Hashing

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 of Hash File Organization

- Hash file organization of *instructor* file, using *dept_name* as key
  - 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 of Hash File Organization

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

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

Example of Hash File Organization

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file

- 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
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned

Handling of Bucket Overflows

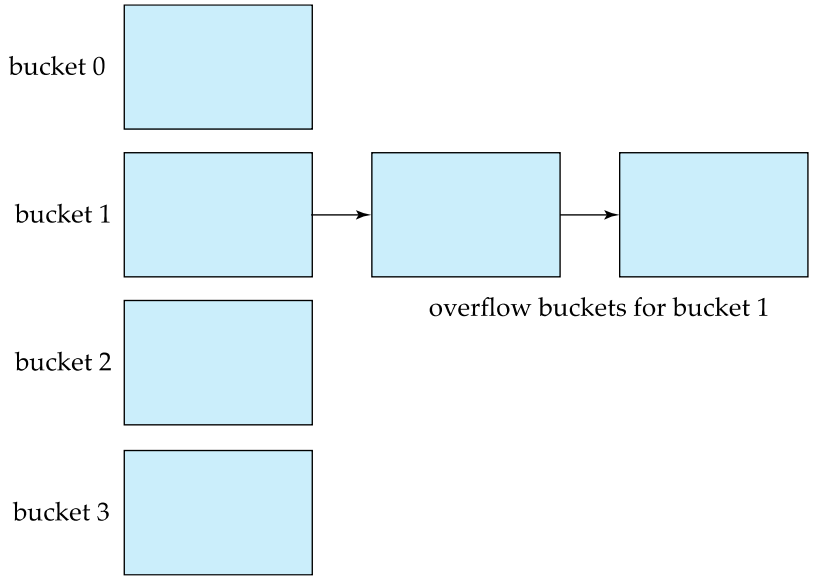
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records; this can occur due to two reasons:
    - Multiple records have same search-key value
    - Chosen hash function produces non-uniform distribution of key values

- Although the probability of bucket overflow can be reduced, it cannot be eliminated;
  - It is handled by using ***overflow buckets***

Handling of Bucket Overflows

- **Overflow chaining:** The overflow buckets of a given bucket are chained together in a linked list
  - Above scheme is called **closed addressing**

- An alternative, called **open addressing**, which does not use overflow buckets, is not suitable for database applications



Hash Indices

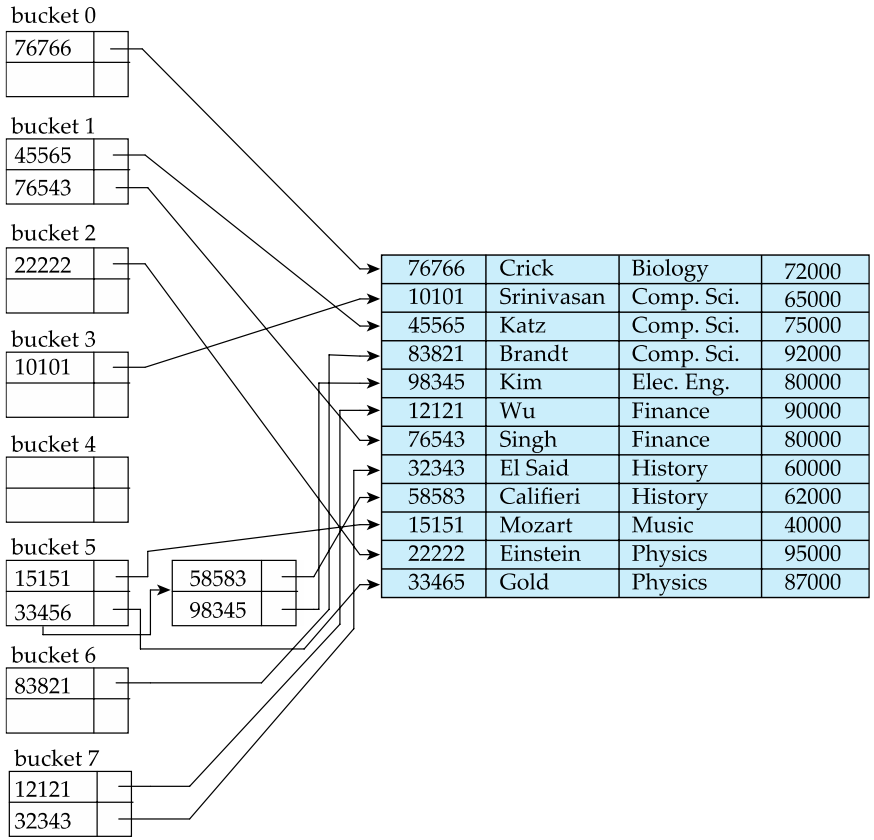
- Hashing can be used not only for file organization, but also for index-structure creation

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure

- Strictly speaking, hash indices are always secondary indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
  - However, we use the term hash index to refer to both secondary index structures and hash organized files

Example of Hash Index

- Hash index on *instructor,* on attribute *ID*
- Computed by adding the digits modulo 8



Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses, as 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 (and buckets will be underfull)
  - If database shrinks, again space will be wasted

- One solution: Periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations

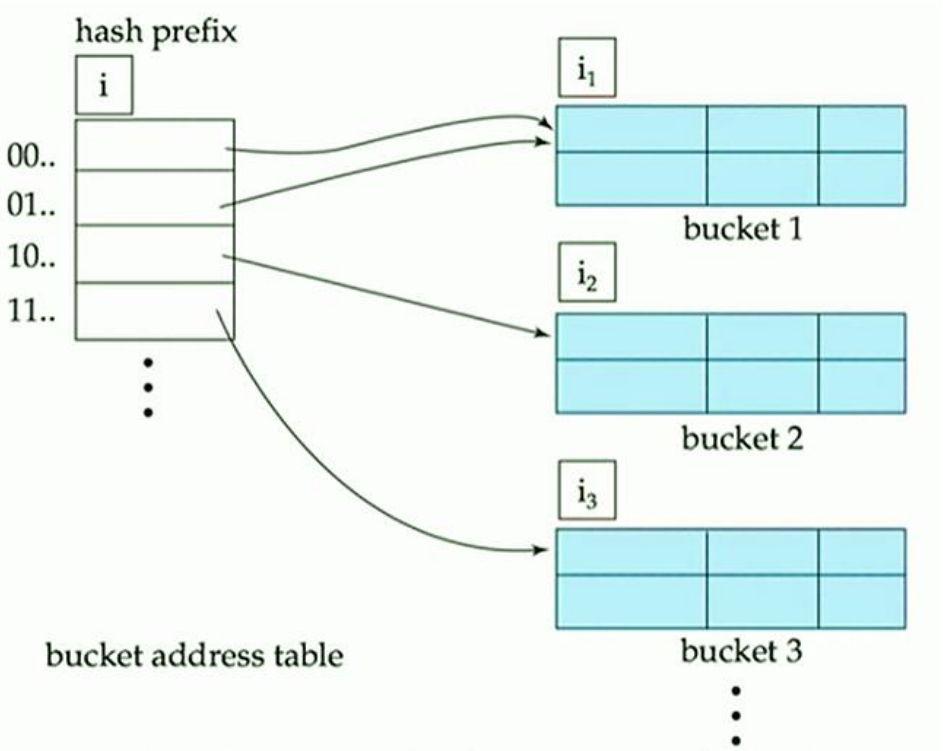
- Better solution: Allow the number of buckets to be modified dynamically

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable 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
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$
    - 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
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets

General Extendable Hash Structure

- In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$
- Decode i_j number of bits to find the record in bucket j
- $i_j \leq i$



Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits

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

- To insert a record with search-key value K_j
- Follow 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
  - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure

- To split a bucket j when inserting record with search-key value K_j
  - If $i > i_j$ (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_j and insert record in the bucket (further splitting is required if the bucket is still full)

- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - Increment i and double the size of the bucket address table
    - Replace each entry in the table by two entries that point to the same bucket
    - Recompute new bucket address table entry for K_j
    - Now $i > i_j$ so use the first case above

Deletion in Extendable Hash Structure

- To delete a key value
  - Locate it in its bucket and remove it
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table)
  - Coalescing of buckets can be done (can coalesce only with a “buddy” bucket having same value of i_j and same i_j–1 prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

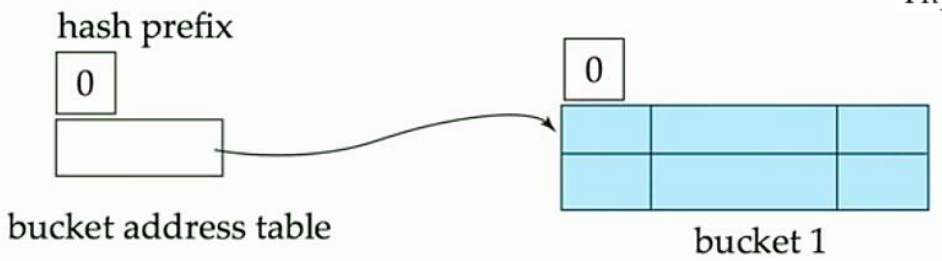
Use of Extendable Hash Structure: Example

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

Use of Extendable Hash Structure: Example

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

- Initial Hash structure; bucket size = 2



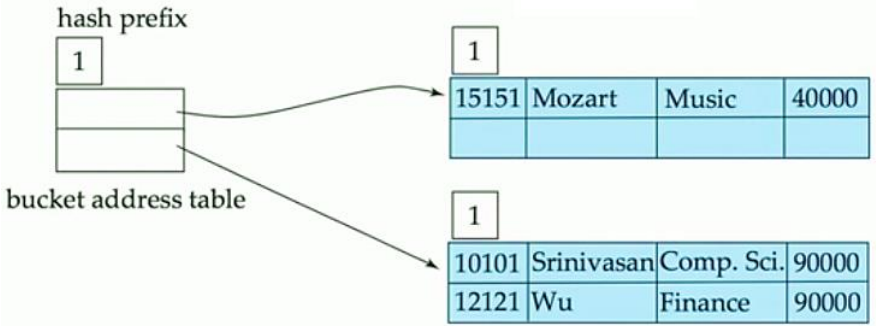
- Insert "Mozart", "Srinivasan", and "Wu" records

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

Use of Extendable Hash Structure: Example

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

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

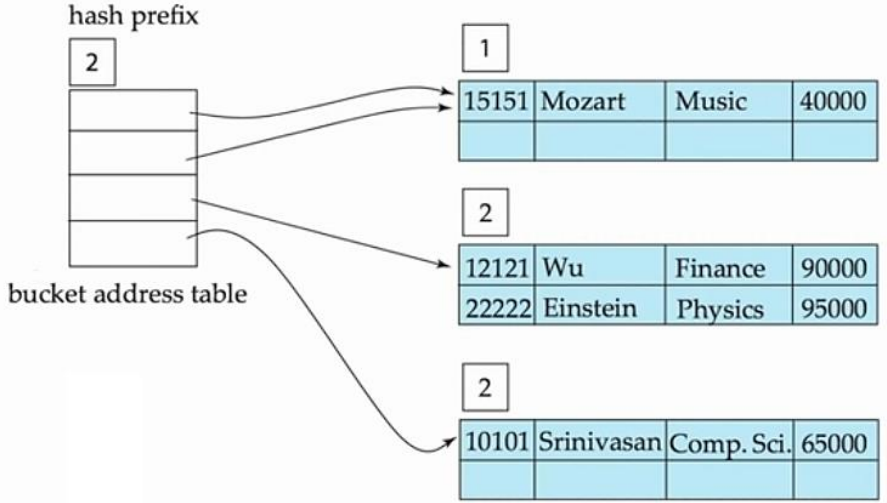


- Insert Einstein record

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

Use of Extendable Hash Structure: Example

- Hash structure after insertion of Einstein record



- Insert Gold and El Said records

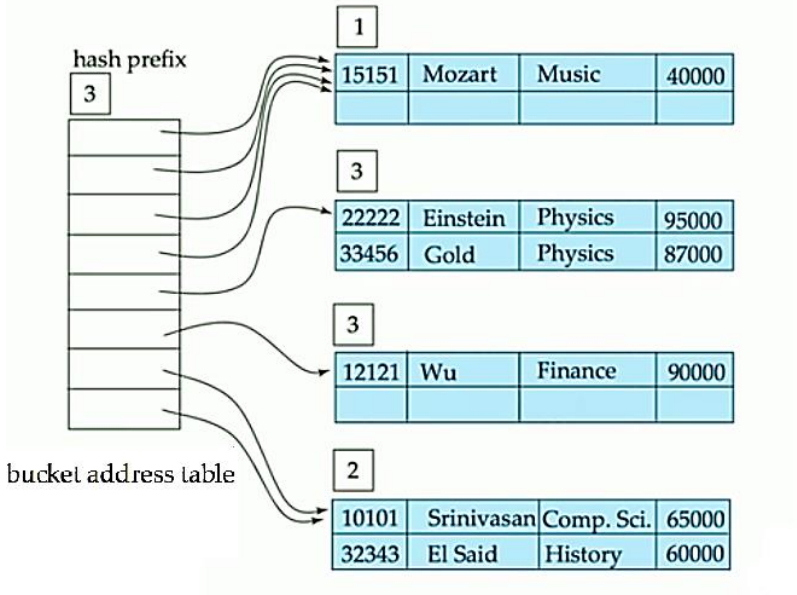
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

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

Use of Extendable Hash Structure: Example

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

- Hash structure after insertion of Gold and El Said records

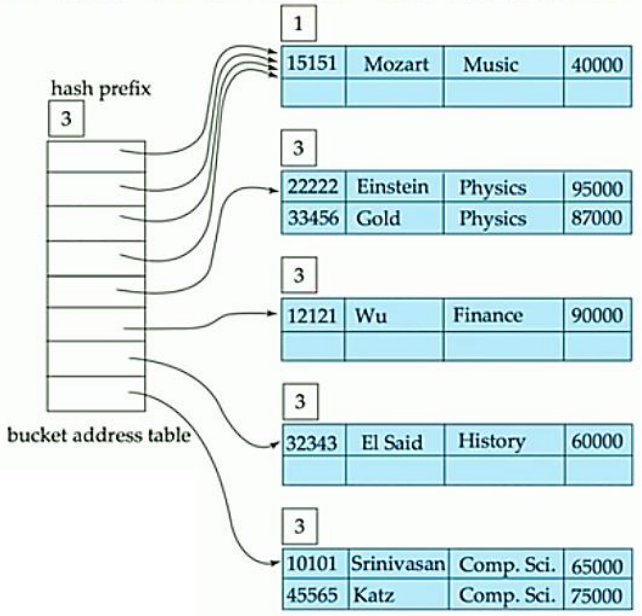


- Insert Katz record

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

Use of Extendable Hash Structure: Example

- Hash structure after insertion of Katz record



- Insert Singh, Califieri, Crick, Brandt record

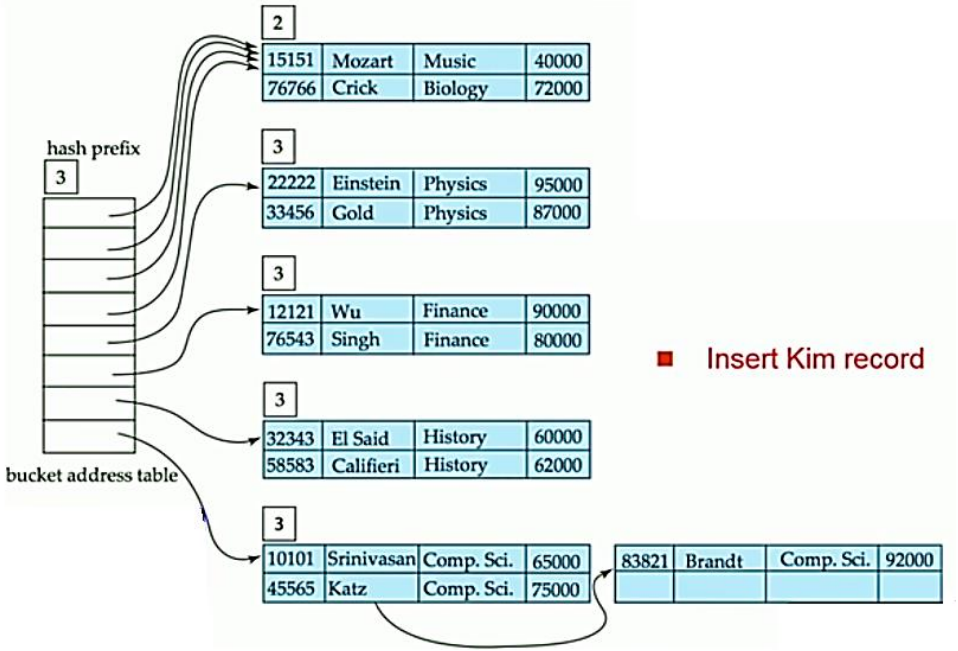
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

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

Use of Extendable Hash Structure: Example

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

- Hash structure after insertion of Singh, Califieri, Crick, Brandt records

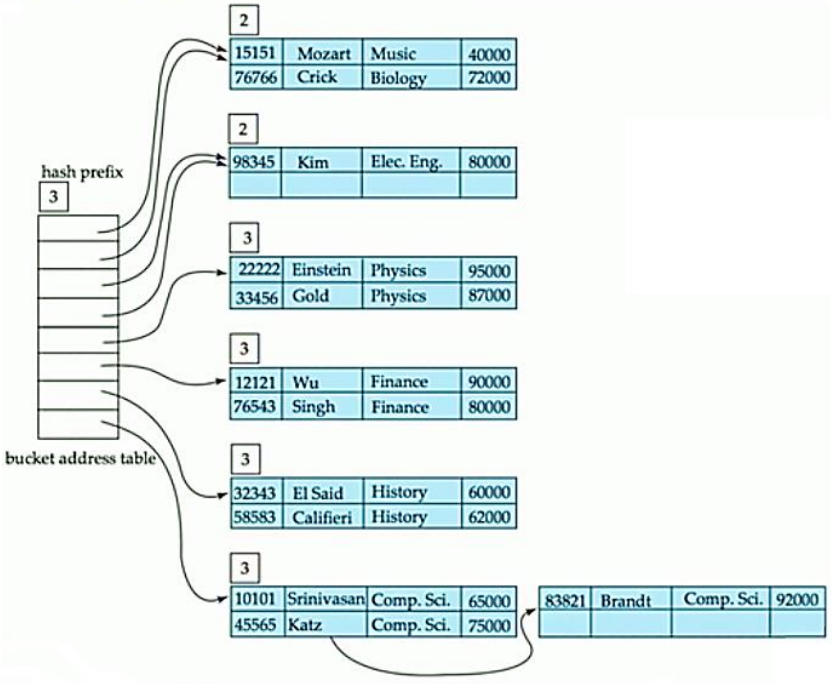


- Insert Kim record

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

Use of Extendable Hash Structure: Example

- Hash structure after insertion of Kim record



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

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

Extendable Hashing vs. Other Schemes

- 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 (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation

- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- 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

- **In practice**:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

- **Bitmap Indices**

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in