Ch 10. Tree-Structured Indexes (Short Version)

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

SKKU VLDB Lab.

(http://vldb.skku.ac.kr/)

Very Large Data Bases

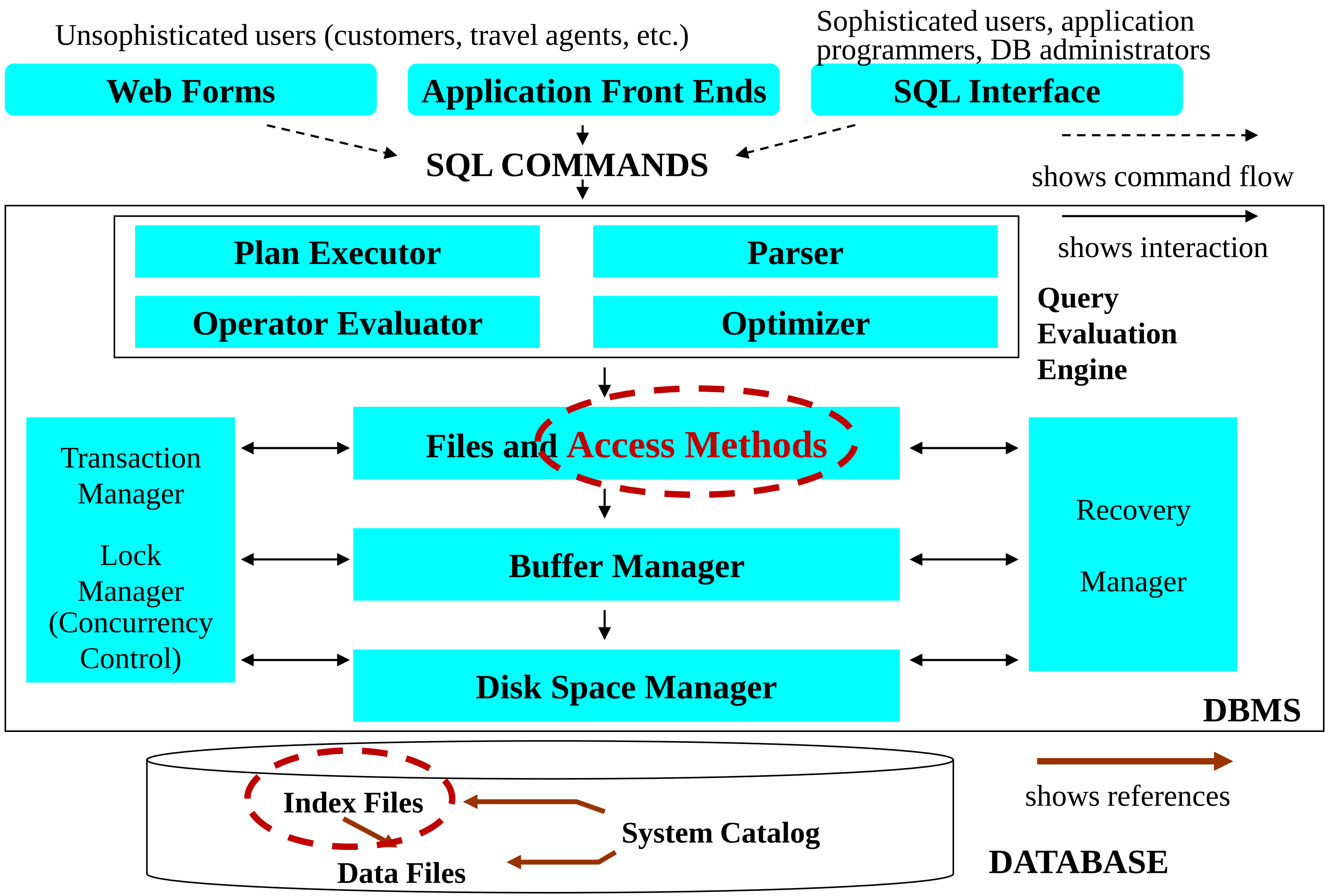


Figure 1.3 Anatomy of an RDBMS

What you should know from this chapter

- Intuition behind tree-structured index
- B+ tree
  - Balanced or Rudolf Bayer??
  - Data structure and related concepts
- How search/insertion/deletion works in B+ tree?
- Cost of B+ tree based access method

- Recommended Readings
  - Pat Helland, Write Amplification vs. Read Perspiration, CACM Nov. 2019

Indexing Problem

- Finding a small subset from a big set with some hints
  - E.g. Dictionary, Library, Bank, Google search, face recognition, music matching@Melon
  - Search key: word, title/author/keyword, account_id/customer name, search keys

The Sumerian Writing System in Mesopotamia:
The first writing of mankind on durable storage media
(i.e., Clay Tablet): to record transactions
(See CH 6. Memory Overload @ Homo Sapiens)

- Solutions: Scan all elements of big set, sorting, librarian's solution?, B-tree index, inverted index
- Other indexing techniques used in DB
  - Bitmap(ch25), R-tree (multi-dimensional, ch28), hash index (ch11), bloom filter (RocksDB, Hash Join)

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - author catalog in library, index @ book: sorted thus binary search
- Search Key - set of attributes used to look up records in a file.

TEST (A,B,C)

```
SELECT B    /* point query */
FROM  TEST
WHERE A = 50000;
```

```
SELECT *    /* range query */
FROM  TEST
WHERE B between 20 and 50;
```

- An **index file** consists of records (called data entry or index entry) of the following form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices: tree vs. hash index

Problem: How to quickly find the records satisfying the given search?

Assume **TEST (A,B,C)** table in SCRIPT (ch10.sql)

- 1M records of each 650 bytes
- 10 records / 8KB page
- Thus, 100,000 pages

```
SELECT B    /* point query */
FROM  TEST
WHERE A = 50000;
```

```
SELECT B     /* range query */
FROM  TEST
WHERE A between 50001 and 50100;
```

Data Page 1

Data Page 2

⋮

Page#: 5000

SLOT #: 10

(50000,50000, 'XX…XX';

⋮

Data Page 100000

TEST SEGMENT
(Heap File)

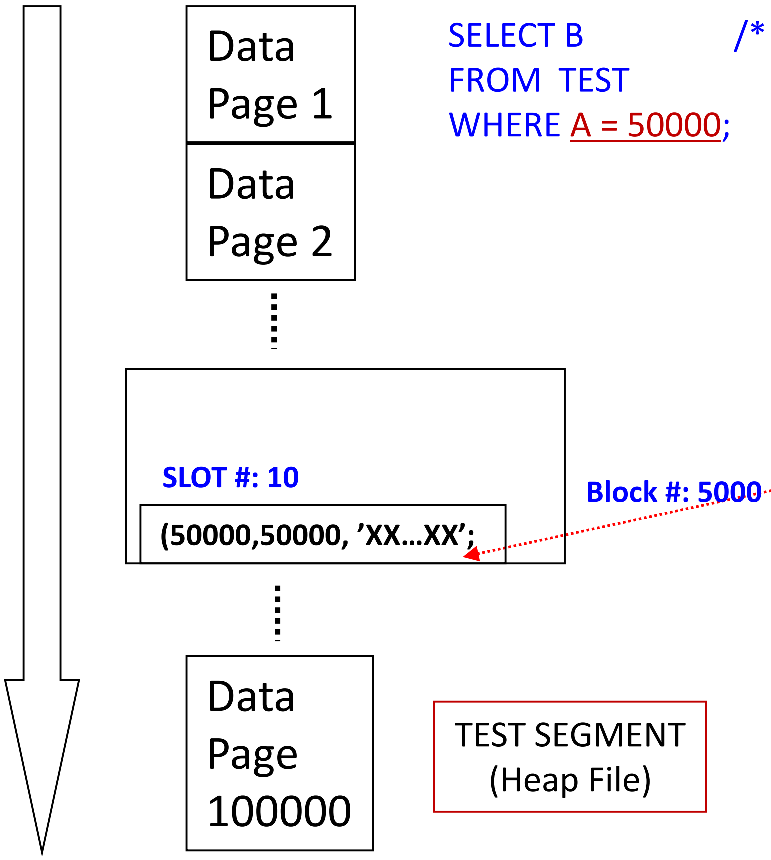
- Solution 1
  - Full table scan
- Solution 2
  - We can use "Binary Search" when all the records are sorted according to the search key column
  - This assumption does not hold in most case
    - ✓ Heap file: for both primary and all other columns

Index-based Access on Heap file

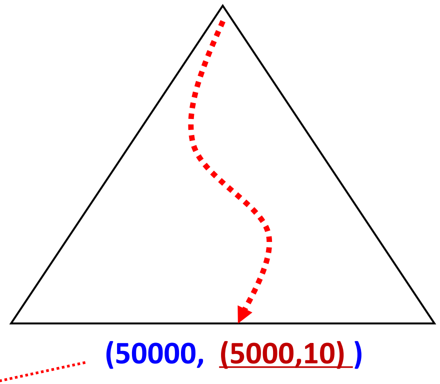
Assume **TEST (A,B,C)** table in SCRIPT

B-tree Index on **TEST(A)**

SEARCH KEY: 50000




```
SELECT B            /* point query */
FROM  TEST
WHERE A = 50000;
```



- Index entry (or data entry): (key-value, record-id)
- Record-id = (page#, slot#)

TEST SEGMENT
(Heap File)

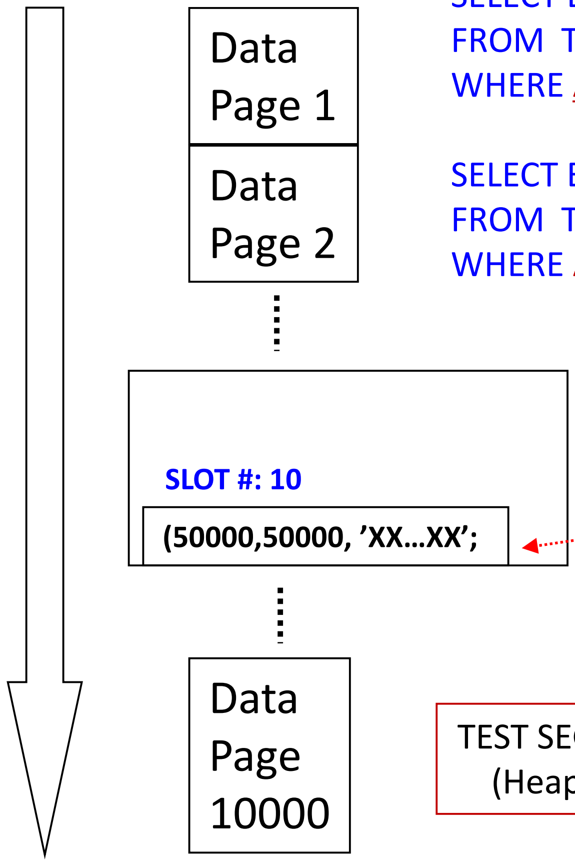
Cost:

- **Full Table Scan:** 100,000 Block Accesses
- **Index:** (3~4) + **Data Block Accesses**
  - Point query: 1
  - Range queries: **depending on range and index's CLUSTERING FACTOR**

Index-based Access on Heap file (2)

Assume TEST (A,B,C) table in SCRIPT

B-tree Index on TEST(A)



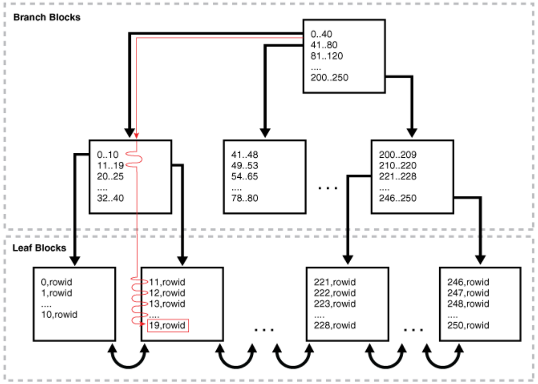
Block #: 5000

```
SELECT B              /* point query */
FROM  TEST
WHERE A = 50000;

SELECT B              /* range query */
FROM  TEST
WHERE A BETWEEN 50001 and 50101;
```

TEST SEGMENT
(Heap File)

SEARCH KEY: 50000



(50000, (5000,10))

```
SQL> set autot on
SQL> select b  from test where a = 50000;
…
Execution Plan
---------------------------------------------------------------------------------------
| Id  | Operation                     | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |        |     1 |     8 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | TEST   |     1 |     8 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN            | TEST_A |     1 |       |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("A"=50000)

Statistics
----------------------------------------------------------
…
          5  consistent gets
          0  physical reads
…
1  rows processed
```

TEST_A SEGMENT
(Index File)

Source: https://docs.oracle.com/database/121/TGSQL/tgsql_optop.htm#TGSQL233

Index Evaluation Metrics

- Which access types supported efficiently?
  - Equality: records with a specified value in the attribute
  - Range search: records with an attribute whose value is in a specified range of values.

- **Trade-off**: Search time **vs.** Space/Insertion/Deletion overhead

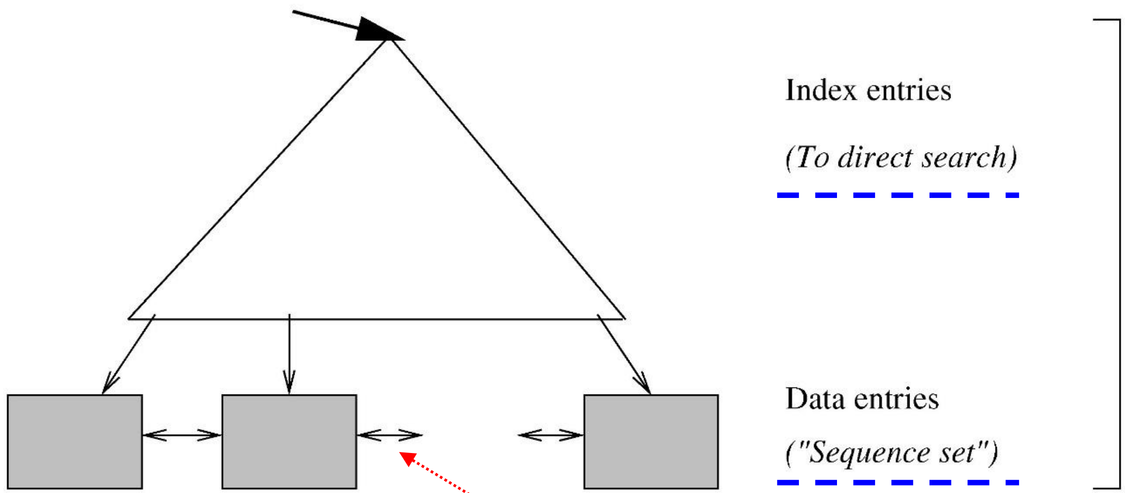
10.3 B^+-Tree Index Files:
A Dynamic Index Structure

- Advantage of B^+-tree index files
  - **automatic** / **local** / **reorganizations**
- Disadvantage of B^+-trees
  1. extra time overhead for insertion/deletion
  2. extra space overhead (2/3 full on average,1/3 empty)
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively.

- Clustered vs. Non-Clustered Index
  - Note that index entries are sorted according to keys
  - If data records are sorted according to the index key, then clustered index. Otherwise, non-clustered index: e.g. ISAM – Clustered!

Structure of B+ Tree

Figure 10.7



Unlike ISAM, leaf pages are doubly linked listed in B+-Tree. **WHY?**

- A B+-tree is a rooted tree
- <u>All paths from root to leaf</u> are of the same length(i.e. height)
  - <u>Balanced</u>

Oracle Formatted Block Dumps

```
SQL> create table tt as select * from emp;

SQL> create index tt_idx on tt(empno);

SQL> select object_name, object_id
        from dba_objects
        where object_type = 'INDEX'
                and owner =  'SCOTT‘
```

```
OBJECT_NAME     OBJECT_ID
-------------   ----------
PK_DEPT         30138

PK_EMP          30140

TT_IDX          30467
```


Very Large Data Bases

Oracle Formatted Block Dumps

SQL> alter session set events 'immediate trace name treedump level 30467'

--- Check the trace file in admin/udump/xxx.trc file.

```
 ----- begin tree dump
leaf: 0x40c65a 4245082 (0: nrow: 14 rrow: 14)
Leaf block dump
===============
header address 2127872092=0x7ed4c05c
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80:opcode=0:iot flags=- is converted=Y
kdxconco 2
kdxcosdc 0
kdxconro 14
kdxcofbo 64=0x40
kdxcofeo 7855=0x1eaf
kdxcoavs 7791
kdxlespl 0
kdxlende 0
kdxlenxt 0=0x0
kdxleprv 0=0x0
kdxledsz 0
kdxlebksz 8036
```

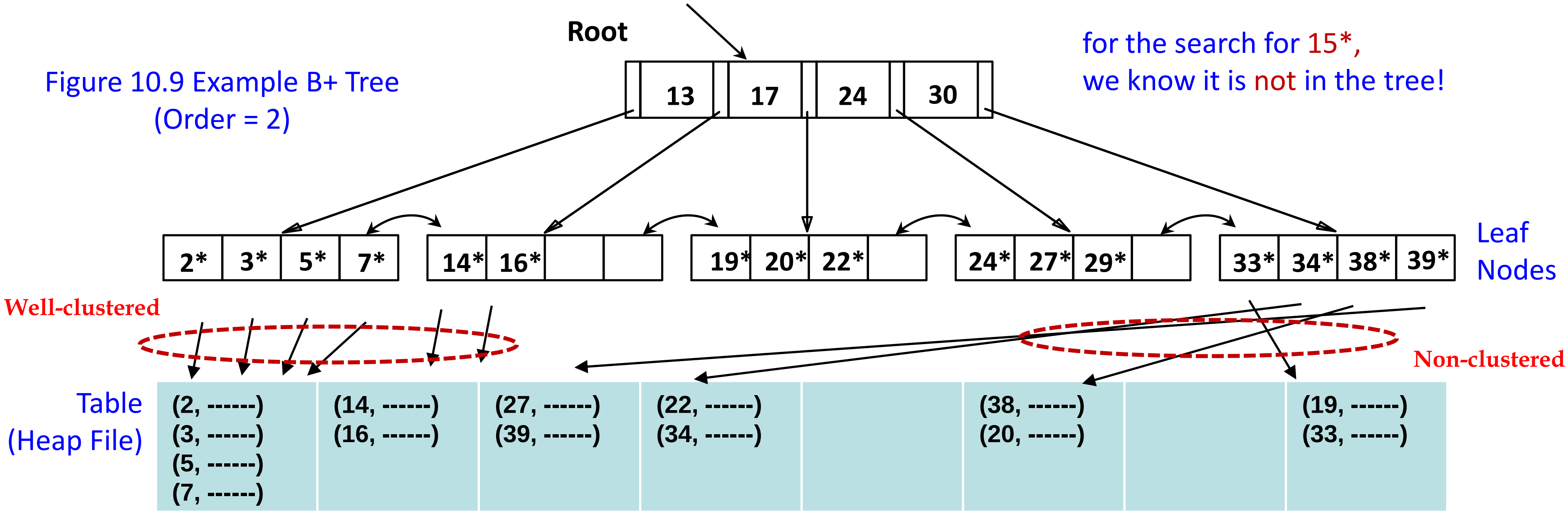
```
row#0[8023] flag: -----, lock: 0
col 0; len 3; (3):  c2 4a 46
col 1; len 6; (6):  00 40 c6 52 00 00
row#1[8010] flag: -----, lock: 0
col 0; len 3; (3):  c2 4b 64
col 1; len 6; (6):  00 40 c6 52 00 01
row#2[7997] flag: -----, lock: 0
col 0; len 3; (3):  c2 4c 16
col 1; len 6; (6):  00 40 c6 52 00 02
row#3[7984] flag: -----, lock: 0
col 0; len 3; (3):  c2 4c 43
col 1; len 6; (6):  00 40 c6 52 00 03
row#4[7971] flag: -----, lock: 0
. . . .
row#12[7868] flag: -----, lock: 0
col 0; len 3; (3):  c2 50 03
col 1; len 6; (6):  00 40 c6 52 00 0c
row#13[7855] flag: -----, lock: 0
col 0; len 3; (3):  c2 50 23
col 1; len 6; (6):  00 40 c6 52 00 0d
----- end of leaf block dump -----
----- end tree dump
```

10.4 Search

- Search begins at root, and key comparisons direct it to a leaf
  - Equality search: 5*, 15*; what about null?
  - Range search: data entries b/w 2*and 7*, b/w 33* and 39*, <= 14*, and >= 27*

Figure 10.9 Example B+ Tree (Order = 2)

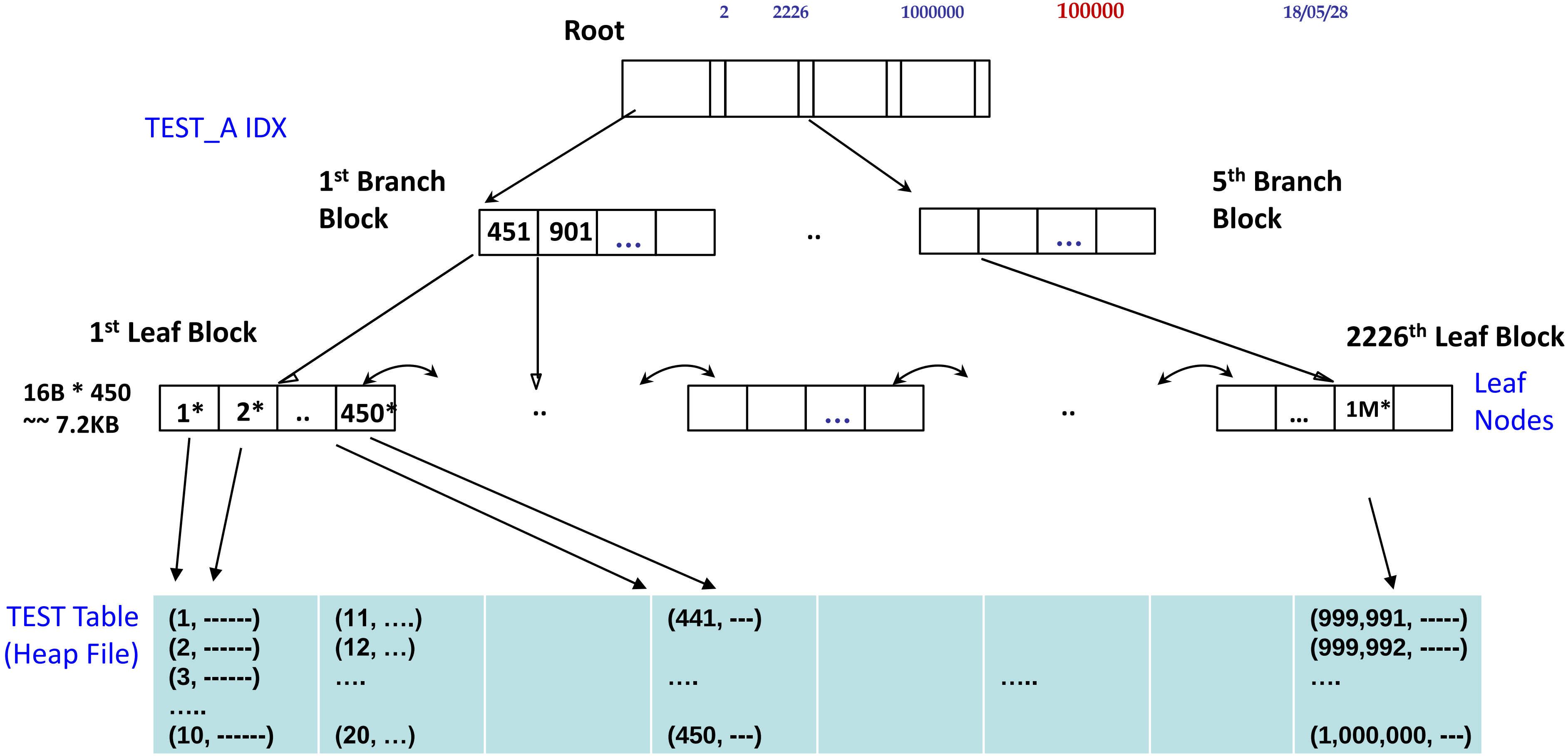
for the search for 15*, we know it is not in the tree!



TEST_A IDX on TEST(A)

HEIGHT	BLOCKS	LF_ROWS	LF_BLKS	LF_ROWS_LEN	BR_ROWS	BR_BLKS	BR_ROWS_LEN
3	2304	1000000	2226	15979802	2225	5	26658

BLEVEL	LEAF_BLOCKS	DISTINCT_KEYS	CLUSTERING_FACTOR	LAST_ANA
2	2226	1000000	100000	18/05/28



Algorithm for B+ Tree Search(Figure 10.8)

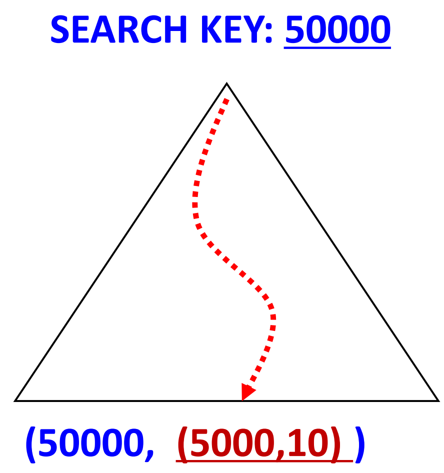
func *find* (search key value K) **returns** nodepointter
// Given a search key value, finds its leaf node
return tree_search(root, K); // searches from root
endfunc

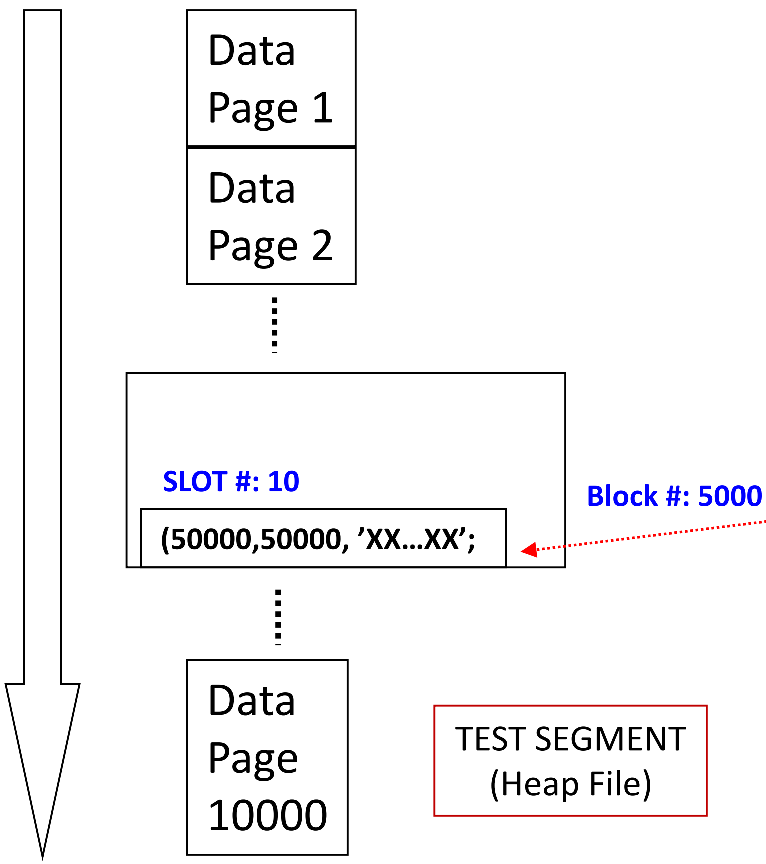
func *tree_search* (nodepointer, search key value K) **returns** nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if $K < K_1$ then return tree_search(P_0, K);
    else,
        if $K \geq K_m$ then return tree_search(P_m, K); // m = # entries
        else,
            find i such that $K_i \leq K < K_{i+1}$;
            return tree_search(P_i, K)
endfunc

Insertion and Deletion

Assume the TEST table in SCRIPT:
TEST (A, B, C)

Index on TEST(B)





On every insertions, deletions or updates to a base table, every relevant indexes should also be changed!

10.5 Inserting a Data Entry into a B+ Tree

- Find correct leaf *L* and 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, then copy up middle key.
    - ✓ insert index entry pointing to *L2* into parent of *L*.
- Split can happen recursively in index node
  - to split index node, redistribute entries evenly, but push up middle key. (cf. leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

Inserting 8* into Example B+ Tree

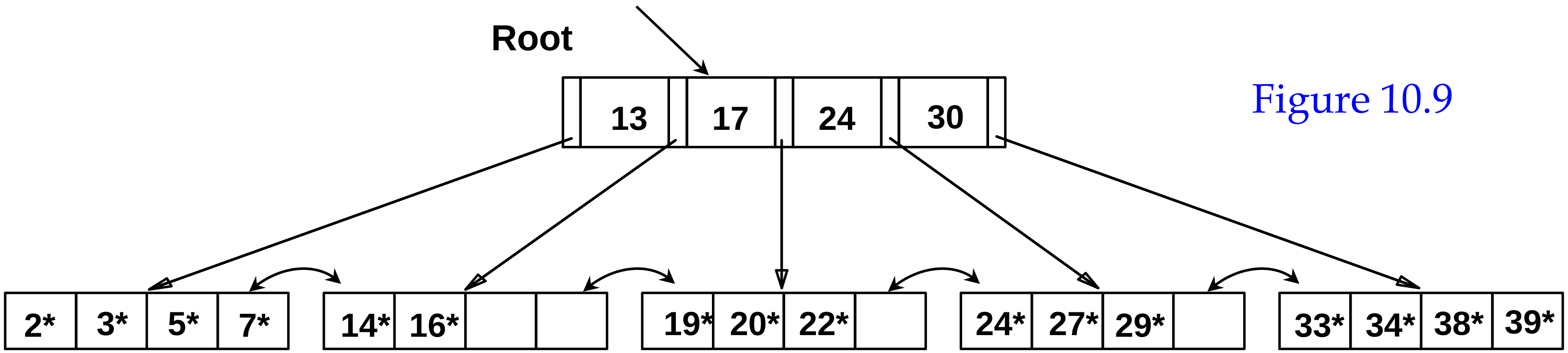


Figure 10.9

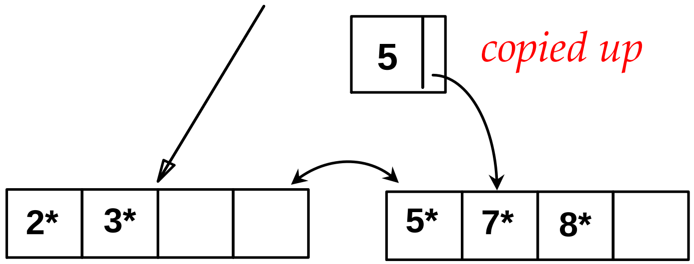


Figure 10.11

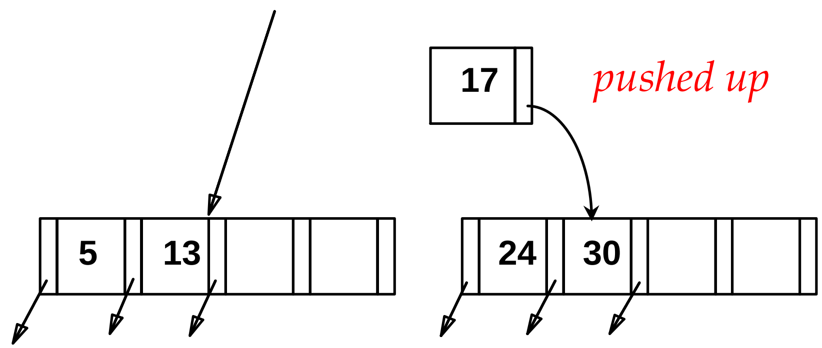


Figure 10.12

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

Very Large Data Bases

Example B+ Tree After Inserting 8*

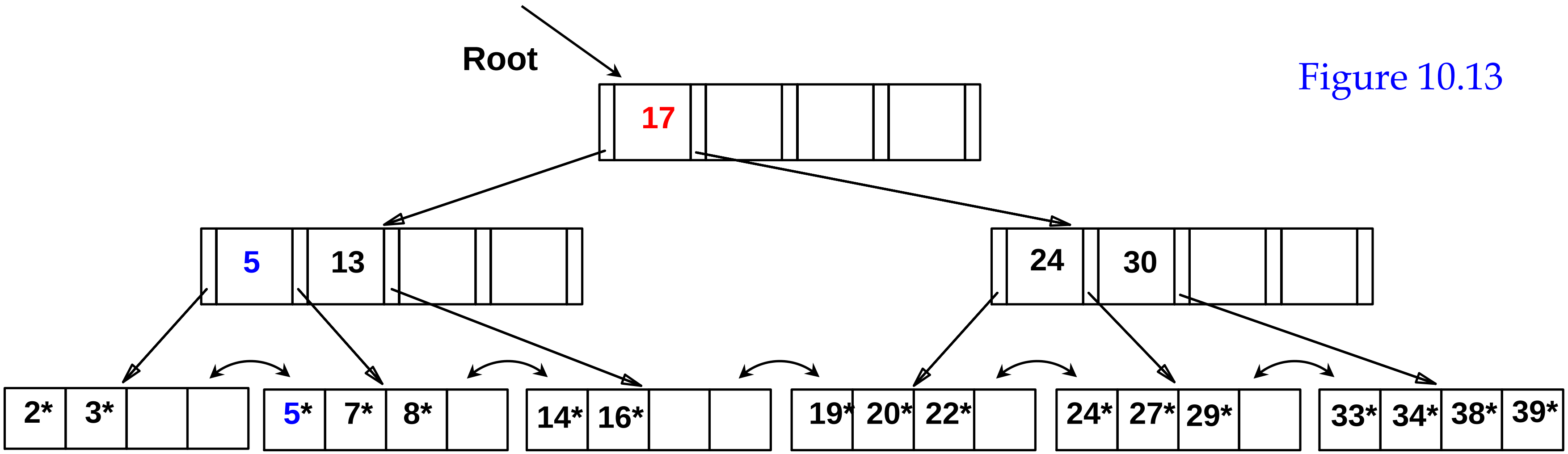


Figure 10.13

- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.
- Some key values in leaf pages also appear in a non-leaf page
- *What if flash*? Overflow instead of split? Page update: 2 vs. 4

B+ Tree: Most Widely Used Index

- Insertion/deletion at ($\log_F N$) cost
  - **F** = fanout; **N** = # leaf pages
  - Keep tree height-balanced.

- Minimum 50% occupancy (except for root node).
  - Each node contains **d <= *m* <= 2d** entries: **d =** *order* of the tree

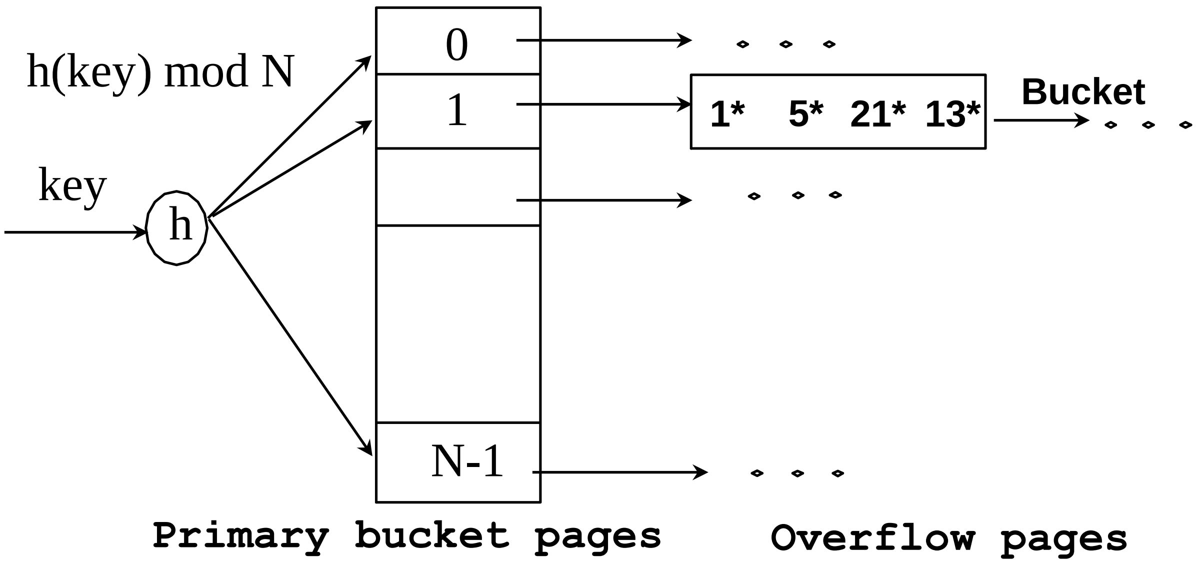
- Supports both equality and range searches efficiently.

10.8 B+ Trees in Practice

- Typically
  - Order: 100
  - Fill-factor: 67%.
  - Fanout: 133
- Typical capacities:
  - height 4: 133^4 = 312,900,700 records
  - height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Ch. 11 Hashing

- Static, Extendible, Linear Hashing
  - Academic excellence!!



- *Hash-based* indexes are best for *equality selections*. ***BUT! cannot*** support range searches.

More Index Techniques

- Bitmap indexes for data warehouse (Ch 25.6.1)

M	*F*
1	0
1	0
0	1
1	0

custid	*name*	*gender*	*rating*
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
112	Woo	M	4

1	*2*	*3*	*4*	*5*
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

- R-tree for spatial and multi-dimensional databases (Ch 28.6)