高级数据结构

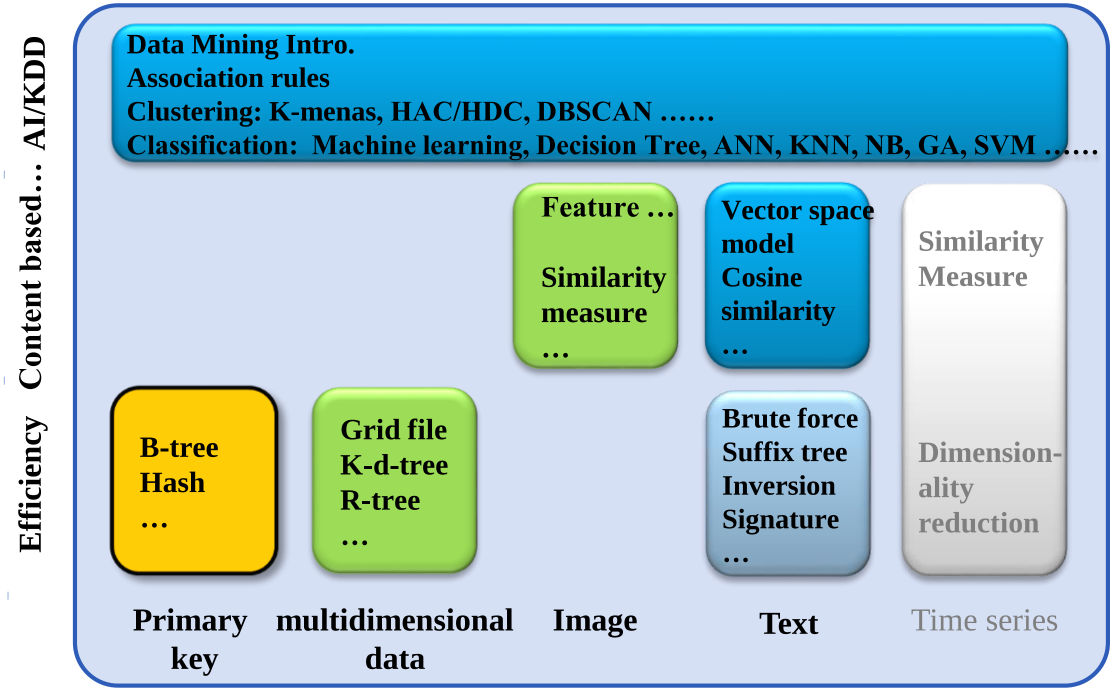
Advanced Data Structure

2. Primary key access methods

Agenda

- Primary key access methods -

  How to access a large collection of values?
  - Application example:
    - Concept of RDBMS
    - Computational model
  - Hash
  - B-tree family



Agenda

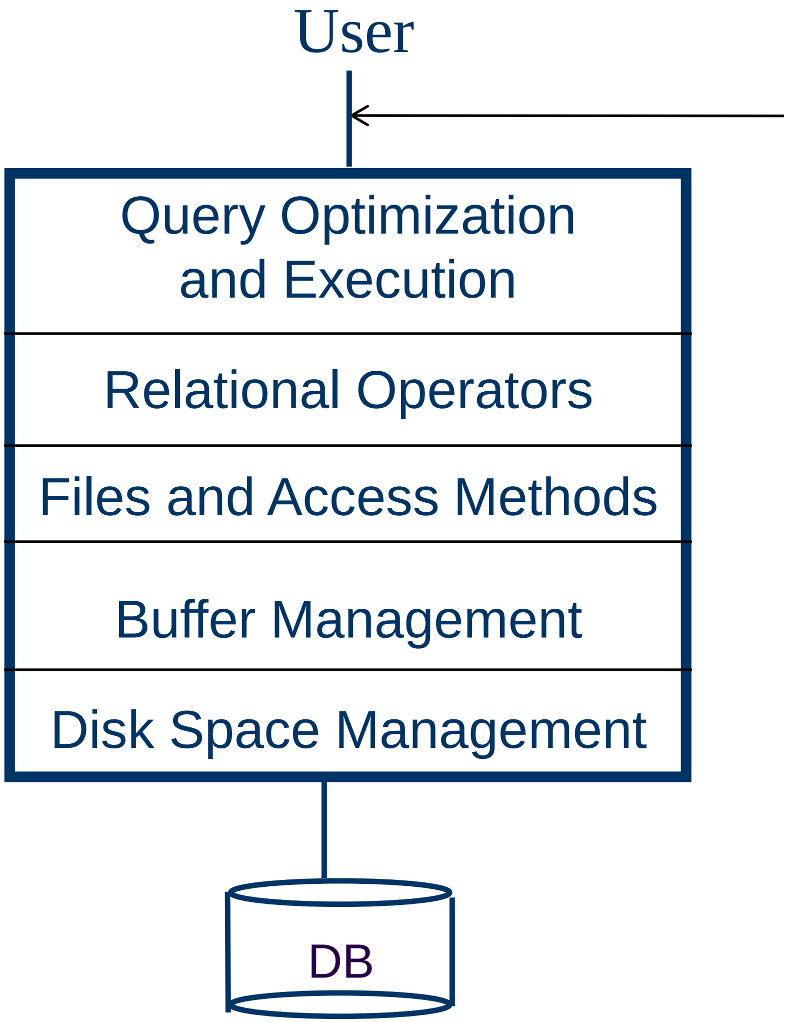
- Primary key access methods
  - Concept of RDBMS
    Computational model
  - Hash
  - B-tree family

Concept of RDBMS (Relational DataBase Management System)

- Representing data using *Relational Model*
  - Data are organized in ***tables*** (***relations***)
  - Rows of table correspond to ***records***
  - Columns correspond to ***attributes***
- Access the data using ***S****tructured* ***Q****uery* ***L****anguage* (SQL)
  - Handle queries on primary keys

ID	Name	Age	Salary
123	S. John	33	30000
456	E. Tom	22	3000
…	…	…	…

Query and Typical DBMS Architecture



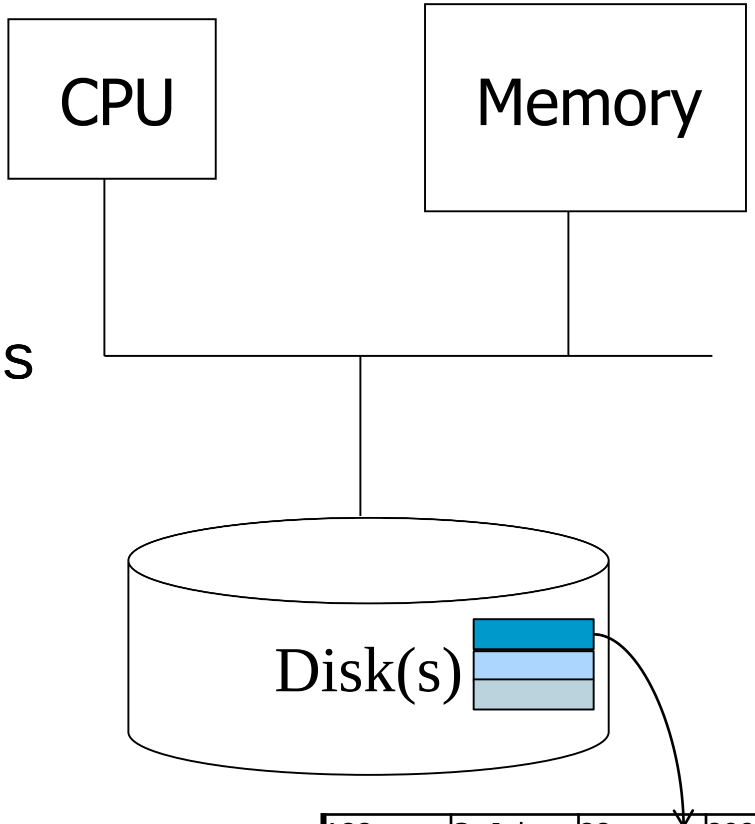
- **Query**
  1. Exact match
     - x=10
  2. Range query
     - $10<x<100$
  3. Nearest neighbor query
     - $\text{argmin}_x(d(x,10))$

       $d(x,y)=|x-y|$

  ……

Model of Computation

- **Data stored on disk(s)**
- **Minimum transfer unit**
  - A ***block*** (***page***) = ***B*** records
- I/O complexity
  - Measured in number of blocks accessed



123	S. John	33	30000
456	E. Tom	22	3000
…	…	…	…

Model of Computation - Index

- Use <u>index</u> to speed up *value* → *physical-storage* processing

- <u>Loading Factor</u> =

  number of elements in the index / maximal number of elements in the index

- Classical index methods for RDBMS
  - **Hashing Methods:** Linear Hashing, extendible hashing
  - **B-tree family:** B tree, B^+-tree, and variations

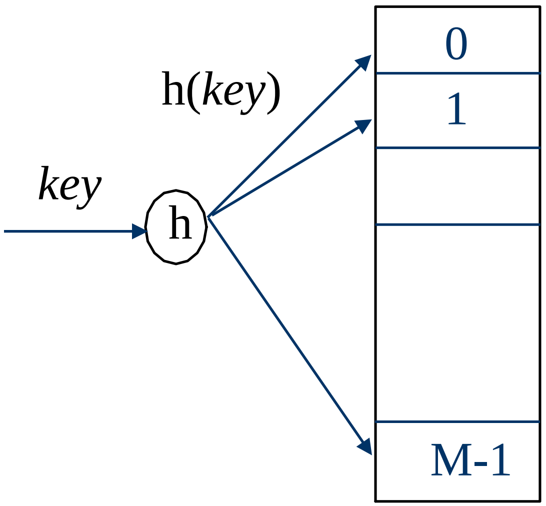
Agenda

- Primary key access methods
  - Concept of DBMS Computational Model
  - Hash
  - B-tree family

Hash - Key Idea

- Use a key-to-address function to direct a record to a disk block
- **h**(*k*) = *bucket* to which data entry with key *k* belongs

  e.g. h(*key*) = *key* mod *M*

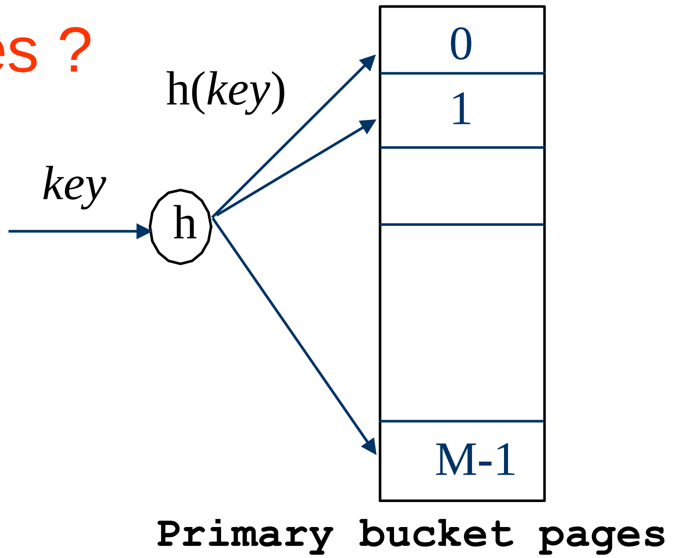


Primary bucket pages

Hash - Challenges

- Overflow handling:
  - Open addressing: re-hashing to another bucket
  - Separate chaining: use a separate *overflow area*

  - Problem in **dynamic** databases ?



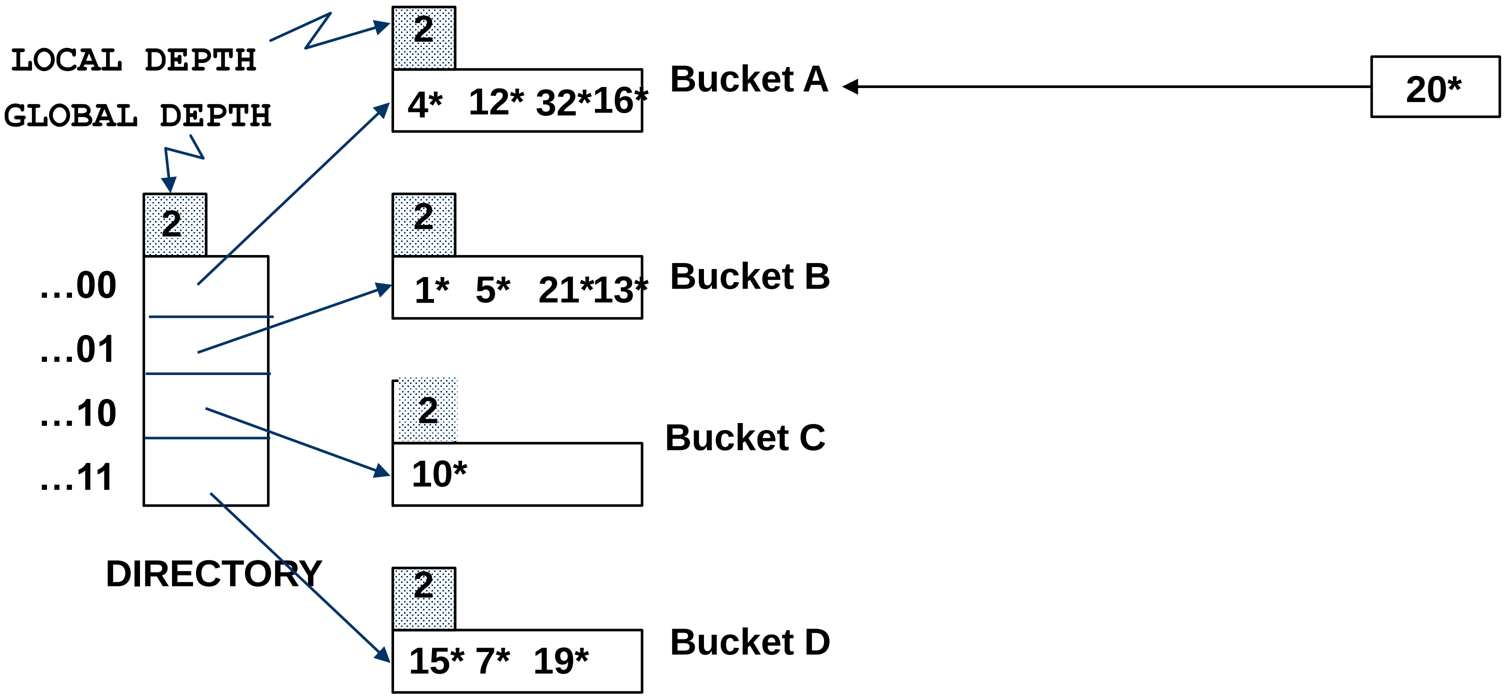
Dynamic Hashing Schemes

- Extendible hashing:
  - Uses a directory that grows or shrinks depending on the data distribution.
  - No overflow buckets
- Linear hashing:
  - No directory
  - Splits buckets in linear order
  - Uses overflow buckets

Extendible Hashing

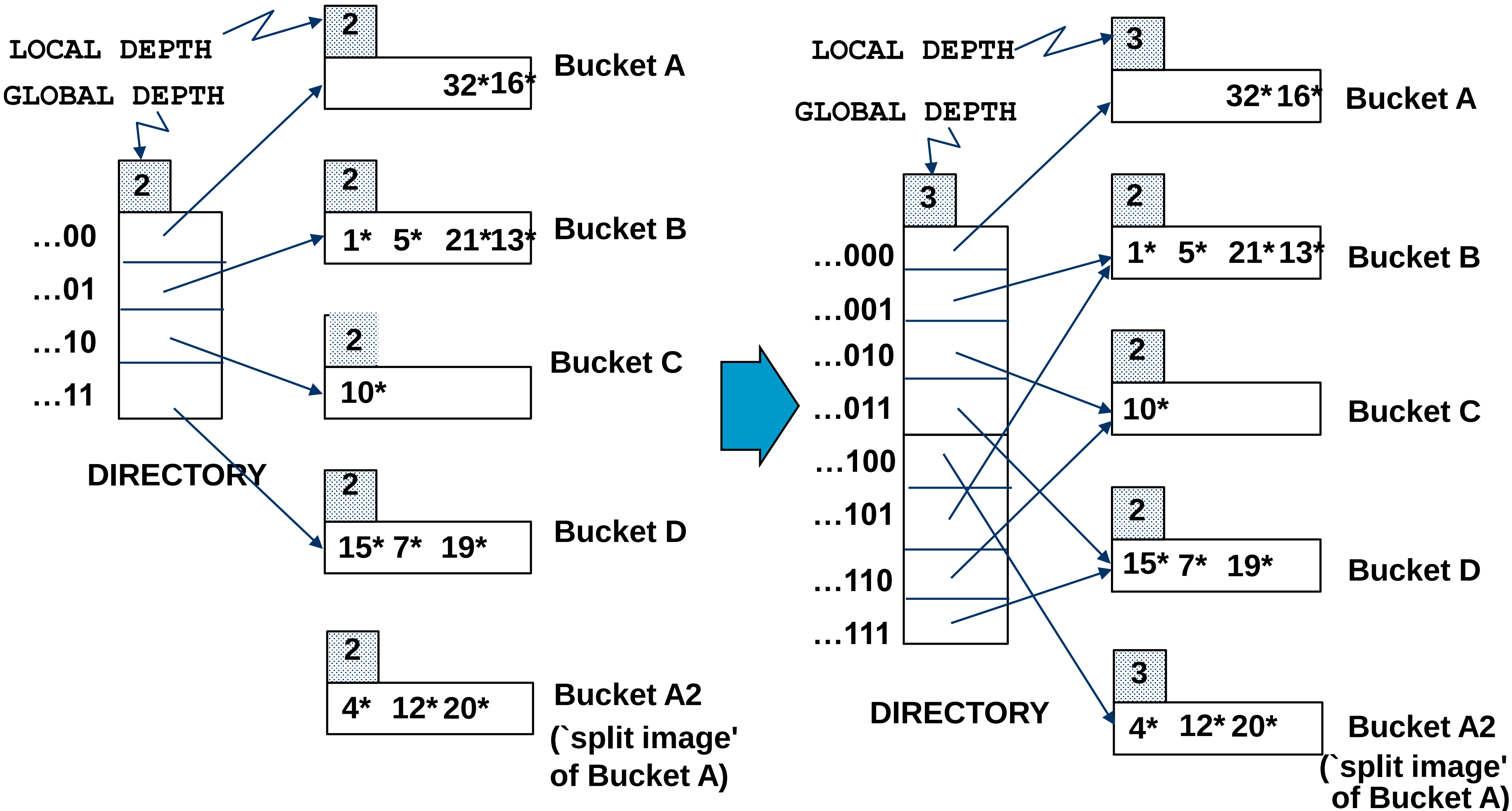
- When bucket becomes full,
  re-organize file by *doubling* the number of buckets?
  - Reading and writing all pages is expensive!
- *Idea*:
  - Use *directory of pointers to buckets*
  - Double # of buckets by *doubling the directory*
  - *S*plitting just the bucket that overflowed!

Insert 20



LOCAL DEPTH
GLOBAL DEPTH
2
2
4* 12* 32*16*
Bucket A
20*
2
…00
…01
…10
…11
1* 5* 21*13*
Bucket B
2
10*
Bucket C
DIRECTORY
2
15* 7* 19*
Bucket D

Insert 20


LOCAL DEPTH
GLOBAL DEPTH
2
32*16*
Bucket A
2
…00
…01
…10
…11
2
1* 5* 21*13*
Bucket B
2
10*
Bucket C
DIRECTORY
2
15* 7* 19*
Bucket D
2
4* 12* 20*
Bucket A2
(`split image' of Bucket A)
LOCAL DEPTH
3
32* 16*
Bucket A
GLOBAL DEPTH
3
…000
…001
…010
…011
…100
…101
…110
…111
2
1* 5* 21* 13*
Bucket B
2
10*
Bucket C
2
15* 7* 19*
Bucket D
DIRECTORY
3
4* 12* 20*
Bucket A2
(`split image' of Bucket A)

Remarks

- Directory is much smaller than the data file, so doubling it is much cheaper.
- Only one disk block (of data entries) is split.

Agenda

- Primary key access methods
  - Concept of DBMS Computational Model
  - Hash
    - Extendible hash
    - Linear hash
  - B-tree family

Linear Hashing

- Motivation:
  - Ext. Hashing requires storage space for directory.
  - Directory grows by doubling.

    Can we do better? (smoother growth)

- **L**inear **H**ashing (LH): another dynamic hashing scheme.
  - Split buckets from "left" to "right", regardless of which one overflowed (simple, but it works!)

Linear Hashing: Example

Initially: h(x) = x *mod N* (*N*=**4**)

Assume **3** records/bucket

Insert 17 17 mod 4 → 1

Split pointer

Bucket id	0	1	2	3
	4 8	5 9 13	6	7 11

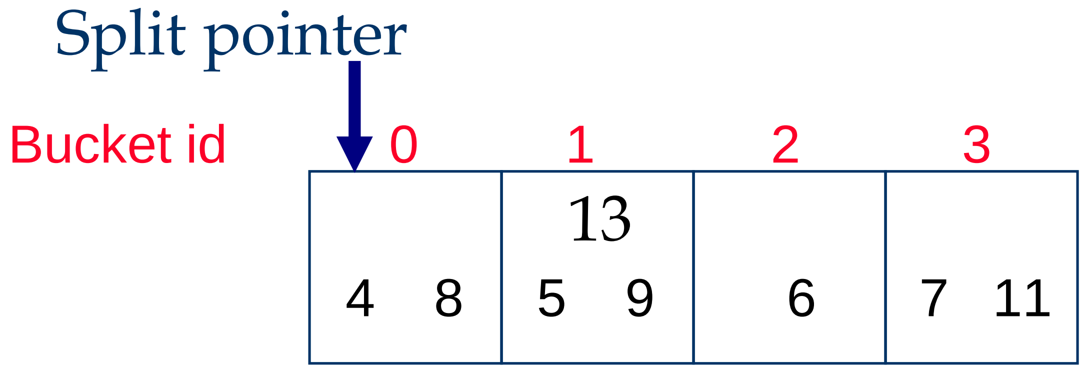
Linear Hashing: Example

Initially: h(x) = x *mod N* (N=4)

Assume 3 records/bucket

Insert 17 = 17 mod 4 → 1

Overflow for Bucket 1

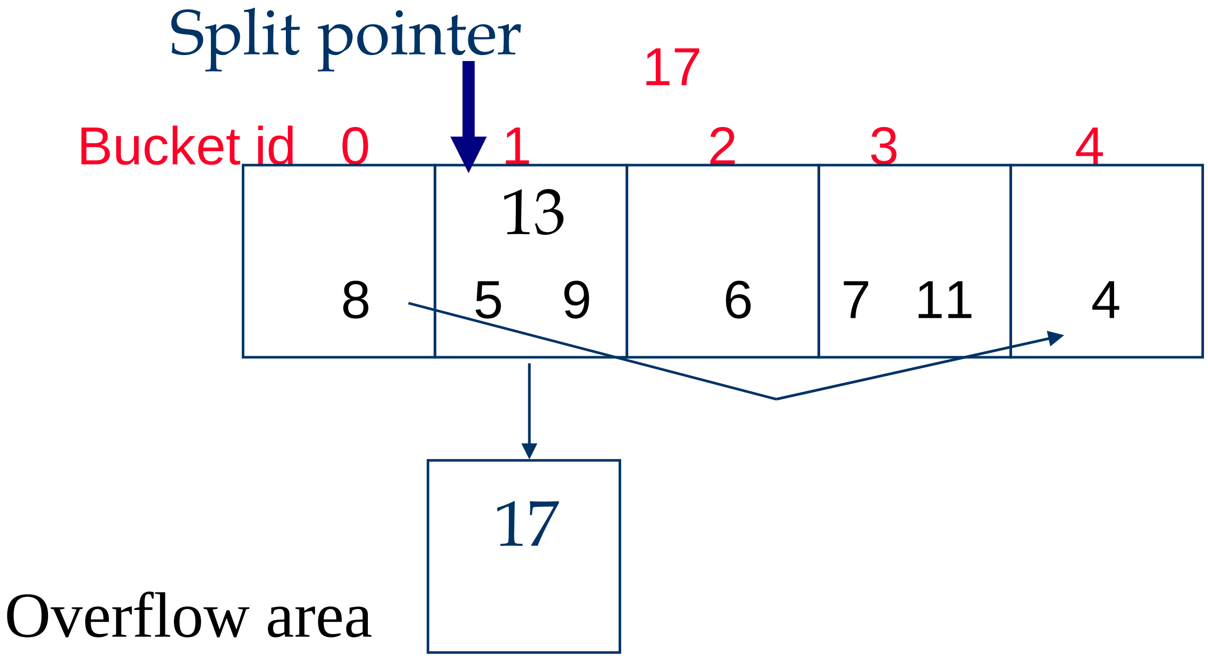


Split bucket 0, anyway!!

Linear Hashing: Example

To split bucket 0, use another function h1(x):

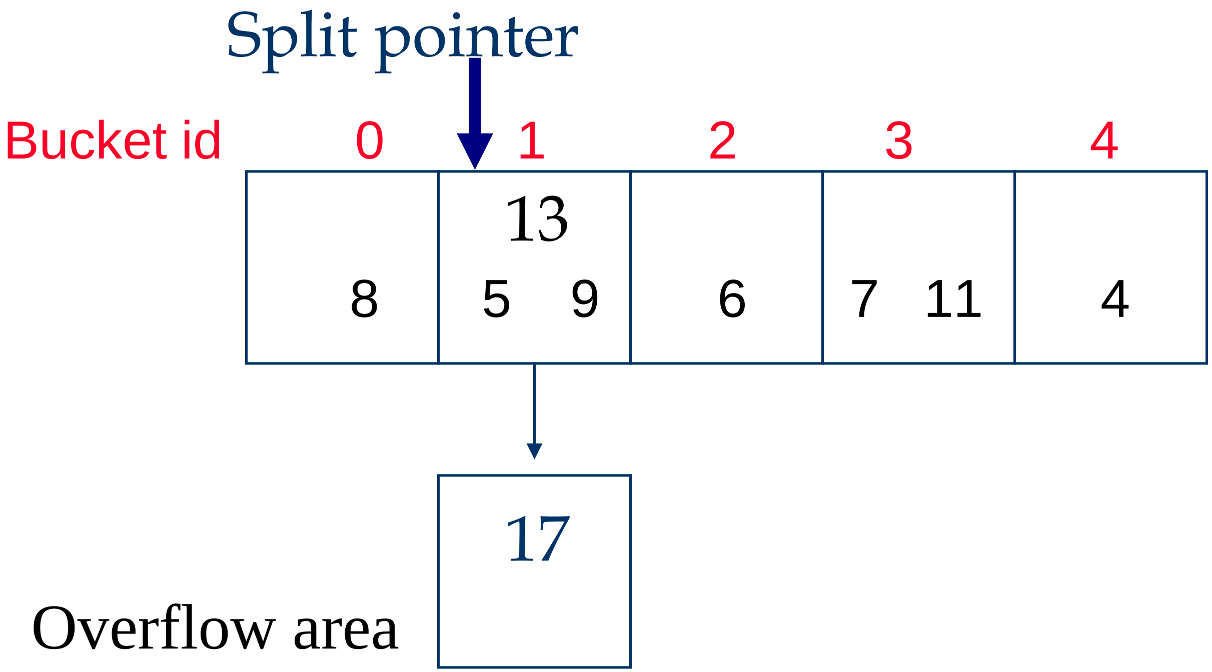
$h0(x) = x \bmod N$, $h1(x) = x \bmod (2^*N)$



Linear Hashing: Example

h0(x) = x mod N , h1(x) = x mod (2*N)

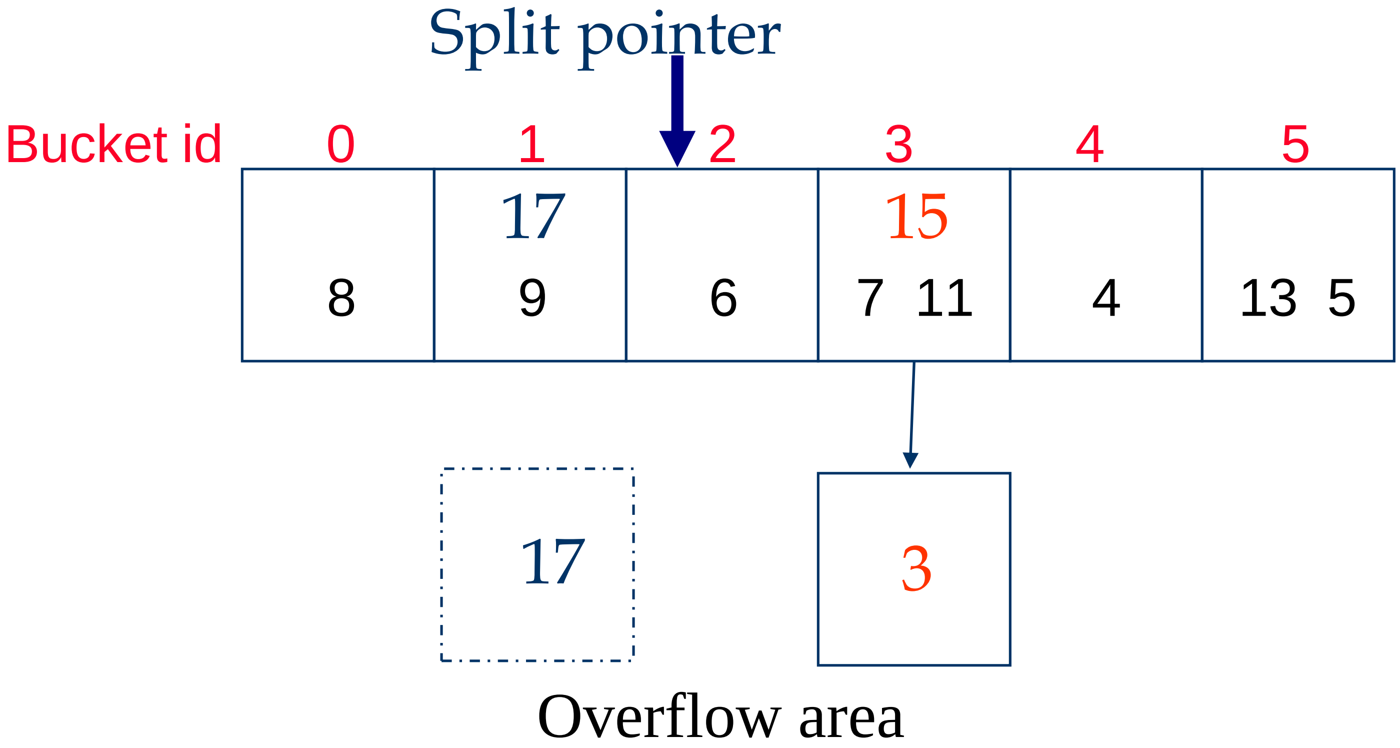
Insert 15 and 3: 15 mod 4→3, 3 mod 4 → 3



Linear Hashing: Example

h0(x) = x mod N , h1(x) = x mod (2*N)

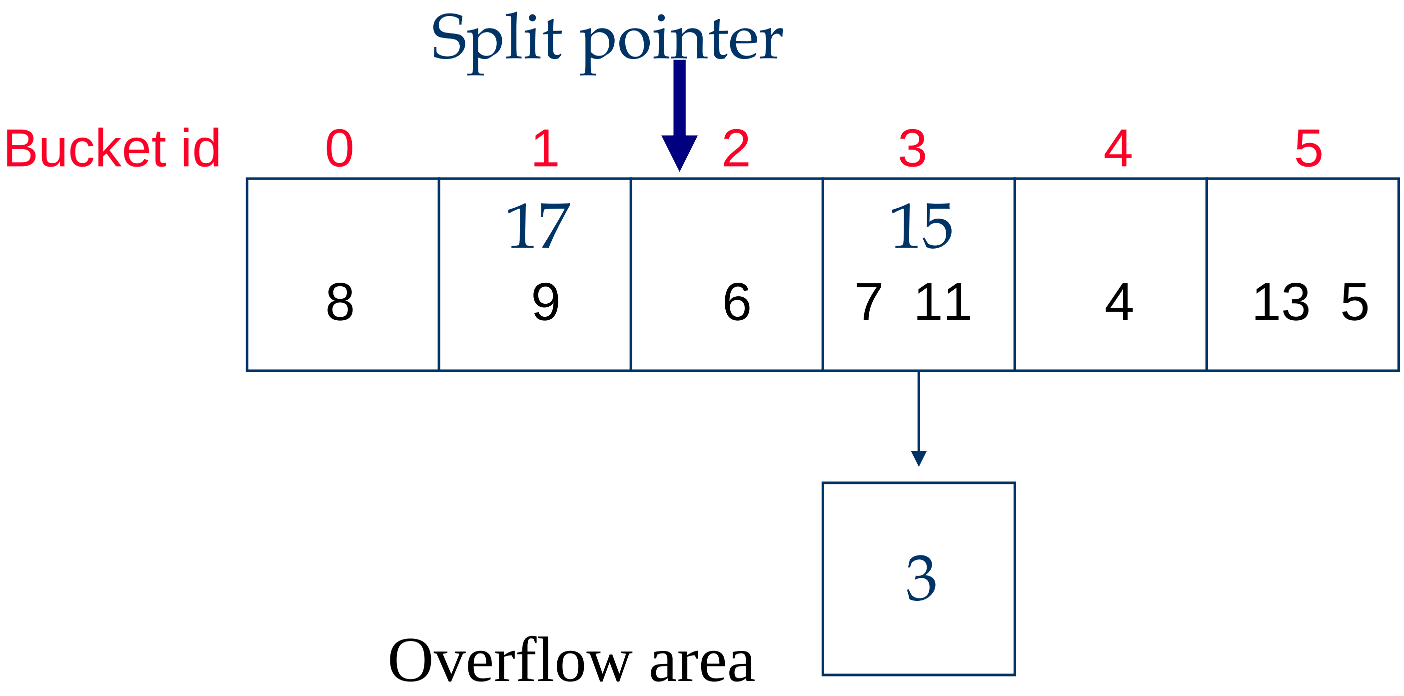
Insert 15 and 3



Linear Hashing: Search

h0(x) = x mod N (*for the un-split buckets*)

h1(x) = x mod (2*N) (*for the split ones*)



Problems of hash

- For records that can be sorted over an attribute
  - E.g., salary, age, etc.

  How to answer range queries using Hash?

  How to answer nearest neighbor queries using Hash?

Agenda

- Primary key access methods
  - Concept of DBMS Computational Model
  - Hash
  - B tree family
    - B tree
    - B* tree
    - B+ tree

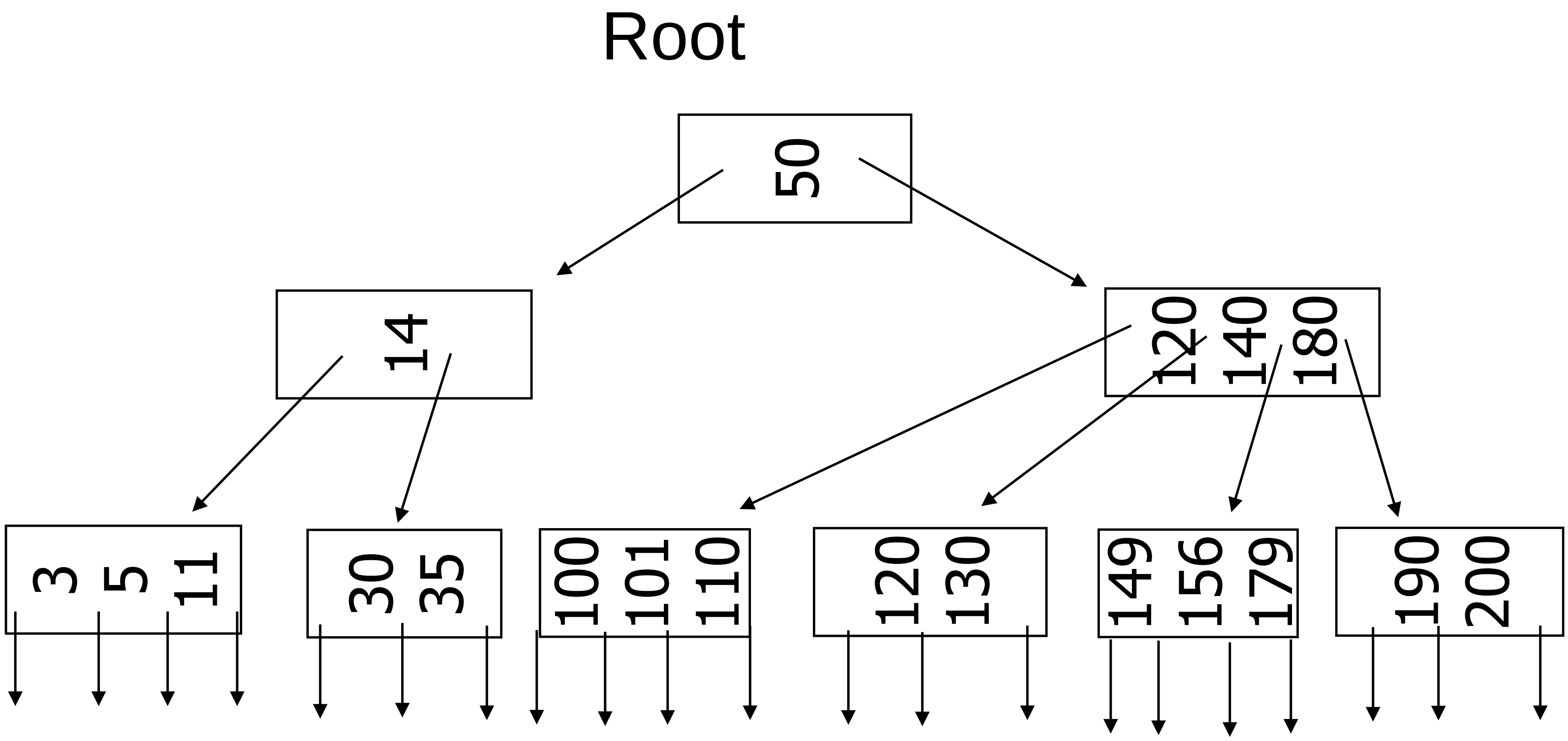
B tree

- B tree is a multi-way search tree with the following properties:
  - The root has at least two children unless it's a leaf
  - Each non-root internal node holds k-1 keys and ***k* pointers** to sub-trees where $\lceil m/2 \rceil \leq k \leq m$
  - Each leaf node holds k-1 keys where $\lceil m/2 \rceil \leq k \leq m$
  - All leaves are on the same level

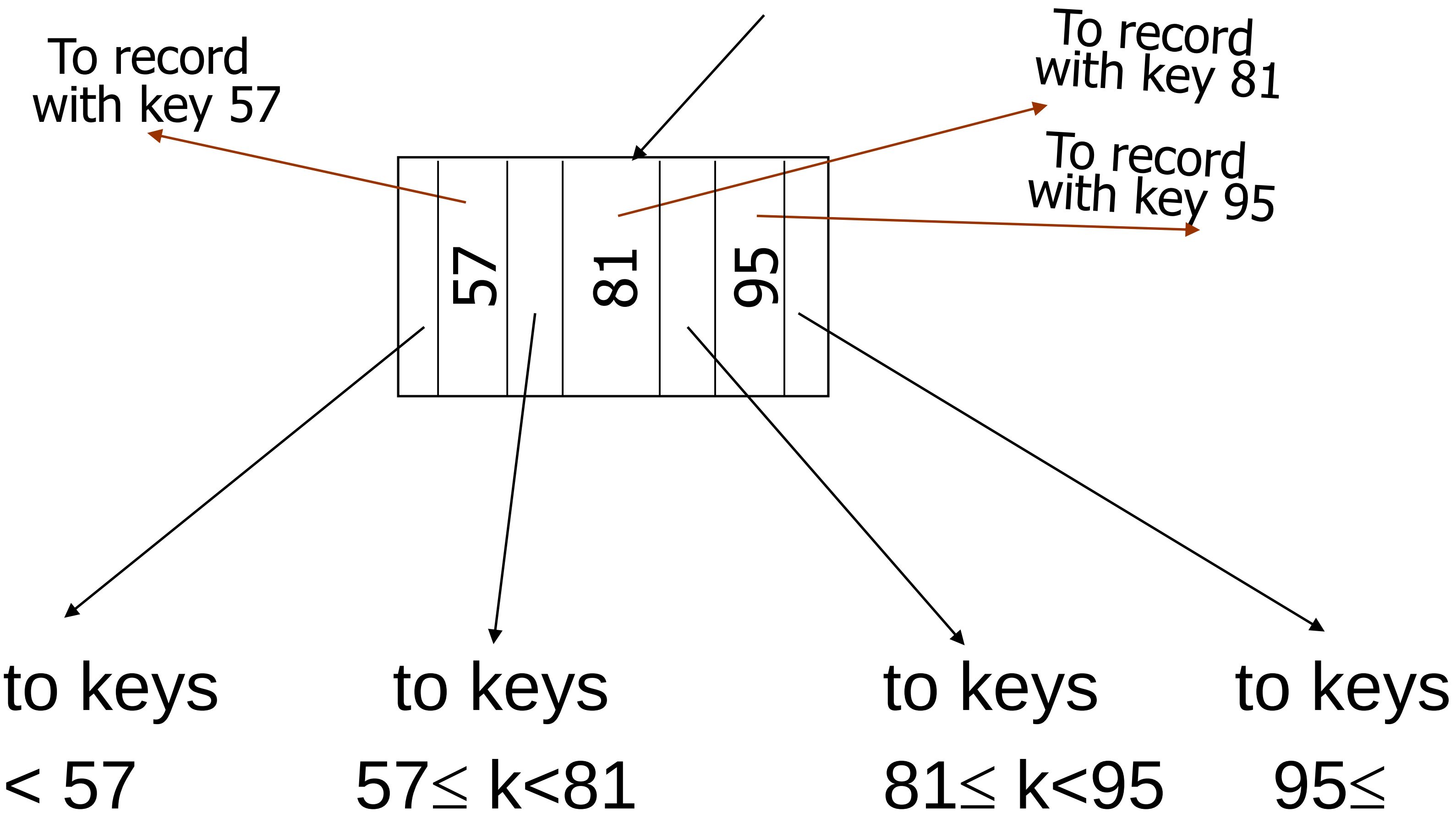
- Each node is fit to one disk block

B tree - Example

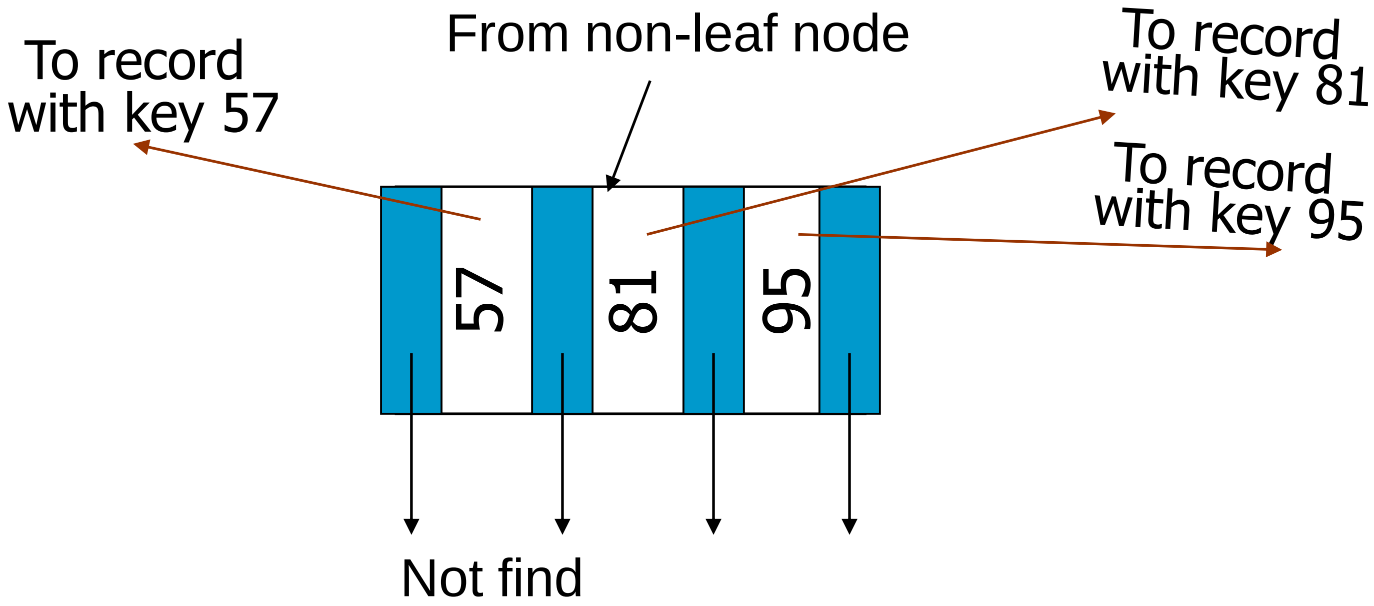
***m*=4**



B tree - internal node



B tree - Leaf node



B tree - Insertion

- 1) Find correct leaf *L.*
- 2) Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, *split* *L (into L and a new node)*
    - split, **move up** middle key.

- This can happen recursively
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

B tree - Deletion

- Start at root, find node *L* where entry belongs.
- Remove the entry.
  - If *L* is at least half-full, *done!*
  - Otherwise,
    - Try to **re-distribute**, borrowing from **sibling** *(adjacent node with same parent as L)*.
    - If re-distribution fails, **merge** *L* and sibling.
- Merge could propagate to root, decreasing height.

B tree - Characteristics

- Loading factor:
  - Minimum 50% occupancy (except for root)
  - Average 69% (Random insertion)
- Always balanced
- Space: linear
- Update and query: logarithmic (in number of I/Os).

Agenda

- Review of complexity analysis
- Primary key access methods
  - Concept of DBMS
  - Computational Model
  - Hash
  - B tree family
    - B tree
    - B* tree
    - B+ tree

B* tree

- Motivation: The fewer nodes that are created, the better.
- Main idea: all nodes except the root are required to be at least 2/3 full
- Split: when overflow:
  - Redistribute the keys by searching its sibling nodes
  - 2 nodes→ 3 nodes
- Generalization:
  - Bn tree: nodes are required to be (n+1)/(n+2) full

B* tree - Characteristics

- Decrease…
  - Fewer splits and guarantee 2/3 space utilization (1/2 for B tree)
- Increase …
  - Programming complexity
  - Insertion time

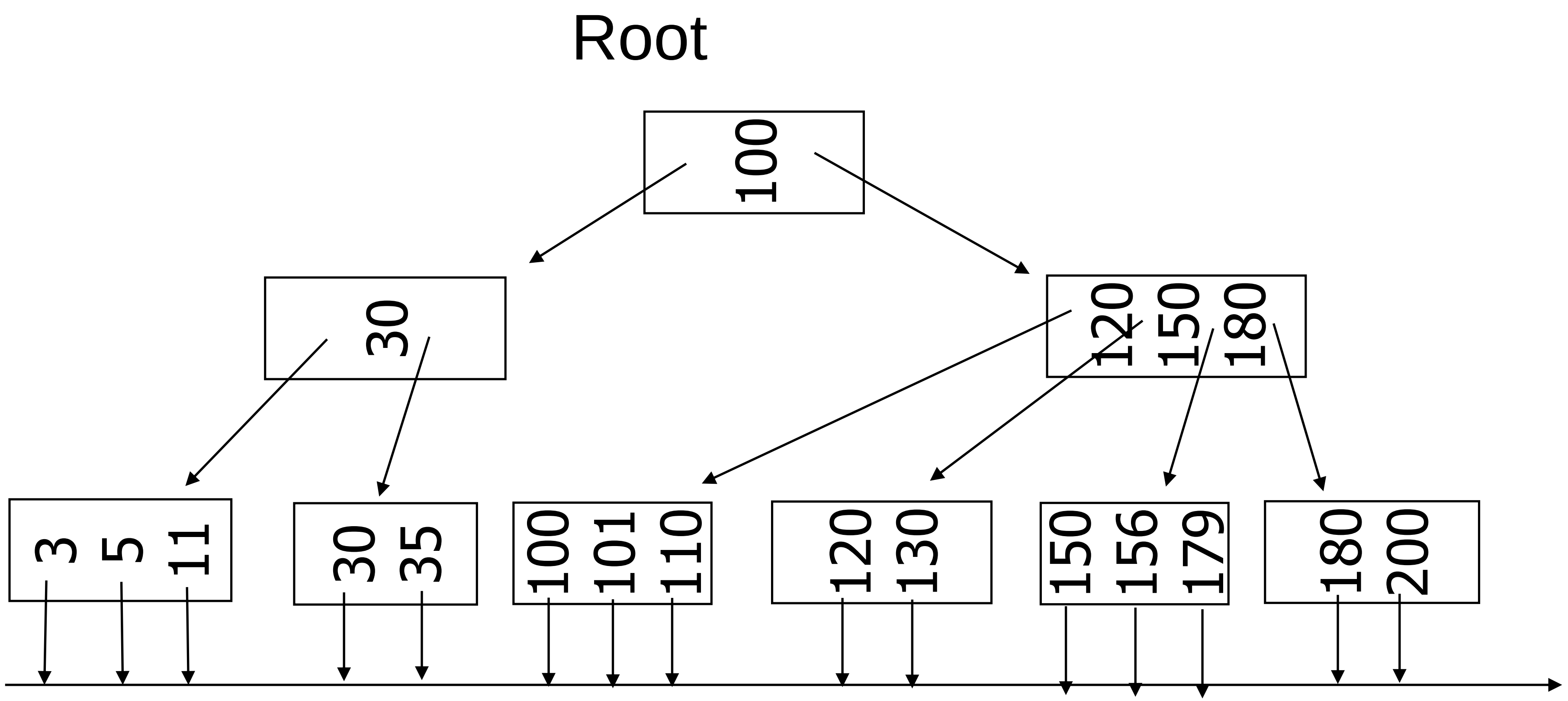
Agenda

- Review of complexity analysis
- Primary key access methods
  - Concept of DBMS
  - Computational Model
  - Hash
  - B tree family
    - B tree
    - B* tree
    - B+ tree

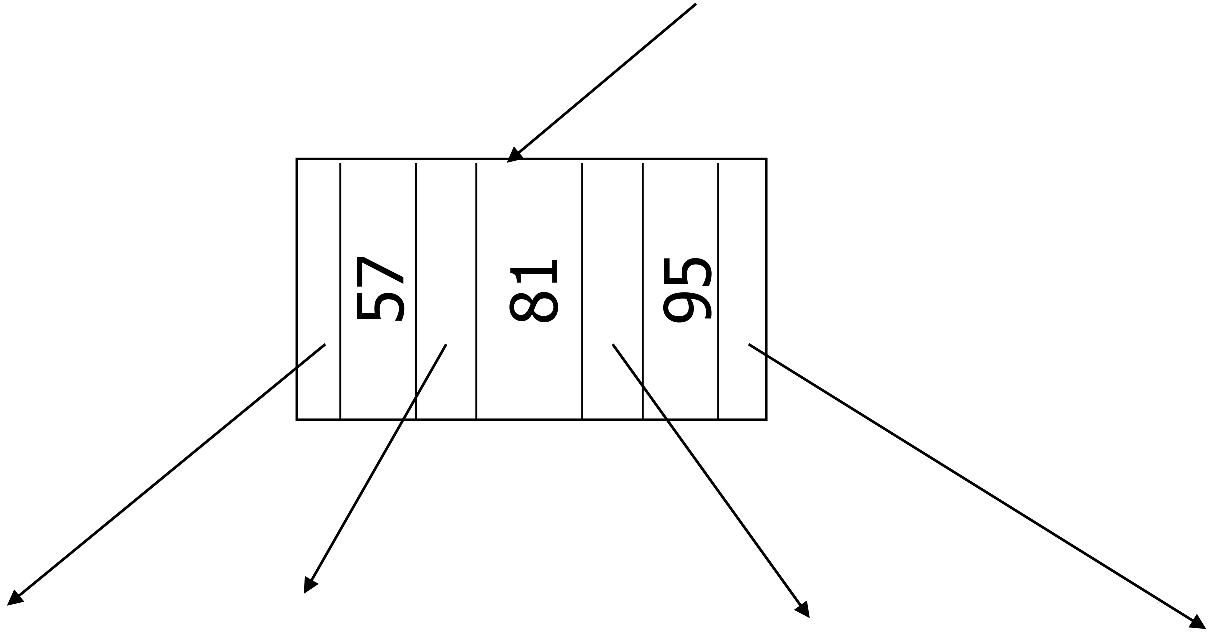
B+ tree

- Motivation: need for range queries.
- Two types of nodes:
  index (internal) nodes and **data** (**leaf**) nodes
  – Data are stored in leaf nodes
- All leaves are chained up

B+ tree - Example



B+ tree - Index node



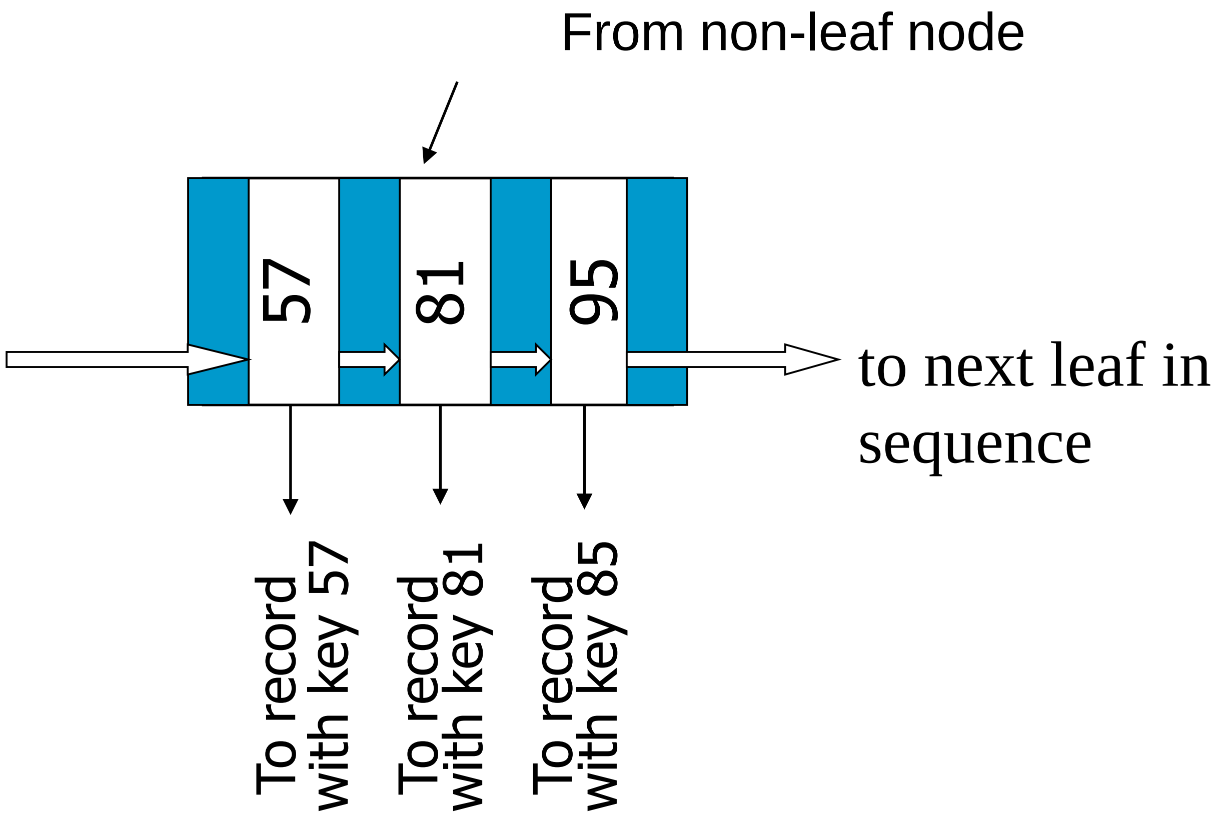
to keys < 57

to keys $57 \leq k < 81$

to keys $81 \leq k < 95$

to keys $95 \leq$

B+ tree - Data node



B+ tree - Characteristics

- Optimal method for 1-d range queries:
- Space: linear
- Update and query: logarithmic (in number of I/Os)

Conclusions: Primary key access methods

- Application: RDBMS
  - Problem:
    - Minimize the number of transfer units (disk blocks)
    - Handle dynamic database
  - Key idea: Indexing
  - Loading factor

Conclusions: B-tree family vs. Hashing

- *Hash-based* indices are best for exact match queries. Faster than *B+-tree*!
  - *Hash-based* indices typically require 1-2 I/Os per query
  - *B*+-tree requires 4-5 I/Os (logarithmic)
- *B family tree* support answering
  - *range queries*
  - *nearest neighbor queries*
  - *ordered sequential scanning*...

Hash-based indices don't support.

Homework

- Read chapters 2, 3
- Homework
  - In linear hash (p23 in ppt), insert some keys
  - In B tree (p28 in ppt), insert some keys and remove some keys

Thanks

Feedback welcome