Data on External Storage

- External Storage: offer persistent data storage
  - Unlike physical memory, data saved on a persistent storage is not lost when the system shutdowns or crashes.

Types of External Storage Devices

- Magnetic Disks: Data can be retrieved randomly

- Tapes: Can only read pages in sequence
  - Cheaper than disks

- Other types of persistent storage devices:
  - Optical storage (CD-R, CD-RW, DVD-R, DVD-RW)
  - Flash memory

- A *record* is a tuple or a row in a table.
  - Fixed-size records or variable-size records
- A *page* is a fixed length block of data for disk I/O.
  - A data page contains a collection of records.
  - A file consists of pages.
- A *file* is a collection of records.
  - Store one table per file, or multiple tables in the same file

- Typical page sizes are 4 and 8 KB.

- *Search Key:* attribute or set of attributes used to look up records in a file

File Organization

- Method of arranging a file of records on external storage.
  - *Record id (rid)* is used to locate a record on a disk
  - *Indexes* are data structures to efficiently search rids of given values

Alternative File Organizations and Comparison of File Organizations

- Many alternatives exist, *each ideal for some situations, and not so good in others:*
  - *1.Heap files*: Records are unsorted. Suitable when typical access is a file scan retrieving all records without any order.
    - Fast update (insertions / deletions)
  - *2.Sorted Files*: Records are sorted. Best if records must be retrieved in some order, or only a `range' of records is needed.
    - Examples: employees are sorted by age.
    - Slow update in comparison to heap file.

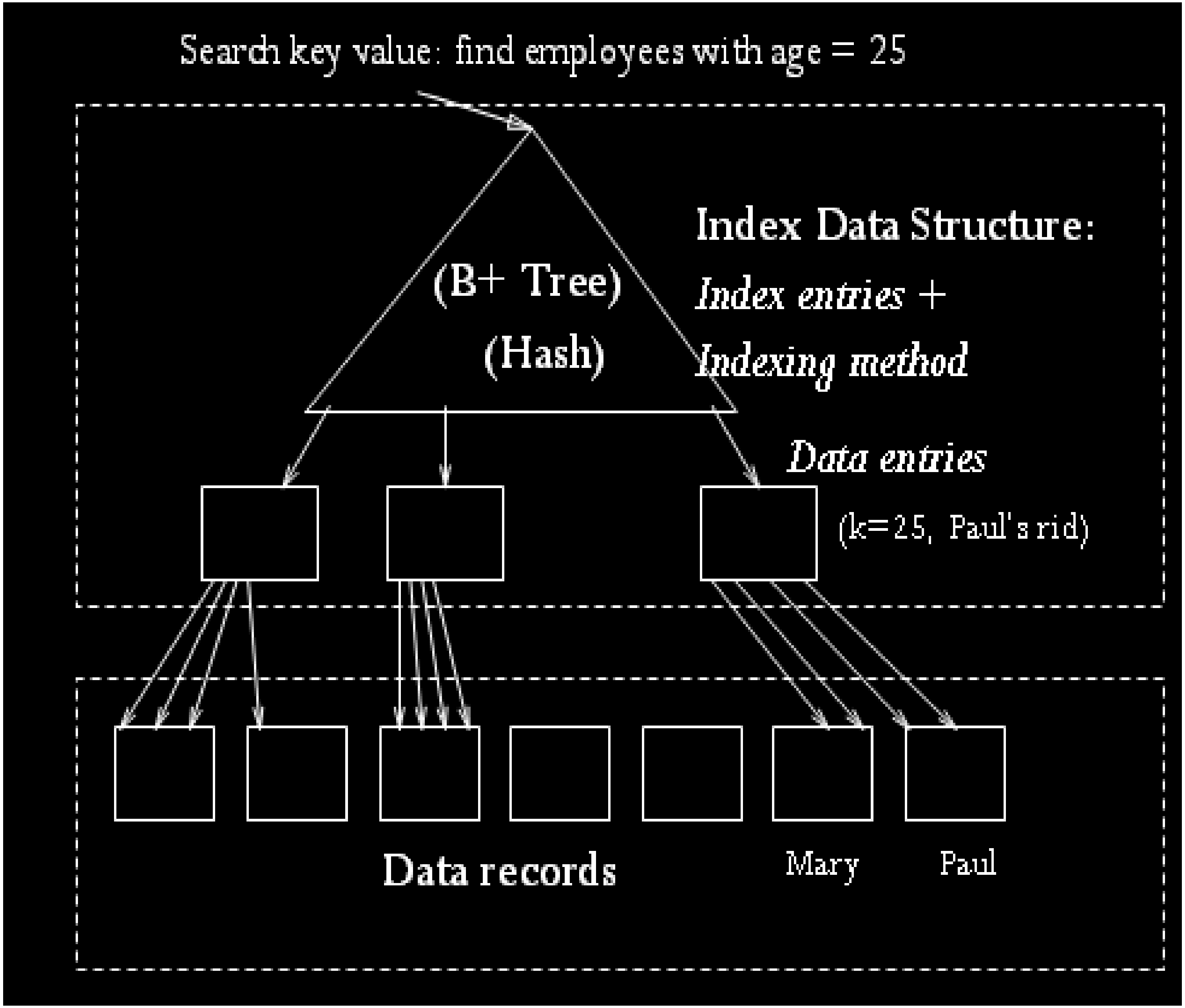
- *3.Indexes*: Data structures to organize records via trees or hashing.
  - For example, create an index on employee age.
  - Like sorted files, speed up searches for a subset of records that match values in certain (“search key”) fields
  - Updates are much faster than in sorted files.

Indexes

- *Indexes* are data structures to efficiently search rids of given values
- Any subset of the attributes of a table can be the search key for an index on the relation.
  - Search key does not have to be candidate key
    - Example: employee age is not a candidate key.
- An index file contains a collection of data entries (called **k*).**
  - Quickly search an index to locate a data entry with a key value k.
    - Example of a data entry: <age, rid>
  - Can use the data entry to find the data record.
    - Example of a data record: <name, age, salary>
  - Can create multiple indexes on the same data records.
    - Example indexes: age, salary, name

- Three alternatives for what to store in a data entry:
  - (Alternative 1): Data record with key value **k**
    - Example data record = data entry: <**age**, name, salary>
  - (Alternative 2): <**k**, rid of data record with search key value **k**>
    - Example data entry: <**age**, rid>
  - (Alternative 3): <**k**, list of rids of data records with search key **k**>
    - Example data entry: <**age**, rid_1, rid_2, …>
- Choice of alternative for data entries is independent of the indexing method.
  - Indexing method takes a search key and finds the data entries matching the search key.
  - Examples of indexing methods: B+ trees or hashing.

Indexing Example



Index File (Small for efficient search)

Data File (Large)

Index Classification

- *Primary index:* In a sequentially ordered file, the index whose search key specifies the sequential order of the file. Also called *clustered index*

  ---Order of data records is same as, or close to the order of data entries

- The search key of a primary index is usually but not necessarily the primary key

• *Secondary index:* an index whose search key specifies an order different from the sequential order of the file.

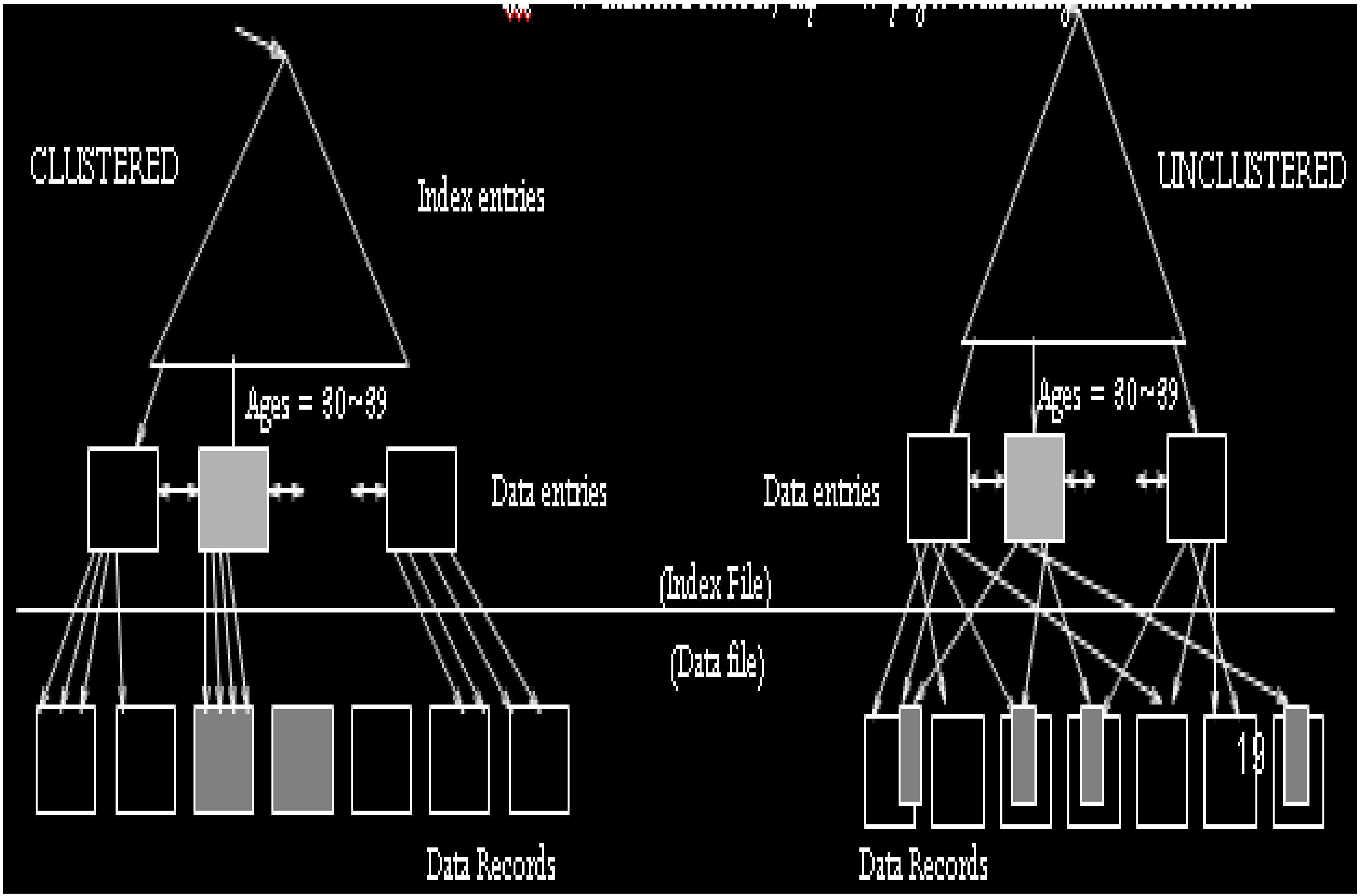
--also called *non-clustered index*

Dense index: Index record appears for every search-key value in the file

Sparse Index: contains index records for only some search-key values

--applicable when records are sequentially ordered on search-key

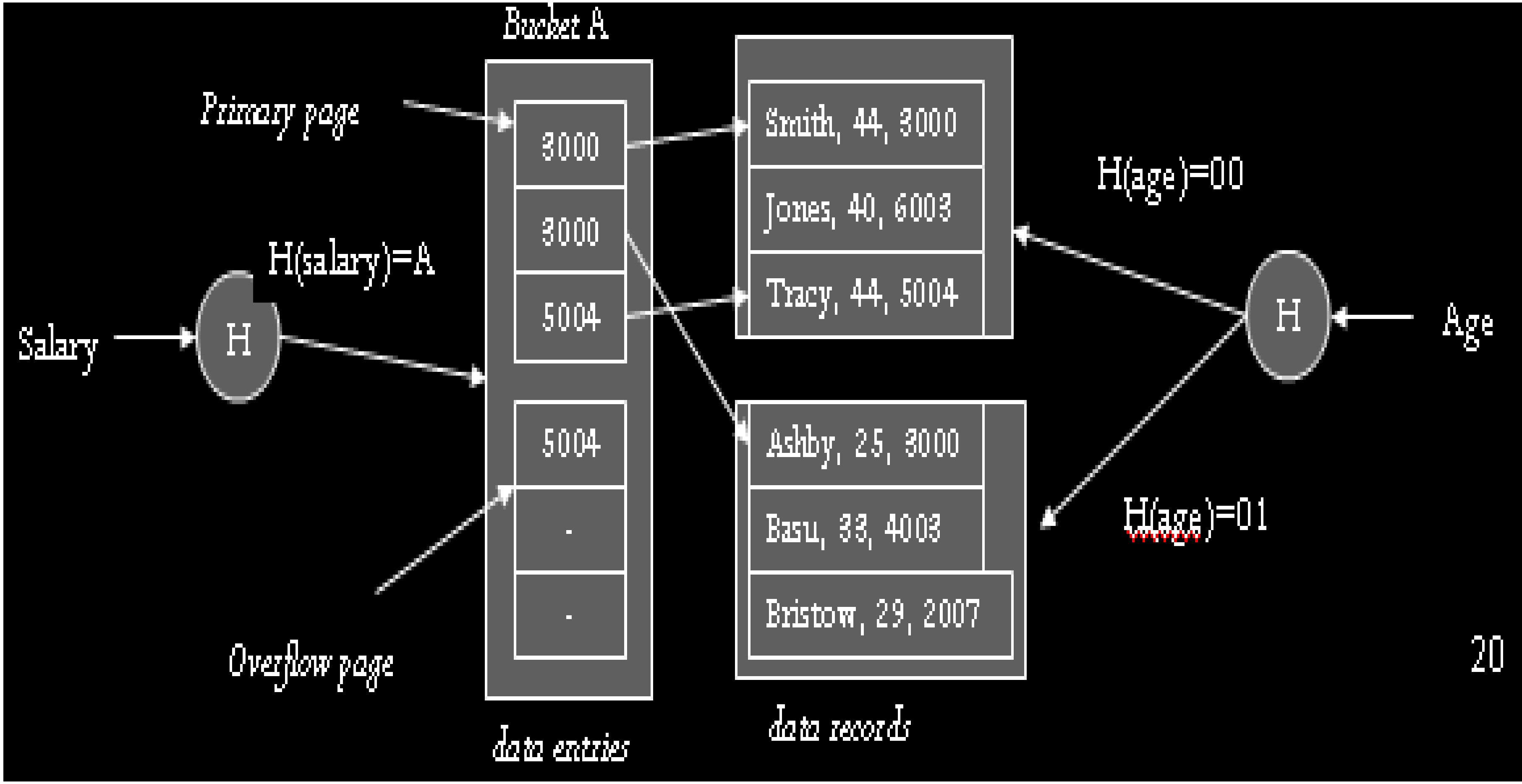
Unique index: Search key contains a candidate key



CLUSTERED
Index entries
UNCLUSTERED
Ages = 30~39
Ages = 30~39
Data entries
Data entries
(Index File)
(Data file)
19
Data Records
Data Records

Hash-Based Indexing

- Good for equality selections.
  - Data entries (key, rid) are grouped into buckets.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(r) = bucket in which record r belongs. **h** looks at the *search key* fields of r.
  - If Alternative (1) is used, the buckets contain the data records.

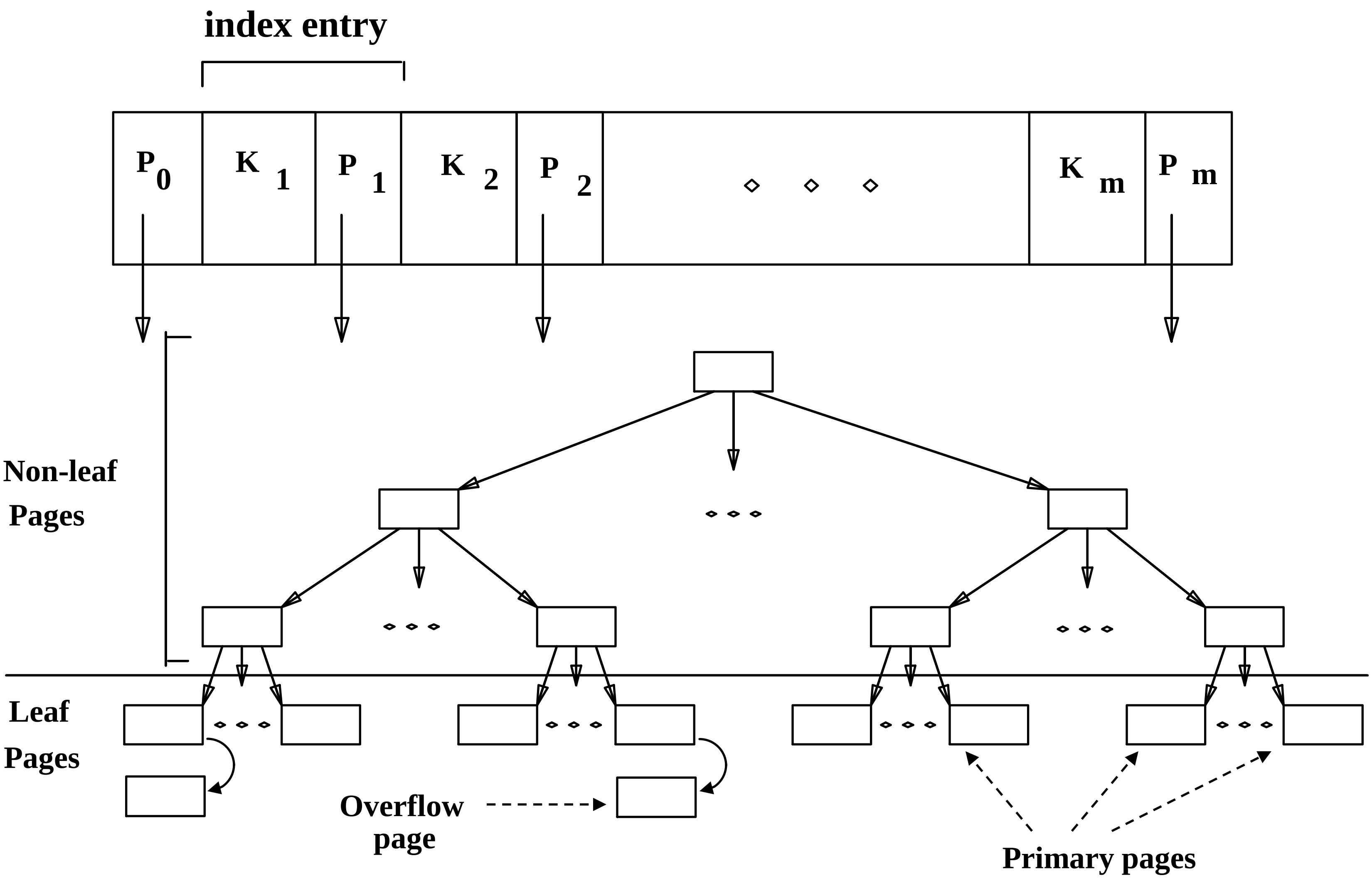

Bucket A
Primary page
3000
3000
5004
5004
.
.
Smith, 44, 3000
Jones, 40, 6003
Tracy, 44, 5004
Ashby, 25, 3000
Basu, 33, 4003
Bristow, 29, 2007
H(age)=00
H(age)=01
H(salary)=A
Salary
H
H
Age
Overflow page
data entries
data records
20

- <u>Search</u> on key value:
  - Apply key value to the hash function -> bucket number
  - Retrieve the primary page of the bucket. Search records in the primary page. If not found, search the overflow pages.
  - Cost of locating rids: # pages in bucket (small)
- <u>Insert a record</u>:
  - Apply key value to the hash function -> bucket number
  - If all (primary & overflow) pages in that bucket are full, allocate a new overflow page.
  - Cost: similar to search.
- <u>Delete a record</u>
  - Cost: Similar to search.

Tree-structured Indexing

- Tree-structured indexing techniques support both *range searches* and *equality searches*
- *ISAM*: static structure;
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Indexed Sequential Access Method



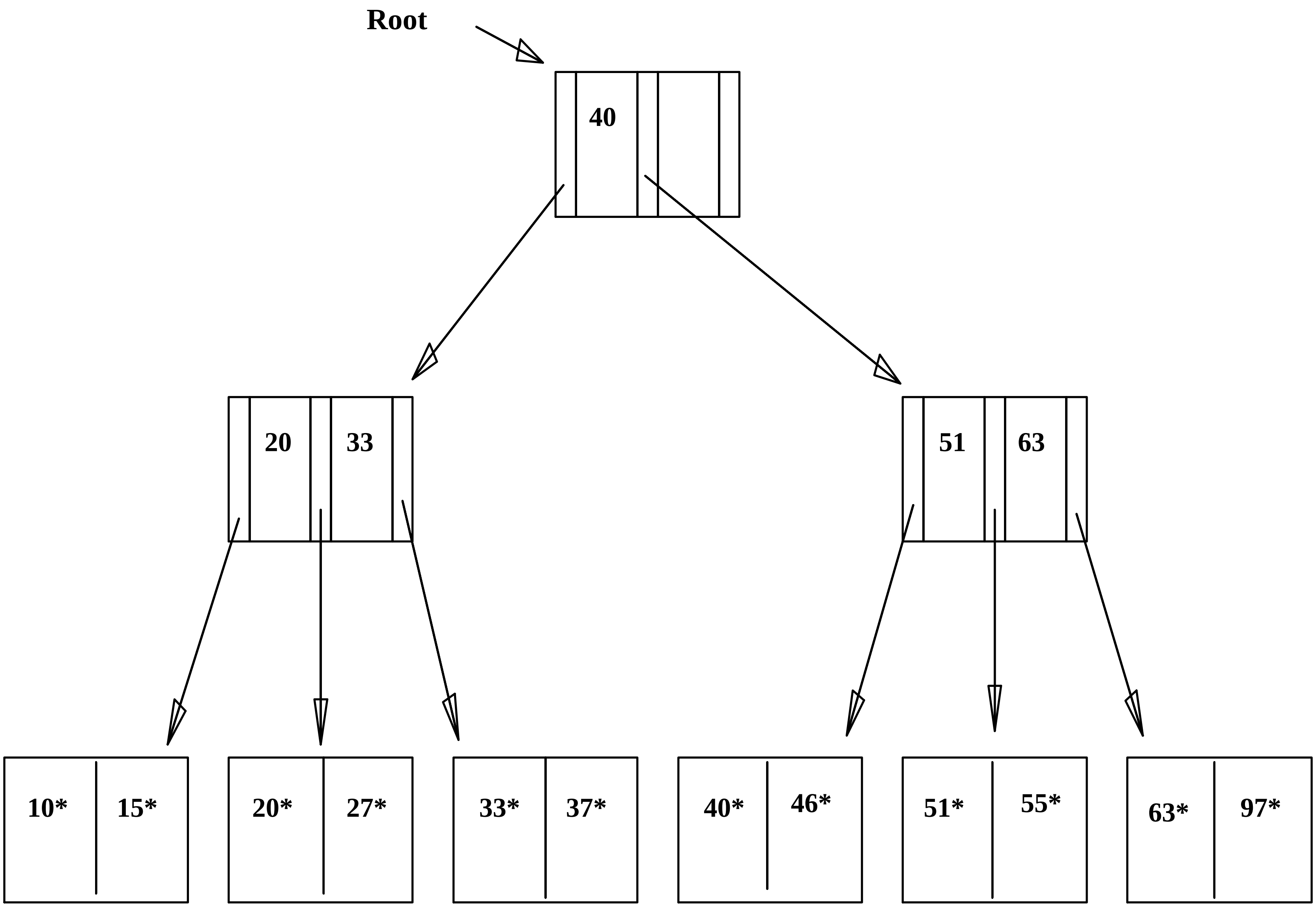
Non leaf nodes contain index entries. Leaf pages contain data entries.

- *Index entries*: <search key value, page id>;
  `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf.
- *Insert*: Find leaf that data entry belongs to, and put it there, which may be in the primary or overflow area.
- *Delete*: Find and remove from leaf; if overflow page is empty, de-allocate.

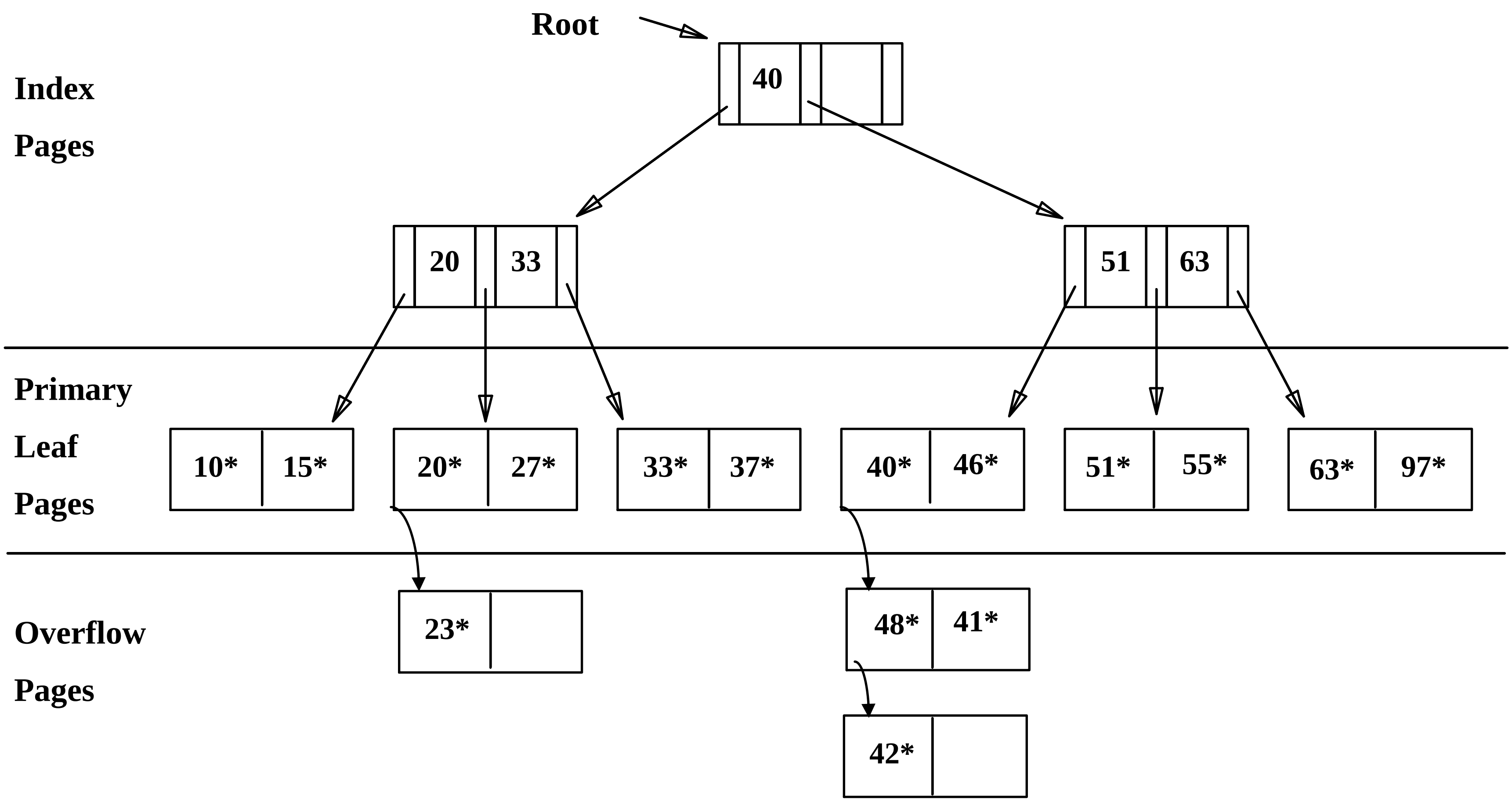
Static tree structure: *inserts/deletes affect only leaf pages.*

- Frequent updates may cause the structure to degrade
  - Index pages never change
  - some range of values may have too many overflow pages

    e.g., inserting many values between 40 and 51.

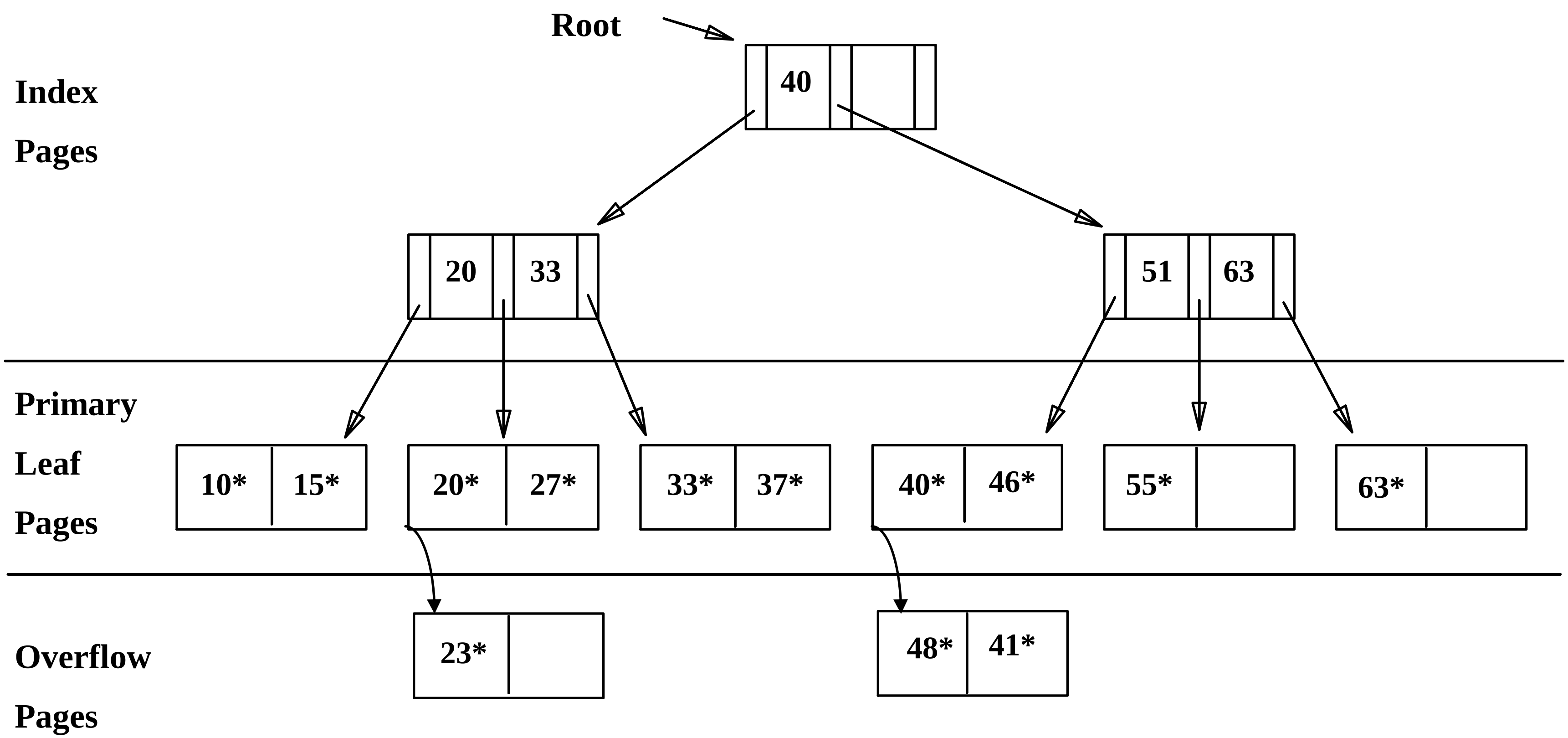

Root
40
20
33
51
63
10*
15*
20*
27*
33*
37*
40*
46*
51*
55*
63*
97*

After Inserting 23*, 48*, 41*, 42* ...



Suppose we now delete 42*, 51*, 97*.

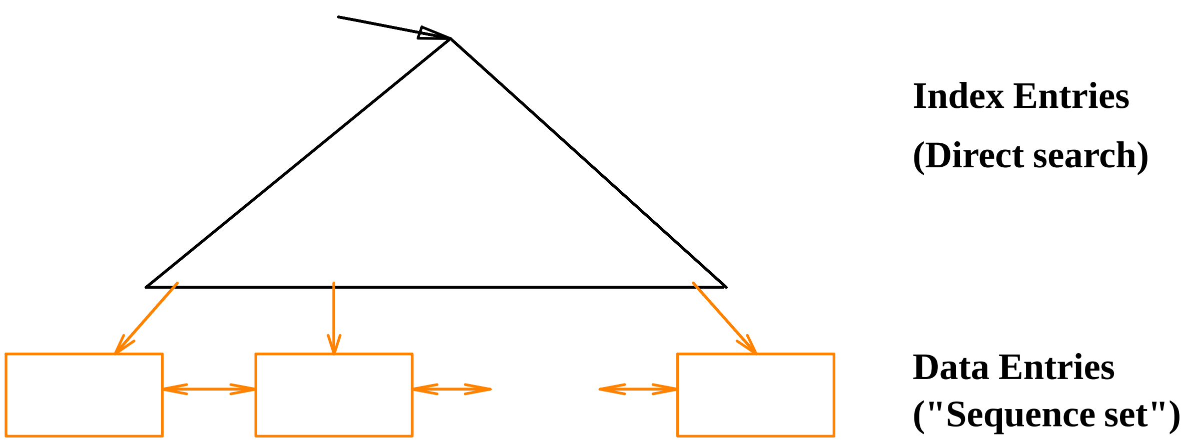
...Then Deleting 42*, 51*, 97*



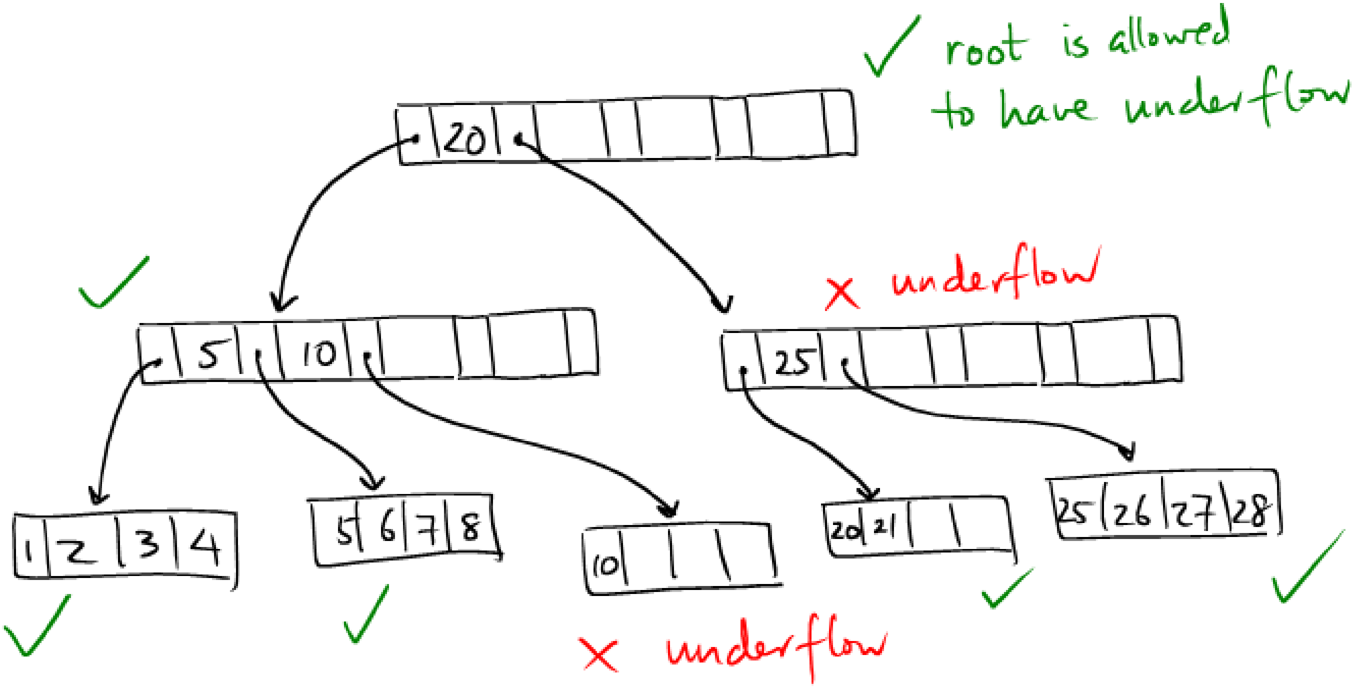
note that 51 still appears in the index page!

B+ Tree: The Most Widely Used Index

- Dynamic structure - can be updated without using overflow pages!
- Balanced tree in which internal nodes direct the search
  - Index entries same as ISAM
  - Data entries one of the 3 alternatives.
- Main characteristics:
  - Minimum 50% occupancy (except for root)
  - All paths from root to leaf are of the same length
  - All nodes (except root) has between ceil(*n/2) and n children.*
  - .[Order 5 means that a node can have a maximum of 5 children and 4 keys]
  - Supports equality and range-searches efficiently.
- Leaf pages are organized into doubly linked lists

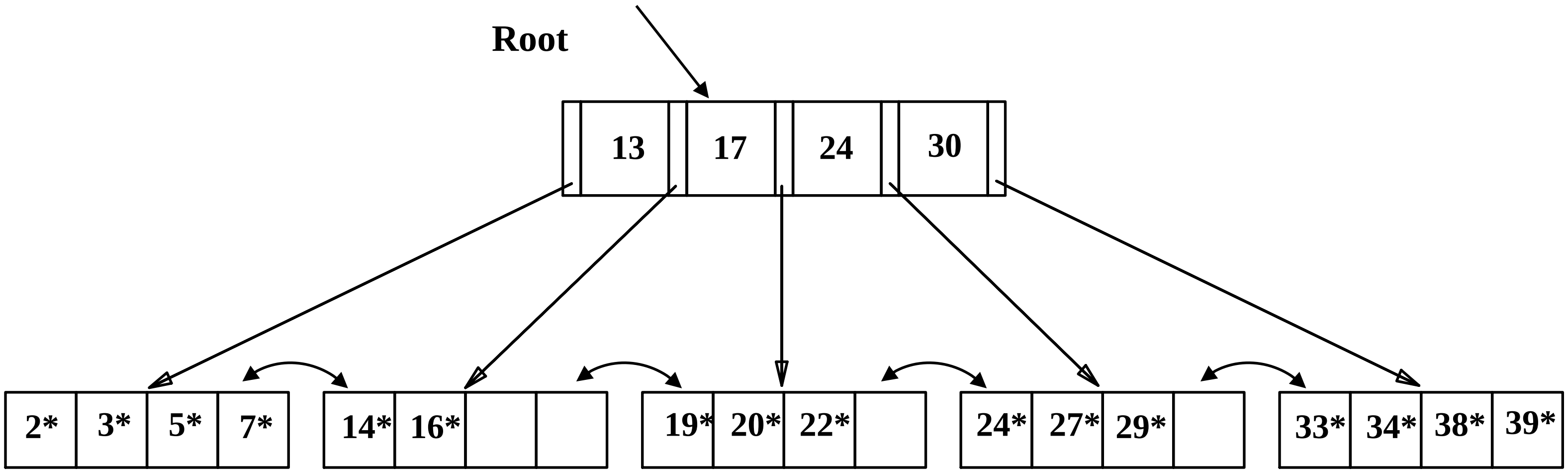


Example:



root is allowed to have underflow
20
5
10
25
× underflow
1
2
3
4
5
6
7
8
10
× underflow
20
21
25
26
27
28

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...



Inserting a Data Entry into a B+ Tree

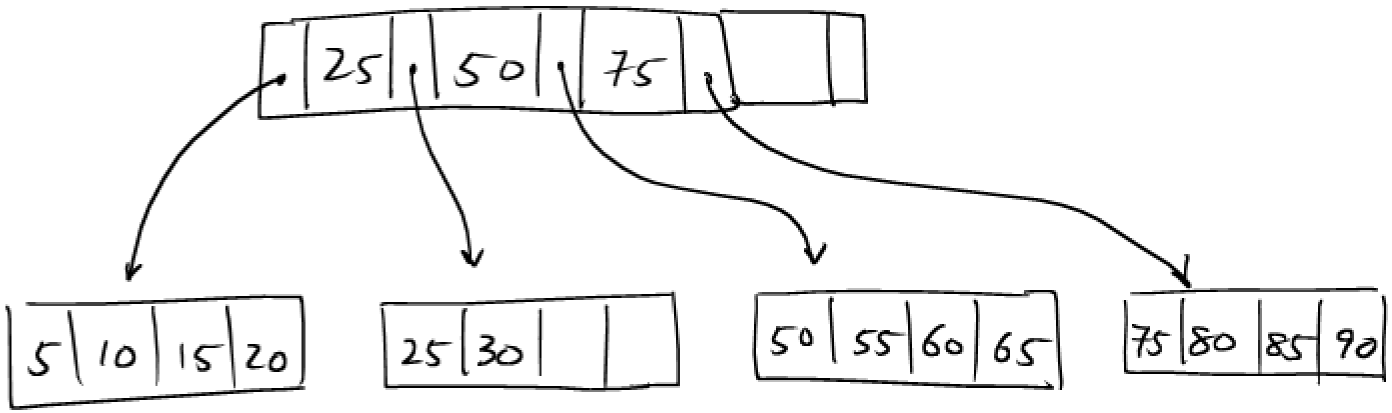
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To 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.*

<u>The tree distinct cases are:</u>

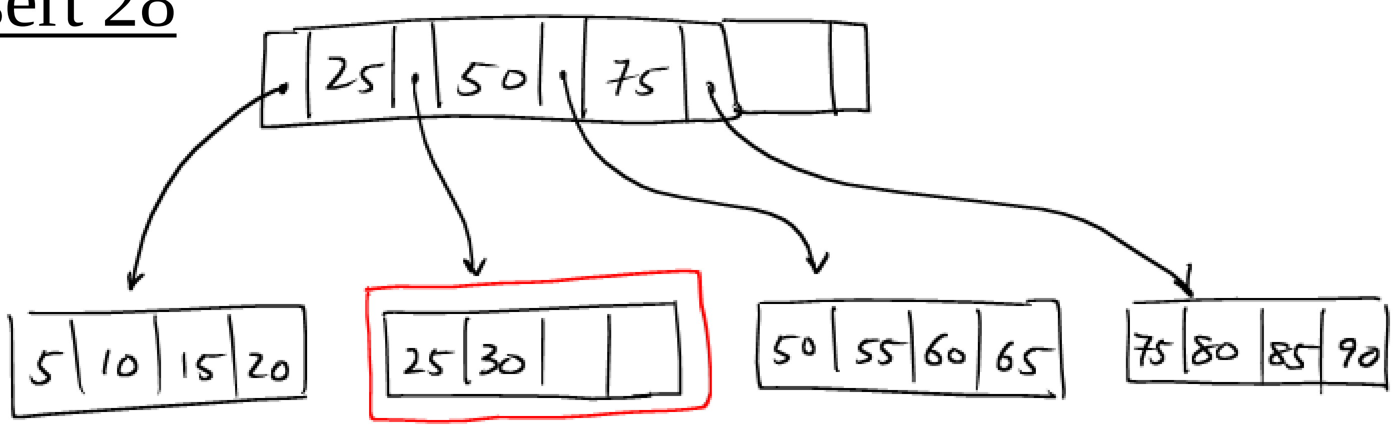
1. the target node has available space for one more key

2. the target node is full, but its parent has space for one more key

3. the target node and its parent are both full.

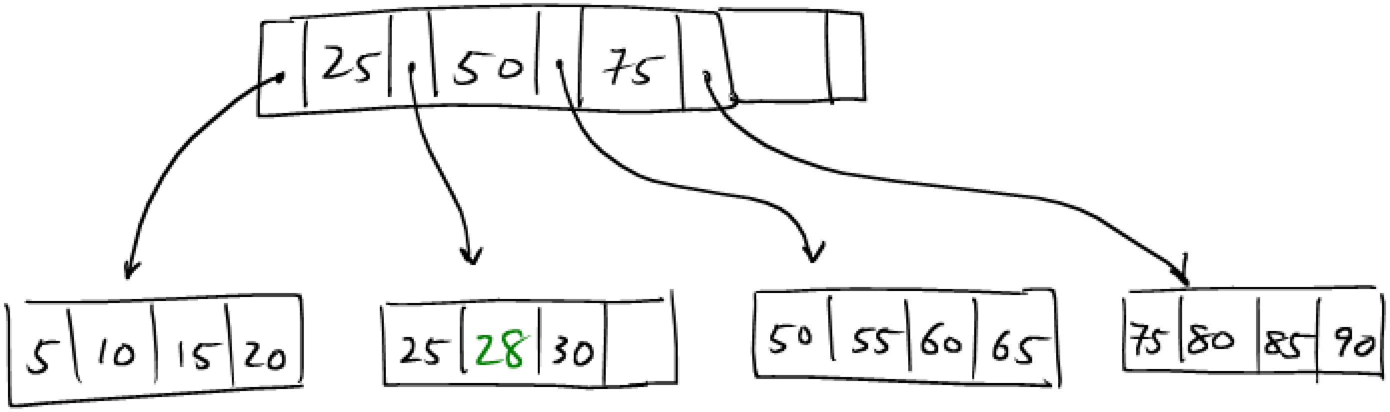


insert 28



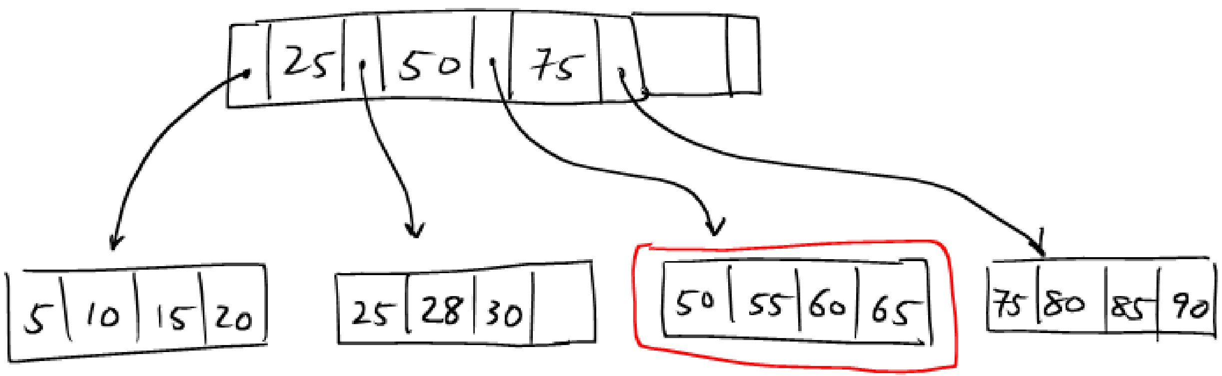
Search (root, 28)

Leaf has vacancy ⇒ CASE 1.



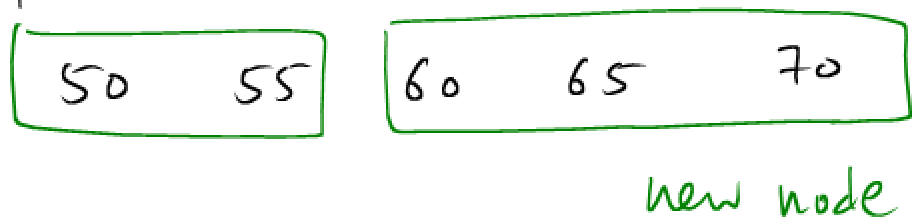
Insert (28, val)
into leaf node

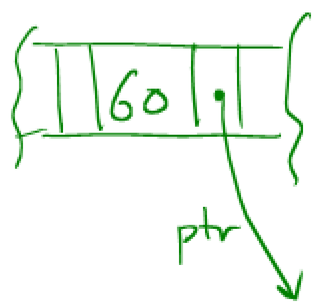
insert 70



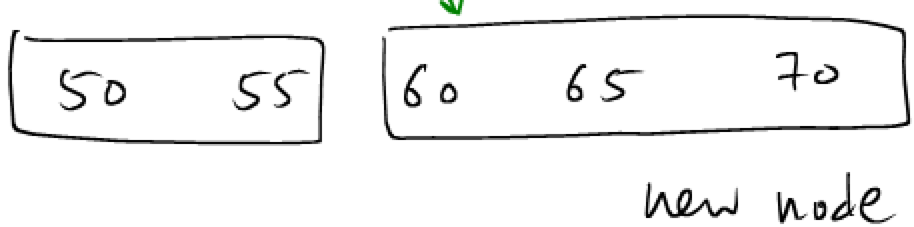
Search(root, 70)
node full, but
parent has space
⇒ CASE 2

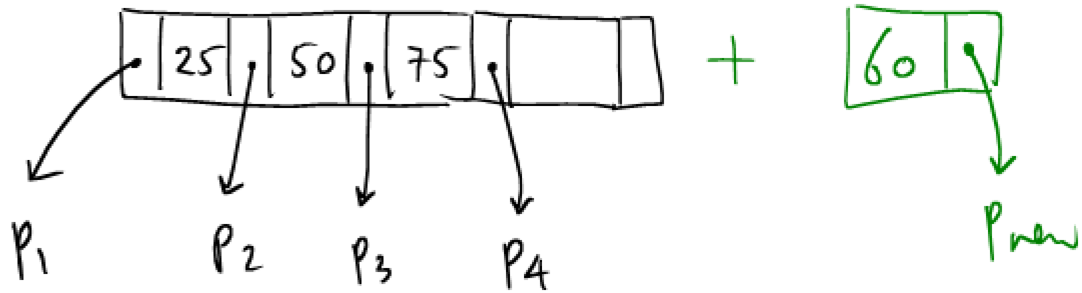
Keys are distributed

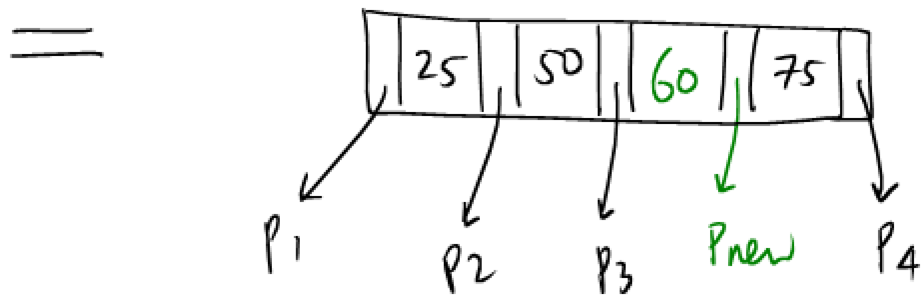




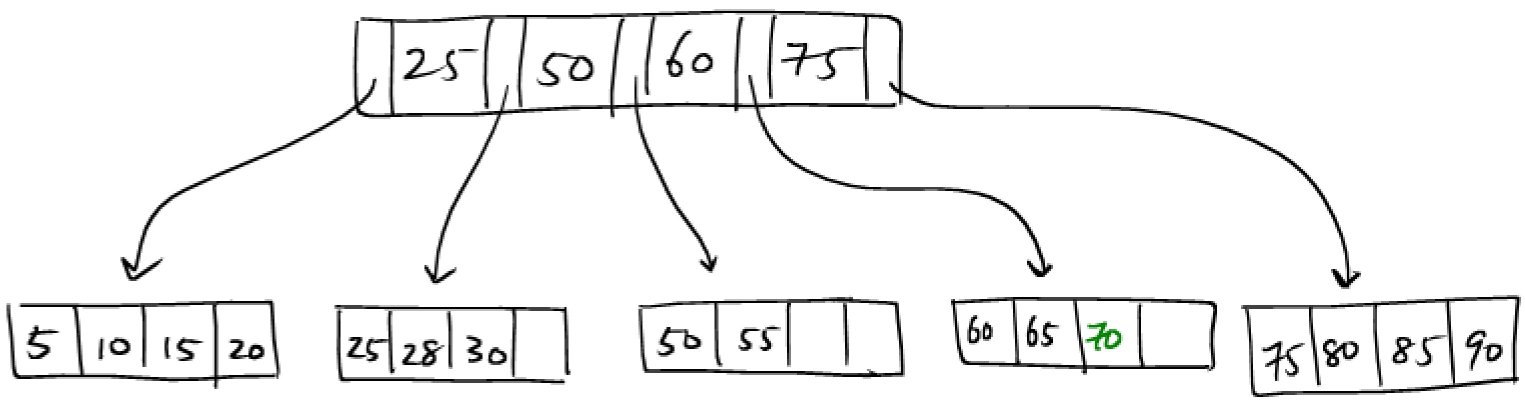
Now we need to insert (60, ptr) into the parent.

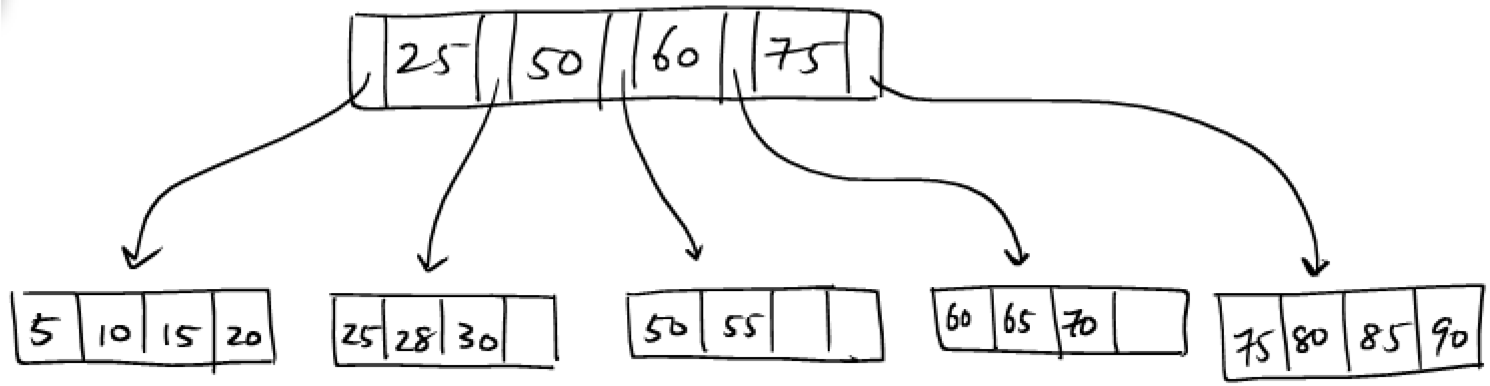



25
50
75
+
60
P1
P2
P3
P4
Pnew

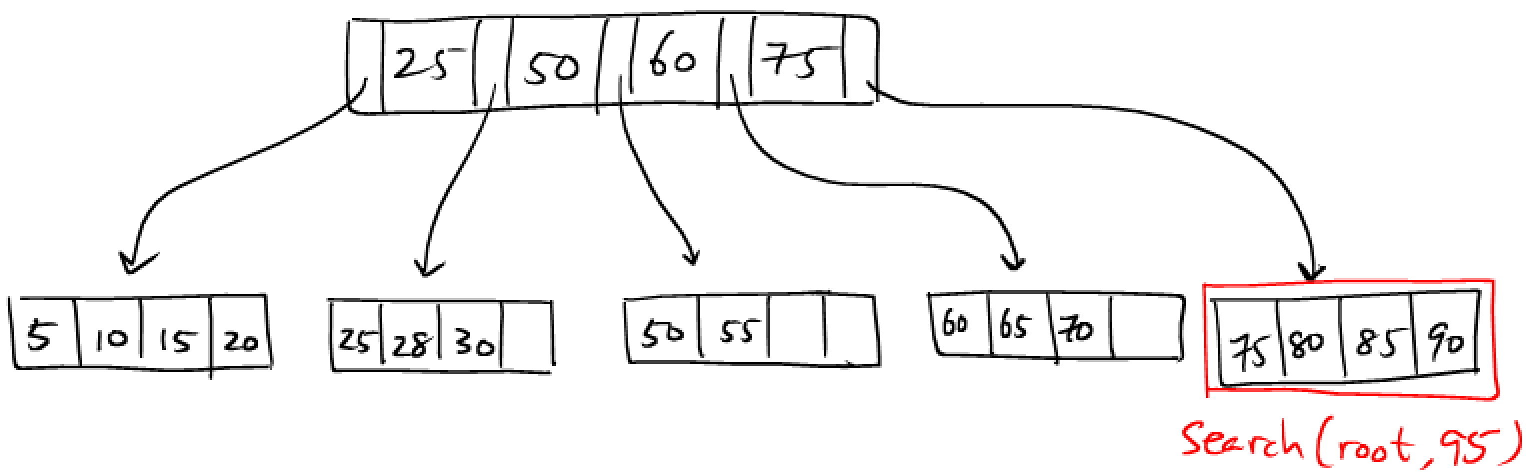

=
25
50
60
75
P1
P2
P3
Pnew
P4

New tree :

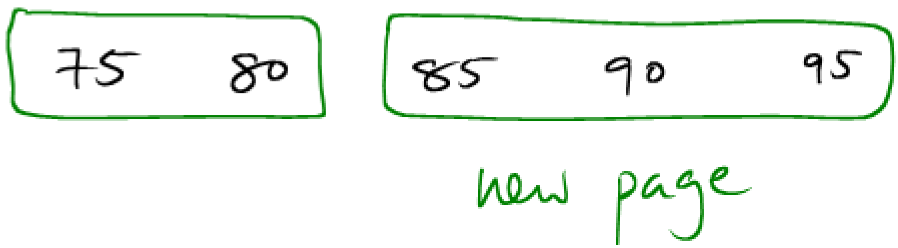

25
50
60
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90

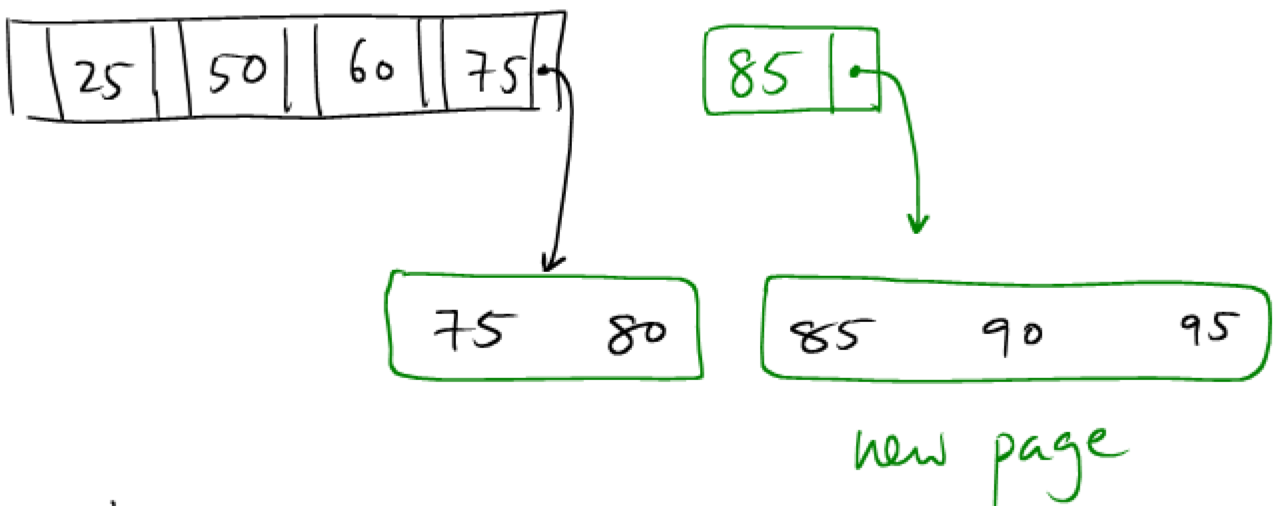

25
50
60
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90

insert 95


25
50
60
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90
Search(root, 95)

Keys at the leaf distributed





This is case #3.

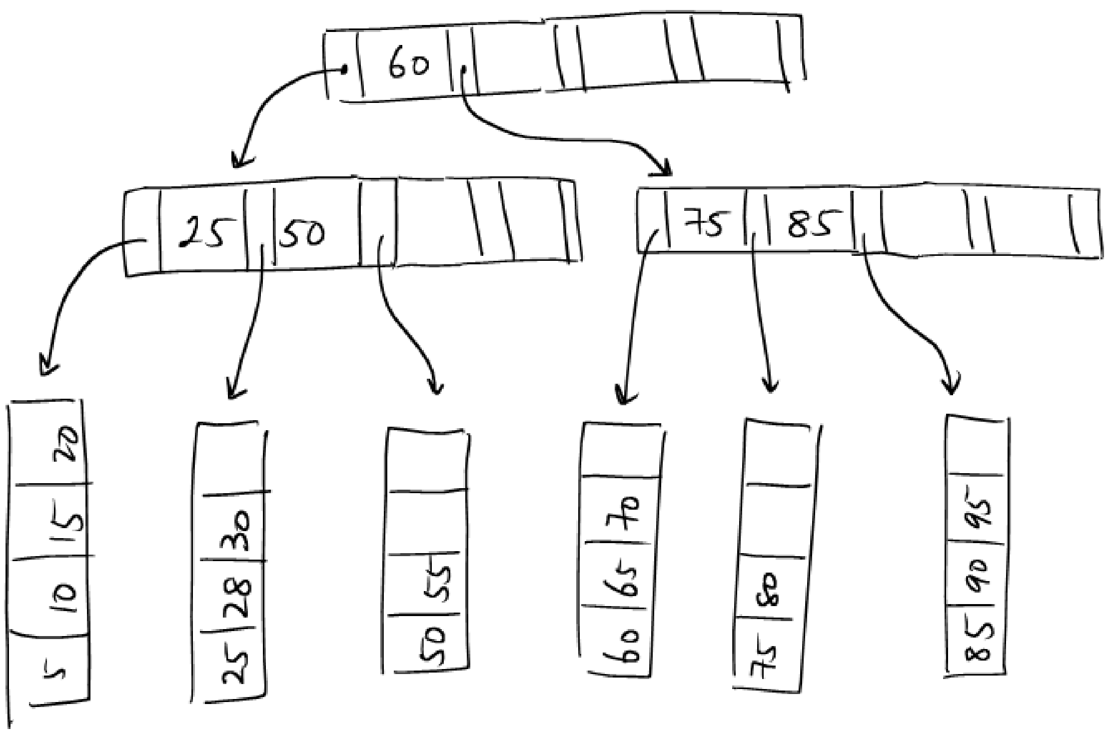
all_keys :

25 50 60 75 85

distribute to left

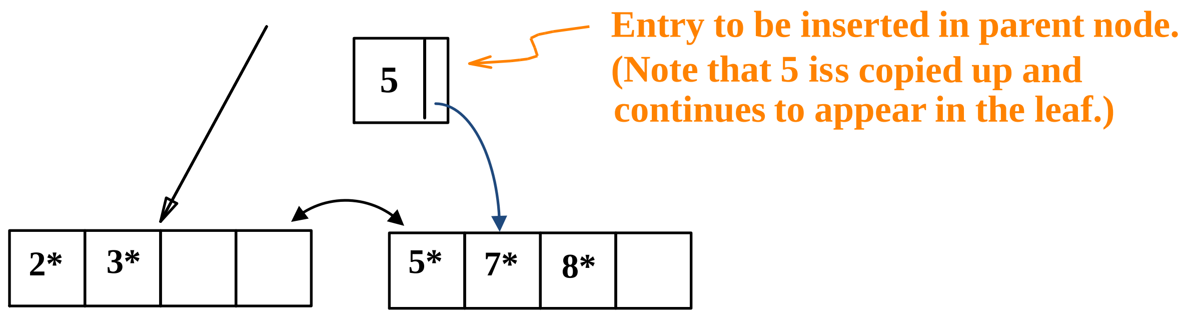
middle key

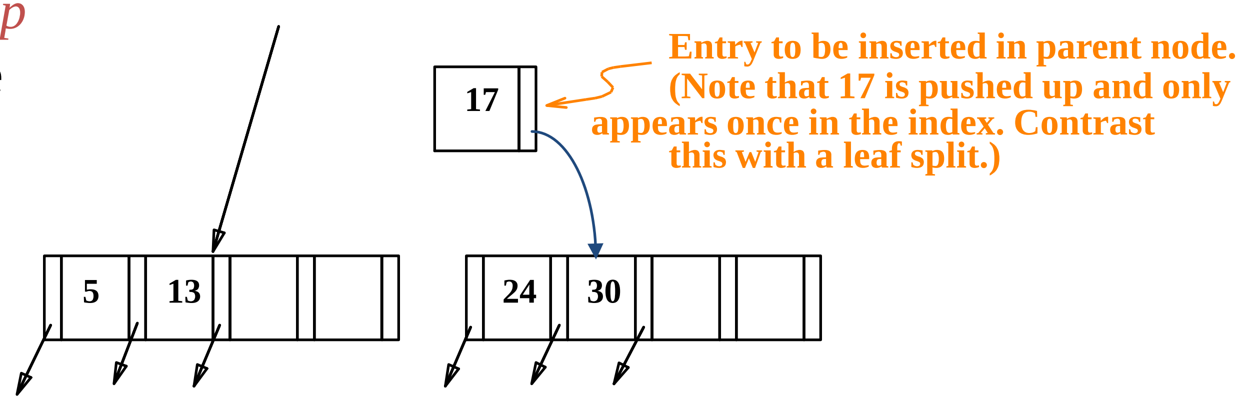
new node, distribute to right



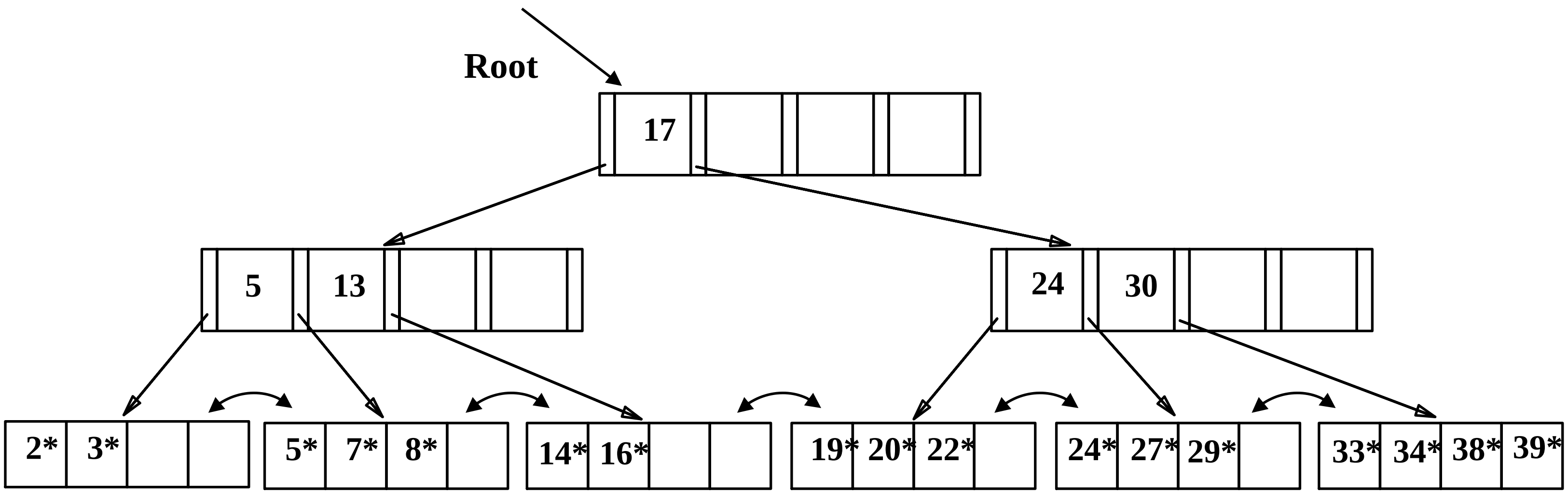
Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.





Example B+ Tree After Inserting 8*

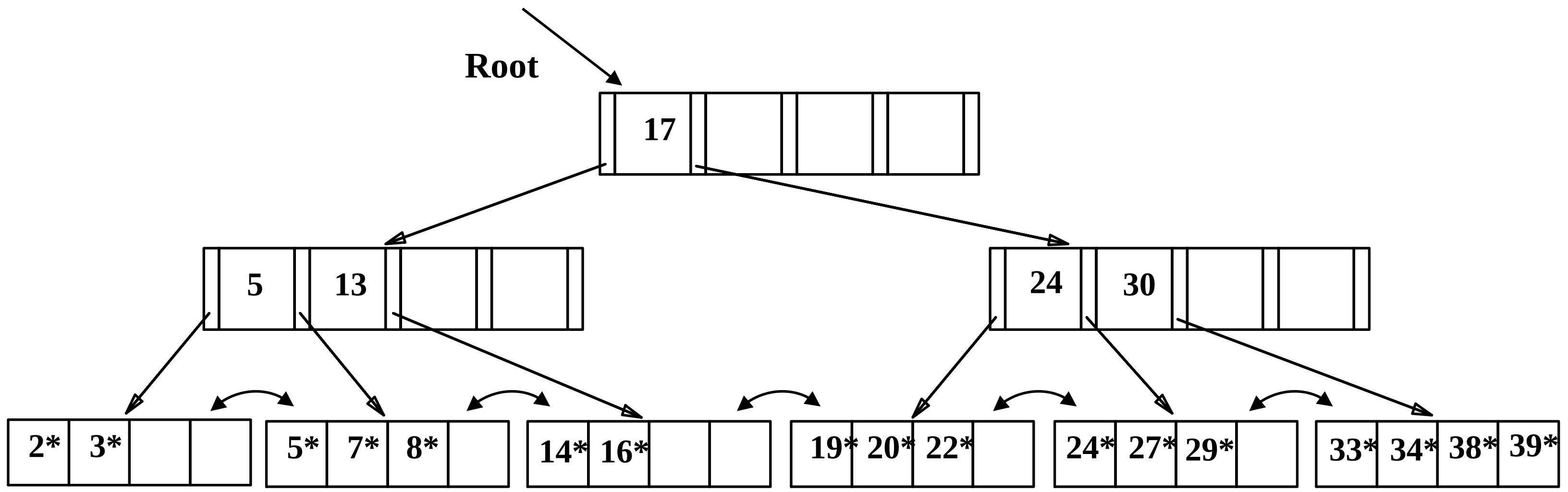


❖ Notice that root was split, leading to increase in height.

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 only **few** entries (less than half),
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *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.

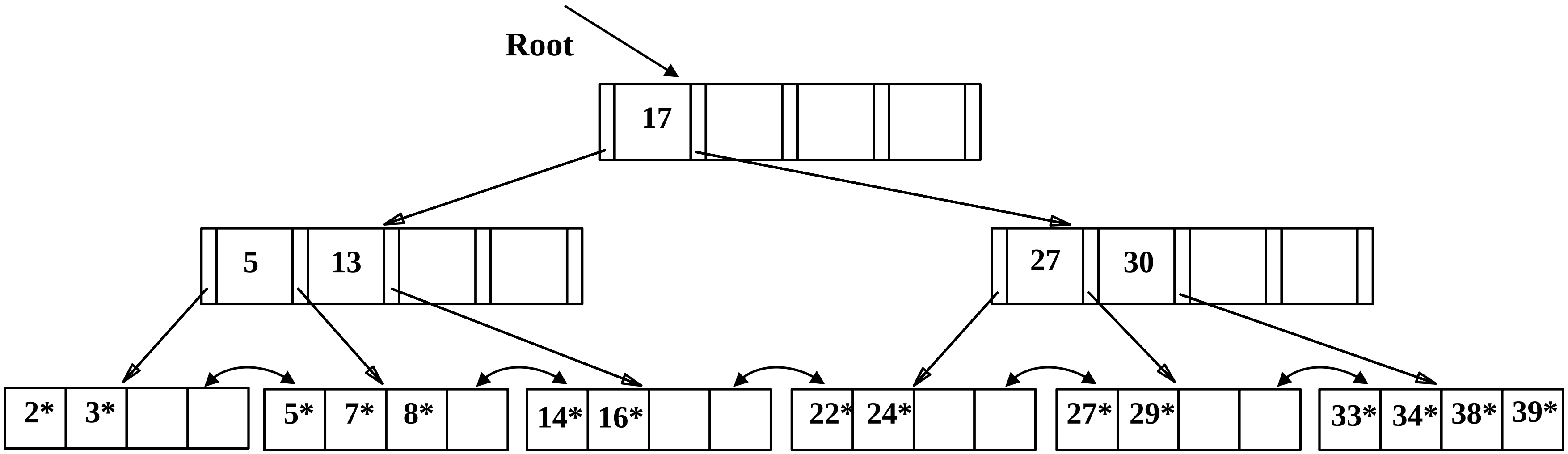
Deleting 19* and then 20*



Deletion of 19* → leaf node is not below the minimum number of entries after the deletion of 19*. No re-adjustments needed.

Deletion of 20* →leaf node falls below minimum number of entries

- re-distribute entries
- copy-up low key value of the second node



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).