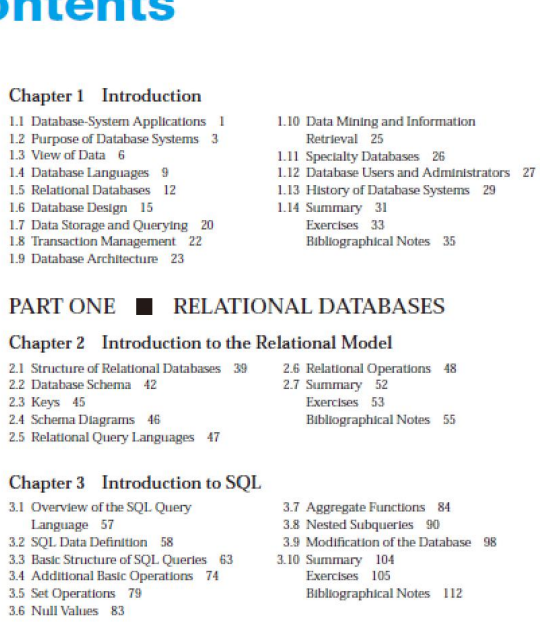
Chapter 11 索引与散列

陈健 教授

ellachen@scut.edu.cn

11.1 基本概念



Contents

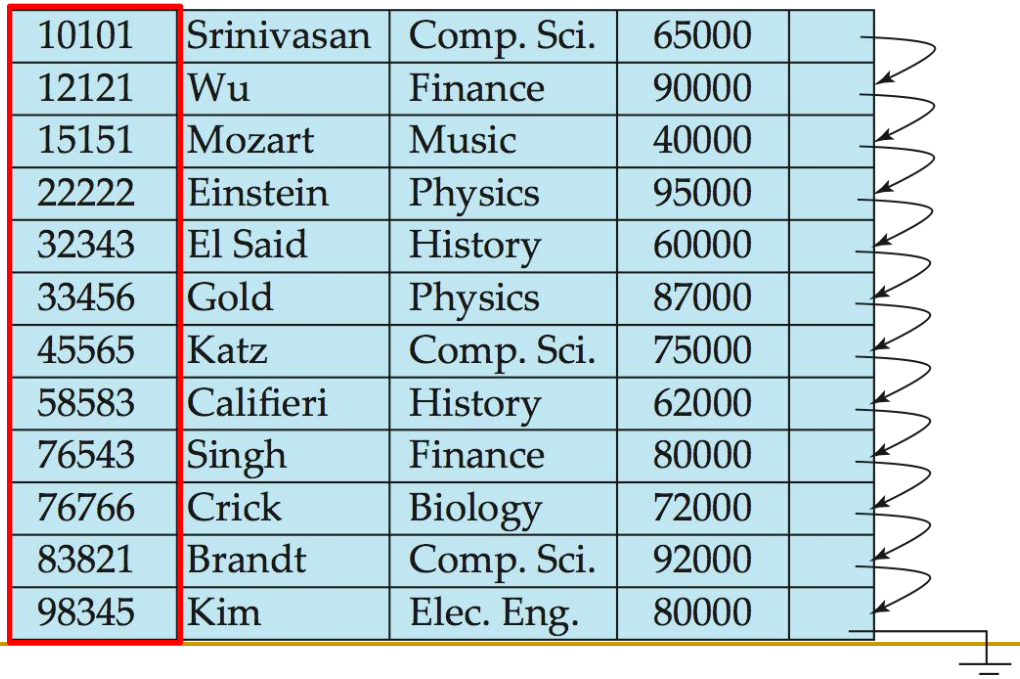
- 文件的索引就像书本的目录一样
- 索引是一个数据结构，它对存储在磁盘上数据记录进行分类整理，目的是获得更好的查询速度。
- 有两种基本的索引类型：
  - 顺序索引（Ordered indices）：按某个属性值排序
  - 散列索引（Hash indices）：基于哈希值分布到不同散列桶中
- 索引的评价：
  - 访问类型（Access types）、访问时间（Access time）、插入时间（Insertion time）、删除时间（Deletion time）、空间开销（Space overhead）

11.2 顺序索引（Ordered Indices）

- 按搜索码在数据文件中是否有序
  - 主索引（Primary index）
  - 辅助索引（Secondary index）
- 按索引项是否完备
  - 稠密索引（Dense index）
  - 稀疏索引（Sparse Index）
- 多级索引（Multilevel Index）

顺序文件

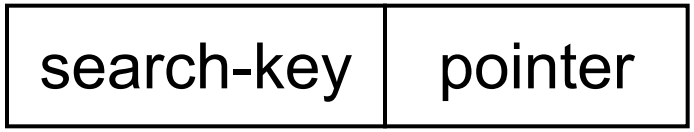
- **顺序文件（ Sequential File ）**
  - Suitable for applications that require sequential processing of the entire file
  - The records in the file are ordered by a search-key



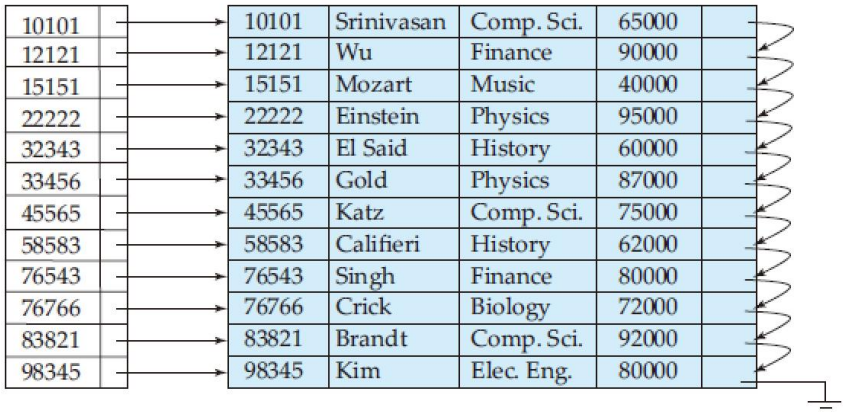
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

1、主索引

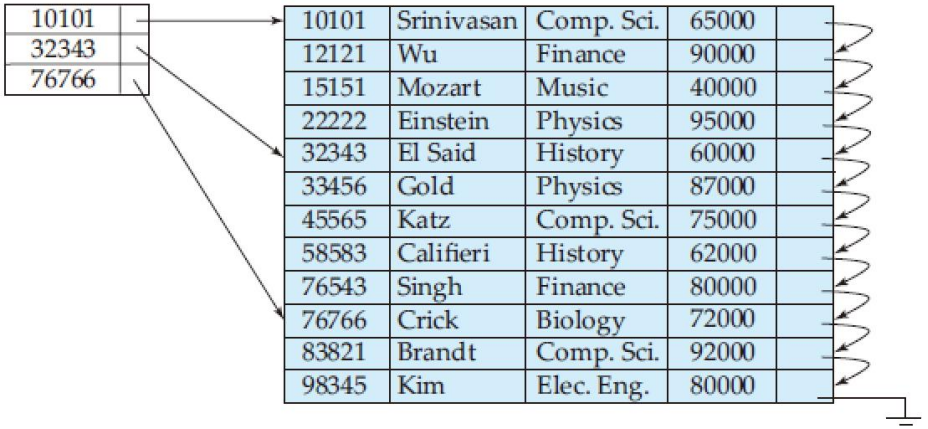
- 主索引(Primary index)是建立在有序文件中的有序字段上；也称为聚簇索引（Clustering Index）。
- 主索引本身也是一个有序文件，它的每个索引入口(index entry，或者索引记录index record)包含两个字段的定长记录
  - 第1个字段与数据文件的搜索码字段(search key)有相同的数据类型
  - 第2个字段是指向一个数据块的指针(块地址和块内偏移)



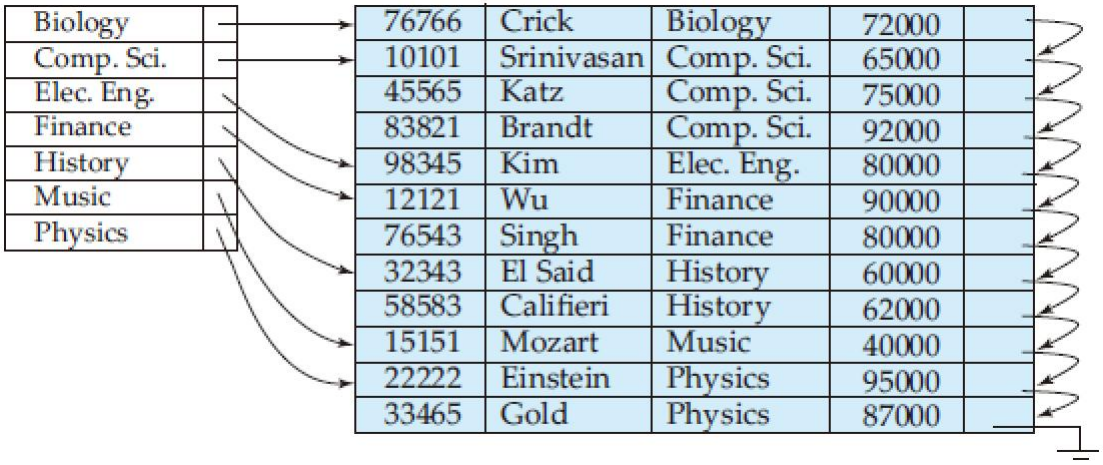
1、主索引



10101	10101	Srinivasan	Comp. Sci.	65000
12121	12121	Wu	Finance	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	Comp. Sci.	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Finance	80000
76766	76766	Crick	Biology	72000
83821	83821	Brandt	Comp. Sci.	92000
98345	98345	Kim	Elec. Eng.	80000



10101	10101	Srinivasan	Comp. Sci.	65000
32343	12121	Wu	Finance	90000
76766	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	32343	El Said	History	60000
	33456	Gold	Physics	87000
	45565	Katz	Comp. Sci.	75000
	58583	Califieri	History	62000
	76543	Singh	Finance	80000
	76766	Crick	Biology	72000
	83821	Brandt	Comp. Sci.	92000
	98345	Kim	Elec. Eng.	80000

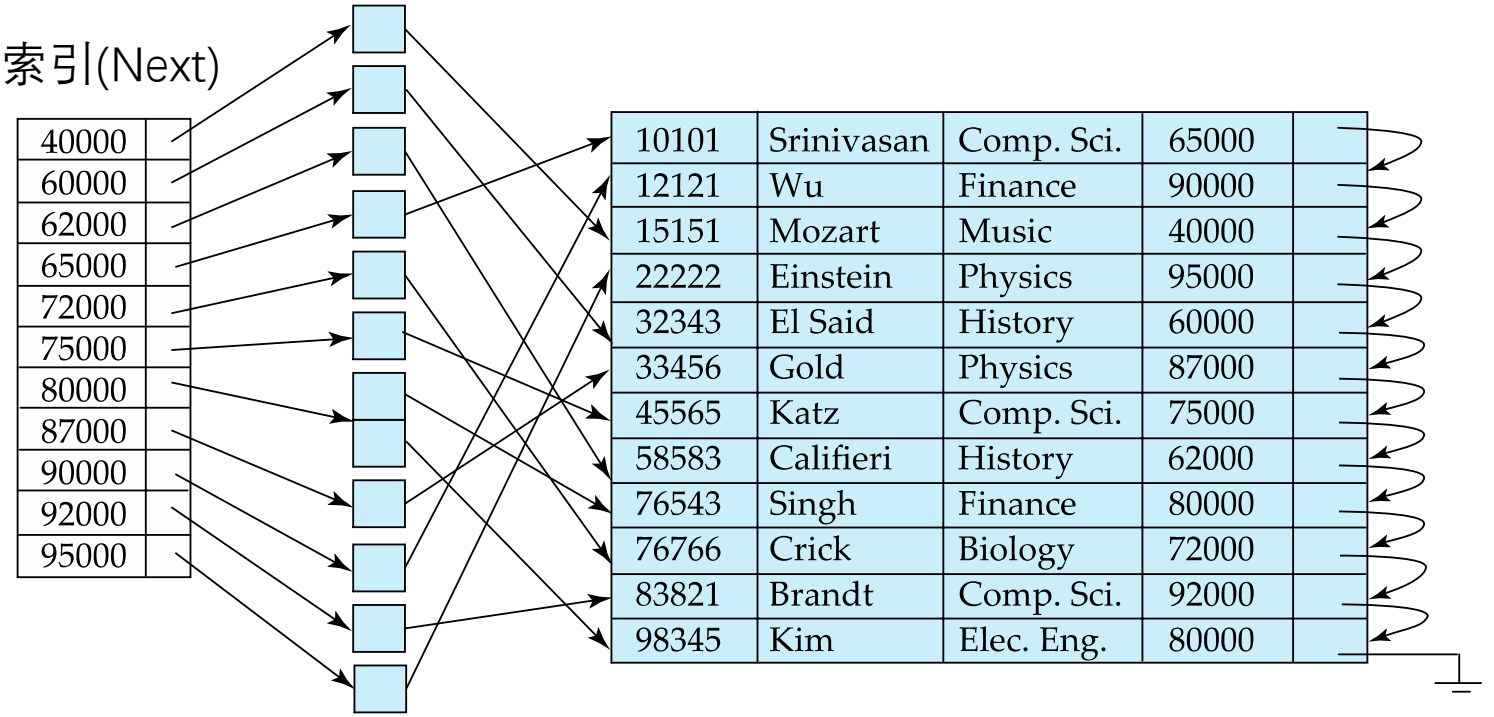


Biology	76766	Crick	Biology	72000
Comp. Sci.	10101	Srinivasan	Comp. Sci.	65000
Elec. Eng.	45565	Katz	Comp. Sci.	75000
Finance	83821	Brandt	Comp. Sci.	92000
History	98345	Kim	Elec. Eng.	80000
Music	12121	Wu	Finance	90000
Physics	76543	Singh	Finance	80000
	32343	El Said	History	60000
	58583	Califieri	History	62000
	15151	Mozart	Music	40000
	22222	Einstein	Physics	95000
	33465	Gold	Physics	87000

Primary Indices on sequential *instructor* file

2、辅助索引

- 索引中搜索码的序与数据文件中对应字段的不同，也称为非聚簇索引(non-clustering index)。
- 索引项的指针指向一个bucket，其中包含指向具有该特定搜索码值的所有实际记录的指针
- 必须是稠密索引(Next)

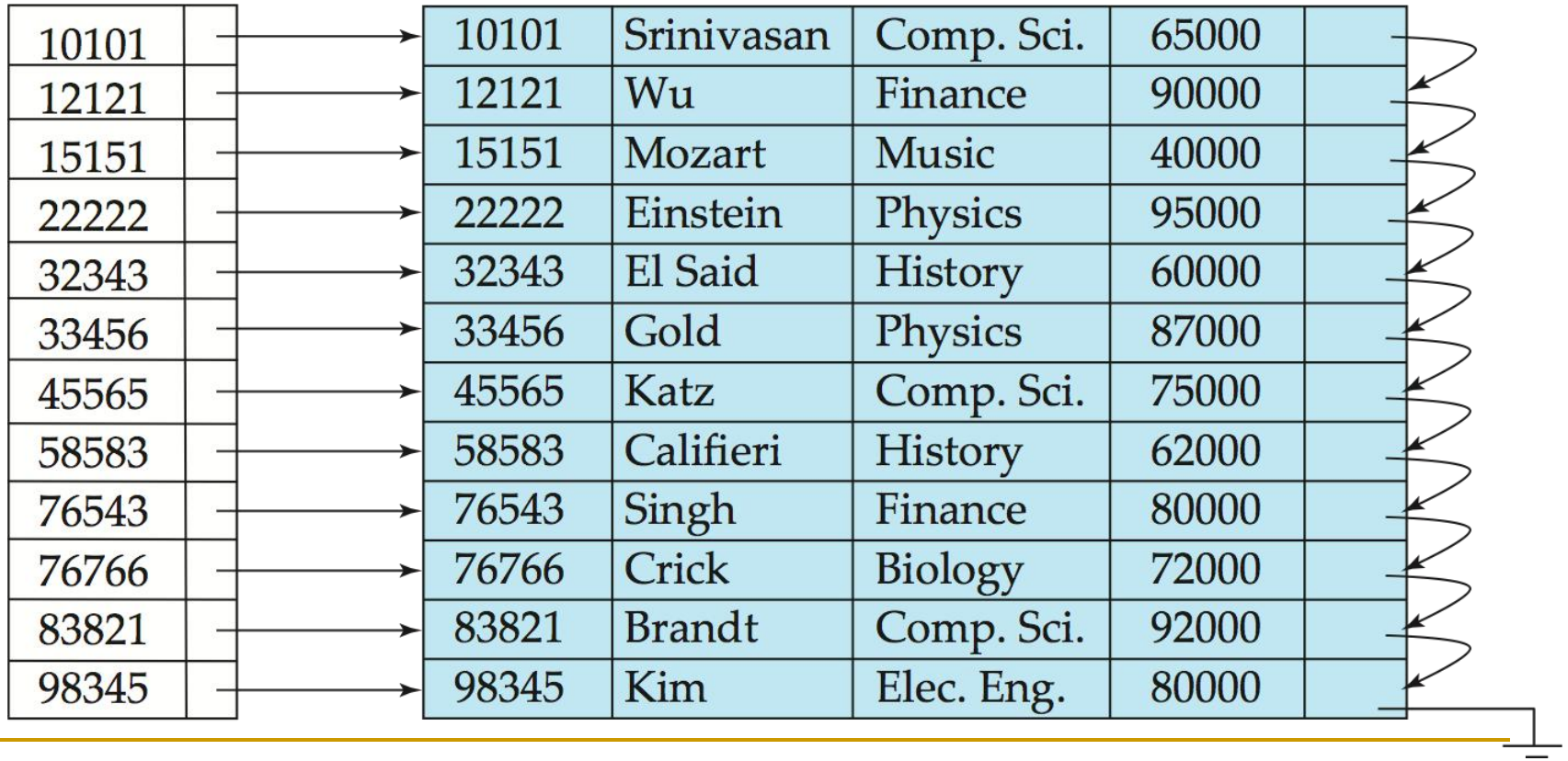


Secondary index on *salary* field of *instructor*

3、稠密索引(Dense Index)

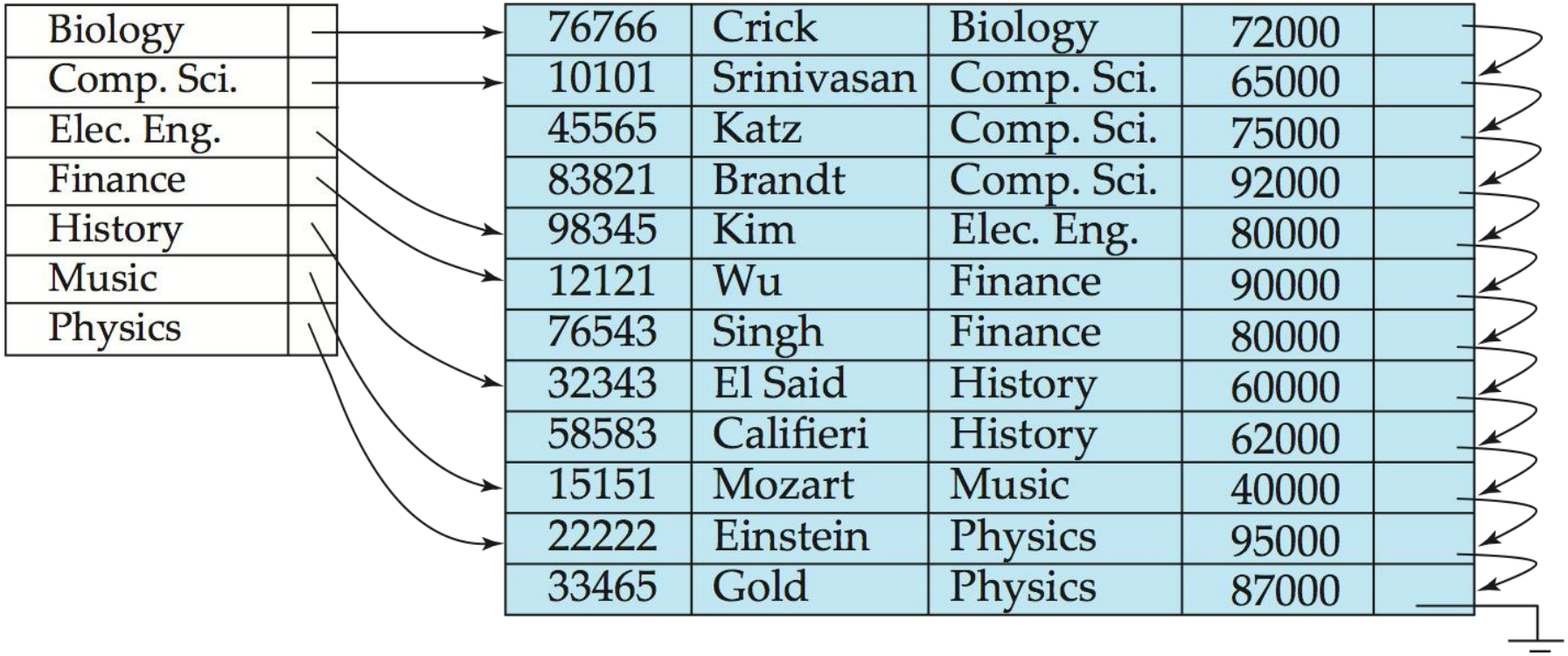
- 每个搜索码值都有一个索引项
- 索引项按搜索码排序

查找：
查找索引项，跟踪指针即可



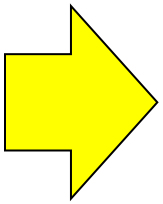
3、稠密索引(Dense Index)

- 索引项包括搜索码值以及指向具有该搜索码值的第一条数据记录的指针
- 具有相同搜索码值的其余记录顺序的存储在第一条记录之后



3、稠密索引(Dense Index)

- 为什么使用稠密索引?
  - 记录通常比索引项要大
  - 可以快速查找索引（例如二分法）
  - 索引可以常驻内存
  - 要搜索码值为K的记录是否存在，不需要访问磁盘数据块
- 稠密索引缺点?
  - 索引占用太多空间

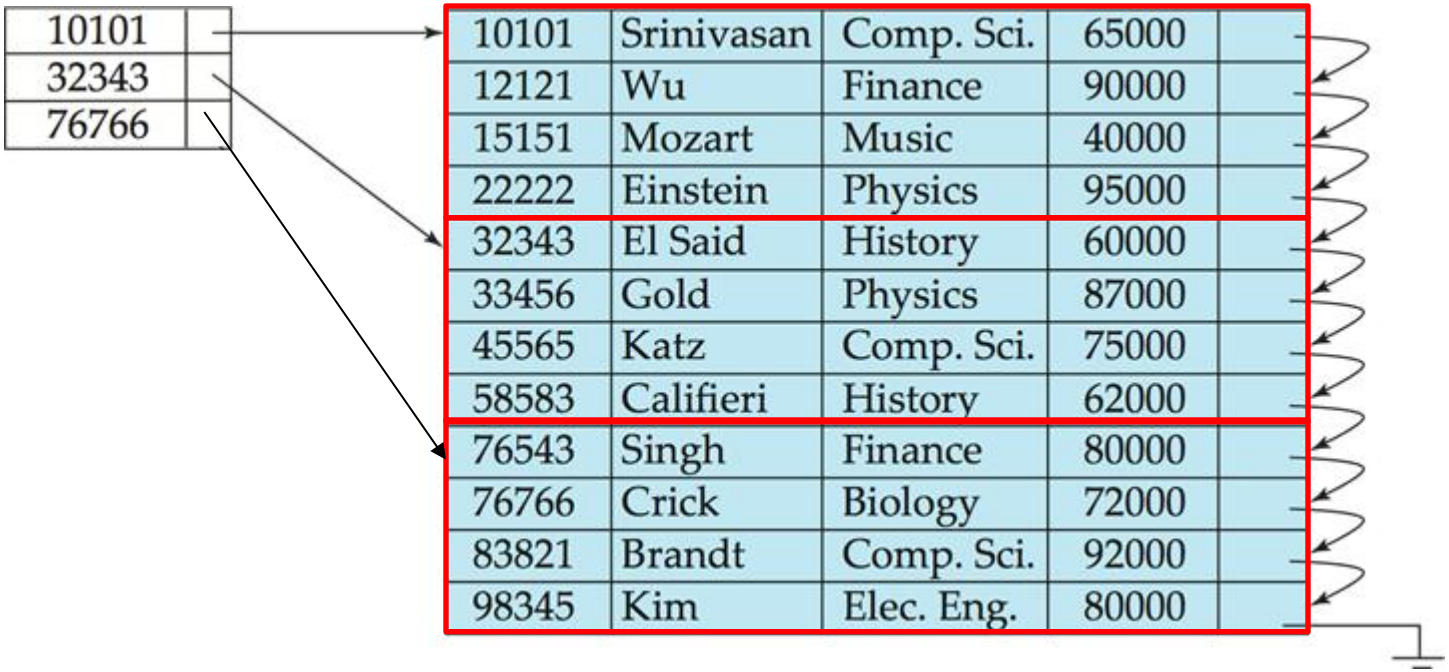


用稀疏索引改进

4、稀疏索引(Sparse Index)

- 仅部分记录有索引项
- 一般情况：为每个数据块的第一个记录建立索引

查找：
查找索引项，跟踪指针找到数据块，并在块中顺序查找

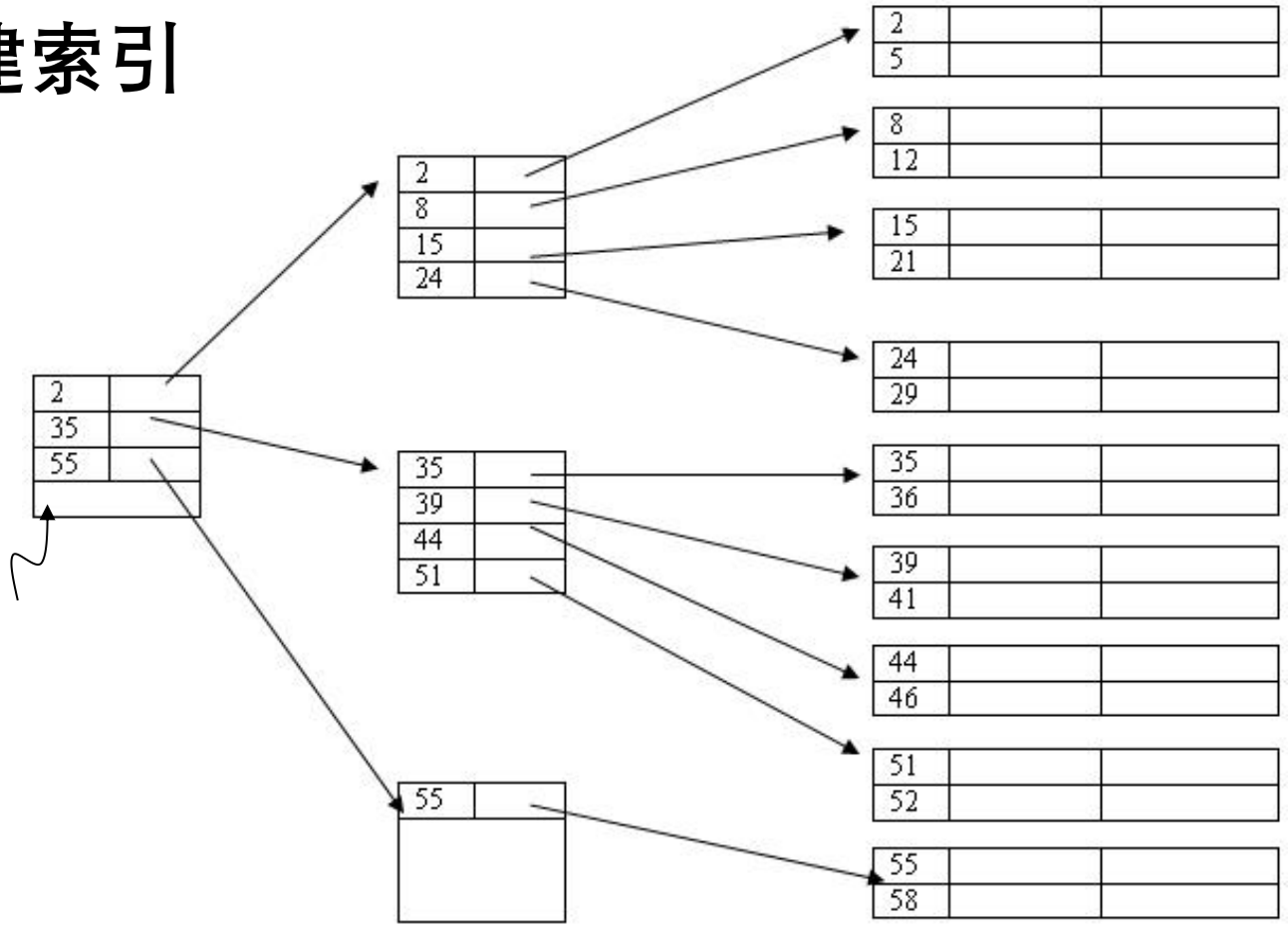


4、稀疏索引(Sparse Index)

- 有何优点？
  - 节省了索引空间
  - 对同样的记录，稀疏索引可以使用更少的索引项
- 有何缺点？
  - 对于“是否存在码值为K的记录？”，需要访问磁盘数据块
- 只有主索引才能采用稀疏索引的形式

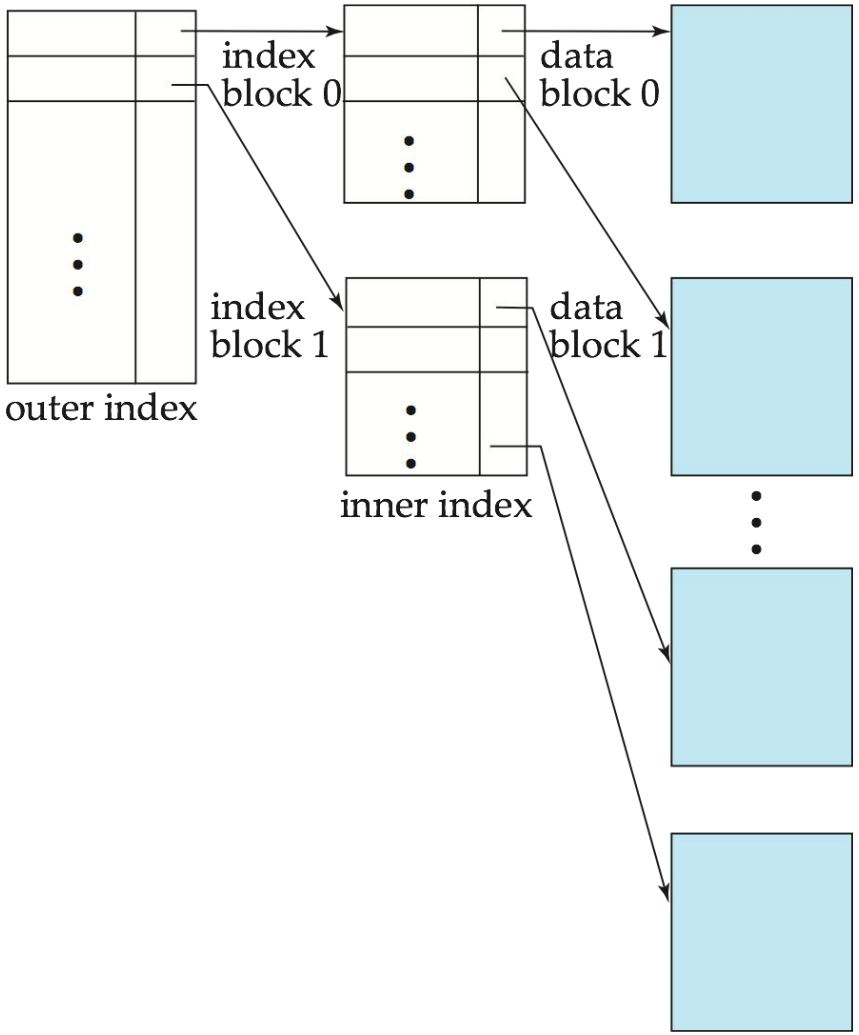
5、多级索引(Multi-level Index)

- **索引上再建索引**



5、多级索引(Multi-level Index)

- **多级索引的好处?**
  - 一级索引可能还太大而不能常驻内存
  - 二级索引更小，可以常驻内存
  - 减少磁盘I/O次数

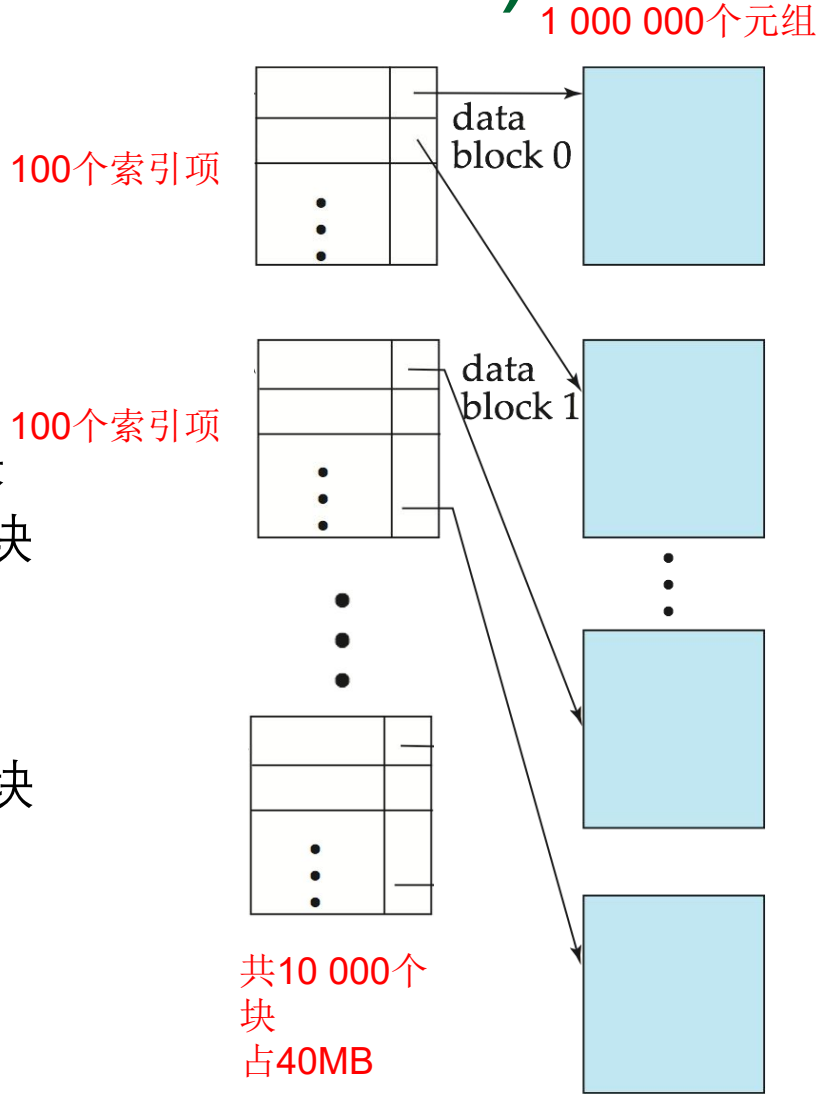


5、多级索引(Multi-level Index)

- 假设我们在具有1 000 000个元组的关系上建立一个稠密索引。由于索引项比数据记录小，所以我们假设一个4KB的块上有100个索引项。因此，该索引占了10 000个块，即40MB的空间。
- 假设一个更大的关系中有100 000 000个元组，则索引将占用1 000 000个块，即4GB的空间。
- 这样大的索引以顺序文件的形式存储在磁盘上
  - 如果索引小，可以放入主存，则搜索时间很短
  - 如果索引大，放不下主存，则需要I/O去装载索引

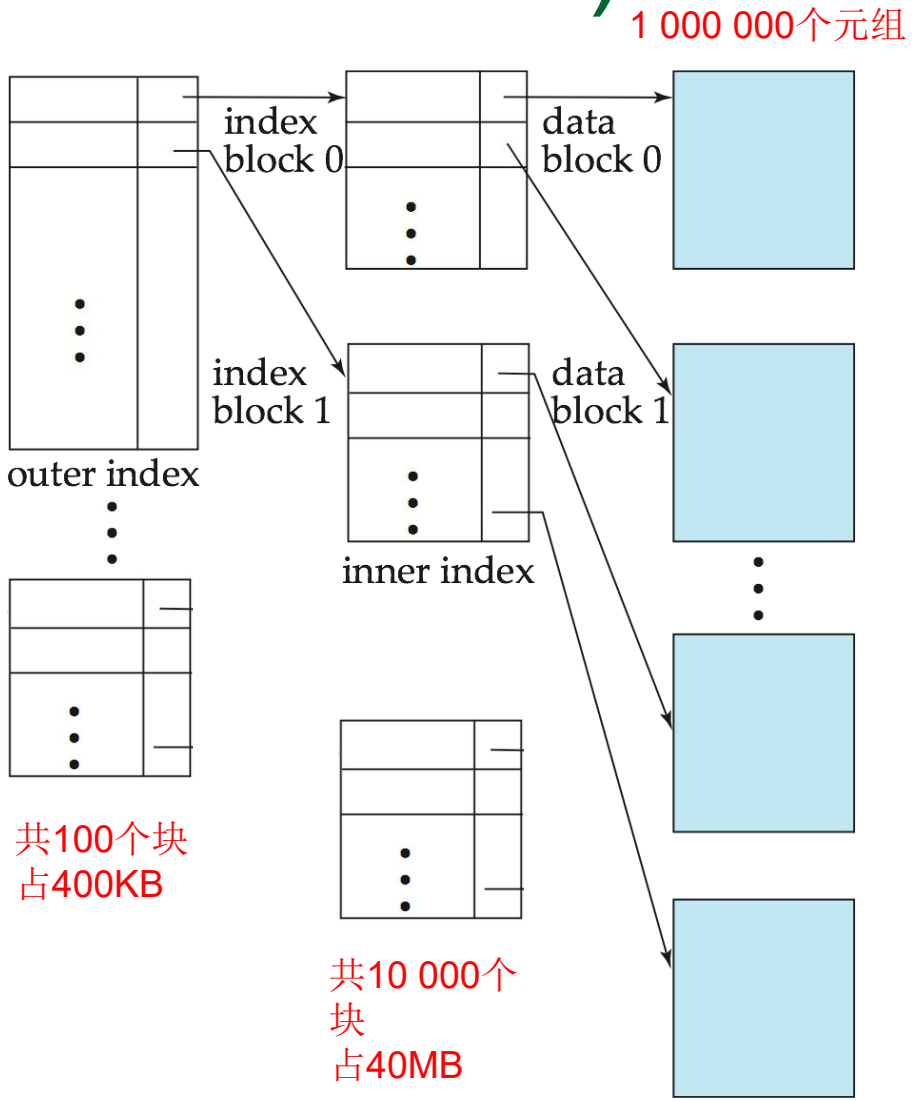
5、多级索引(Multi-level Index)

- 按一级索引查找：
  - 顺序查找：
    - 最好：1个索引块+1个数据块
    - 平均：5000个索引块+1个数据块
    - 最坏：10000次索引块+1个数据块
  - 二分查找：
    - $Log_2 10000 \approx 13$ 次I/O定位索引块
    - 1个数据块
    - 共约14次I/O



5、多级索引(Multi-level Index)

- 按二级索引查找：
  - outer index(不在主存)
    - 顺序：
      - 最好：1
      - 平均： 50
      - 最坏：100
    - 二分：$Log_2100 \approx 7$ 次
  - 但一般outer index比较小，常驻主存，此时查询代价：0
  - inner index：1
  - 数据块：1
  - 大约2次I/O



6、索引顺序文件上的修改

- **索引顺序文件上的修改动作**
  - 创建或删除一个空存储块
  - 创建或删除一个溢出块
  - 插入一条记录到一个空块中
  - 删除记录
  - 将记录移动相邻的块中

6、索引顺序文件上的修改

行为	稠密索引	稀疏索引
创建空溢出块	无	无
删除空溢出块	无	无
创建空顺序块	无	插入
删除空顺序块	无	删除
插入记录	插入	更新?
删除记录	删除	更新?
移动记录	更新	更新?

11.3 B+树索引文件

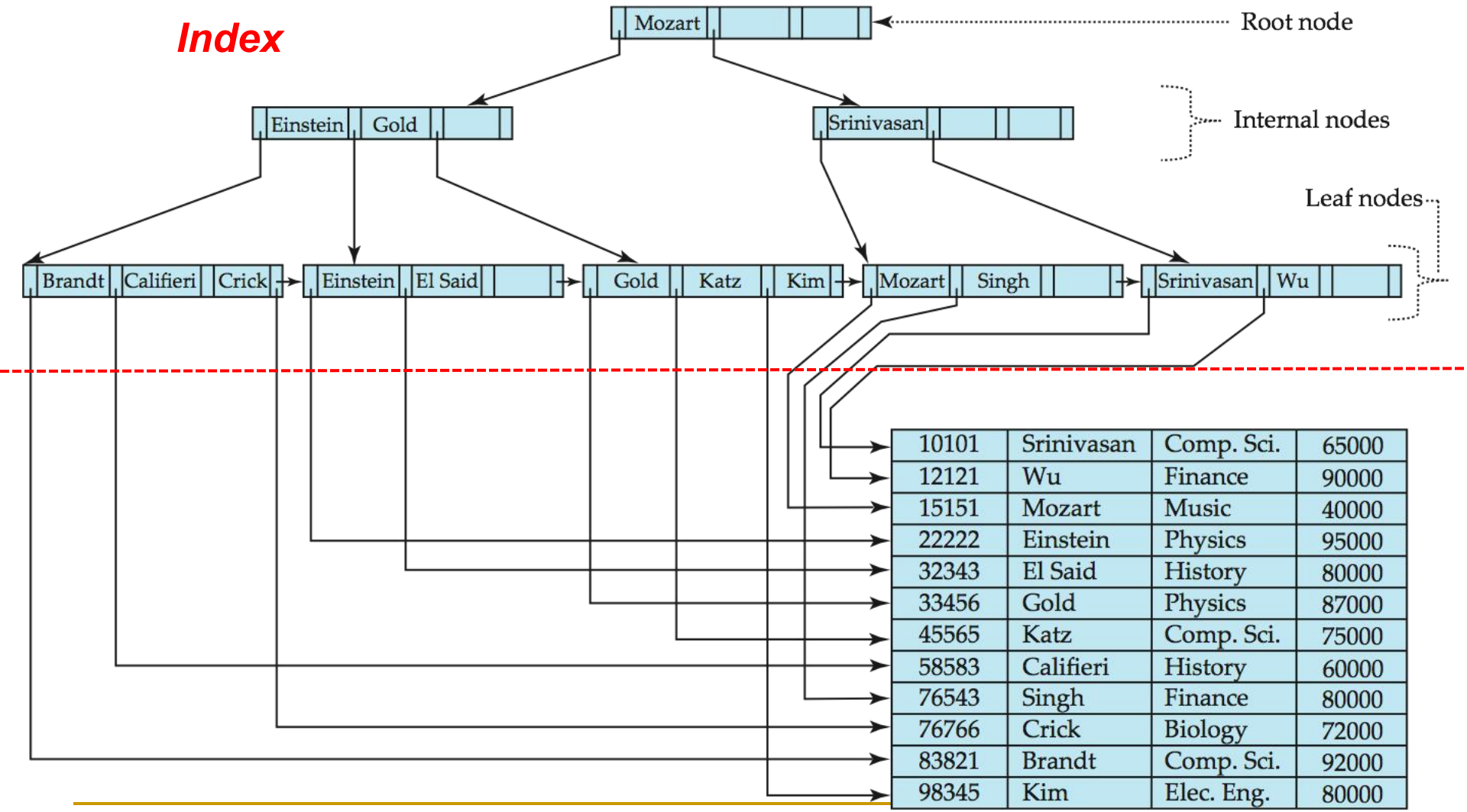
- 一种树型的多级索引结构
- 树的层数与数据大小相关，通常为3层
- 所有结点格式相同：n-1个搜索码，n个指针
- 所有叶结点位于同一层
- 适用于主索引，也可用于辅助索引

11.3 B+-Tree Index Files

B+-tree 索引是使用最广泛的几种索引结构之一。

- 索引顺序文件的缺点
  - 随着文件的增大，索引查找性能和数据顺序扫描性能都会下降
  - 需要周期性的重组整个索引文件。
- B+-tree索引的优点
  - 对于插入和删除操作，可以通过局部更改重新组织自身结构
  - 无需重新组织整个文件即可保持性能
- (Minor) B+-tree索引的缺点:
  - 额外的插入和删除开销、空间开销

Example of B+-Tree



B^+-Tree Index Files (properties)

- **All paths from root to leaf are of the same length**
- **Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.**
- **A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values**
- **Special cases:**
  - **If the root is not a leaf, it has at least 2 children.**
  - **If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.**

1、B+-Tree Node Structure

- **Typical node**

  - **K_i are the search-key values**
  - **P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).**

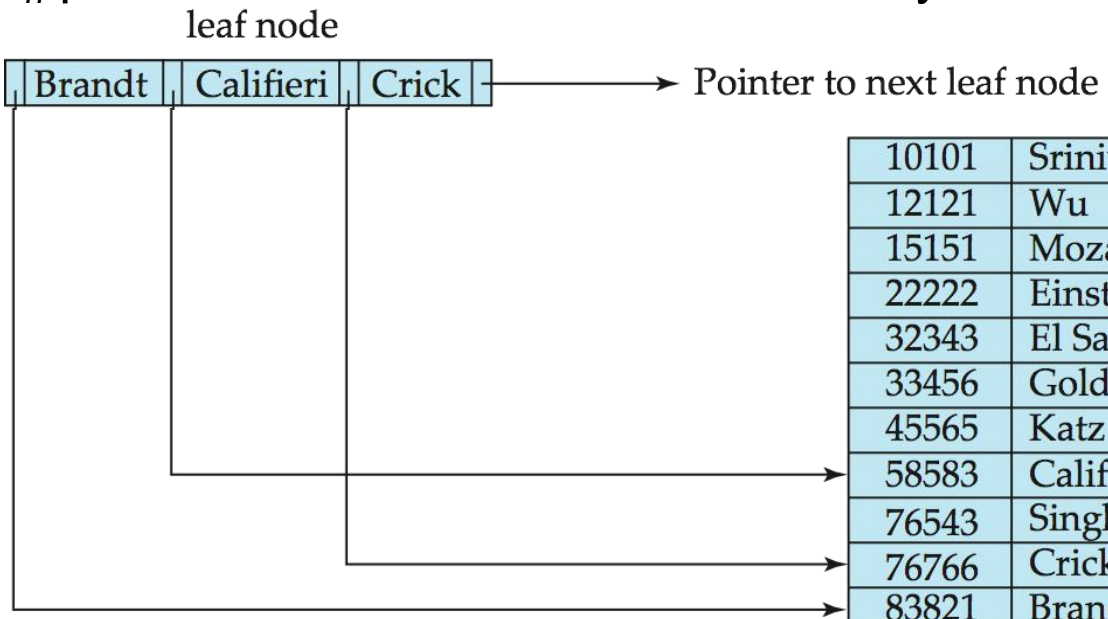
- **The search-keys in a node are ordered**

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees(Properties)

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order

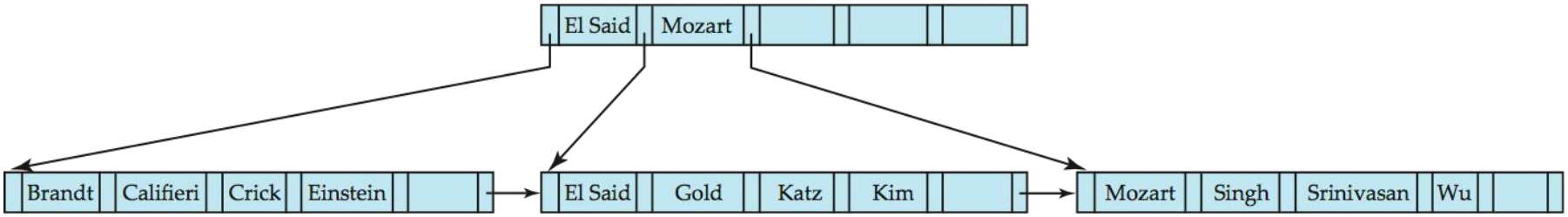


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Example of B+-tree



B+-tree for *instructor* file (n = 6)

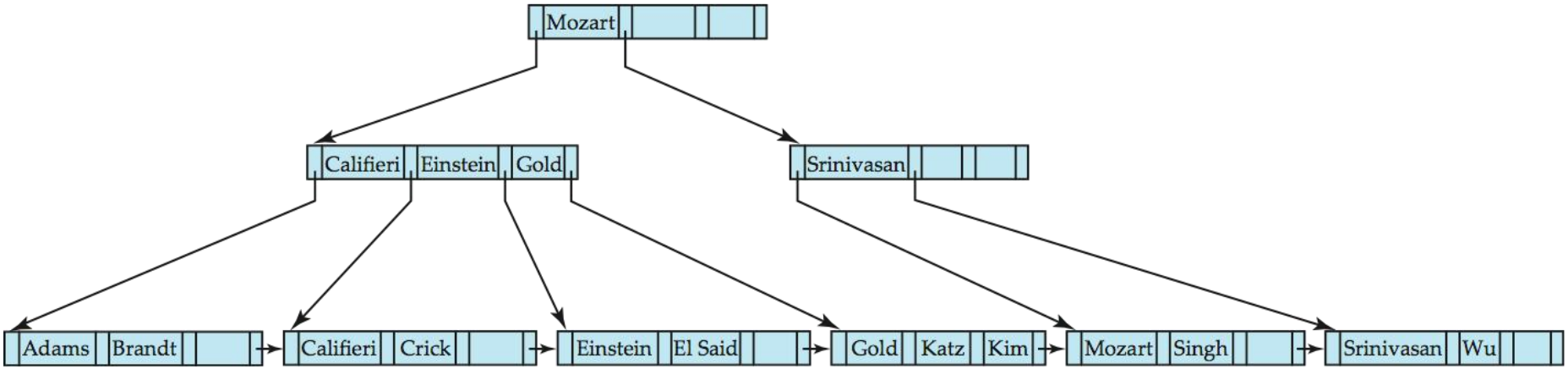
- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with n = 6).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with n =6).
- Root must have at least 2 children.

Observations about B+-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
  - Level below root has at least $2* \lceil n/2 \rceil$ values
  - Next level has at least $2* \lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

2、Queries on B+-Trees

- Book Fig. 11-11



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no **more than** $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

B+树的效率

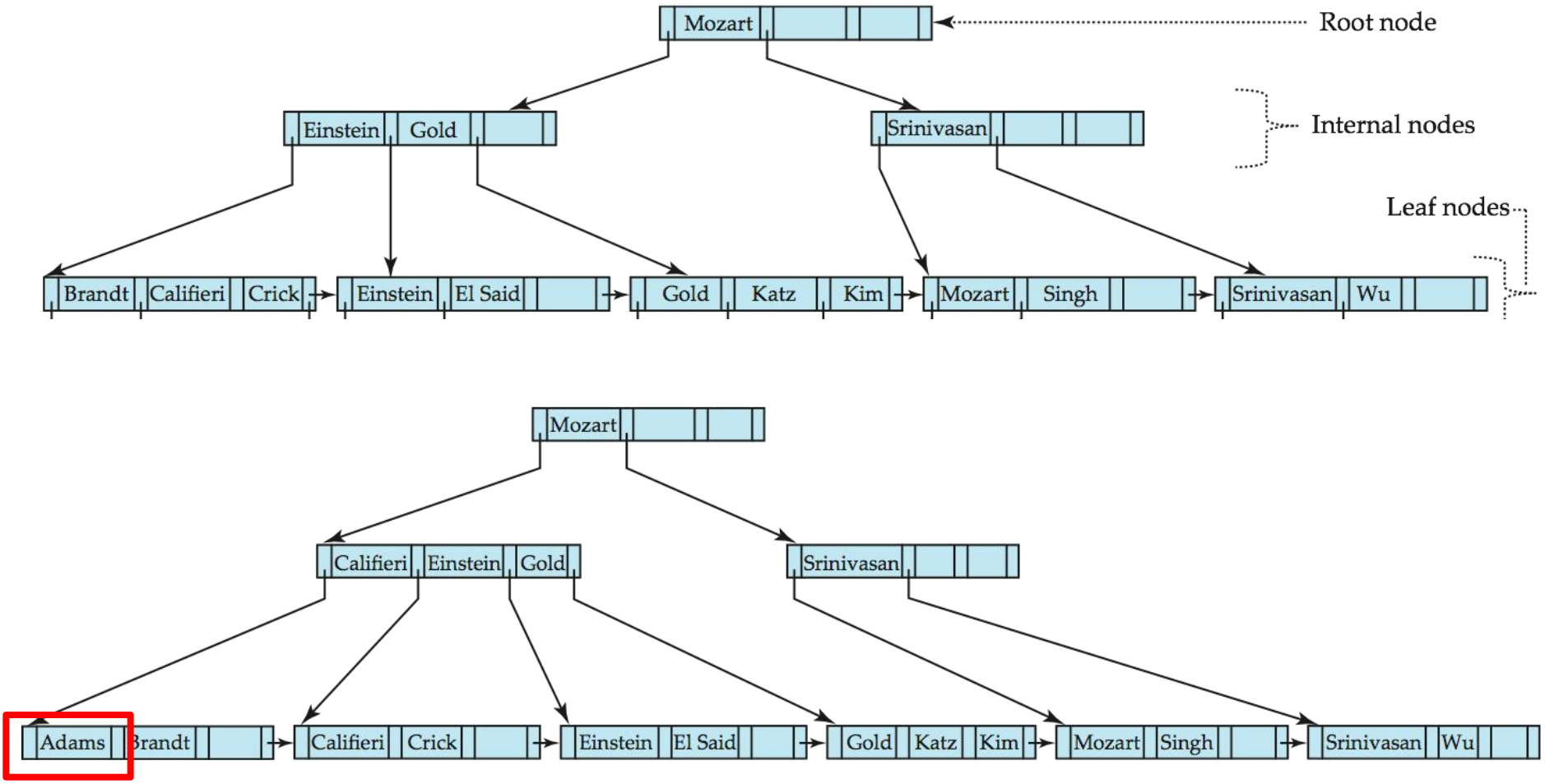
- 设块大小8KB，搜索码2B（smallint），指针2B，则一个块可放2048个索引项

层数	索引大小（块数/大小）	索引记录空间
1	1/8KB	2047
2	(1 + 2048)/约16M	约419万(2^{22})
3	(1+2^{11}+2^{22})/约32G	约85亿(2^{33})

3、Updates on B^+-Trees: Insertion

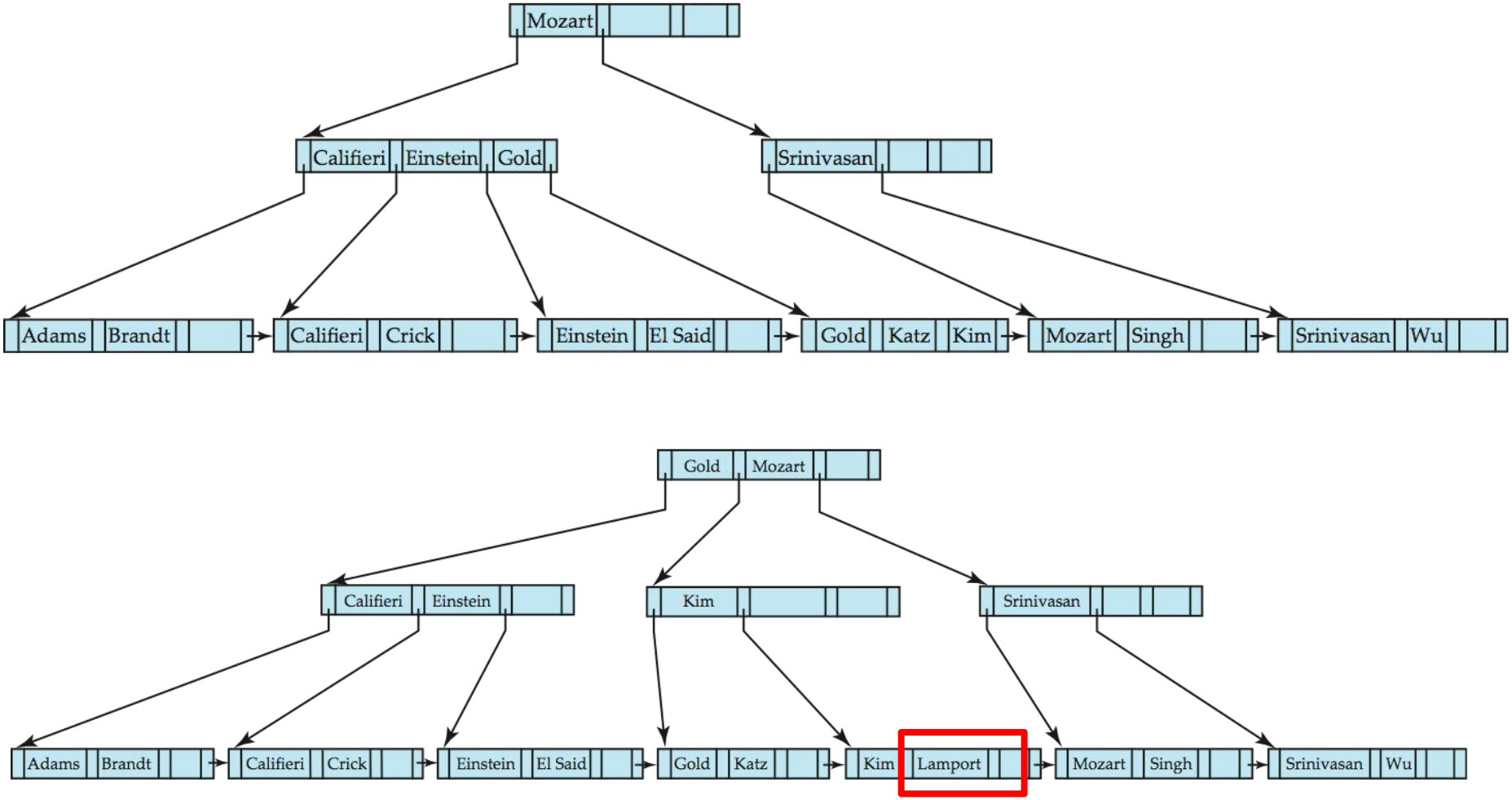
- Book Fig. 11-15

B+-Tree Insertion (n=4)



B+-Tree before and after insertion of “Adams”

B+-Tree Insertion

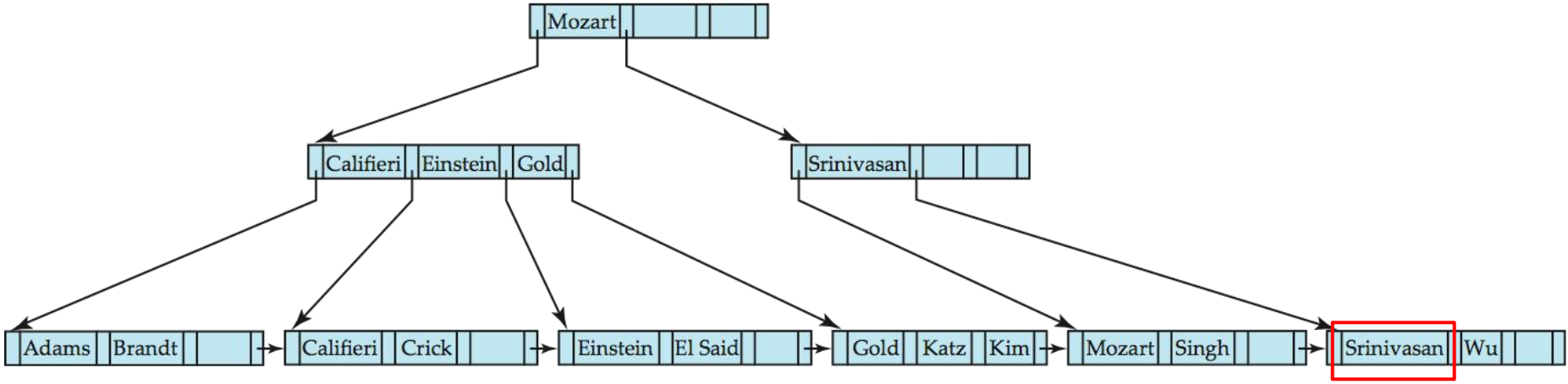


B+-Tree before and after insertion of “Lamport”

4、Updates on B^+-Trees: Deletion

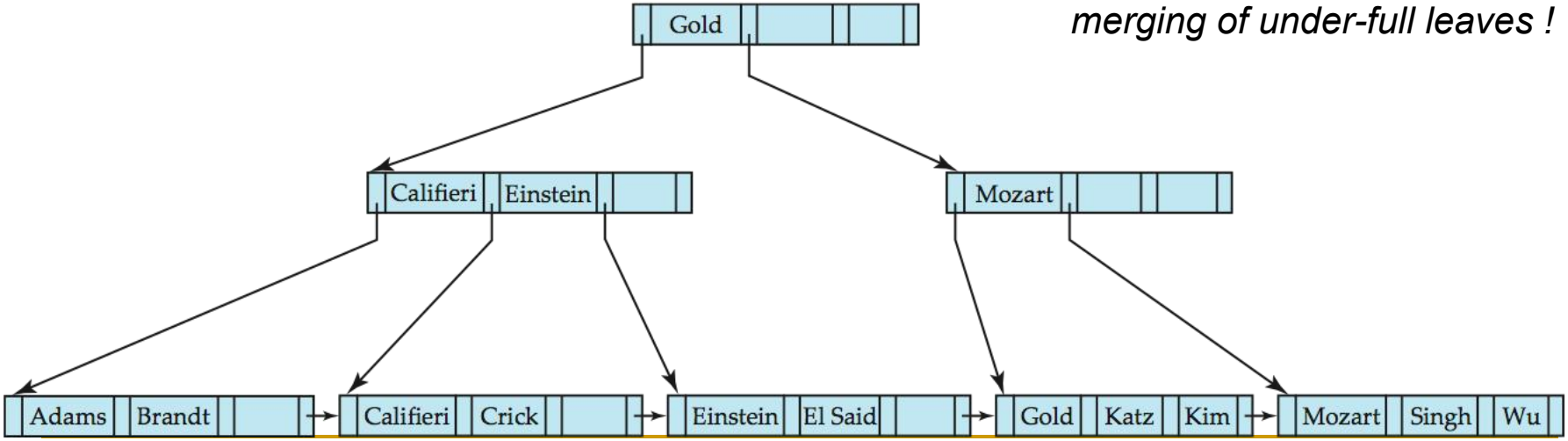
- Book Chapter 11.3.3.2

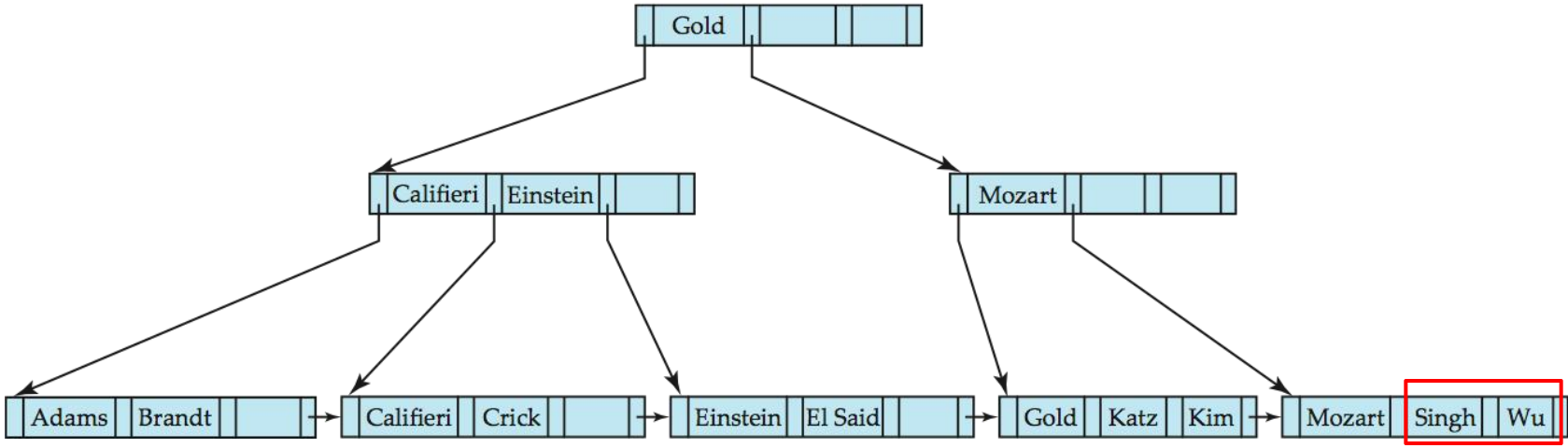
Examples of B+-Tree Deletion



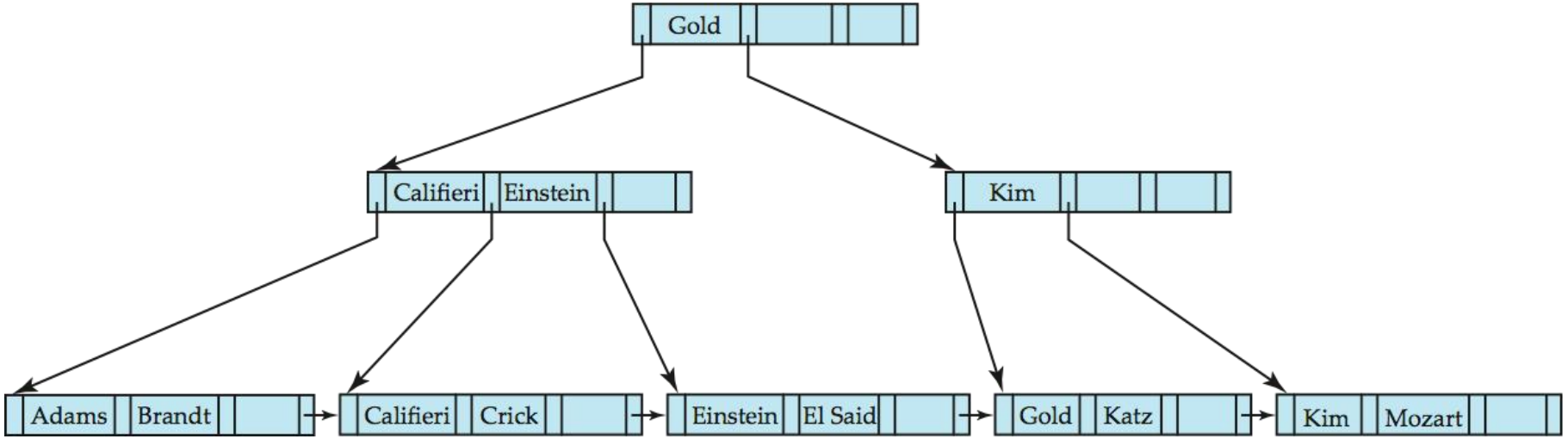
Before and after deleting "Srinivasan"

Deleting "Srinivasan" causes merging of under-full leaves !

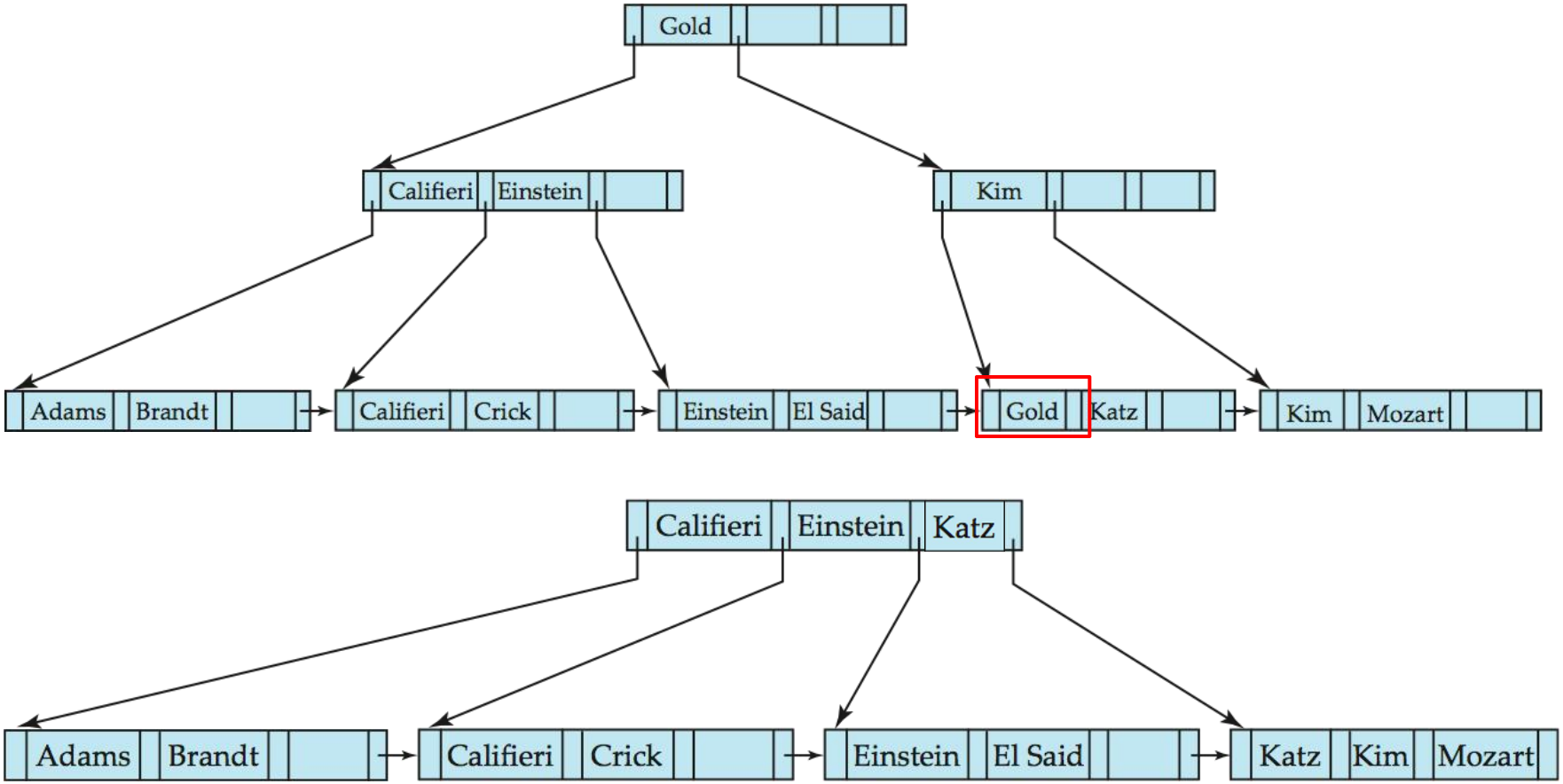




Deletion of "**Singh**" and "**Wu**" from result of previous example



- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result



Before and after deletion of "**Gold**" from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

11.6 静态散列(Static Hashing)

- 散列函数(Hash Functions)：均匀、随机
  - h：搜索码(散列键) → [0…B – 1]
  - 桶(Buckets), numbered 0,1,…, B-1
- 散列索引方法及I/O代价估算
  - 给定一个搜索码K，对应的记录必定位于桶h(K)中
  - 若一个桶中仅一块，则 I/O次数 = 1
  - 否则由参数B决定，平均 = 总块数/B

静态散列文件的例子

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key.

1、散列表查找

- 查找
  - 对于给定的散列码值K，计算h(K)
  - 根据h(K)定位桶
  - 查找桶中的块

h(Biology) = 5

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

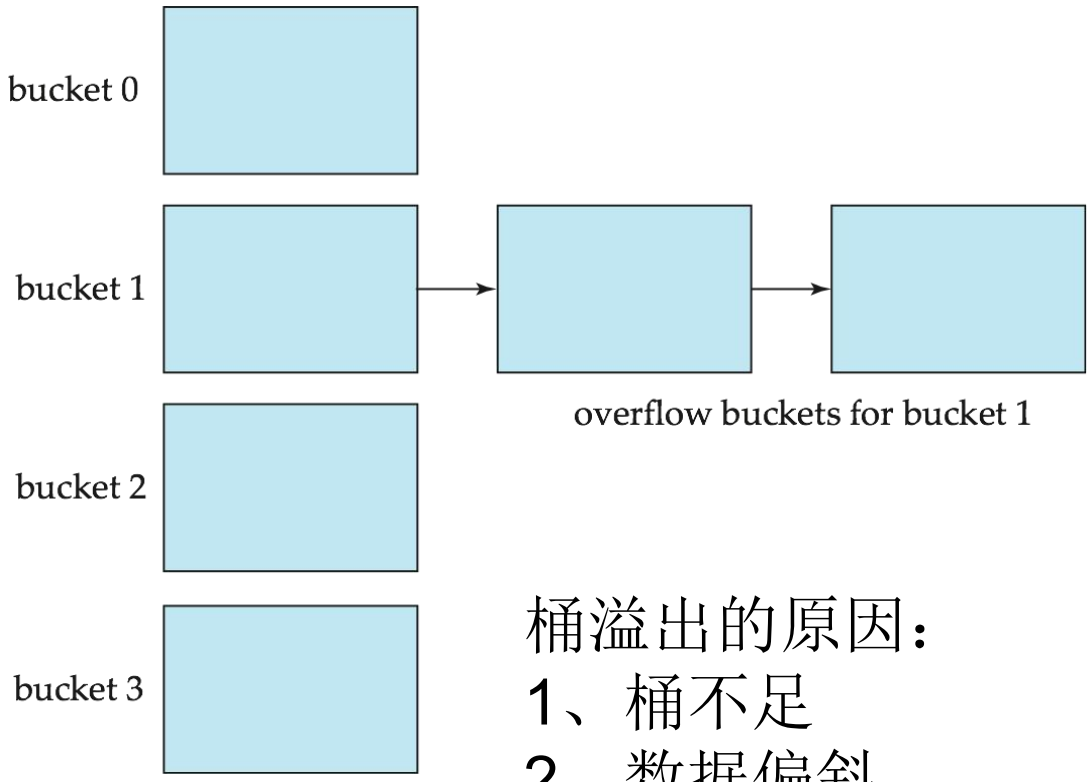
bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

2、散列表插入

- 计算插入记录的h(K)，定位桶
- 若桶中有空间，则插入
- 否则
  - 创建一个溢出桶并将记录置于溢出桶中

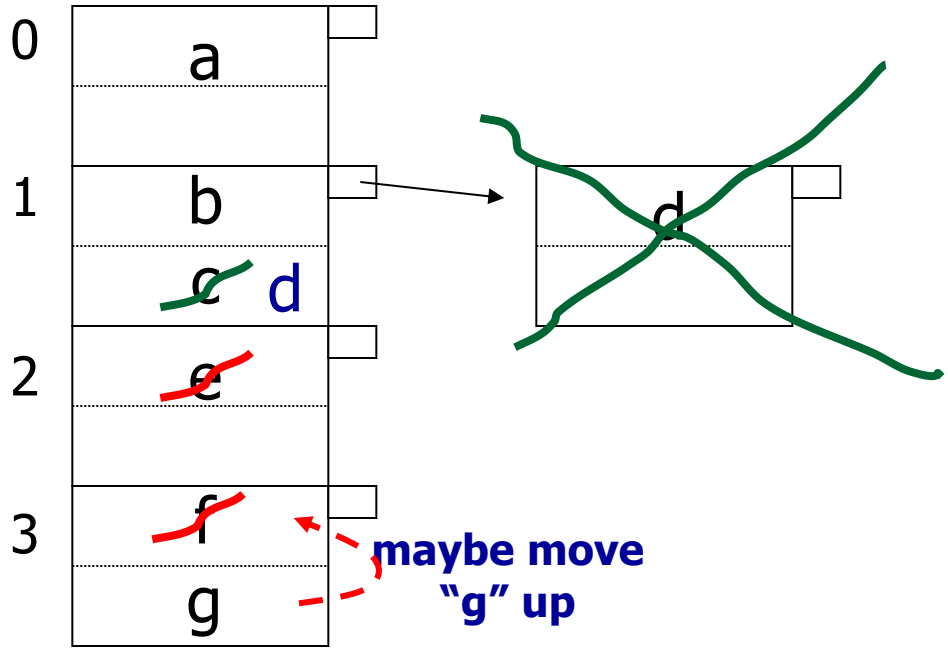


桶溢出的原因：
1、桶不足
2、数据偏斜

3、散列表删除

- 根据给定码值K计算h(K)，定位桶和记录
- 删除
  - 回收溢出块？

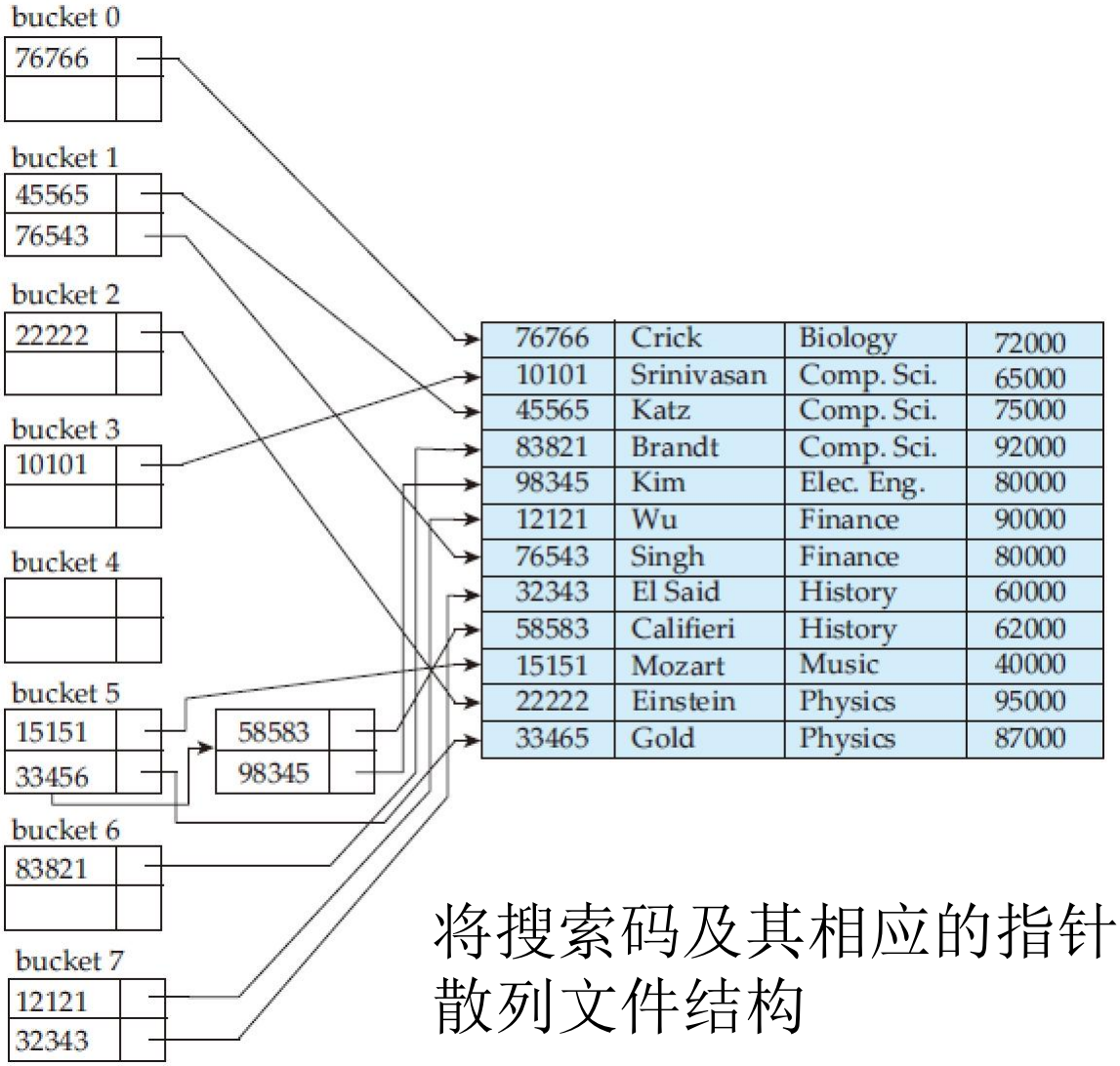
Delete:
e
f
c



4、散列表空间利用率问题

- 空间利用率
  - 实际码值数 / 所有桶可放置的码值数
  - <50%：空间浪费
  - >80%：溢出问题
  - 50%到80%之间（GOOD！）

散列索引



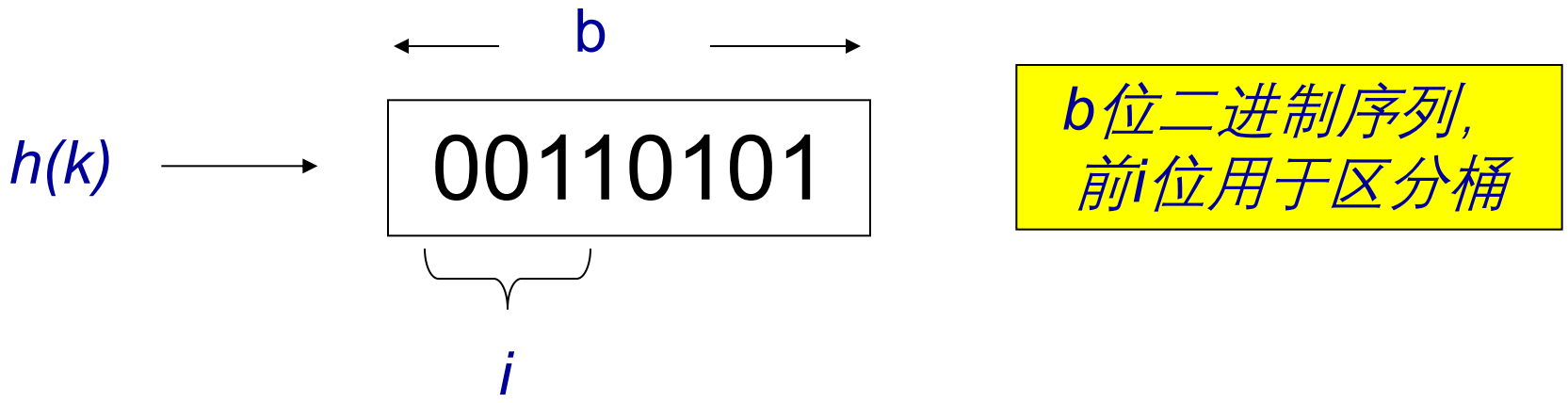
将搜索码及其相应的指针组织成散列文件结构

5、文件增长

- 数据文件的增长使桶的溢出块数增多，增加I/O
  - 采用动态散列表解决
    - 可扩展散列表（Extensible Hash Tables）
      - 成倍增加桶数目
    - 线性散列表（Linear Hash Tables）(自行查阅)
      - 线性增加

11.7 动态散列

- 散列函数 h(k) 是一个 b 位(足够大) 二进制序列，前 i 位表示桶的数目。
- i 的值随数据文件的增长而增大



Dynamic Hashing

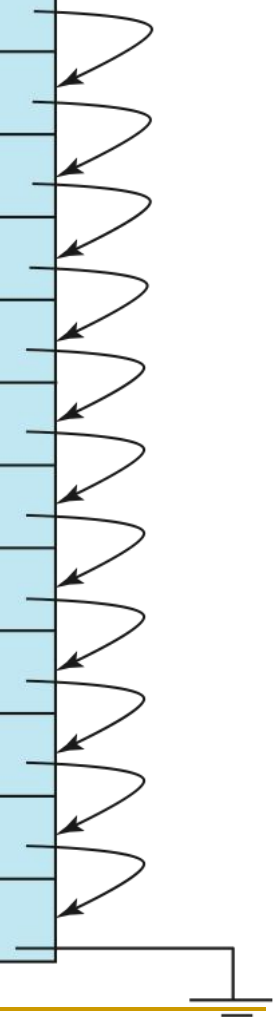
- **Represent the hash value of each record in binary.**

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example

The original table

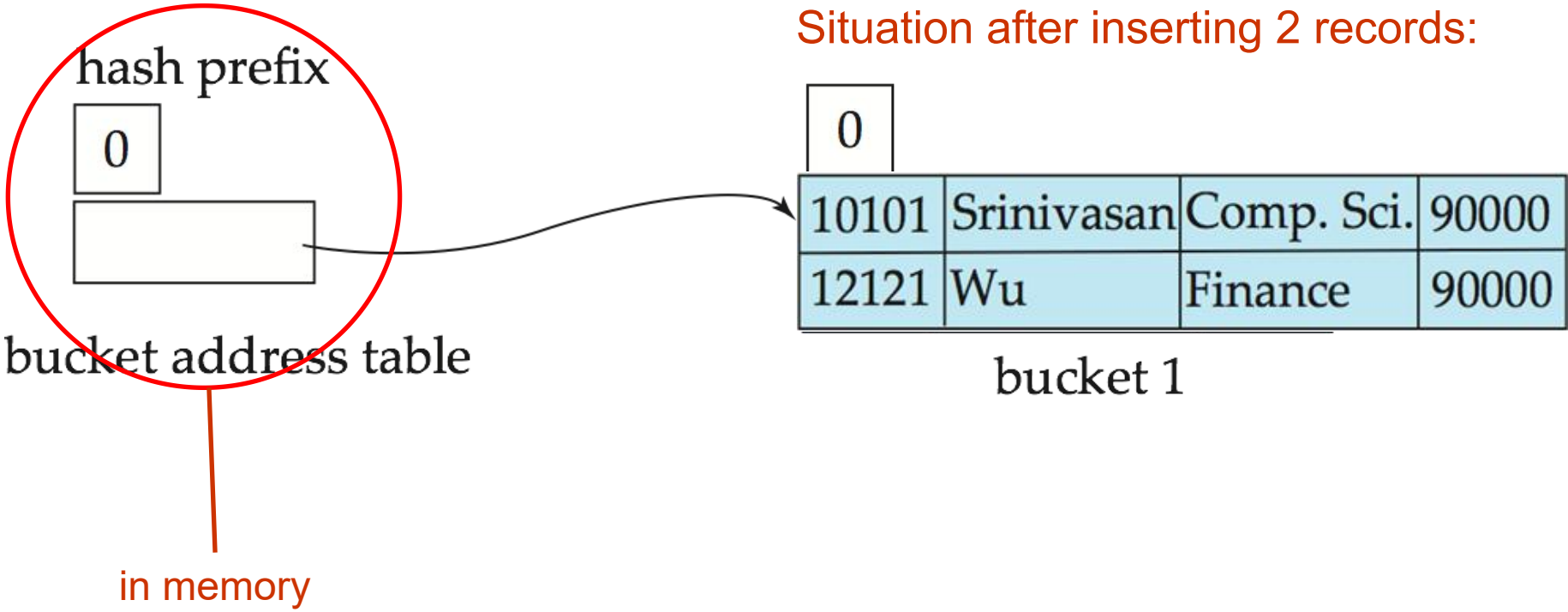
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	

Example (Cont.)

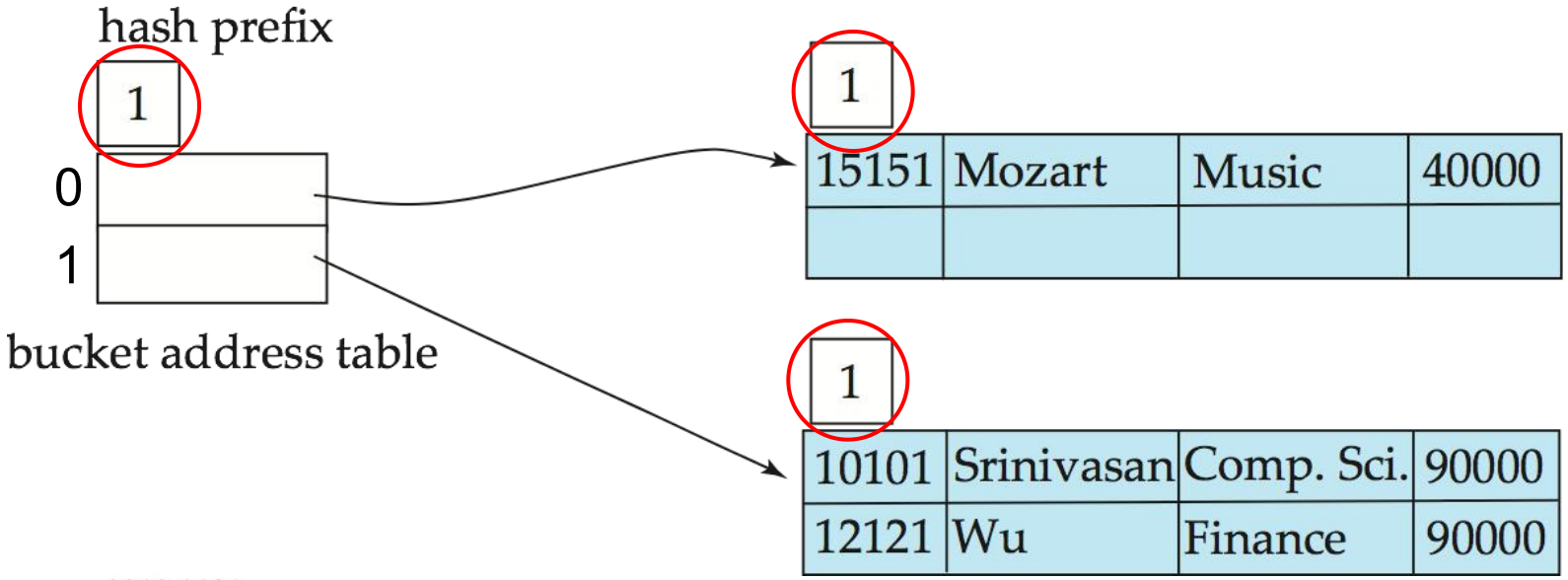
- Initial Hash structure; bucket size = 2



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	

Example (Cont.)

- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records

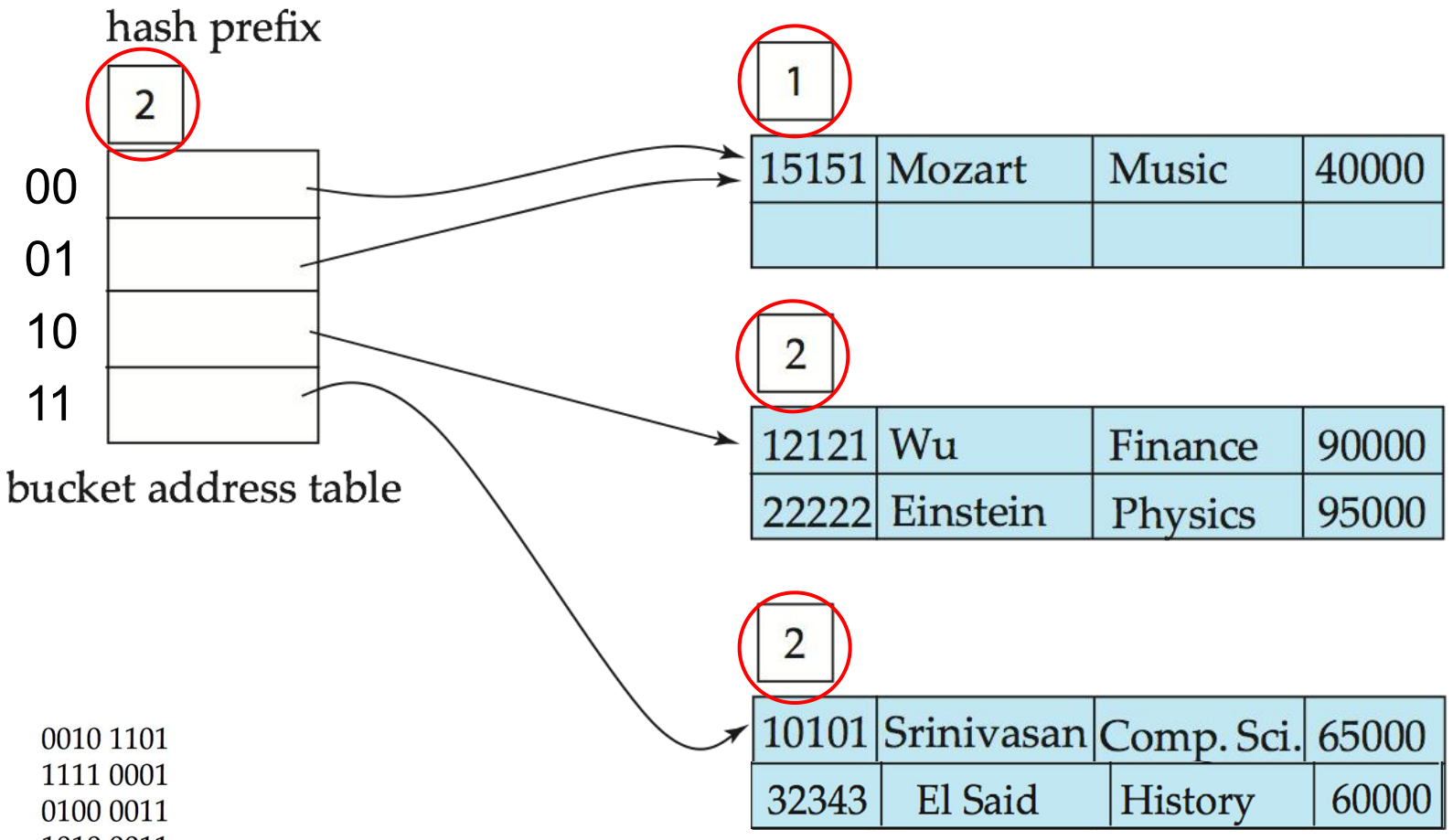


Biology	0010 1101
Comp. Sci.	1111 0001
Elec. Eng.	0100 0011
Finance	1010 0011
History	1100 0111
Music	0011 0101
Physics	1001 1000

Example (Cont.)

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	

- Hash structure after insertion of "Einstein" and "EL Said" records

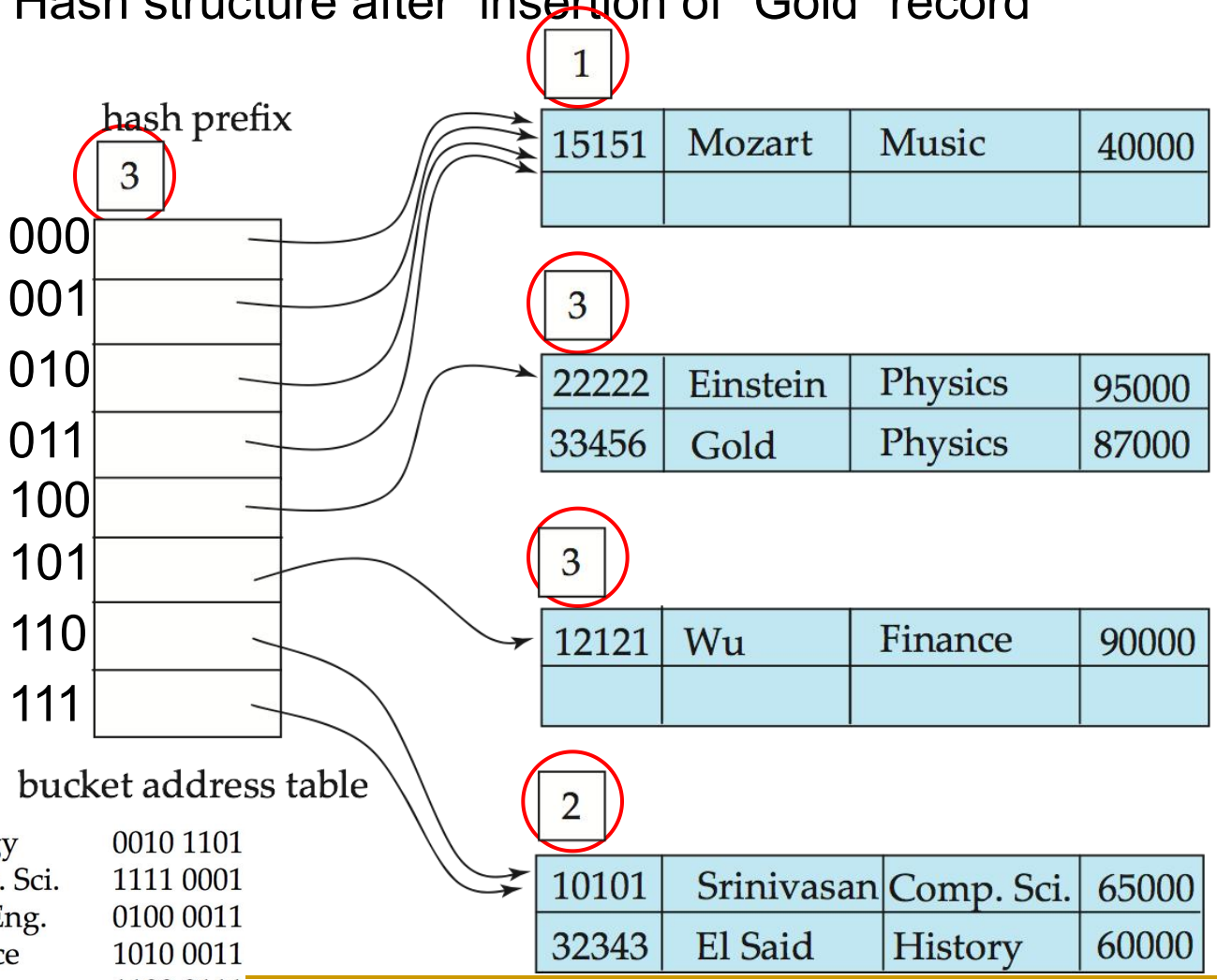


Biology	0010 1101
Comp. Sci.	1111 0001
Elec. Eng.	0100 0011
Finance	1010 0011
History	1100 0111
Music	0011 0101
Physics	1001 1000

Example (Cont.)

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	

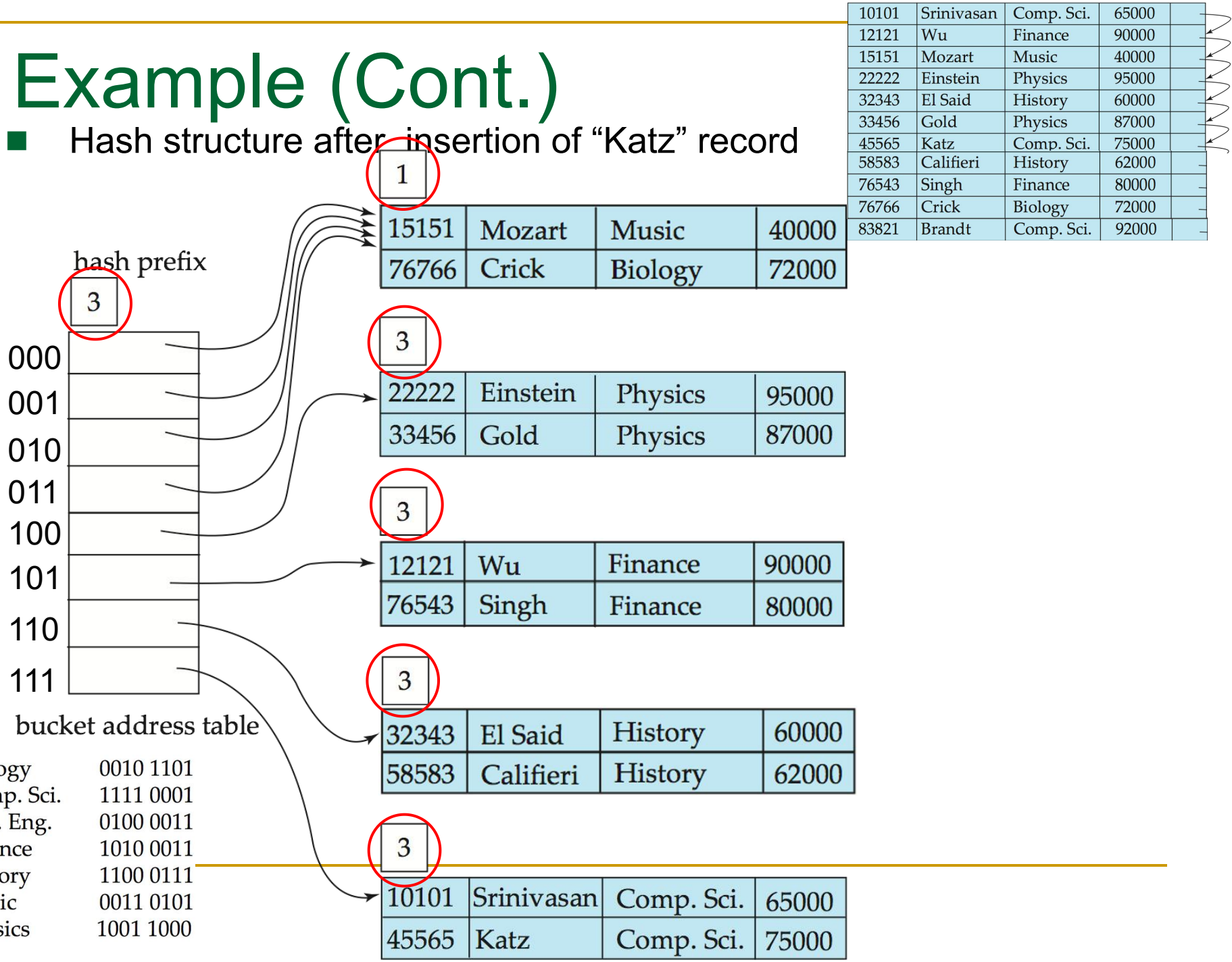
- Hash structure after insertion of "Gold" record



Biology	0010 1101
Comp. Sci.	1111 0001
Elec. Eng.	0100 0011
Finance	1010 0011
History	1100 0111
Music	0011 0101
Physics	1001 1000

Example (Cont.)

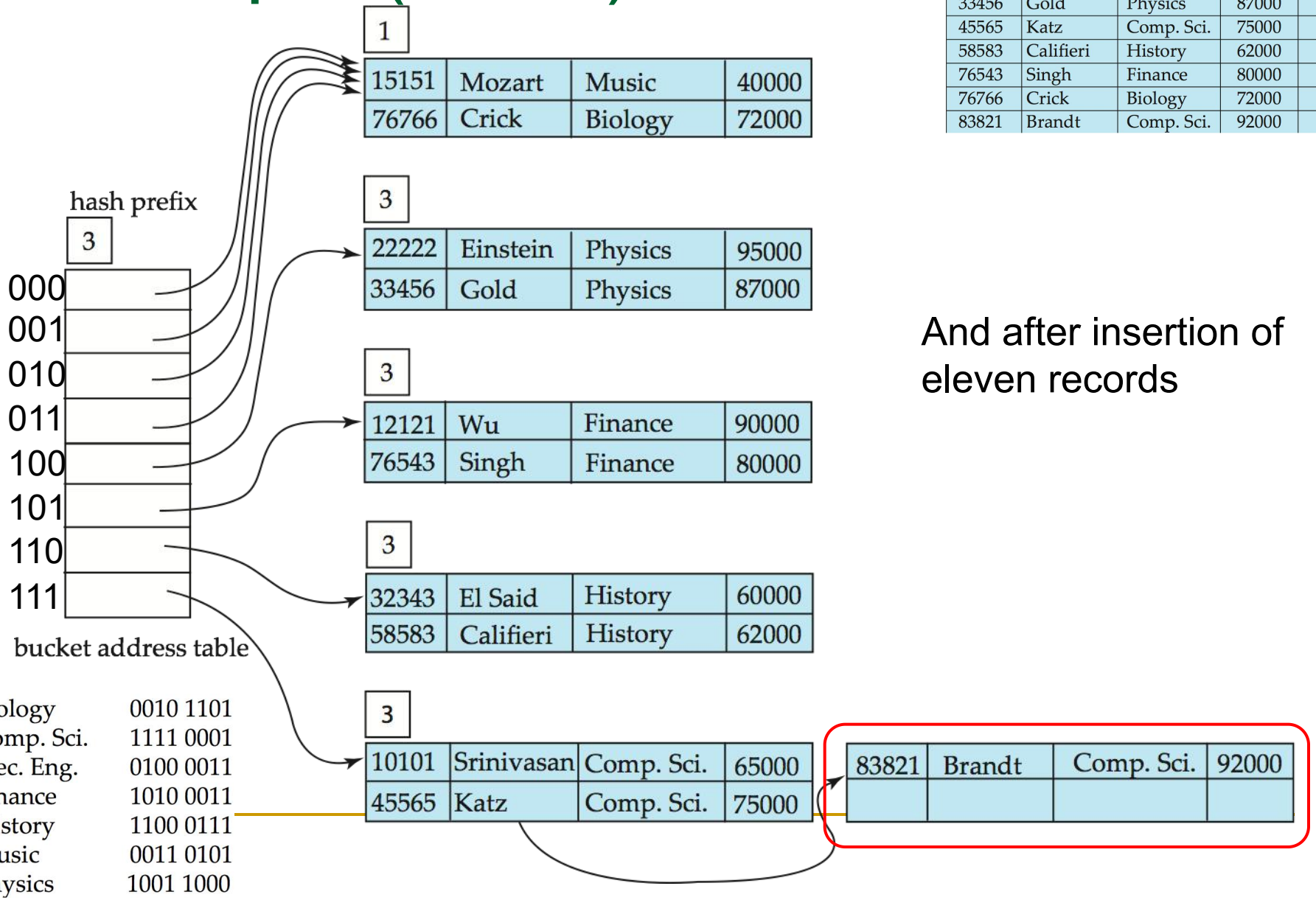
- Hash structure after insertion of "Katz" record



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	

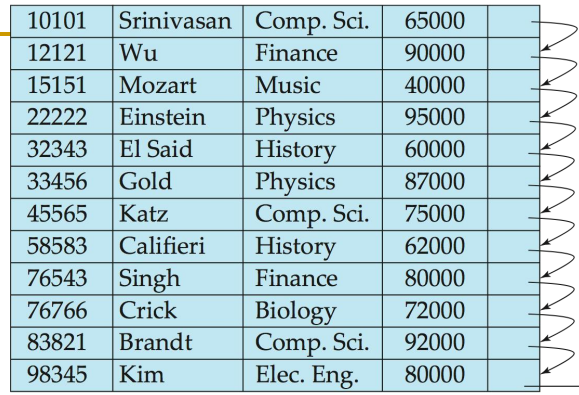
Example (Cont.)

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	

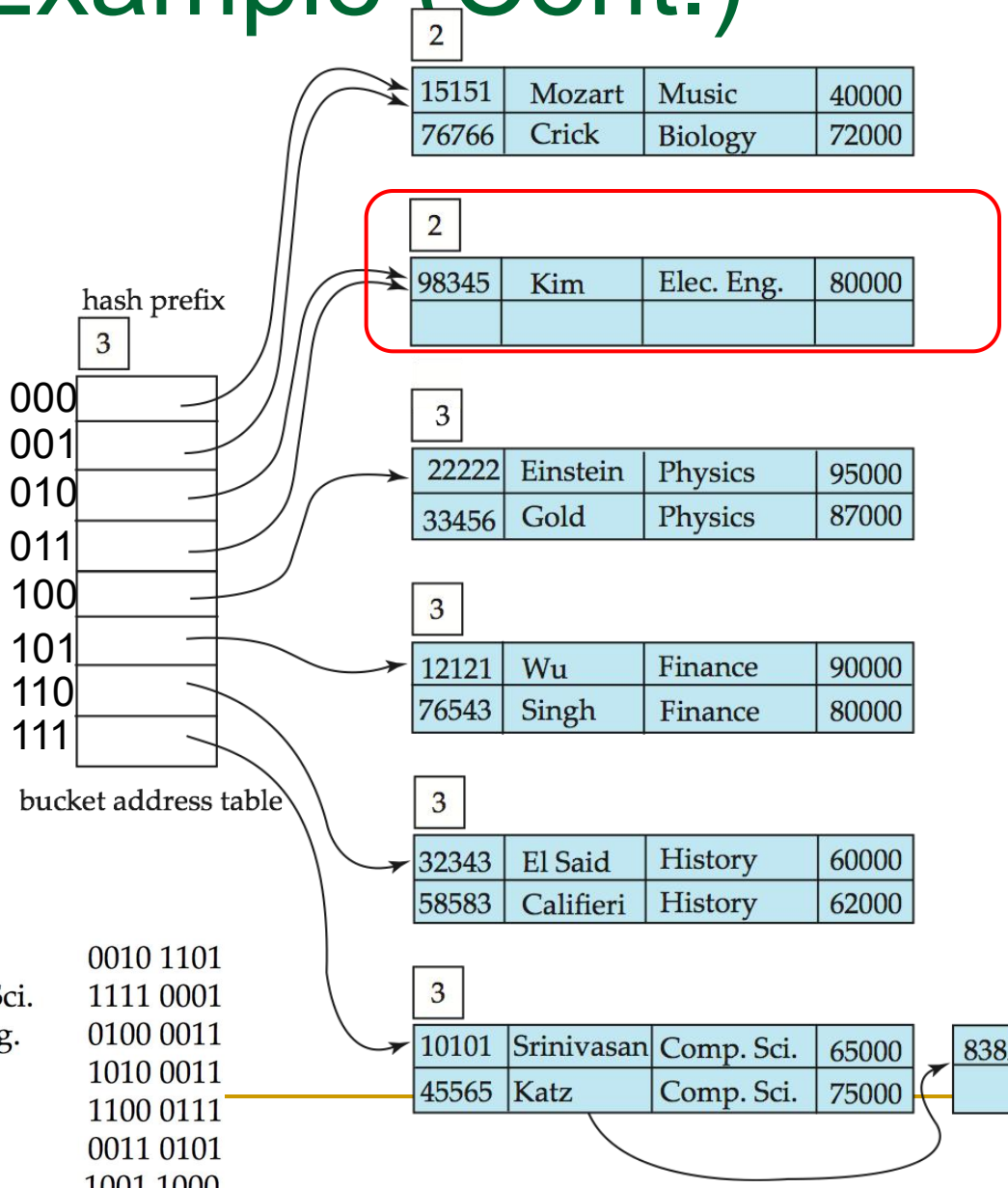


And after insertion of eleven records

Example (Cont.)



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



And after insertion of "Kim" record in previous hash structure

Biology	0010 1101
Comp. Sci.	1111 0001
Elec. Eng.	0100 0011
Finance	1010 0011
History	1100 0111
Music	0011 0101
Physics	1001 1000

11.9 位图索引

- 位图索引是一种使用位图的特殊数据库索引。
- 主要针对大量相同值的列而创建(例如：性别)
- 索引块的一个索引行中存储搜索码值和起止Rowid,以及这些搜索码值的位置编码
- 位置编码中的每一位表示搜索码值对应的数据行的有无.一个块可能指向的是几十甚至成百上千行数据的位置

11.9 位图索引

- 当根据搜索码值查询时,可以根据起始Rowid和位图状态,快速定位数据
- 当根据搜索码值做and,or或 in(x,y,..)查询时,直接用索引的位图进行或运算,快速得出结果行数据.
- 当select count() 时,可以直接访问索引就快速得出统计数据.

位图索引

Select * From R Where item="H" and city="V";

Select count(*) From R Where city="V";

基本表

RID	item	city
R1	H	V
R2	C	V
R3	P	V
R4	S	V
R5	H	T
R6	C	T
R7	P	T
R8	S	T

item位图索引表

RID	H	C	P	S
R1	1	0	0	0
R2	0	1	0	0
R3	0	0	1	0
R4	0	0	0	1
R5	1	0	0	0
R6	0	1	0	0
R7	0	0	1	0
R8	0	0	0	1

码值H的位图

city位图索引表

RID	V	T
R1	1	0
R2	1	0
R3	1	0
R4	1	0
R5	0	1
R6	0	1
R7	0	1
R8	0	1

位图索引

Select * From R Where item="H" and city="V";

Select count(*) From R Where city="V";

基本表

RID	item	city
R1	H	V
R2	C	V
R3	P	V
R4	S	V
R5	H	T
R6	C	T
R7	P	T
R8	S	T

item位图索引表

RID	H	C	P	S
R1	1	0	0	0
R2	0	1	0	0
R3	0	0	1	0
R4	0	0	0	1
R5	1	0	0	0
R6	0	1	0	0
R7	0	0	1	0
R8	0	0	0	1

city位图索引表

RID	V	T
R1	1	0
R2	1	0
R3	1	0
R4	1	0
R5	0	1
R6	0	1
R7	0	1
R8	0	1

11.10实际系统中的索引

- SQL Server
  - B+-树
  - 聚簇索引是稀疏的
  - 更新时维护索引
- DB2
  - B+-树，R-树
  - 聚簇索引是稠密的
  - 索引重组通过显式命令进行
- Oracle
  - B+-树，哈希，位图，R-树
- TimesTen (主存数据库)
  - T-树

Assignment

- **11.3**
- **11.4**
- **11.6**
- **11.7**