Implementation of Relational Operators/Estimated Cost

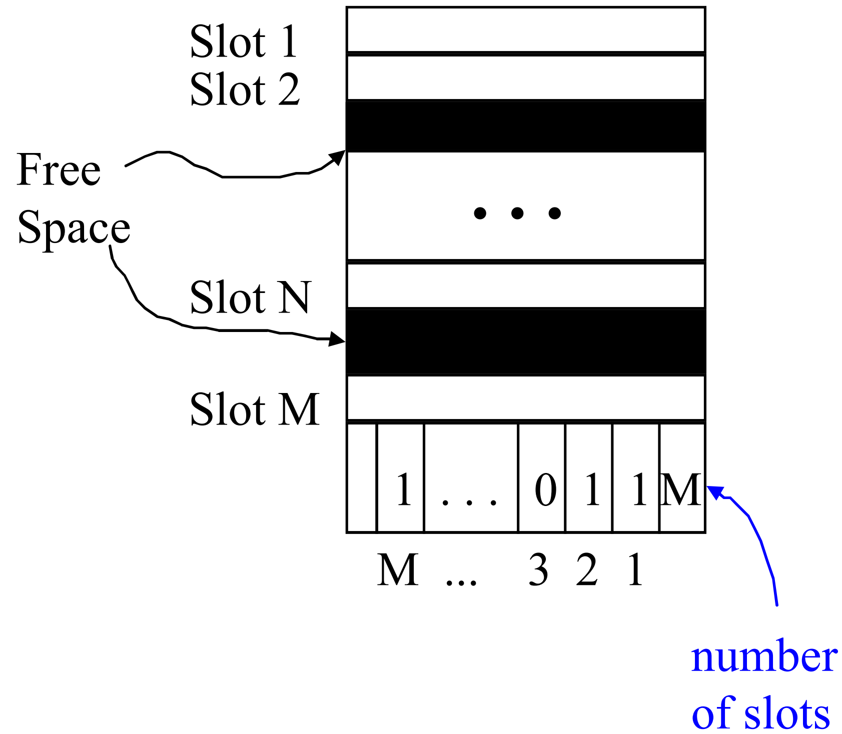
1. Select
2. Join
3. Project

SELECT Operator $\sigma_{R.A \text{ op } value}(R)$

- Retrieve all tuples in R whose values on attribute A satisfy certain condition
- Factors to estimate the cost of performing a select operation
  1. No index
     - unsorted data
     - sorted data
  2. Index
     - tree index
     - hash-based index

Sailors(<u>sid</u>, sname, rating, age)

- Each Sailors tuple is 50 bytes (fixed length record format)
- Total number of tuples: 40,000
- A page size is 4K bytes
- All pages are full, unpacked bitmap; 96 bytes are reserved for slot directory
- How many pages for Sailors?



- One page can contain at most (4096-96)/50 = 80 tuples
- Sailors occupies pages 40,000/80 = 500

No index, unsorted data

$$\sigma_{R.A = value}(\mathrm{R})$$

Suppose R is Sailors

Best access path: File Scan
I/O Cost: 500 pages
I/O time cost: 500 * time to access each page
Complexity: O(|R|)

Notation: |R| is the number of pages in R

No Index, sorted file on R.A

$$\sigma_{R.A = value}(\mathrm{R})$$

Suppose R is Sailors

Sorted on R.A

Best Access Path:

- Binary search to locate the first tuple with R.A=Value
- Scan the remaining records

I/O Cost:

- $\log_2(|R|)$+Cost of scan for remaining tuples ($0 \sim |R|$)

Tree Index on R.A

$$\sigma_{R.A = value}(\mathrm{R})$$

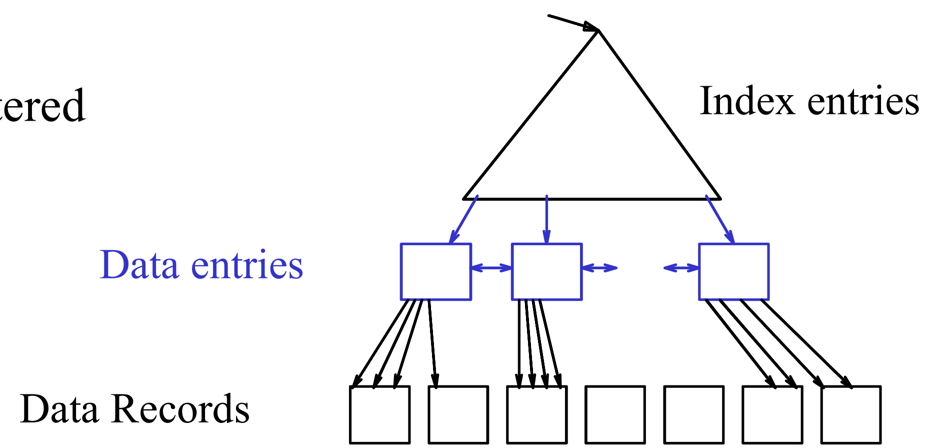
Selection Cost =

cost of traversing from the root to the leaf +

cost of retrieving the pages in the sequence set +

cost of retrieving pages containing the data records.

- Need to know
  - Clustered or unclustered
  - Dense or sparse

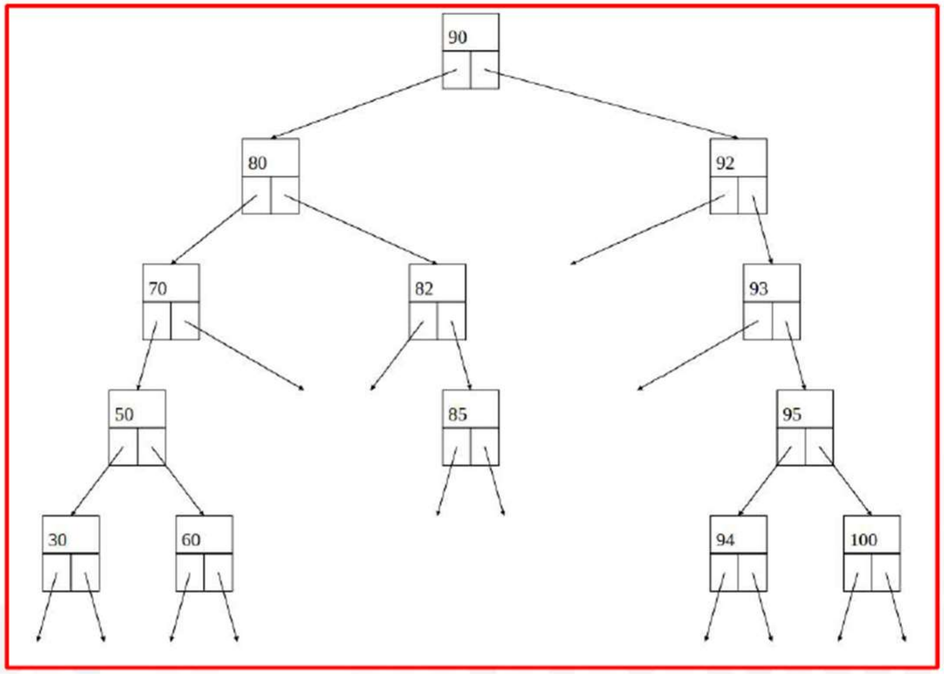


B+-tree, mostly used indexing in RDBMS

Why not Binary Tree

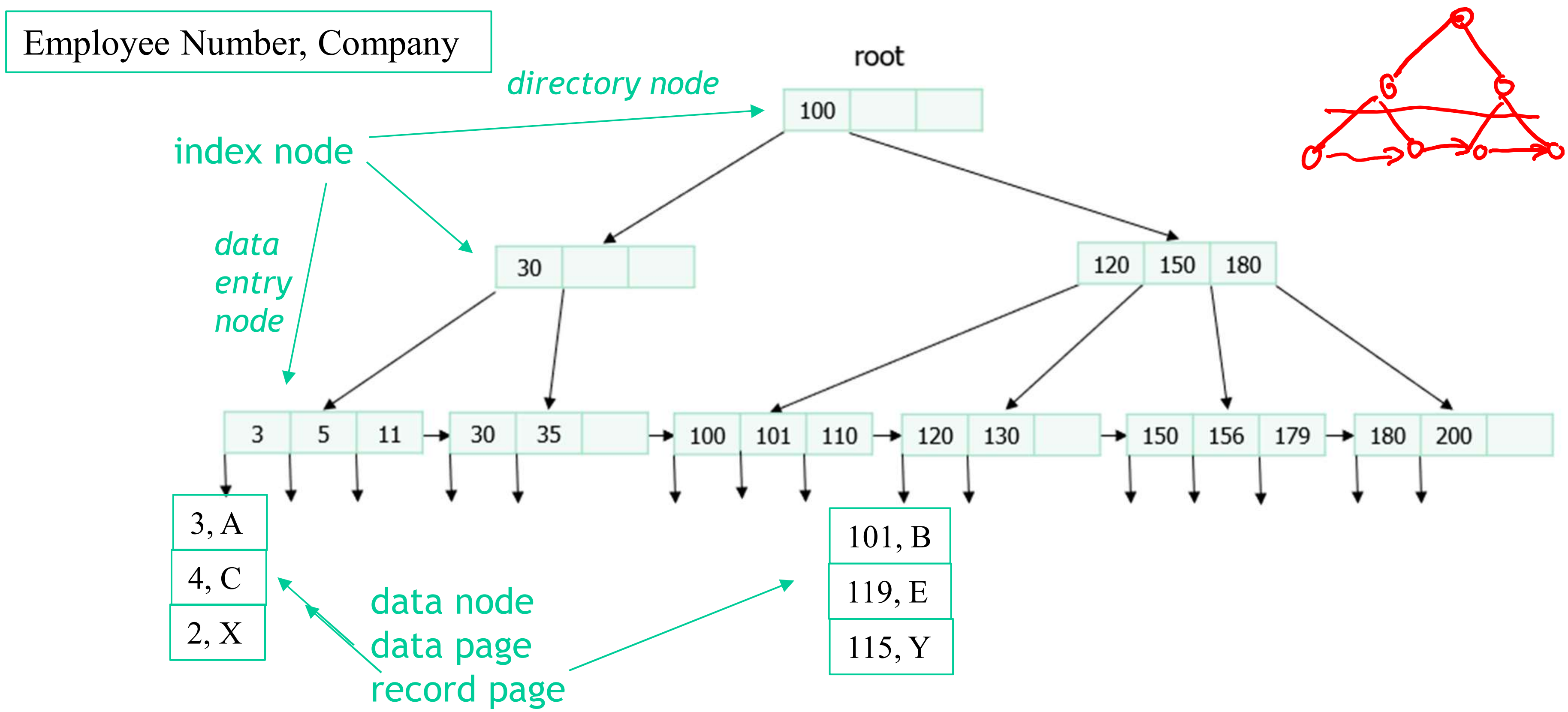
```
bst = BST()

bst.insert(90, ['Bugs Bunny', 'CS411', 90])
bst.insert(92, ['Donald Duck', 'Bio300', 92])
bst.insert(93, ['Donald Duck', 'CS423', 93])
bst.insert(95, ['Donald Duck', 'CS411', 95])
bst.insert(80, ['Bugs Bunny', 'CS423', 80])
bst.insert(70, ['Mickey Mouse', 'CS423', 70])
bst.insert(94, ['Peter Pan', 'CS411', 94])
bst.insert(50, ['Charlie Brown', 'Econ101', 50])
bst.insert(82, ['Peter Pan', 'Econ101', 82])
bst.insert(60, ['Eeyore', 'Bio300', 60])
bst.insert(85, ['Mickey Mouse', 'Econ101', 85])
bst.insert(100, ['Ariel', 'CS411', 100])
bst.insert(30, ['Fred Flintstone', 'CS423', 30])
```



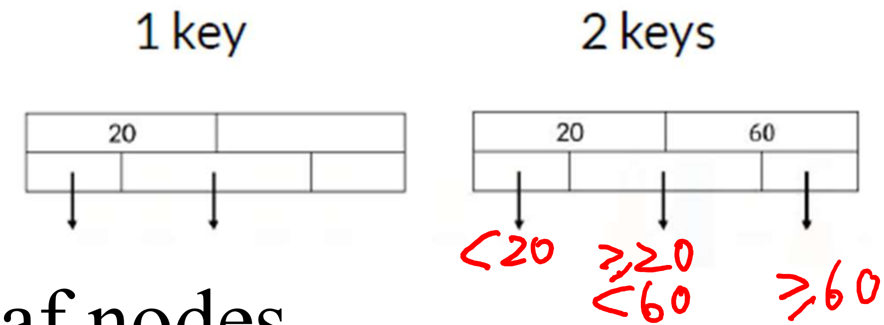
B^{+}-Tree

- Two types of index node
  - A directory node stores a set of index entries, each linking to another index node
  - A data entry node or a set of data entries, each linking to a data record or a data node (or data page)
  - The data entry nodes are chained to form a sequence
  - The entries in an index node are sorted
- A data node stores a set of records, which may or may not be sorted

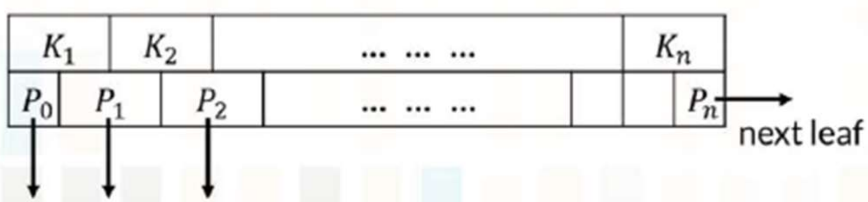


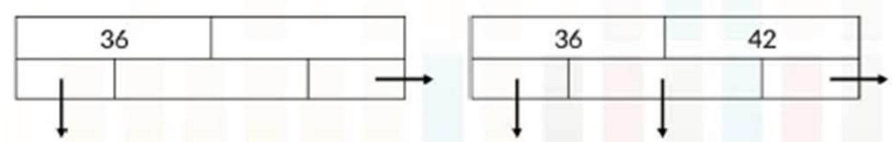
B+ Tree Nodes Zoom In

- directory nodes are internal nodes



- data entry nodes are leaf nodes



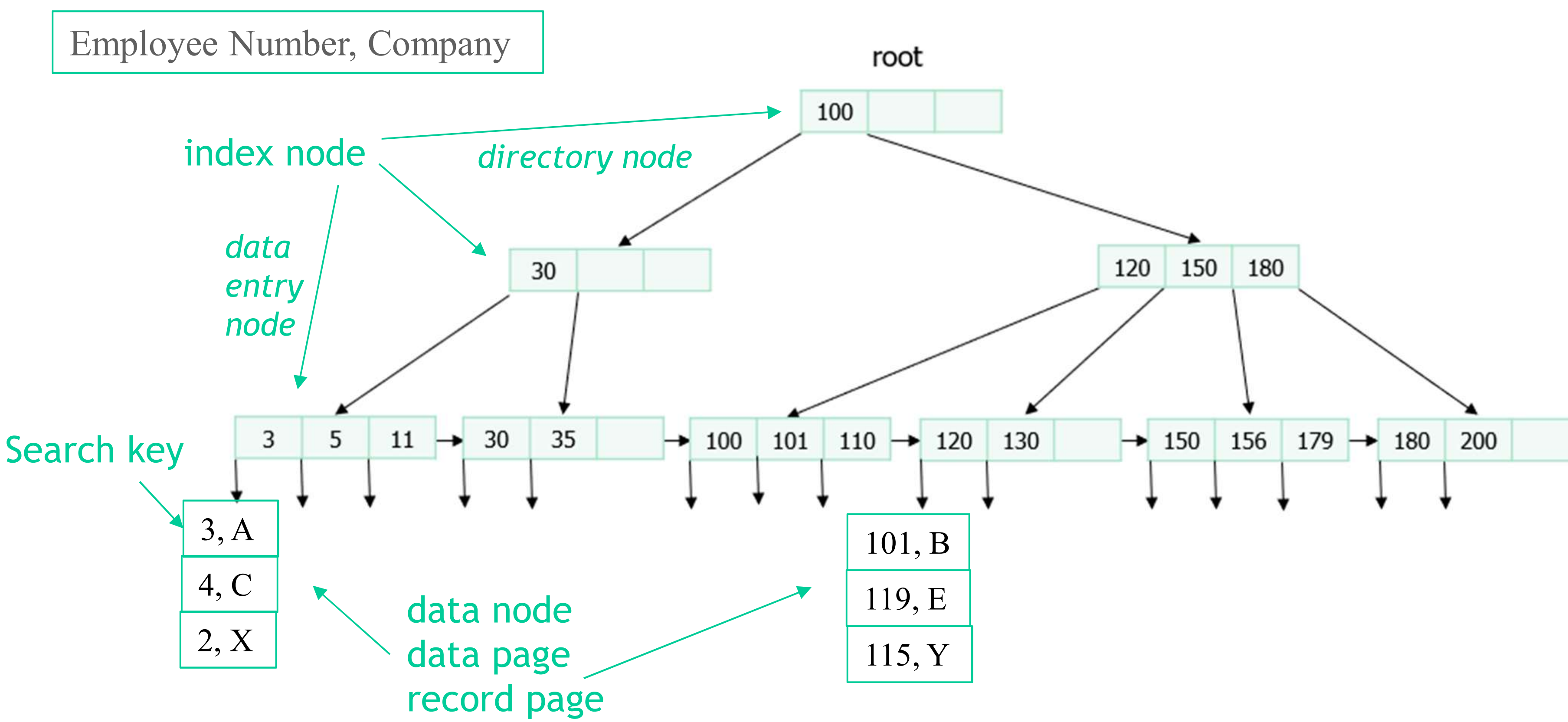


- Each node has $[\lfloor d \rfloor, 2d]$ keys (except root), where d is call the degree or order.
  - $d = 2.5$

B+-Tree

- Paged-tree: each node takes a whole page
  - For an index node, a page stores n keys and n+1 pointers, where n is the largest number subject to

    n * SearchKeySize + (n+1) * PointerSize <= PageSize
  - For a data node, a page stores m records, where m = PageSize/RecordSize
- B+-tree is featured being short (usually 3 or 4 layers) and fat (with a large number of fanouts)



B$^+$-Tree

- Paged-tree: each node takes a whole page
  - For an index node, a page stores n keys and n+1 pointers, where n is the largest number subject to

    8 4 1K

    n * SearchKeySize + (n+1) * PointerSize <= PageSize

  Eg. SearchKeySize = 8 bytes, PointerSize = 4 bytes, PageSize = 1 Kb

  $8n + 4(n + 1) \leq 1024 \qquad n \leq 85$

  Fanout(max)=n+1=86. That is, one node can point to 86 children.

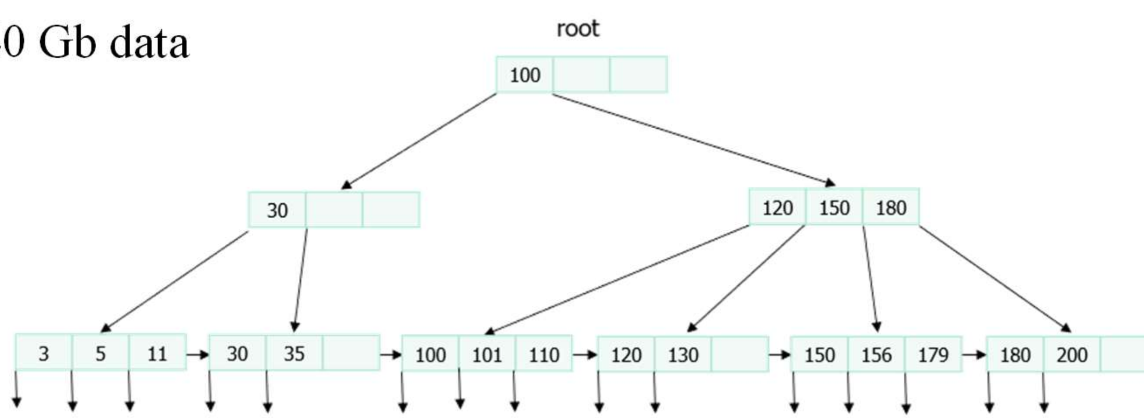
  How many data pages can be indexed by a tree with depth of 1? About 7 Mb

  depth of 3? About 54 Gb

  For depth of 3, what is the required space for indexing? about 640 Mb

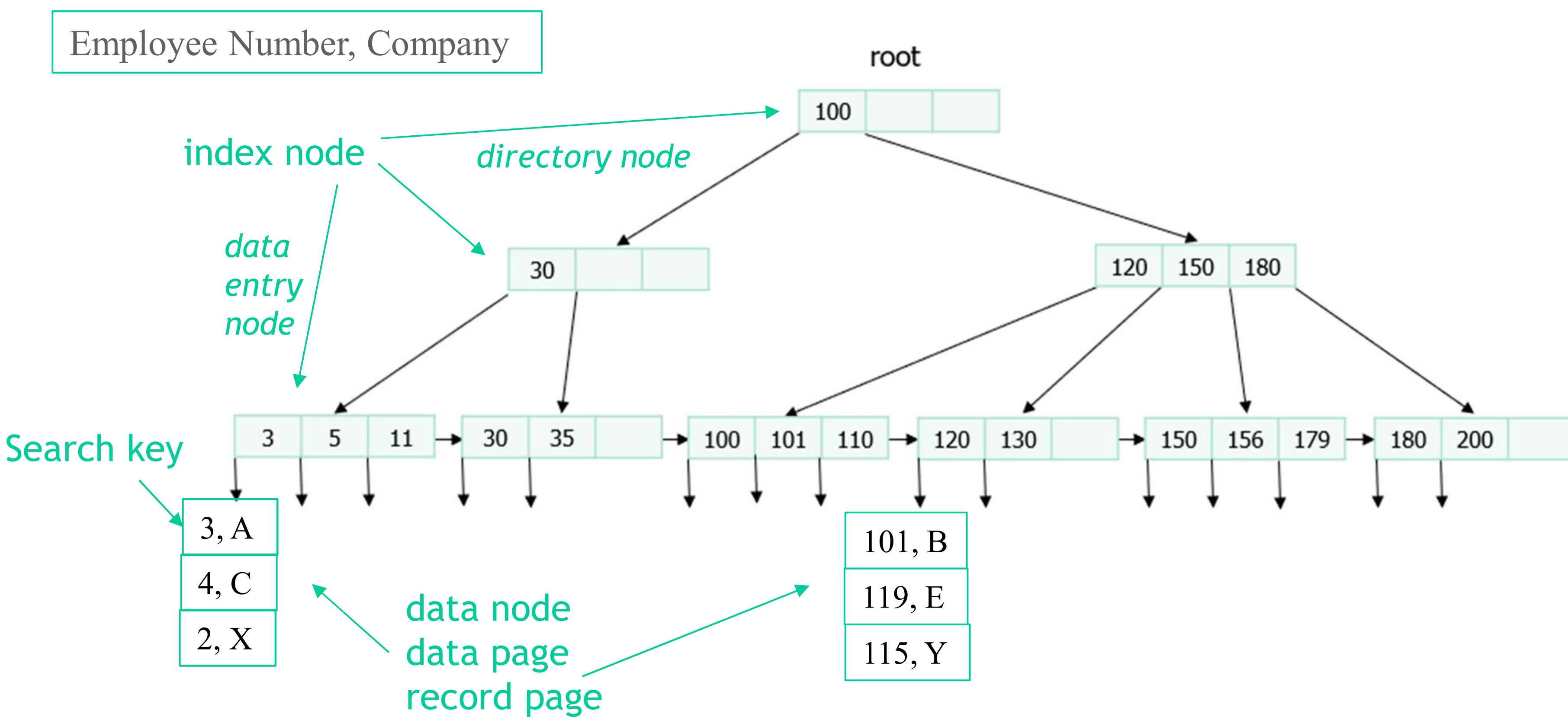
  What if pagesize= 4kb (config for x86-64)

  You can use 100 Mb to index 40 Gb data



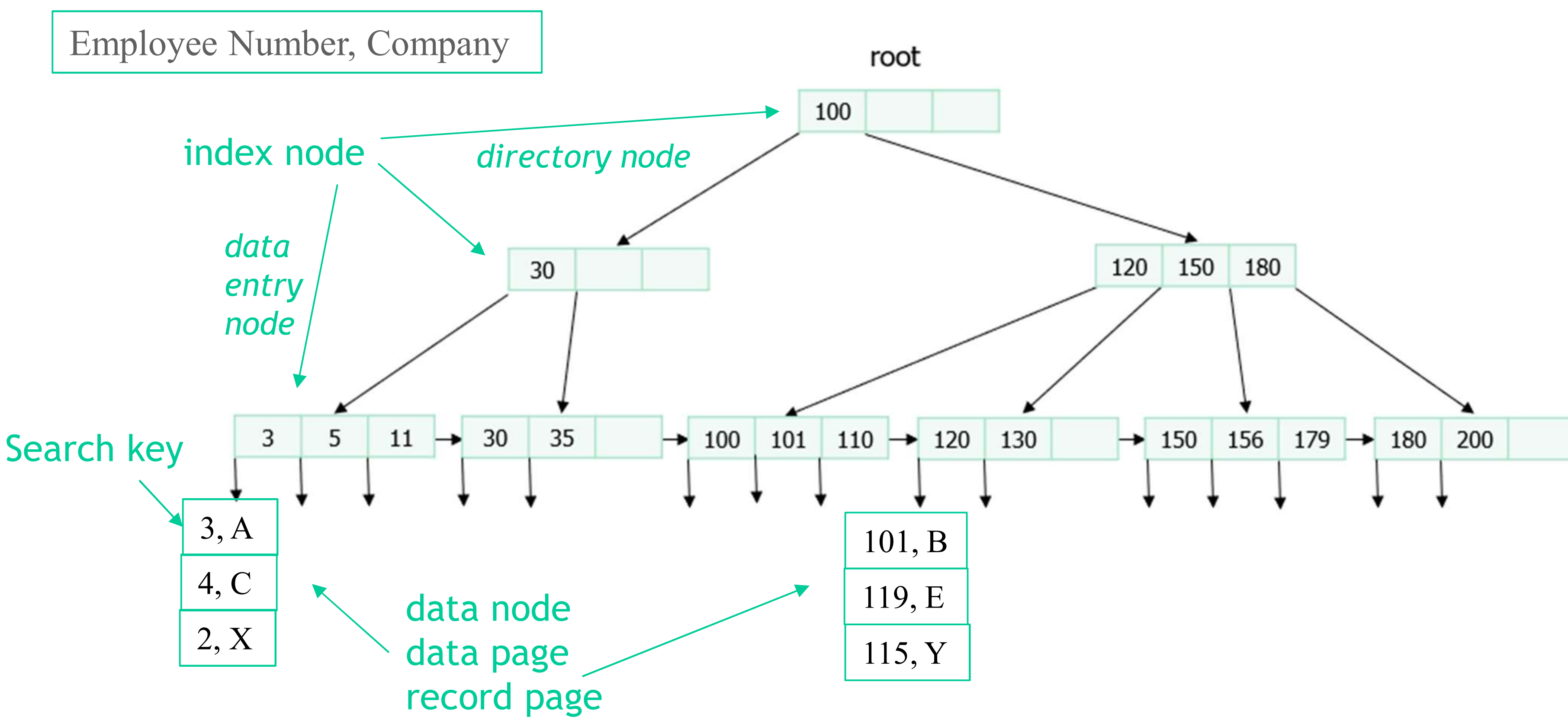
B^{+}-Tree

- Equality Search
  - Start from the root, find the pointer whose left key and right key values contains the search key value
  - Follow the pointer to the linked node, repeat until reaching a data node
  - Search the records in the data node and return those that match
- Range search (similar procedure)



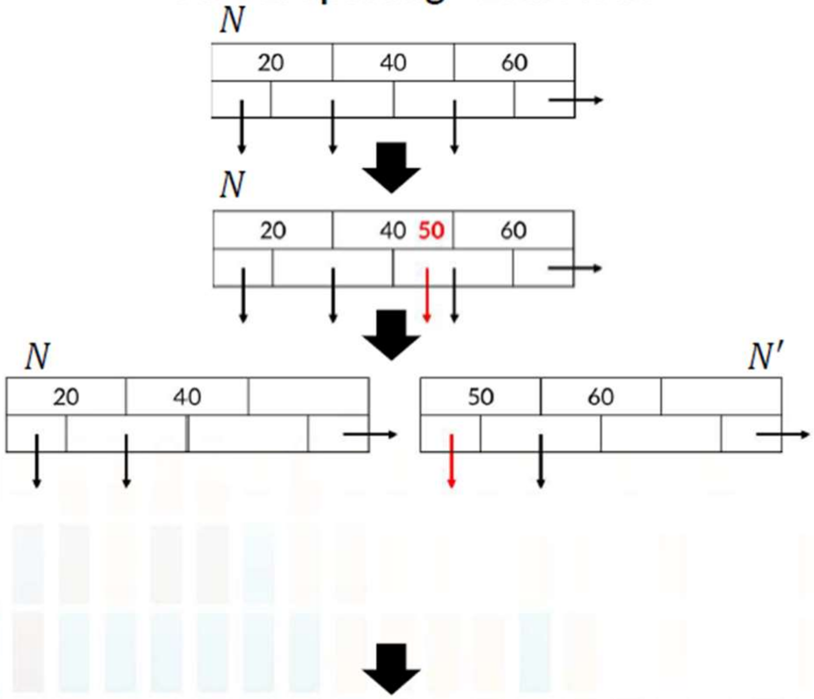
B^+-Tree

- Insert a record with a key value of k
  - Search the data entry node that contains k
  - Insert the record to the linked data node
  - The insertion may cause the data node to full, in which case the data node is split
  - The split may propagate up, causing the root node to split, in which case a new root is created
- Delete (reverse process of Insert)



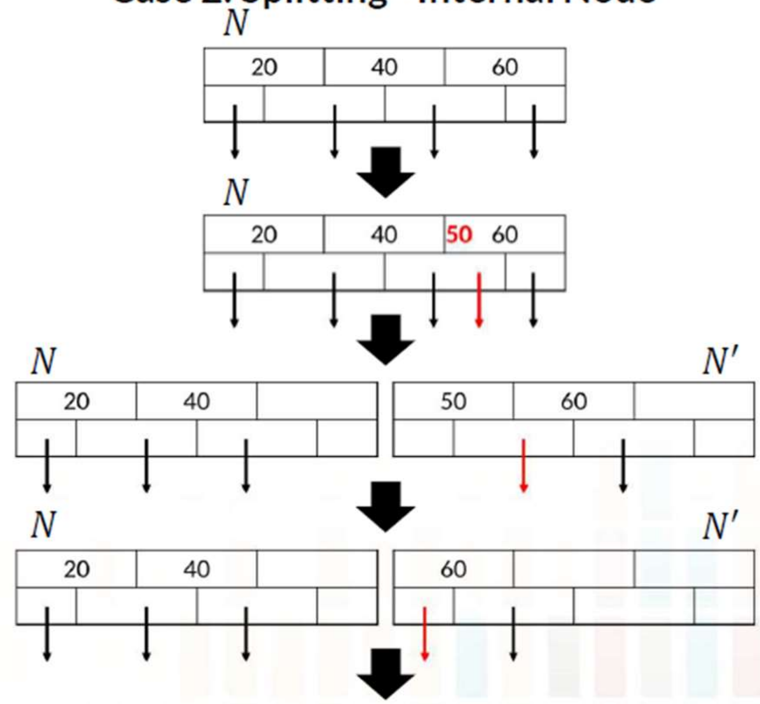
B-Tree Insertion/Splitting: Cases by Examples

Case 1: Splitting-- Leaf Node



Next: Insert P' (pointer of N') to parent.

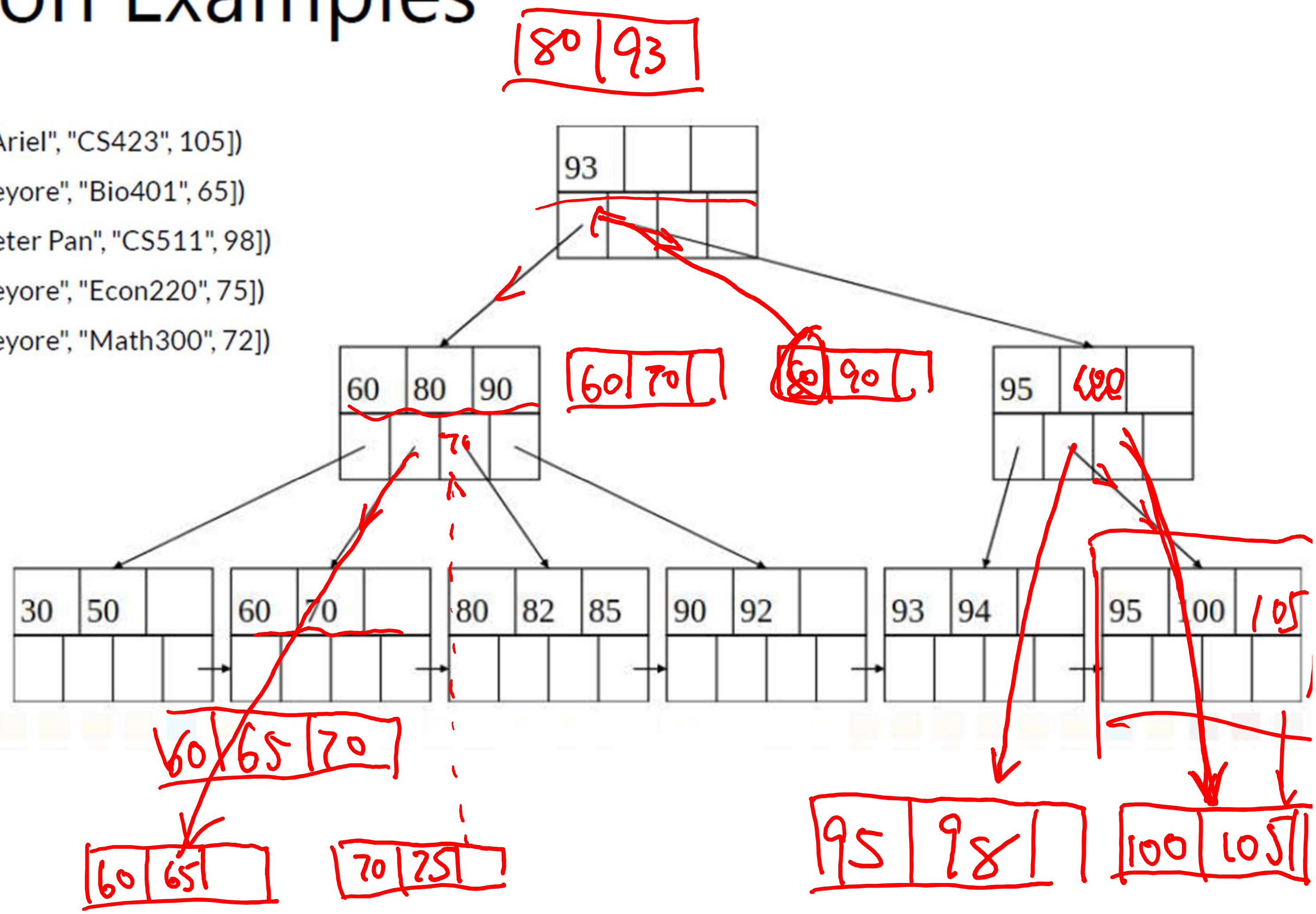
Case 2: Splitting– Internal Node



Next: Insert P' (pointer of N') to parent.

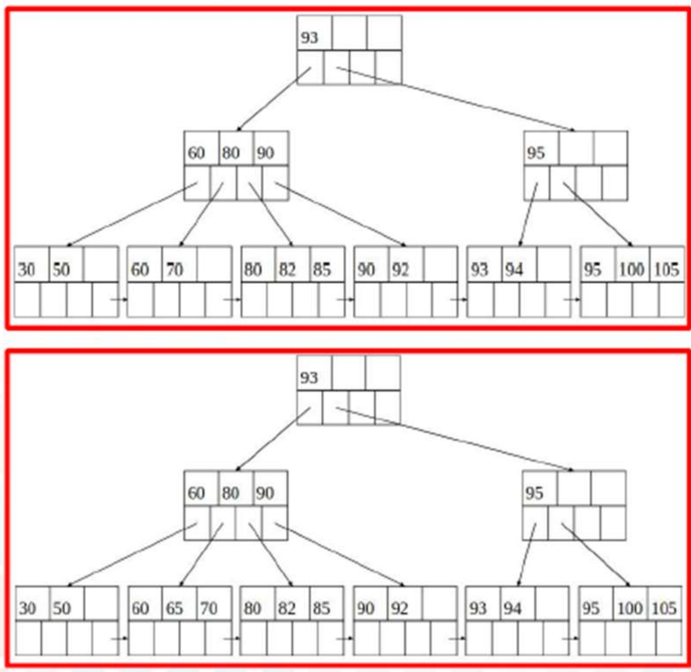
Insertion Examples

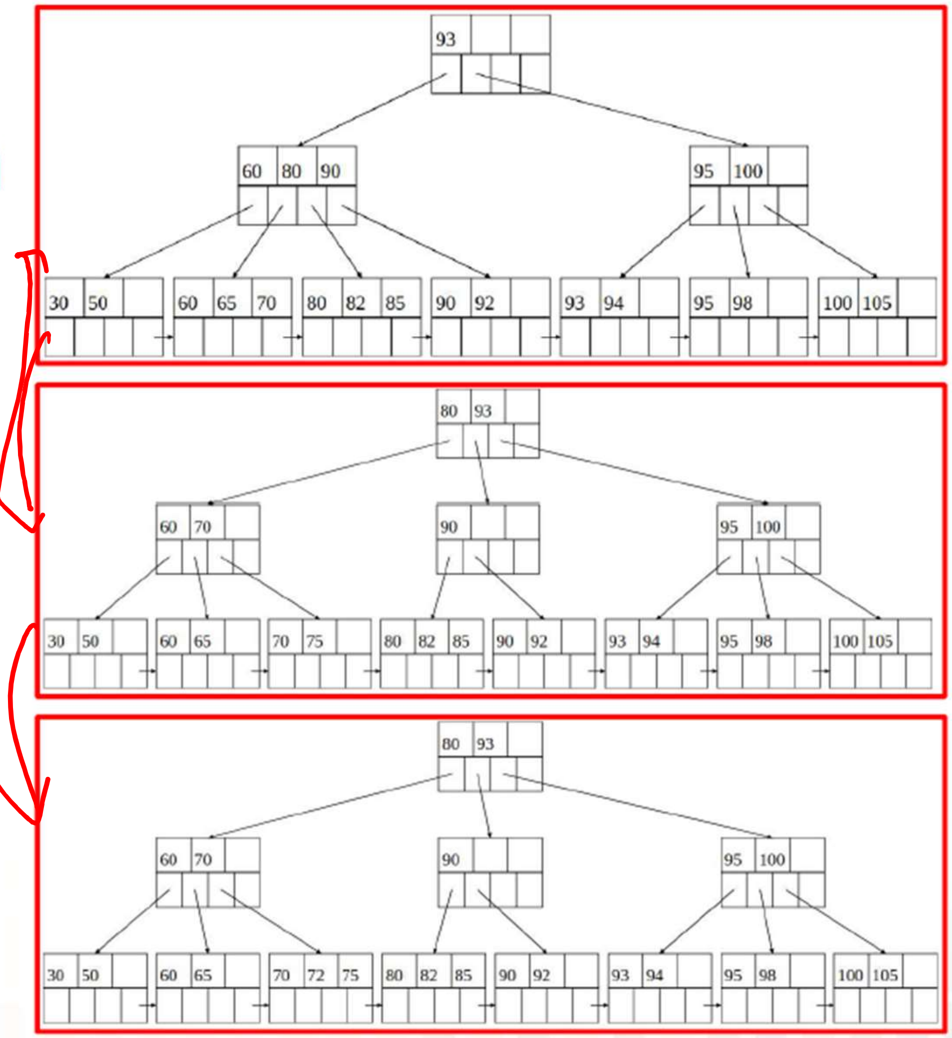
1. b.insert(105, ["Ariel", "CS423", 105])
2. b.insert(65, ["Eeyore", "Bio401", 65])
3. b.insert(98, ["Peter Pan", "CS511", 98])
4. b.insert(75, ["Eeyore", "Econ220", 75])
5. b.insert(72, ["Eeyore", "Math300", 72])



Results after Insertion

1. b.insert(105, ["Ariel", "CS423", 105])
2. b.insert(65, ["Eeyore", "Bio401", 65])
3. b.insert(98, ["Peter Pan", "CS511", 98])
4. b.insert(75, ["Eeyore", "Econ220", 75])
5. b.insert(72, ["Eeyore", "Math300", 72])

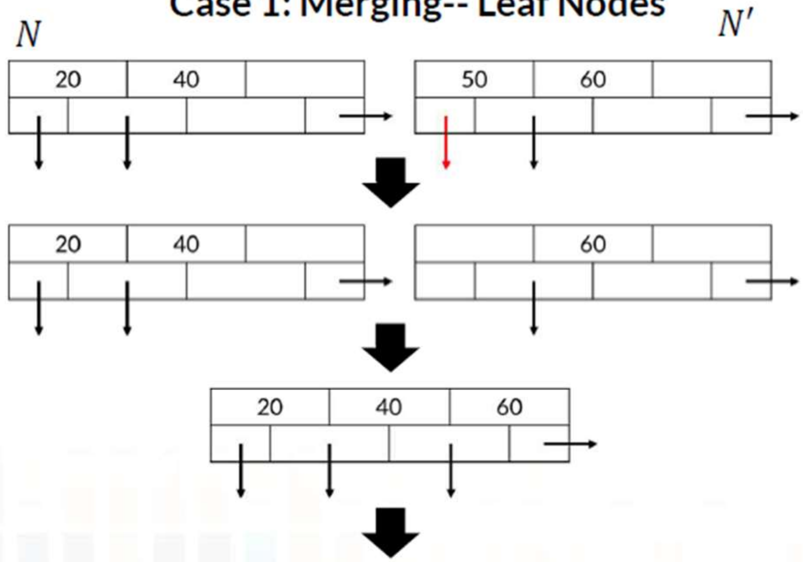




Results of the insertion examples

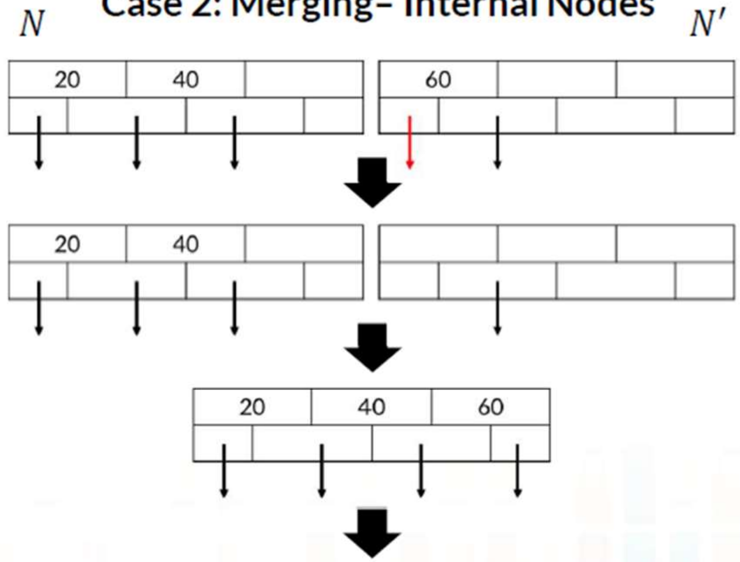
B-Tree Deletion/Merging: Cases by Examples

Case 1: Merging-- Leaf Nodes



Next: Remove P' (pointer of N') from parent.

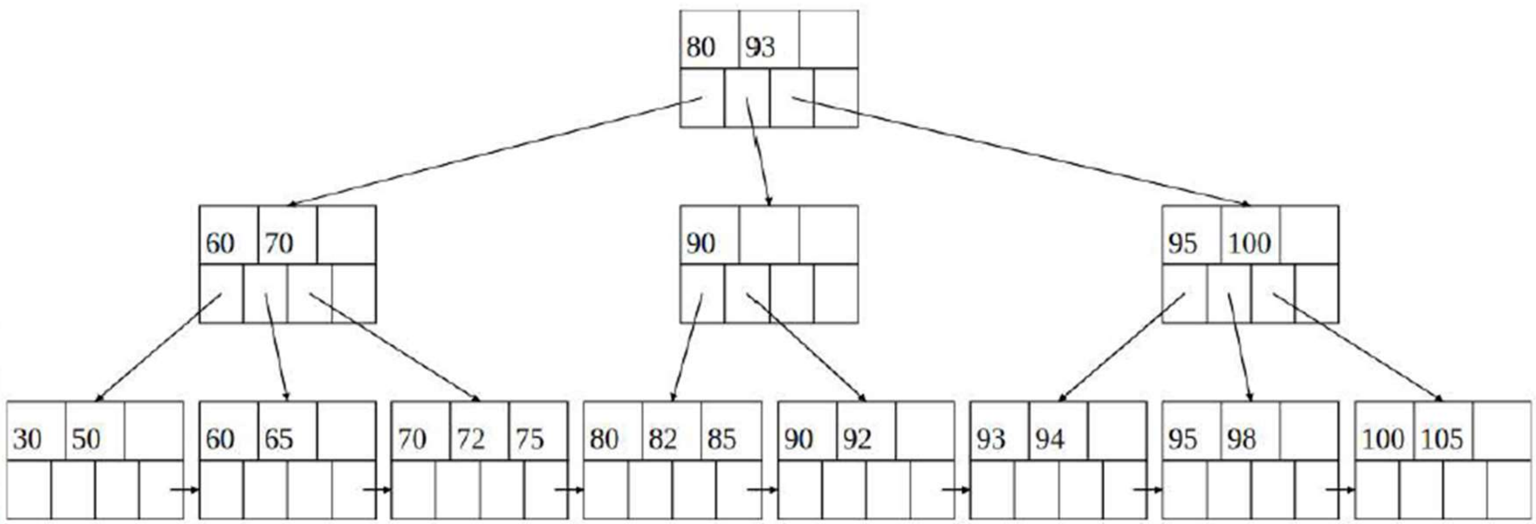
Case 2: Merging– Internal Nodes



Next: Remove P' (pointer of N') from parent.

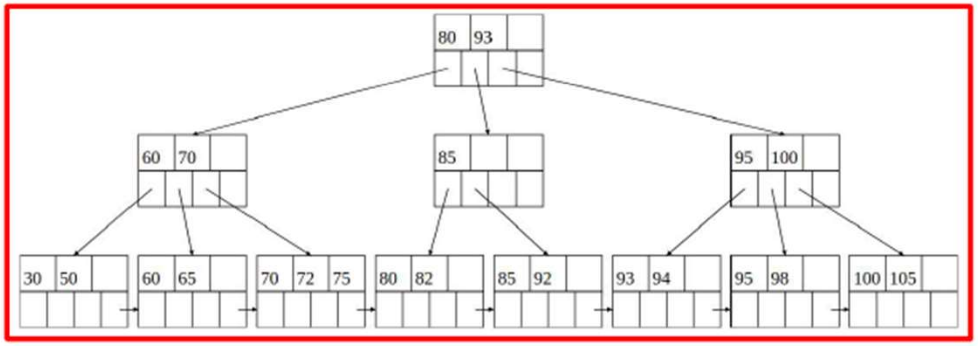
Deletion Examples

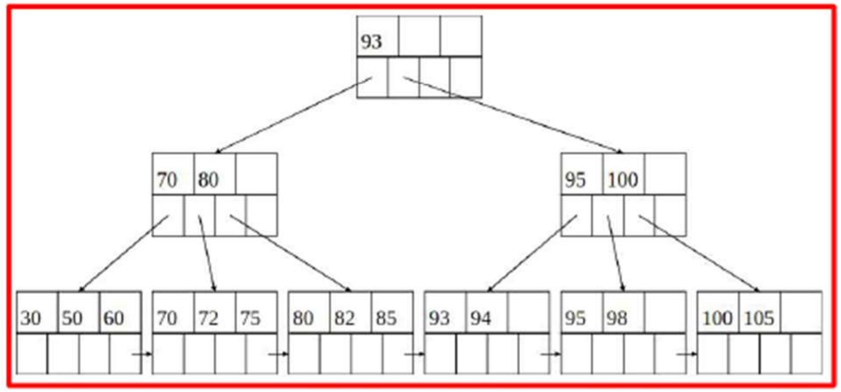
1. b.delete(90)
2. b.delete(92)
3. b.delete(65)

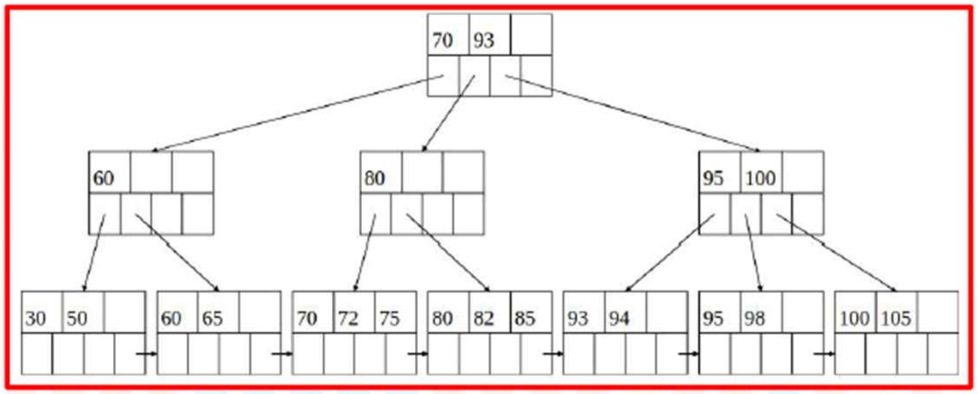


Results after Deletions

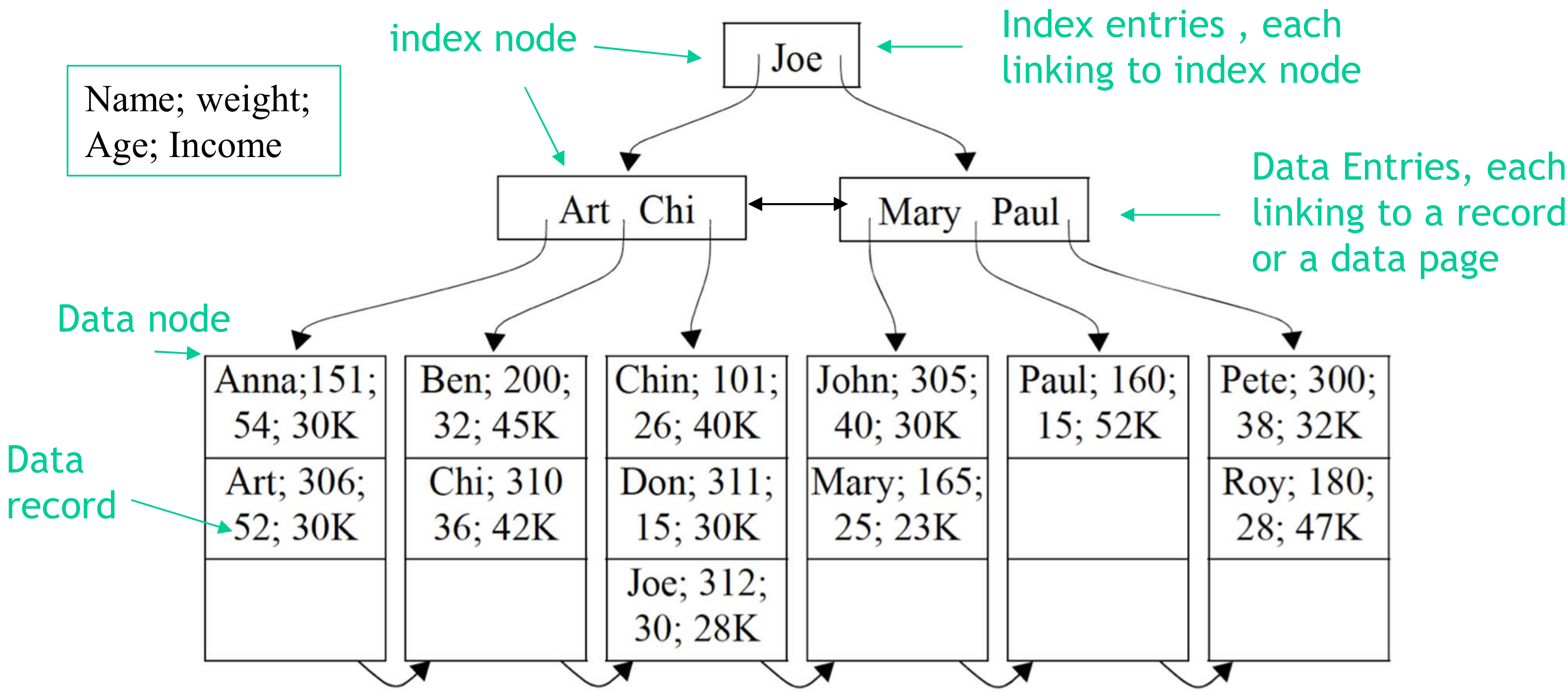
1. b.delete(90)
2. b.delete(92)
3. b.delete(65)







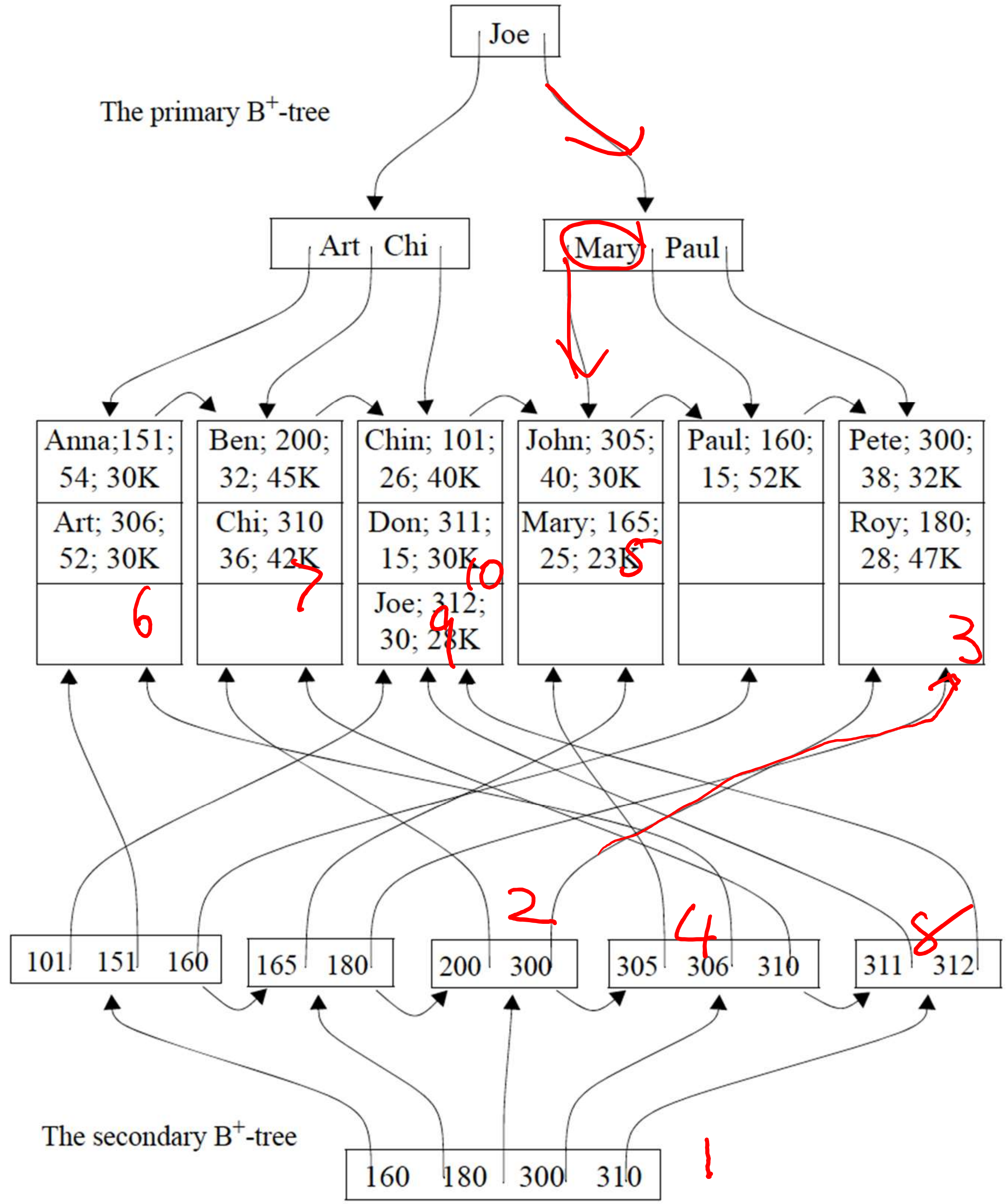
B$^+$-Tree: Another example



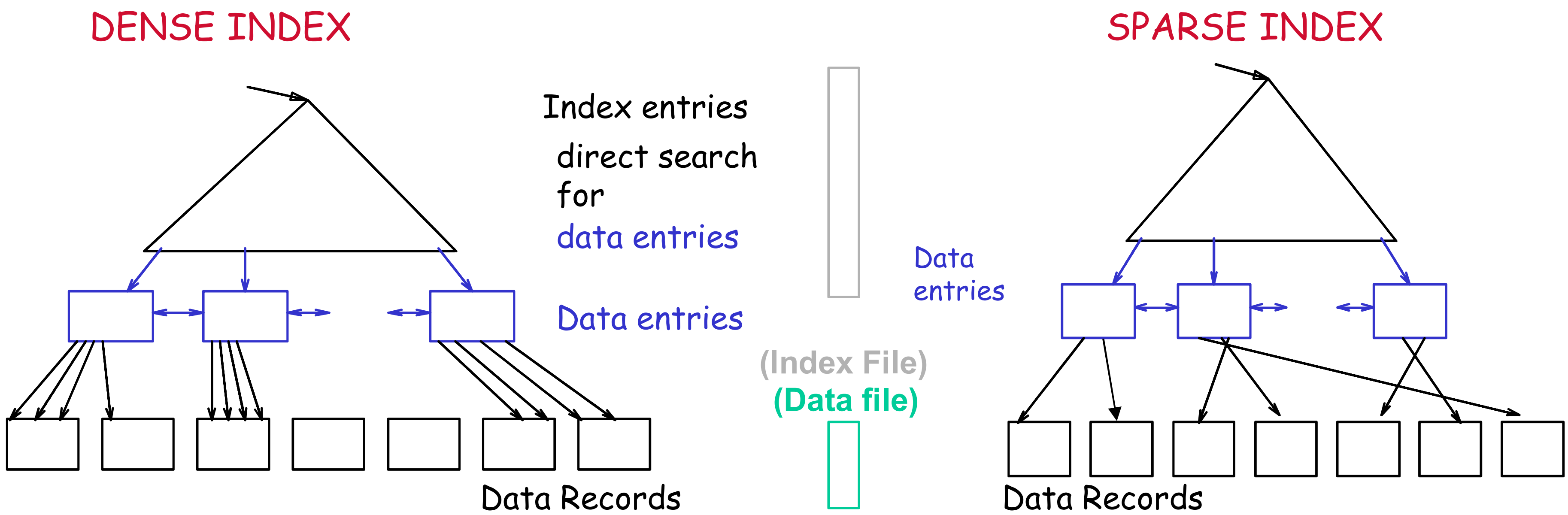
1. Primary index on names
   - can be sorted
2. Secondary index on ID
   - cannot sorted, otherwise would have duplicated the records

Type of B+-tree

- Clustered/unclustered
- Dense/sparse



Dense or Sparse

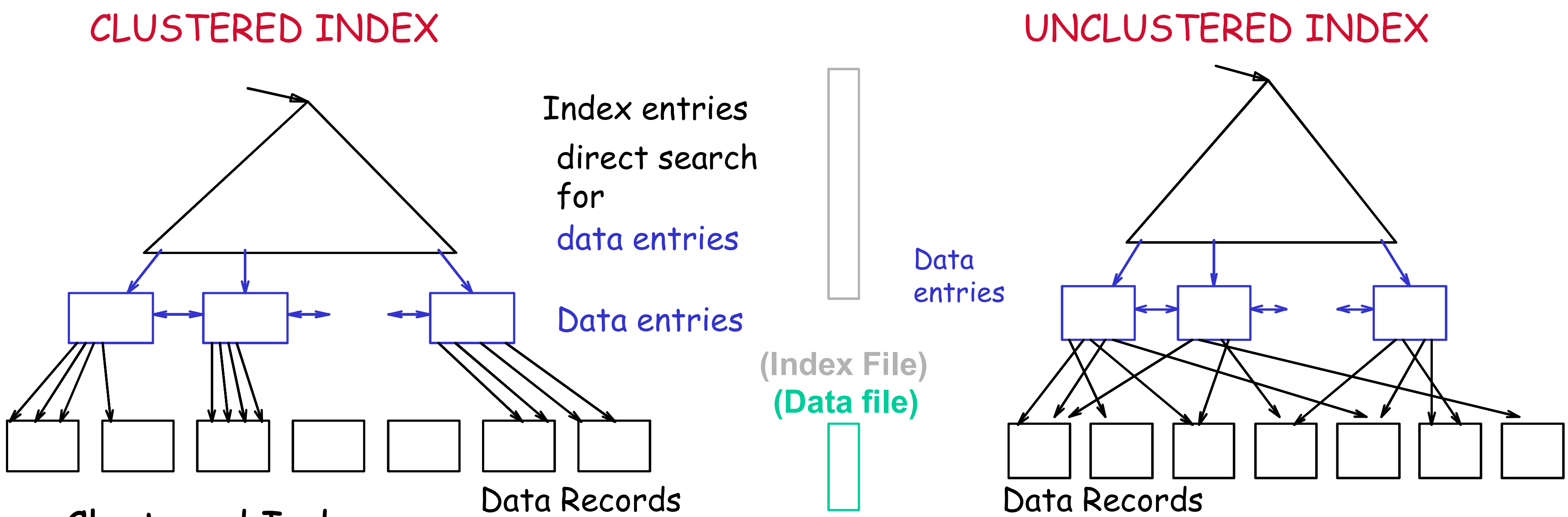


- Dense: At least one data entry per data record
- Sparse: At least one data entry per block/page

Pros and Cons:

- Dense: less space-efficient, but great for both equality and range search
- Sparse: more space-efficient, but need sequential search within a page

Clustered or Unclustered



Clustered Index :

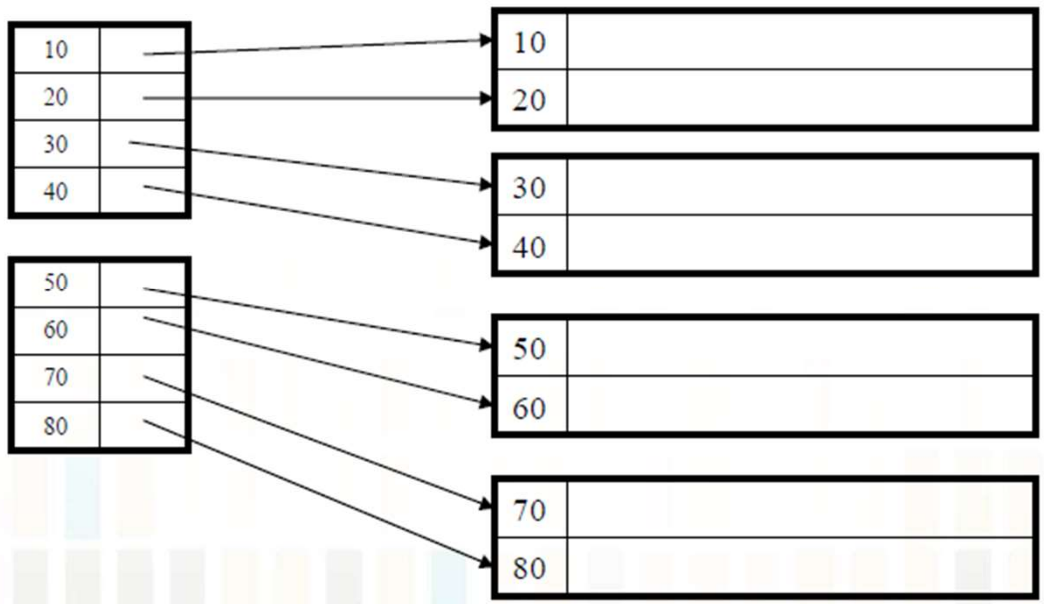
- The ordering of data records is organized the same as or close to the ordering of data entries in the index
  - Sparse index is always clustered, e.g., alternative 1 can be regarded as sparse (why?)
  - A clustered index does not have to be sparse (why?)

Pros and Cons:

- Clustered: maintenance cost high, but great for range search
- Unclustered: low maintenance cost, but high retrieval cost
  - Retrieving one record may need to load one page

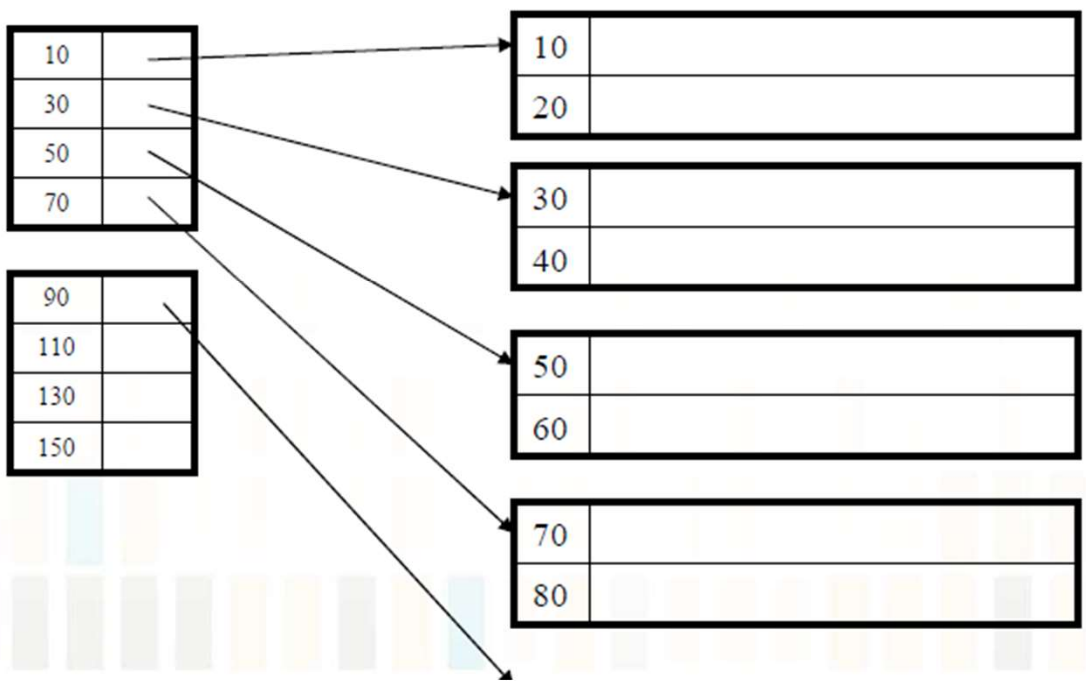
Clustered, Dense Index

- Clustered: File is sorted on the index attribute.
- Dense: Keys cover all values.



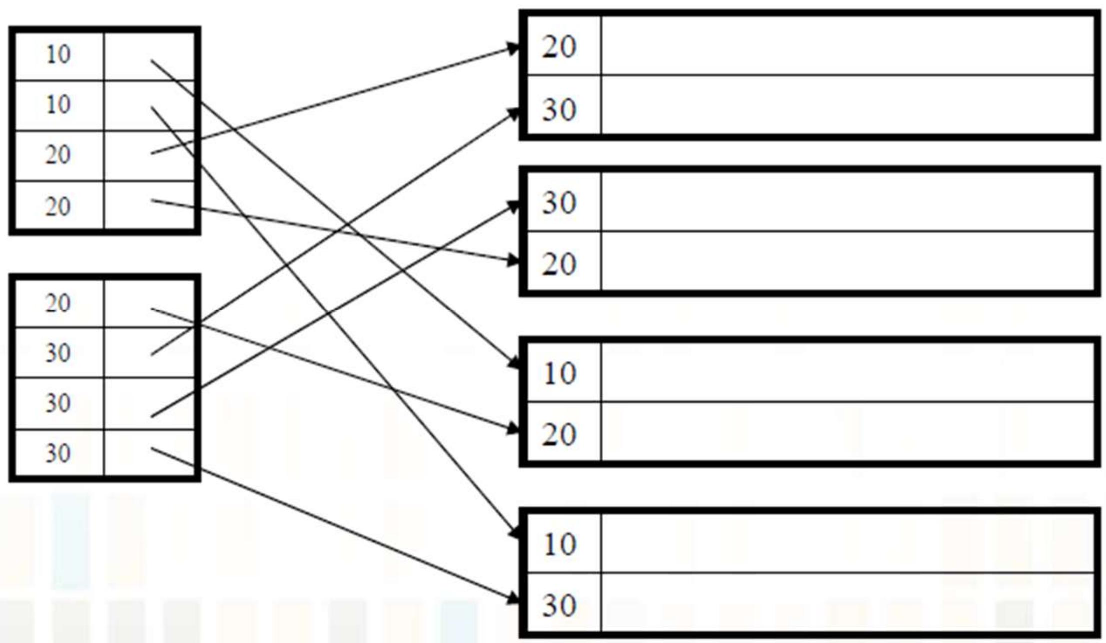
Clustered, Sparse Index

- Sparse index: one key per data block.

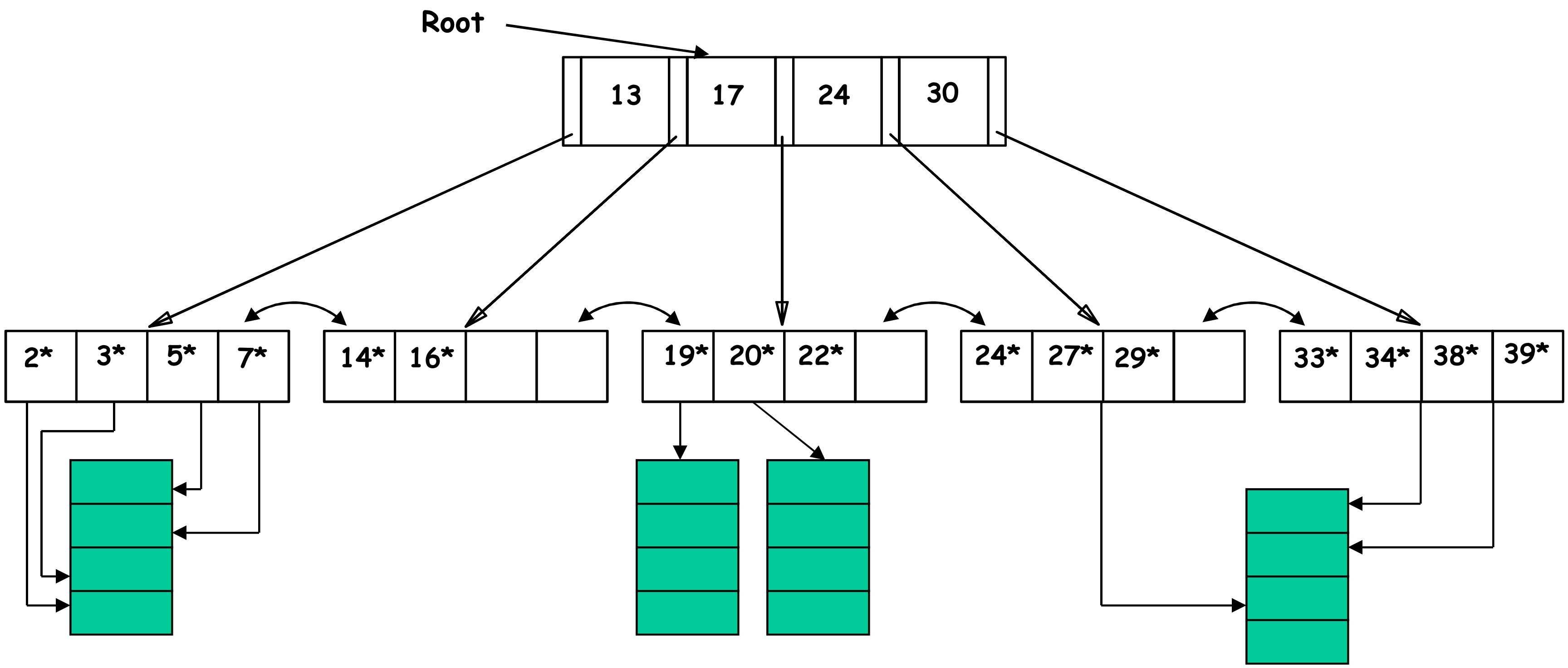


Unclustered Indexes: Always Dense

- Often for indexing other attributes than primary key
- Always dense (why ?)



Dense/Sparse Clustered/Unclustered



Our textbook as example--Indexes?

- How many indexes? Where?
- What are keys? What are records?
- Clustered?
- Dense?

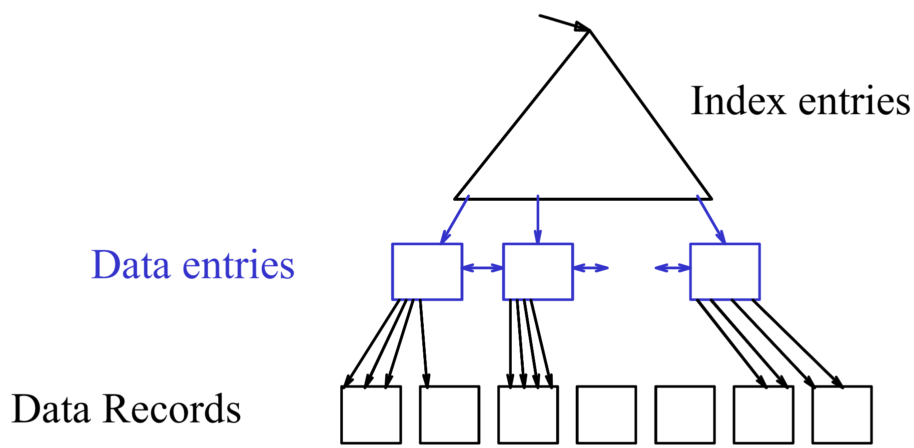
Tree Index on R.A

$$\sigma_{R.A\,=\,value}(\mathrm{R})$$

Selection Cost =

cost of traversing from the root to the leaf +
cost of retrieving the pages in the sequence set +
cost of retrieving pages containing the data records.

- Need to know
  - Clustered or unclustered
  - Dense or sparse



B^+Tree Index on R.A

$$\sigma_{R.A = value}(\mathrm{R})$$

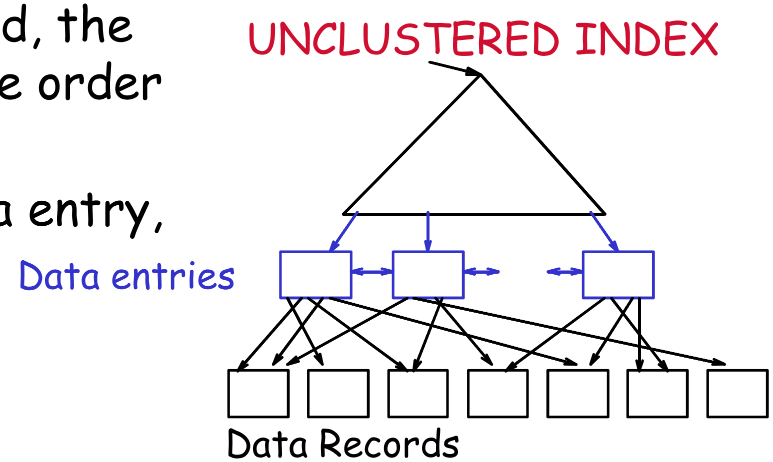
- dense, unclustered
- Size of data entry = 20 bytes;
- Page size=4K bytes; 96 bytes are reserved
- Total number of records = 100,000; record size = 40 bytes
- Reduction Factor = 0.1
  - #matching entries/#total entries

Total Cost =

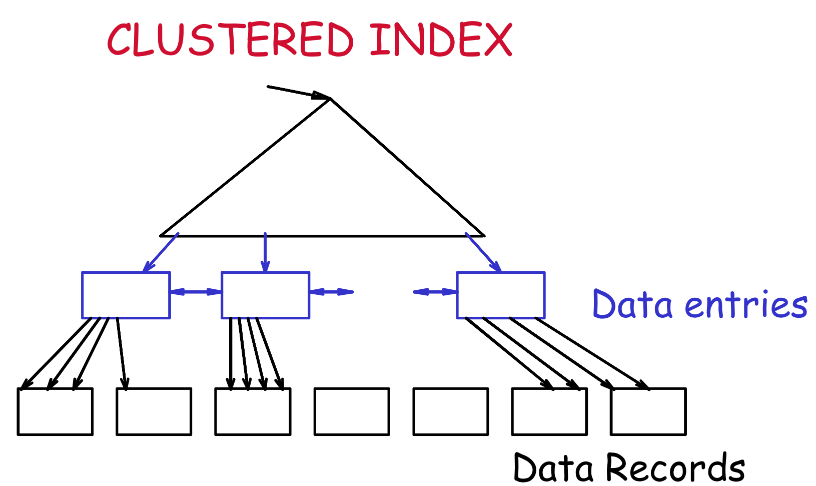
Cost of traversing from the root to the leaf (assume 4 I/Os) +
Cost of retrieving the pages in the sequence set +
the cost of retrieving pages containing the data records

- Size of data entry = 20 bytes;
- Page size=4K bytes; 96 bytes are reserved
- Total number of records = 100,000; record size = 40 bytes
- Reduction Factor = 0.1

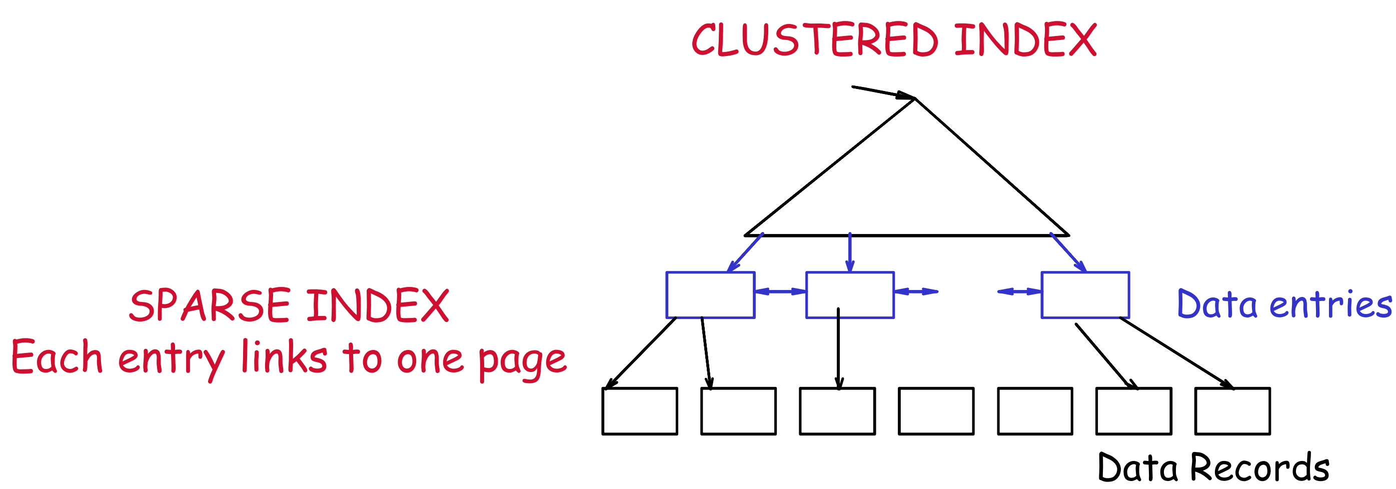
- B^+tree, dense, unclustered
- I/O cost of retrieving pages of qualifying data entries
  - Matching data entries: 0.1*100,000=10,000 entries
  - #Date entries per page: $\left\lfloor \frac{4096-96}{20} \right\rfloor = 200$
  - Pages of matching data entries = 10,000/200 = 50 pages
- I/O cost of retrieving qualifying tuples
  - 10,000 pages since the index is unclustered, the qualifying tuples are not always in the same order as the data entries.
  - In the worst case, for each qualifying data entry, one I/O is needed
- Total I/O Cost = 4+ 50+10,000 pages



- B^+tree, dense, clustered
- I/O cost of retrieving pages of qualifying data entries
  - Matching data entries: 0.1*100000=10000 entries
  - #Date entries per page: $\left\lfloor \frac{4096-96}{20} \right\rfloor = 200$
  - #Pages of matching data entries = $\left\lceil \frac{10000}{200} \right\rceil = 50$
- I/O cost of retrieving qualifying tuples
  - #Matching tuples: 10000
  - Since the index is dense and clustered, the qualifying tuples are also clustered
  - # pages: 10000/100=100 due to (4096-96)/40=100 tuples per page
- Total I/O Cost = 4+ 50+100 pages

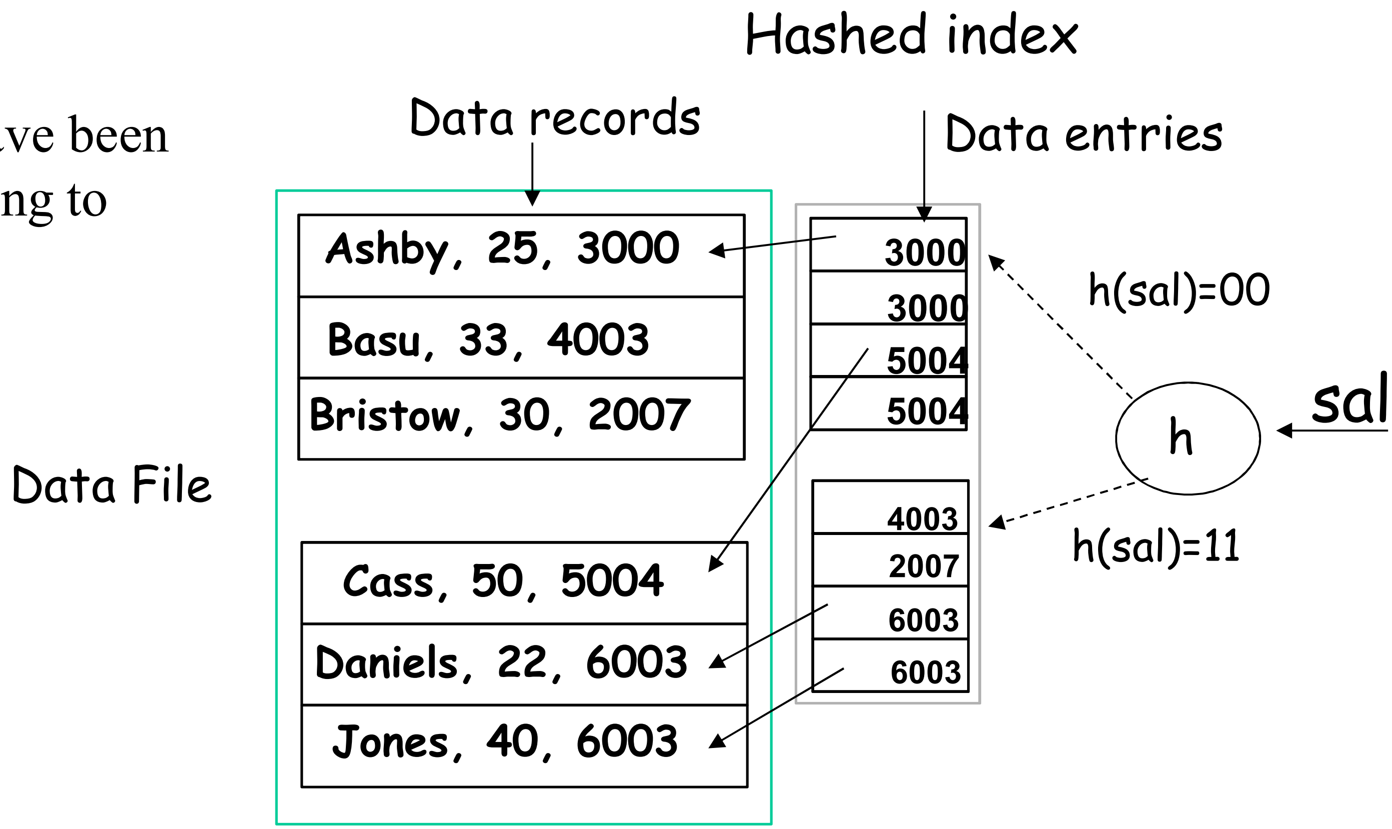


- B^+tree, sparse (must be clustered)
- I/O cost of retrieving qualifying tuples
  - #Matching tuples: 0.1*100,000=10,000
  - Since the index is clustered, the qualifying tuples are also clustered
  - # pages: $\left\lceil \frac{10000}{100} \right\rceil$ due to 100 tuples per page
- I/O cost of retrieving pages of qualifying data entries
  - Matching data pages: 100
  - #Data entries per page: $\left\lfloor \frac{4096-96}{20} \right\rfloor = 200$
  - #Pages of matching data entries = $\left\lceil \frac{100}{200} \right\rceil$ =1 page
- Total I/O Cost = 4+ 1+100 pages



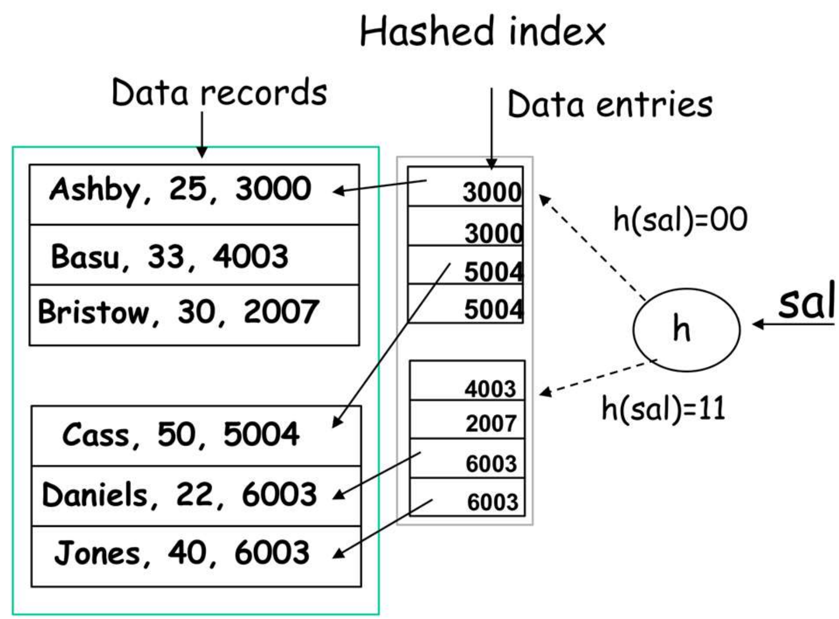
Hash-based Index $\sigma_{R.A = value}(R)$

The records have been sorted According to names.



I/O cost = cost for retrieving the matching data entries + cost for retrieving the qualifying tuples

- Total number of records: 100,000
- Reduction Factor: 0.01 (the rate of the records that satisfy the query condition over the total number of records)
- Cost for searching the matching data entry: 1.2 I/O
- Each page holds 1000 of data entries



I/O cost = cost for retrieving the matching data entries

+ cost for retrieving the qualifying tuples

- I/O cost of retrieving pages of matching data entries
  - Matching data entries: 0.01*100,000
  - Number of pages of matching data entries: 0.01*100,000 / 1,000 = 1 ~~10~~ pages
  - I/O cost = 1 ~~10~~ *1.2 = ~~12 I/Os~~ $\lceil 1.2 \rceil$ 2
- I/O cost for retrieving the qualifying tuples = 1,000 I/Os (or the maximum pages) min(1000, # page)
- Total cost = 12+1,000 = 1,012 I/Os

Factors to Consider

$$\sigma_{R.A \,\mathrm{op}\, value}(R)$$

- No index
  - unsorted data
  - sorted data
- Index
  - tree index
    - clustered/unclustered
    - dense/sparse
  - hash-based index