Database Development and Design (CPT201)

Lecture 3a: Indexing Techniques

Dr. Wei Wang

Department of Computing

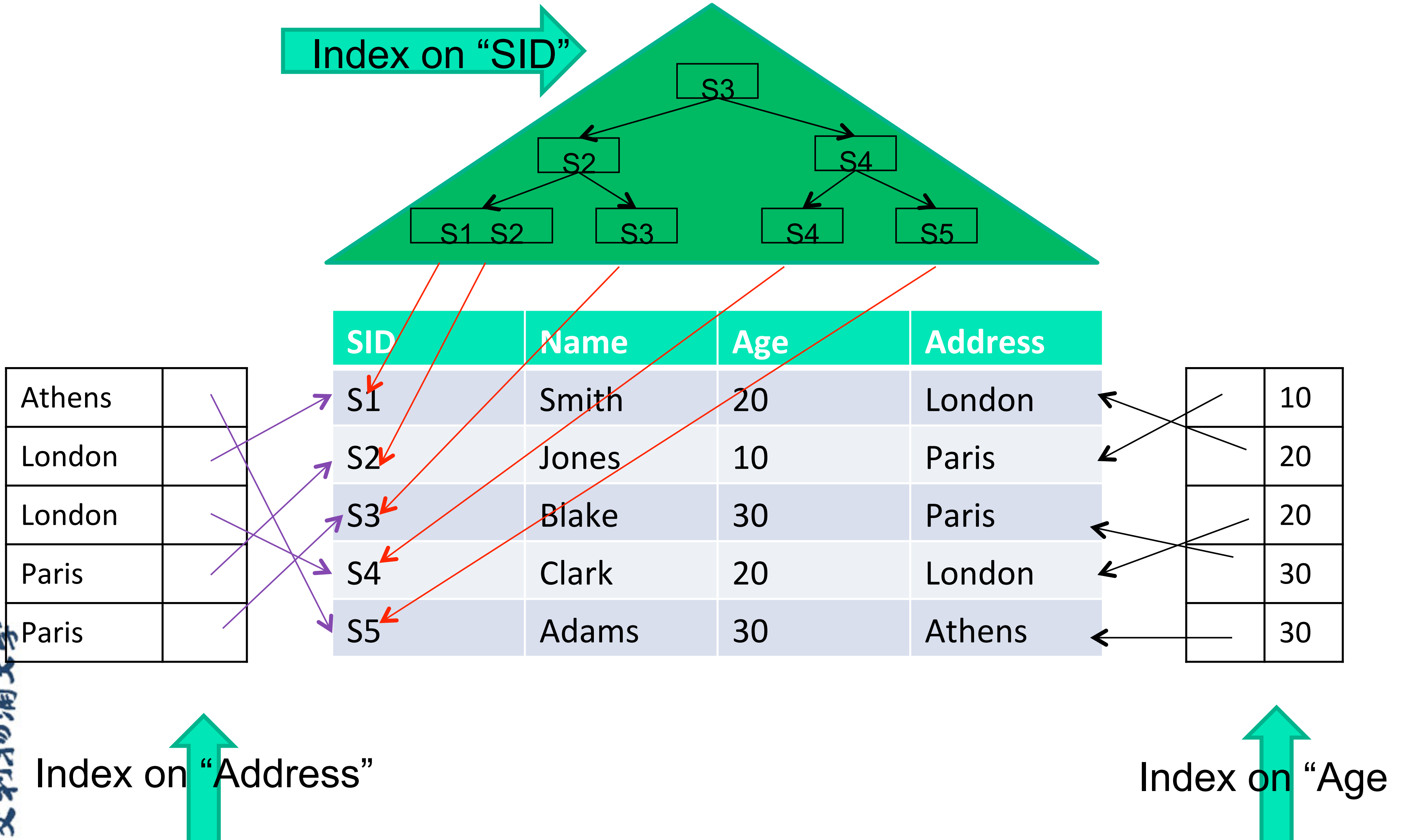
Learning Outcomes

- The Structure of Index
- Ordered Index
- Primary Index vs. Secondary Index
- Dense Index vs. Sparse Index
- Multilevel index

Motivation: Search Records

- To scatter records of a relation to different blocks is not efficient.
  - SELECT * FROM C;
    - problem: search all the blocks on the disk
    - solution: keep records of a certain relation on adjacent cylinders
  - SELECT * FROM C WHERE age=10;
    - problem: search all the blocks and check the condition on the disk
    - solution: create indices on some attributes
- Indexing mechanisms used to speed up access to the desired data.

Index



Index on “SID”
S3
S2
S4
S1 S2
S3
S4
S5
SID
Name
Age
Address
S1
Smith
20
London
S2
Jones
10
Paris
S3
Blake
30
Paris
S4
Clark
20
London
S5
Adams
30
Athens
Athens
London
London
Paris
Paris
10
20
20
30
30
Index on “Address”
Index on “Age

The Structure of Index

- Data file: collection of blocks holding records on disk
- Index file: an data structure allowing the DBMS to find particular records in a data file more efficiently.
  - An index file consists of records (called index entries) of the form:

search-key	pointer

- Search Key: one or set of attributes used to look up records in a file.
- Relationship: a search key K in the index file is associated with a pointer to a data-file record that has search key K.

Index Evaluation Metrics

- Access types (supported)
  - records with a specified value in the attribute or
  - records with an attribute value falling in a specified range.
- Access time
- Insertion time
- Deletion time
- Space overhead

Indexing Techniques

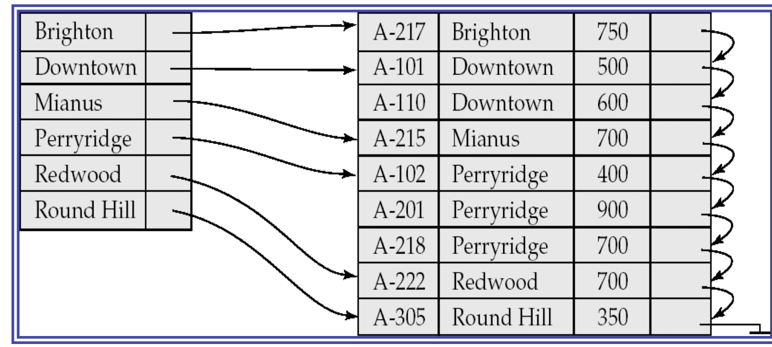
- Depending on the organisation of index file, an index can be:
  - an ordered Index where index entries are sorted on the search key value.
  - a hashing Index where hashing technique is employed to organise index entries.

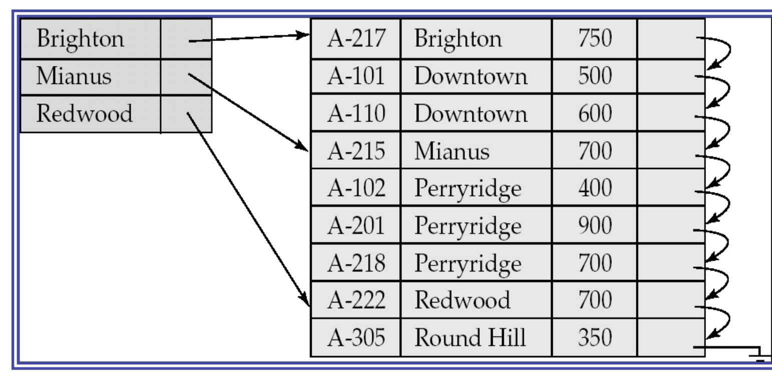
Ordered Indices

- Ordered index: index entries in the index are sorted on the search key value.
- An ordered index can be:
  - Dense index: index record appears for every search-key value in the file.
  - Sparse Index: contains index records for only some search-key values.

Dense Index vs. Sparse Index

- Index size
  - Sparse index is smaller
- Requirement on data file
  - The data file must be sequential file
- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists.
- Update
  - Sparse index requires less space and maintenance for insertion and deletion.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.





Ordered Indices cont'd

- An ordered index can also be:
  - Primary index: an index whose search key specifies the sequential order of the file.
    - Also called clustering index. The search key of a primary index is usually but not necessarily the primary key.
    - Can be sparse
  - Secondary index: an index whose search key specifies an order different from the sequential order of the file.
    - Also called non-clustering index.
    - Can not be sparse
- Index-sequential file: ordered sequential file with a primary index.

Dense Index Files

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Dense Index Files cont'd

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

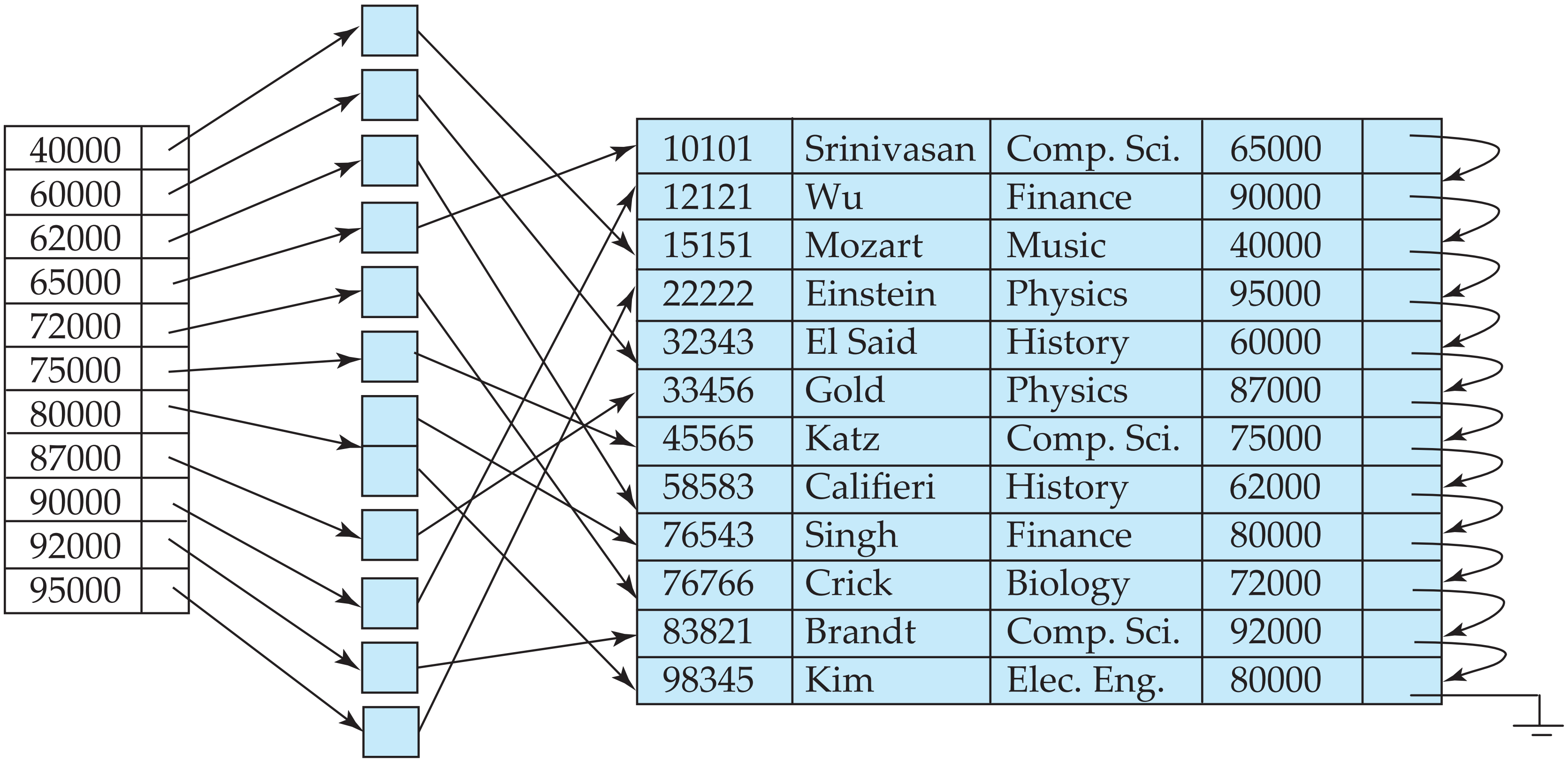
76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein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

Sparse Index Files

10101	
32343	
76766	

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	

Secondary Index



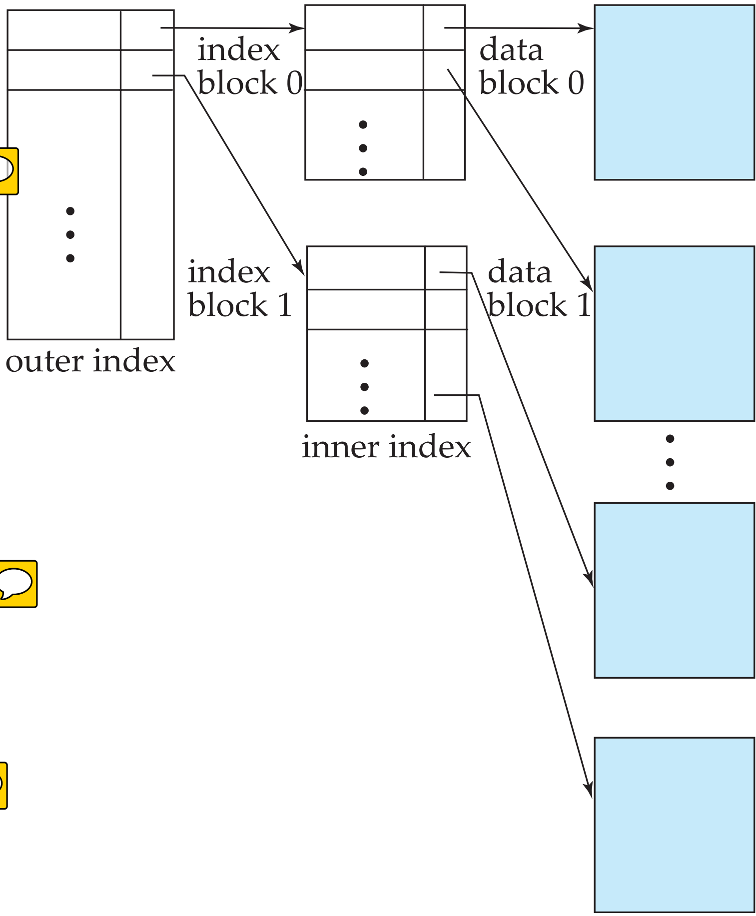
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- But updating indices imposes overhead on database modification - when a file is modified, every index on the file must be updated
- Sequential scan using primary index is efficient
- But a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds; versus about 100 nanoseconds for memory access

Multilevel Index (Index on index)

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Index Definition in SQL

- Create an index
  - create index <index-name> on <relation-name> (<attribute-list>)
  - E.g.: create index b-index on branch(branch_name)
- To drop an index
  - drop index <index-name>
- Most database systems allow specification of type of index.

End of Lecture

- Summary
  - The Structure of Index
  - Ordered Indices
  - Primary index vs. Secondary index
  - Dense index vs. sparse index
  - Multilevel index
- Reading
  - Database System Concepts, 6th edition, chapter 11.1, 11.2
  - Database System Concepts, 7th edition, chapter 14.1, 14.2

Database Development and Design (CPT201)

Lecture 3b:
B+ Tree Index

Dr. Wei Wang
Department of Computing

Learning Outcomes

- B+-Tree Index
  - Queries
  - update

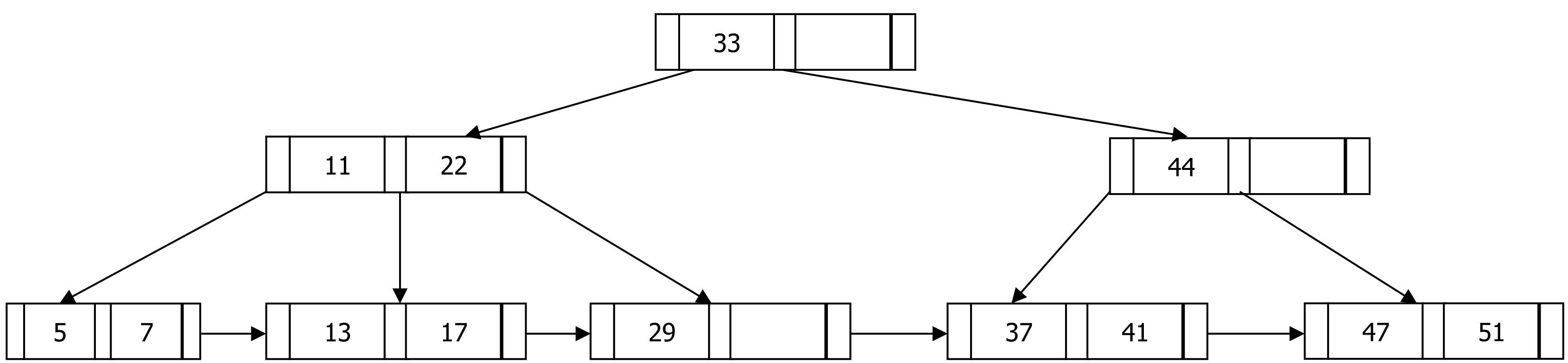
B+-Tree Index

- B+-Tree is "short" and "Fat"
  - Disk-based: usually one node per block; large fan-out
  - Balanced (more or less): good performance guarantee.
- In a B+-Tree,
  - n (or sometimes N) is the number of pointers in a node; pointers: P1, P2, …Pn
  - Search keys: K1 < K2 < K3 < . . . < Kn–1
  - All paths (from root to leaf) have same length
  - Root must have at least two children
  - In each non-leaf node (inner node), more than 'half' (≥⌈n/2⌉) pointers must be used
  - Each leaf node must contain at least ⌈(n-1)/2)⌉ keys

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

Example

- An Example B+-Tree with n = 3
  - All paths have same length.
  - Root has (at least) two children
  - In each non-leaf node (inter node), more than half ($\geq\lceil 3/2\rceil = 2$) pointers are used
  - Each leaf node contains at least $\lceil (3-1)/2) \rceil = 1$ key



Queries on B$^+$-Trees

- Find record with search-key value V.
  - 1. C=root
  - 2. While C is not a leaf node
    - 2.1. Let i be least value such that V ≤ Ki.
    - 2.2. If no such exists, set C = last non-null pointer in C
    - 2.3. Else { if (V= Ki) Set C = Pi +1 else set C = Pi}
  - 3. Let i be least value such that Ki = V
  - 4. If there is such a value i, follow pointer Pi to the desired record.
  - 5. Else no record with search-key value V exists.

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.
- If there are K search-key values in the file
  - The B^+-tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
  - 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.

Observations about B+-trees cont'd

- Searching can be conducted efficiently.
  - a node is generally the same size as a disk block, typically 4 kilobytes
  - 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.
- Insertion and deletion to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Updates on B^+-Trees: Insertion

- 1. Find the leaf node in which the search-key value would appear
- 2. If the search-key value is already present in the leaf node
  - 2.1. Add record to the file
  - 2.2. If necessary add a pointer to the bucket.
- 3. If the search-key value is not present, then
  - 3.1. add the record to the main file (and create a bucket if necessary)
  - 3.2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - 3.3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

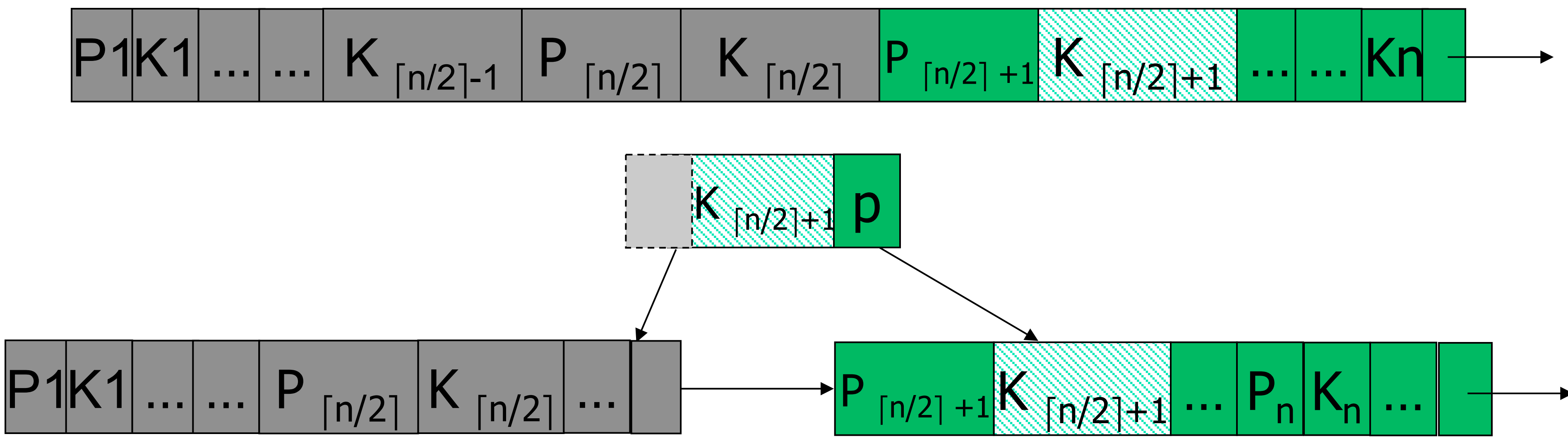
Updates on B$^+$-Trees: Insertion cont'd

- Splitting a leaf node:
  - take the (search-key value, pointer) pairs and the one being inserted) in an in-memory area M in sorted order. Assume there are n search key values in total.
  - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and propagate the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split, increasing the height of the tree by 1.

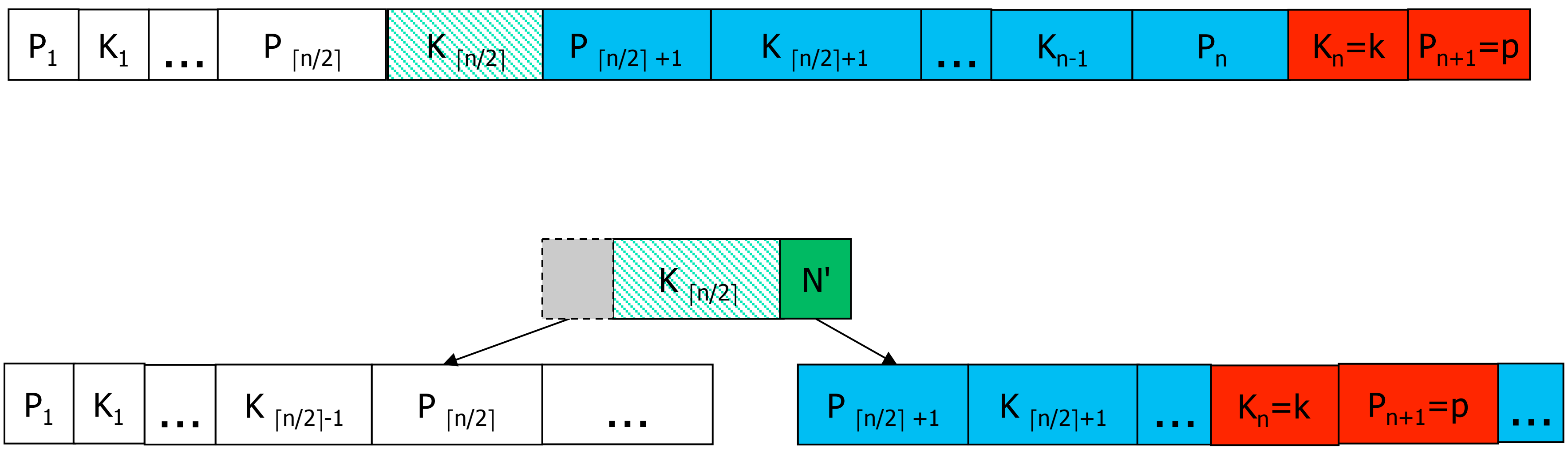
Updates on B^+-Trees: Insertion cont'd

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M in sorted order
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert ($K_{\lceil n/2 \rceil}$, N') into parent N

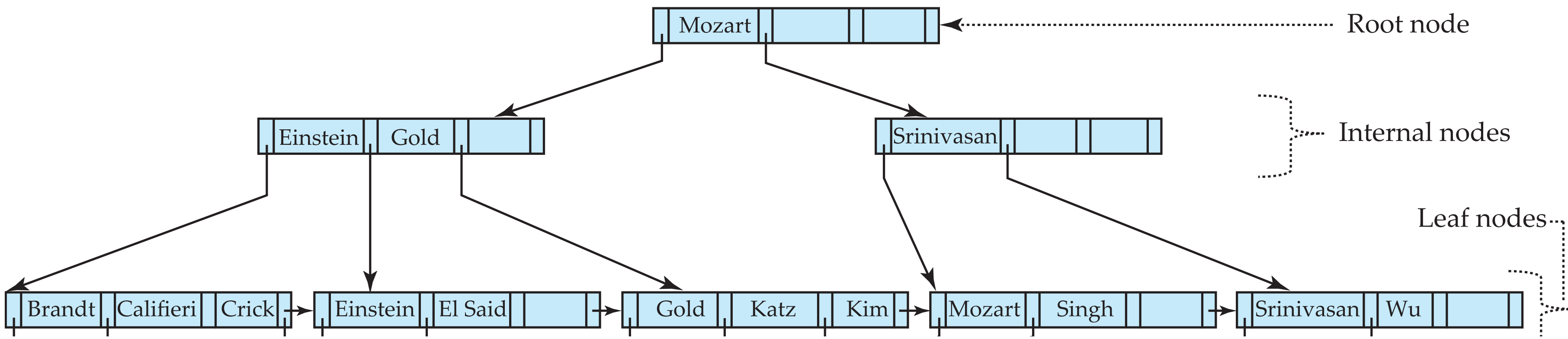
Splitting a Leaf Node

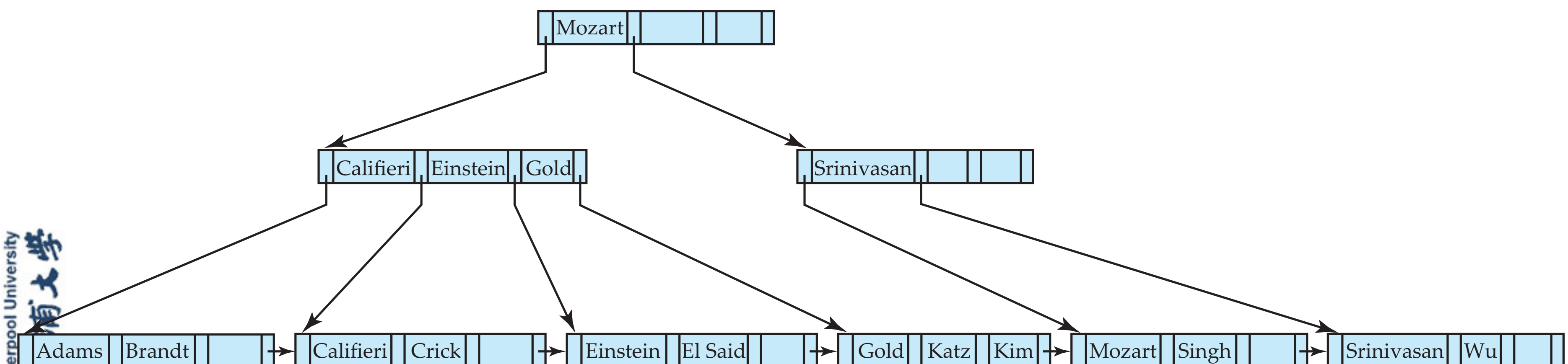


Splitting a Non-leaf Node



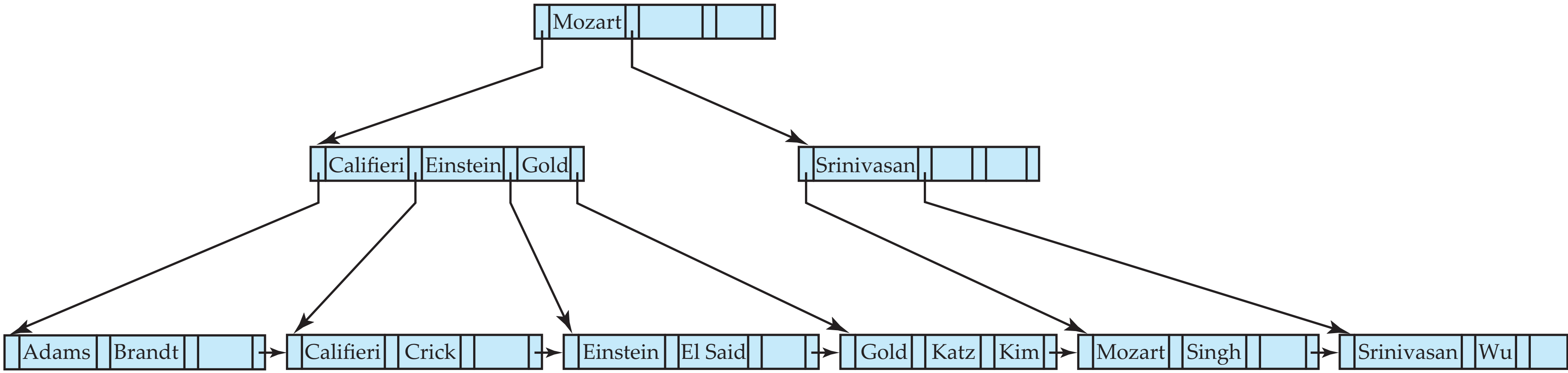
Insertion Example





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

Insertion Example cont'd



Question:

What will happen after insertion of "Lamport"?

Read pseudocode in textbook!

Exercise

- Construct a B+ tree for the following set of key values for n=3.
  - (2, 3, 5, 7, 11, 13, 17)

Updates on B$^+$-Trees: Deletion

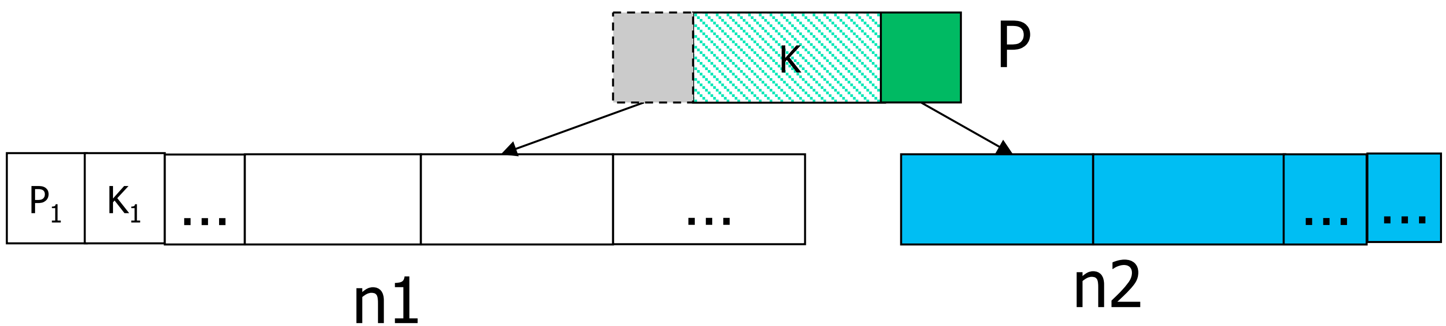
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then merge siblings:
  - Insert all the search-key values in the two nodes into a single node, and delete the other node.
  - If it is a non-leaf node, copy the value from the parent (between the two nodes) into the merged node
  - Delete the the value from the parent (between the two nodes). (Change may propagate to upper levels.)

Updates on B+-Trees: Deletion cont'd

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then redistribute pointers:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries; update the corresponding search-key value in the parent of the node.
  - If leaf node: take a proper value from sibling (value removed from sibling) and insert it to the underfull node; update the value in parent.
  - If non-leaf node: insert the value at (and remove from) parent to the underfull node, remove the value from sibling and update the parent.
  - Read pseudocode in textbook!
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

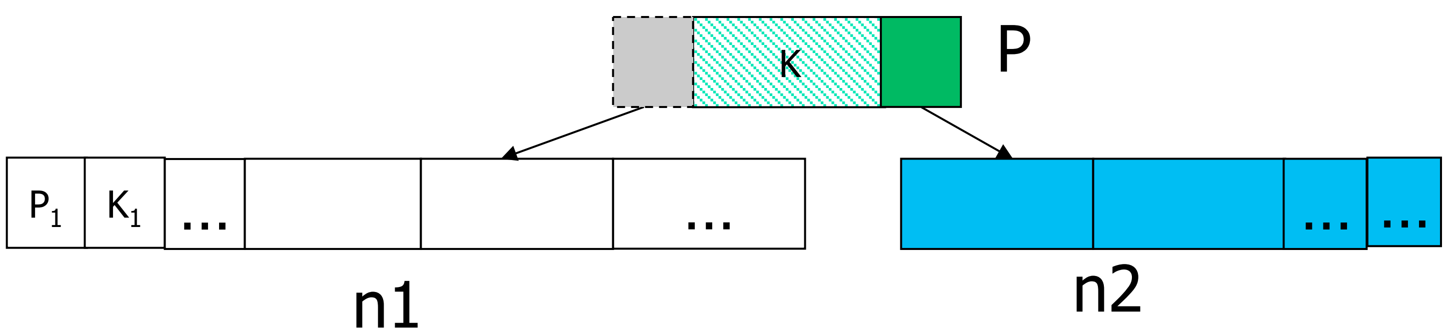
Merge Siblings – at Leaf Node

- Merge siblings n1 and n2
- Delete K (and the appropriate pointer) from parent P



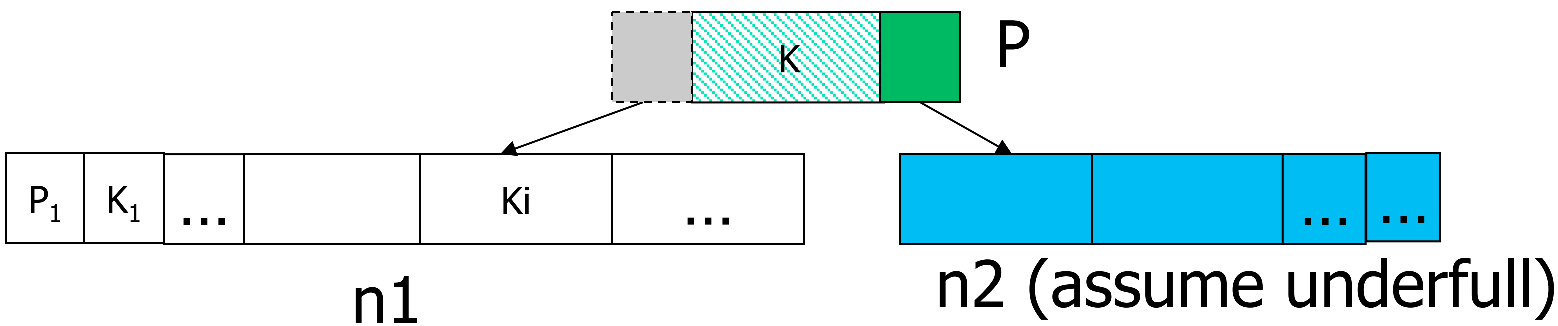
Merge Siblings – at non-Leaf Node

- Merge siblings n1 and n2 and K
- Delete K (and the appropriate pointer) from parent P



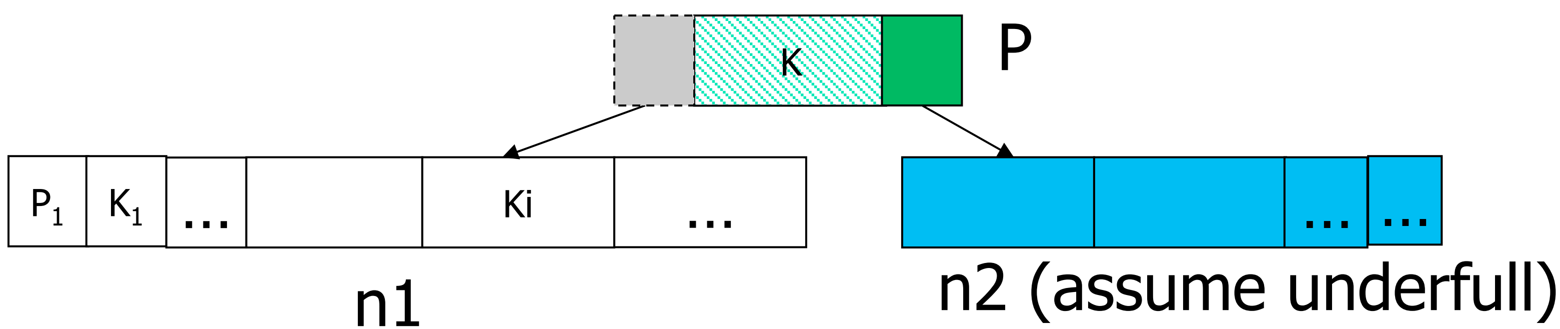
Redistribute Pointers – at Leaf Node

- Copy Ki from n1 and add it to n2
- Delete Ki from n1
- Replace the old value K in parent P with Ki

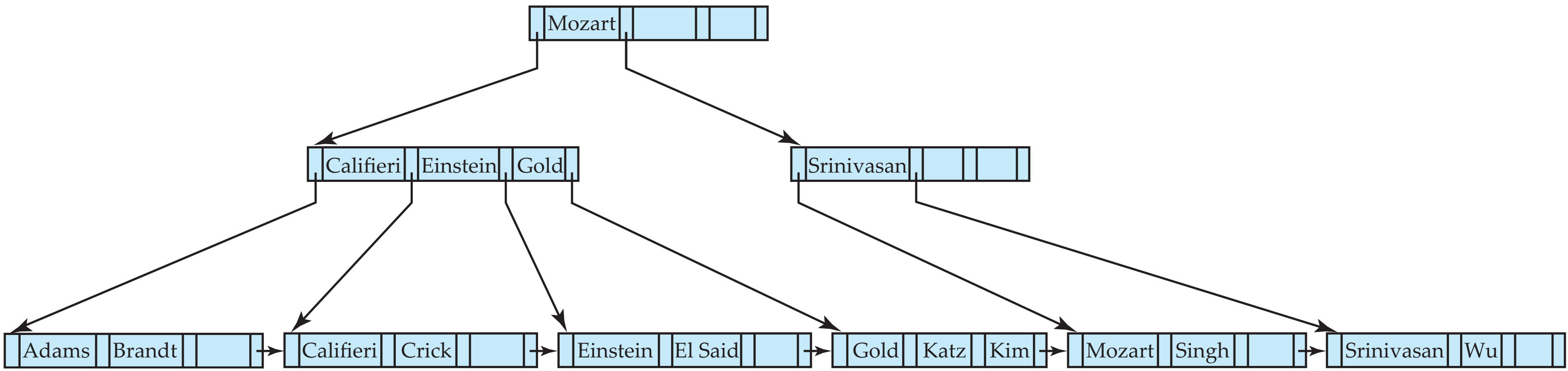


Redistribute Pointers – at non-Leaf Node

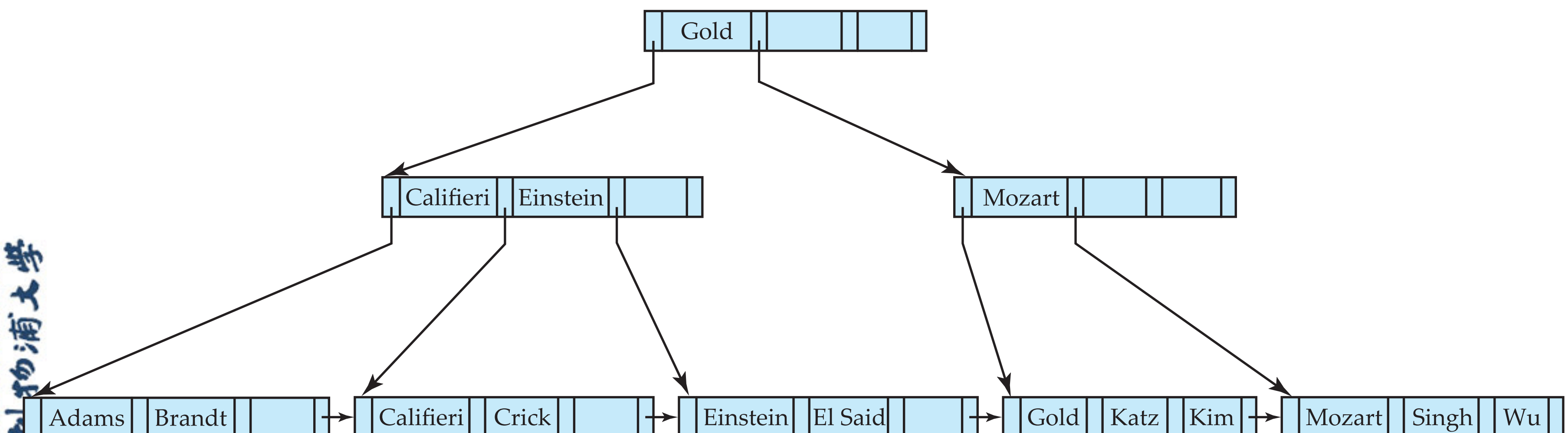
- Copy K from parent P and add it to n2
- Replace the old value K in parent P with Ki from n1
- Delete Ki from n1



Deletion Example

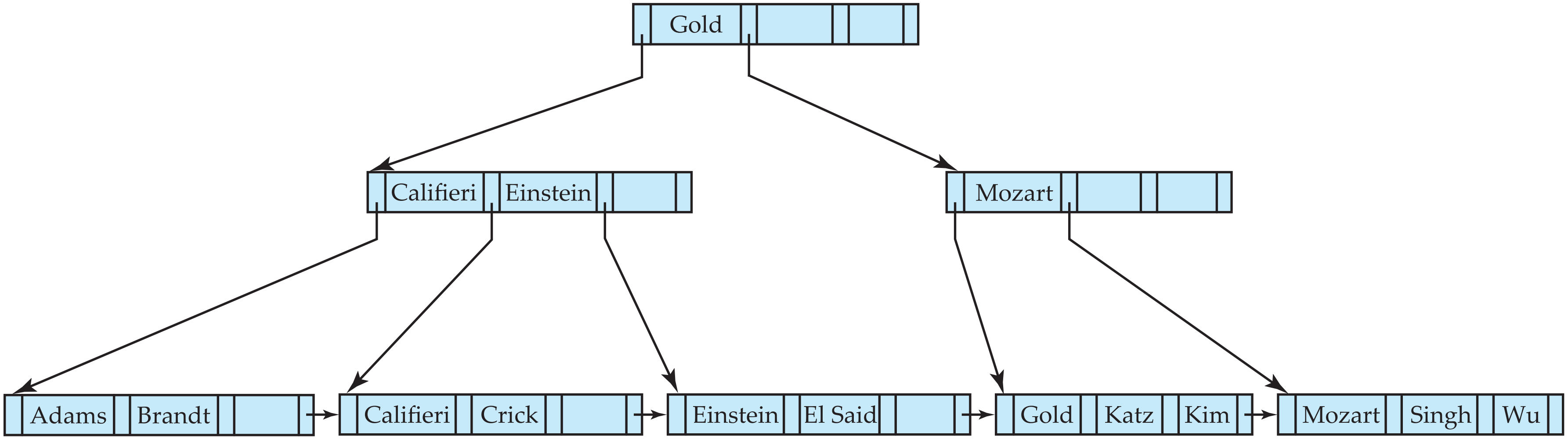


Before and after deleting "Srinivasan"

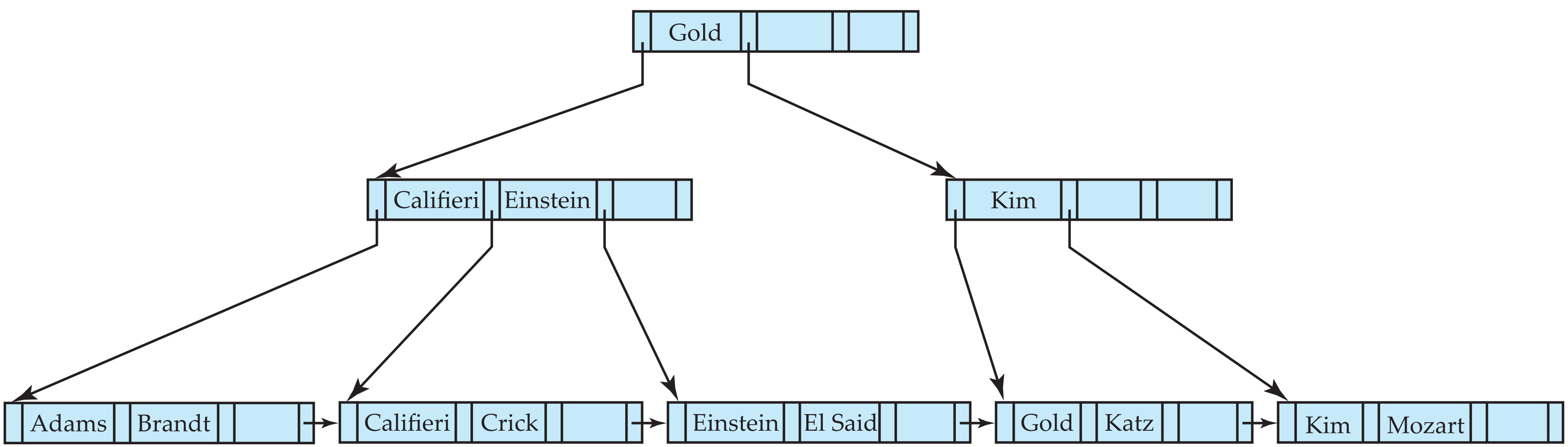


- Deleting "Srinivasan" causes merging of under-full leaves

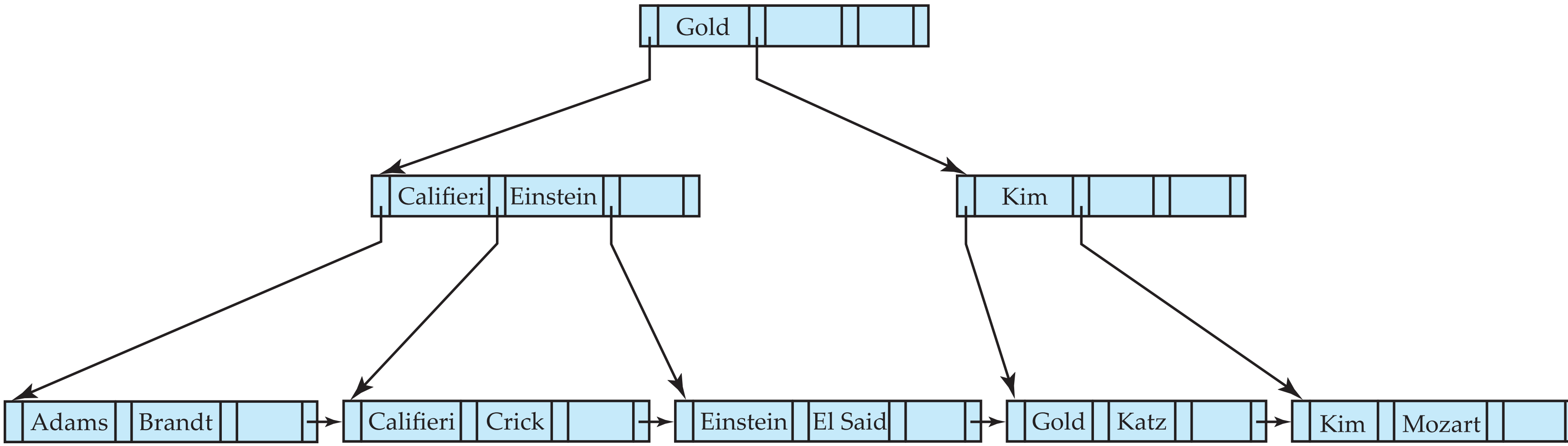
Deletion Example cont'd



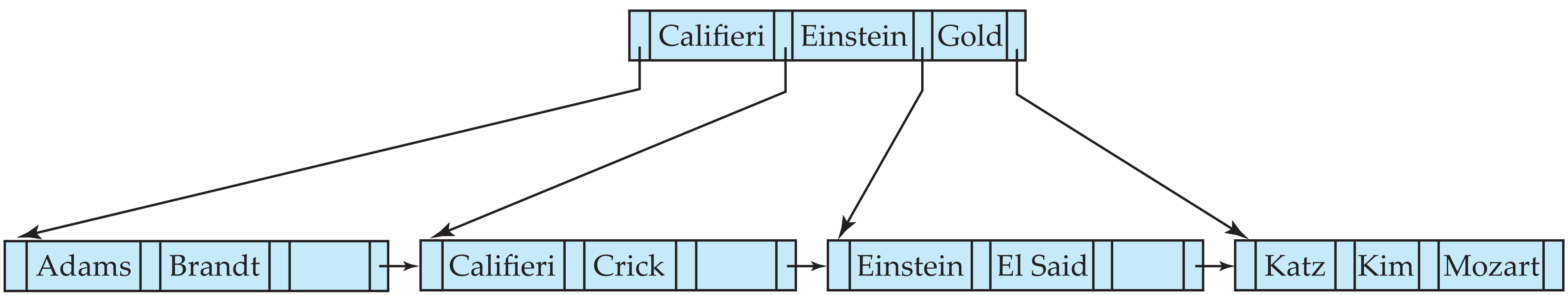
Before and after deleting "Singh and Wu"



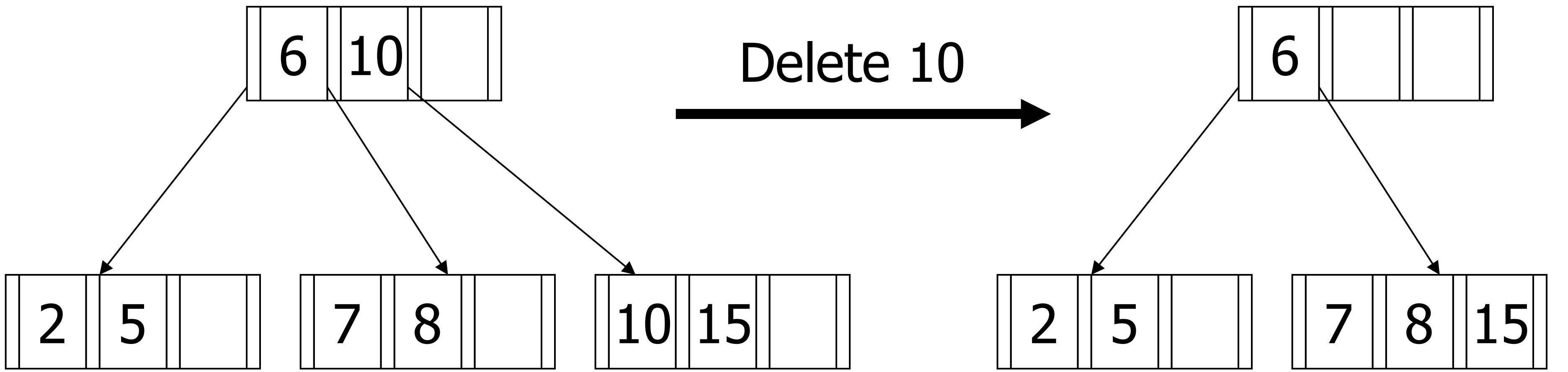
Deletion Example cont'd



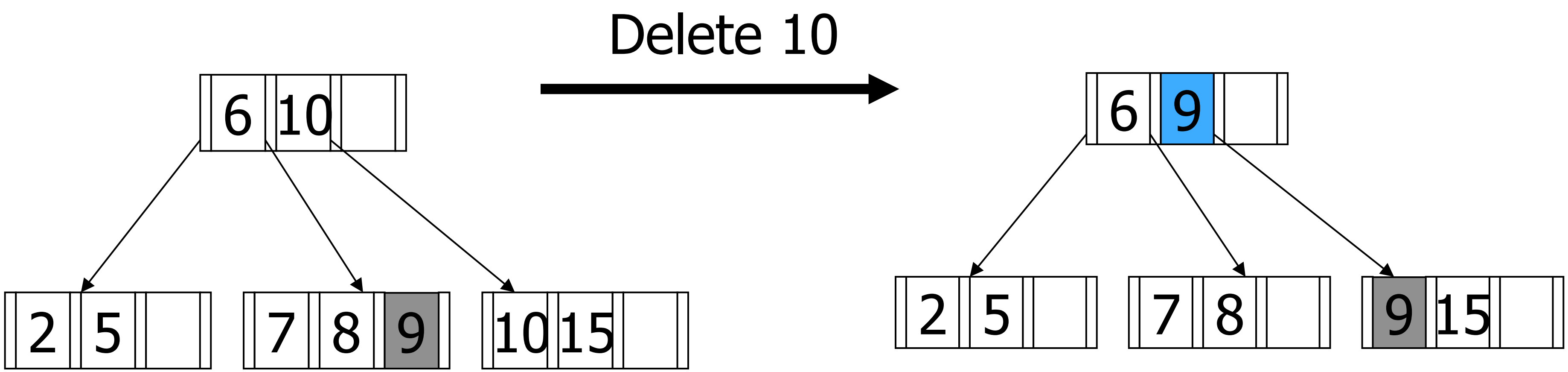
Before and after deleting "Gold"



More Example



Another Example



20/9/14

26

End of Lecture

- Summary
  - B+-Tree Index Files
    - lookup
    - Insertion
    - Deletion
- Reading
  - Database System Concepts, 6^{th} edition, chapter 11.1, 11.2, 11.3
  - Database System Concepts, 7^{th} edition, chapter 14.1, 14.2, 14.3

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

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

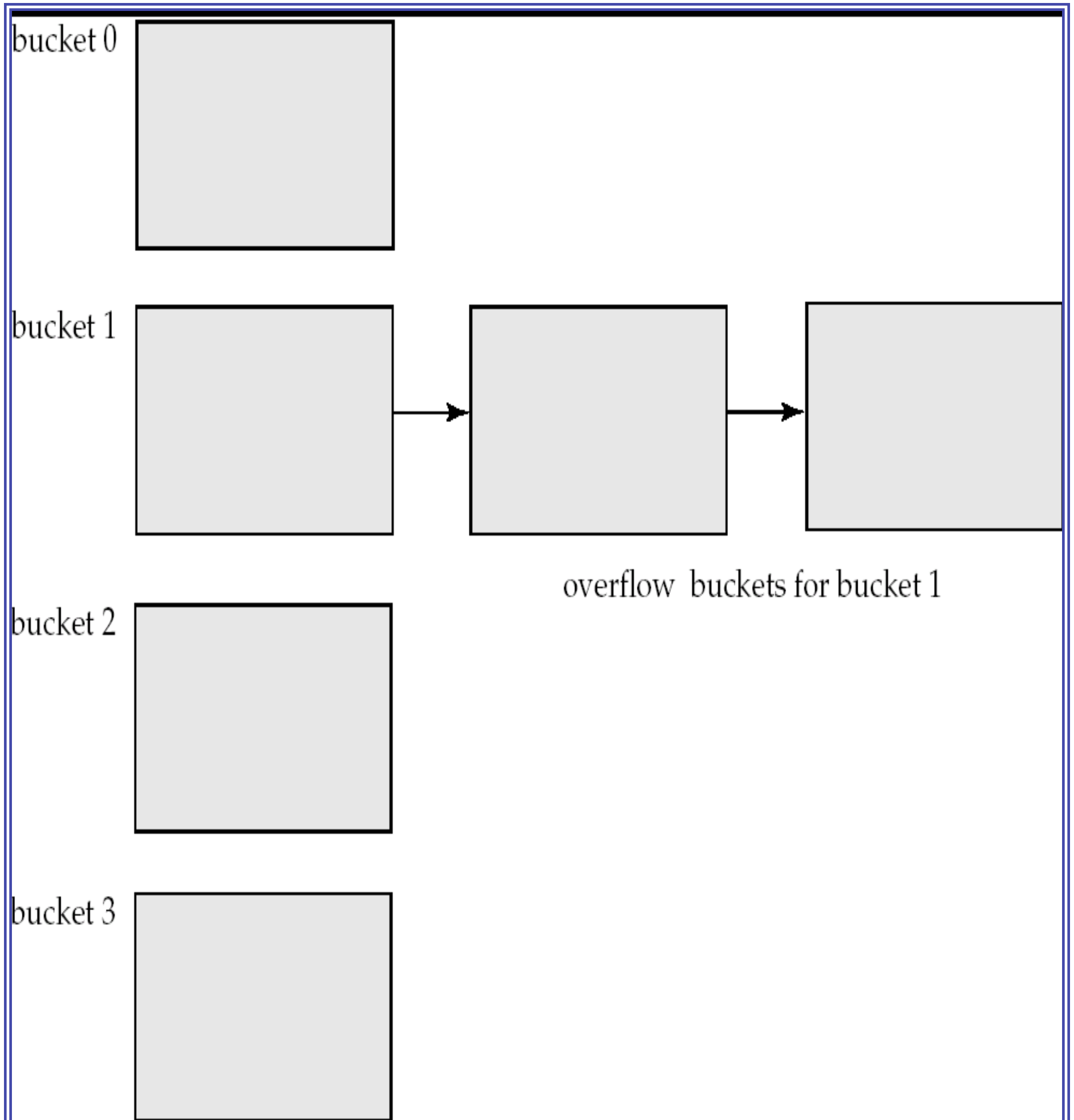
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.
- Assume that
  - There 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

Handling of Bucket Overflows

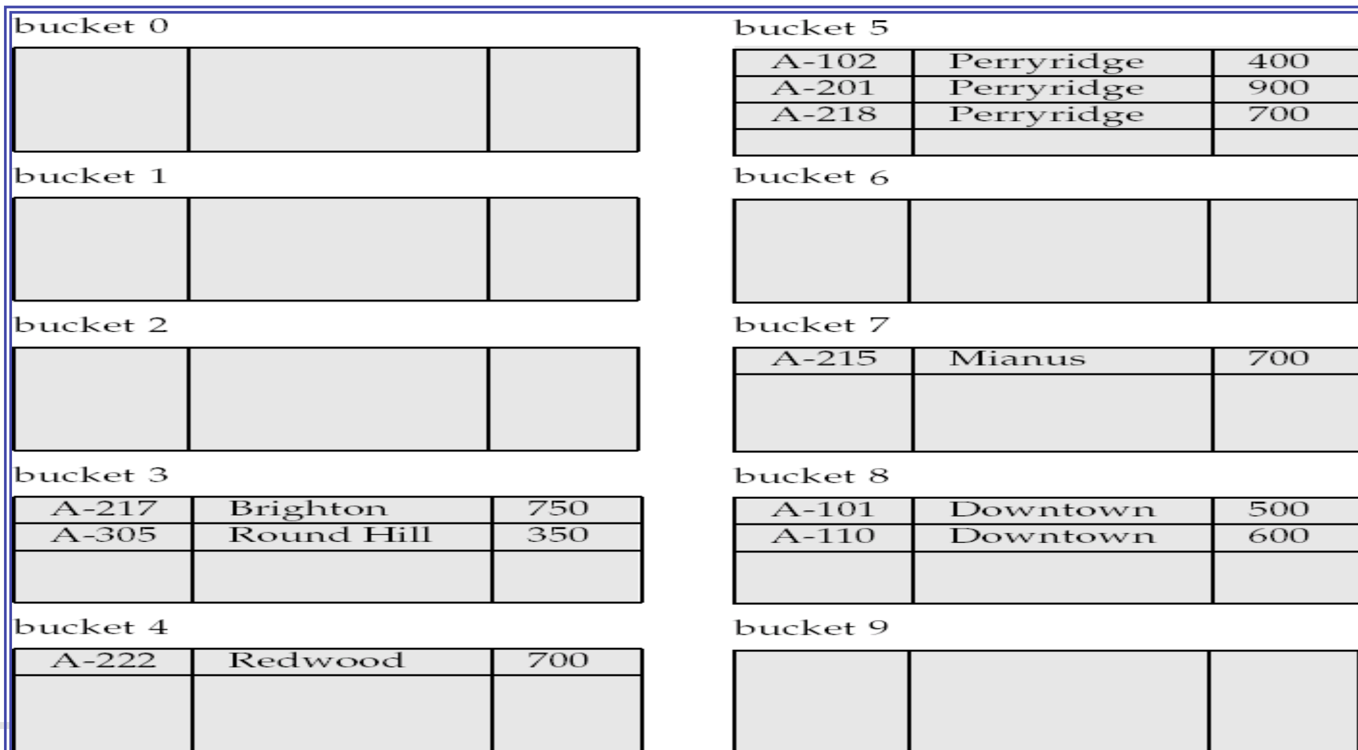
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow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

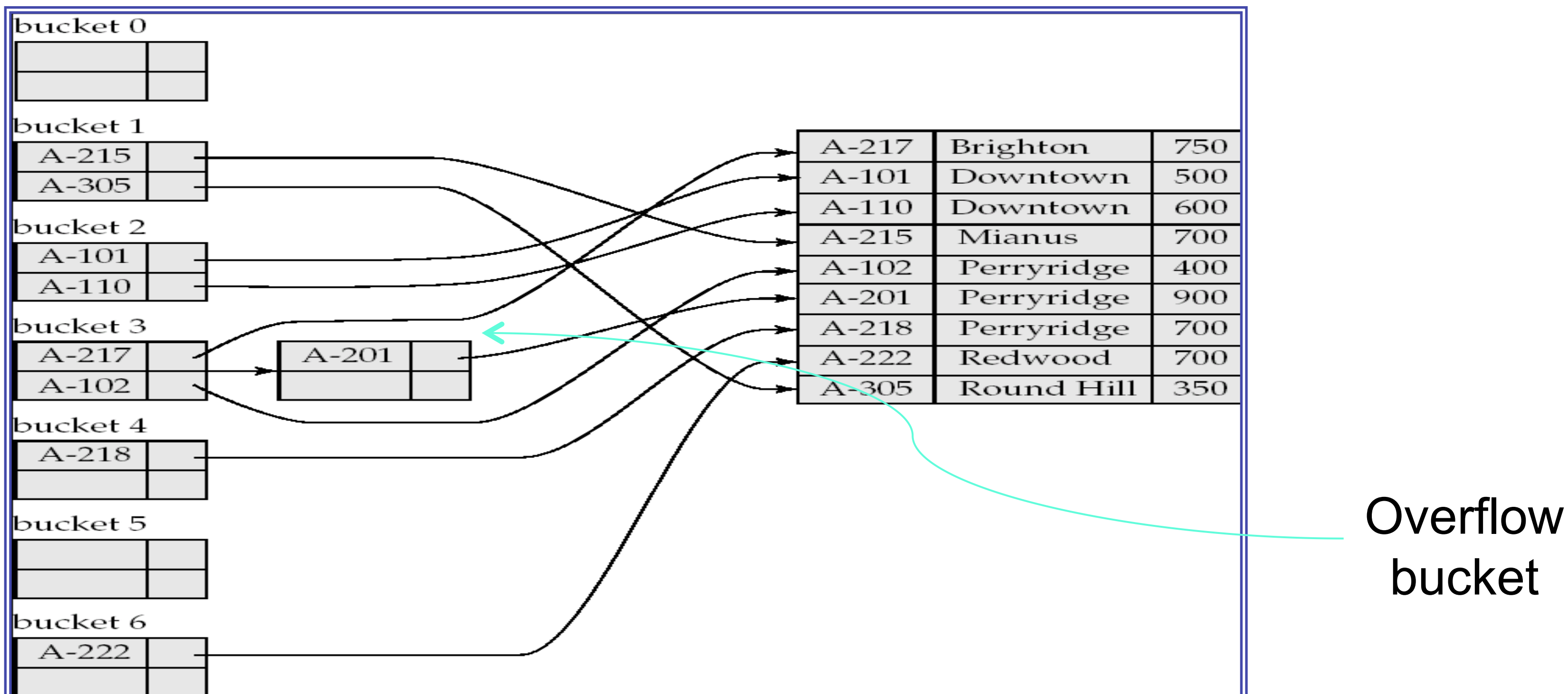


Hash Indices

- 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.

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

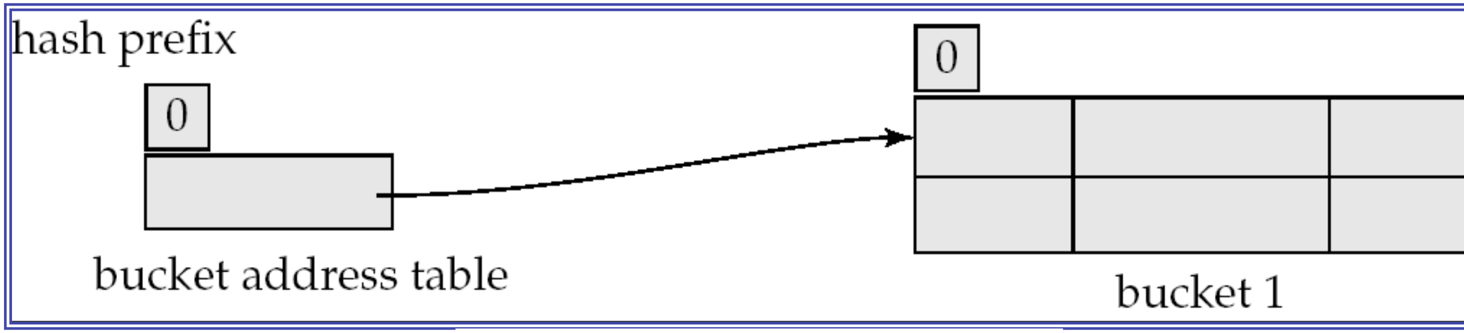
- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers, e.g. b = 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be i bits, $0 \leq i \leq 32$.
    - 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 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

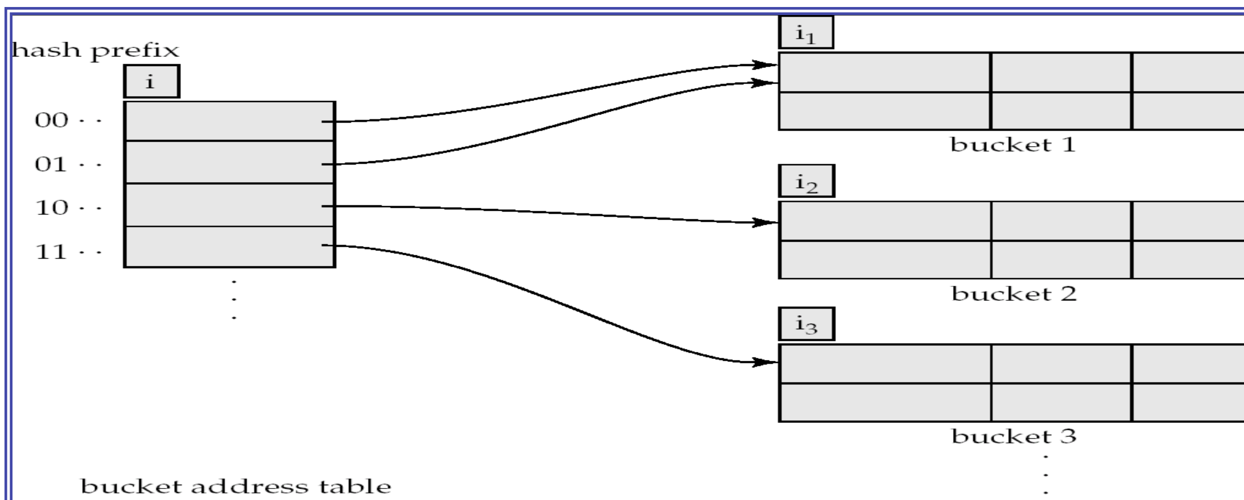
- i=3
  - 000, 001, 010, 011, 100, 101, 110, 111
- i=4
  - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

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:

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

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

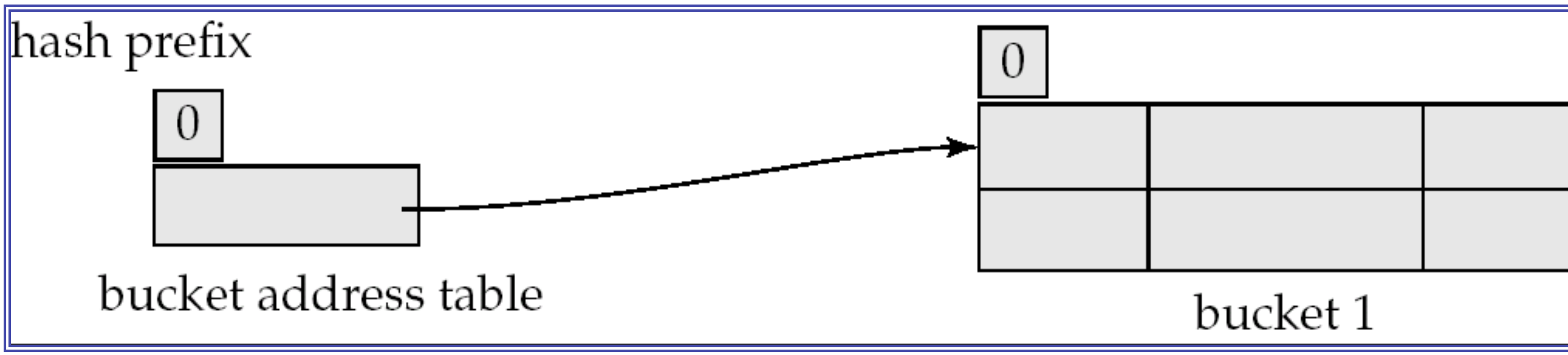
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table
    - replace each entry in the table by two entries that point to the 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

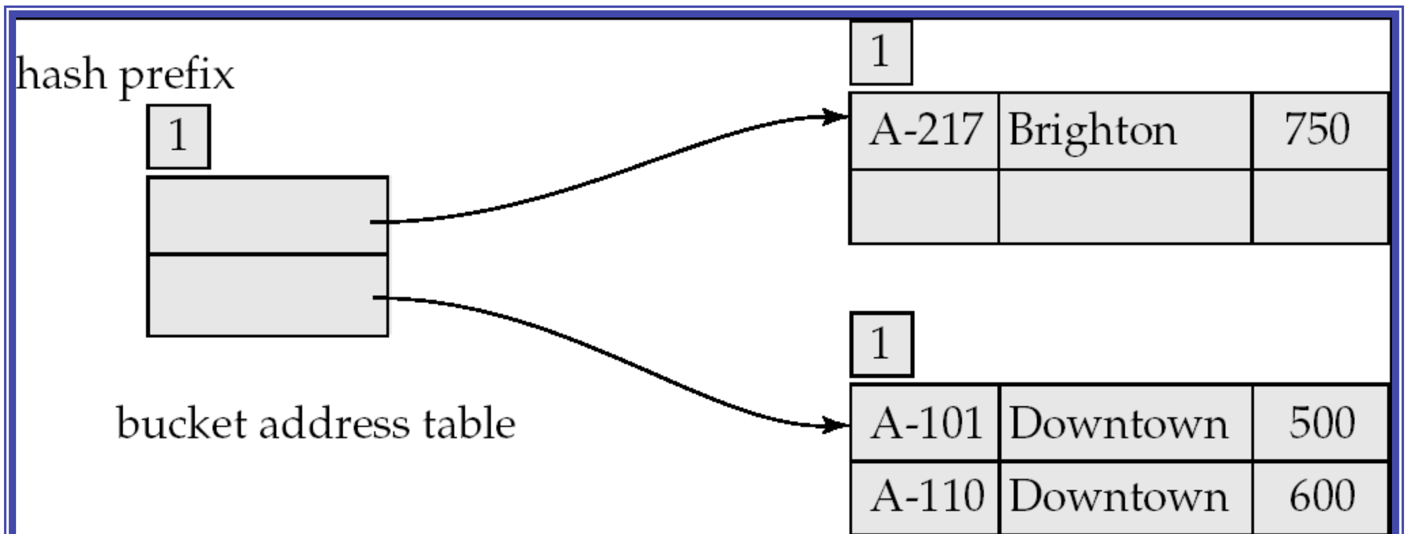
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



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

Example cont'd

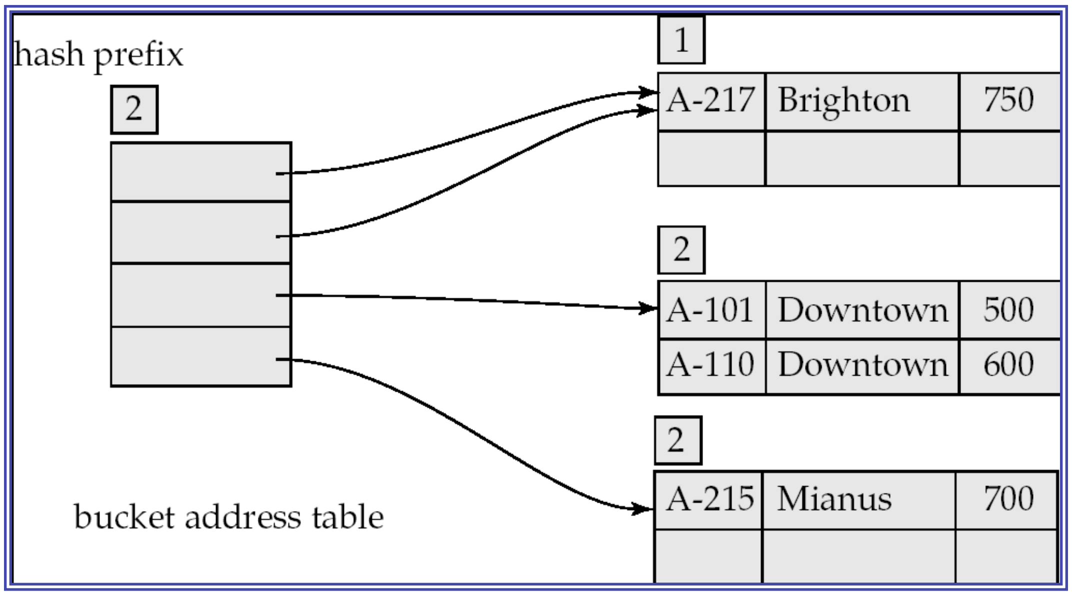
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



Hash structure after insertion of one Brighton and two Downtown records

Example cont'd

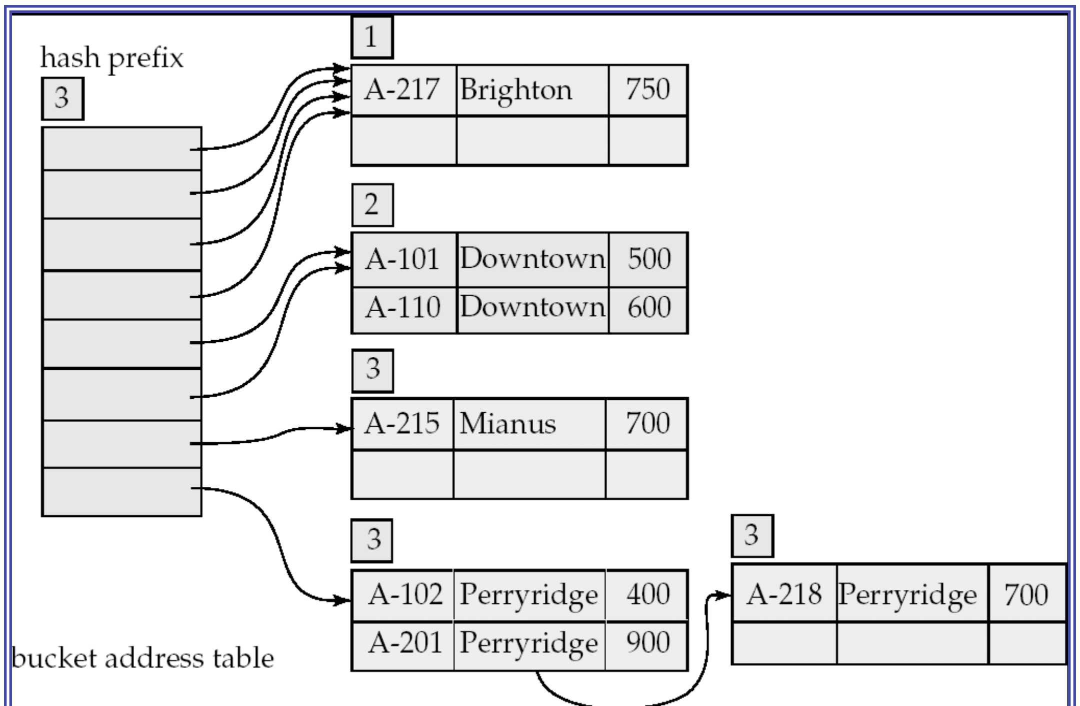
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



Hash structure after insertion of Mianus record

Example cont'd

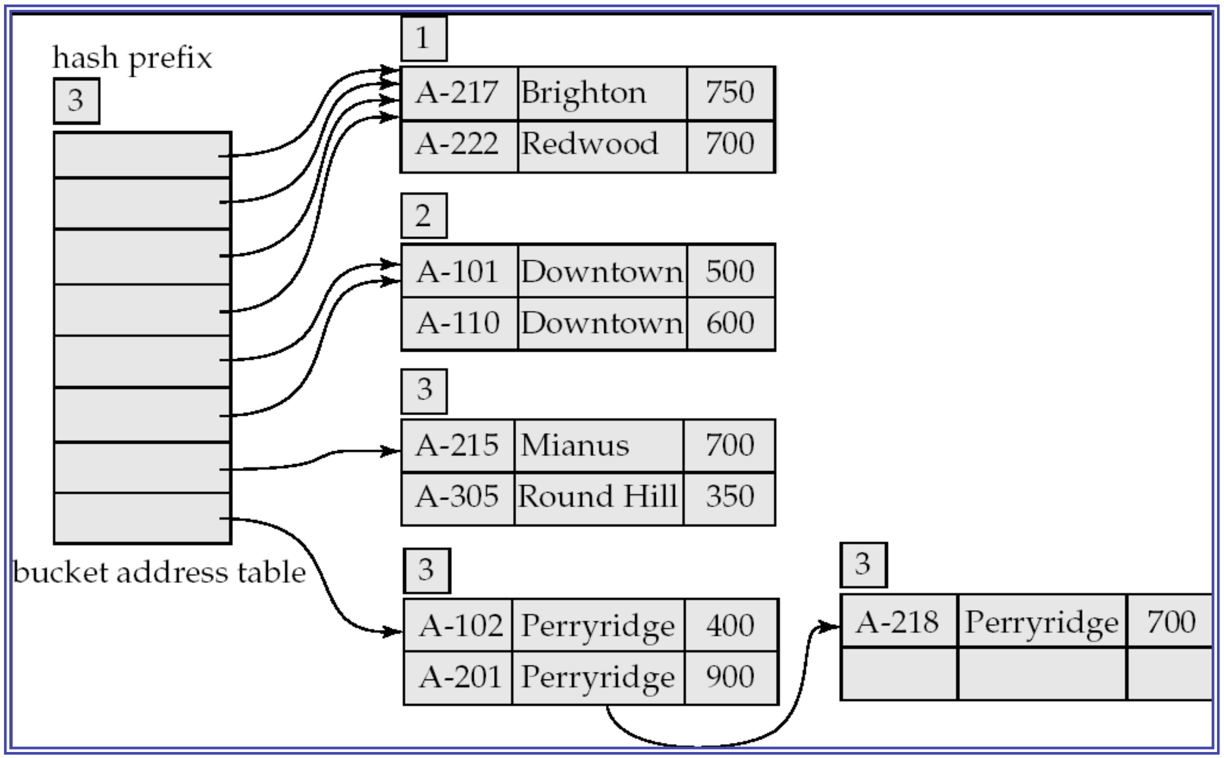
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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

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