]

- 给定大小为N的邻接链表L，L存储在连续的外存空间，计算每个节点的rank（在链表中的序号）

定义[General List Ranking Problem]

- 给定大小为N的邻接链表L，L存储在连续的外存空间，L中的每个节点v上存储权重w_v，计算每个节点v的rank（从头节点到v的权重和）

两个观察：

- 如果合并链表上的子序列为一个节点，将权重和做为该节点的权重，不影响前后节点的rank值
- 如果链表大小至多为M，O(M/B)次I/O可解

问题7：List Ranking问题

List Ranking算法：输入大小为N的外存链表L

① 寻找L中的一个顶点独立集X

② 将X中节点“跳过”，构建新的、更小的外存链表L′

③ 递归地求解L′

④ 将X中的节点“回填”，根据L′的rank构建L的rank

算法代价：假设独立集大小为 $\alpha N (0 < \alpha \le 1)$

- ✓ 第1、2、4步可在$O(sort) = O\left(\frac{N}{B} \cdot \log_{M/B}\left(\frac{N}{B}\right)\right)$次 I/O 求解
- ✓ 递归方程：$T(N) = T\big((1-\alpha)N\big) + O\left(\frac{N}{B} \cdot \log_{M/B}\left(\frac{N}{B}\right)\right)$

$$\mathrm{T(N)} = O\left(\frac{N}{B} \cdot \log_{M/B}\left(\frac{N}{B}\right)\right)$$

问题7：List Ranking问题

List Ranking算法：(Step 2)

- ① 将链表L拷贝一份得到链表L′
- ② 将链表L′ 按照节点后继节点的地址（ID）排序
- ③ 同时扫描L′和L获得节点及其后继节点的信息
  - 在L′中，将后继节点属于X的节点指针和权重rewrite
- ④ 将L′ 重新按照地址（ID）排序，并去除X中的节点

算法代价：$O(sort) = O\left(\frac{N}{B} \cdot \log_{M/B}\left(\frac{N}{B}\right)\right)$次I/O

问题7：List Ranking问题

List Ranking算法：(Step 4)

- ① 将X中节点添加到L′中，并按照地址（ID）排序
- ② 同时扫描L′和L，如果节点不属于X，L中rank参照L′更新
- ③ 将链表L 按照节点后继节点的地址（ID）排序
- ④ 同时扫描L′和L获得节点及其后继节点的信息
  - 将L′中属于X的节点rank更新（根据L中前驱节点的rank计算）
- ⑤ 将L重新按照地址（ID）排序，扫描L′和L，将L中属于X的节点rank参照L′更新

算法代价：$O(sort) = O\left(\frac{N}{B} \cdot \log_{M/B}\left(\frac{N}{B}\right)\right)$次I/O

问题7：List Ranking问题

List Ranking算法：(Step 1)

（1）可以用随机化方法实现，$\alpha \approx 1/4$

（2）确定的3-coloring方法，$\alpha \geq 1/3$

- ① 按照节点的ID顺序, 分为forward链表和backward链表
- ② 将forward链表(除去尾部节点)按**red**和**blue**间隔染色
- ③ 将backward链表(除去尾部节点)按**green**和**blue**间隔染色
- ④ 选取染色多的一类节点

算法代价：扫描链表并**维护所有可能的前驱节点**

利用外存最小堆（B树可以实现）

$\Rightarrow O(sort) = O(N/B \cdot \log_{M/B}(N/B))$ 次I/O

进一步深入阅读

- Jeffrey Scott Vitter. *External memory algorithms and data structures: dealing with massive data*. ACM Comput. Surv., 2001, 33, 2, 209–271.
- Arge, L. (2002). *External Memory Data Structures*. In: Abello, J., Pardalos, P.M., Resende, M.G.C. (eds) Handbook of Massive Data Sets. Massive Computing, vol 4. Springer, Boston, MA.
- Vitter, J.S. (2002). *External Memory Algorithms*. In: Abello, J., Pardalos, P.M., Resende, M.G.C. (eds) Handbook of Massive Data Sets. Massive Computing, vol 4. Springer, Boston, MA.

☛ 空间高效大数据算法