Database Development and Design (CPT201)

Lecture 3c:
Hash-based Indexing

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Hash-based Indexing
  - Static Hashing
  - Dynamic Hashing
- Comparison of Ordered Indexing and Hash-based Indexing

Xi'an Jiaotong-Liverpool University 西交利物浦大學

Structure of Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
- 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.

Hash Functions

- 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, i.e., it does not depend on the actual distribution of search-key values in the file.
- If we have N buckets, numbered 0 to N-1, a hash function h of the following form works well in practice.
  - h(value) = (a*value + b) mod N

An Example of Hash Function

- 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.
- Assume that
  - There are 10 buckets,
  - The binary representation of the ith character in the alphabet is assumed to be the integer I
- The hash function returns the sum of the binary representations of the characters modulo 10
  - h(Perryridge) = (16+5+18+18+25+18+9+ 4+7+5) Mod 10 =5
  - h(RoundHill) = 3
  - h(Brighton) = 3

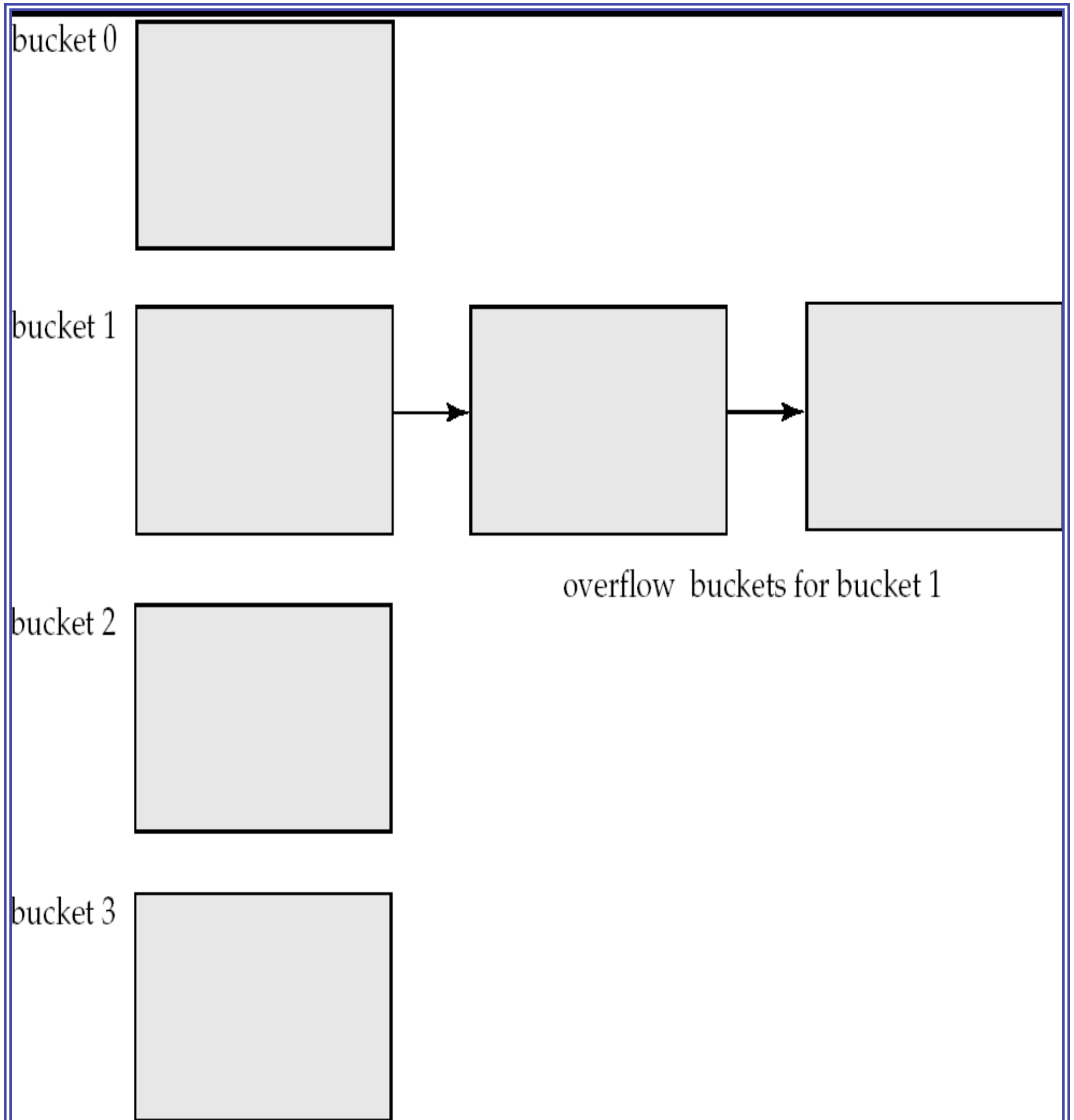
A	B	C	D	E	F	G	H	I	G	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	25	25	26

Xi'an Jiaotong-Liverpool University 西交利物浦大學

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.
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.

Structure of Static Hashing cont'd



Hash File Organisation

- Hash file organisation, the records in a file is stored in the buckets
- Hash file organisation of account file, using branch_name as key.

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

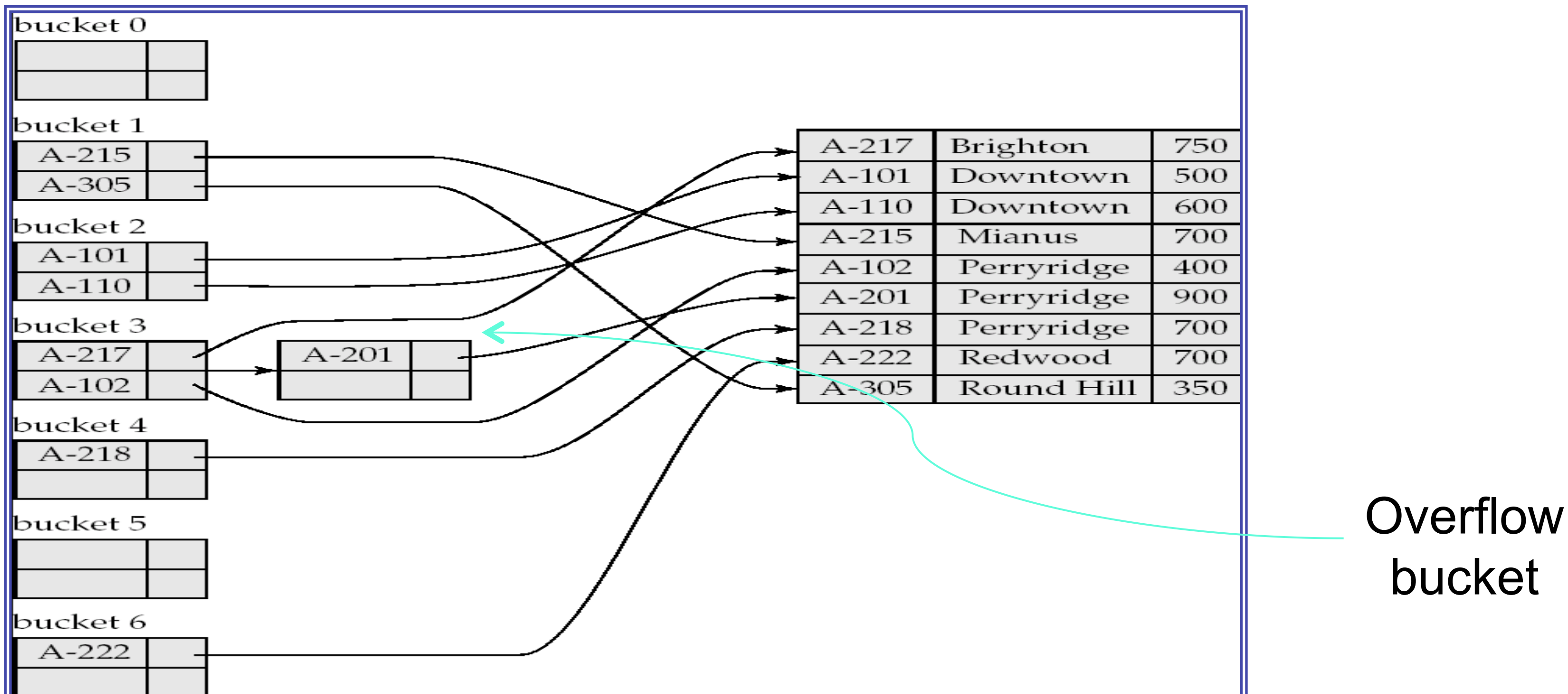
Hash Indices

- Hashing can be used not only for file organisation, but also for index-structure creation.
- A hash index organises the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices.

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Example of Hash Index

- Assume that each Bucket can only contains two (key, pointer) pairs.
- The hash function h used here computes the sum of digits of a account number module by 7, e.g., h(A-217)=(2+1+7) mod 7=3.



Deficiencies of 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 many 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-organisation 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!

Dynamic Hashing

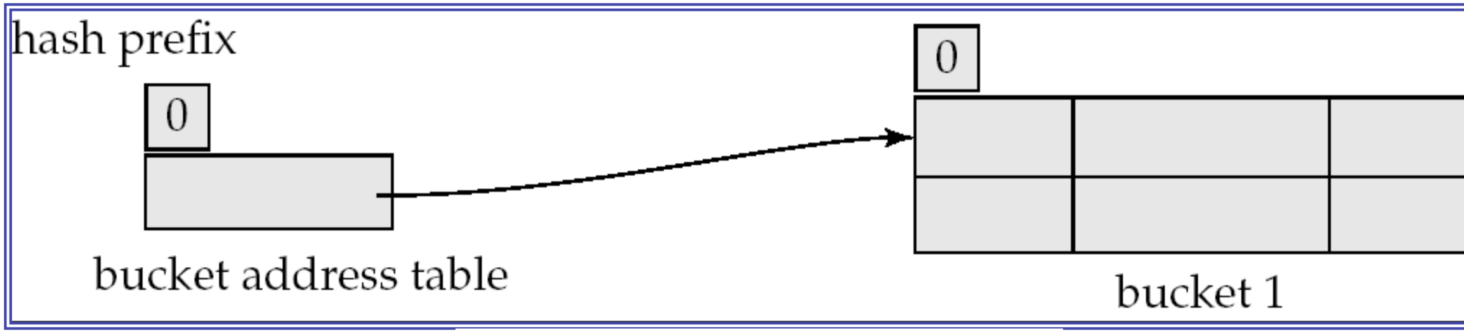
- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers, e.g. b = 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be i bits, $0 \le i \le 32$.
    - 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 the same bucket.
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

Example of Binary Representation

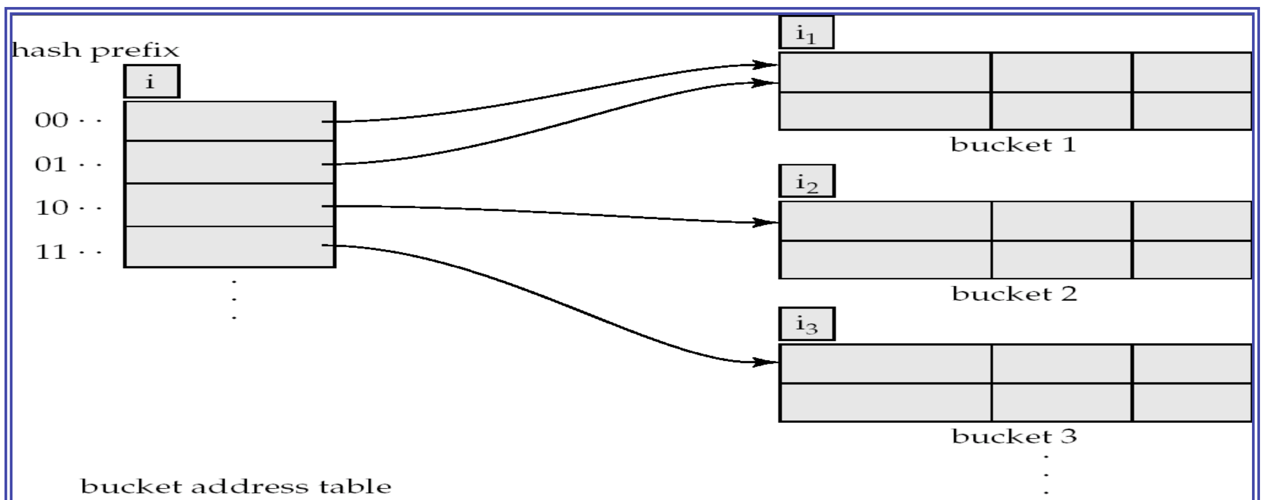
- i=3
  - 000, 001, 010, 011, 100, 101, 110, 111
- i=4
  - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

1	=	1	=	1
10	=	2+0	=	2
11	=	2+1	=	3
100	=	4+0+0	=	4
101	=	4+0+1	=	5
110	=	4+2+0	=	6
111	=	4+2+1	=	7
1000	=	8+0+0+0	=	8
1001	=	8+0+0+1	=	9
1010	=	8+0+2+0	=	10
1011	=	8+0+2+1	=	11
1100	=	8+4+0+0	=	12
1101	=	8+4+0+1	=	13
1110	=	8+4+2+0	=	14
1111	=	8+4+2+1	=	15
10000	=	16+0+0+0+0	=	16
10001	=	16+0+0+0+1	=	17
10010	=	16+0+0+2+0	=	18
10011	=	16+0+0+2+1	=	19
10100	=	16+0+4+0+0	=	20
10101	=	16+0+4+0+1	=	21
10110	=	16+0+4+2+0	=	22
10111	=	16+0+4+2+1	=	23
11000	=	16+8+0+0+0	=	24
11001	=	16+8+0+0+1	=	25
11010	=	16+8+0+2+0	=	26
11011	=	16+8+0+2+1	=	27
11100	=	16+8+4+0+0	=	28
11101	=	16+8+4+0+1	=	29
11110	=	16+8+4+2+0	=	30
11111	=	16+8+4+2+1	=	31

General Extendable Hash Structure



Initial Hash structure



In this structure $i = 2$, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

Use of Extendable Hash Structure

- Let the length of the prefix be i bits (write it on the top of the bucket-address-table)
- Each bucket j stores a value i_j (write it on the top of the bucket)
- All the entries in the bucket-address-table that point to the same bucket have the same hash values on the first i_j bits. The number of bucket-address-table entries that point to bucket j is:

$$2^{(i - i_j)}$$

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Queries

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

Insertion

- 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 (next slide)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion cont'd

- 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)
    - re-compute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)

Insertion cont'd

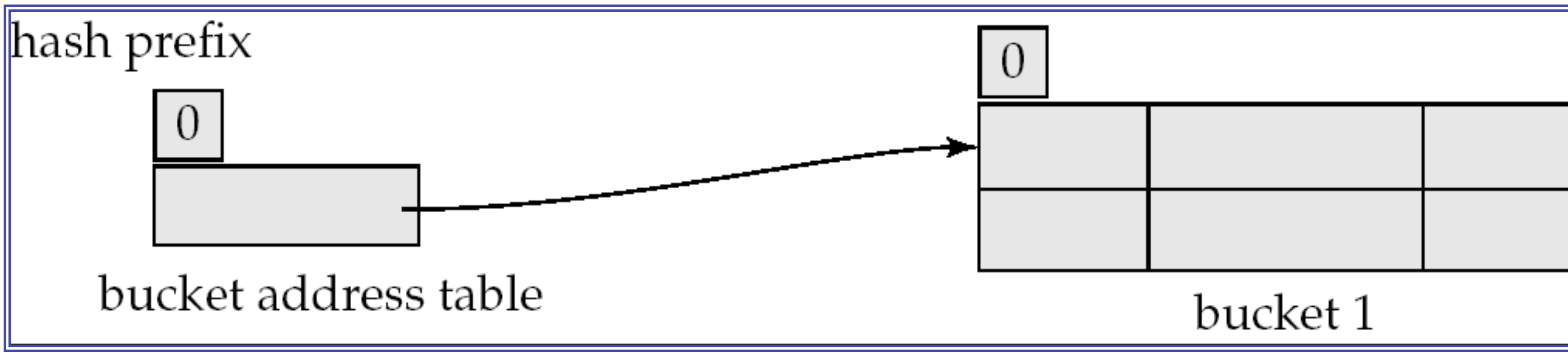
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table
    - replace each entry in the table by two entries that point to the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

Deletion

- 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

Example

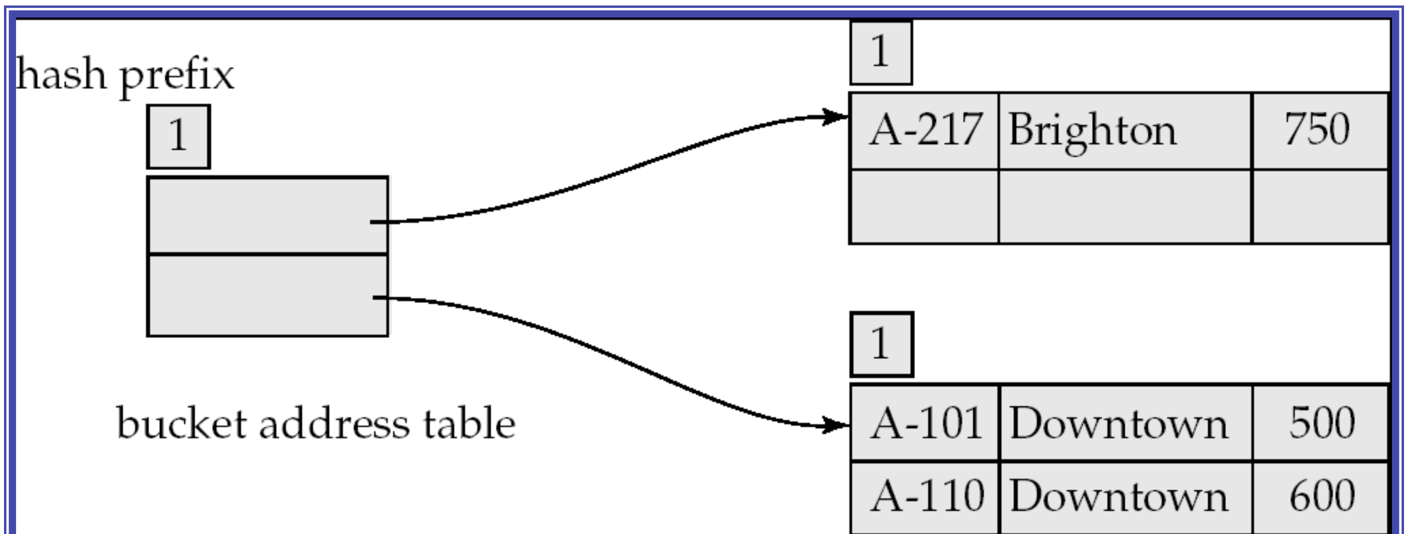
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



- Initial Hash structure (bucket size = 2)
- Each bucket can hold up to two records

Example cont'd

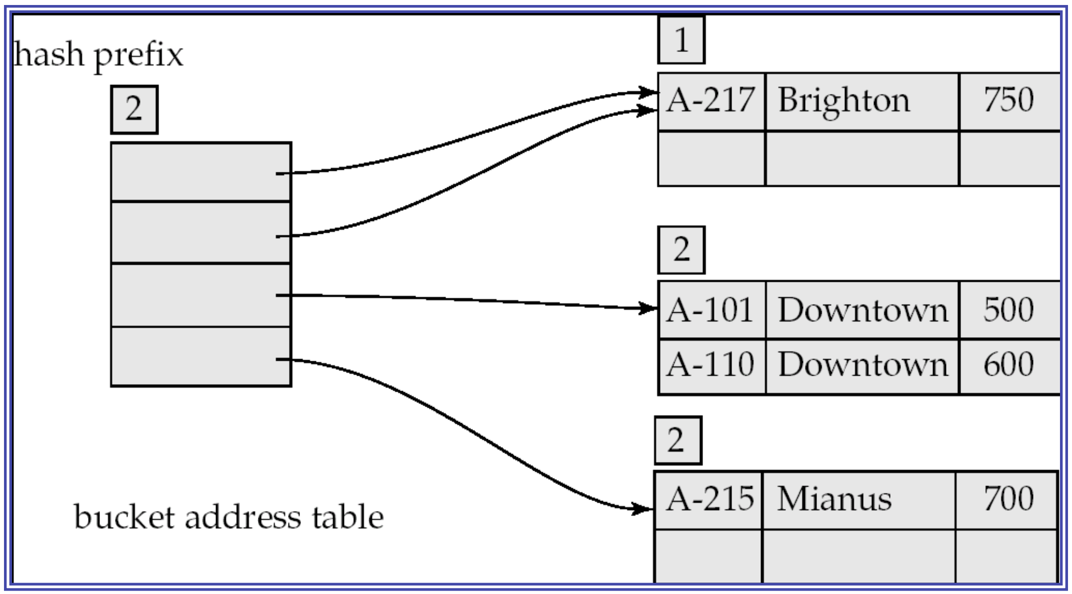
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Hash structure after insertion of one Brighton and two Downtown records

Example cont'd

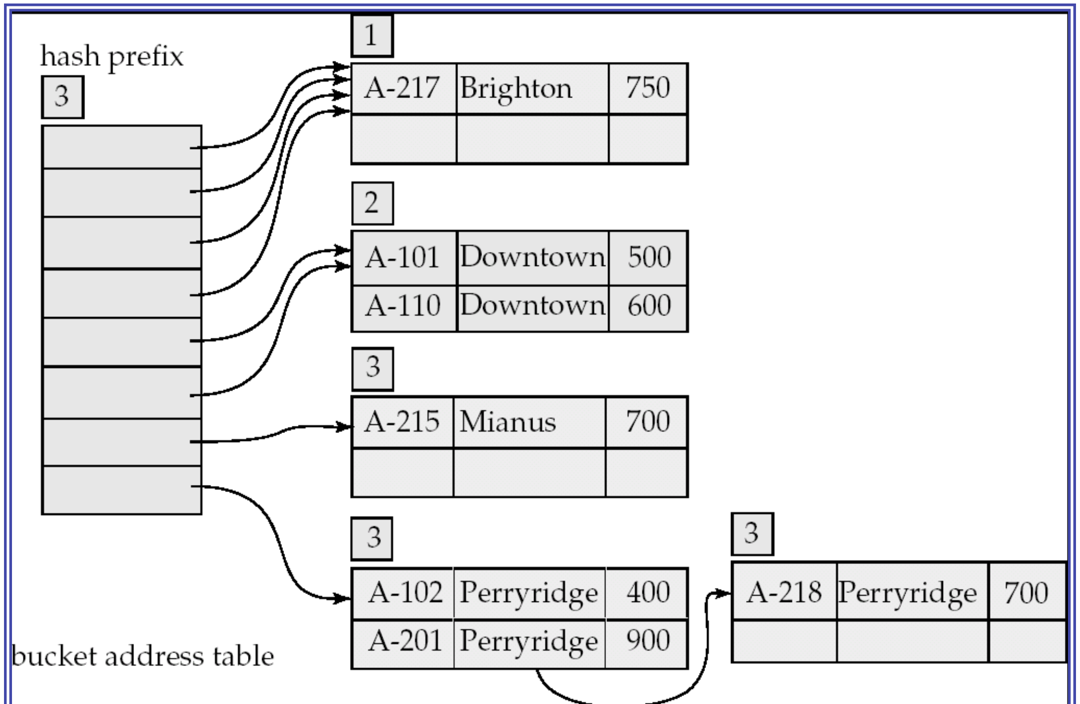
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Hash structure after insertion of Mianus record

Example cont'd

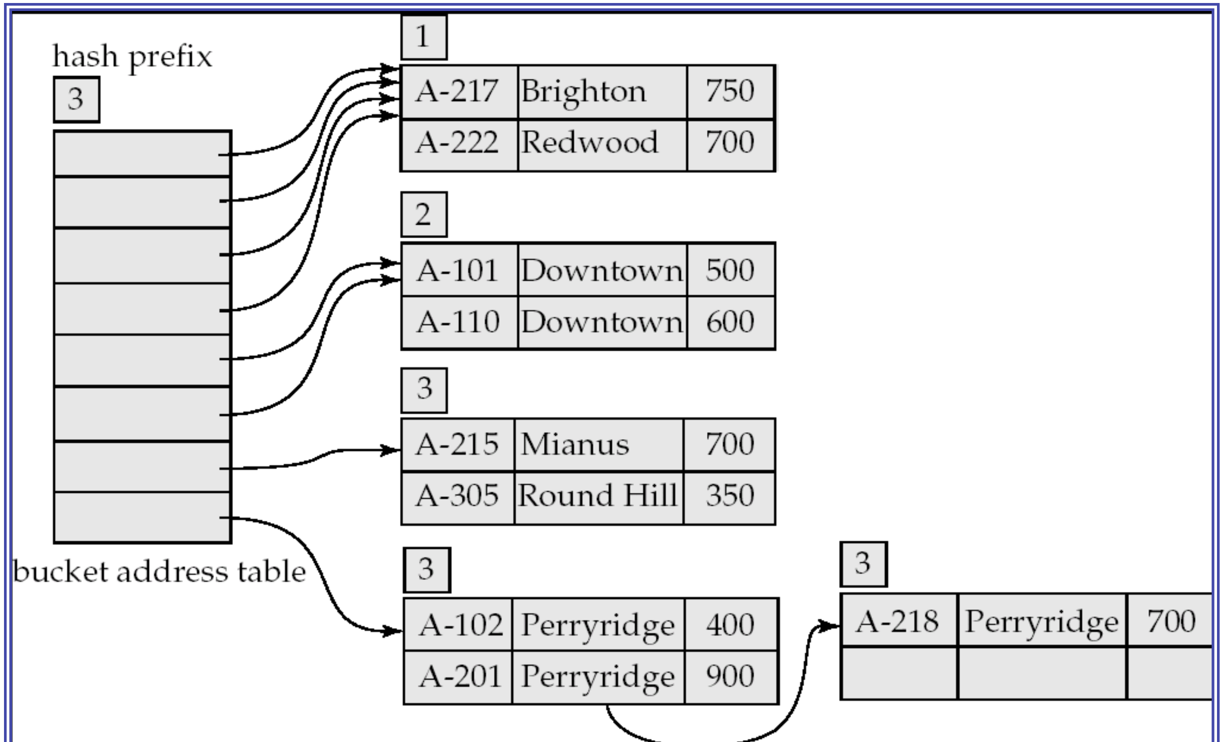
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Hash structure after insertion of three Perryridge records

Example cont'd

branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Hash structure after insertion of Redwood and Round Hill records

Errata

- In textbook 6 edition, Figure 11.33 on PP. 521, the number of the first bucket should be changed from 2 to 1 as there are four pointers point to it.

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 (not covered here)

Comparison of Ordered Indexing and Hashing

- File can be organised as
  - Ordered: index-sequential organisation or B+-tree
  - Hashing
  - Heap
- The choice depends on
  - Cost of periodic re-organisation
  - Relative frequency of insertions and deletions
  - Is it desirable to optimise average access time at the expense of worst-case access time?
  - Expected type of queries
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organisation, but not hash indices
  - SQLServer supports only B+-trees

Type of Queries and Indices

- For Queries of the form:
  - Hashing is generally better at retrieving records having a specified value of the key.

```
select A1, A2, … An
from r
where  Ai = c
```

- For Queries of the form:
  - If range queries are common, ordered indices are to be preferred

```
select A1, A2, … An
from r
where  Ai ≥c2  and Ai ≤ c1
```

Xi'an Jiaotong-Liverpool University 西交利物浦大学

End of Lecture

- Summary
  - Hash-based Indexing
    - Static Hashing
    - Dynamic Hashing
  - Comparison of Ordered Indexing and Hash-based Indexing

- Reading
- Textbook, chapter 11.6, 11.7, and 11.8