Hashing Techniques For File Organization

Introduction to Hashing

- Each data-item with hash key value K is stored in location i, where i=h(K), and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a address that is already full
  - An overflow file is kept for storing such records.

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*.
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example File organization with Hashing

Hash file organization of *instructor* file, using ***dept_name*** as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example File organization with Hashing

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 (see previous slide for details).

Mapping to Secondary Memory

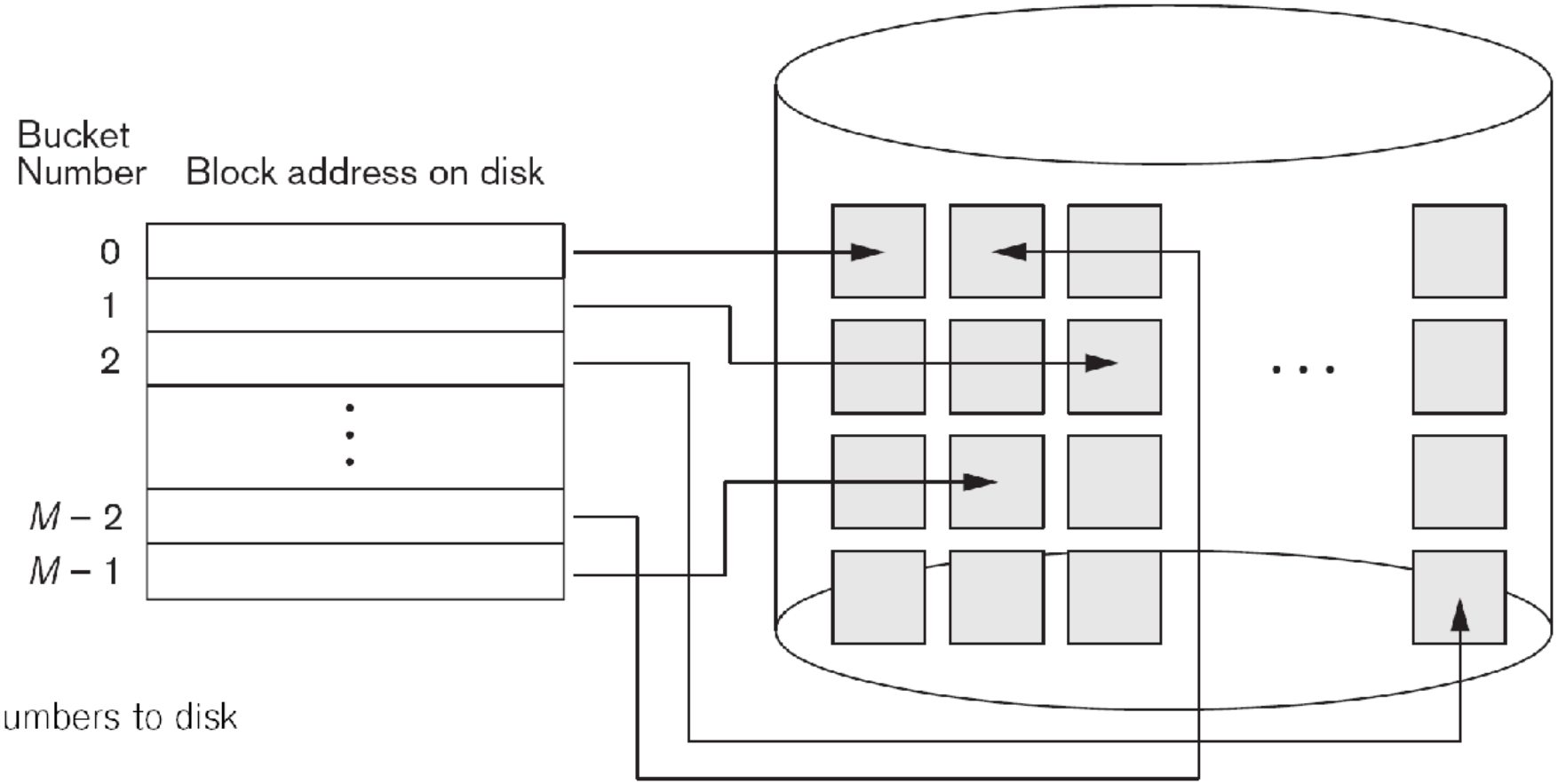


Figure 17.9
Matching bucket numbers to disk block addresses.

Desirable properties of a Hash Function

- Worst hash function maps all search-key values to the same bucket;
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.

Handling Collisions Hashing

- **Bucket overflow can occur because of**
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

Handling Collisions Hashing

- There are numerous methods for collision resolution:
  - **Open addressing**: Proceeding from the occupied position check the subsequent positions in order until an unused position is found.
  - **Chaining**: various overflow locations are kept, usually by extending the array with a number of overflow positions.

Which of these are suitable for Databases?

Handling Collisions in Hashing

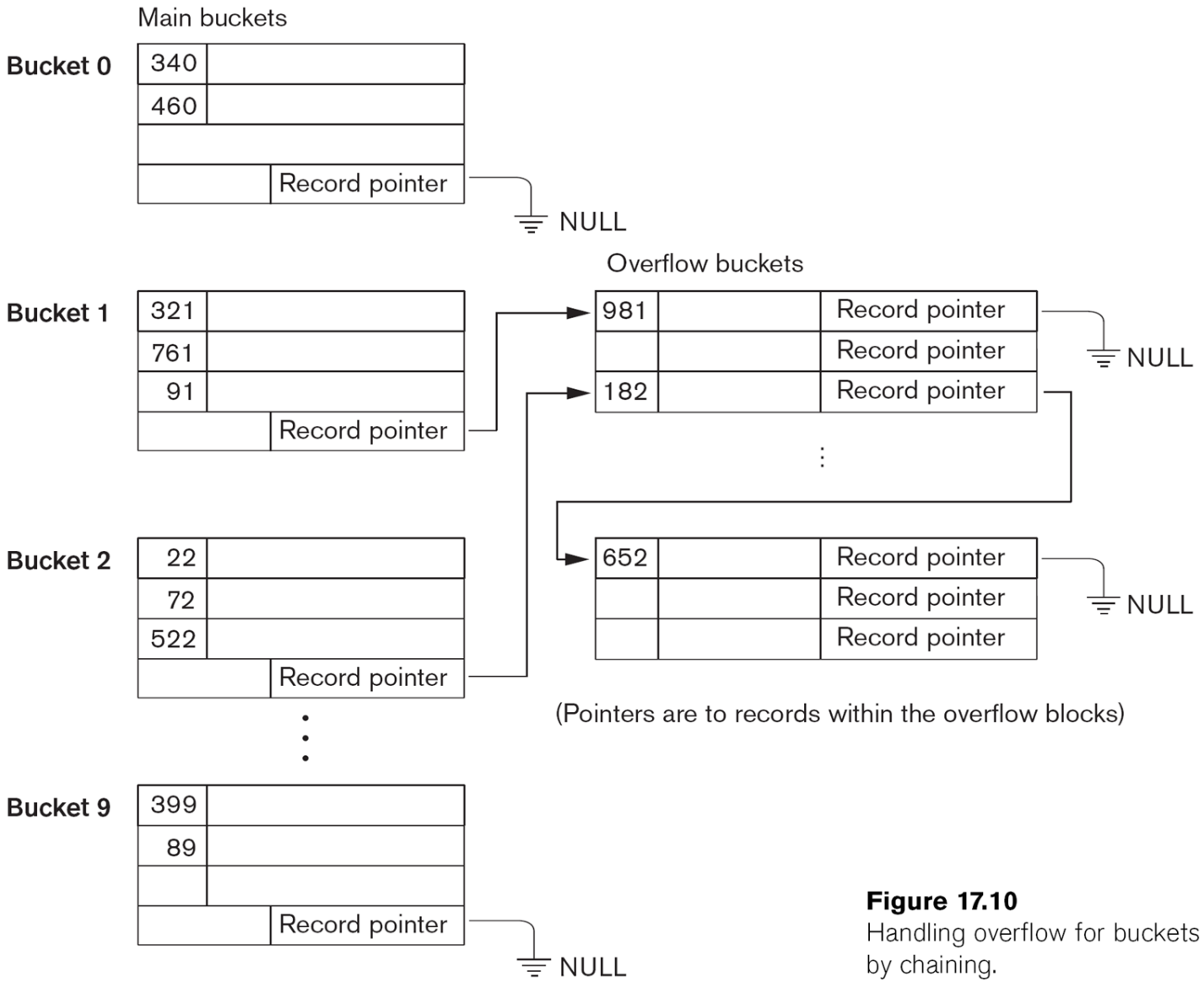


Figure 17.10
Handling overflow for buckets by chaining.

Lets Evaluate Static Hashing

Think in following terms:

- **Time required for search and insert.**
- **Space utilization?**

Lets Evaluate Static Hashing

Think in following terms:

- **Time required for search and insert.**
- **Space utilization?**

What if Database grows or shrinks with time ?

Lets Evaluate Static Hashing

- In **static hashing**, function *h* maps search-key values to a **fixed set of *B* of bucket addresses**.
  - **Databases grow or shrink with time**.
  - If **initial** number of **buckets** is **too small**, and **file grows**, performance will degrade due to **too much overflows**.

Lets Evaluate Static Hashing

- In **static hashing**, function *h* maps search-key values to a **fixed set of *B* of bucket addresses**.
  - **Databases grow or shrink with time**.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for **anticipated growth**, a significant amount of **space will be wasted initially** (buckets will be under full).
  - If database shrinks, again space will be wasted.

Lets Evaluate Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially.
- **One solution:**
  - **Periodic re-organization with a new hash function**
  - **Its expensive, disrupts normal operations**

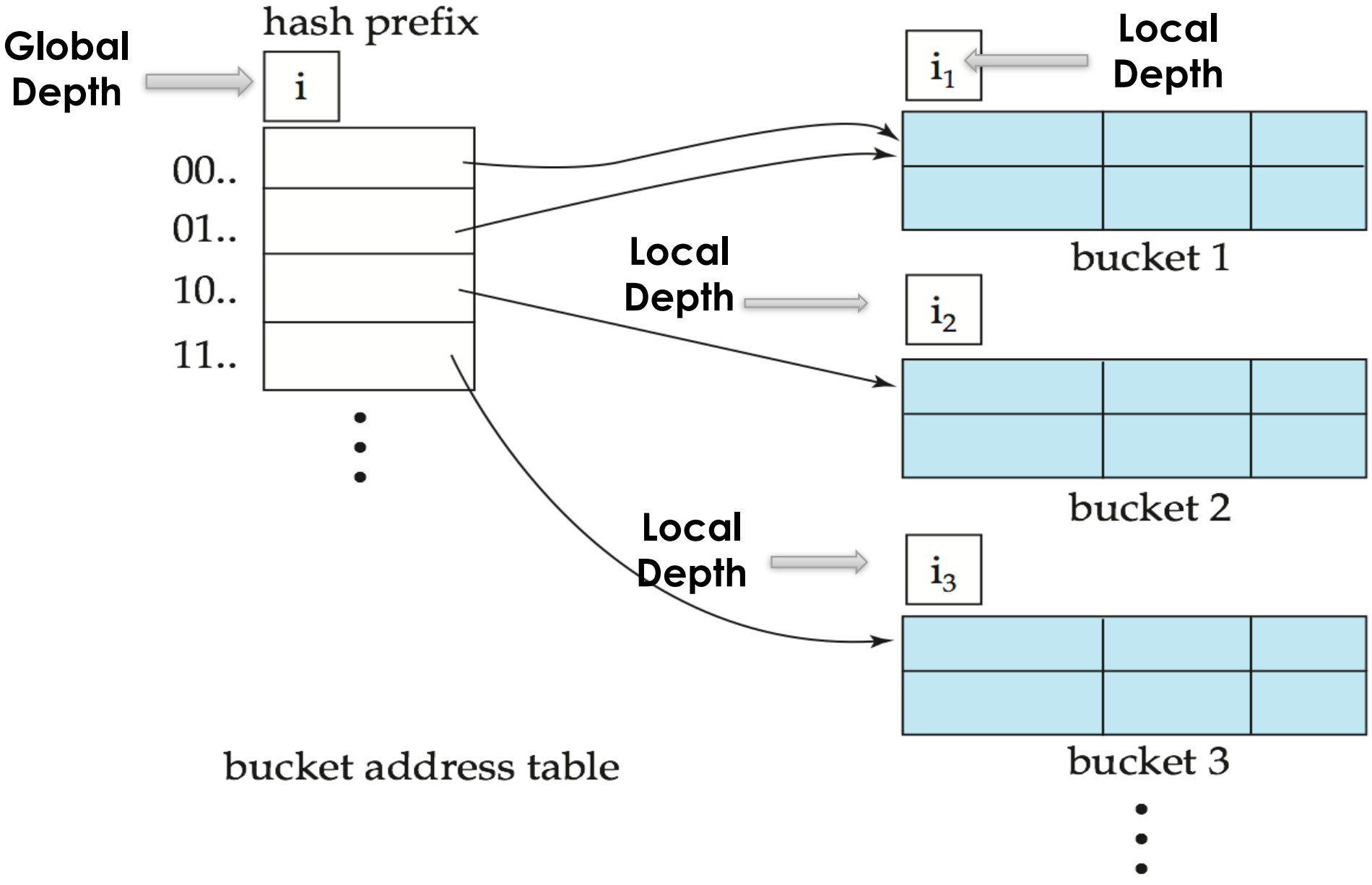
Hashing For Dynamic File Extension

- **Extendible hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with *b* = 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.

Hashing For Dynamic File Extension

- **Extendible hashing**
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 31$.
    - Bucket address table size = 2^i Initially $i = 0$
    - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket (why?)

Extendible Hashing



Extendible Hashing

- **Local Depth:** Each bucket *j* stores a value i_j *as its local depth*
  - All the entries that point to the same bucket have the same values on the first i_j bits.

Extendible Hashing

- **To locate the bucket containing search-key *K*:**
  1. Compute *h(K)* = *X*
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket

Extendible Hashing

- **To insert a record with search-key value K_{new}**
  - same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j* insert record in the bucket.
  - **Else the bucket must be split and insertion re-attempted**.

Splitting a bucket in Extendible Hash

- If **Global Depth > Local Depth** $i > i_j$ (more than one pointer to bucket *j*)
  - Allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to *j,* to point to z
  - Remove each record in bucket *j* and reinsert (in *j* or z)
  - Recompute new bucket for K_{new} and insert record in the bucket
  - Depending on implementation logic further splitting may or may not be done if the new bucket is still overflowing.

Splitting a bucket in Extendible Hash

- If **Global Depth = Local Depth** (only one pointer to bucket *j*)
  - If *i* reaches some limit *b (depends on implementation)*, or too many splits have happened in this insertion, create an overflow bucket
  - **Else (Idea for bucket address table expansion)**
    - increment *i* and double the size of the bucket address table.
    - replace each entry in the table by two entries pointing to the same bucket. Local depths remain same as original.
    - recompute new bucket address table entry for K_{new}
      Now $i > i_j$ (global dep > local dep) so use the first case of insert described previously on slide 68.

Illustrating an Extendible Hash: Dataset

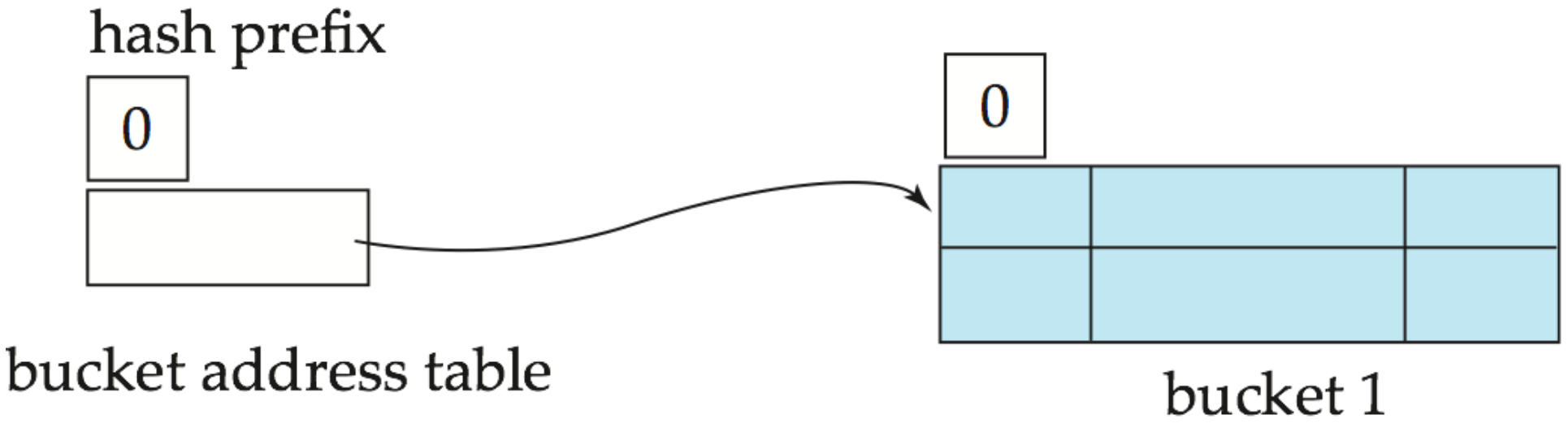
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000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

Illustrating an Extendible Hash

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

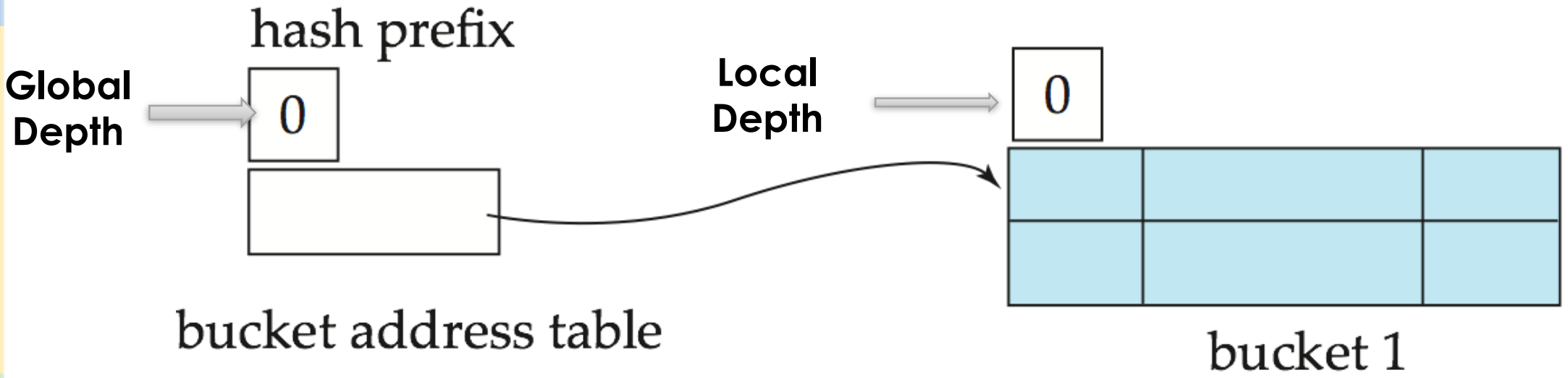
Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2



Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2



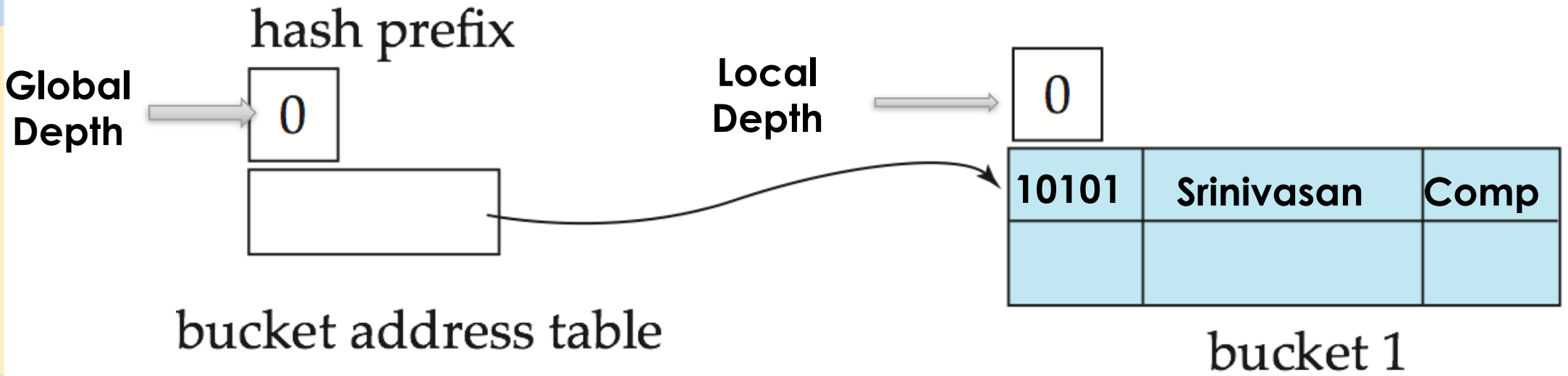
Insert this Record

10101	Srinivasan	Comp. Sci.	65000

Comp. Sci. 1111 0001 0010 0100 1001 0011 0110 1101

Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2



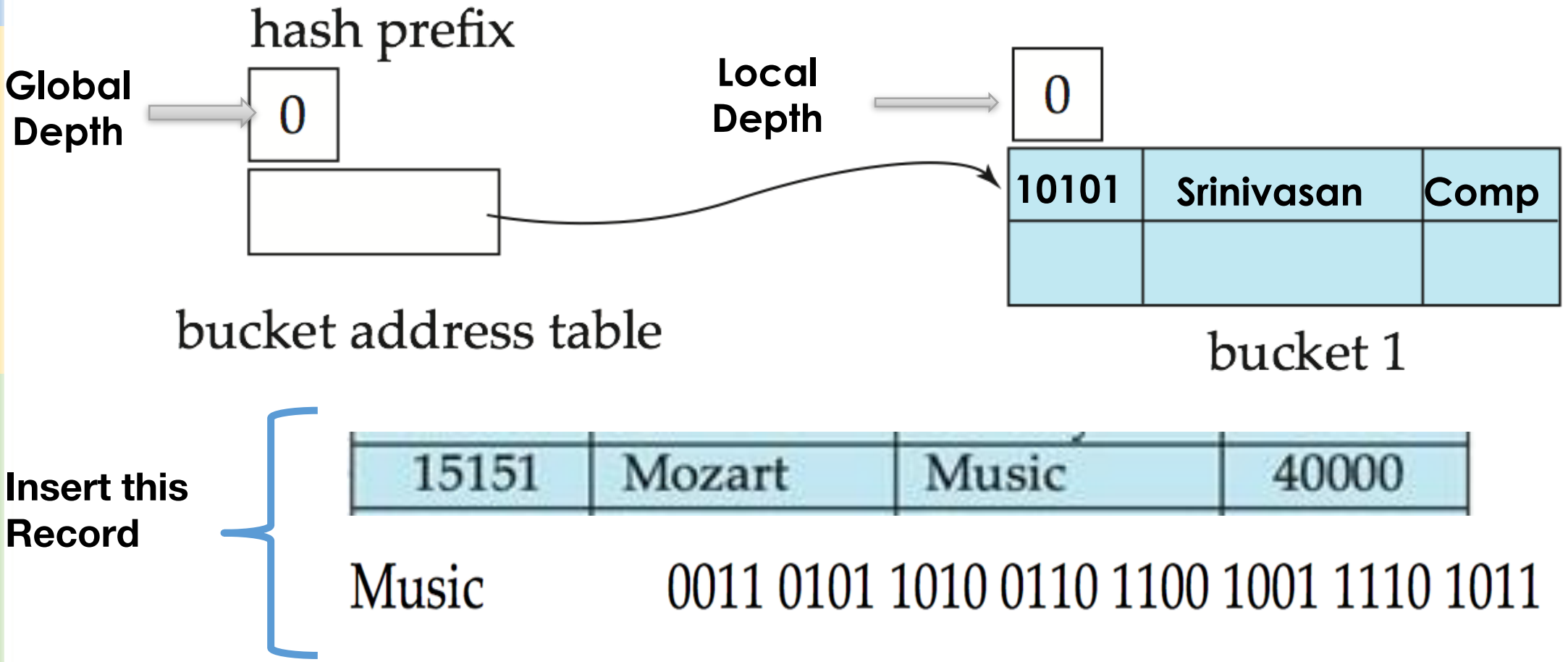
Insert this Record

10101	Srinivasan	Comp. Sci.	65000

Comp. Sci. 1111 0001 0010 0100 1001 0011 0110 1101

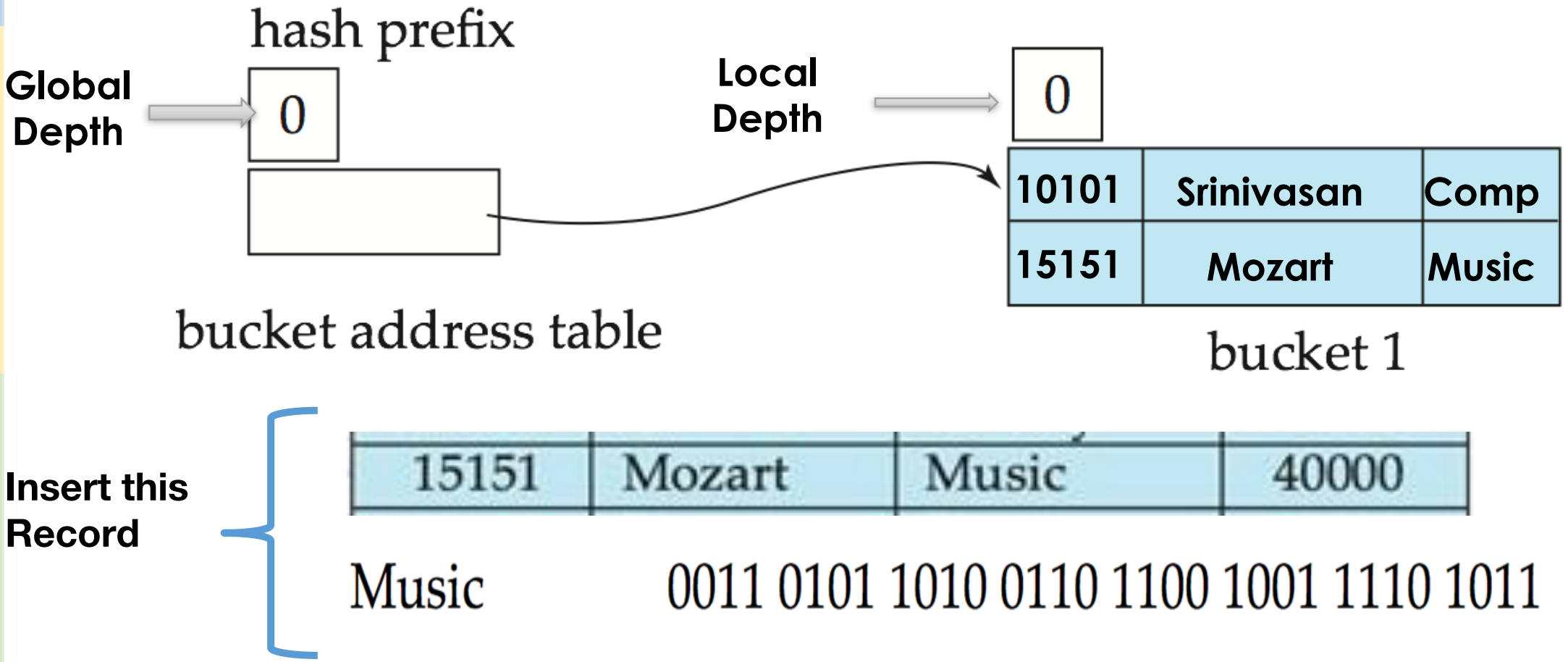
Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2



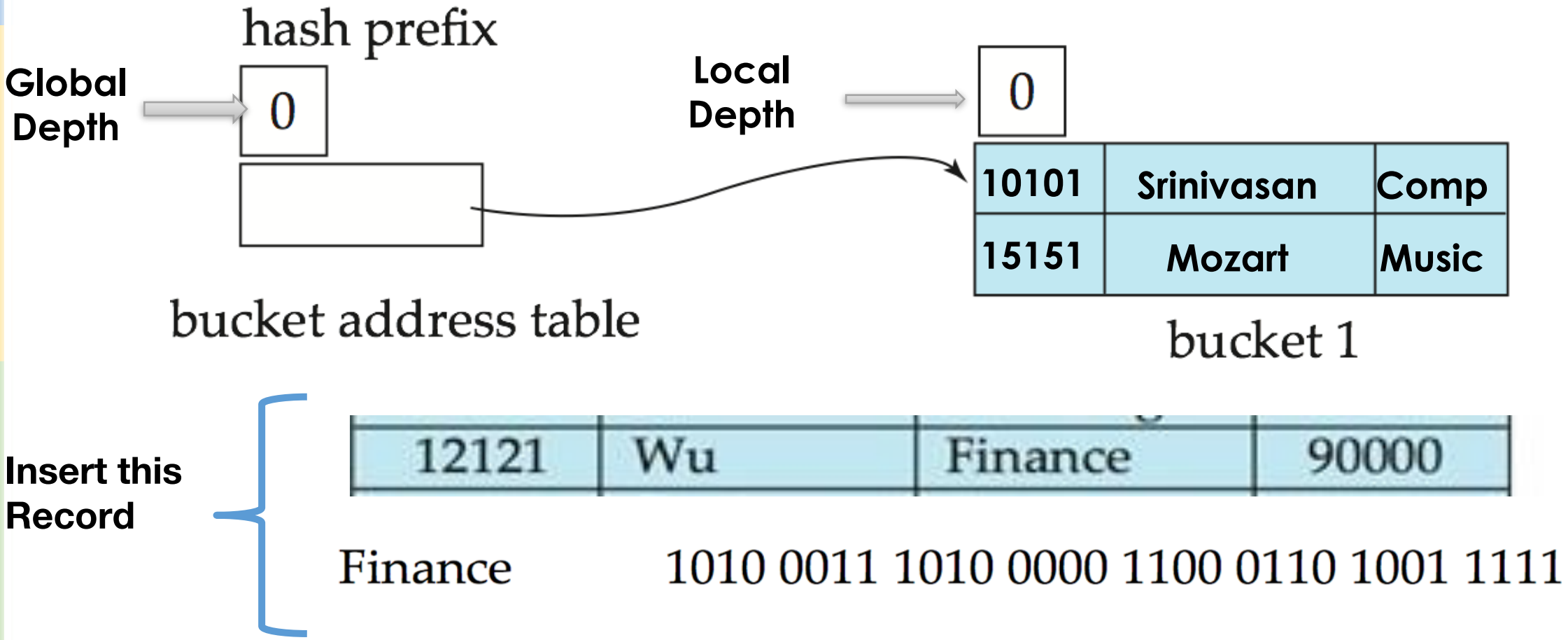
Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2



Illustrating an Extendible Hash

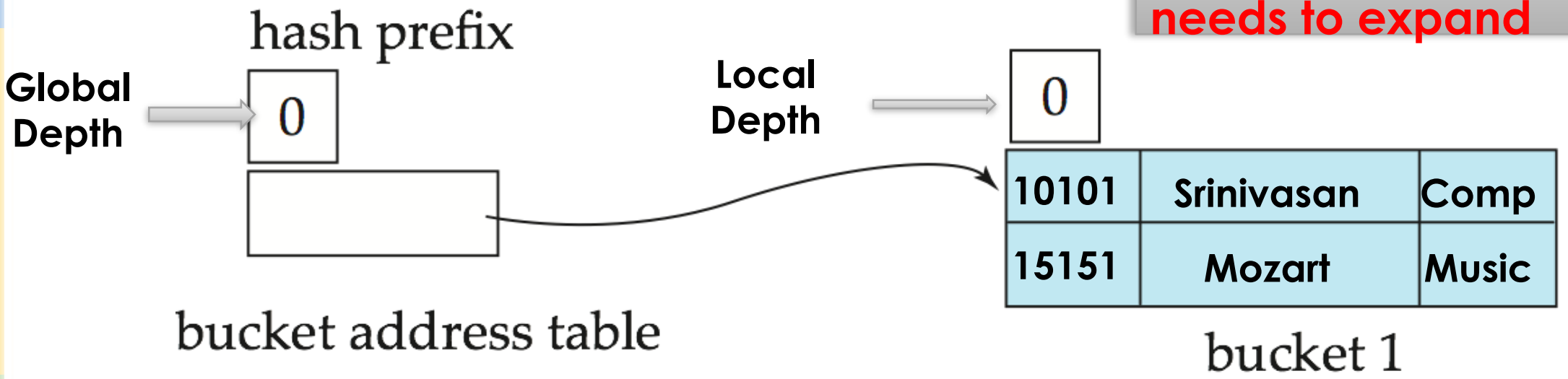
- Initial Hash structure; bucket size = 2



Illustrating an Extendible Hash

- Initial Hash structure; bucket size = 2

Local Depth == Global Depth
Bucket address table needs to expand



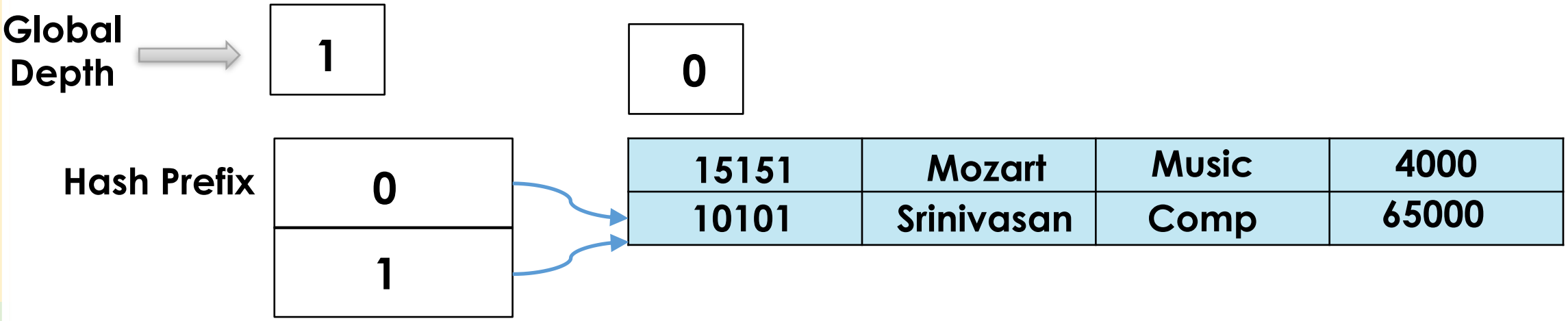
Insert this Record

12121	Wu	Finance	90000

Finance 1010 0011 1010 0000 1100 0110 1001 1111

Illustrating an Extendible Hash

- **Step 1: Increase the directory size. Each entry in directory spawns two children (one with 1 suffix and another with 0 suffix)**



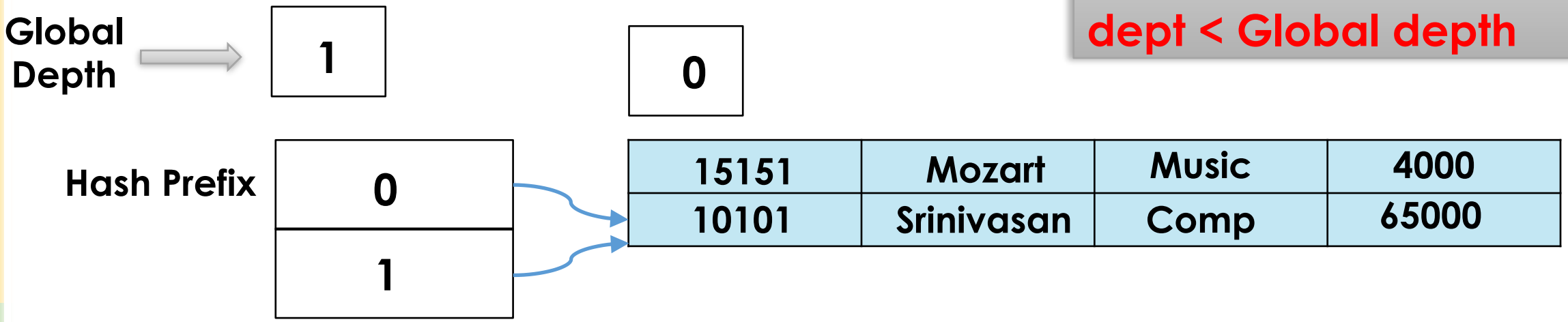
12121	Wu	Finance	90000

Finance 1010 0011 1010 0000 1100 0110 1001 1111

Illustrating an Extendible Hash

- **Step 2: Re-attempt to insert K_new**

Re-attempt to insert the Finance Dept record. It would be an overflow with local dept < Global depth

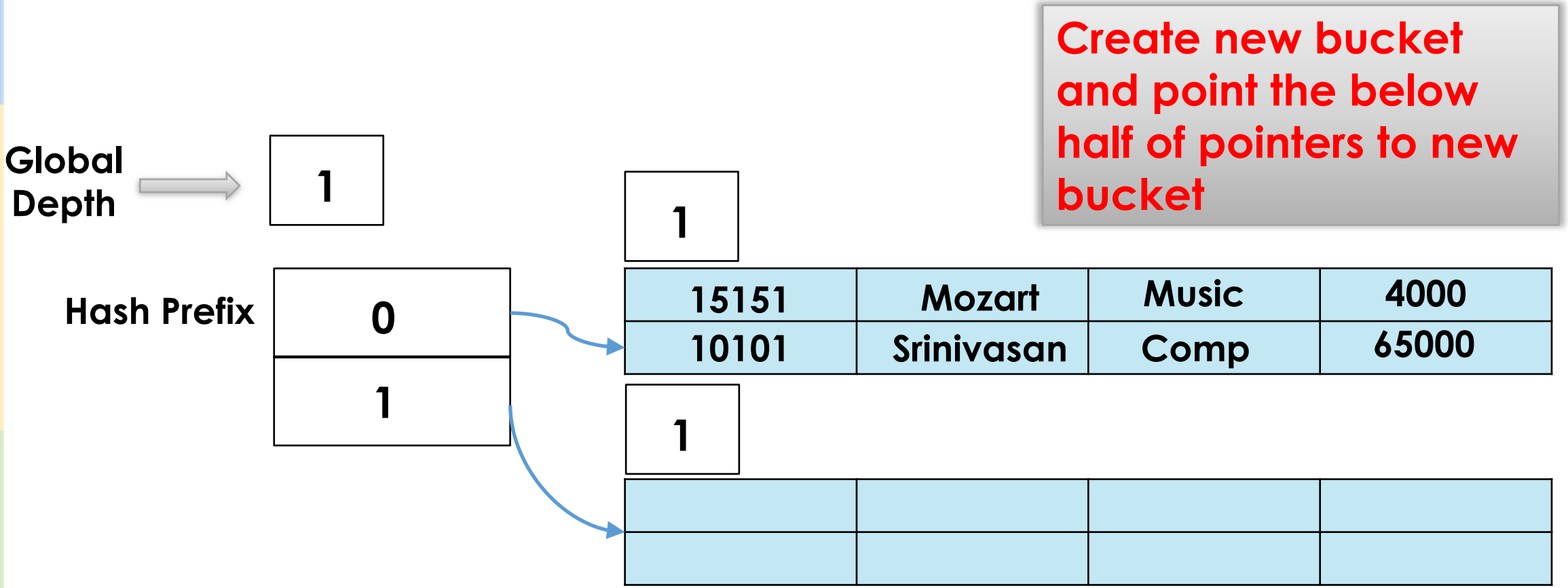


12121	Wu	Finance	90000

Finance 1010 0011 1010 0000 1100 0110 1001 1111

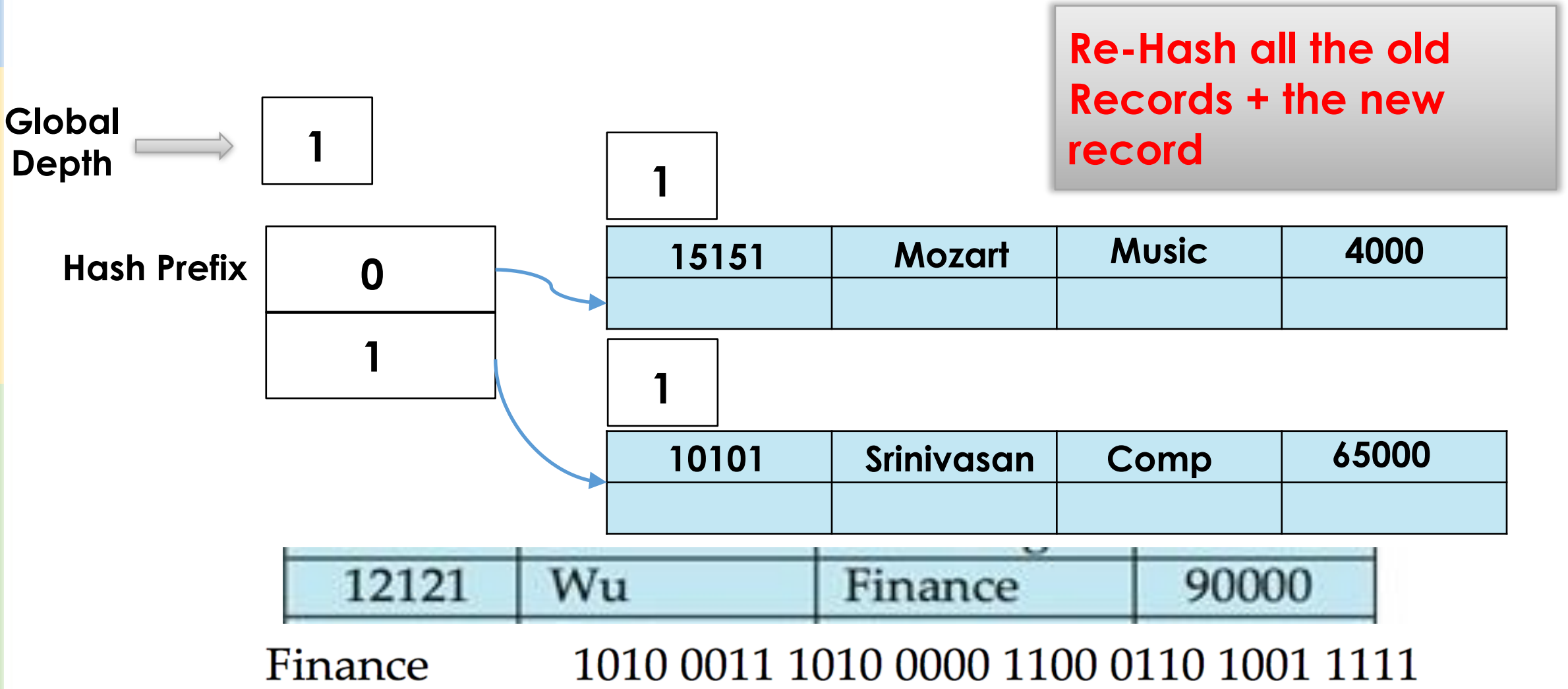
Illustrating an Extendible Hash

- **Handling overflow when local depth < Global Depth**



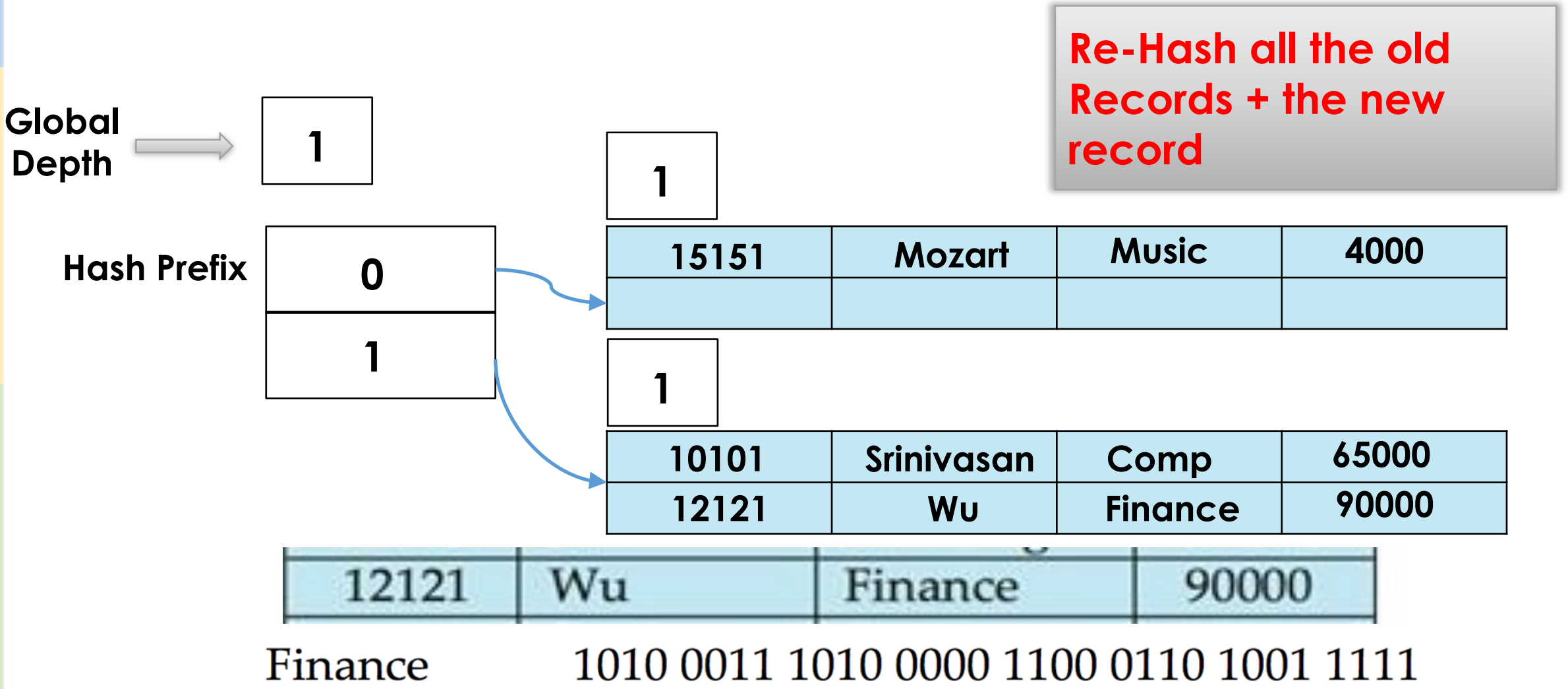
Illustrating an Extendible Hash

- **Handling overflow when local depth < Global Depth**

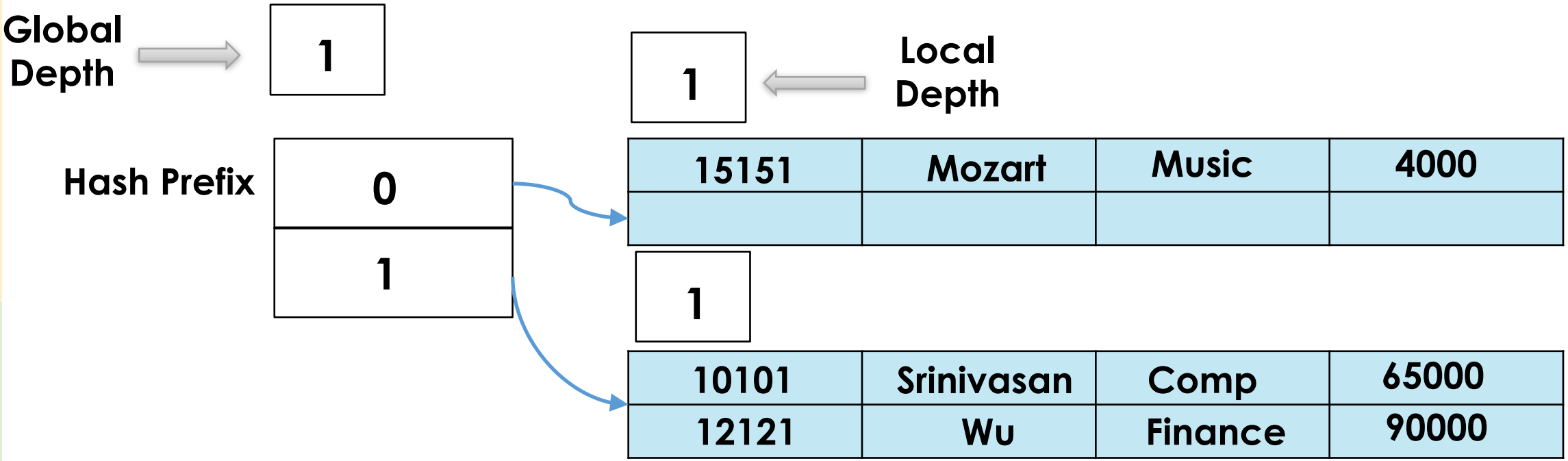


Illustrating an Extendible Hash

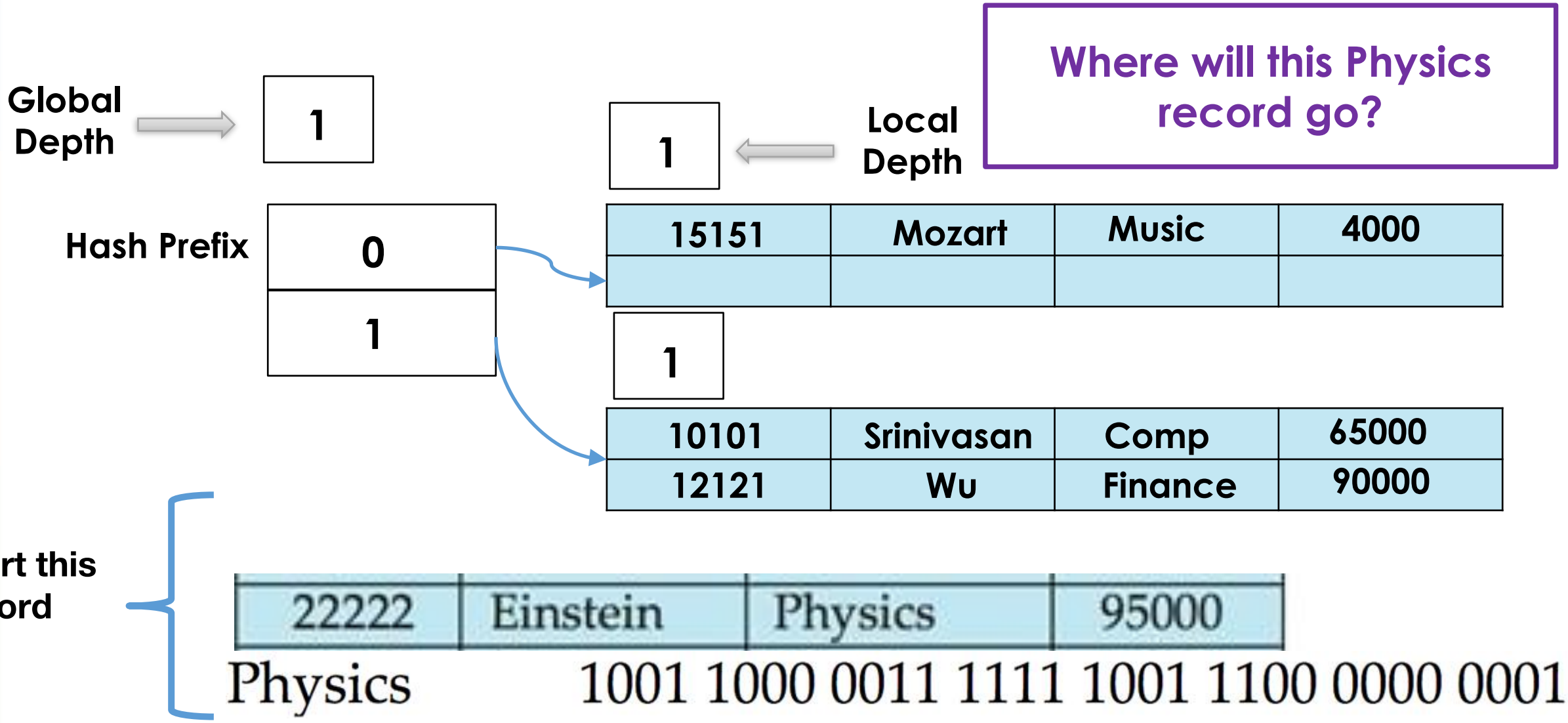
- **Handling overflow when local depth < Global Depth**



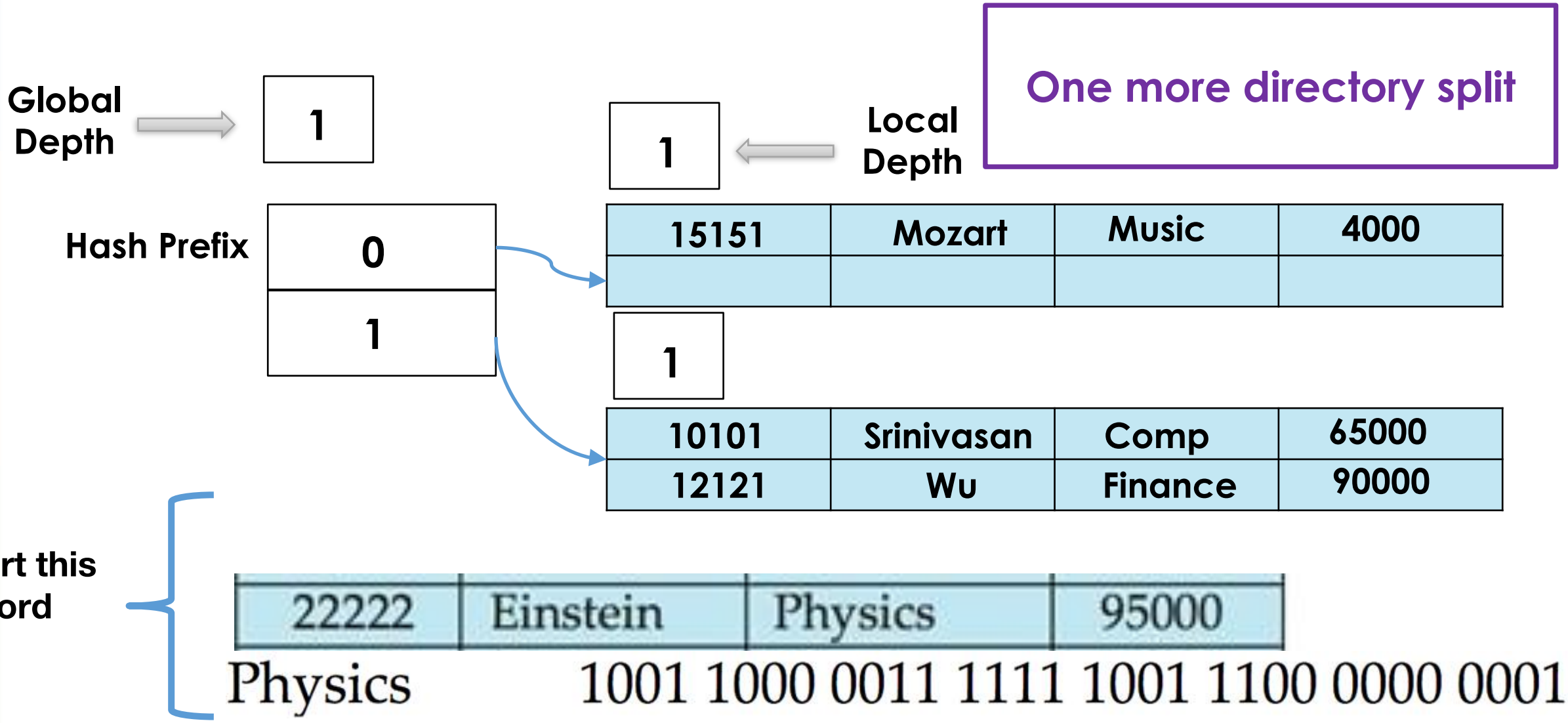
Illustrating an Extendible Hash



Illustrating an Extendible Hash

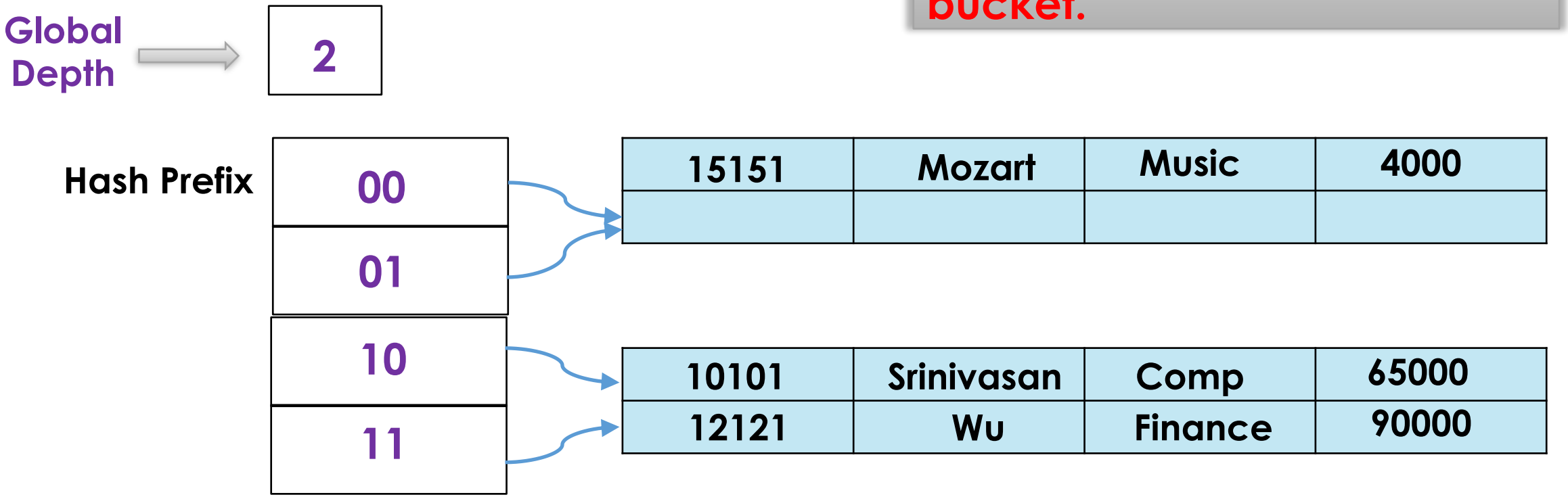


Illustrating an Extendible Hash



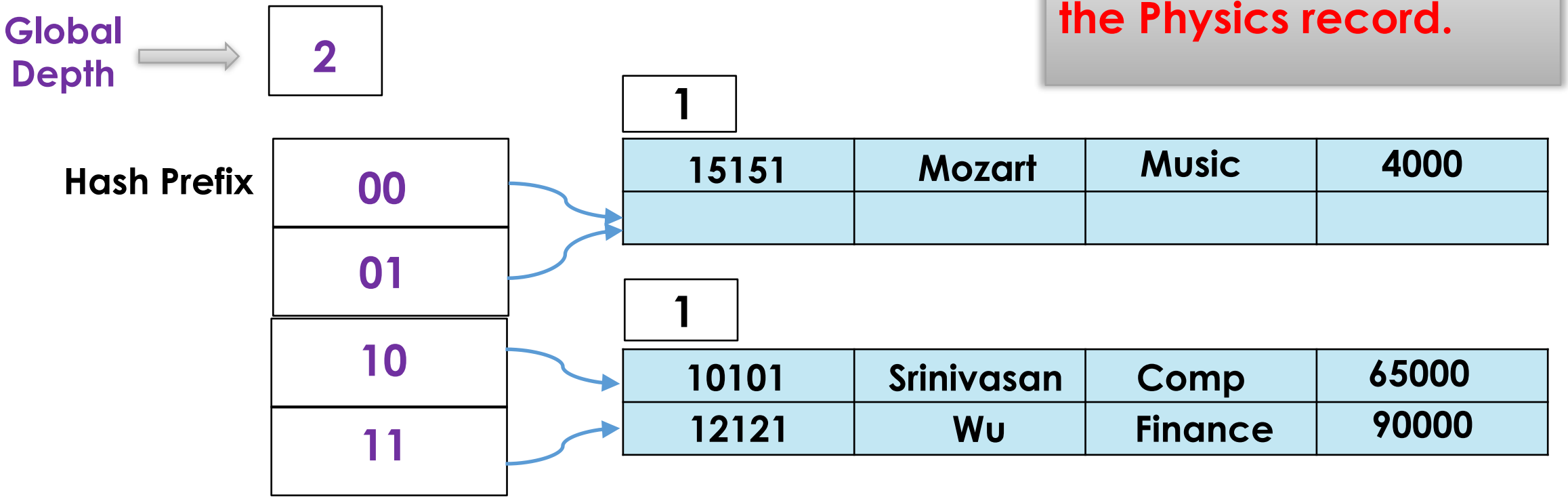
Illustrating an Extendible Hash

Expand directory. Children of a entry (e.g, 00 and 01 of prefix 0) would point to old bucket.



Illustrating an Extendible Hash

Re-attempt to insert the Physics record.

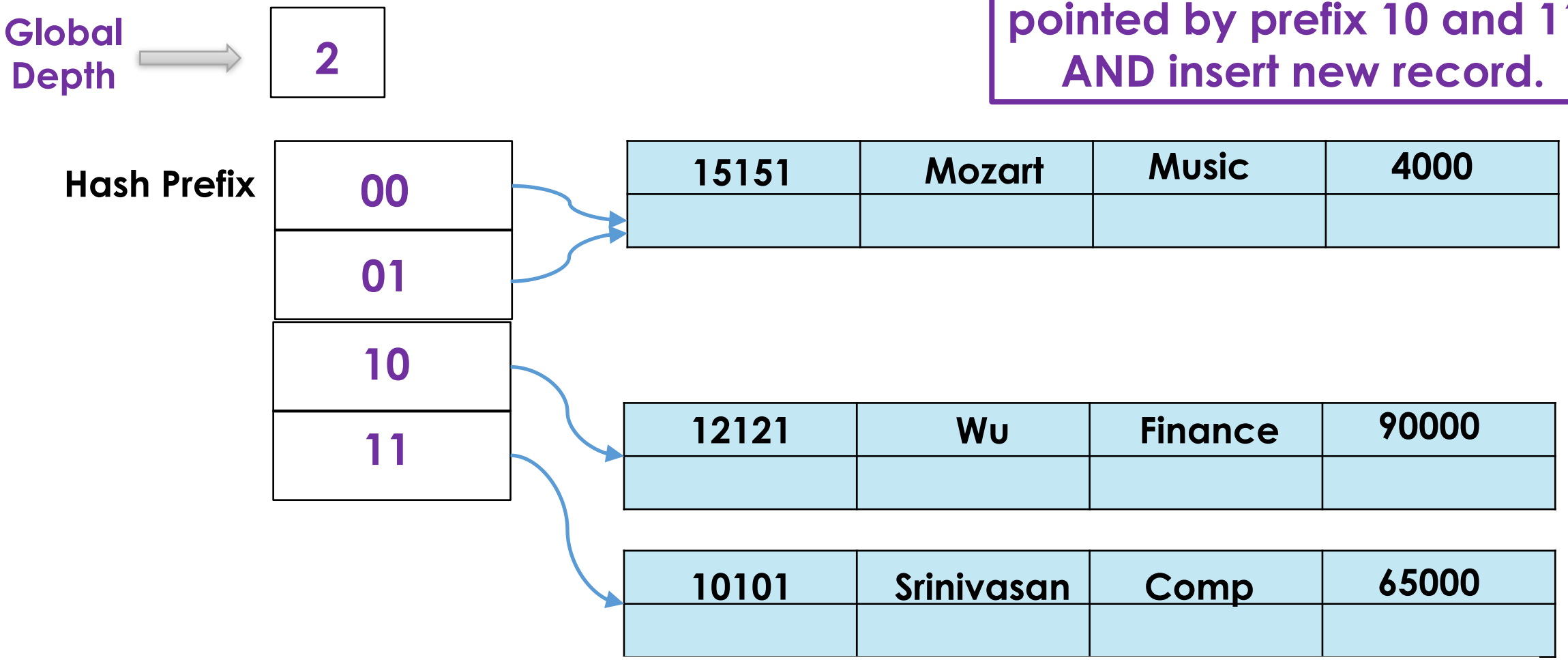


22222	Einstein	Physics	95000

Physics 1001 1000 0011 1111 1001 1100 0000 0001

Illustrating an Extendible Hash

Re-hash the records pointed by prefix 10 and 11. AND insert new record.



Comp. Sci. 1111 Finance 1010 0011 1010 0000 1100 0110 1001 1111

Illustrating an Extendible Hash

Re-hash the records pointed by prefix 10 and 11. AND insert new record.

Global Depth → 2

Hash Prefix

Hash Prefix
00
01
10
11

15151	Mozart	Music	4000

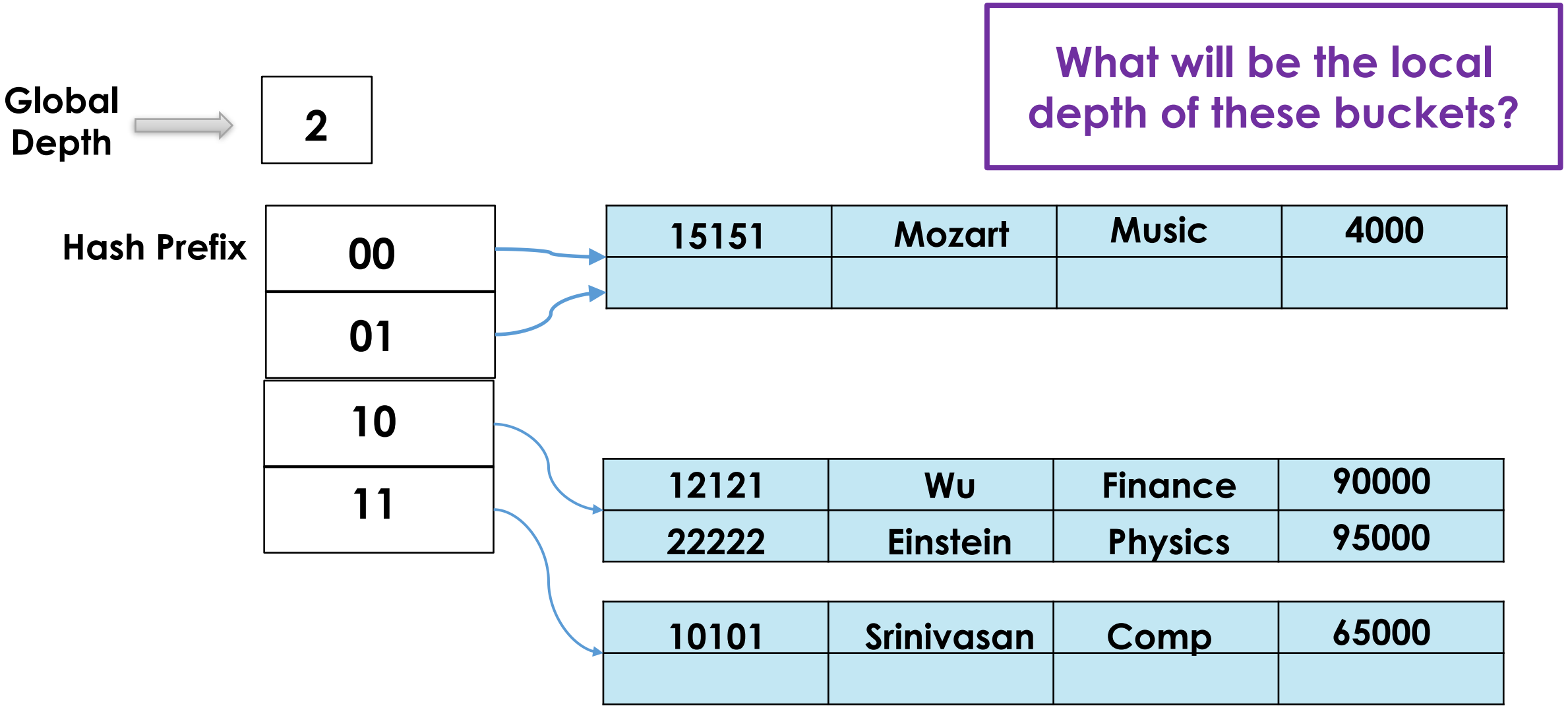
12121	Wu	Finance	90000
22222	Einstein	Physics	95000

10101	Srinivasan	Comp	65000

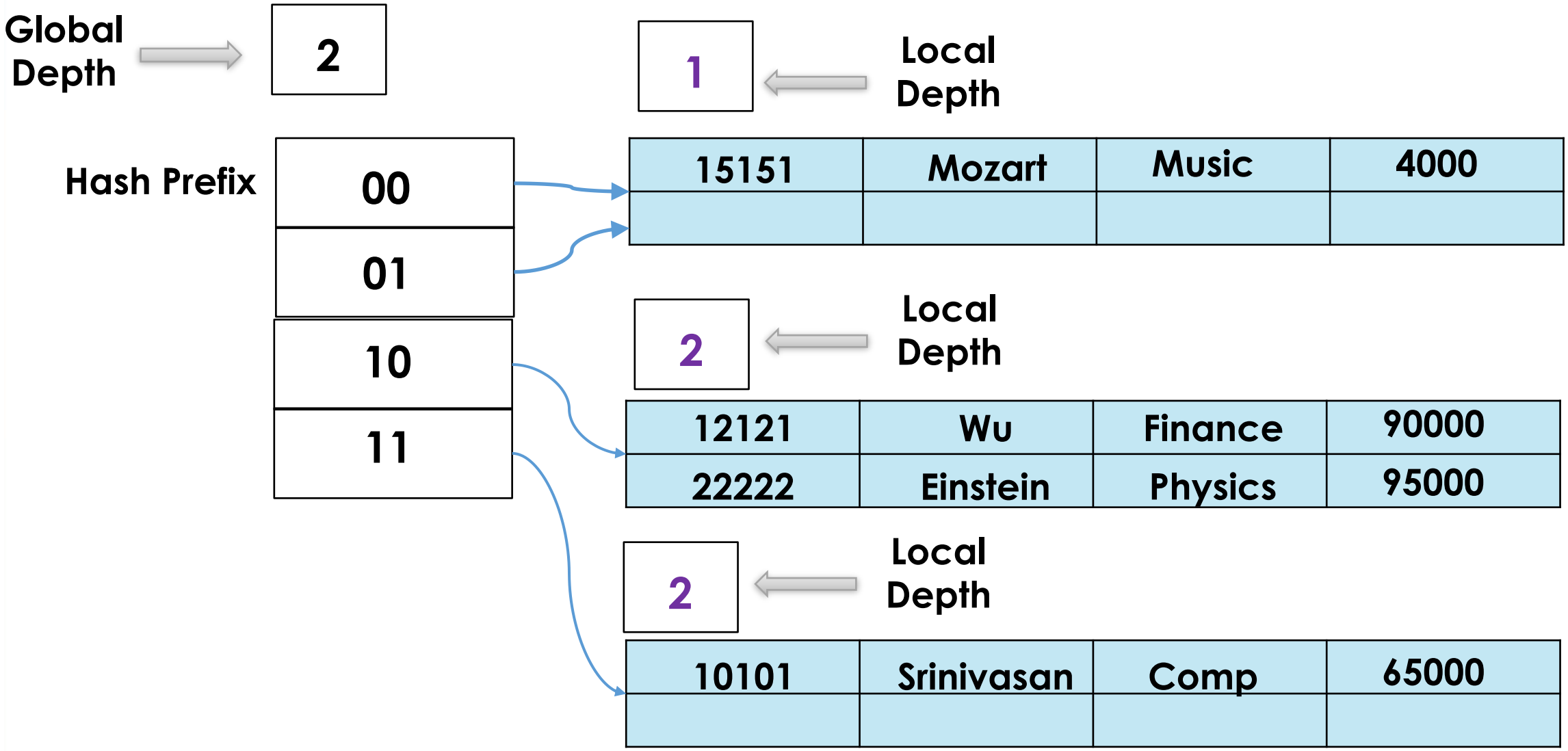
Physics 1001

22222	Einstein	Physics	95000

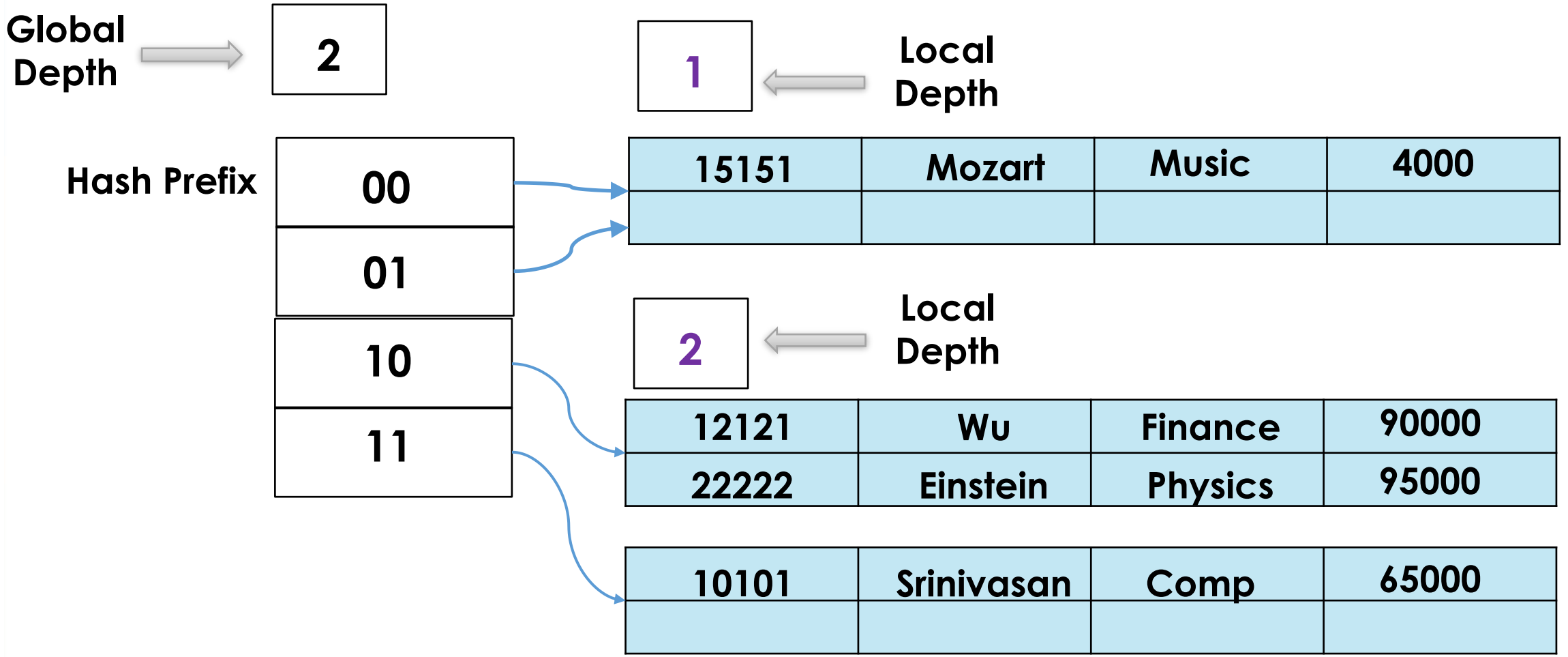
Illustrating an Extendible Hash



Illustrating an Extendible Hash

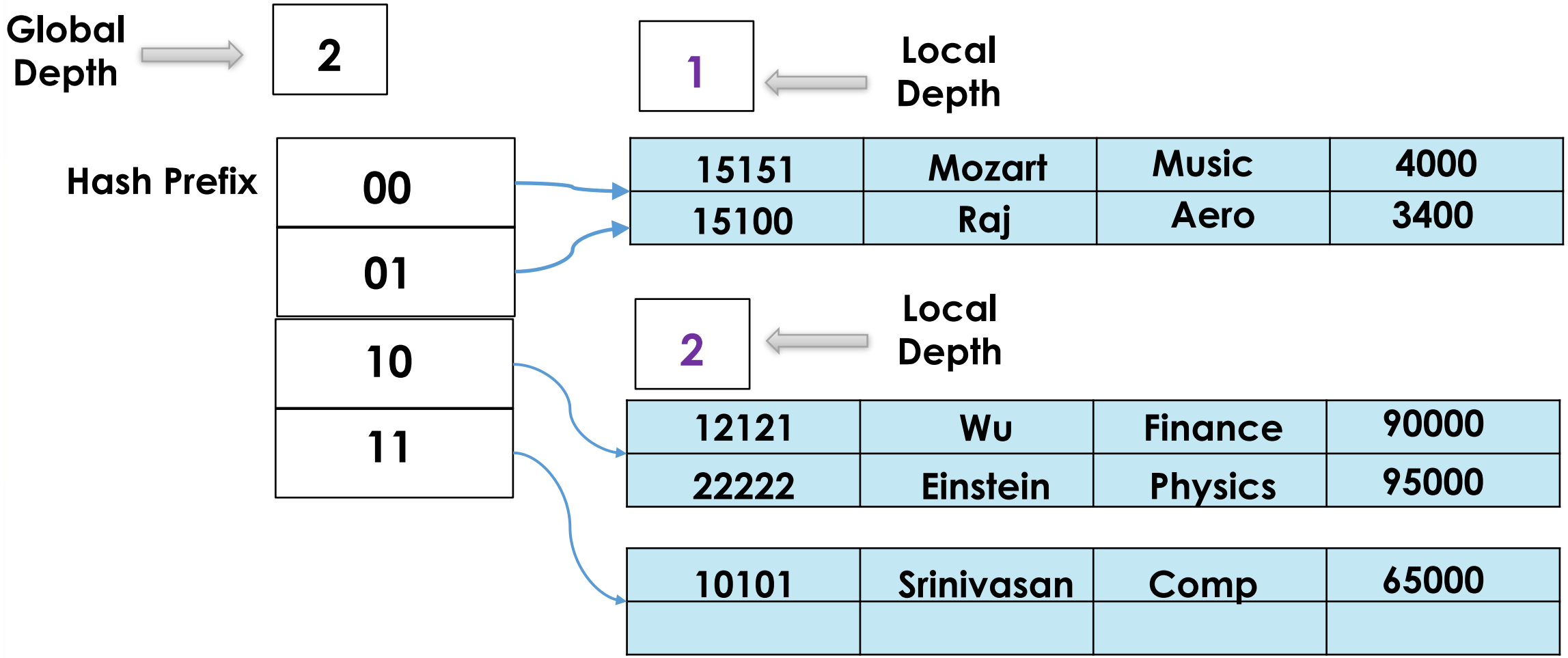


Illustrating an Extendible Hash

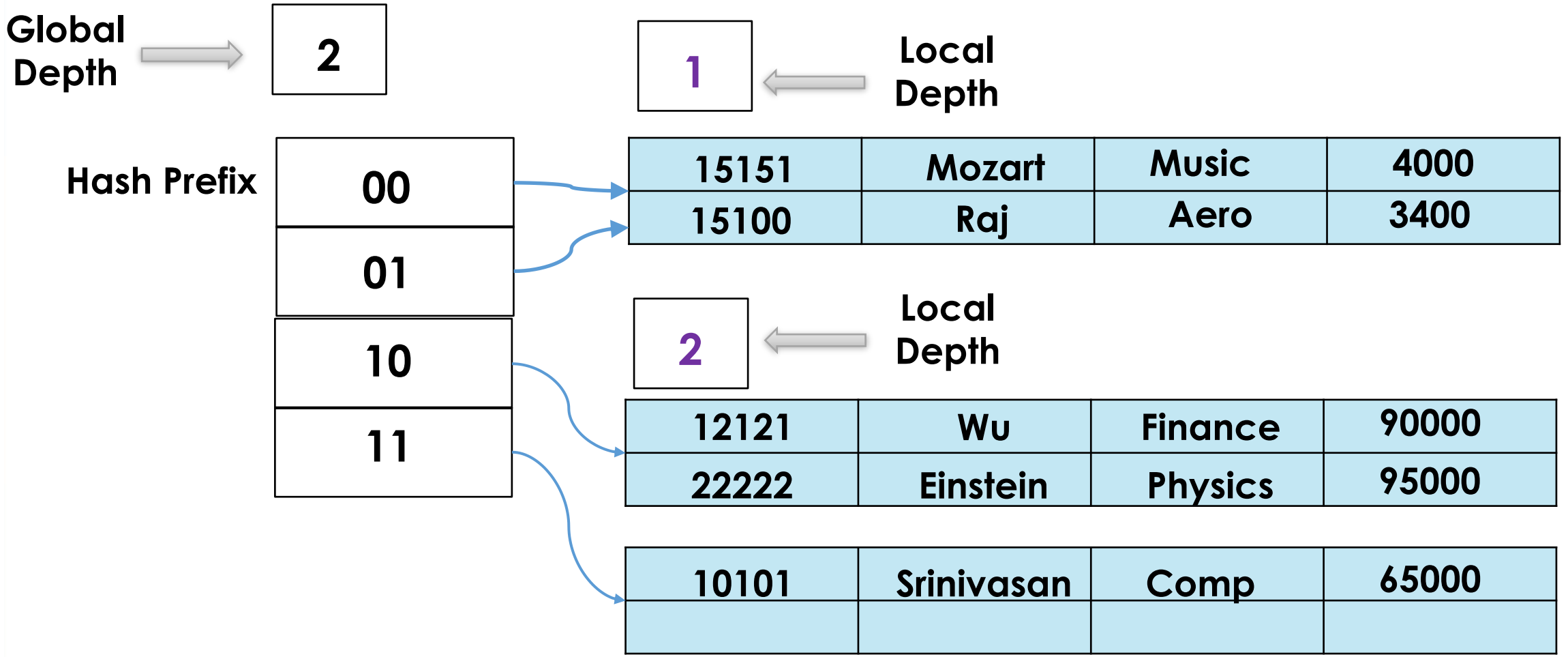


Insert one record Raj with Aero dept. Assume H(Aero) = 010……

Illustrating an Extendible Hash

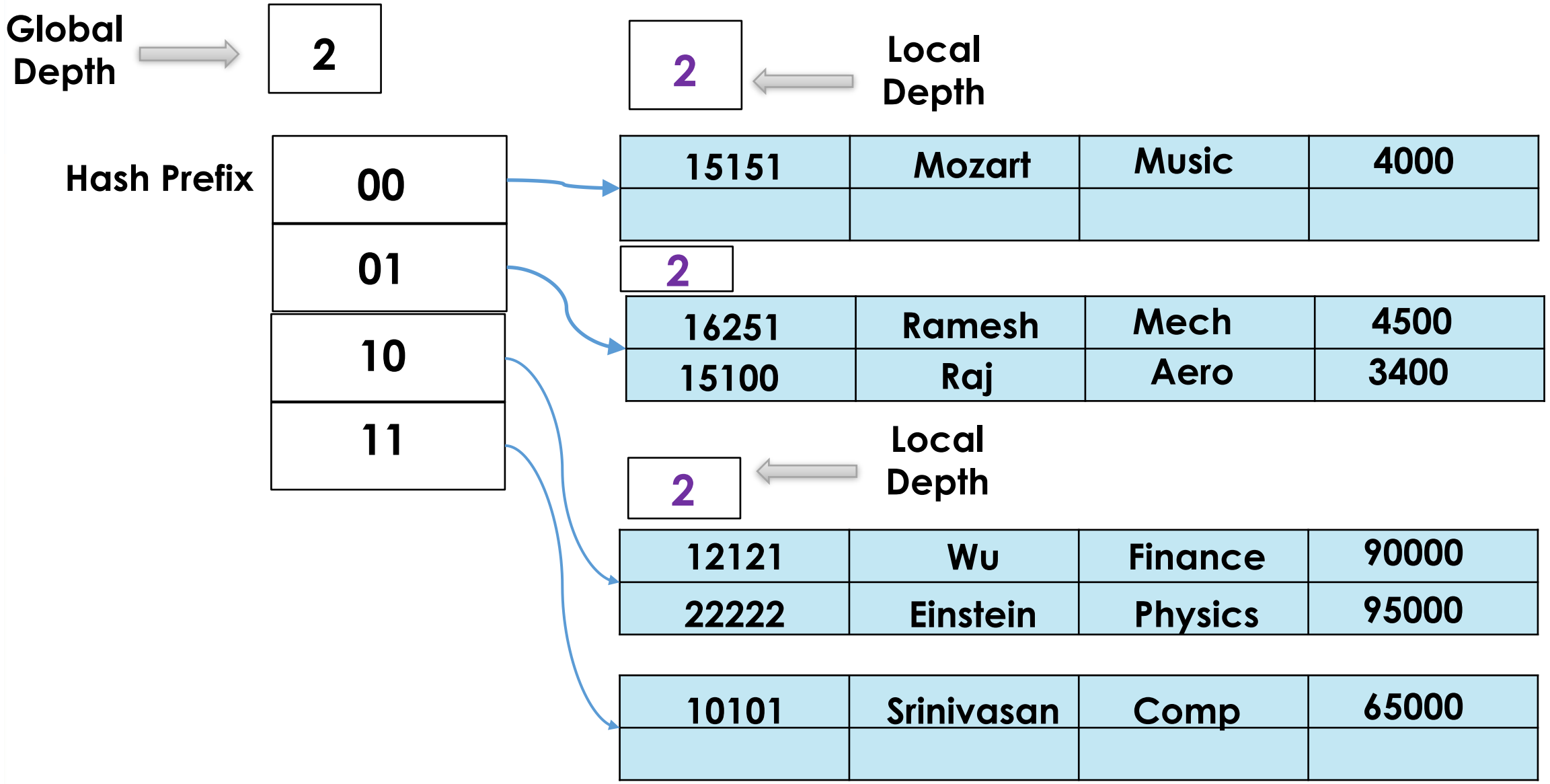


Illustrating an Extendible Hash

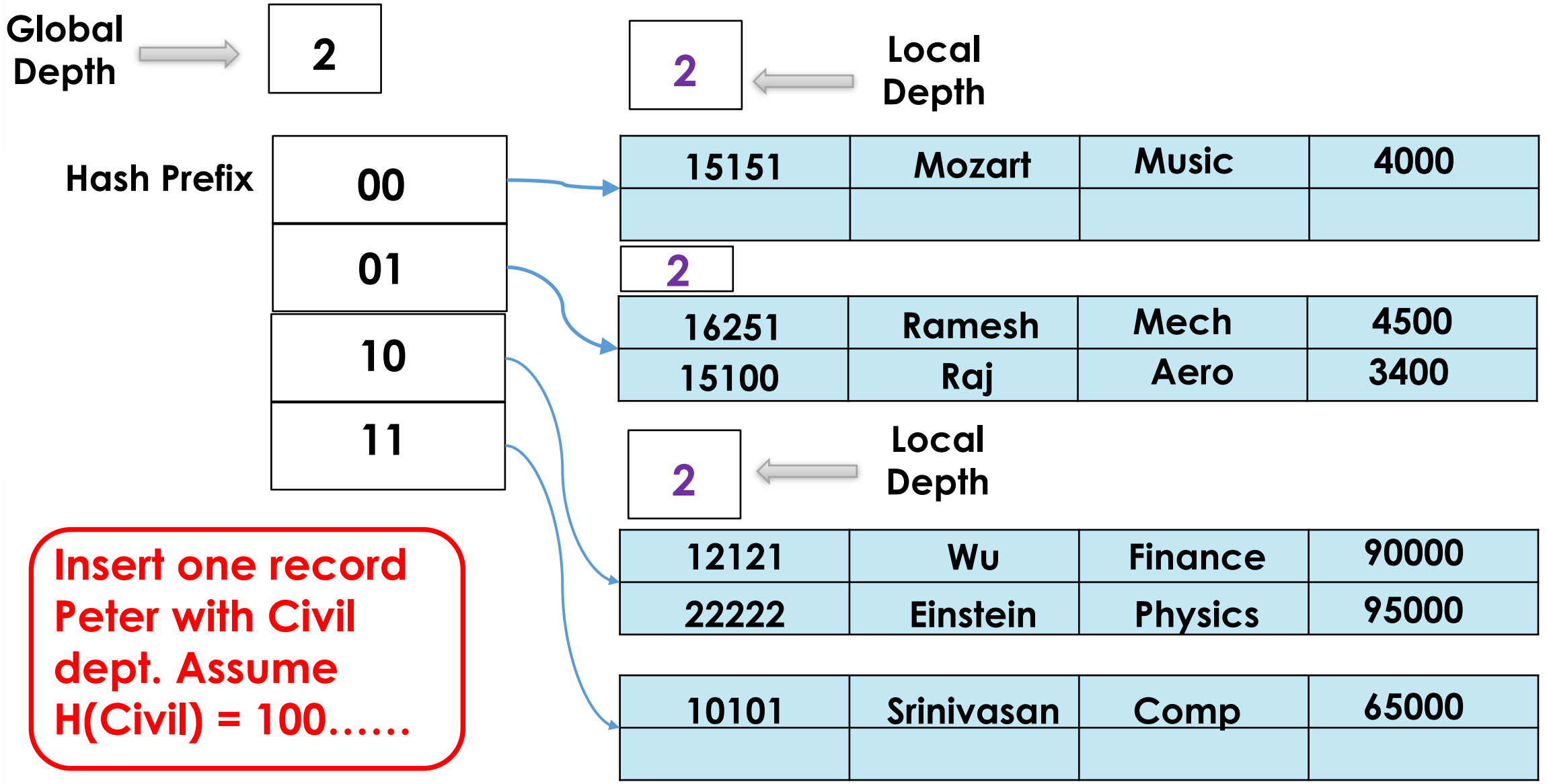


Insert one record with Ramesh Mech dept. H(Mech) = 011……

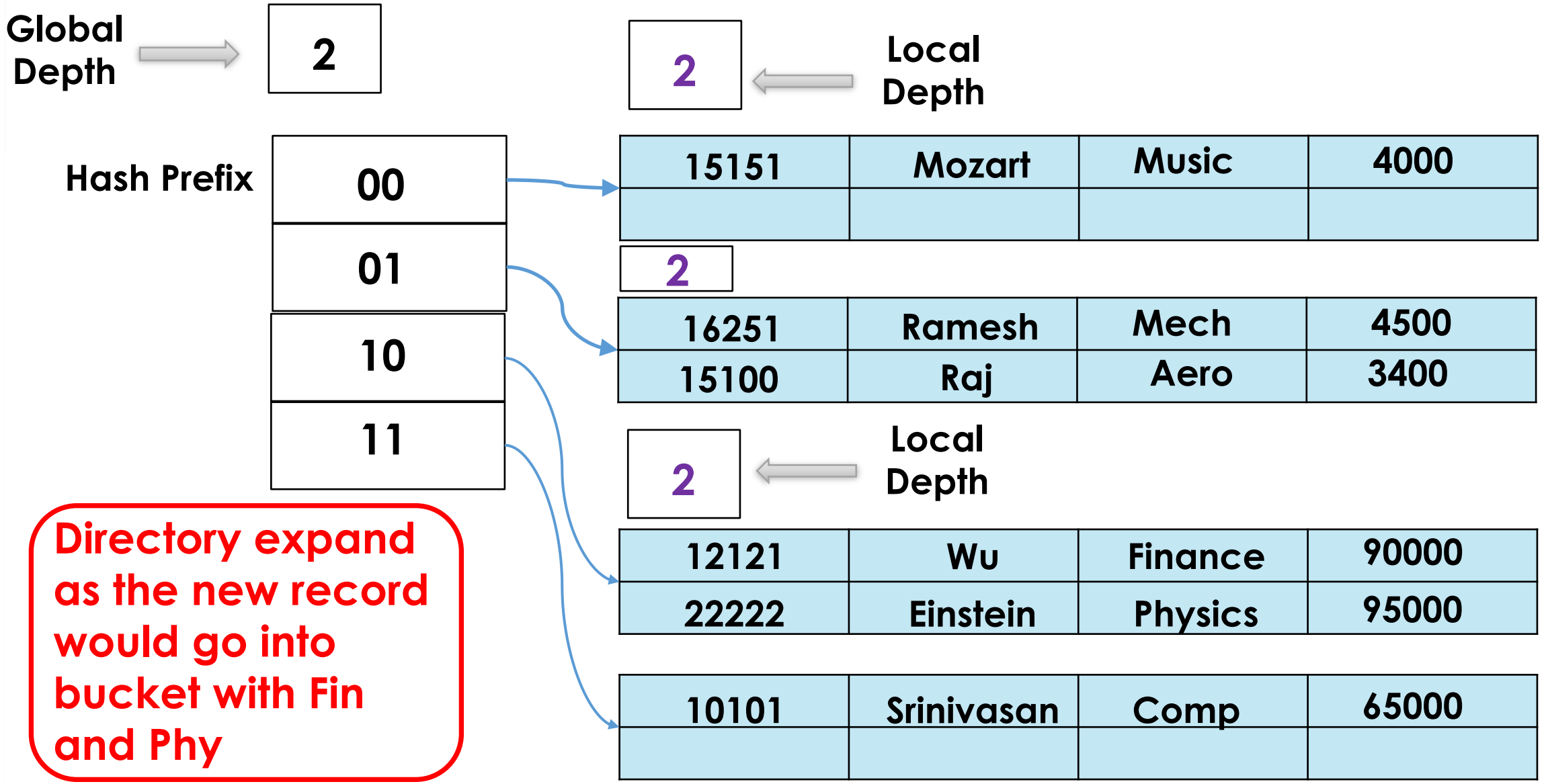
Illustrating an Extendible Hash

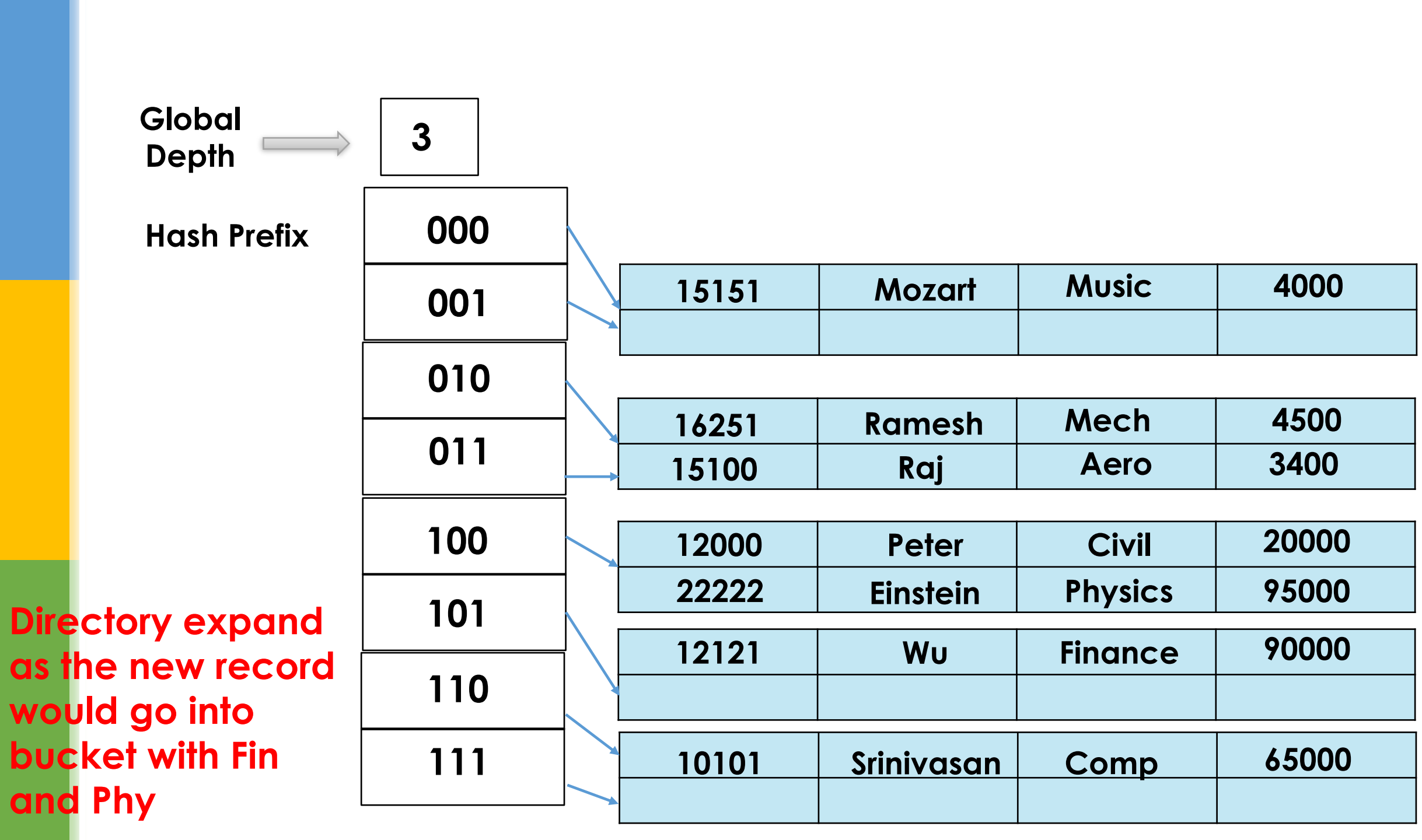


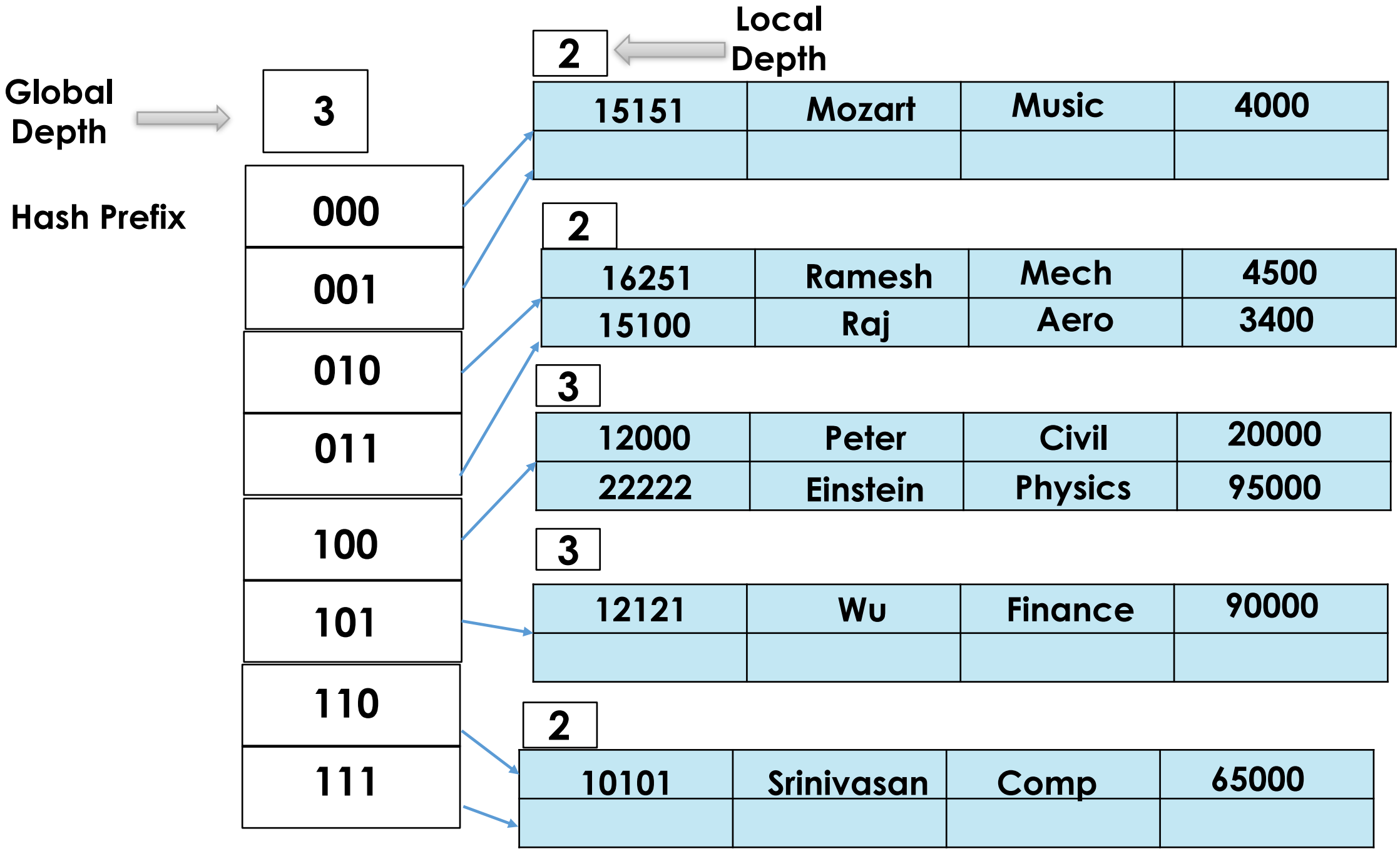
Illustrating an Extendible Hash



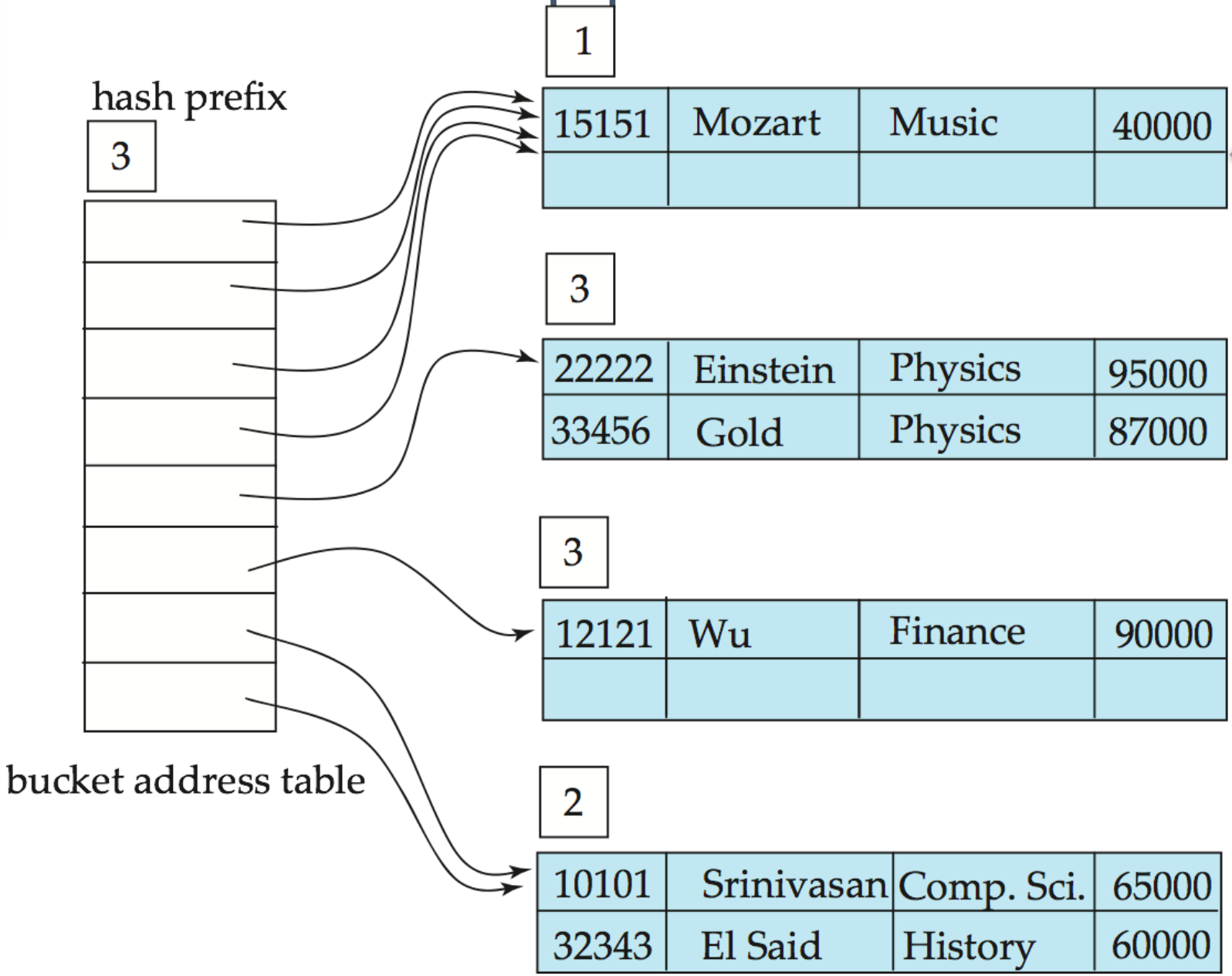
Illustrating an Extendible Hash




Global Depth
3
Hash Prefix
000
001
010
011
100
101
110
111
15151 Mozart Music 4000
16251 Ramesh Mech 4500
15100 Raj Aero 3400
12000 Peter Civil 20000
22222 Einstein Physics 95000
12121 Wu Finance 90000
10101 Srinivasan Comp 65000
Directory expand as the new record would go into bucket with Fin and Phy


Global Depth
3
Hash Prefix
000
001
010
011
100
101
110
111
Local Depth
2
15151
Mozart
Music
4000
2
16251
Ramesh
Mech
4500
15100
Raj
Aero
3400
3
12000
Peter
Civil
20000
22222
Einstein
Physics
95000
3
12121
Wu
Finance
90000
2
10101
Srinivasan
Comp
65000

What will happen here?



Assume two more records come to this bucket

Comments on Extendible Hash

- **Benefits of extendable hashing:**
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- **Disadvantages of extendable hashing**
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big
    - Cannot allocate very large contiguous areas on disk either
  - Changing size of directory (aka bucket address table) is expensive.

Comments on Extendible Hash

- **Expected type of queries:**
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred