Index

Problem

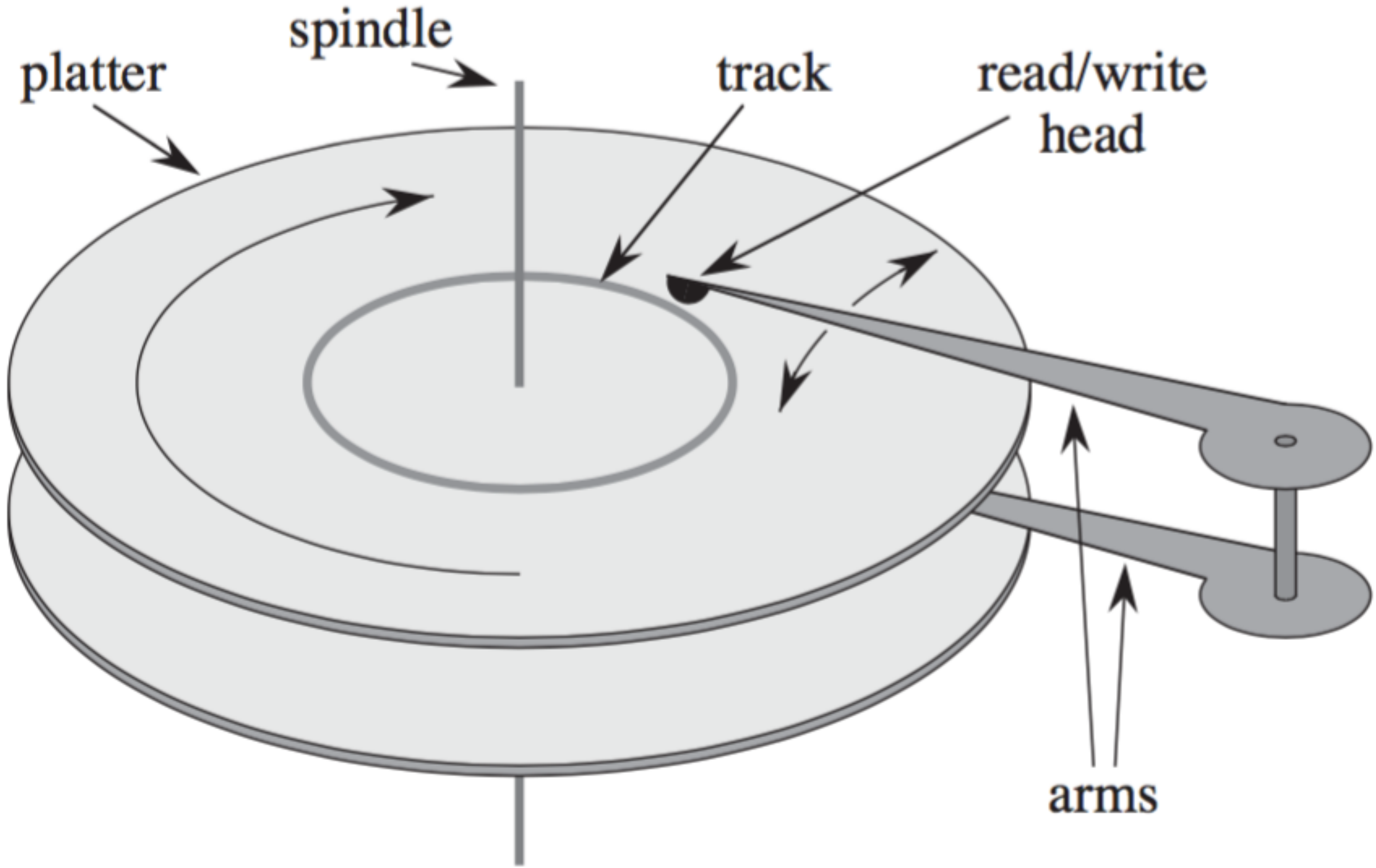
- why use B-tree for MySQL index ?
- Why use an auto-incrementing as primary key ?
- some query compares

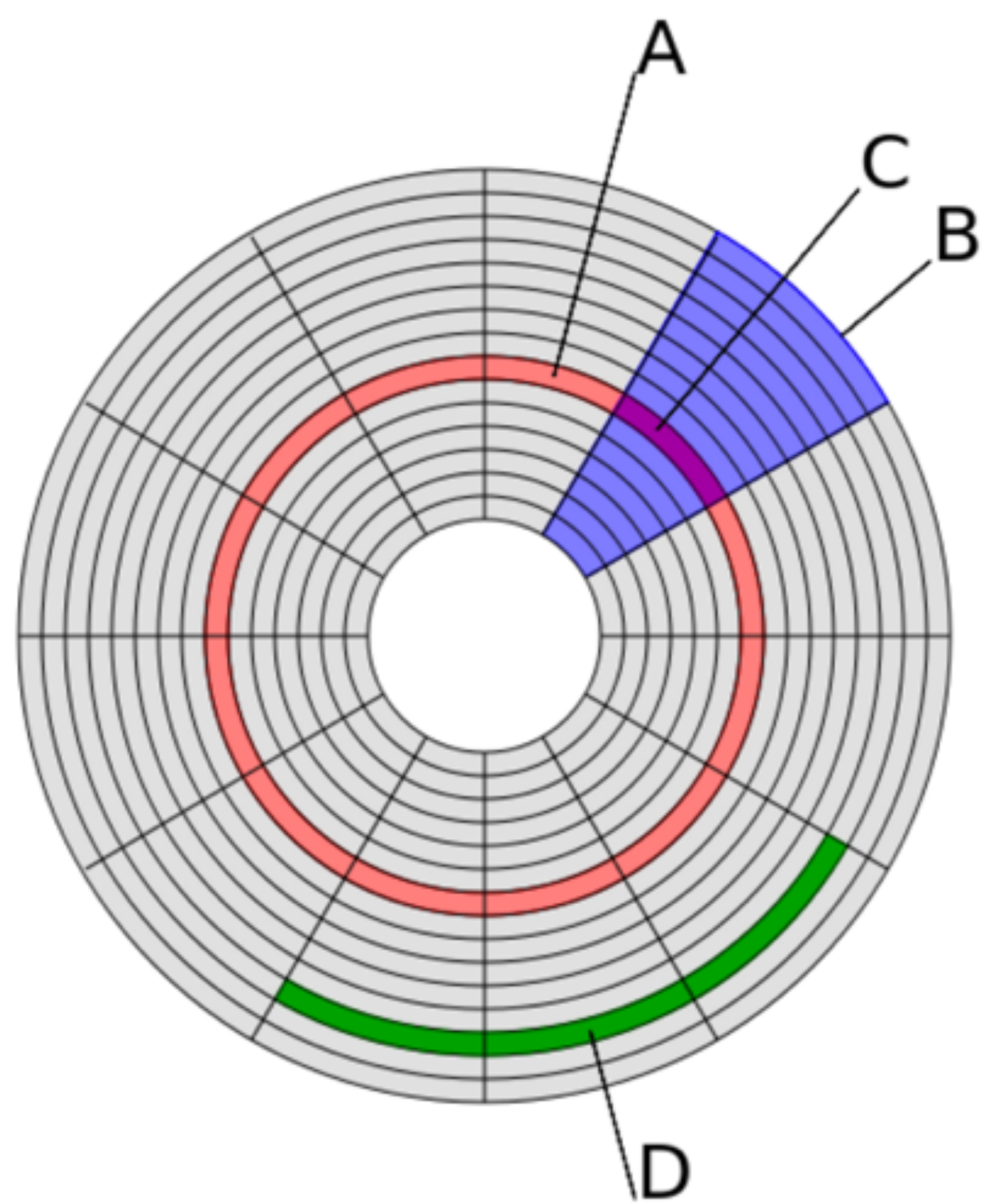
IO time + Memory time

Memory

- 页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。
- 局部性原理，当一个数据被用到时，其附近的数据也通常会马上被使用。
- 程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

Hard Disk Drives


spindle
platter
track
read/write
head
arms



Disk structure:

(A) track

(B) geometrical sector

(C) track sector

(D) cluster

- Sector also known as a page is a subdivision of a track on a magnetic disk or optical disc. Each sector stores a fixed amount of user-accessible data

- A cluster or allocation unit is a unit of disk space allocation for files and directories. To reduce the overhead of managing on-disk data structures, the filesystem does not allocate individual disk sectors by default, but contiguous groups of sectors, called clusters.

- A page, memory page, or virtual page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system.

B-Trees

A ***B-tree*** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,

   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

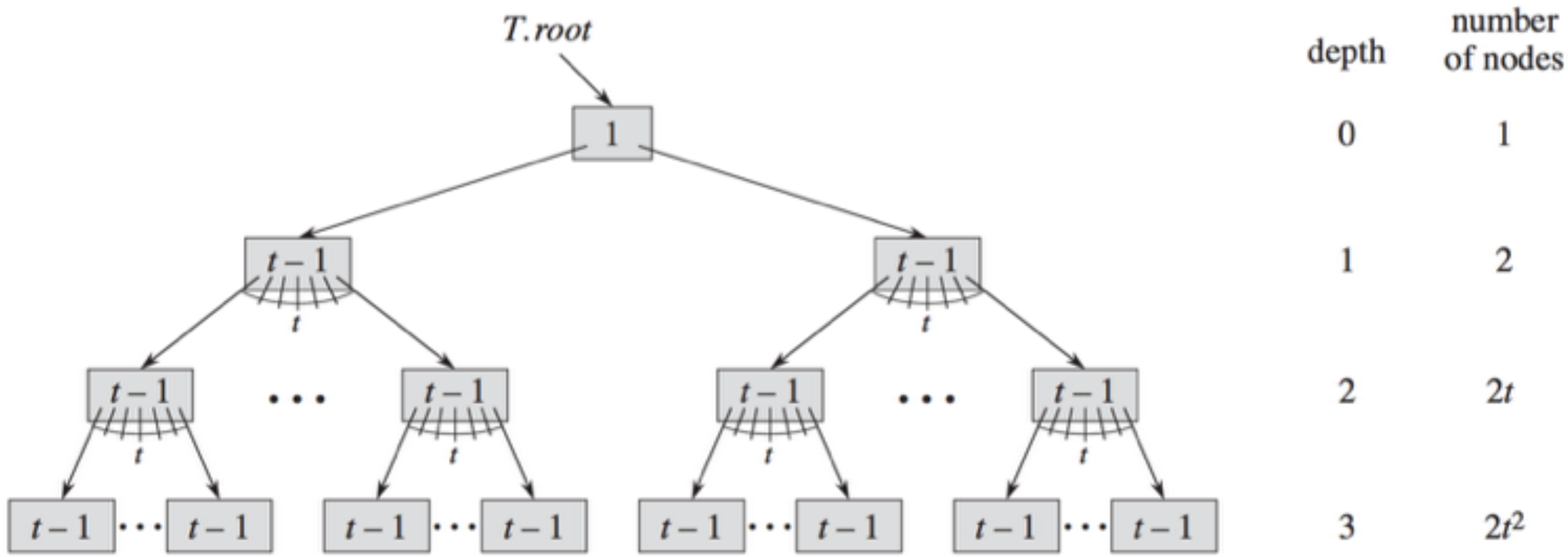
   $$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1} .$$

4. All leaves have the same depth, which is the tree's height h.

5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]


T.root
depth
number of nodes
1
t − 1
t
0
1
2
3
1
2
2t
2t²

$$
\begin{aligned}
n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\
&= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\
&= 2t^h - 1 \,.
\end{aligned}
$$

If n 1, then for any n-key B-tree T of height h and minimum degree t >= 2

$$h \le \log_t \frac{n+1}{2} .$$

In this section, we present the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that we never need to perform a DISK-READ on the root; we do have to perform a DISK-WRITE of the root, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.

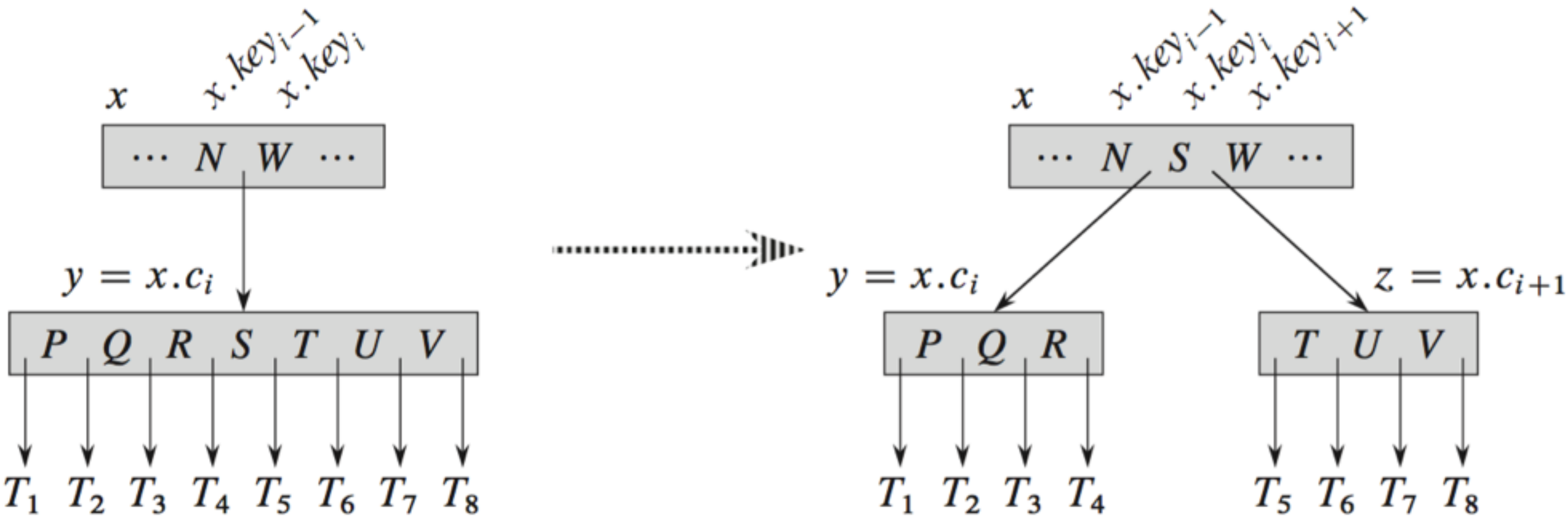
B-tree Search

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

B-tree Create

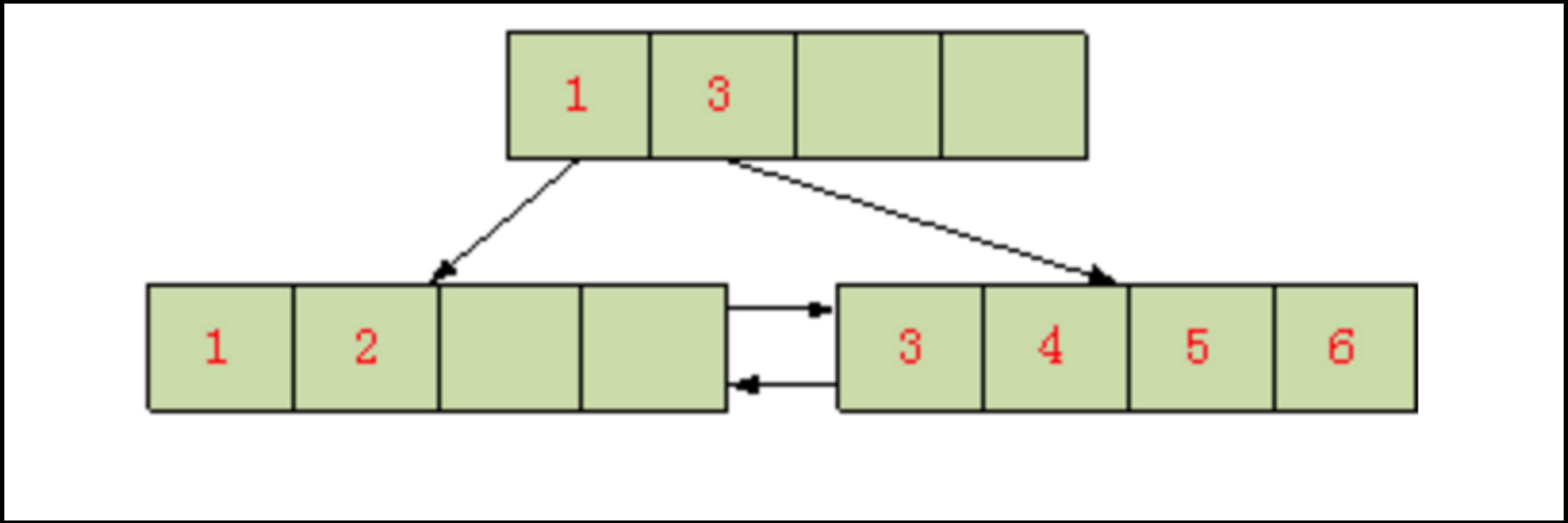
```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

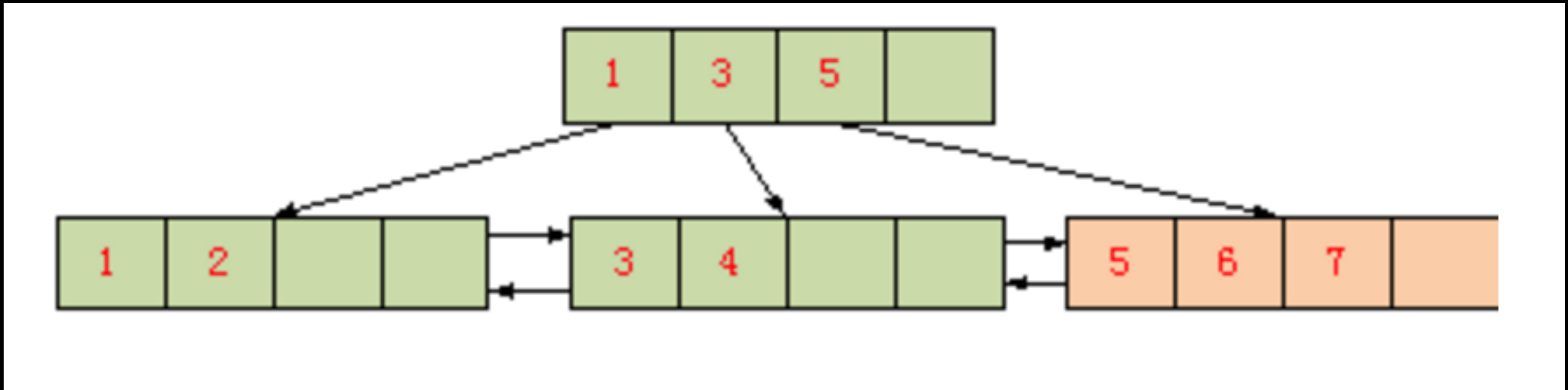
B-tree Insert


x
x.key_{i-1}
x.key_i
… N W …
y = x.c_i
P Q R S T U V
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8
x
x.key_{i-1}
x.key_i
x.key_{i+1}
… N S W …
y = x.c_i
z = x.c_{i+1}
P Q R
T U V
T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8

```
B-TREE-SPLIT-CHILD(x, i)
z = ALLOCATE-NODE()
y = x.c_i
z.leaf = y.leaf
z.n = t - 1
for j = 1 to t - 1
    z.key_j = y.key_{j+t}
if not y.leaf
    for j = 1 to t
        z.c_j = y.c_{j+t}
y.n = t - 1
for j = x.n + 1 downto i + 1
    x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i
    x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

- B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因。
- B树的好处，就是成功查询特别有利，因为树的高度总体要比B+树矮。不成功的情况下，B树也比B+树稍稍占一点点便宜。
- 有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走，前提是需要对query做统计，而且要对key做一些变化。
- B树也好B+树也好，根或者上面几层因为被反复query，所以这几块基本都在内存中，不会出现读磁盘IO，一般已启动的时候，就会主动换入内存。


1
3
1
2
3
4
5
6



1
3
5
1
2
3
4
5
6
7

- 按照原页面中50%的数据量进行分裂，针对当前这个分裂操作，3，4记录保留在原有页面，5，6记录，移动到新的页面。最后将新纪录7插入到新的页面中

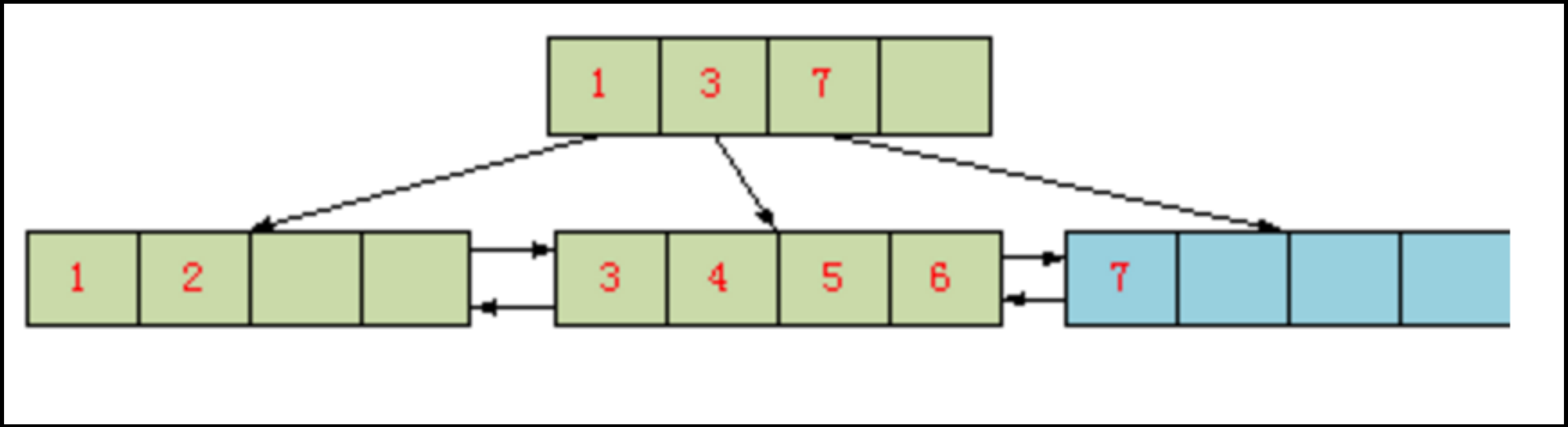
- 50%分裂策略的优势：

    - 分裂之后，两个页面的空间利用率是一样的；如果新的插入是随机在两个页面中挑选进行，那么下一次分裂的操作就会更晚触发；

- 50%分裂策略的劣势：

    - 空间利用率不高：按照传统50%的页面分裂策略，索引页面的空间利用率在50%左右；

    - 分裂频率较大：针对如上所示的递增插入(递减插入)，每新插入两条记录，就会导致最右的叶页面再次发生分裂；



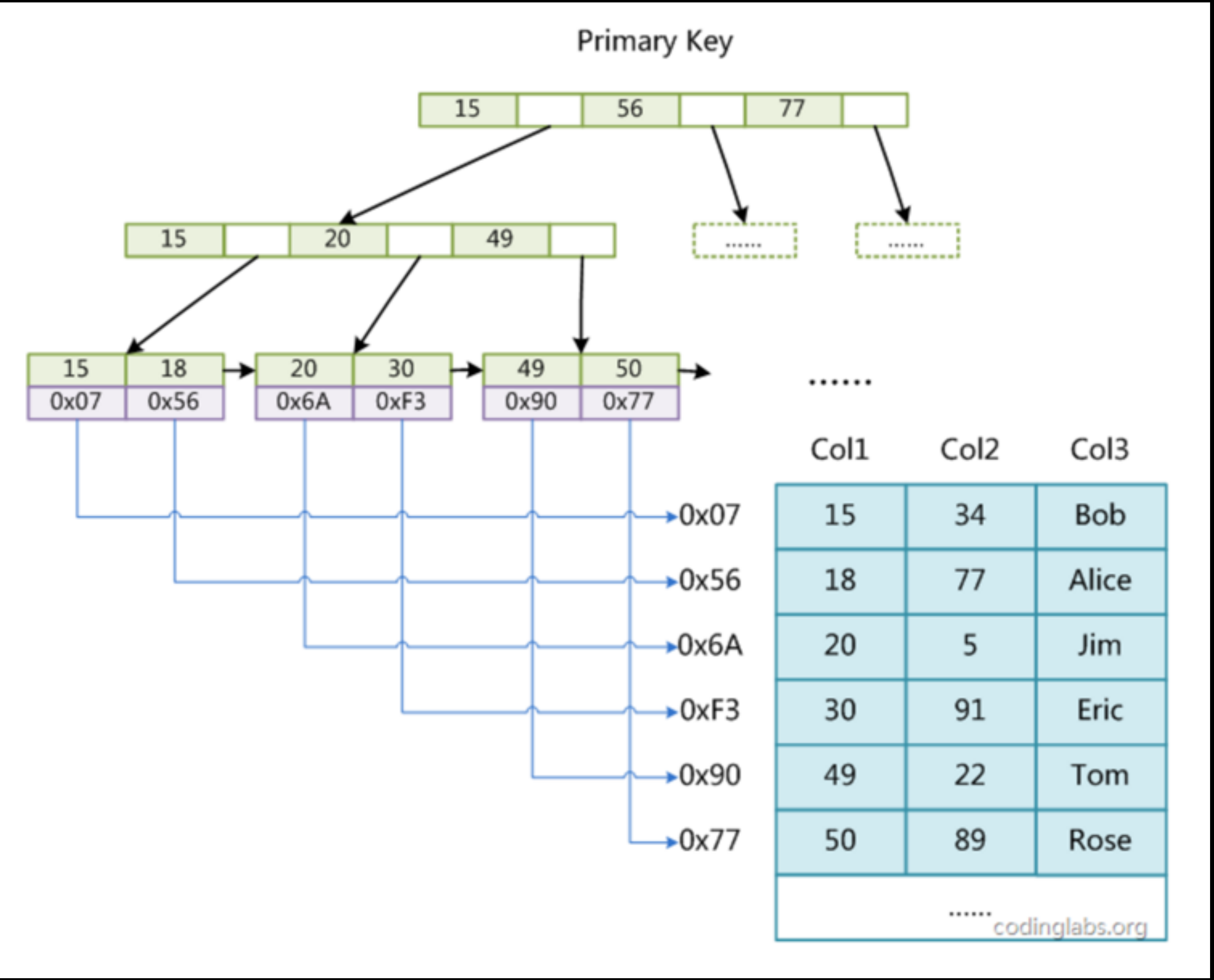
1
3
7
1
2
3
4
5
6
7

- 新的分裂策略，在插入7时，不移动原有页面的任何记录，只是将新插入的记录7写到新页面之中；
- 原有页面的利用率，仍旧是100%；
- 优化分裂策略的优势：
    - 索引分裂的代价小：不需要移动记录；
    - 索引分裂的概率降低：如果接下来的插入，仍旧是递增插入，那么需要插入4条记录，才能再次引起页面的分裂。相对于50%分裂策略，分裂的概率降低了一半；
    - 索引页面的空间利用率提高：新的分裂策略，能够保证分裂前的页面，仍旧保持100%的利用率，提高了索引的空间利用率；

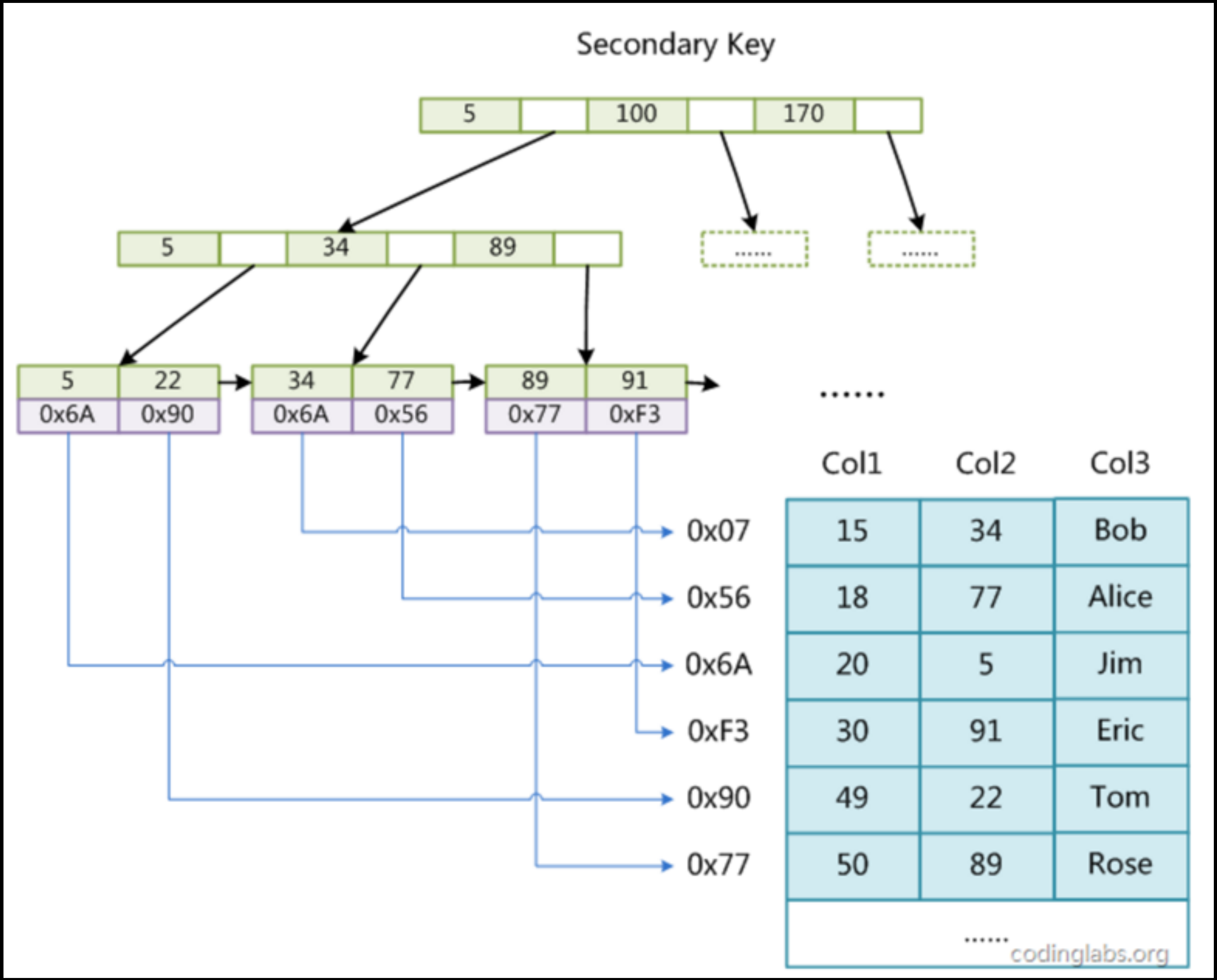
在InnoDB的实现中，为每个索引页面维护了一个上次插入的位置，以及上次的插入是递增/递减的标识。根据这些信息，InnoDB能够判断出新插入到页面中的记录，是否仍旧满足递增/递减的约束，若满足约束，则采用优化后的分裂策略；若不满足约束，则退回到50%的分裂策略。

- In MyISAM data pointers point to physical offset in the data file
  - All indexes are essentially equivalent
- In Innodb
  - PRIMARY KEY (Explicit or Implicit) - stores data in the leaf pages of the index, not pointer
  - Secondary Indexes – store primary key as data pointer

MyISAM

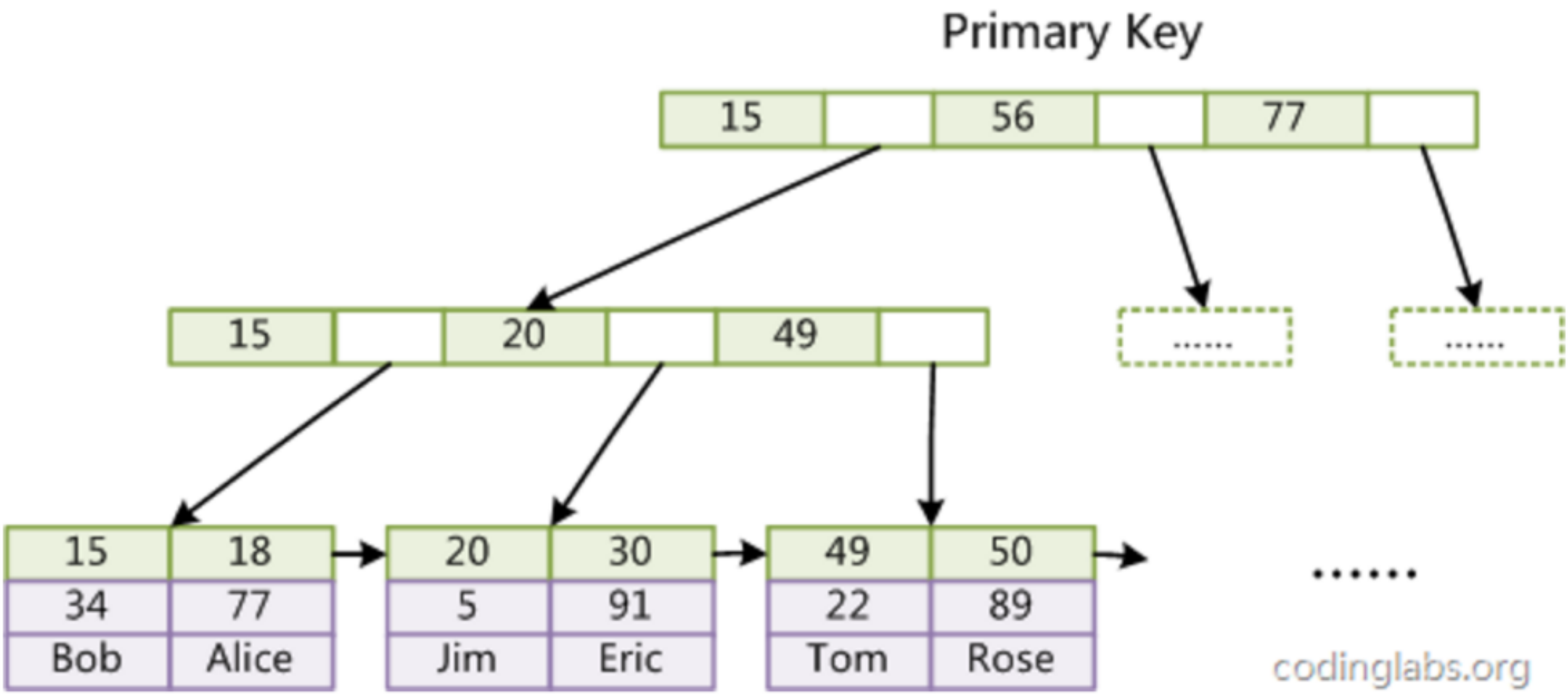

Primary Key
15
56
77
15
20
49
......
......
15
18
0x07
0x56
20
30
0x6A
0xF3
49
50
0x90
0x77
......
0x07
0x56
0x6A
0xF3
0x90
0x77
Col1
Col2
Col3
15 34 Bob
18 77 Alice
20 5 Jim
30 91 Eric
49 22 Tom
50 89 Rose
......
codinglabs.org

MyISAM



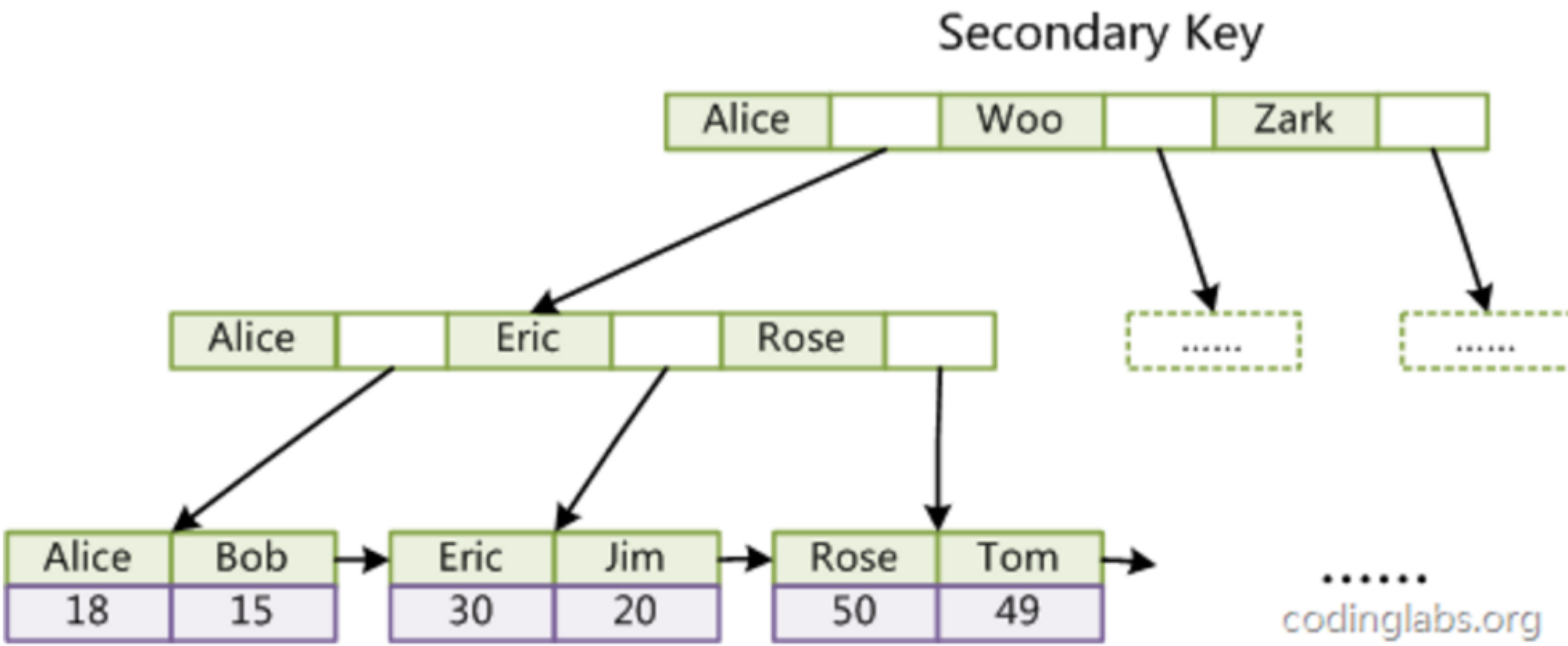
Secondary Key
5
100
170
5
34
89
......
......
5
22
34
77
89
91
0x6A
0x90
0x6A
0x56
0x77
0xF3
......
0x07
0x56
0x6A
0xF3
0x90
0x77
Col1
Col2
Col3
15
34
Bob
18
77
Alice
20
5
Jim
30
91
Eric
49
22
Tom
50
89
Rose
......
codinglabs.org

Innodb



Primary Key
15
56
77
15
20
49
......
......
15
18
34
77
Bob
Alice
20
30
5
91
Jim
Eric
49
50
22
89
Tom
Rose
......
codinglabs.org

Innodb


Secondary Key
Alice
Woo
Zark
Alice
Eric
Rose
......
......
Alice
Bob
18
15
Eric
Jim
30
20
Rose
Tom
50
49
......
codinglabs.org

- 减少磁盘的操作次数，便于内存管理
- 充分利用内存和磁盘
- 减少比较次数

- 节点分裂，数据移动，提高节点空间利用率
- 减少磁盘空间的使用
- 提高查询效率（特定字段）

- SELECT to_date FROM employees.titles WHERE from_date=‘1986-06-26'
- SELECT emp_no FROM employees.titles WHERE from_date=‘1986-06-26'
- SELECT * FROM employees.titles WHERE from_date='1986-06-26';

查询出的字段是否在正在使用的索引中的对比

```
110 | 0.00035000 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26'
111 | 0.00026900 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26'
112 | 0.00035800 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26'
113 | 0.00035000 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26'
114 | 0.00036400 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26'
115 | 0.00028200 | SELECT emp_no FROM employees.titles WHERE from_date='1986-06-26'
116 | 0.00028400 | SELECT emp_no FROM employees.titles WHERE from_date='1986-06-26'
117 | 0.00028400 | SELECT emp_no FROM employees.titles WHERE from_date='1986-06-26'
118 | 0.00028200 | SELECT emp_no FROM employees.titles WHERE from_date='1986-06-26'
119 | 0.00028400 | SELECT emp_no FROM employees.titles WHERE from_date='1986-06-26'
```

查询出的字段不在索引时一个字段与所有字段的对比

```
87 | 0.00029400 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
88 | 0.00043600 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
89 | 0.00035100 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
90 | 0.00033800 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
91 | 0.00037900 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
92 | 0.00030300 | SELECT to_date FROM employees.titles WHERE from_date='1986-06-26' |
93 | 0.00035100 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
94 | 0.00047800 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
95 | 0.00038400 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
96 | 0.00031500 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
97 | 0.00035000 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
98 | 0.00043700 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
99 | 0.00038100 | SELECT * FROM employees.titles WHERE from_date='1986-06-26'       |
-----+------------+-------------------------------------------------------------------+
```

- The index earns one star if it places relevant rows adjacent to each other,
- a second star if its rows are sorted in the order the query needs
- a final star if it contains all the columns needed for the query.

- https://www.wikiwand.com/en/Disk_sector
- https://www.wikiwand.com/en/Data_cluster
- https://www.wikiwand.com/en/Page_(computer_memory)
- http://hedengcheng.com/?p=525
- http://www.slideshare.net/myxplain/mysql-indexing-best-practices-for-mysql
- http://www.file-recovery.com/recovery-hard-disk-drive-sectors.htm

郑重证明：此文资料和图表来自于《算法导论》何登成老师的 blog 张洋老师的 blog 版权是属于他们的，我只是个搬运工，没有任何原创。