Unit-6

- Data on External Storage
- File Organization and Indexing
- Cluster Indexes, Primary and Secondary Indexes
- Index data Structures
- Hash Based Indexing
- Tree base Indexing
- Comparison of File Organizations
- Indexed Sequential Access Methods (ISAM)
- B+ Trees: A Dynamic Index Structure.

Data on External Storage

- **Storage:** Offer persistent data storage, data saved on a persistent storage is not lost when the system shutdowns or crashes.
- **Magnetic Disks:** Can retrieve random page at fixed cost.
- **Tapes:** Can only read pages in sequence. Cheaper than disks; used for archival storage.

- Other types of persistent storage devices:

  Optical storage (CD-R, CD-RW, DVD-R, DVD-RW)

  Flash Memory.
- Each record in a file has a unique identifier called a record id or rid.

Select * from student where id=1002;

DBMS software

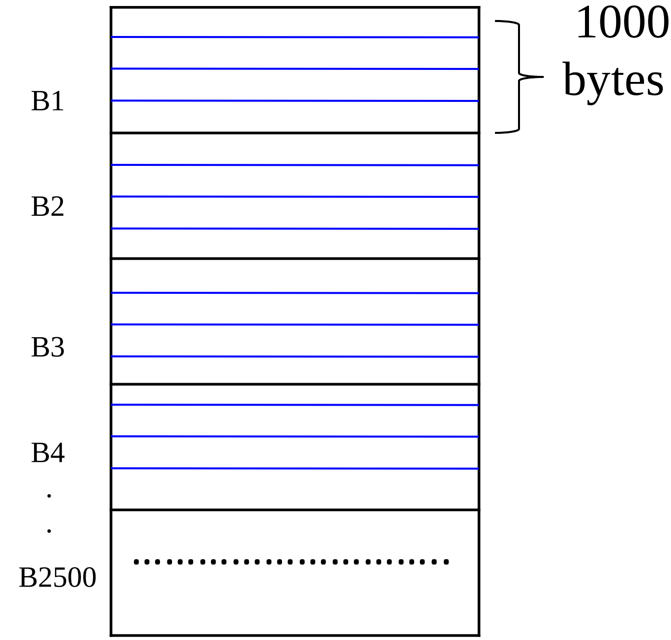
CPU

RAM

Index Table

Key	Pointer

Hard Disk



Example:

Total number of records in student table =10,000

Size of each record= 250 bytes

Size of each block in hard disk= 1000bytes

Total number of records in each block= 1000/250=4 records

Total number of blocks in hard disk= 10,000/4=2,500 blocks

unordered records

Best case = 1

Worst case = 2,500

Average case = 2,500/2 =1250

Id
201
102
1
6
19
5

I/O cost increases

ordered records

Binary search

$\log_2(N) = 12$

N=2,500

Id
1
2
3
4
5
6

- 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.

- The unit of information read from or written to disk is page. Typically the size of a page is 4kb or 8kb.

File Organization and Indexing

- 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.

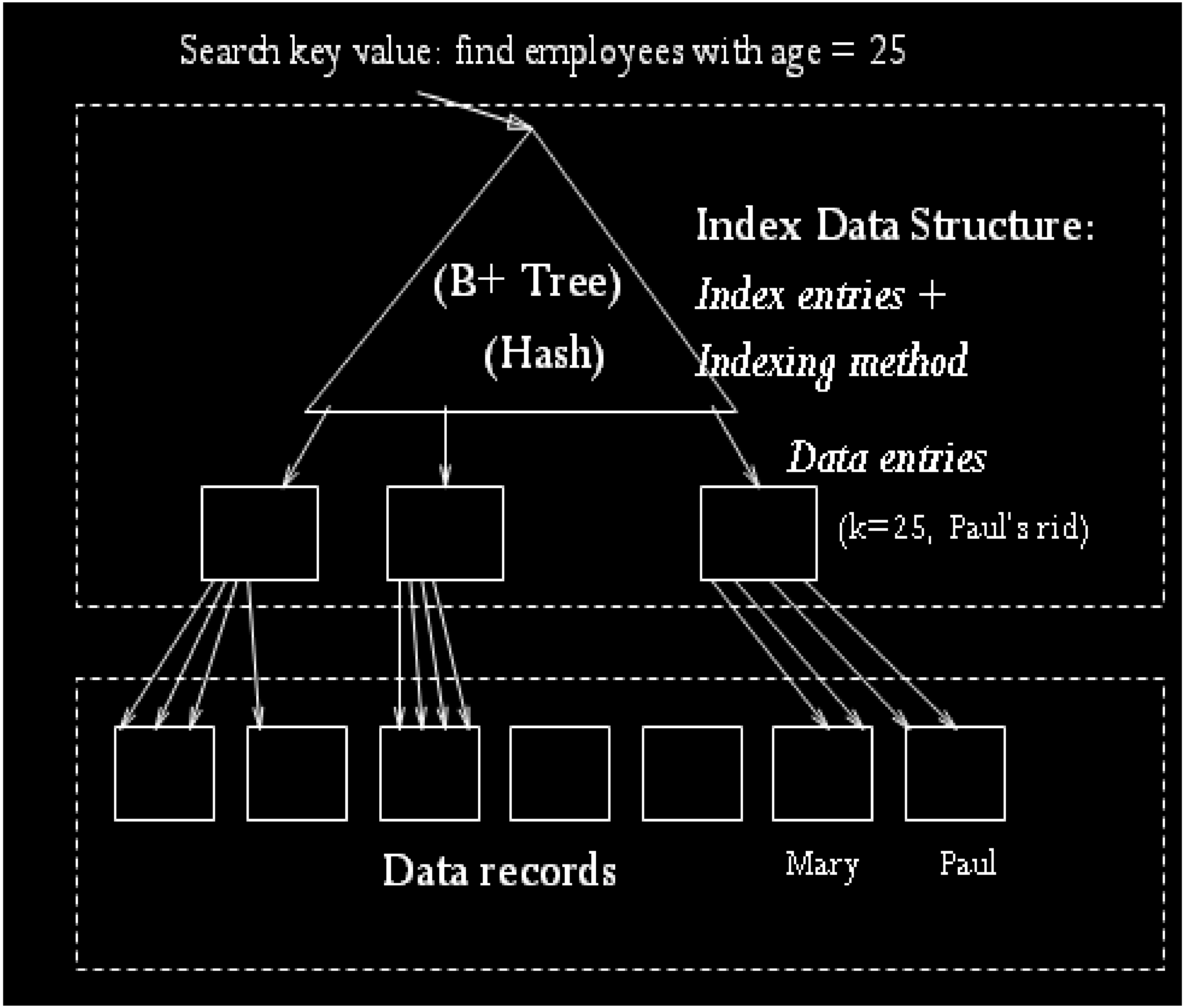
- **3.Indexes:** Data structures to organize records via trees or hashing.
  - For example, create an index on employee age.
  - Like sorted files, speed up searches for a subset of records that match values in certain (“search key”) fields
  - Updates are much faster than in sorted files.

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*).**

- 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)

- An Index is a small table having only two columns.

- The first column comprises a copy of the primary or candidate key of a table (or) any subset of the attributes of a table can be the search key for an index on the relation

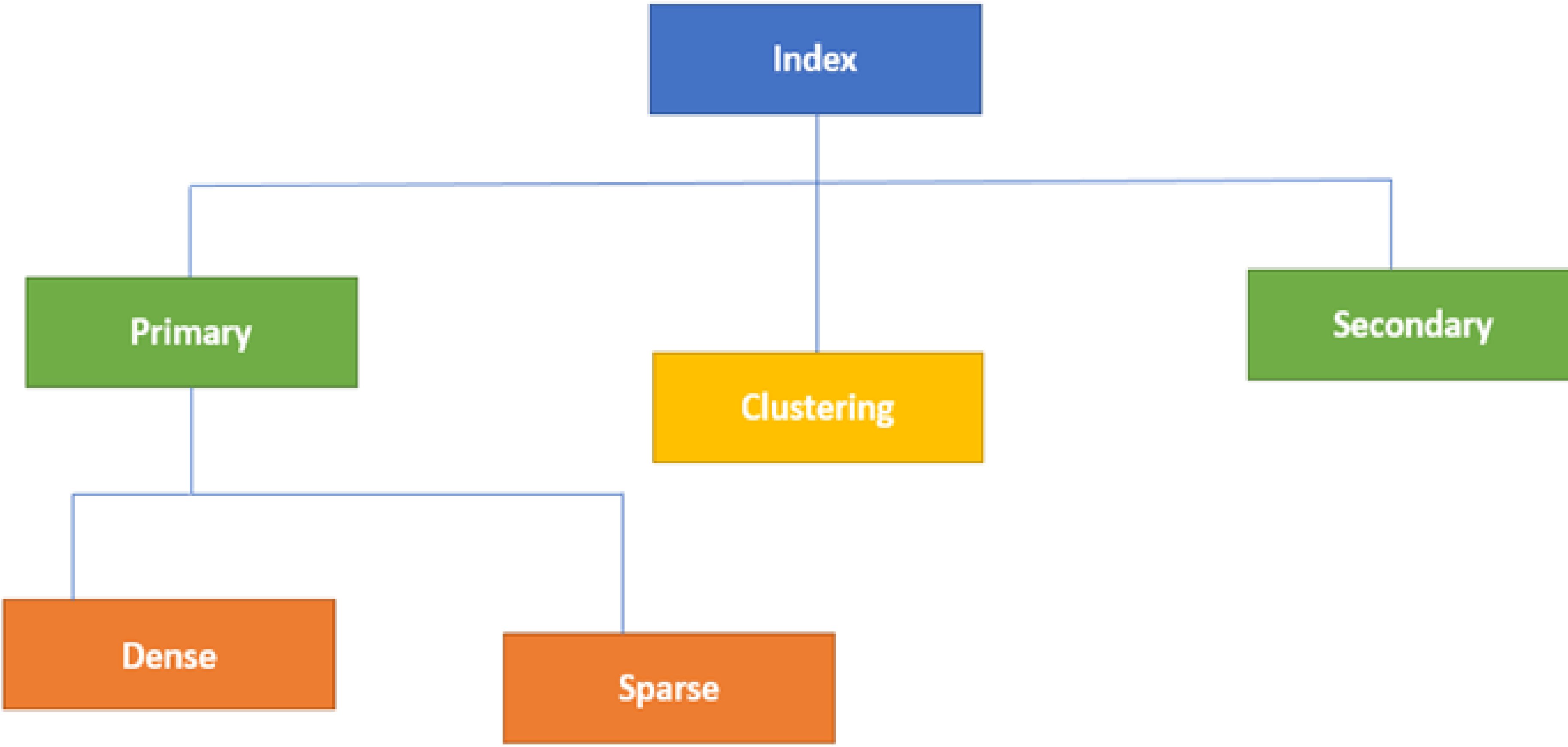
- Its second column contains a set of pointers for holding the address of the disk block where that specific key value stored.

Key	**Pointer**

An index:

•Takes a search key as input
•Efficiently returns a collection of matching records.

Types of Index



Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing.
- These primary keys are unique to each record and contain 1:1 relation between the records.
- These are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

Dense Index

- The dense index contains an index record for **every search key value** in the data file. It makes searching **faster.**
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs **more space** to store index record itself.
- The index records have the search key and a pointer to the actual record on the disk.

UP	
USA	
Nepal	
UK	

UP	Agra	1,604,300
USA	Chicago	2,789,378
Nepal	Kathmandu	1,456,634
UK	Cambridge	1,360,364

Sparse Index

- In the data file, **index record appears only for a few items** in the data file. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

UP	•
Nepal	•
UK	•

UP	Agra	1,604,300
USA	Chicago	2,789,378
Nepal	Kathmandu	1,456,634
UK	Cambridge	1,360,364

Clustering Index

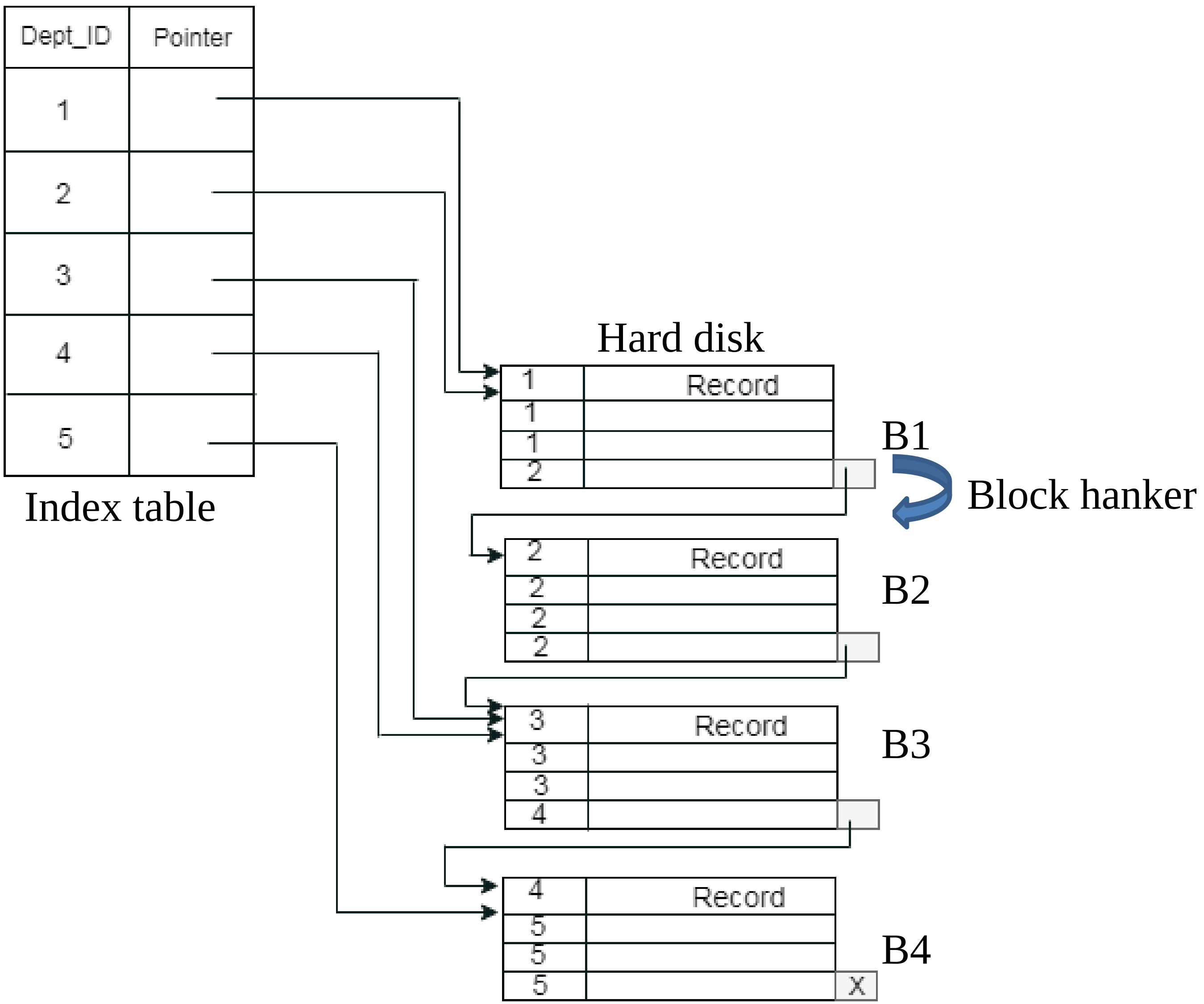
- Clustered index is defined on an ordered data file. The data file is **ordered on a non-key field**.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these groups.

Example:

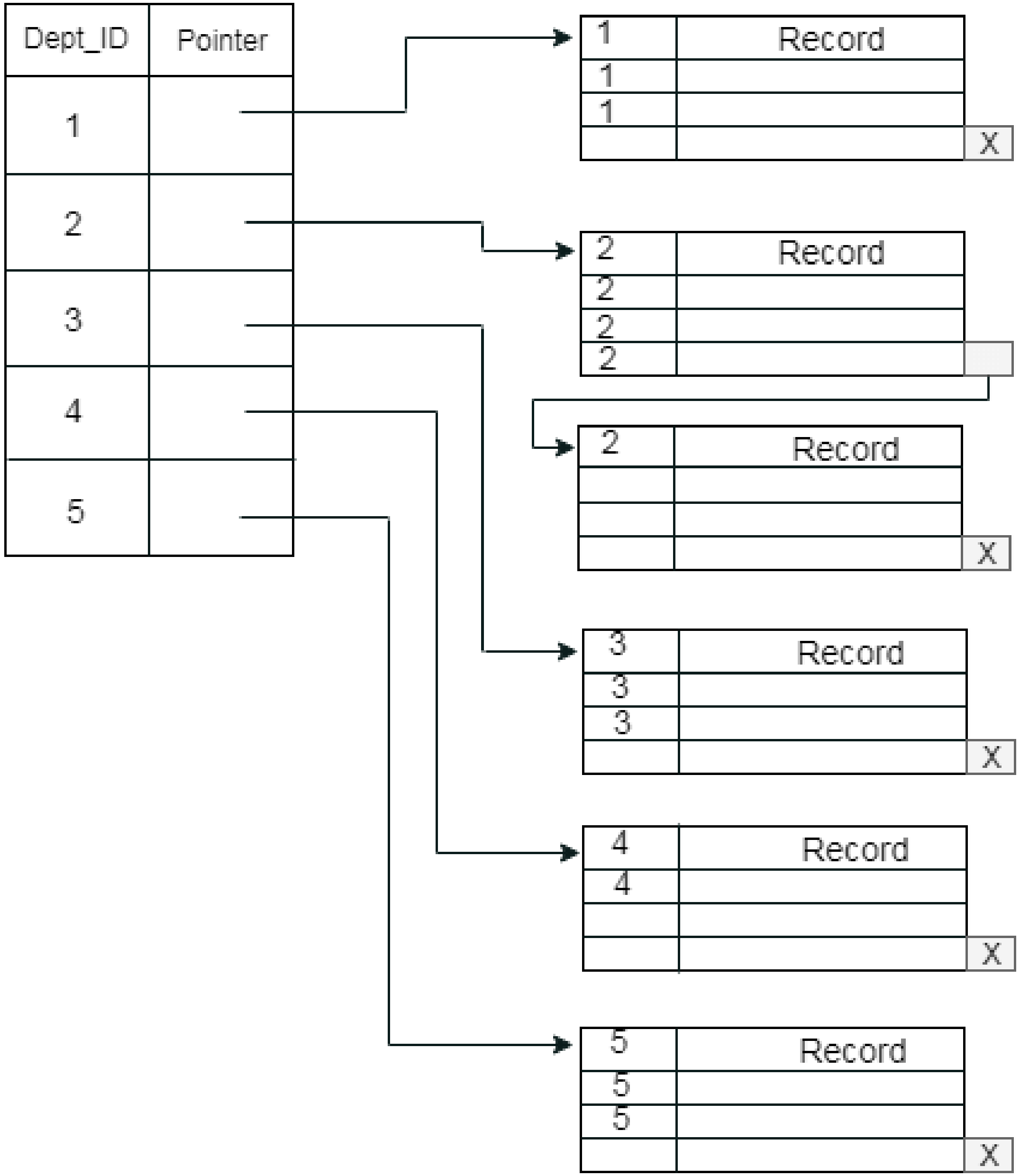
•Suppose a company contains several employees in each department.

•Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole.

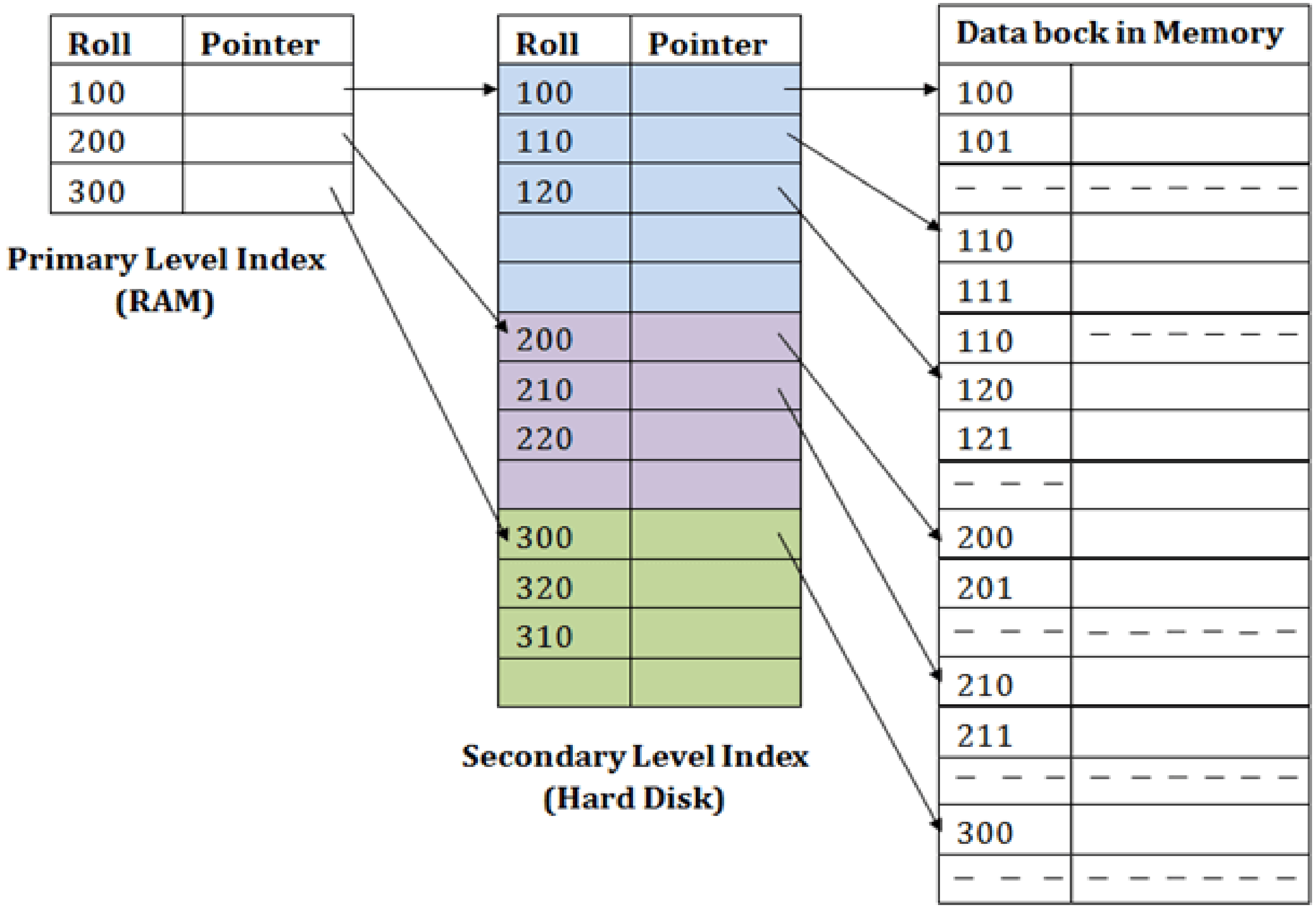
•Here Dept_Id is a non-unique key.



•The previous schema is little confusing because one disk block is shared by records which belong to the different cluster.
•If we use separate disk block for separate clusters, then it is called better technique.



- **Secondary Index:** Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.

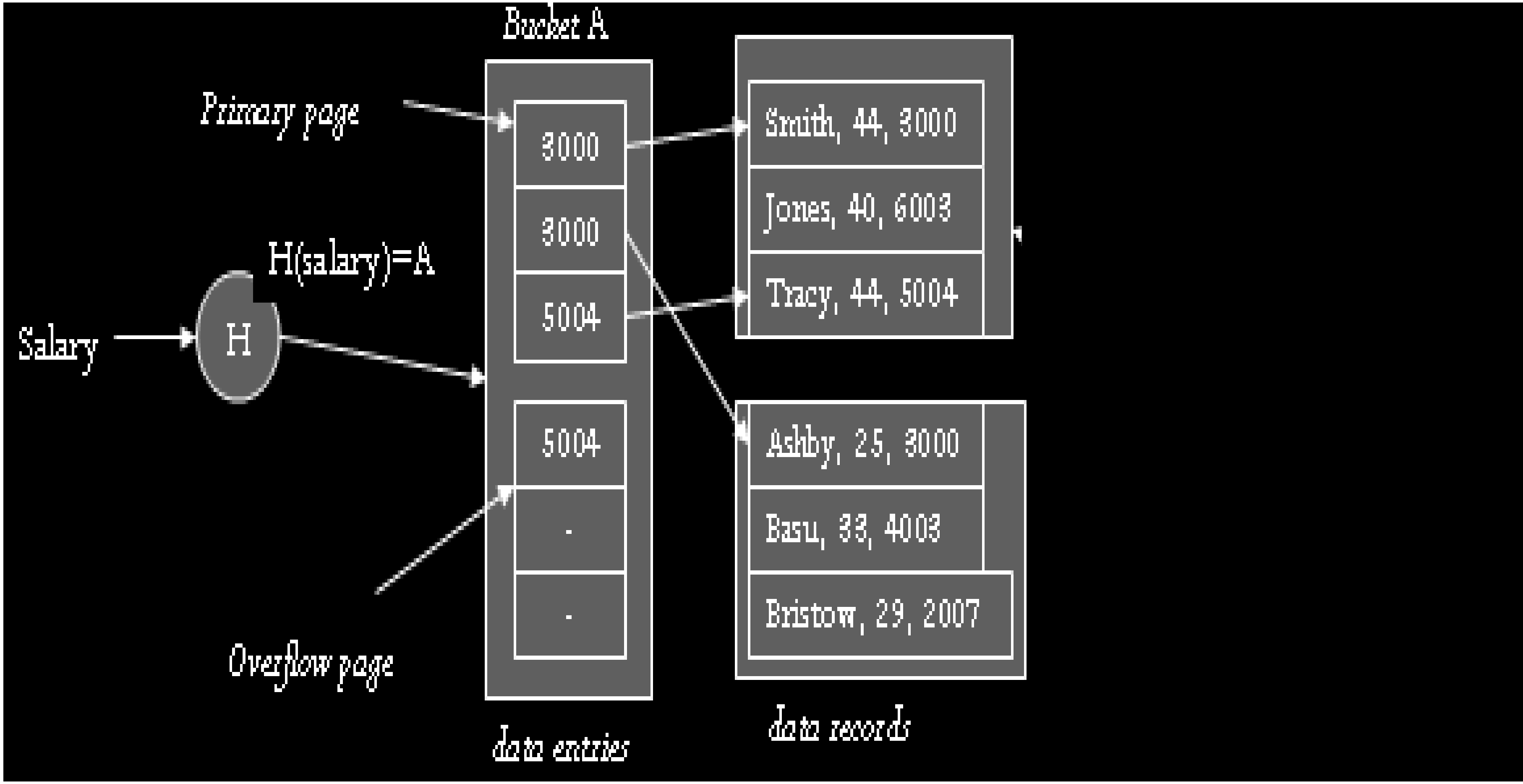


Index data Structures

- Hash Based Indexing
- Tree based Indexing

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*.


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

- Search 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)

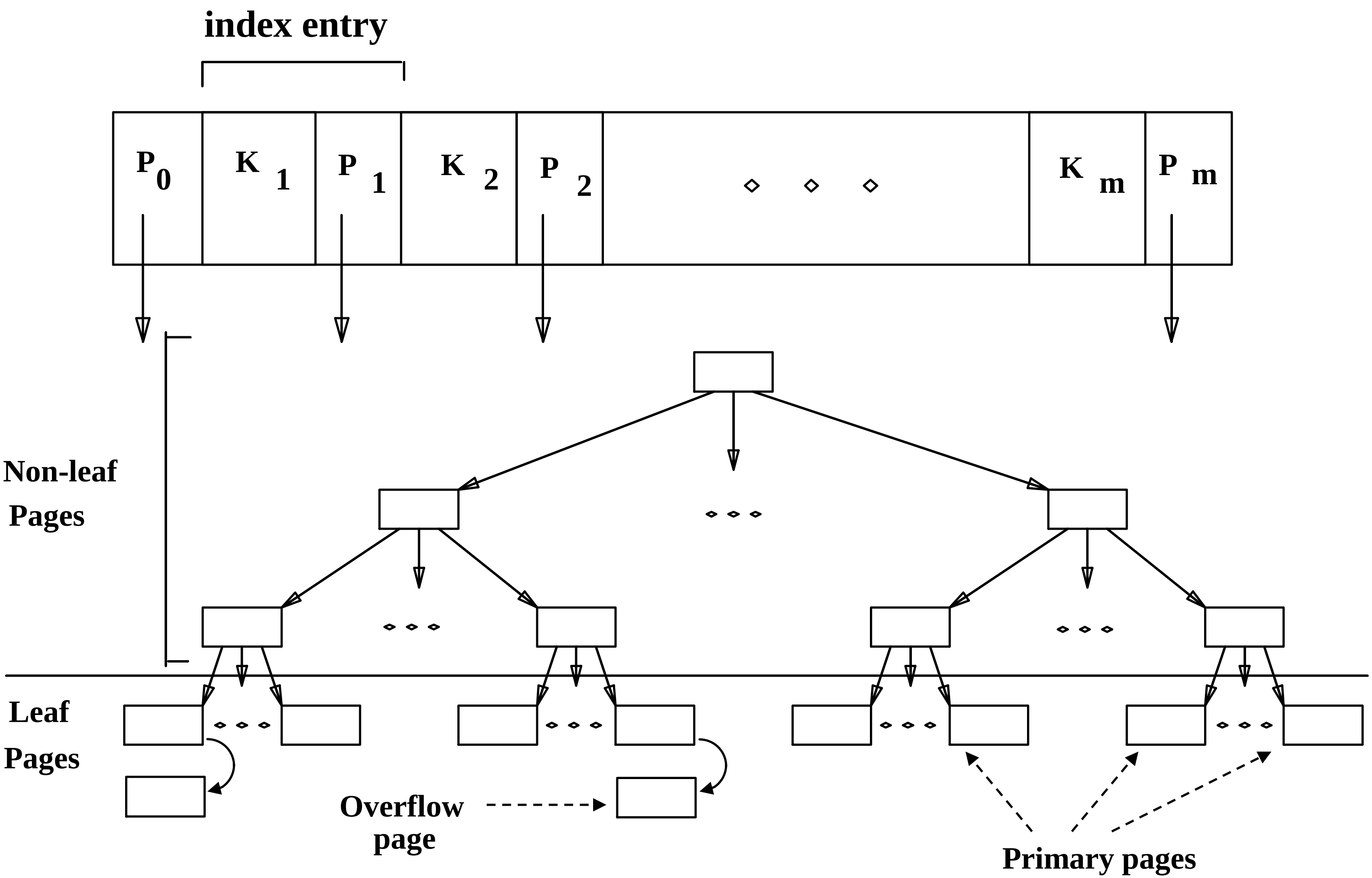
- Insert a record:
  - Apply key value to the hash function -> bucket number
  - If all (primary & overflow) pages in that bucket are full, allocate a new overflow page.
  - Cost: similar to search.

- Delete a record
  - Cost: Similar to search.

Tree based Indexing

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

Indexed Sequential Access Method



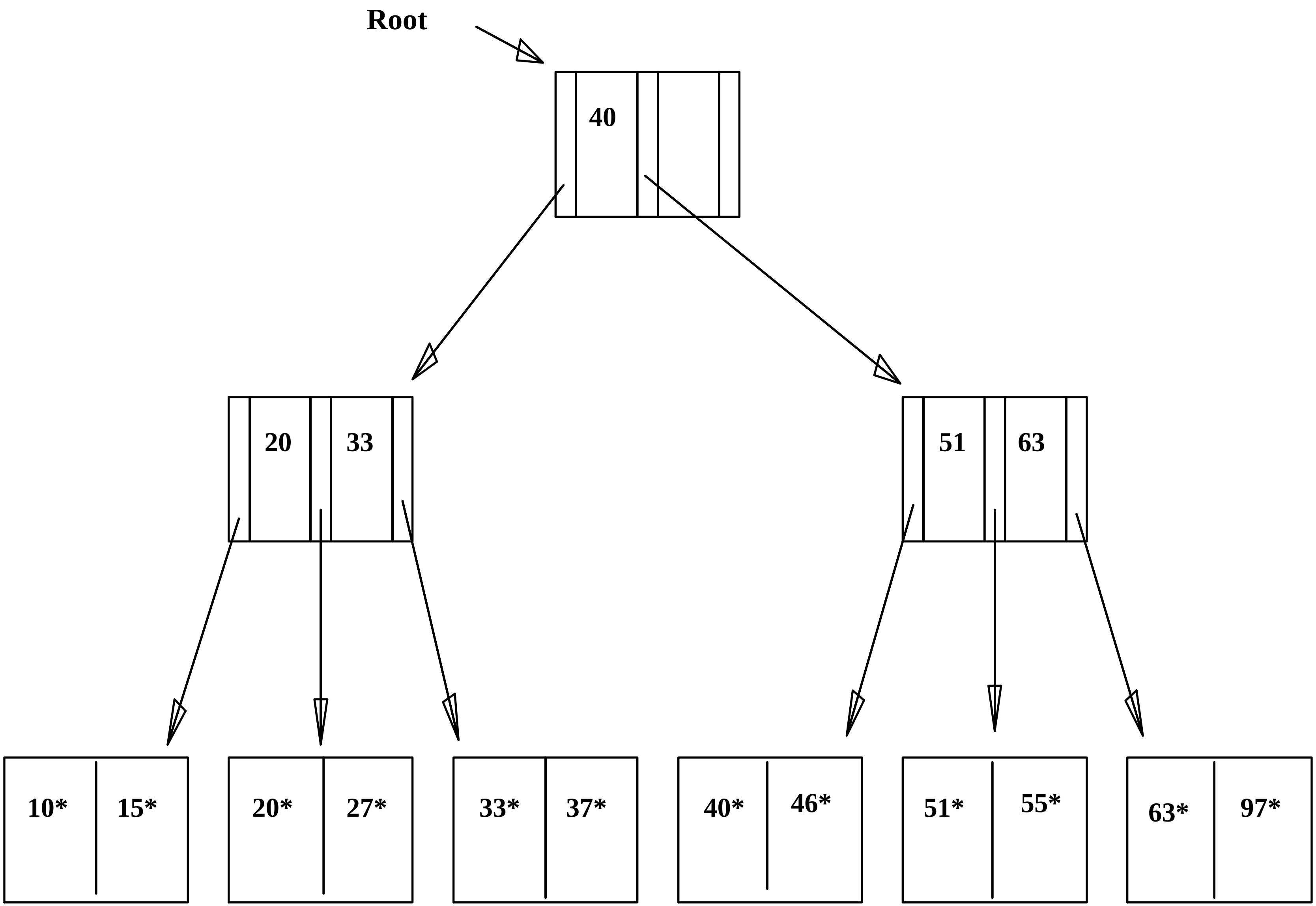
Non-leaf pages contain index entries. Leaf pages contain data entries.

- *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.

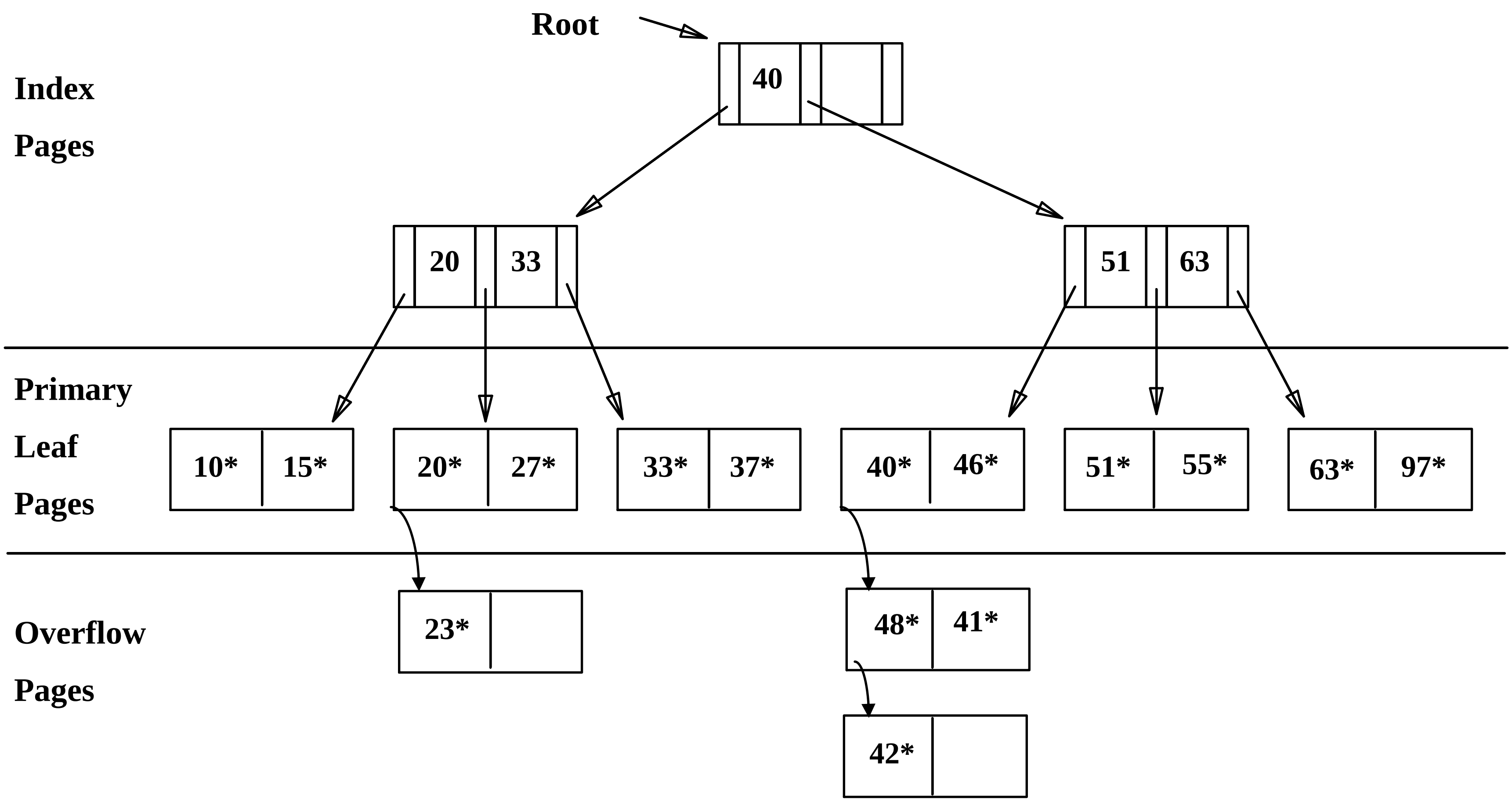
Static tree structure: *inserts/deletes affect only leaf pages*.

- 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.

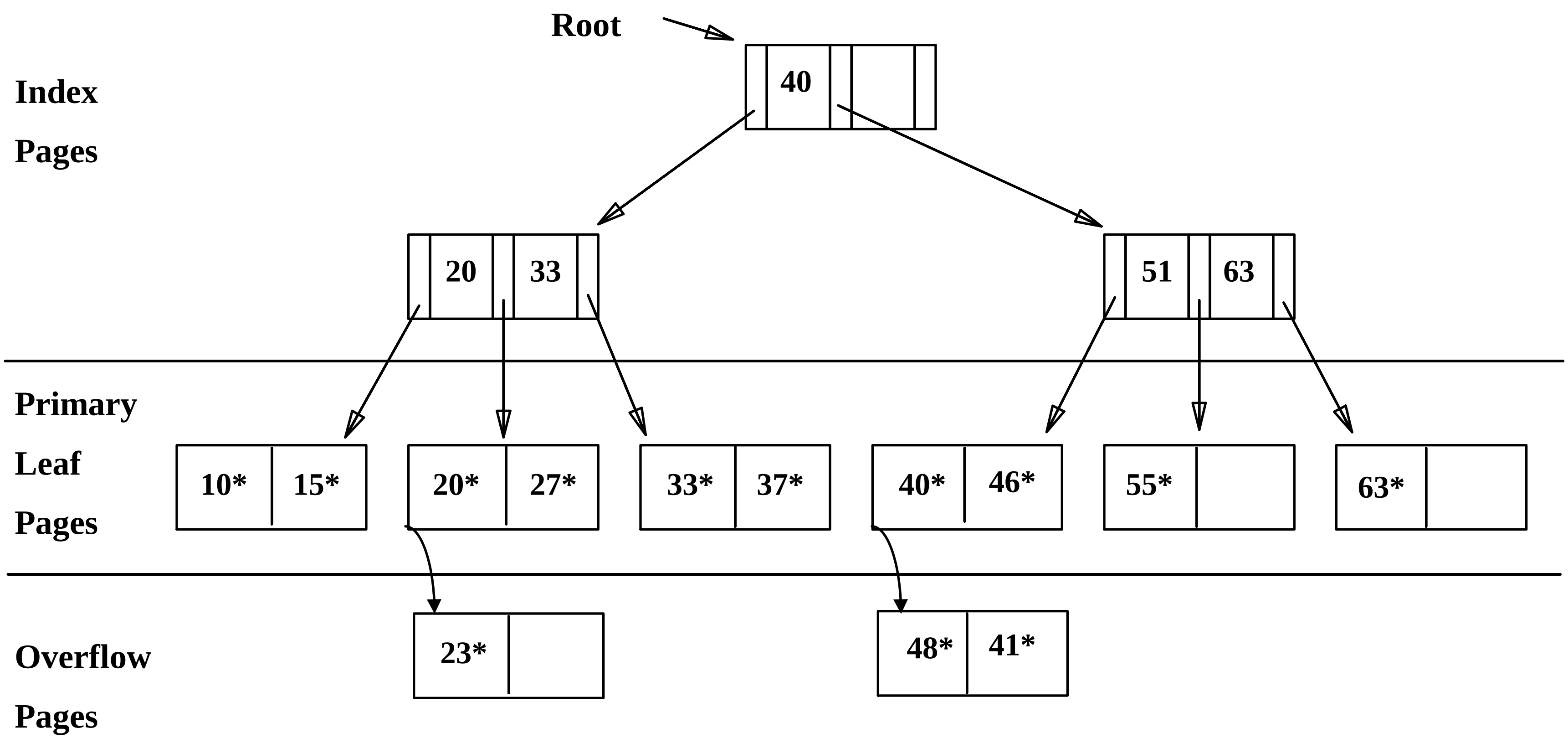

Root
40
20
33
51
63
10*
15*
20*
27*
33*
37*
40*
46*
51*
55*
63*
97*

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



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

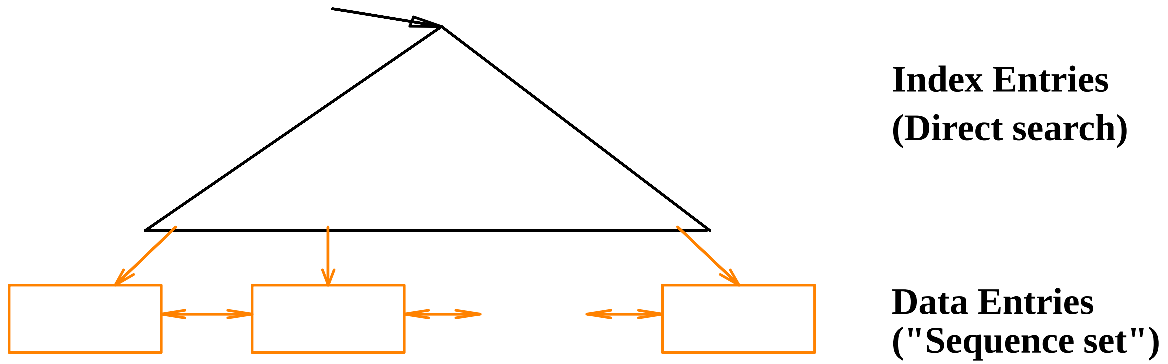
...Then Deleting 42*, 51*, 97*



note that 51 still appears in the index page!

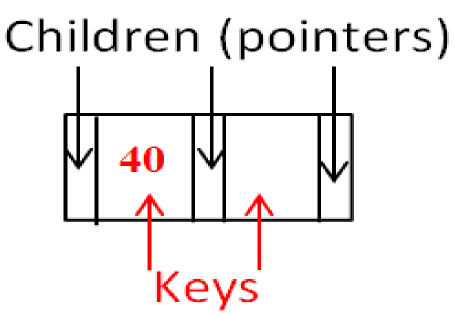
B+ Tree

- Dynamic structure - can be updated without using overflow pages!
- Main characteristics:
  - Minimum 50% occupancy (except for root).
  - Supports equality and range-searches efficiently.

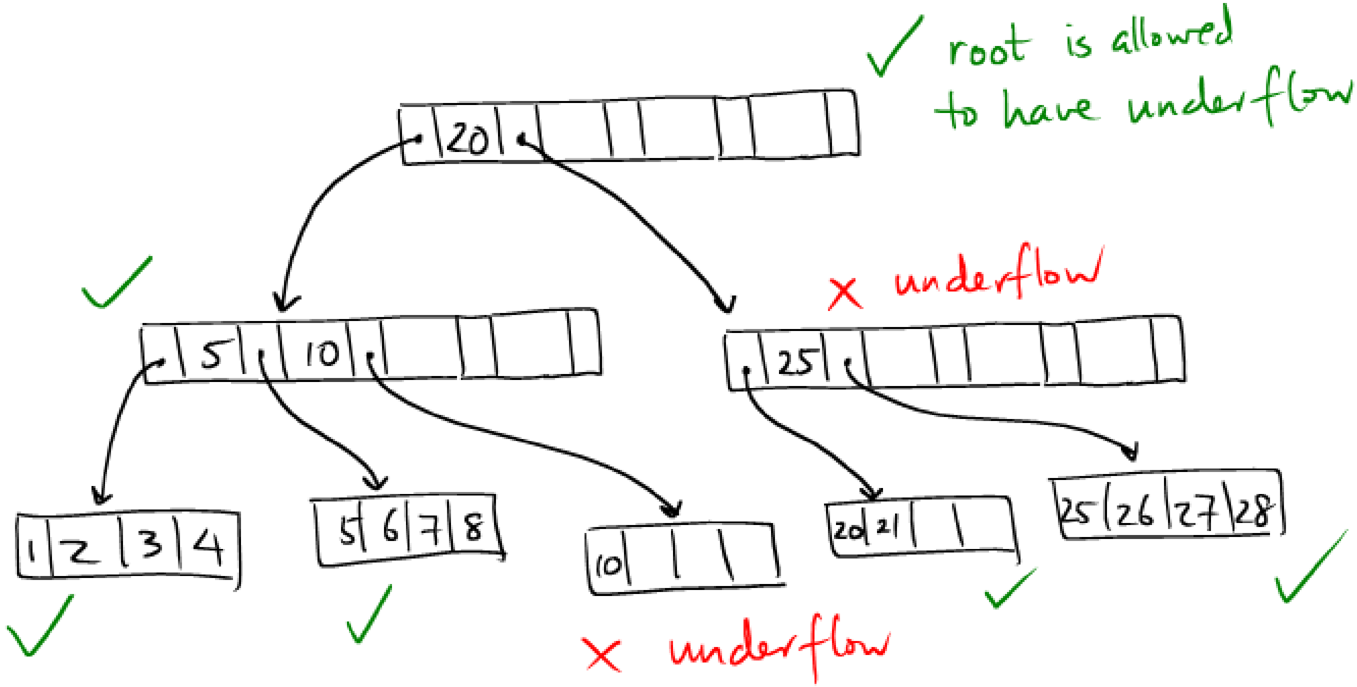


B+ tree with order m

- m is number of children
- Root node should have two children (at least 1 search key)
- Other nodes should have minimum ceil(m/2) children (at least ceil(m/2) - 1 search keys)



Example:


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

Inserting a Data Entry into a B+ Tree

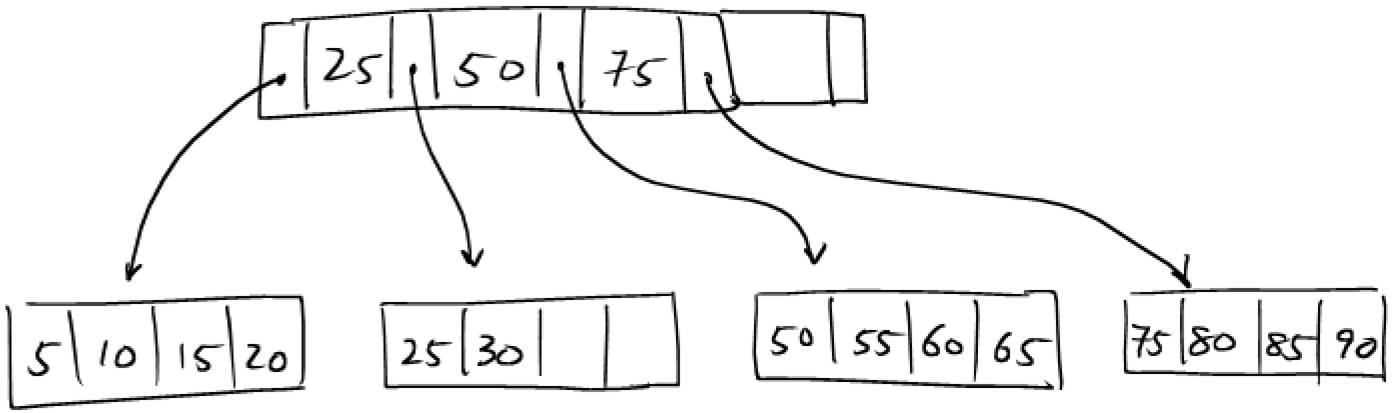
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To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

The tree distinct cases are:

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

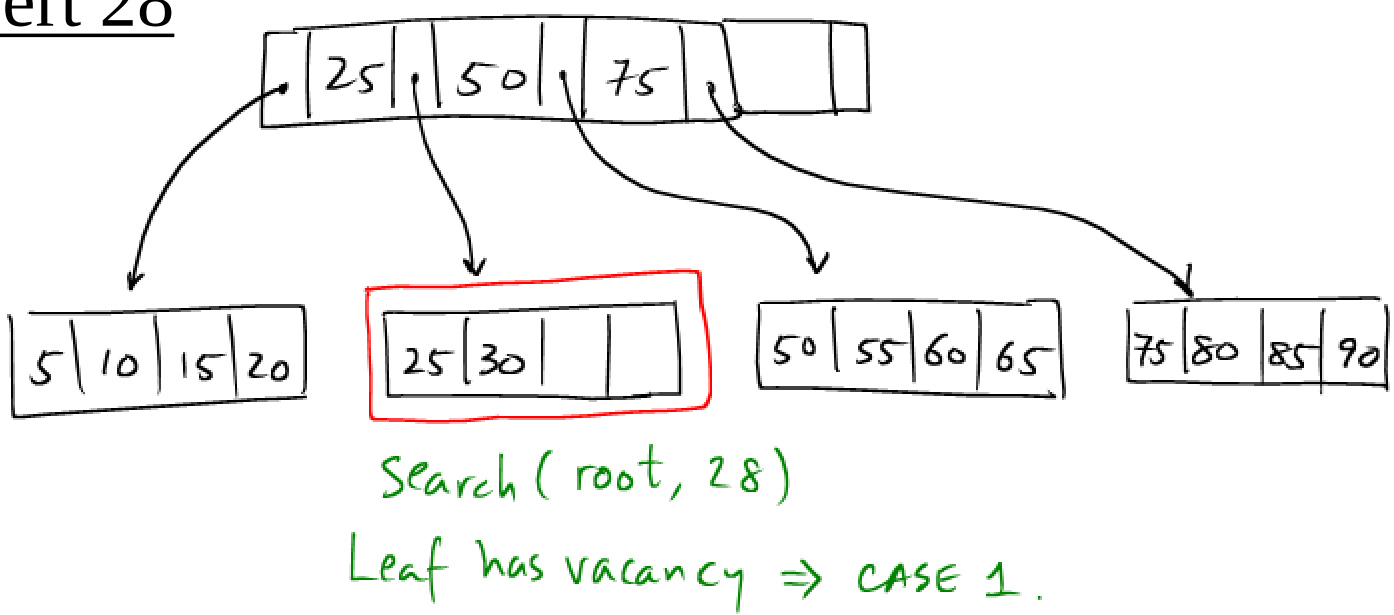
2. the target node is full, but its parent has space for one more key

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

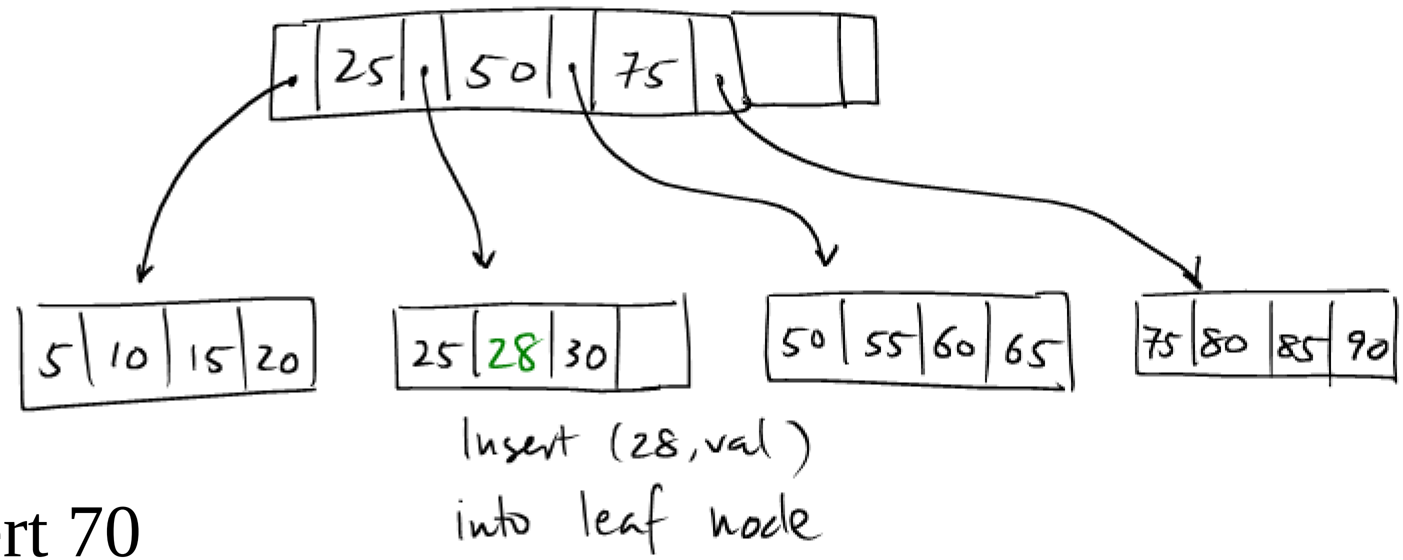


25
50
75
5
10
15
20
25
30
50
55
60
65
75
80
85
90

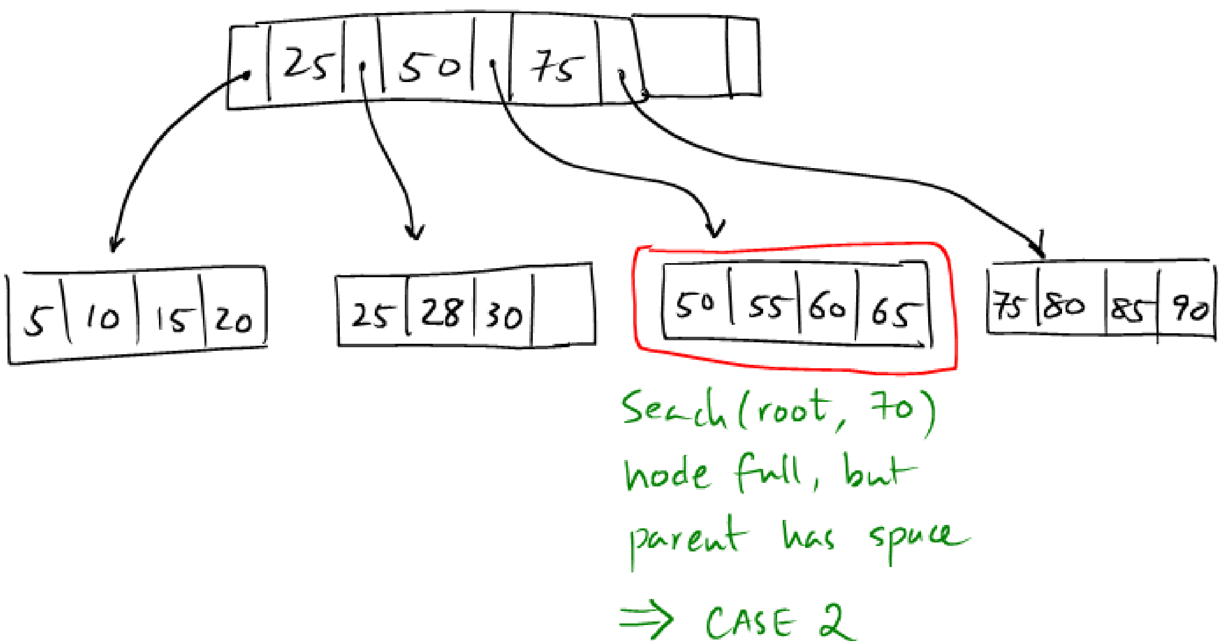
insert 28



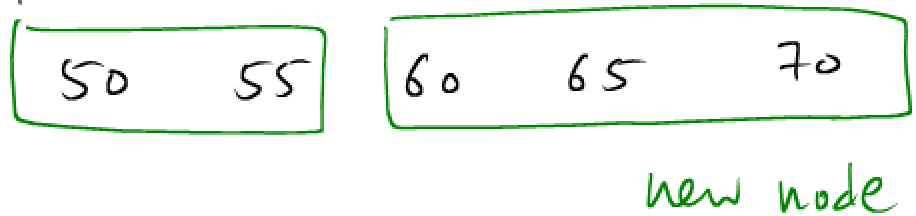
25
50
75
5
10
15
20
25
30
50
55
60
65
75
80
85
90
Search (root, 28)
Leaf has vacancy ⇒ CASE 1.

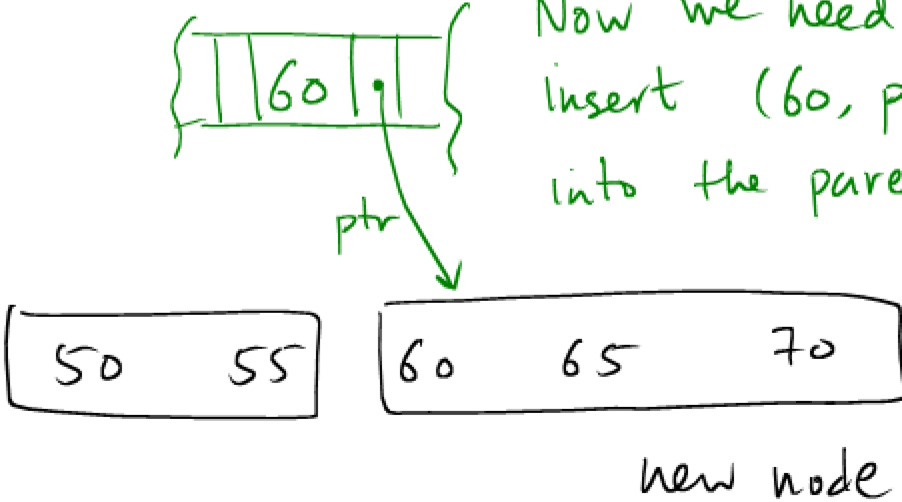

25
50
75
5
10
15
20
25
28
30
50
55
60
65
75
80
85
90
Insert (28, val)
into leaf node

insert 70

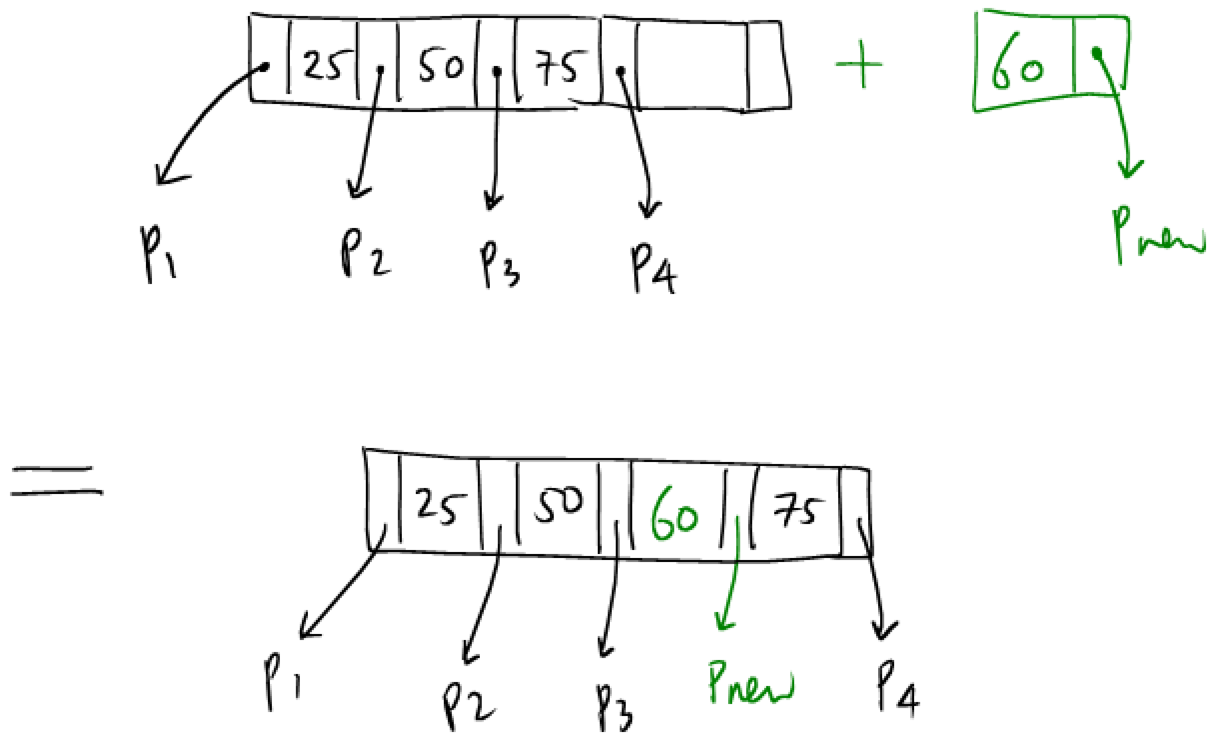

25
50
75
5
10
15
20
25
28
30
50
55
60
65
75
80
85
90
Search (root, 70)
node full, but
parent has space
⇒ CASE 2

Keys are distributed

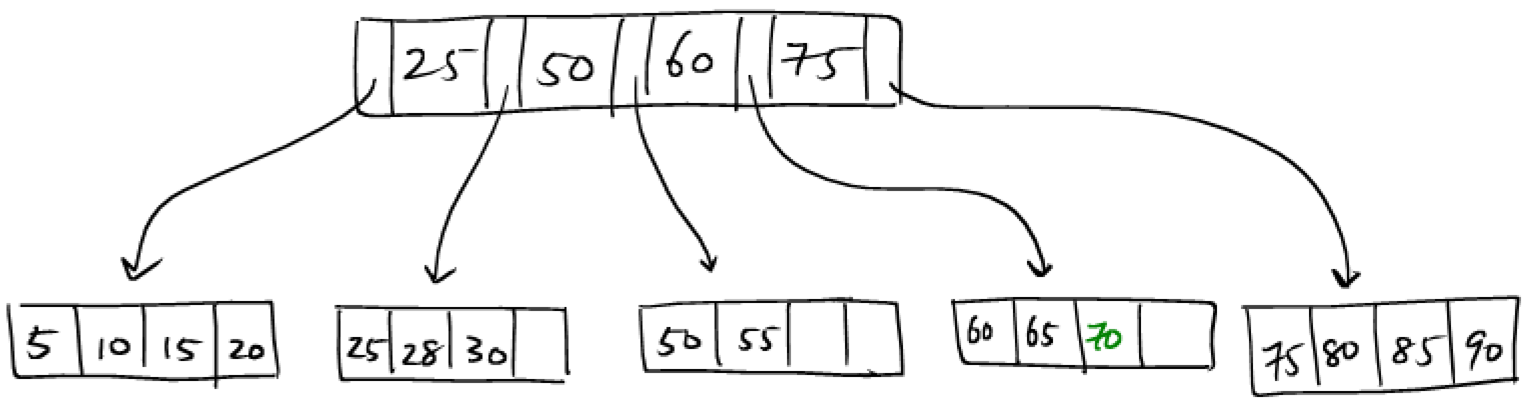


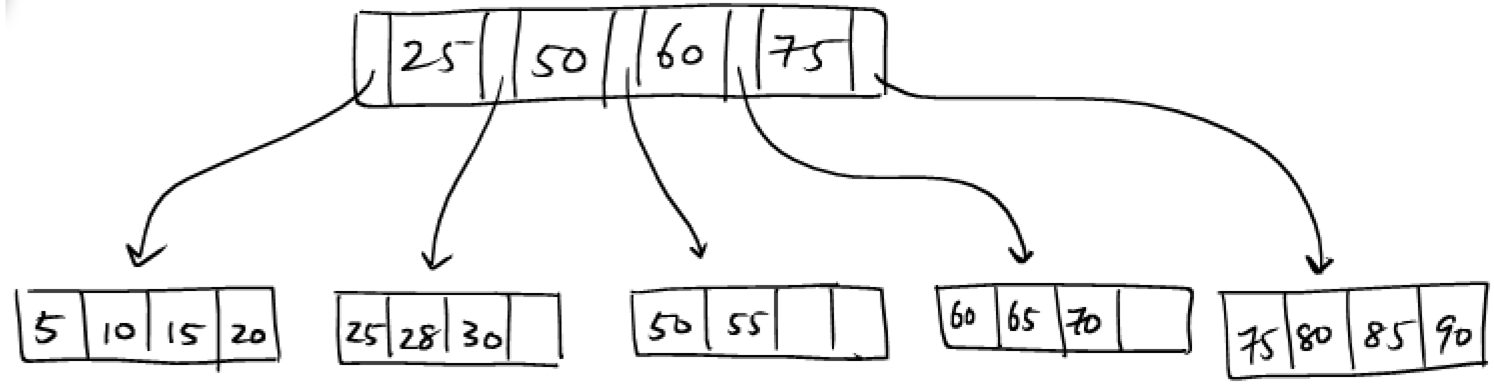


Now we need to insert (60, ptr) into the parent.

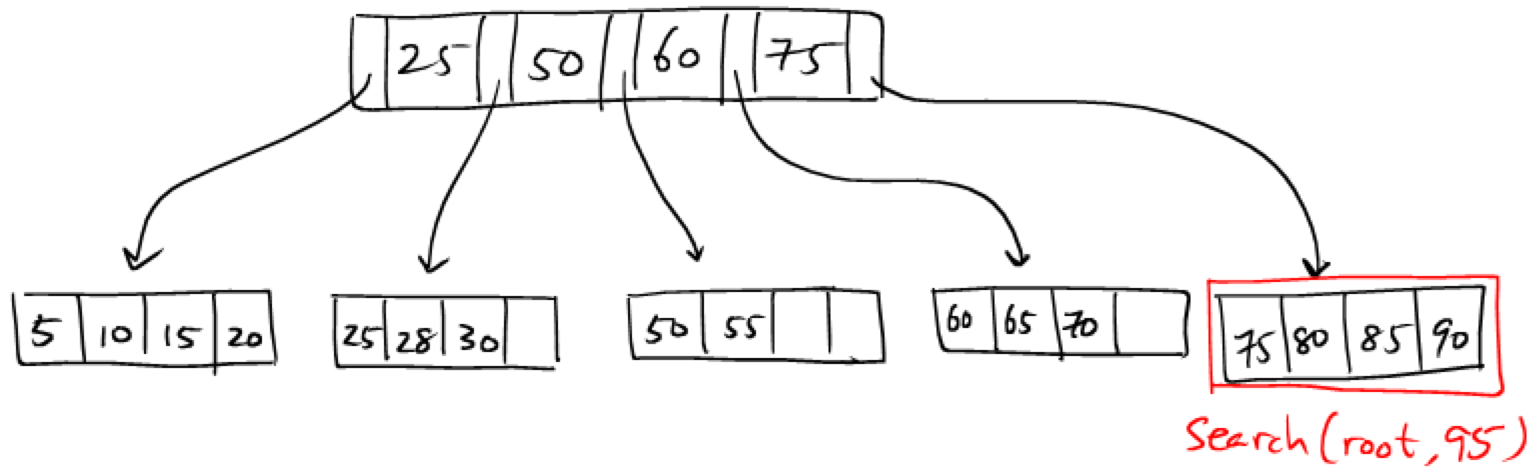

25
50
75
+
60
P1
P2
P3
P4
Pnew
=
25
50
60
75
P1
P2
P3
Pnew
P4

New tree :

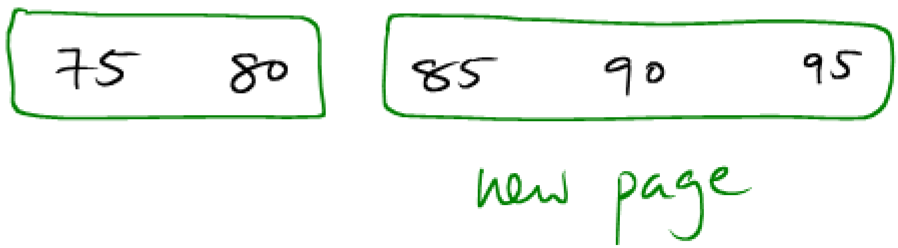

25
50
60
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90

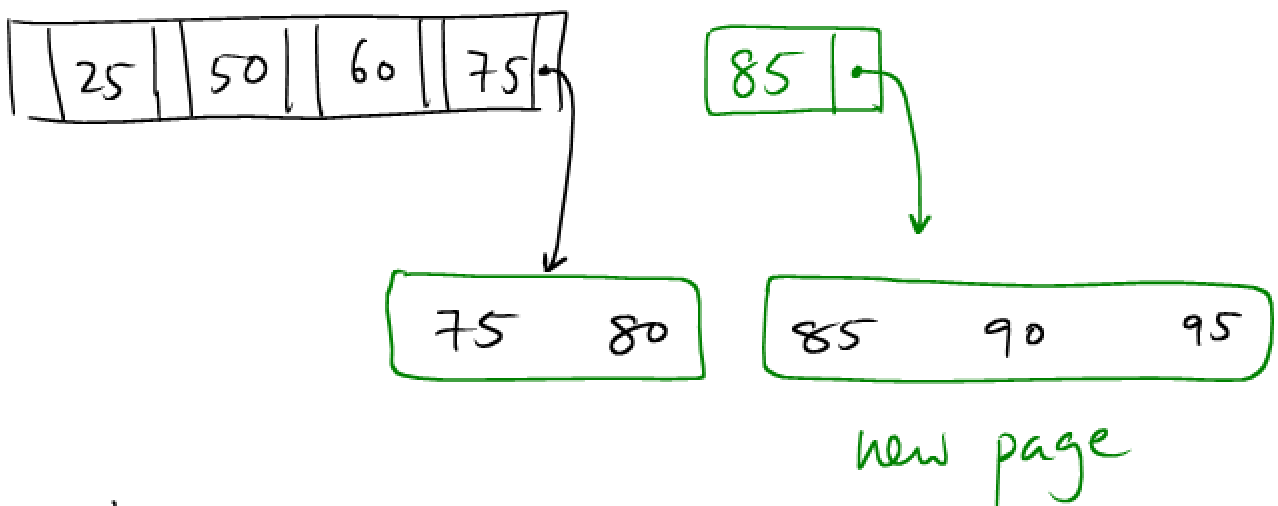

25
50
60
75
5
10
15
20
25
28
30
50
55
60
65
70
75
80
85
90

<u>insert 95</u>


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.

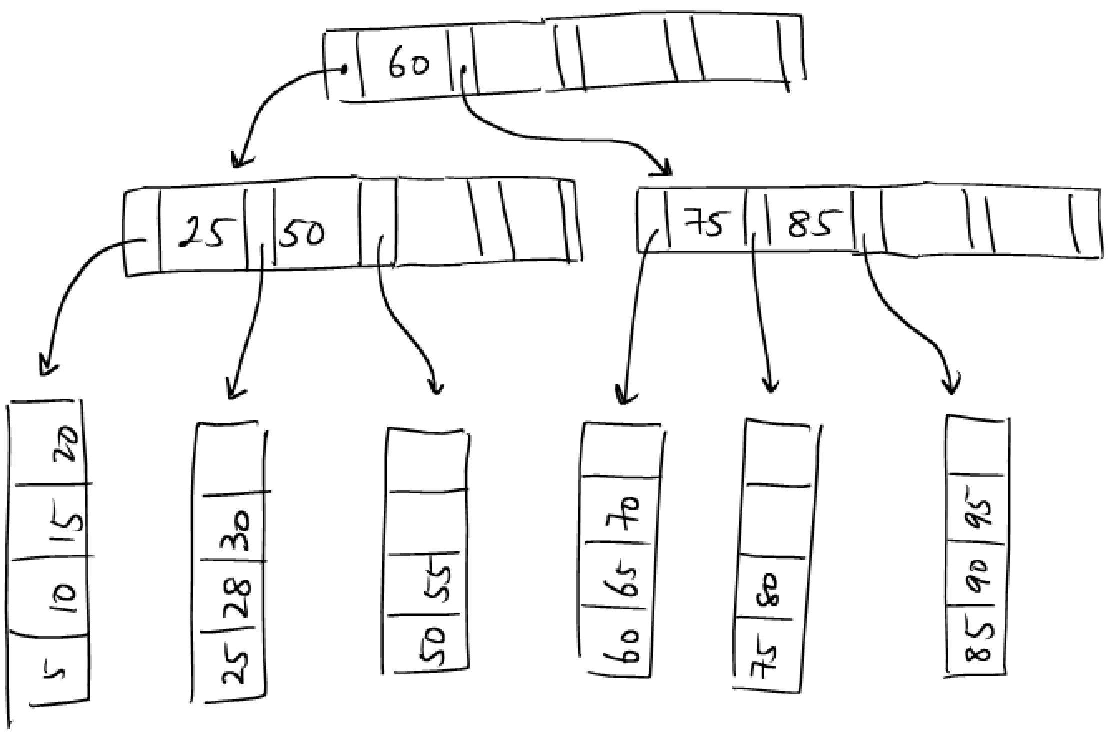
all_keys:

25 50 60 75 85

distribute to left

middle key

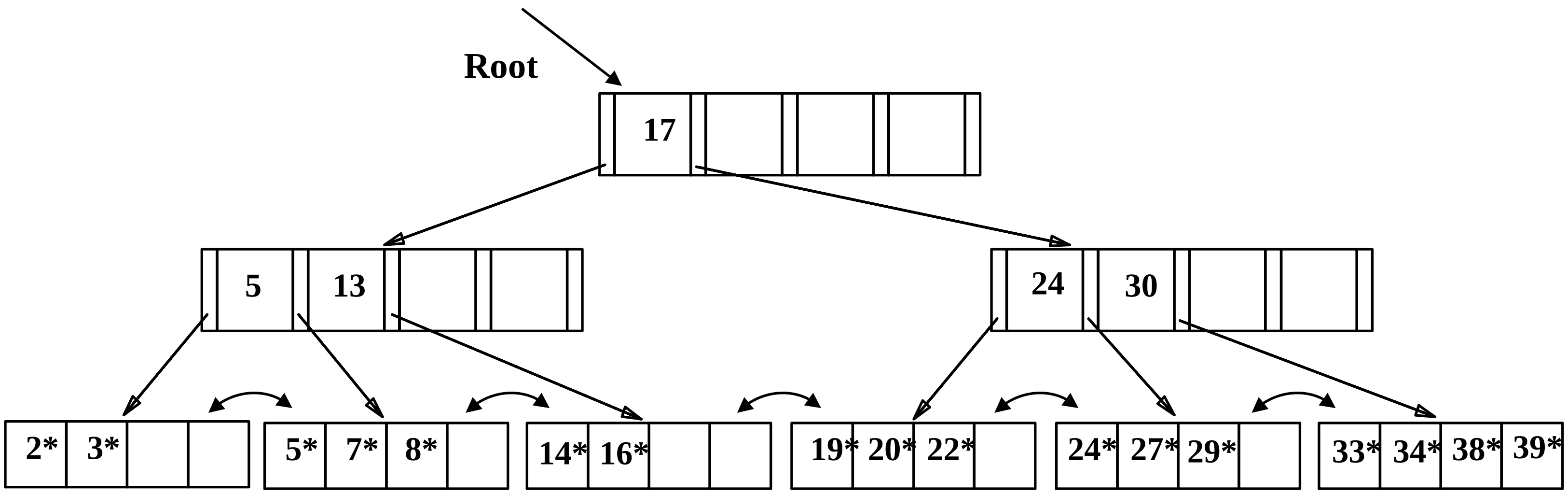
new node, distribute to right



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 **d-1** entries,
    - 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.

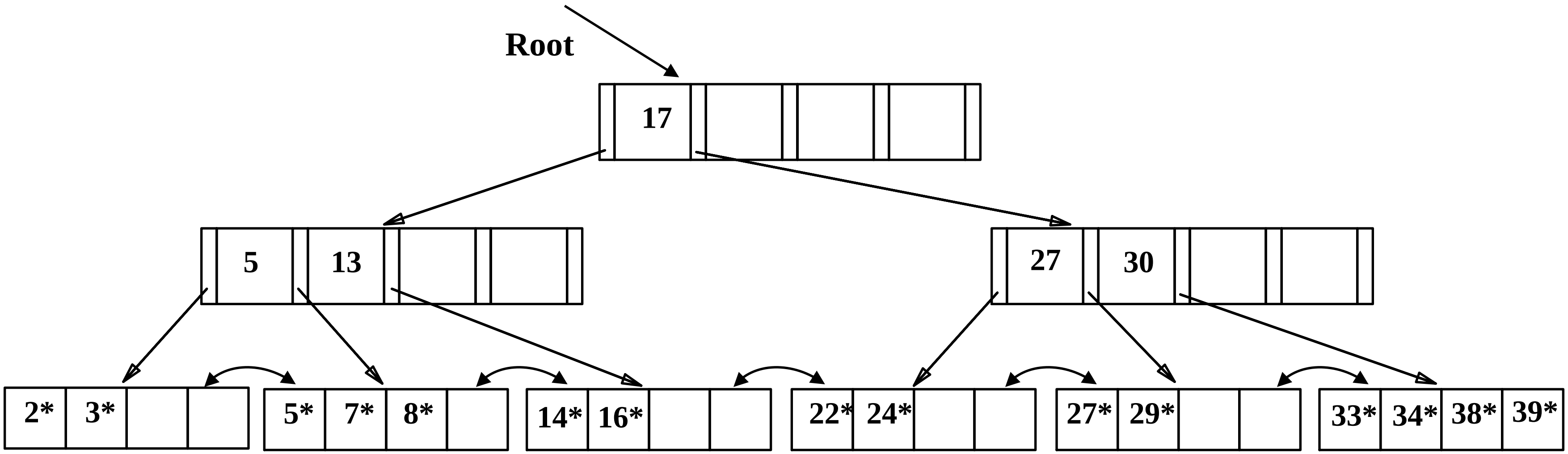
Deleting 19* and then 20*



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*