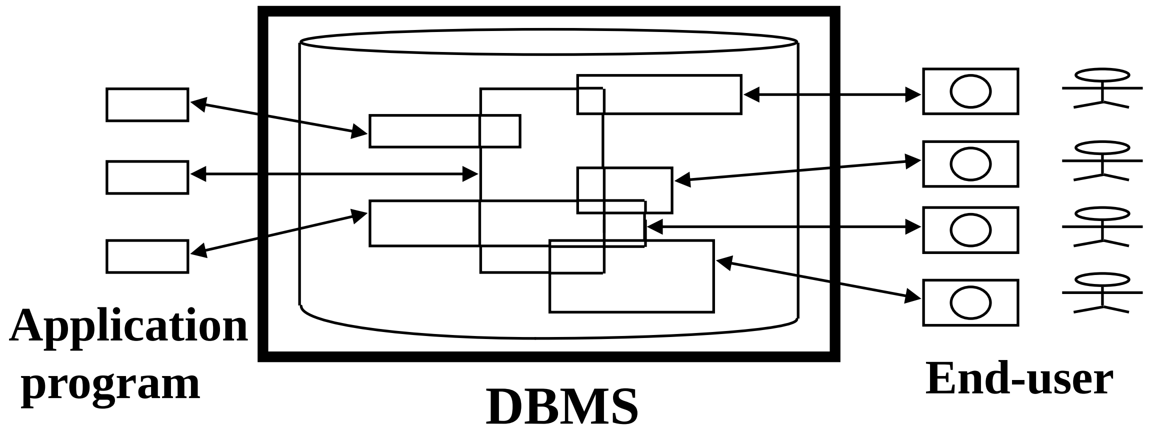
B+ Trees Indexing

Dr. Shaheen Khatoon



Indexes: Review

- Indexing mechanisms used to speed up access to desired data.
- Search Key - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called index entries) of the form

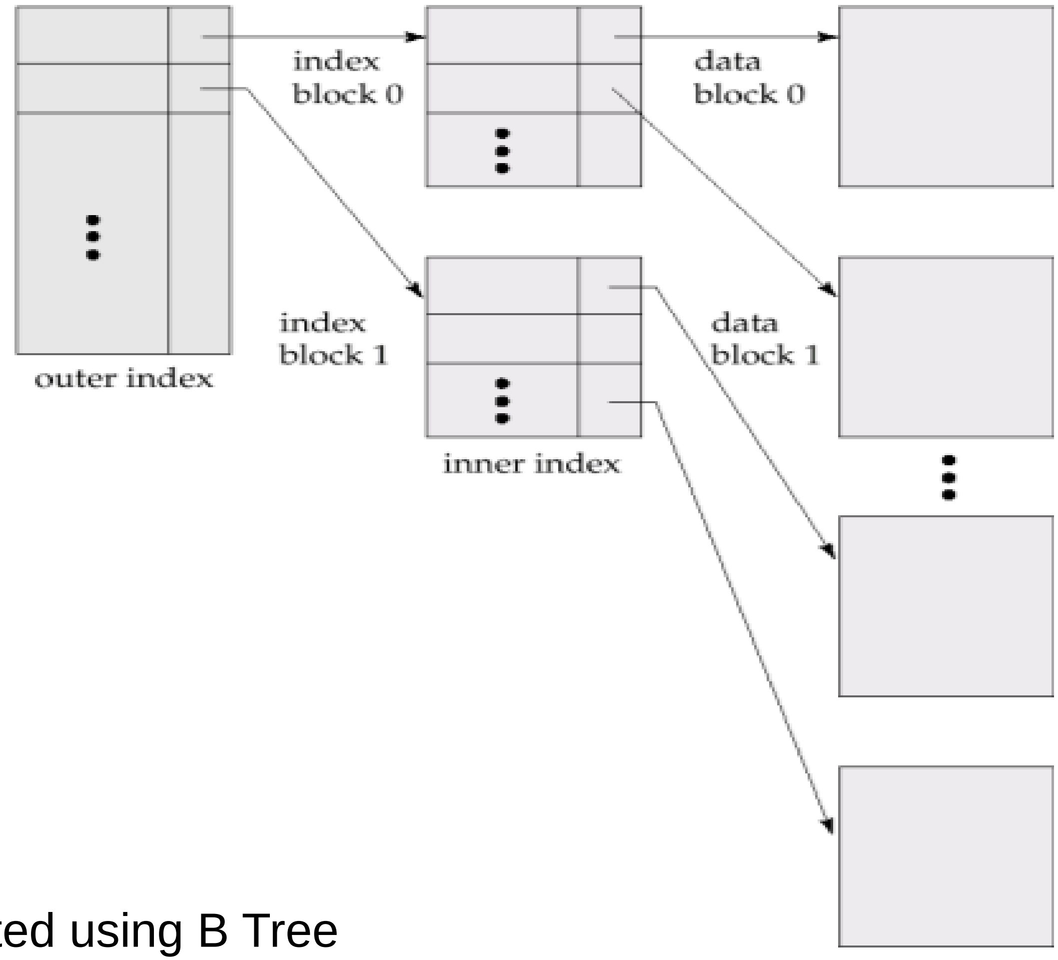
search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of implementations:
  - Tree indices: search keys are stored in sorted order!
  - Hash indices: search keys are distributed uniformly across "buckets" using a "hash function".

Multilevel Index: review

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, 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.

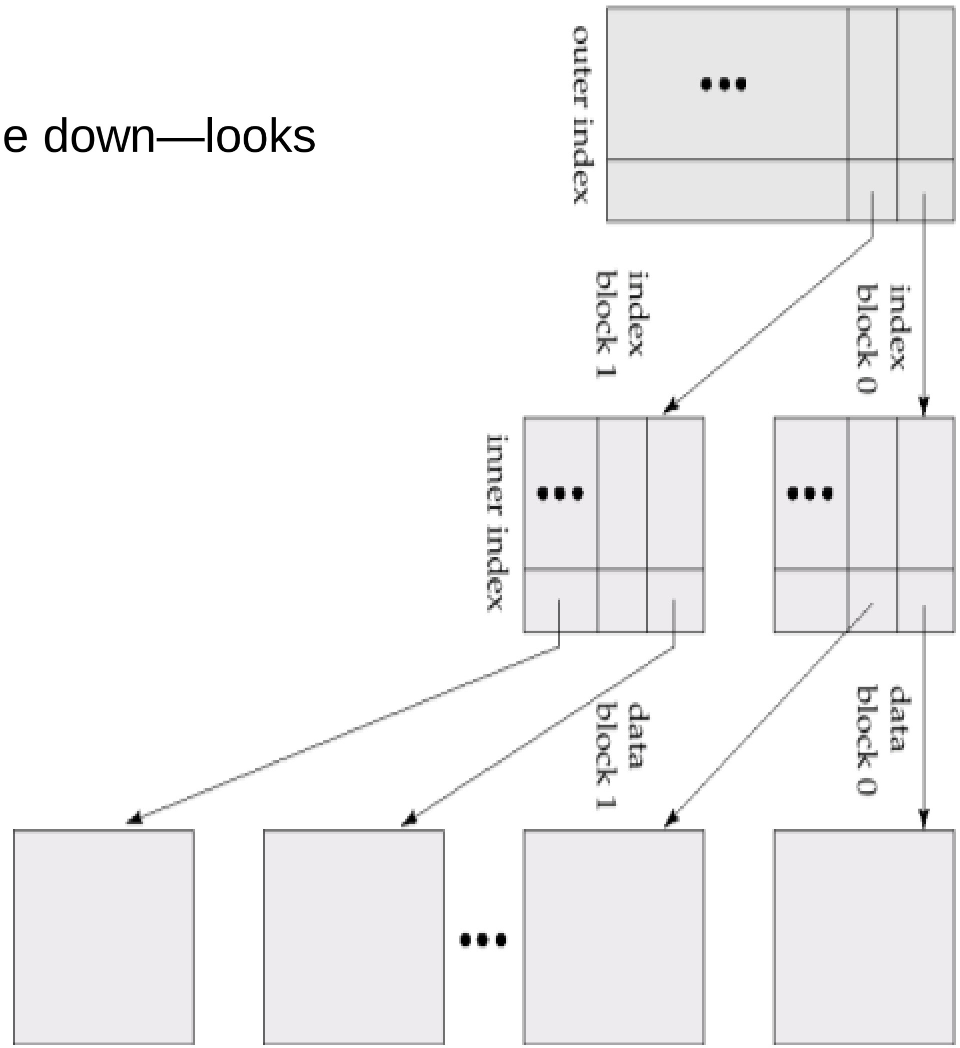
Multilevel Index (Cont.)



Implemented using B Tree

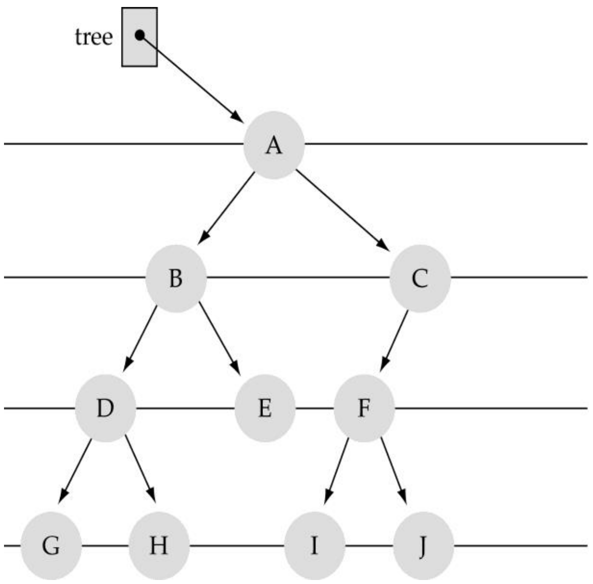
Multilevel Index (Cont.)

Turn multi-lev index upside down—looks like a tree

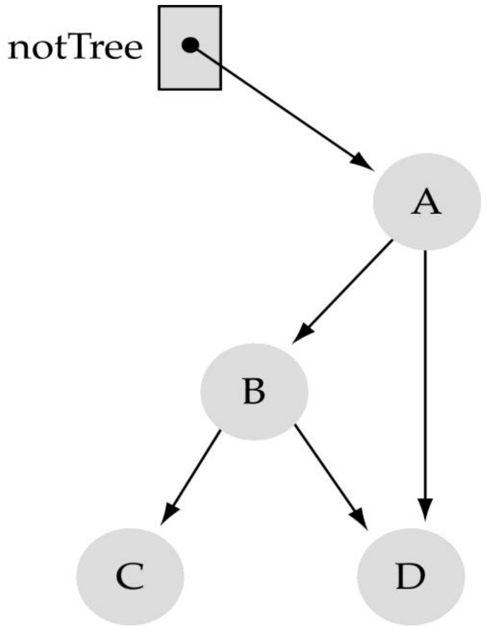


What is a binary tree?

- *Property 1:* each node can have up to two successor nodes.



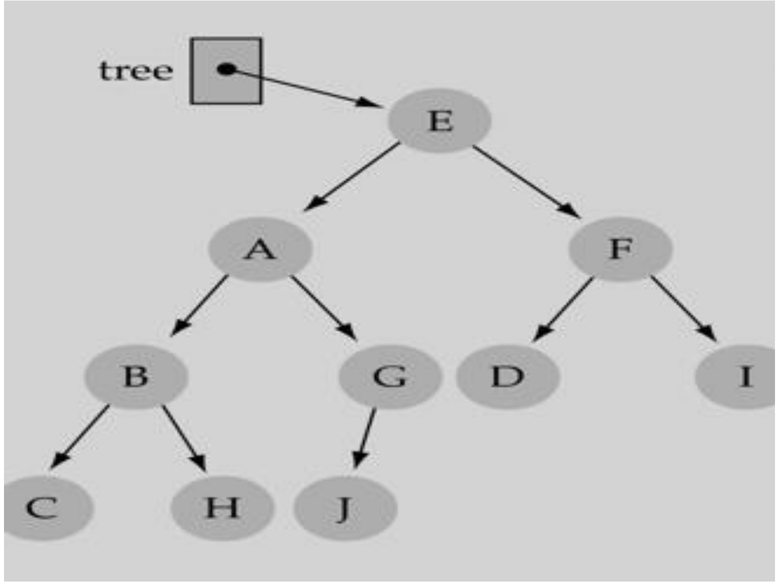
- *Property 2:* a unique path exists from the root to every other node



Not a valid binary tree

Some terminology

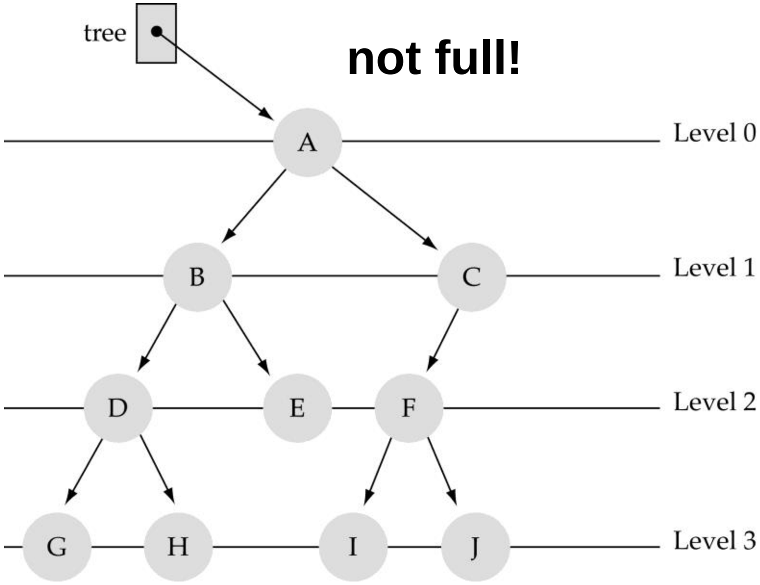
- The successor nodes of a node are called its *children*
- The predecessor node of a node is called its *parent*
- The "beginning" node is called the *root* (has no parent)
- A node without *children* is called a *leaf*



Some terminology (cont'd)

- Nodes are organize in levels (indexed from 0).
- **Height of a tree h**: #levels = L
- **Full tree:** every node has exactly two children ***and*** all the leaves are on the same level.

not full!



B + tree Indexing: Concept

- Each node can have up to **n** children
  - where n is fan out (FO), maximum degree or order of tree.
  - Each node can stores n pointer and (n-1) search values

- Allow efficient and fast exploration at the expense of using slightly more space.

- Support more efficiently queries like: SELECT * FROM R WHERE a = 11

- SELECT * FROM R WHERE 0<= b and b<42

B+ Tree node Structure

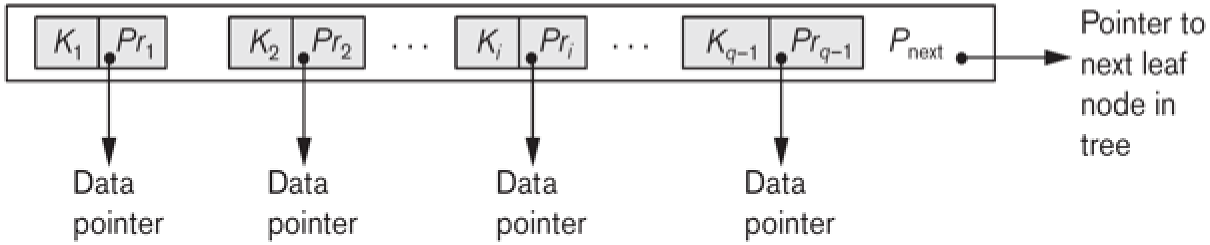
- Typical node Structure

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

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records (for leaf nodes).
- The search-keys in a node are ordered $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
- Each node stored in **one disk block**

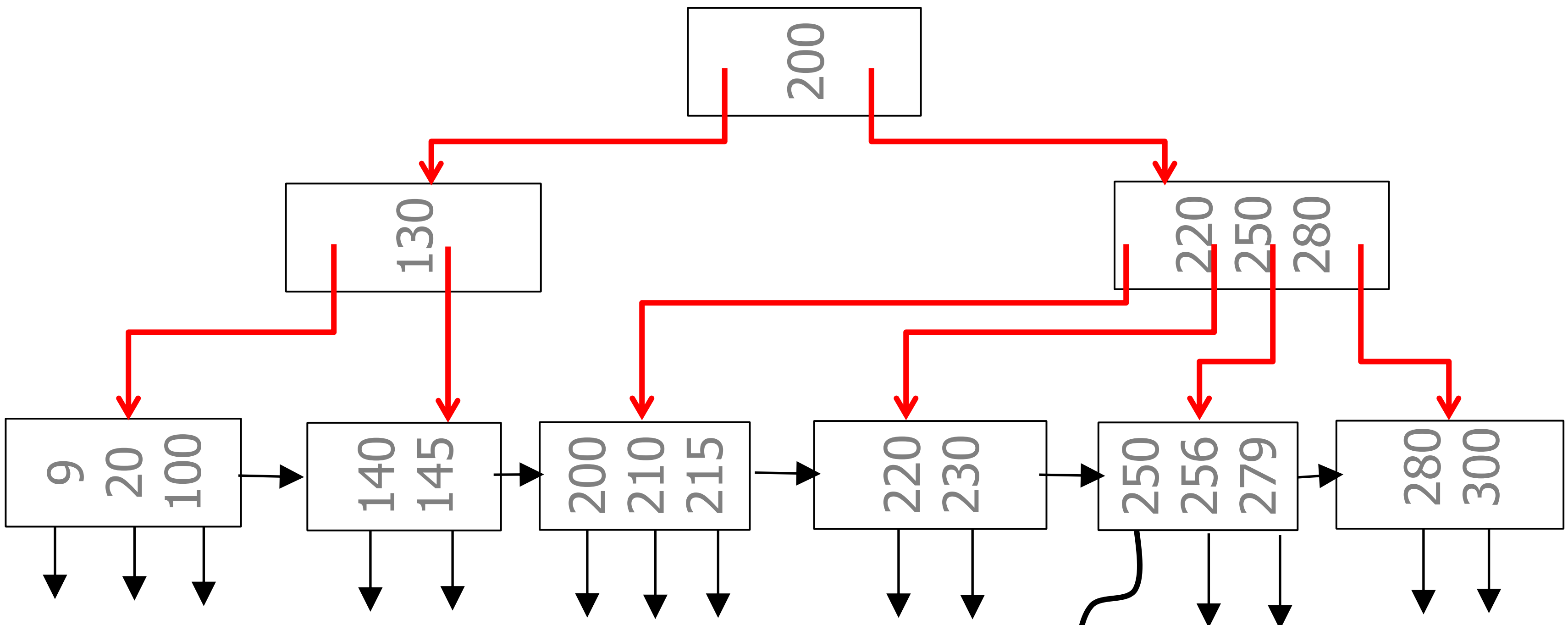
Leaf Node in B+ Tree

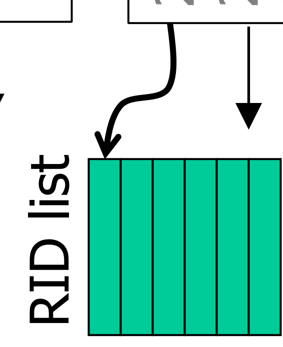
- For i = 1, 2, . . ., n–1, pointer Pi points to a file record with search-key value Ki
- If Li , Lj are leaf nodes and i < j, Li' s search-key values are less than Lj 's search-key values
  - Pn points to next leaf node in search-key order



B-tree Indexing: Example

- B+ tree of order 4
  - Maximum 4 pointer and 3 keys





Looking for Empno 250
Number of block accesses needed to find a record
= Height of tree (Ht+1)
In this case 3+1 = 4

Maximum Capacity of B+ Tree

- In a 3 level B+ tree with 100 key and 101 pointers in each node:
  - we can index as many as a million records (101 * 101 * 100 = 1 million+).
- With such an index, accessing a database record requires only 3 index-block read and 1 data-block read. Very efficient!
- Given the depth of the tree is 4, with 3 pointers
  - Index 3x3x3x2 = 54 records in database. This is the maximum capacity of this B+ tree index.
- If we use a B+ tree with **order, p = 100**, maximum how many records or blocks can we index using a **3-level** tree?
  - 100x100x99 = 990,000 (almost a million)

Searching in B+ Tree

- Generalization of binary search
  1. Give a search key, start from root node
  2. If key is present in root node then success, else
  3. If current node is a leaf node and key not present in node, then key not in database
  4. Search for tree pointer p_i such that $k_{i-1} < k_i < k_{i+1}$
  5. Return step 2 to continue search

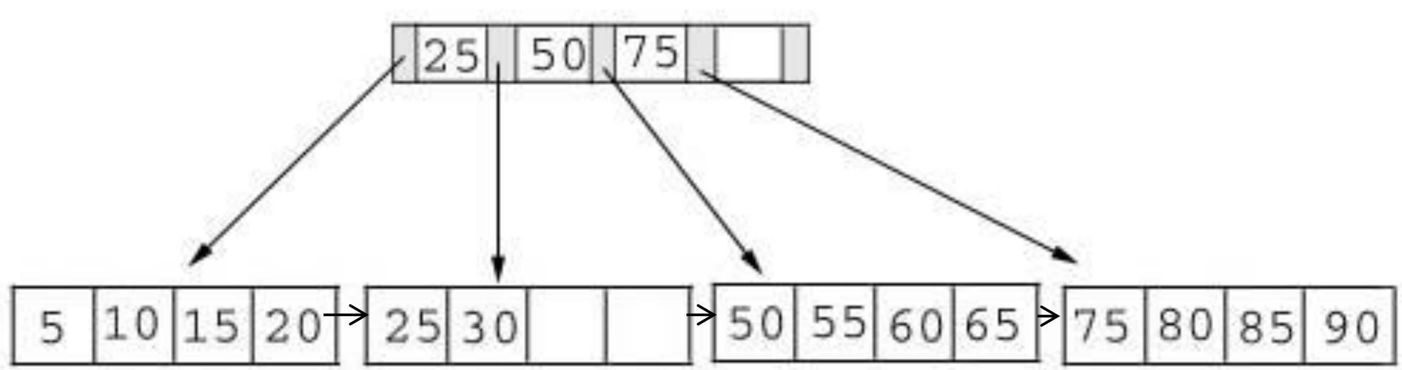
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Example of B+ tree

This table shows a B+ tree. As the example illustrates this tree does not have a full index page. (We have room for one more key and pointer in the root page.) In addition, one of the data pages contains empty slots

B+ Tree with four keys



Illustrations of the insert algorithm

- The key value determines a record's placement in a B+ tree.
- The **leaf pages** are maintained in **sequential order**
- **Doubly linked** list connects each leaf page with its **sibling page(s)**.
- This doubly linked list **speeds data movement** as the pages grow and contract.

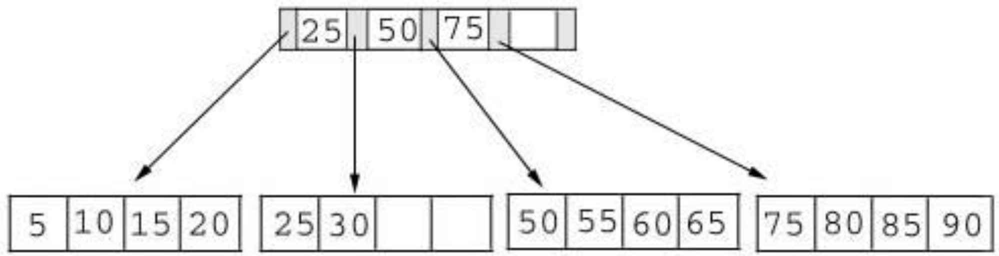
Illustrations of the insert algorithm

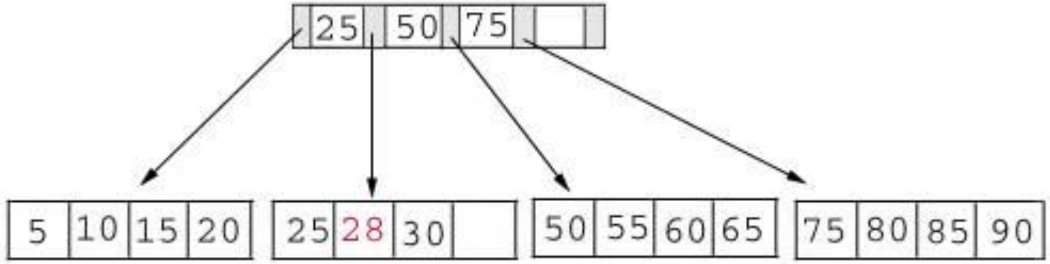
- The following examples illustrate each of the **insert** scenarios.

- We begin with the simplest scenario:
  - inserting a record into a leaf page that is not full.
  - Since only the leaf node containing **25 and 30** contains expansion room, we're going to insert a record with a key value of 28 into the B+ tree.
  - The following figures shows the result of this addition.

Add to a non-full tree

- Add Record with Key 28





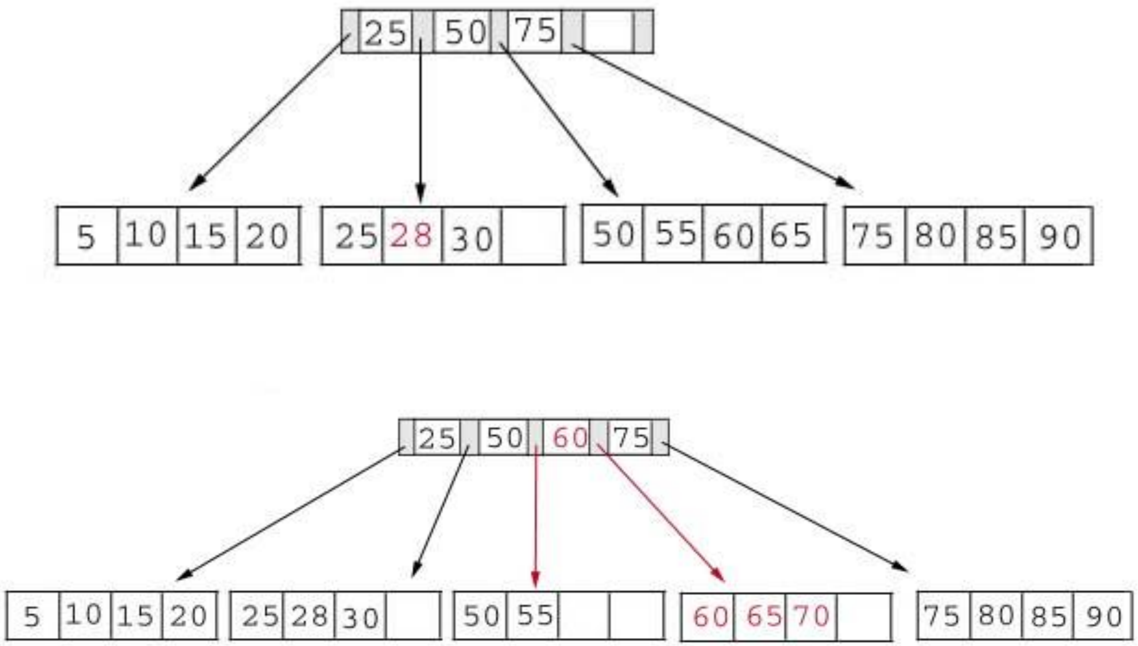
Adding a record when the leaf page is full but the index page is not

- we're going to insert a record with a key value of 70 into our B+ tree.
- This record should go in the leaf page containing 50, 55, 60, and 65.
- Unfortunately this **page is full**. This means that we must split the page as follows

Left Leaf Page	**Right Leaf Page**
50 55	60 65 70

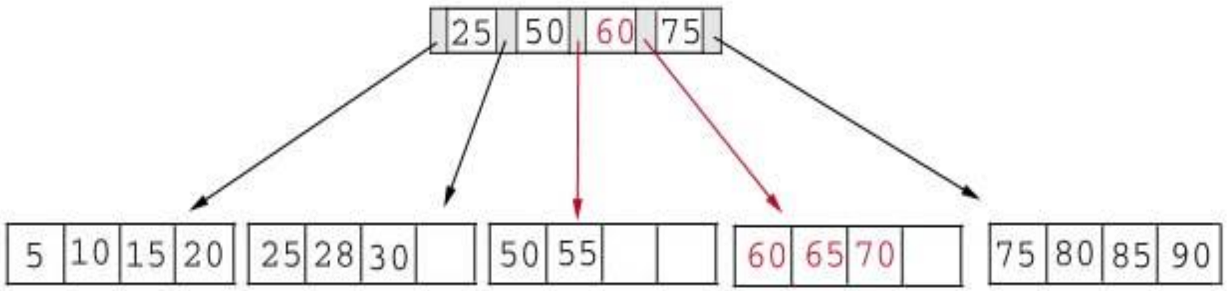
Add Record with Key 70 : Leaf node split example

- The middle key of 60 is placed in the index page between 50 and 75.
- The following table shows the B+ tree after the addition of 70.

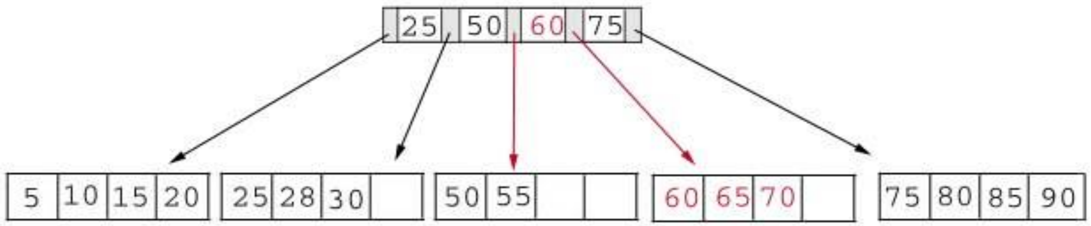


Adding a record when both the leaf page and the index page are full

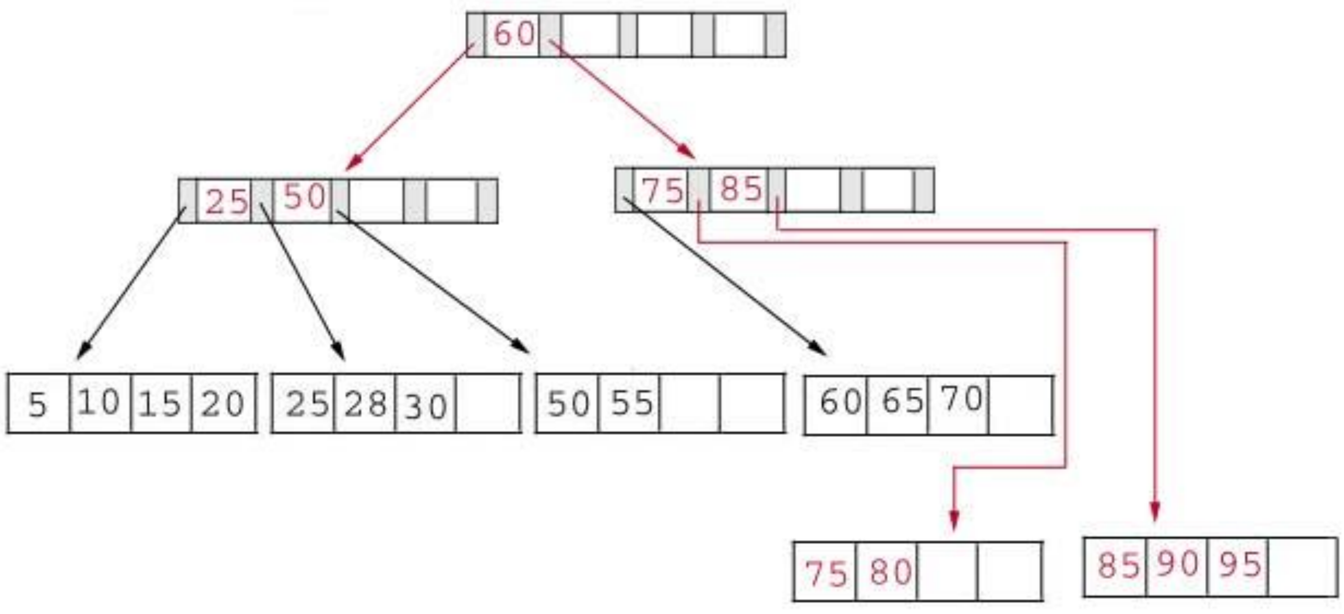
- Add 95 to B+ tree.
- This record belongs in the page containing **75, 80, 85, and 90**. Since this page is full, split it into two pages:
- **Left Leaf Page   Right Leaf Page**
- 75 80   **85** 90 95



- The middle key, **85**, rises to the index page.
- Unfortunately, the index page is also full, so we split the index page:
  - **Left Index Page** **Right Index Page** **New Index Page**
  - **25 50** **75 85** **60**



The following table illustrates the addition of the record containing 95 to the B+ tree.

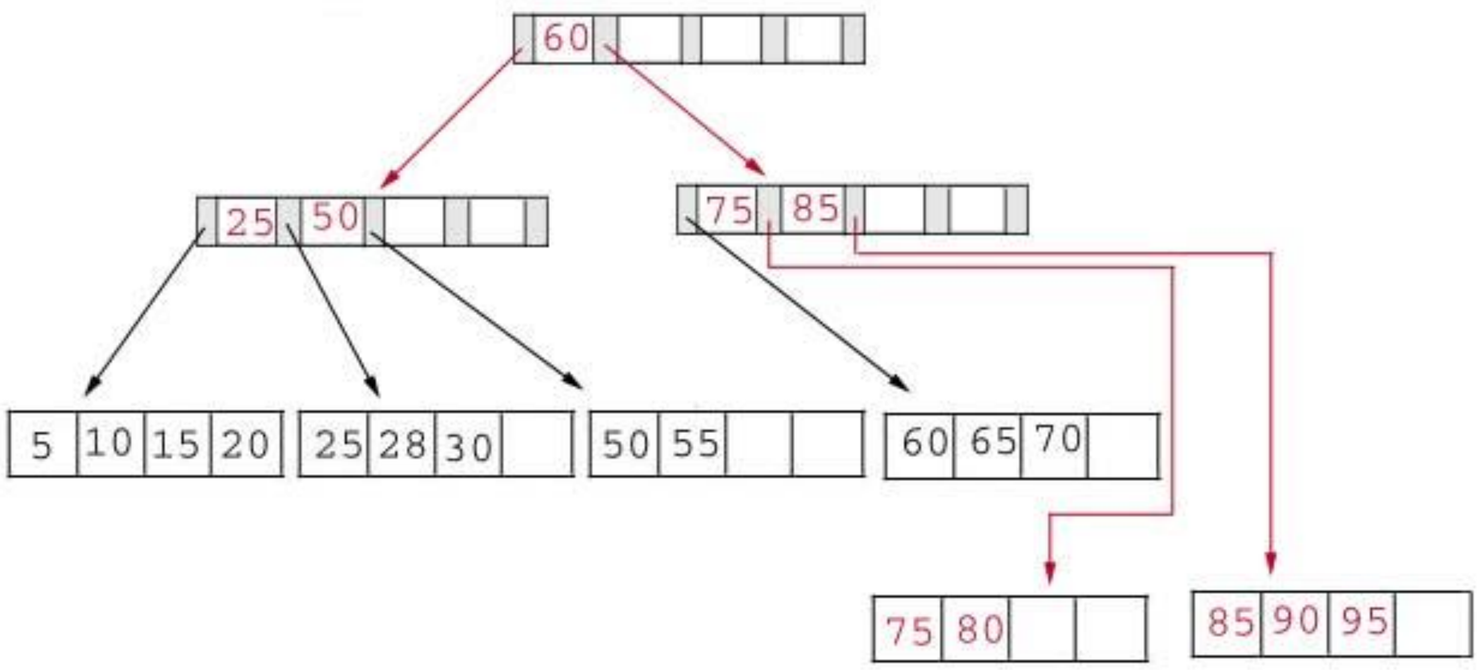


Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has less then half entries,
    - Try to re-distribute, borrowing from <u>*sibling*</u> *(adjacent node with same parent as L)*.
    - If re-distribution fails, <u>*merge*</u> *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Deleting Keys from a B+ tree

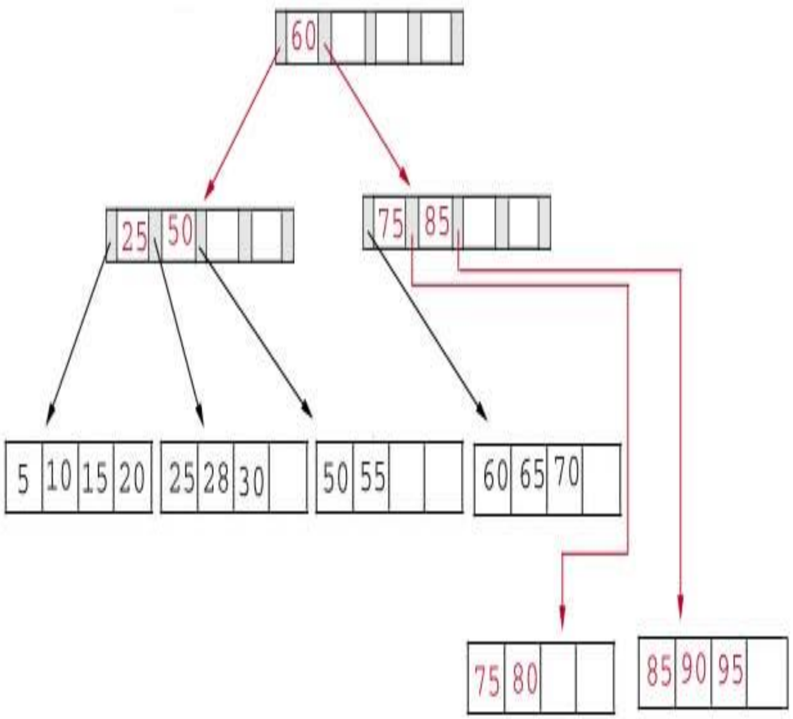
- As our example, we consider the B+ tree after we added 95 as a key.
- As a refresher this tree is printed in the following table.

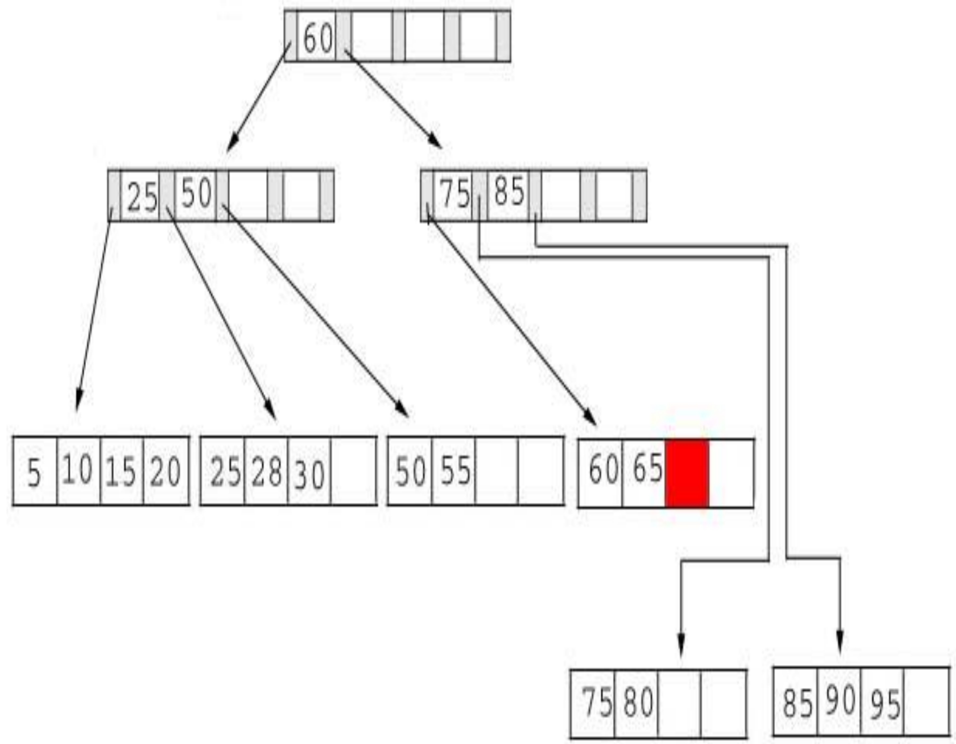


Delete 70 from the B+ Tree

- We begin by deleting the record with key 70 from the B+ tree.
- This record is in a **leaf page** containing **60, 65 and 70.**
- This page will contain 2 records after the deletion.
- Since our fill factor is 50% or (2 records) we simply delete 70 from the leaf node.
- The following table shows the B+ tree after the deletion.

Delete Record with Key 70

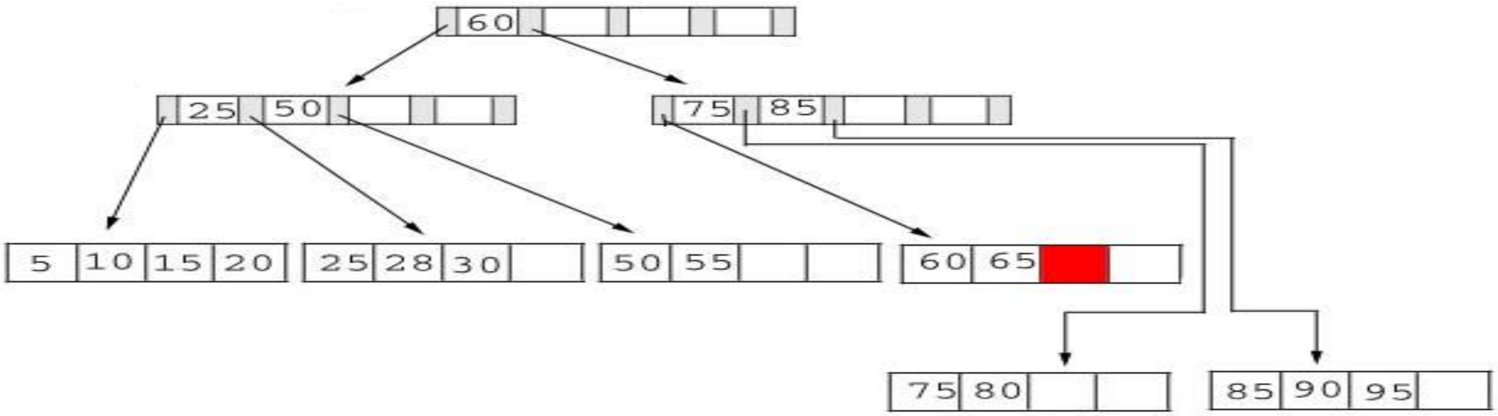


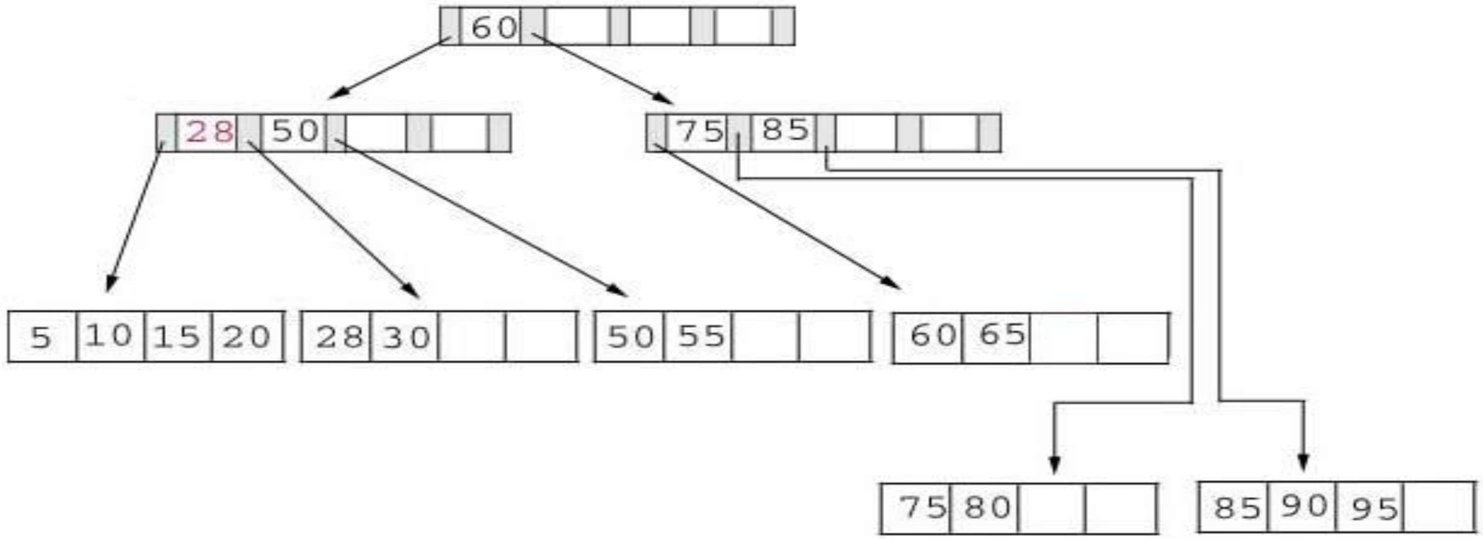


Delete 25 from the B+ tree

- Delete the record containing 25 from the B+ tree.
- This record is found in the leaf node containing 25, 28, and 30.
- The fill factor will be 50% after the deletion; however, **25 appears in the index page**.
- Thus, when we delete 25 we must replace it with 28 in the index page.

Delete 25 from the B+ tree



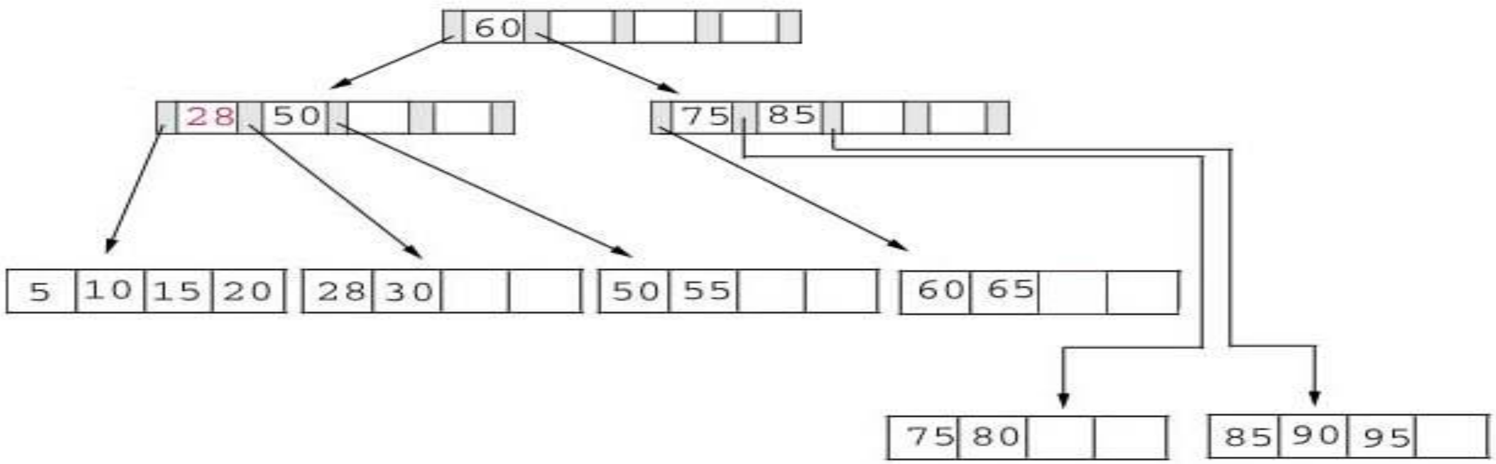


Delete 60 from the B+ tree

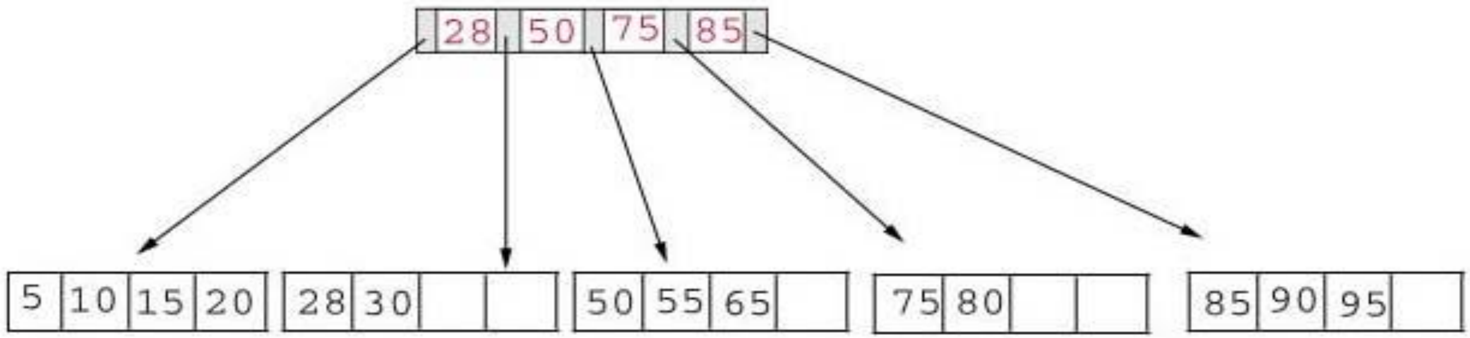
- Delete 60 from the B+ tree. This deletion is interesting for several reasons:

1. The **leaf page** containing 60 (60 65) will be **below the fill factor** after the deletion. Thus, we must **combine leaf pages**.
2. With recombined pages, the **index page** will be reduced by one key. Hence, it will also **fall below the fill factor**. Thus, we must **combine index pages**.
3. **Reduce index level**: 60 appears as the only key in the root index page. Obviously, it will be removed with the deletion.

The following table shows the B+ tree after the deletion of 60. Notice that the tree contains a single index page.



Delete Record with Key 60



Exercise

- Create an initial B+ tree with a B+-tree with p = 3 and pleaf = 2
  - First Insert 8, 5,
  - Then insert in the following sequence
    - 1, 7, 3, 12, 9, 6
  - At each insertion observe, if it is increasing width or height of the tree
- Deletion Sequence: 5, 12, 9

Use this visualization tool to enhance your understanding:
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Hashing

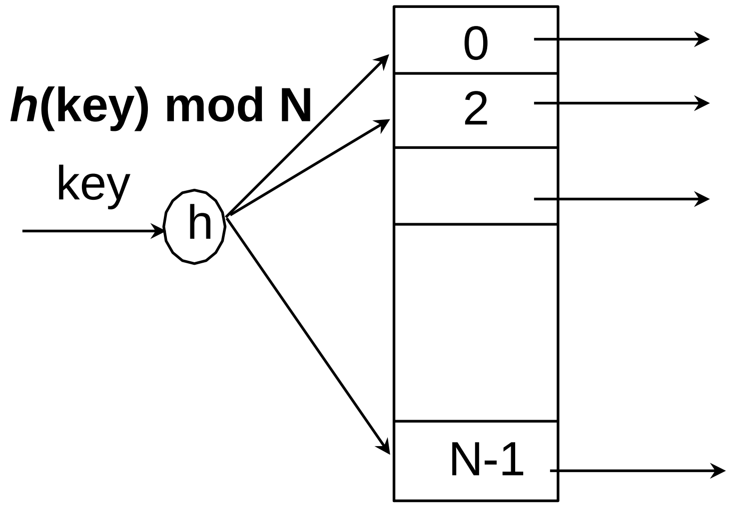
- Provides *rapid, non-sequential, direct access* to records, by providing a ***Search Key*** value

- **Hashing:** A **key record** field is used to calculate the record address by subjecting it to some calculation; a process called ***hashing.***
  - A hash function is **computed on some attribute** of each record.
  - The result of the function specifies in which block of the file the record should be placed

Hashing

- A hash structure (or table or file) is a *generalization* of the simpler notion of an ordinary array
  - In an array, an arbitrary position can be examined in O(1)

- A hash function ***h*** is used to map keys into a range of *bucket numbers*



Primary bucket pages

Hash Indexes

- The **hash index** is a secondary structure to access the file by using hashing on a search key
- The index entries are of the type
  - <*K*, *Pr*> or <*K*, *P*>,
  - where *Pr* is a pointer to the record containing the key, or *P* is a pointer to the block(bucket) containing the record for that key.

Hash-Based Indexing

- What indexing technique can we use to support *range searches* (e.g., “Find s_name where gpa >= 3.0)?
  - Tree-Based Indexing

- What about *equality selections* (e.g., “Find s_name where sid = 102”?
  - Tree-Based Indexing
  - Hash-Based Indexing (*cannot support range searches!*)

Downsides of Indexes

1) Extra space

2) Index creation

3) Index maintenance

Picking which indexes to create

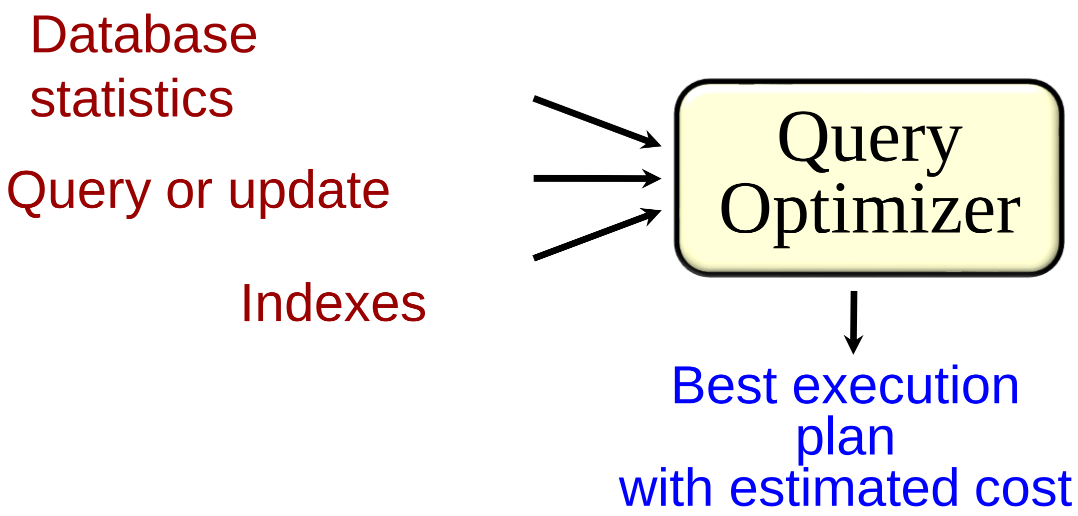
Benefit of an index depends on:

- Size of table (and possibly layout)
- Data distributions
- Query vs. update load

“Physical design advisors”

Input: database (statistics) and workload

Output: recommended indexes



Summary

- Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B+-Trees
- Hash Indexes