CS 5330

Indexing

Indexing

- So far storage discussed
  - Heap file : no ordering, easy to maintain, but provide no help (speedup) in queries
  - Sequential file : help in some cases, but can be tricky to maintain, also sorting from scratch
- Need to have something that help searching but without too much overhead to maintain
- Indices (plural for index)

Indexing

- Assume we have a table
- An index is defined by a search key
    - Typically an attribute of the table
        - Denote it as the key (attribute)
    - Does NOT have to be unique (e.g. one can index on salary)
- An index (file) consists of index entries: (search key, pointer)
    - Search key: the value of the attribute
    - Pointer: pointer to the next location to access the record

Indexing

- Index structure assume the tuples in a file is on secondary storage (hard drive/SSD)
- They also make no assumption on how large the index is
  - The index structure is designed such that it can be stored on secondary storage also
  - Or some portion of it on storage while others in main memory
- Since secondary storage is used, a page is a basic unit of storage and reference
  - E.g: pointer to a tuple typically only point to the page that store the data

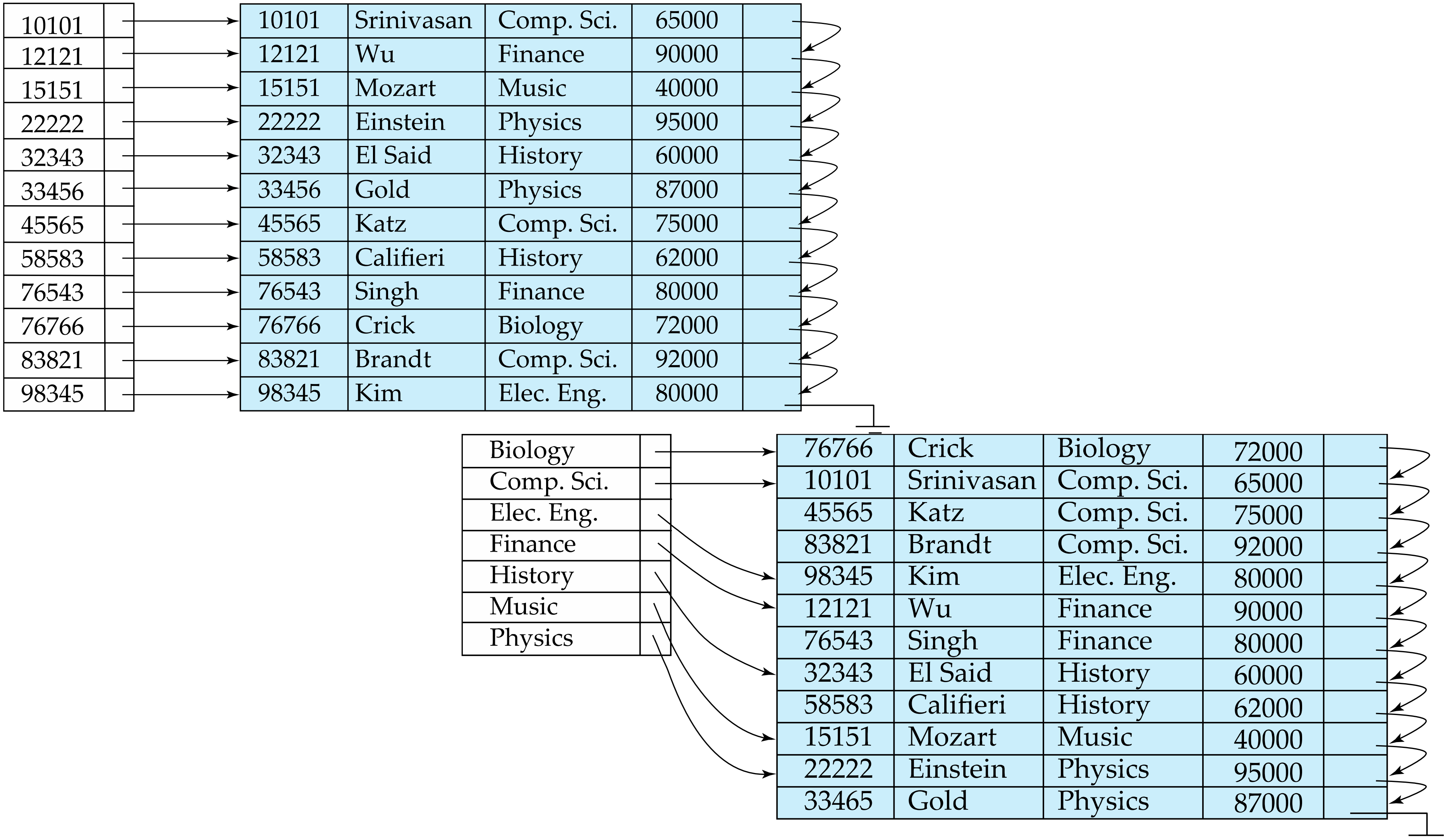
Indexing

- Two types of data structure used for indexing (called index structure)
    - Hash-based
        - Hash function is used to group data
        - Tuples are NOT sorted (since hash function does not guarantee to maintain order)
        - However, tuples that have the same key value will be grouped together
    - Ordered
        - Key attributes can be ordered
        - Tuples in the table (file) are ordered via the key (like a sequential file)
        - The index structure exists to enable fast query while not being penalized for updates
        - Typically tree-based (but not binary search tree!)

Indexing

- Two ways of generating index record
- Dense index
  - Each distinct value of key have at least one index record
    - Recall for secondary index, each tuple will have one record
- Sparse index
  - Some values of key do not have any index record
    - Mostly for ordered index
    - Location of the tuples with such key can be inferred

Dense index (example)



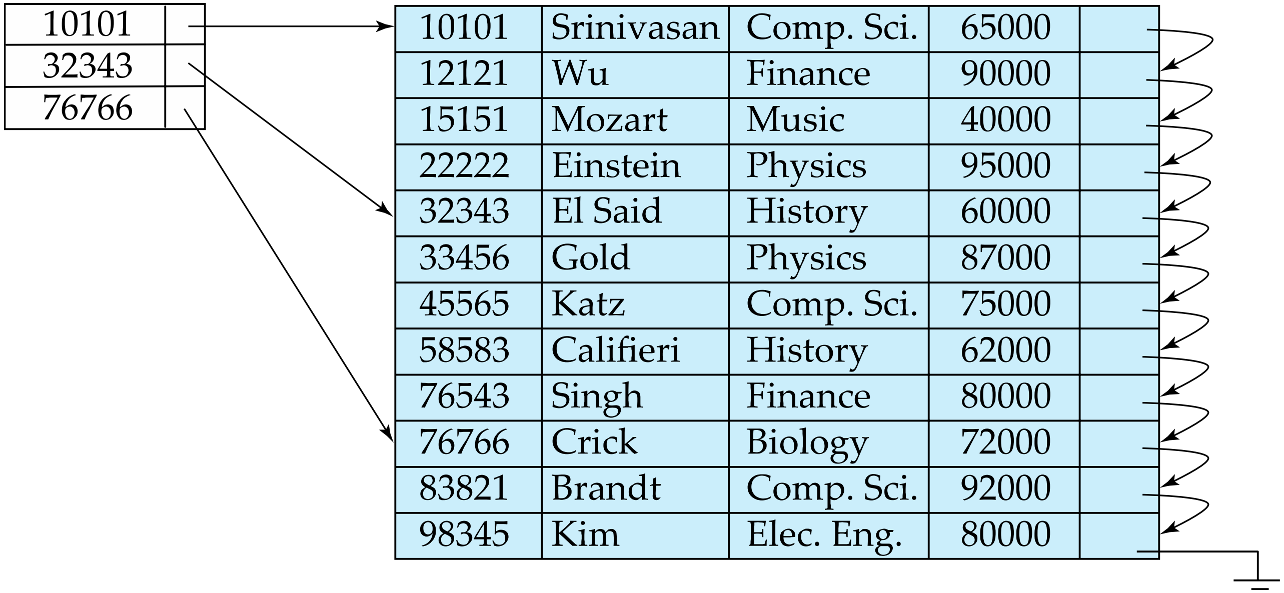
10101
12121
15151
22222
32343
33456
45565
58583
76543
76766
83821
98345

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

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 (example)



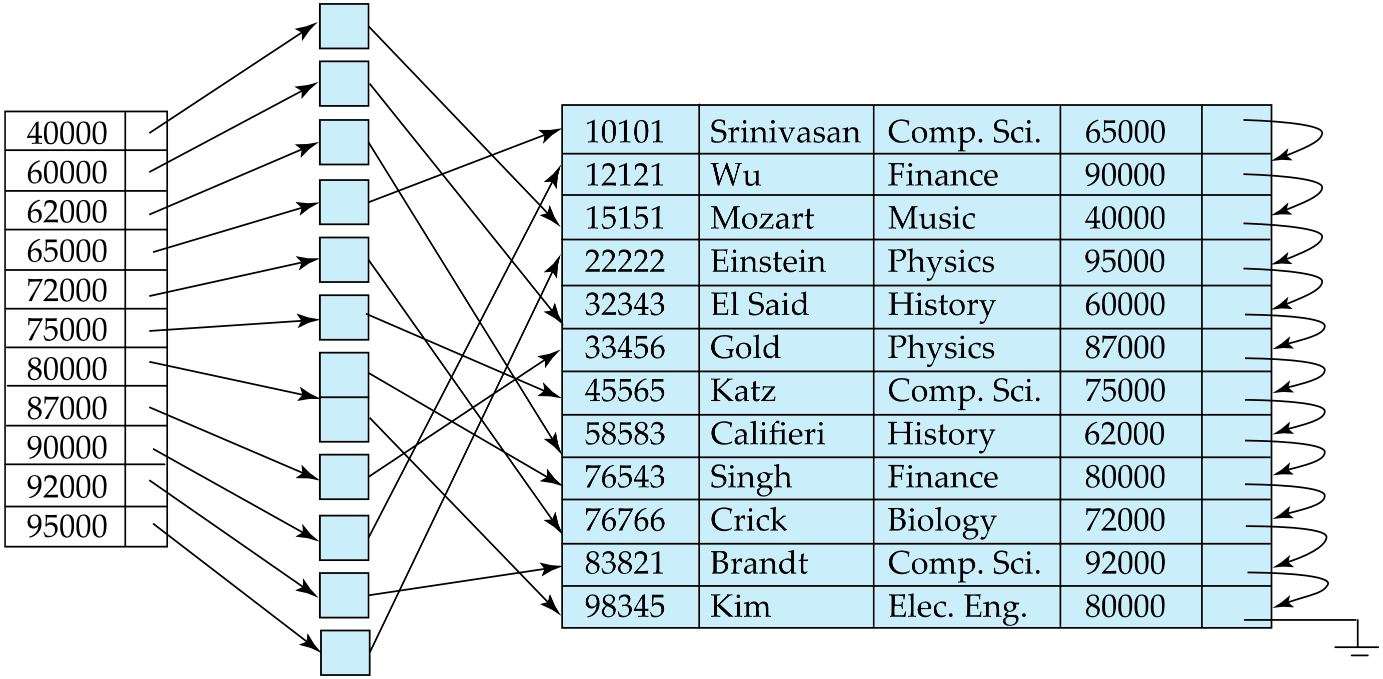
10101	
32343	
76766	

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	

Indexing

- Two different ways for tuples to be organized under an index
  1. The tuples are ordered exactly like the search key (clustering index / primary index)
     - The tuples themselves are typically ordered by the search key
       - For hash tables, it is not sorted, but tuples that have the same key values are typically stored together in same/adjacent pages
  2. The tuple are not ordered as the search key (non-clustering index / secondary index)
     - Need to have an index file,
     - For each tuple in the table, have a index record (key, pointer),
     - key value of the key attribute for that tuple
     - pointer points to where the tuple is (typically the page # of the page where the tuple resides
     - An index is then build on the index record

Secondary index (example)



40000
60000
62000
65000
72000
75000
80000
87000
90000
92000
95000

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

Notice that secondary index have to be dense (why?)

Clustering vs. Non-clustering index

- What’s the significant difference between the two?
- Consider the case
  - Instructor table of 20,000 tuples
  - Assume each page can store 100 tuples
  - If there is no empty space, 200 pages
- Case 1: a clustered ordered index on salary
  - Notice that the tuples are sequentially ordered by salary
- Case 2: an unclustered ordered index on salary

Clustering vs. Non-clustering index

- Now consider the following query:
  - SELECT * FROM instructor where salary > 100,000
- Case 1, for the clustered order index
  - First use the index to find all the page that have tuple that have the smallest salary > 100,000
  - Then read all the pages from that page onwards
  - Let say k tuples satisfies the query
  - Then the number of pages read = k/100
    - Cannot be larger than the size of the table

Clustering vs. Non-clustering index

- Now consider the following query:
  - SELECT * FROM instructor where salary > 100,000
- Case 2, for the unclustered order index
  - First use the index to find the page for each tuple that have the salary > 100,000
  - Then read all the pages from that page onwards
  - Let say k tuples satisfies the query
  - Then the number of pages read = k in the worst case
  - If k = 2, then at worst 2 page read fine
  - If k = 200, then worst case one have to read the whole table
    - The index is useless

Fundamental problem in query execution/optimization

- Whether you want to use an index or not depend on the size of the result of a query

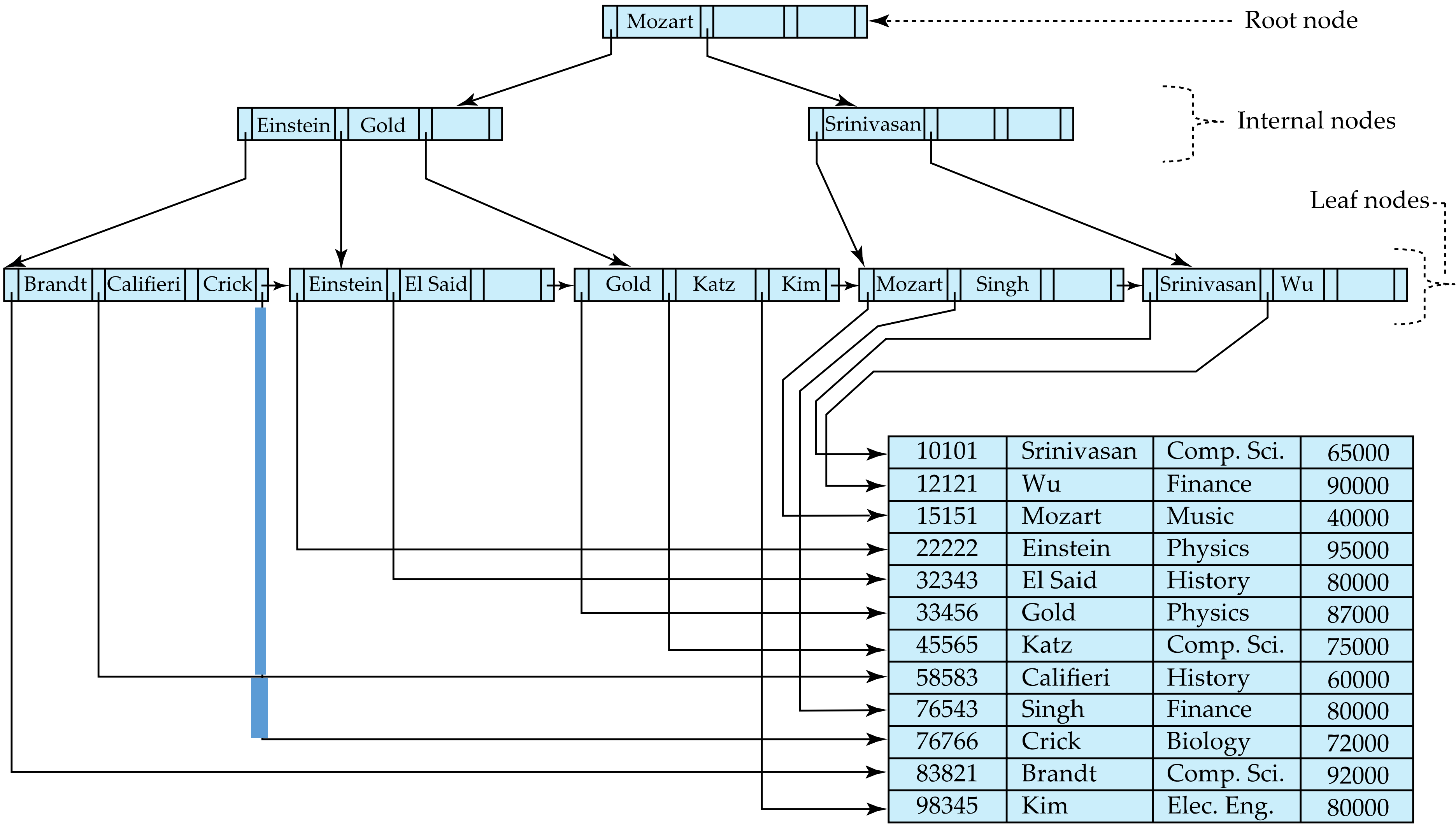
BUT

- You do not know the size of the result unless you run the query, which you will have to decide whether to use the index

Index structures

- Two most commonly used index structure for databases
    - B+-trees
        - Modification of B-trees for databases
        - Good for ordered data
    - Hash tables
        - Modification of main memory hashing function
        - Good for exact queries (WHERE attribute = value)

B+-tree



Root node

Mozart

Internal nodes

Einstein | Gold

Srinivasan

Leaf nodes

Brandt | Califieri | Crick

Einstein | El Said

Gold | Katz | Kim

Mozart | Singh

Srinivasan | Wu

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

B+-tree

- Tree-based structure
- Tree is made up of nodes
  - Typically each node corresponds to a page on secondary storage
- Data are stored at leaves only
  - Different from B-tree
  - If primary index, then tuples themselves are stored
  - If secondary index, then store index records (key, pointer)

B+-tree

- Each node can store multiple items
  - Assume a page has 1000 bytes
  - Now if a tuple is 100 bytes, then a leaf can store 10 nodes
  - Now if the B+-tree is used as secondary index, each node will store key + pointer, typically around 20-25 bytes total. So each leaf can store 40-50 records

B+-tree

- Internal node:

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

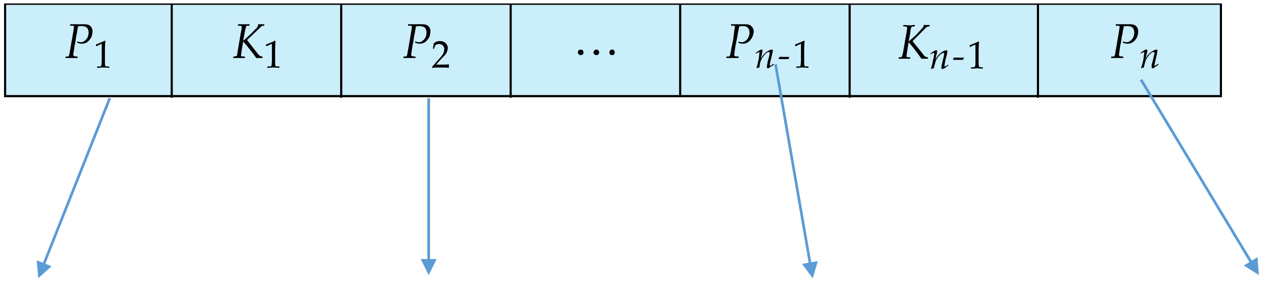
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

B+-tree

- Internal node:



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

Interpretation:

- All data that is stored under the subtree pointed by P_j has $K_{j\text{-}1} \leq$ key value $< K_j$
  - All data under P_1 is $< K_1$
  - All data under P_n is $>= K_{n\text{-}1}$
  - equal sign can shift from left to right
  - If duplicates are allowed (e.g. secondary index on non key attributes), it is possible for equal sign to be on both sides

B+-tree: Query

- Query is just like any tree search
- To find the tuple/index record with key value x
- Then starting at the root node,
    - For each internal node, find j such that $K_{j-1} <= x < K_j$
    - Then follow the pointer of P_{j-1} to the next internal node
    - Repeat the process until a leaf node is reached
    - Search the leaf node to see if the tuple/index record is present
    - Boundary cases, when $x < K_1$ and $x >= K_{n-1}$ (go to P_1 and P_n respectively)

B+-tree: Query

- Similar case for a range query
- Find all tuples with key value between a and b
- For each internal node, check if the range (K_{j-1} ,K_j) intersect with the range (a, b); if so, continue on pointer P_{j-1}
    - Once again, consider boundary cases (K_1 < a, b < K_{n-1})
- Also, often the leaf nodes are either stored as a sequential file, or have links connected them in order
    - Then only need so search for a, and then follow the order / links
- Query time (not counting time to retrieve the data) is proportional to the height of the tree

B+-tree: Structure

- Recall all data are store in the leaves
  - Internal nodes’ key value are used to guide the search
- Goal of updates, to maintain two conditions
  - The tree remain balanced (i.e. all the leaf nodes are at the same level)
  - Every node (except the root) has to be at least half full
- The two conditions combined ensure the height of the tree to be logarithmic to the number of items

B+-tree: Insertion

- Suppose one want to insert a tuple (or record, same below) of key value x
- First, run a query to find the leaf node where x would have been located
- If that leaf node is not full, then insert the tuple into the leaf node, and insertion is done
- What if the node is already full?

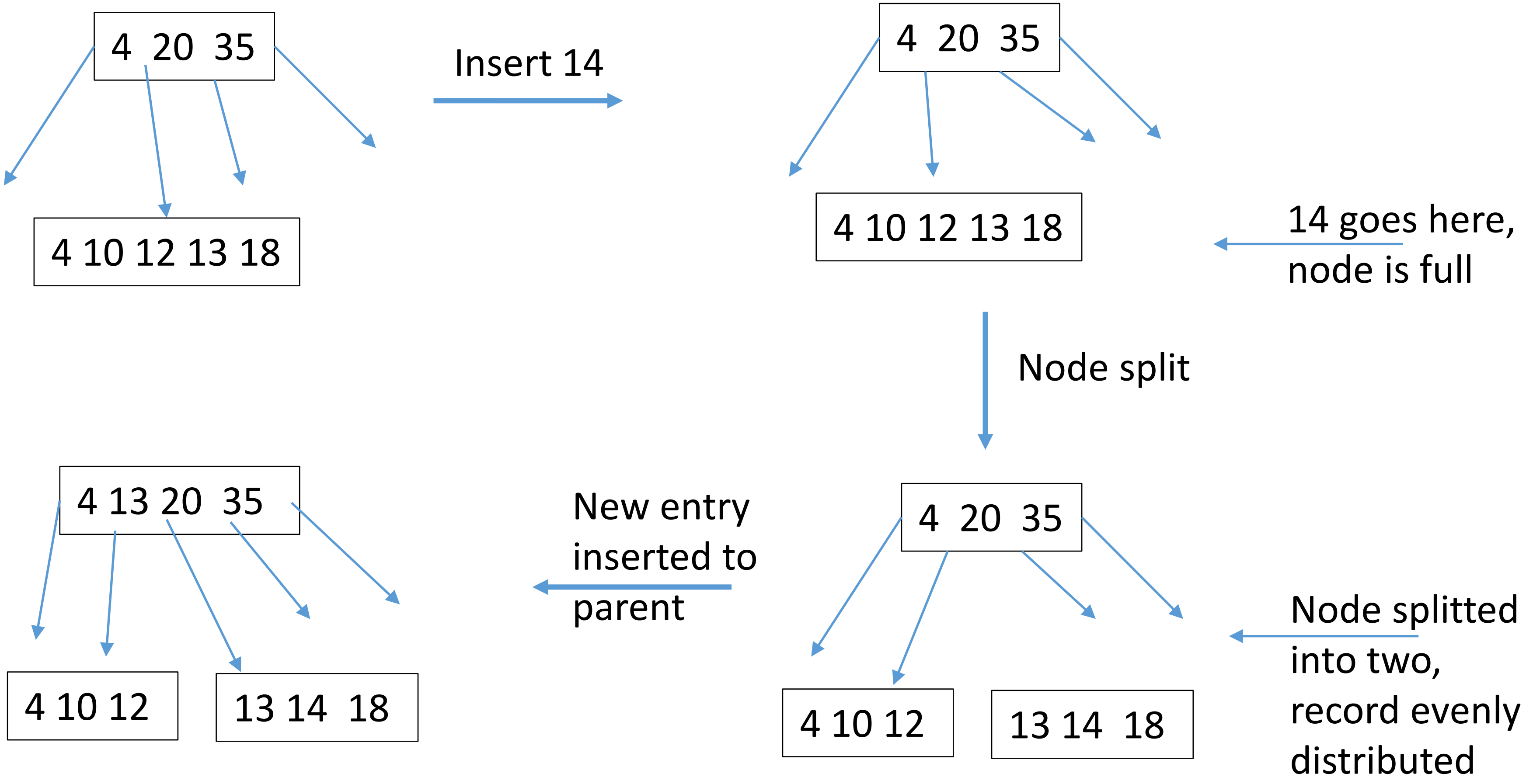
B+-tree: Insertion

- If the node is full
    - Need more space to store the data
    - Create a new node
    - Distribute the data evenly across both nodes, in order
    - Now the new node also need to be references
    - So the original node’s parent need to create a new entry

B+-tree: Insertion

Example:

- assume leaf node can hold 5 records, internal nodes can hold 5 pointers & 4 numbers
- Leaf node contain tuple/index records that are not shown



4 20 35

4 10 12 13 18

Insert 14

4 20 35

4 10 12 13 18

14 goes here, node is full

Node split

4 20 35

4 10 12

13 14 18

Node splitted into two, record evenly distributed

New entry inserted to parent

4 13 20 35

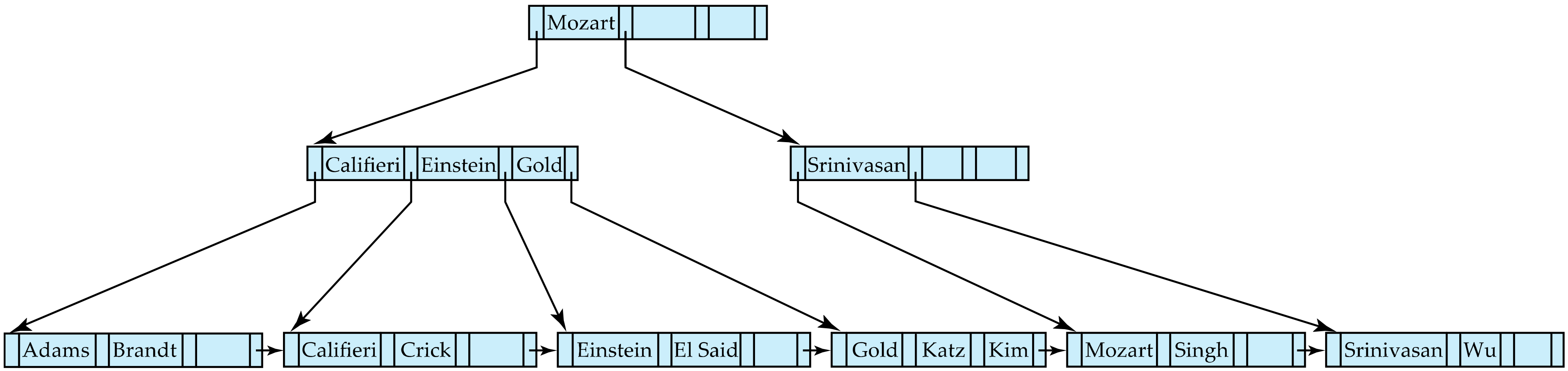
4 10 12

13 14 18

B+-tree: Insertion

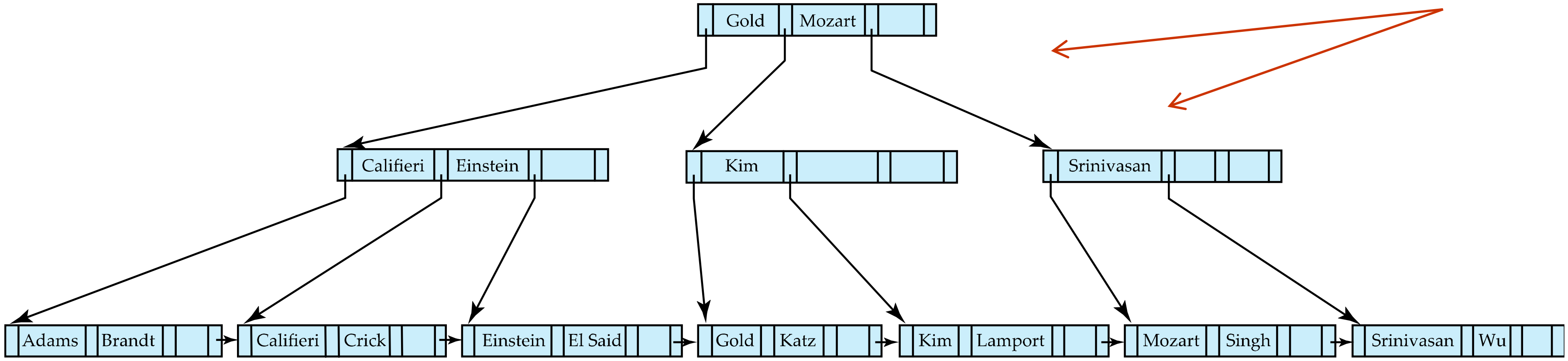
- What if the parent is full?
  - Need more space for the parent
  - Create a new internal node
  - Distribute the (key, pointers) evenly
  - Need to create a new entry for the parent's parent to insert
- Essentially the same code for the leaf split
- The split will continue propagate upwards if necessary
- When the propagation reaches the root, the root is split, and a new root is build on top to be the parent of the two nodes
  - This is where the tree grows in height
  - This is also the reason why the root may be less than half full

Example:



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

Affected nodes

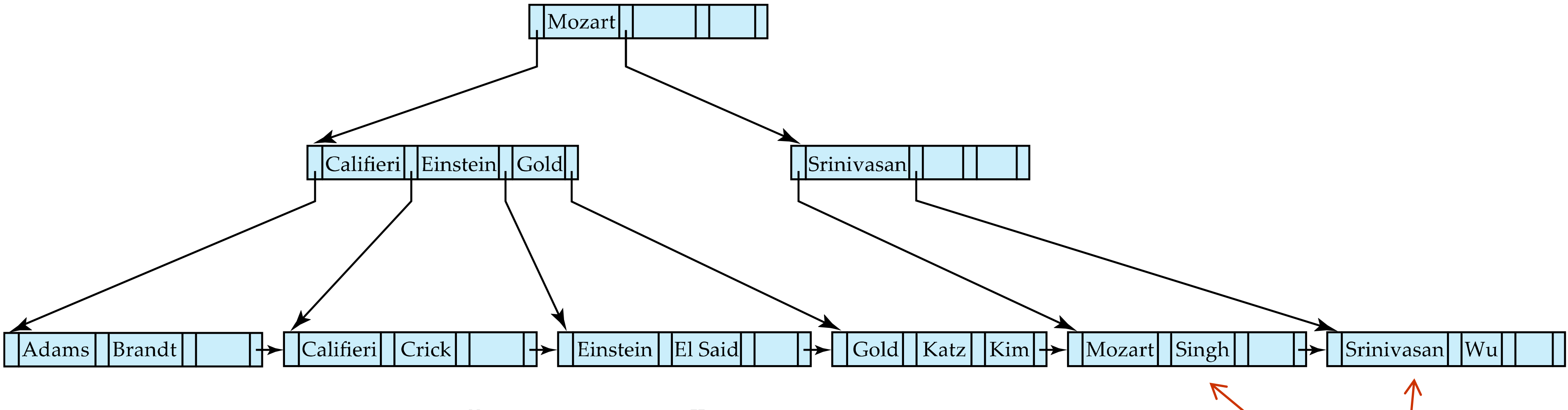


Affected nodes

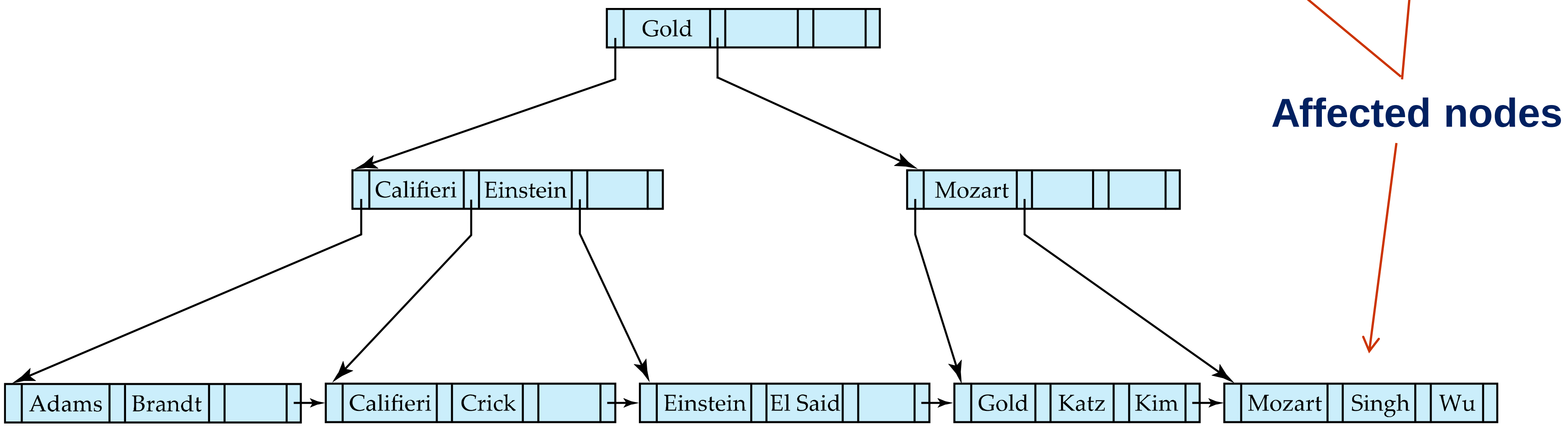
B+-tree : Deletion

- Similar approach
- First find the leaf node storing the item to be deleted
- Then remove it
- If the node still is at least half-full, then done
- Otherwise:
  - Examine the node’s neighbor
  - If anyone of its neighbor has is more than half-full, then move some data around to fill all the nodes to half-full
    - Will need to update the parent’s key value to maintain the correctness
  - Otherwise (all the neighbors are exactly half-full), merge the node with one of its neighbor
    - Remove the corresponding key/pointer from the parent
    - If at the top level, the root will only have one child – remove the root

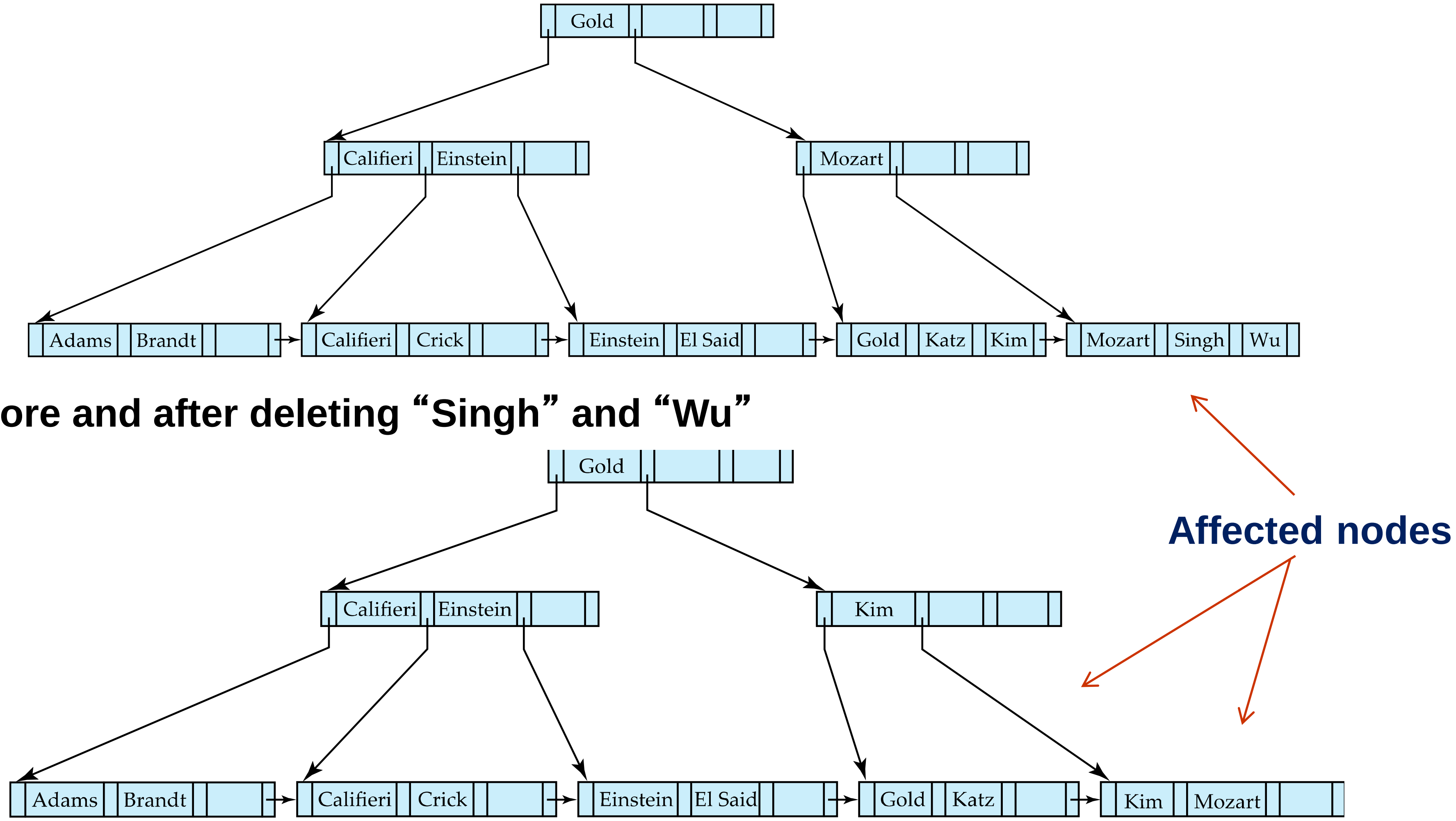
Example:



Before and after deleting “Srinivasan”

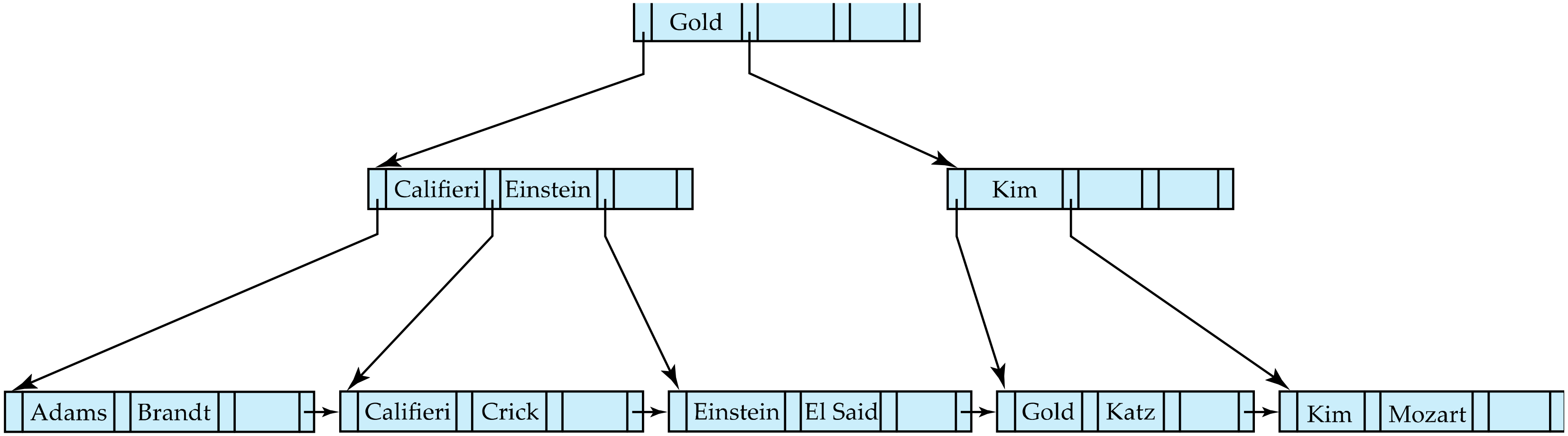


- Deleting “Srinivasan” causes **merging** of under-full leaves

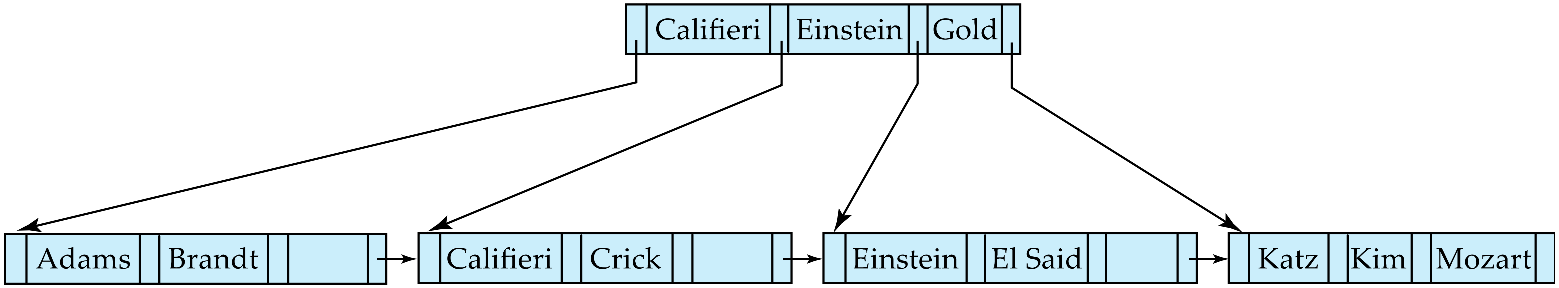


Before and after deleting “Singh” and “Wu”

- 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”



- 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

B+-tree: Update cost

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

B+-tree as the clustering index / file organization

- B^+-Tree File Organization:
  - Leaf nodes in a B^+-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node at least half full

Indexing strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys in leaf node can be compressed by sharing common prefixes

B+-tree : Bulk loading

- Inserting entries one-at-a-time into a B^+-tree requires $\geq$ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

B+-tree: Other issues

- SSD / Flash memory
  - Random I/O cost much lower on flash
  - Writes are not in-place, and (eventually) require a more expensive erase
  - Optimum page size therefore much smaller
  - Bulk-loading still useful since it minimizes page erases
  - Need specialized write-optimized tree structures have been adapted to minimize page writes for flash-optimized search trees
- Main memory
  - Random access in memory
    - Much cheaper than on disk/flash
    - But still expensive compared to cache read
    - Data structures that make best use of cache preferable
    - Binary search for a key value within a large B^+-tree node results in many cache misses
  - B^+- trees with small nodes that fit in cache line are preferable to reduce cache misses
  - Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array

Hashing based techniques

- Recall basis of hashing
- You want to store items. Each item has a key value x
- Create a hash function h(x) to map x to an integer between [0..n-1]
- Have a hash table of n slots.
- Item x will be stored in slot h(x)
    - Multiple techniques are used to handle collisons (multiple

Hashing based techniques

Modification for database system

- Each entry of a hash table is now a “bucket”
    - Typically size of a page
    - Thus more than one element
    - All items with the same hash values (even different keys) stored in the same bucket
- Overflow still possible
    - Instead of using complicated schemes, simple add overflow buckets
        - Essentially building a link list of buckets for each entry
        - Notice that in secondary storage, this can be a file itself
        - This is known as chaining
    - Other techniques are not used

Hashing based techniques -- Example

- 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

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

Hashing based techniques

- The above technique is also known as ***static hashing***
  - Static as in the number of buckets is fixed
  - The size of each bucket can grow (overflow)
- Problem with database with a lot of insert/delete
  - 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***

Dynamic hashing

- Key idea: the number of buckets can increase and decrease with the amount of data
- Implication: the hash function has to be able to adapt to the number of buckets available
- General approach
    - Convert the key (via some function) to a number x
    - Bucket for the item = x MOD (current # of buckets)
- Another important issue
    - Need to keep track of location of each bucket
    - Need some form of table to keep track of it (or some other convention, see later)
        - Usually a table, with each entry pointing to the first page of a bucket

Dynamic hashing

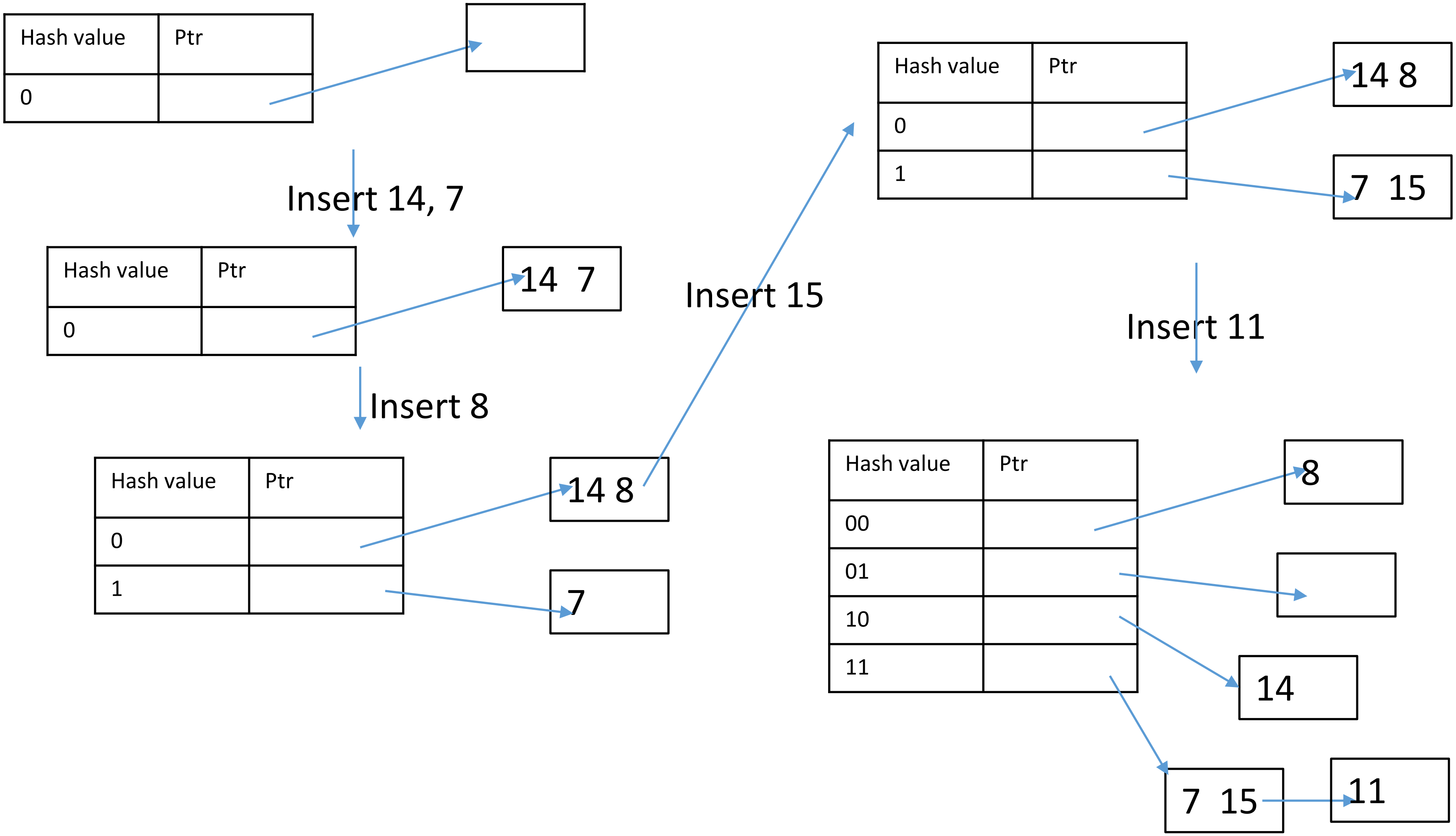
- Key idea: the number of buckets can increase and decrease with the amount of data
- Implication: the hash function has to be able to adapt to the number of buckets available
- General approach
  - Convert the key (via some function) to a number x
  - Bucket for the item = x MOD (current # of buckets)
- Another important issue
  - Need to keep track of location of each bucket
  - Need some form of table to keep track of it (or some other convention, see later)
    - Usually a table, with each entry pointing to the first page of a bucket

Dynamic hashing

- Start with 1 bucket (so no hashing)
- At any given time
  - When an insertion make a bucket overflow
  - Double the number of buckets
    - The table storing pointers need to be doubled
    - Each bucket will start with a single page
  - Rehash the whole file (Some data will need to go to a new bucket)
- This implies at any stage, the hash function becomes looking at the rightmost bits of the number (written in binary) converted from the key

Example:

- Items to be stored are numbers
- Assume each bucket store 2 numbers



Hash value	Ptr
0	

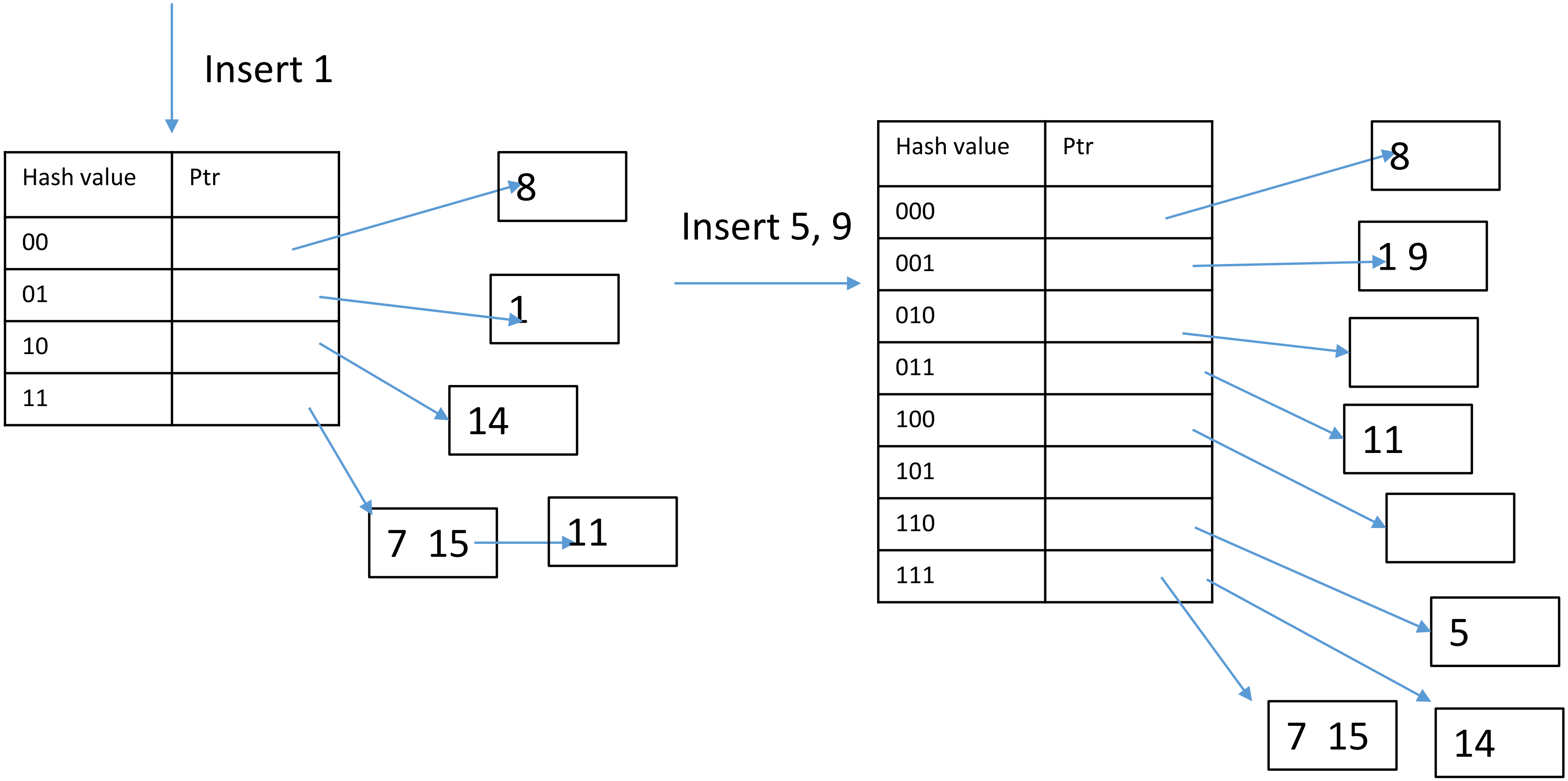
Hash value	Ptr
0	

Hash value	Ptr
0	
1	

Hash value	Ptr
0	
1	

Hash value	Ptr
00	
01	
10	
11	

Ctd from last slide



Insert 1

Hash value	Ptr
00	
01	
10	
11	

Insert 5, 9

Hash value	Ptr
000	
001	
010	
011	
100	
101	
110	
111	

Dynamic hashing

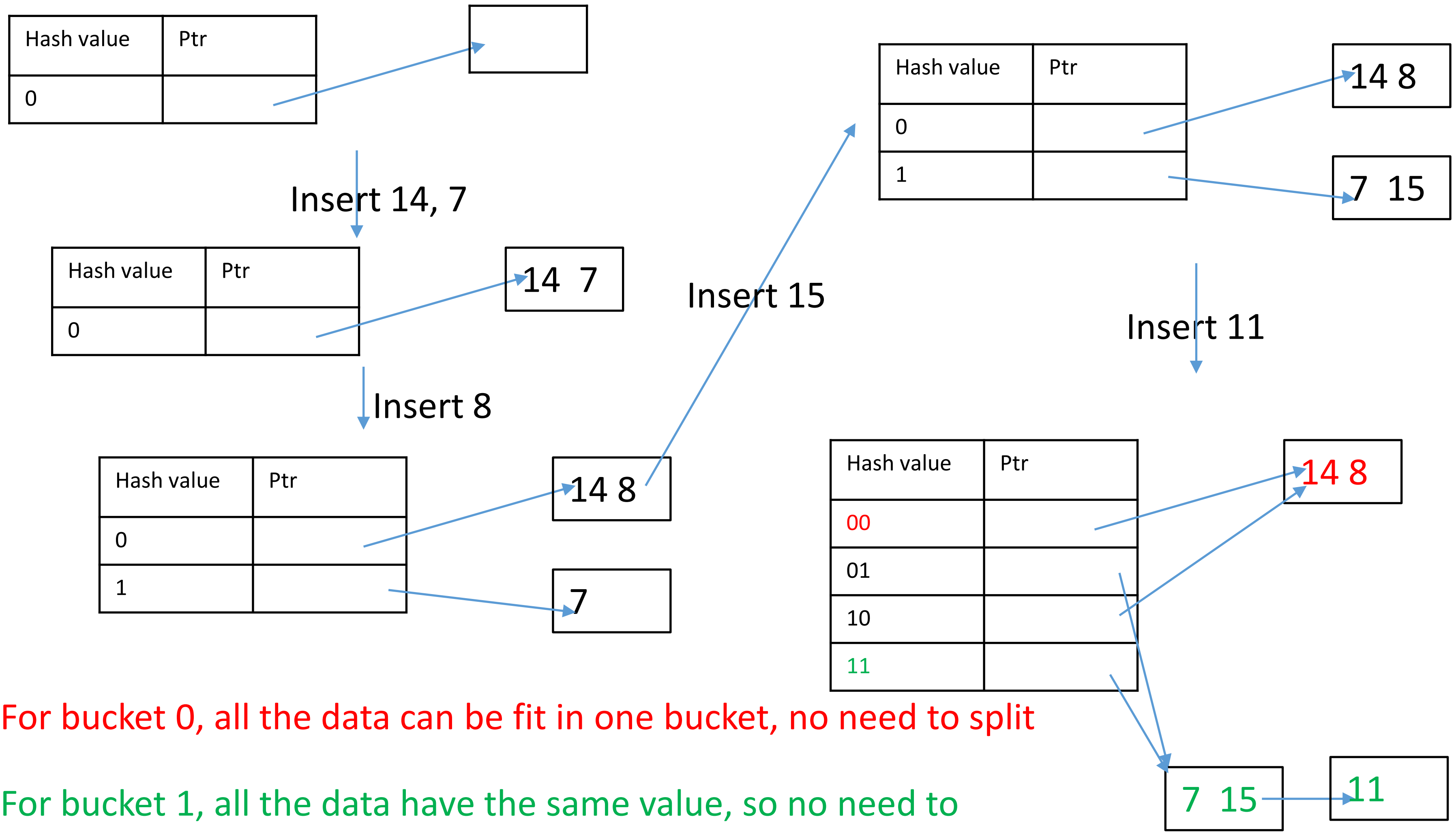
- Limitation
  - Double of number of buckets – exponential growth
  - A lot of empty buckets potentially
- Variations
  - One does not have to immediately rehash when a bucket is overflown
    - Allow some overflow bucket, can slow down the growth
    - Also avoid empty buckets
    - Price : slow down access

Extensible hashing

- One problem with dynamic hashing
  - When rehashing, the number of buckets are doubled
  - Some buckets may be unnecessary
- One way to get around it, extensible idea
- Key idea:
  - When you rehash, only create new buckets when necessary

Example:

- Items to be stored are numbers
- Assume each bucket store 2 numbers



Hash value	Ptr
0	

Insert 14, 7

Hash value	Ptr
0	

Insert 8

Hash value	Ptr
0	
1	

Insert 15

Hash value	Ptr
0	
1	

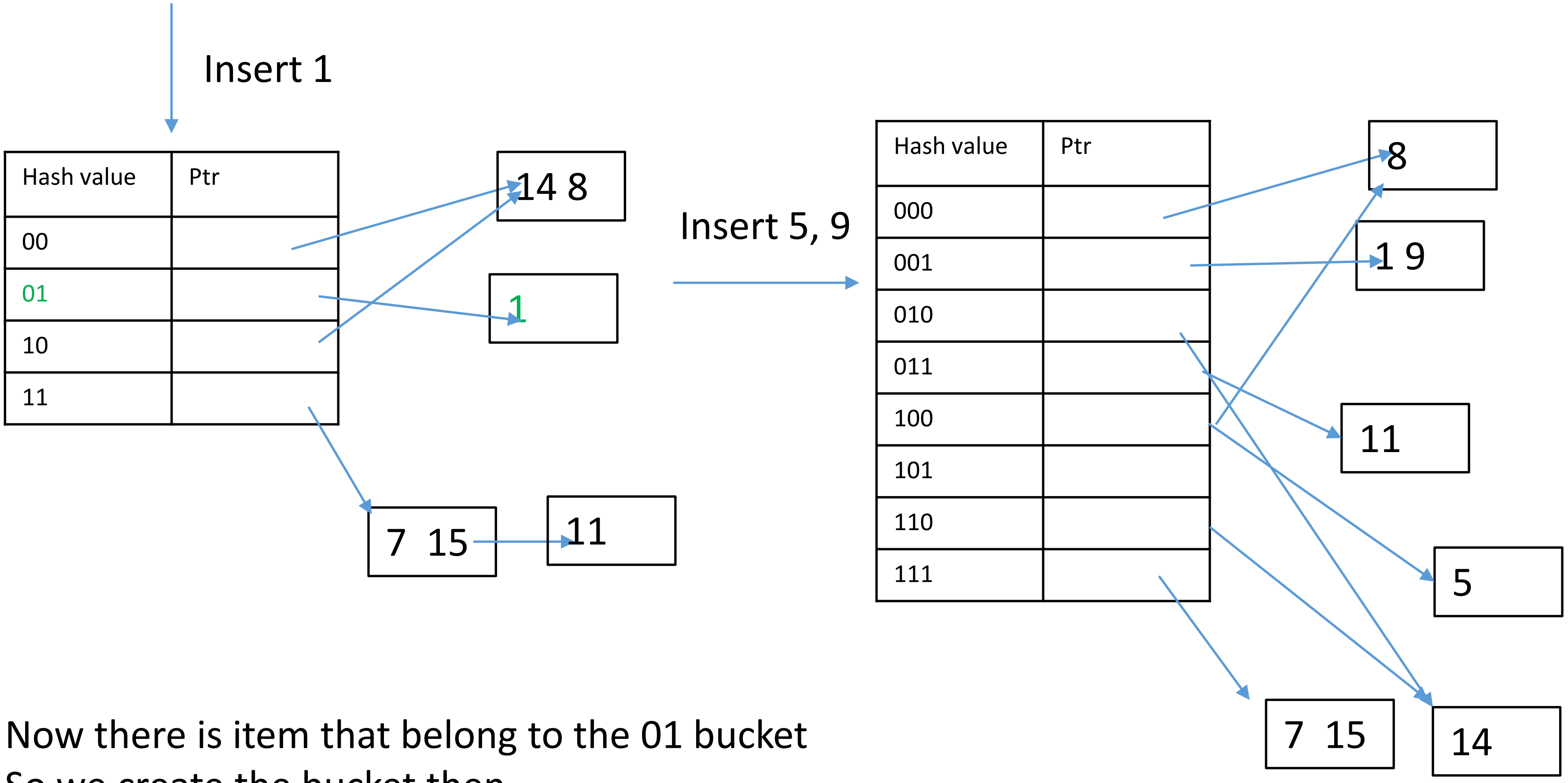
Insert 11

Hash value	Ptr
00	
01	
10	
11	

For bucket 0, all the data can be fit in one bucket, no need to split

For bucket 1, all the data have the same value, so no need to create a new bucket fo r hash value 01

Ctd from last slide



Hash value	Ptr
00	
01	
10	
11	

Hash value	Ptr
000	
001	
010	
011	
100	
101	
110	
111	

Now there is item that belong to the 01 bucket
So we create the bucket then

Linear Hashing

- Why?
    - Most extensible hashing techniques require some sort of exponential growth
        - Introducing one more bit will double the size of the hash table/index
        - Massive rehashing slow things down

Linear hashing

- Key ideas
  - Hash table entries grow in a linear fashion
  - Have a pre-defined order of splitting the bucket, ***regardless of which bucket is overflowing***
    - A overflow bucket must wait for its turn to be split (rehashed)
    - Need overflow buckets (linked list)
  - This ensure the growth is linear
  - Price to pay: overflow bucket slow down access
    - Can be serious is hash function is lousy / data distribution is lousy

Linear hashing

- Assume Hash function = rightmost k bit of the number (k changes with the algorithm)
- For linear hashing, maintain two variables
    - Level: the current level of hashing,
        - Starts at 0
        - At the start of level k, there will be 2^k buckets
            - E.g. k = 2, then buckets are 00, 01, 10, 11
        - A level finishes when all the buckets at the start of the level is split

Linear hashing

- Assume we are hashing function
- Hash function = rightmost k bit of the number (k changes with the algorithm)
- For linear hashing, maintain two variables
    - Ptr: Points to the next bucket to be split
        - Whenever ANY bucket overflows, it is the bucket that ptr points to that split
        - When a bucket is split, it split into 2 buckets by adding 0 and 1 as the new leftmost bit
            - E.g. bucket 01 is split into bucket 001 and 101
            - Only the bucket that is split need to be rehashed
        - Reset to 0 at the start of each level
        - Once the bucket is split, increment ptr by 1 (until end of level, by then it reset back to 0)

Linear hashing

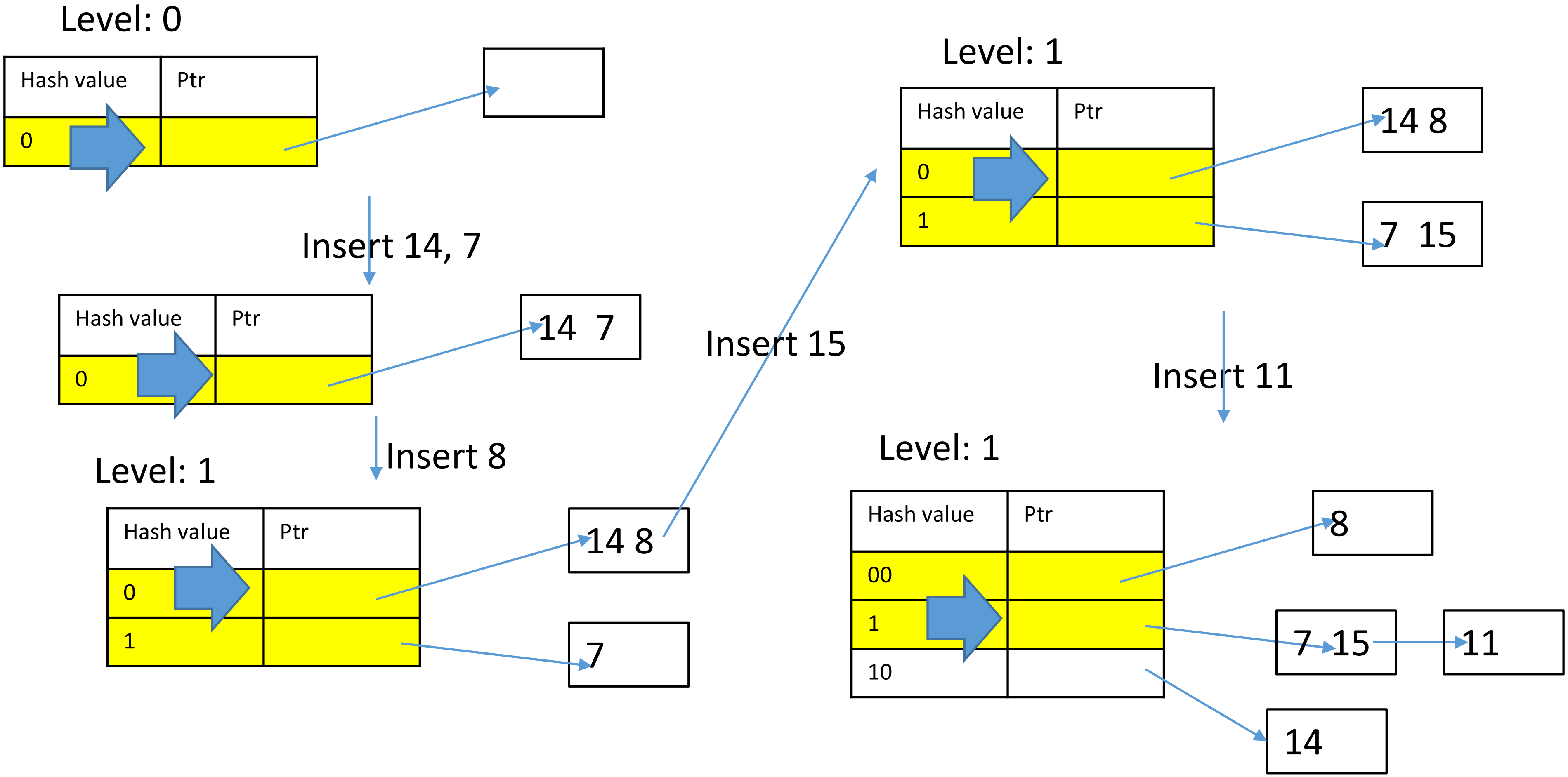
- Ptr also help us to determine which bucket should one search
  - Consider we are at level k
    - All buckets that are not split are represented by k bits
    - All buckets that are split are represented by k+1 bits
  - When I search for a number
    - Look at the rightmost k bit of the hash value
    - If it is >= ptr, then go straight to this bucket
    - If it is < ptr, than look at one more bit at the left, and that will denote the bucket to search

Linear hashing

- Splitting occurs when one inserted into a bucket that is full (or overfull).
- Remember that the bucket that is overfull may not be the one that is split
    - In such case, we use overflow buckets (building a link list) to store the extra value
- When a bucket is split, it is split into two
    - Even if the split bucket is overfull, we do NOT continue the splitting
    - Rehashing is done on the split bucket only (and only that is needed)
- More advanced versions of linear hashing will change when splitting occurs (e.g. do not wait till a bucket overflows).

Example:

- Items to be stored are numbers
- Assume each bucket store 2 numbers



Level: 0

Hash value	Ptr
0	

Insert 14, 7

Hash value	Ptr
0	

Insert 8

Level: 1

Hash value	Ptr
0	
1	

Insert 15

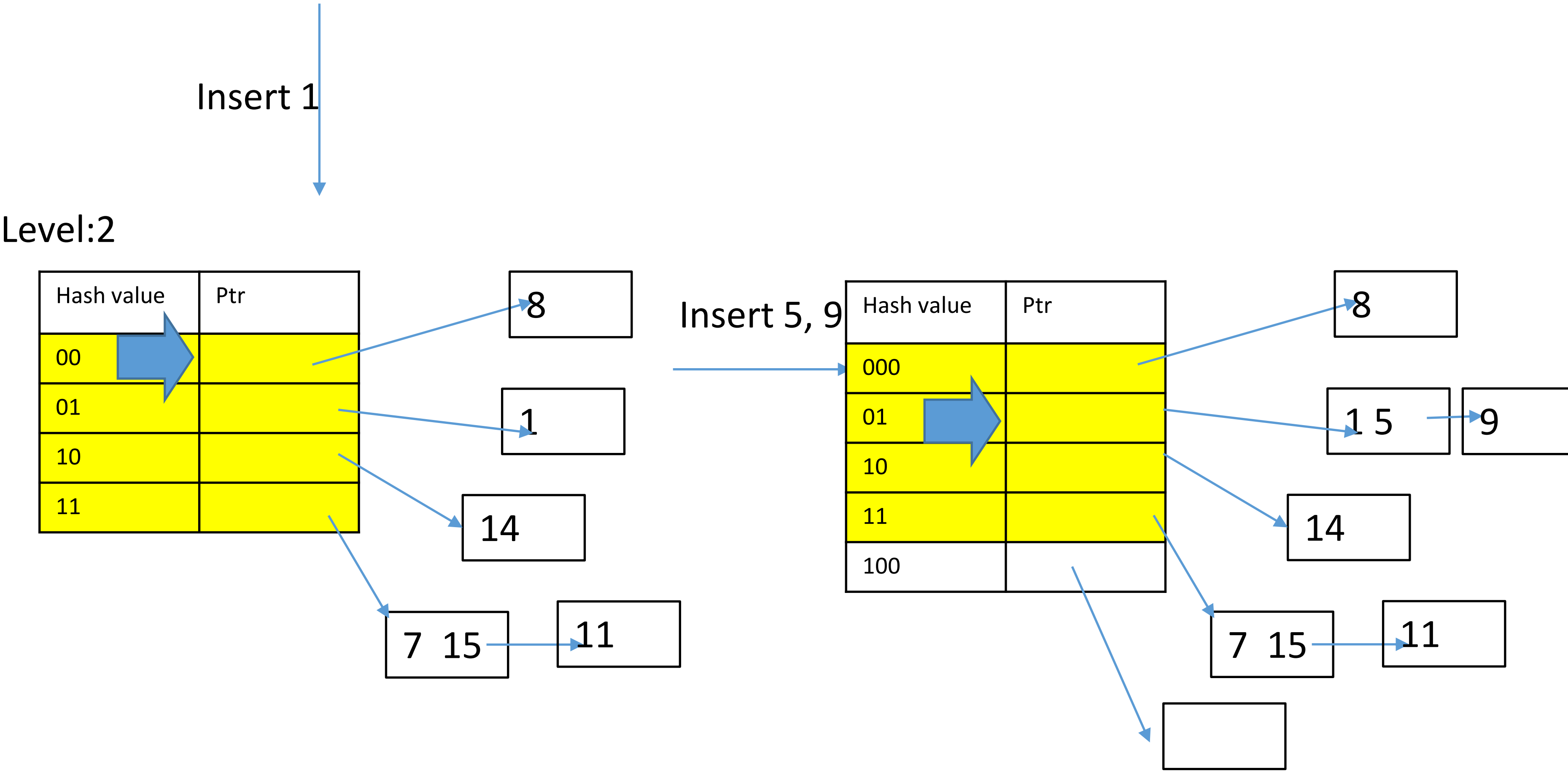
Level: 1

Hash value	Ptr
0	
1	

Insert 11

Level: 1

Hash value	Ptr
00	
1	
10	



Insert 1

Level:2

Hash value	Ptr
00	
01	
10	
11	

Insert 5, 9

Hash value	Ptr
000	
01	
10	
11	
100	