Big Data Processing Technologies

Chentao Wu
Associate Professor
Dept. of Computer Science and Engineering
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Schedule

- lec1: Introduction on big data and cloud computing
- lec2: Introduction on data storage
- lec3: Data reliability (Replication/Archive/EC)
- lec4: Data consistency problem
- lec5: Block storage and file storage
- lec6: Object-based storage
- lec7: Distributed file system
- lec8: Metadata management

Collaborators

Google 谷歌

阿里云 aliyun.com

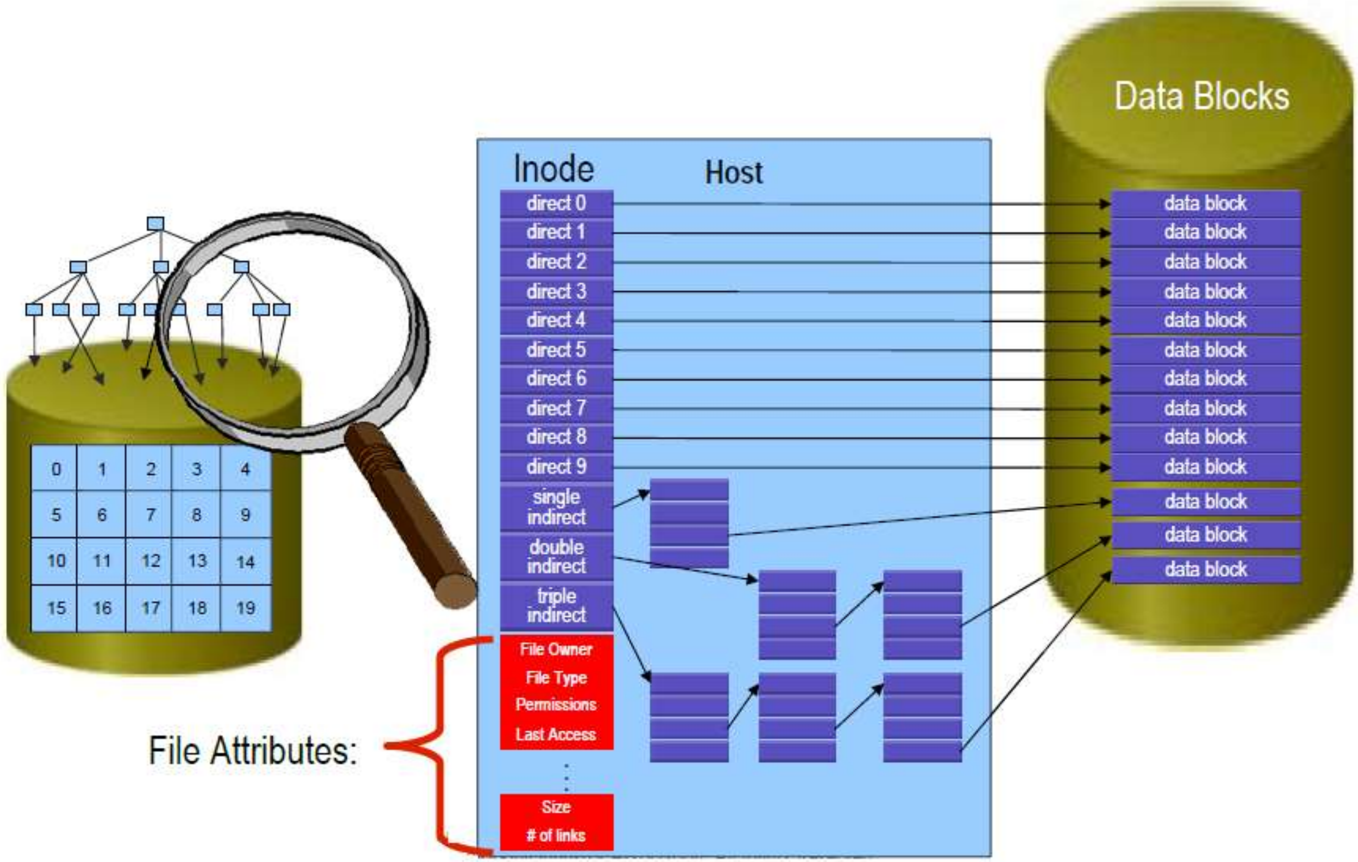
DELL EMC

Contents

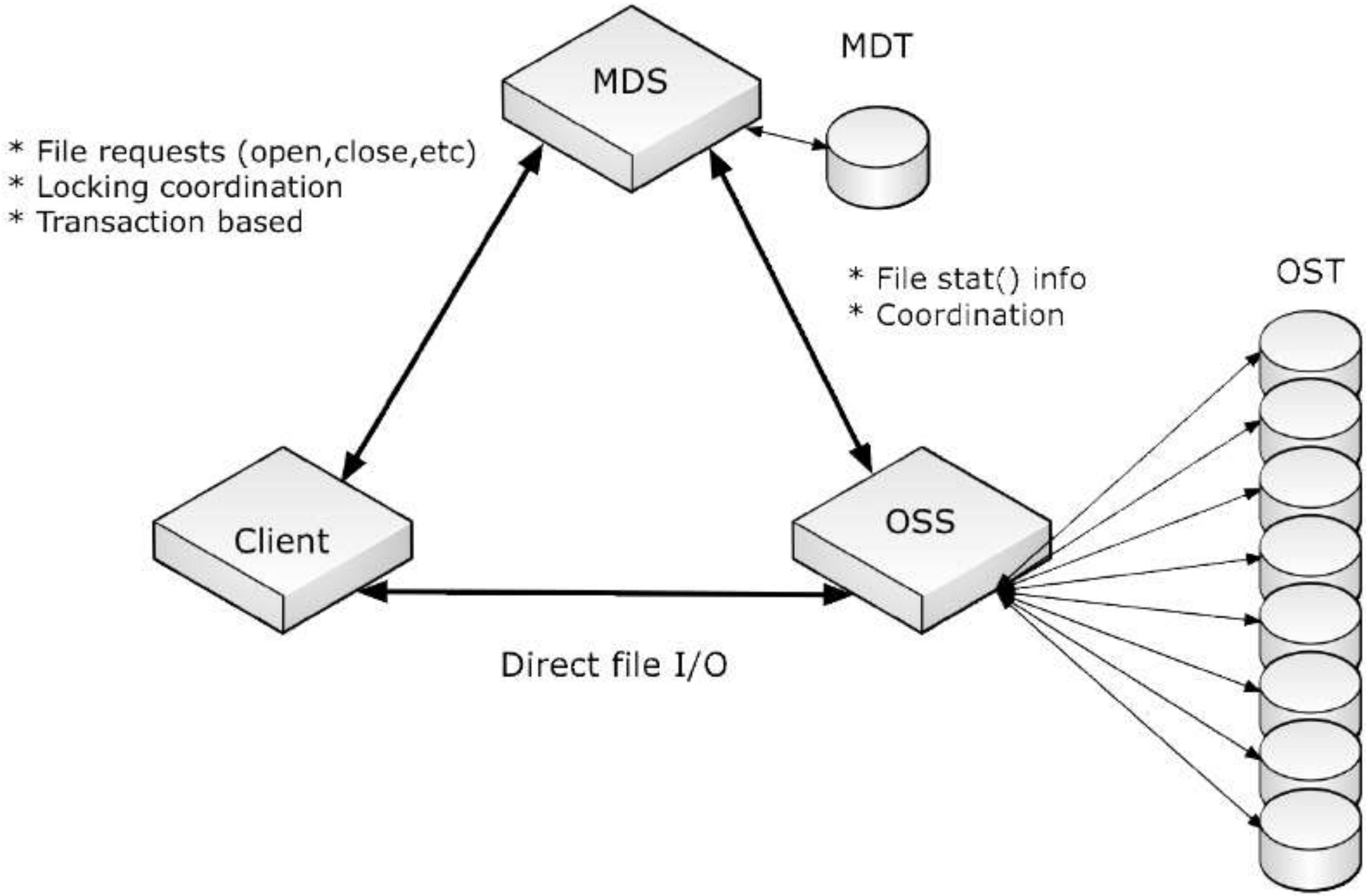
1 Metadata in DFS

Metadata

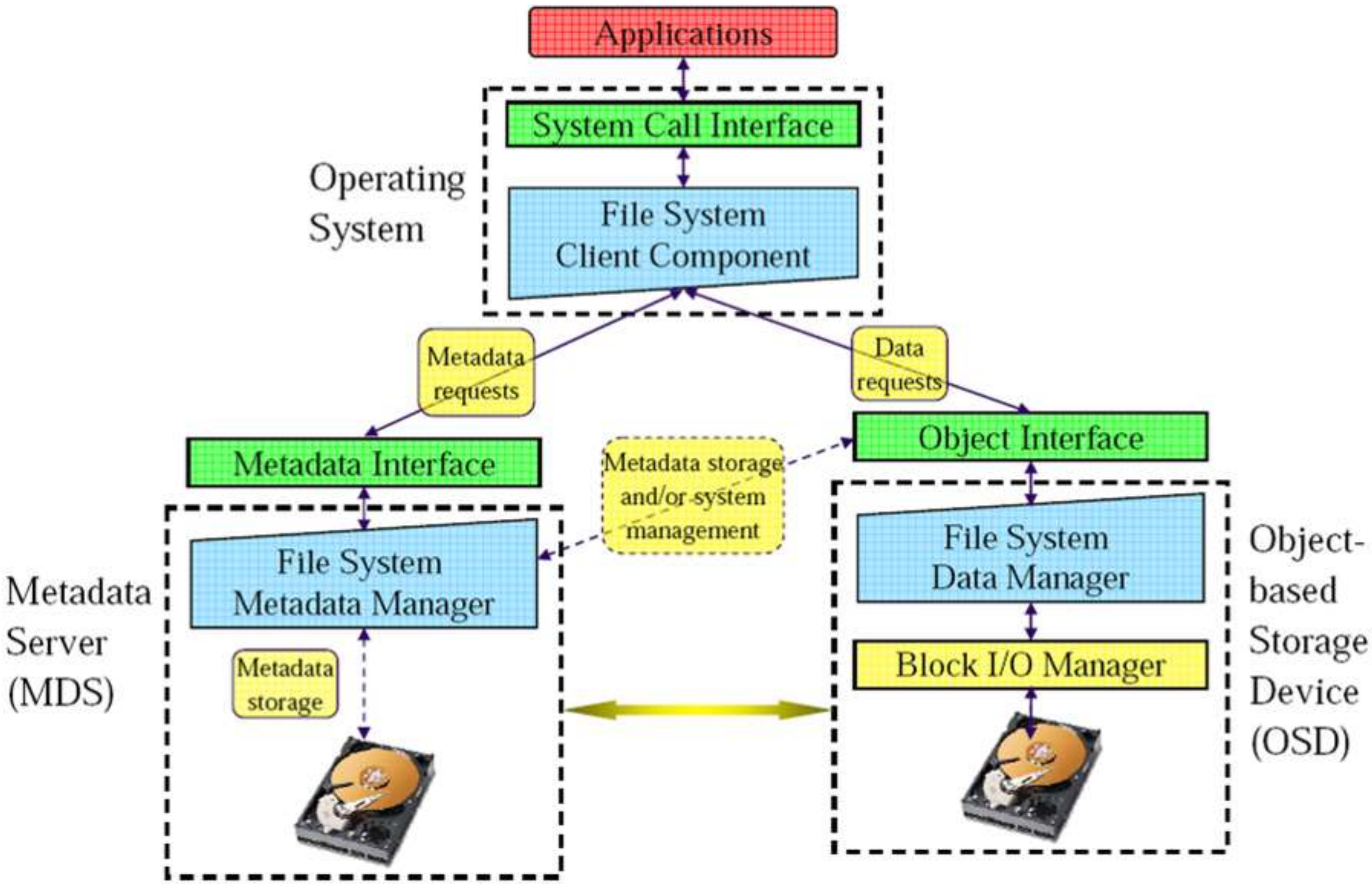
- **Metadata = structural information**
  - File/Objects: attributes in inode/onode
  - Main problem for metadata in DFS: indexing



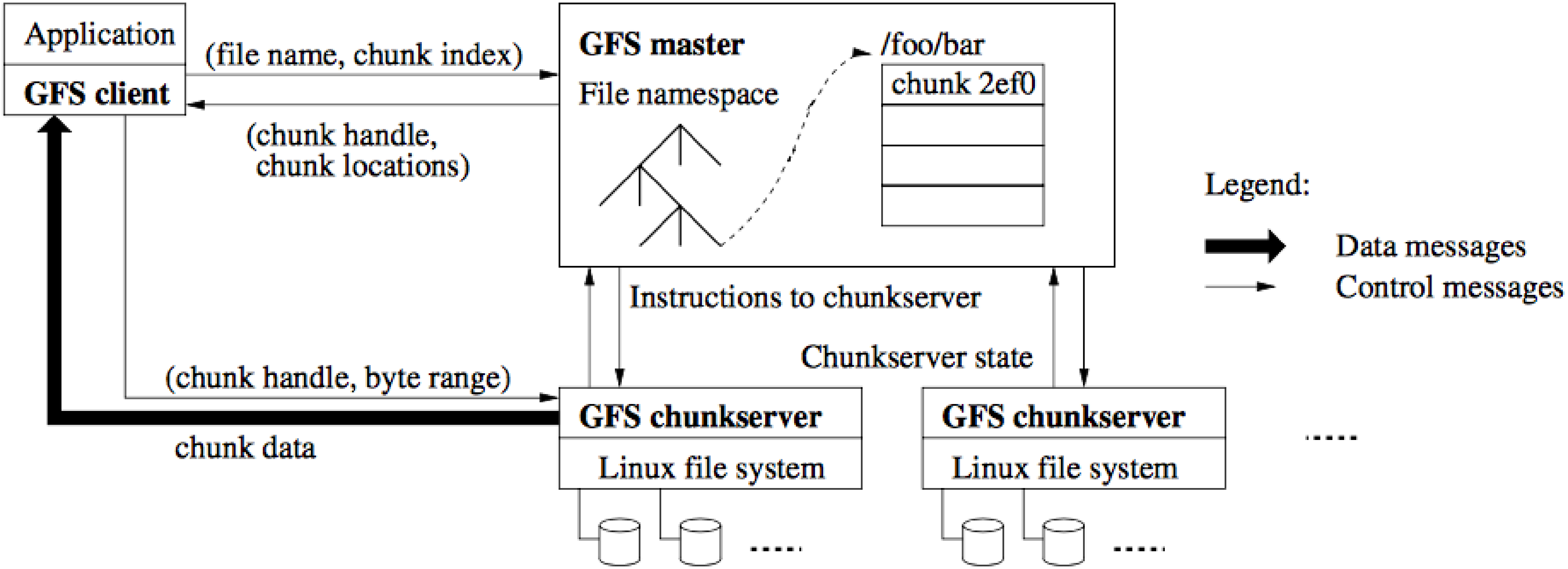
Metadata Server in DFS (Lustre)



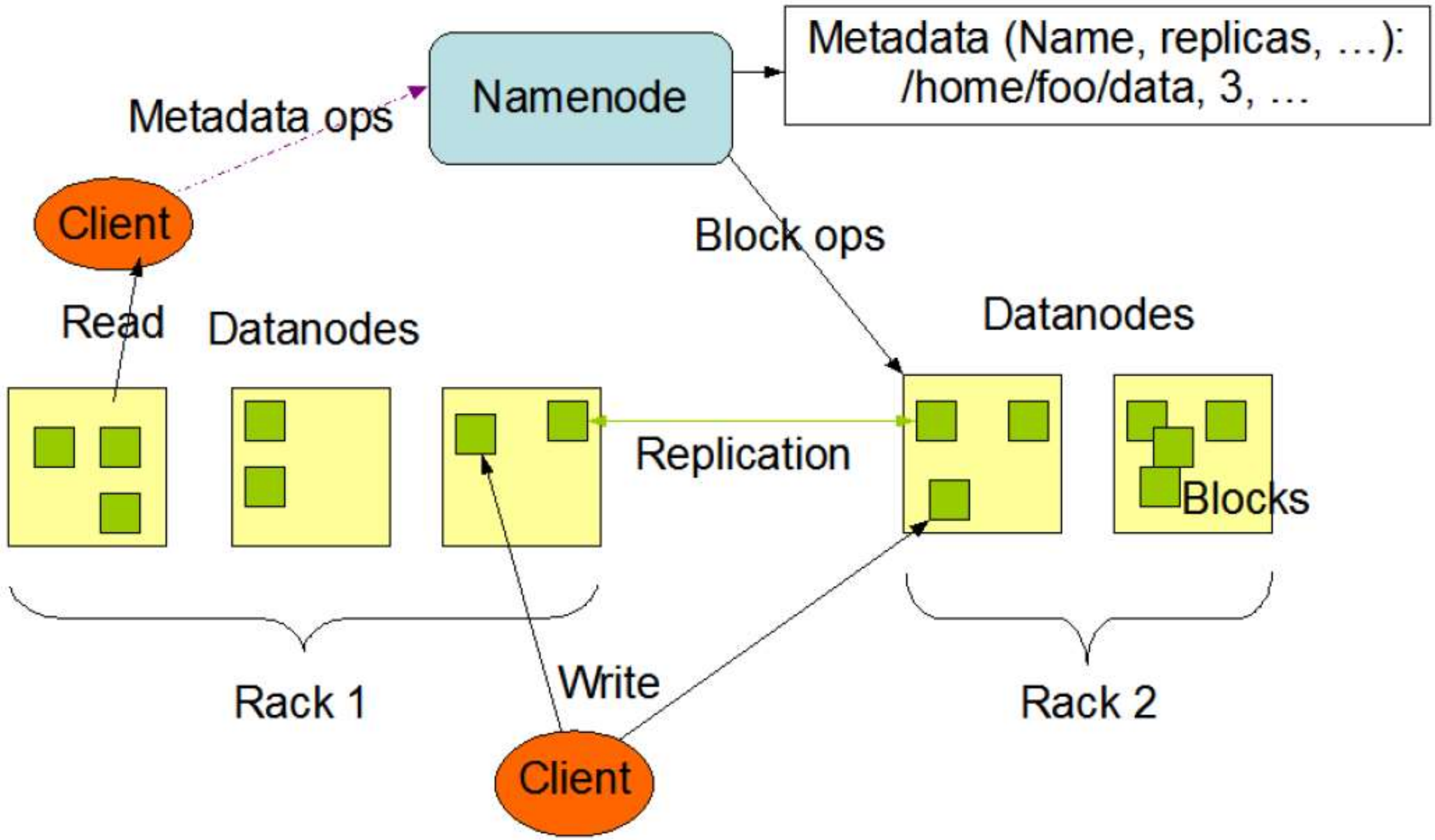
Metadata Server in DFS (Ceph)



Metadata Server in DFS (GFS)



Metadata Server in DFS (HDFS)



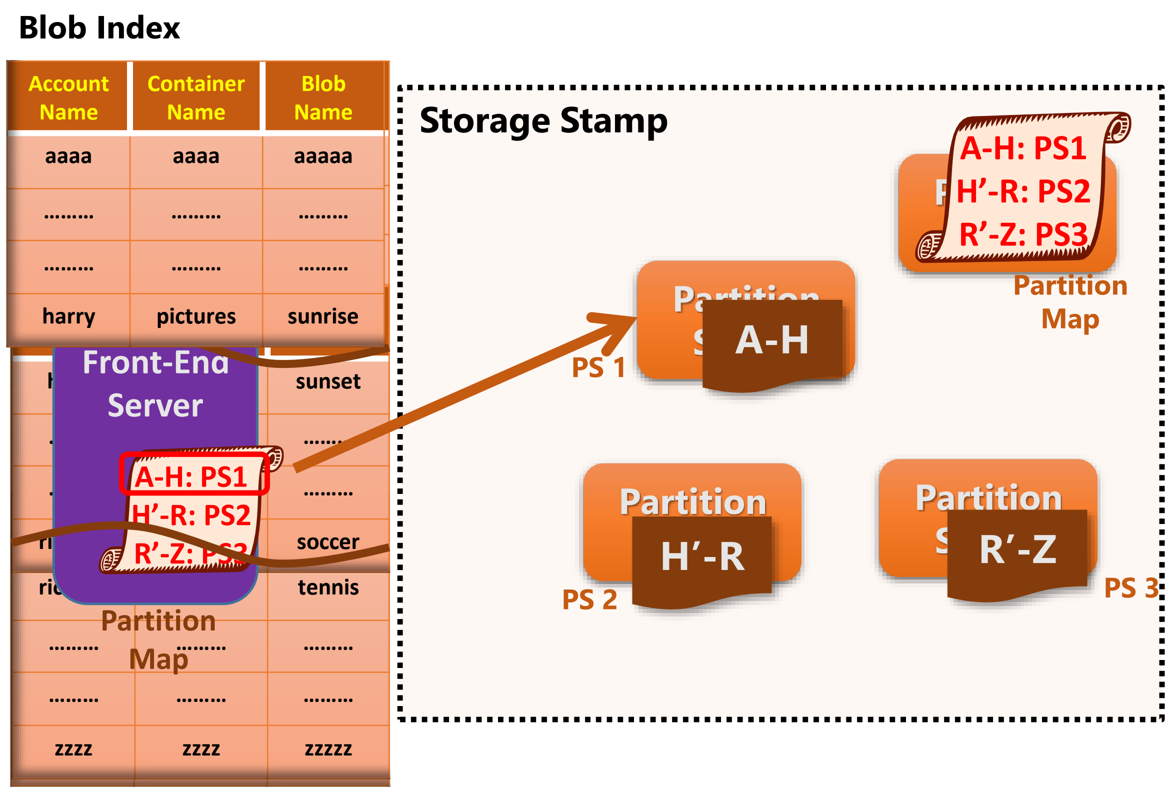
NameNode Metadata in HDFS

- **Metadata in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **Types of Metadata**
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- **A Transaction Log**
  - Records file creations, file deletions. etc

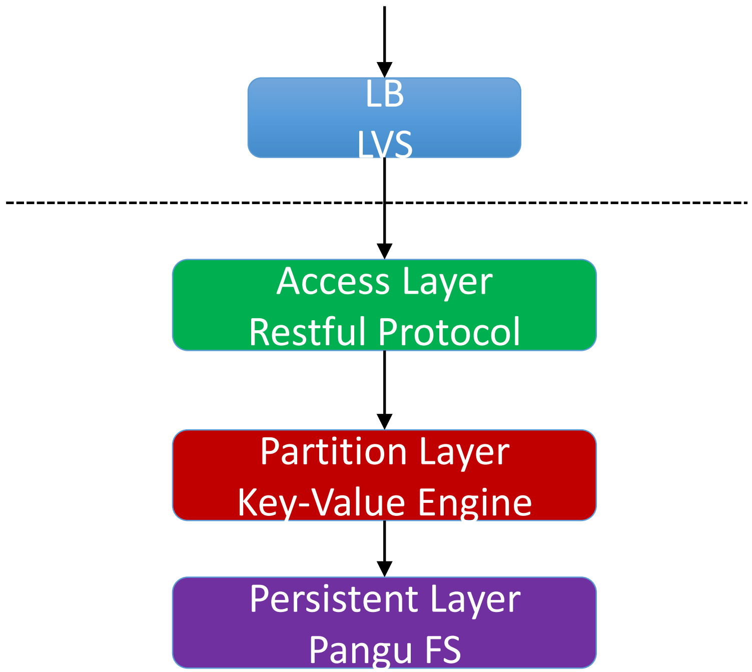
Metadata level in DFS (Azure)
Partition Layer – Index Range Partitioning

Microsoft Azure

- Split index into RangePartitions based on load
- Split at PartitionKey boundaries
- PartitionMap tracks Index RangePartition assignment to partition servers
- Front-End caches the PartitionMap to route user requests
- Each part of the index is assigned to only one Partition Server at a time



Metadata level in DFS (Pangu)
Partition layer



Load Balancing

Protocol Manager & Access Control

Partition & Index

Persistent, Redundancy & Fault-Tolerance

2

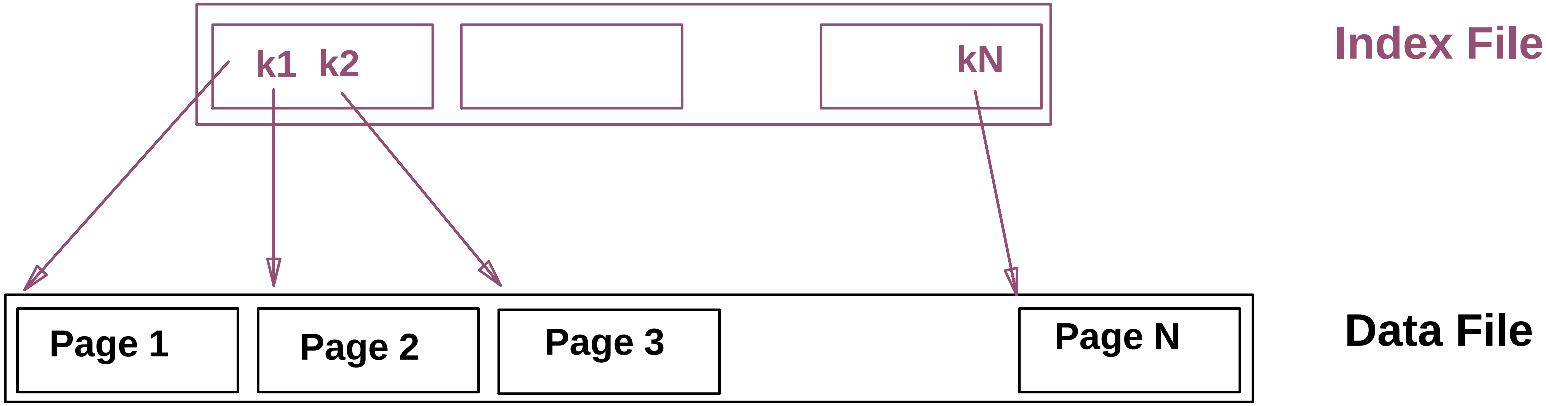
ISAM & B+ Tree

Tree Structures Indexes

- *Recall: 3 alternatives for data entries* k*:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries k*.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
  - *ISAM (Indexed Sequential Access Method)*: static structure
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

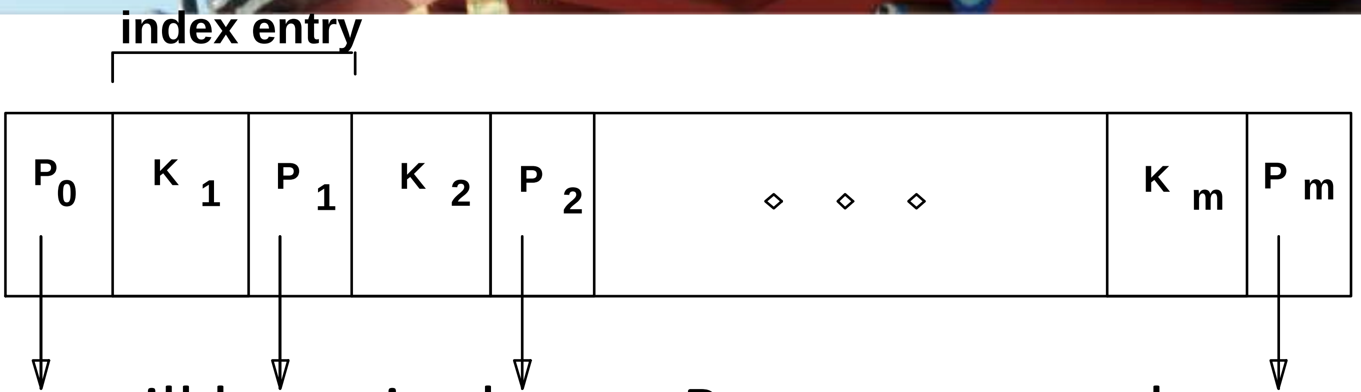
Range Searches

- Choose``*Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.
  - Level of indirection again!

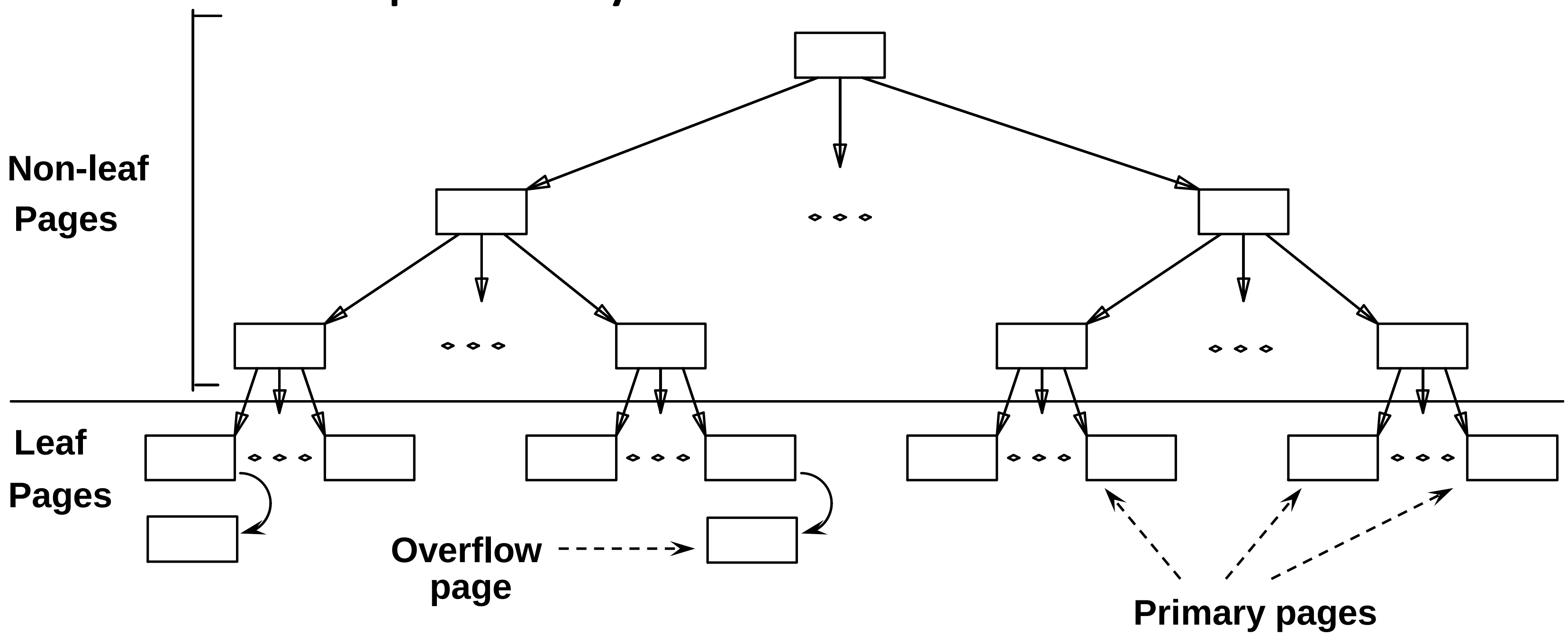


Can do binary search on (smaller) index file!

ISAM

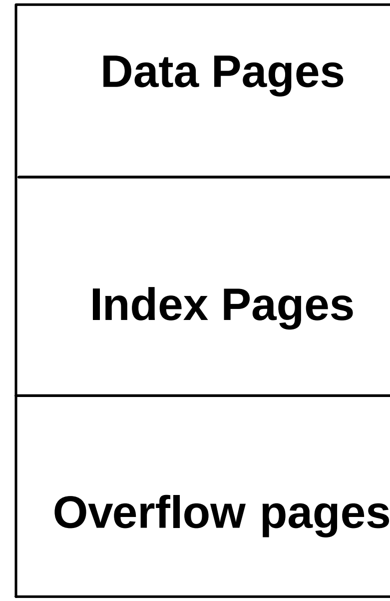


- Index file may still be quite large. But we can apply the idea repeatedly!



Leaf pages contain data entries

Comments on ISAM

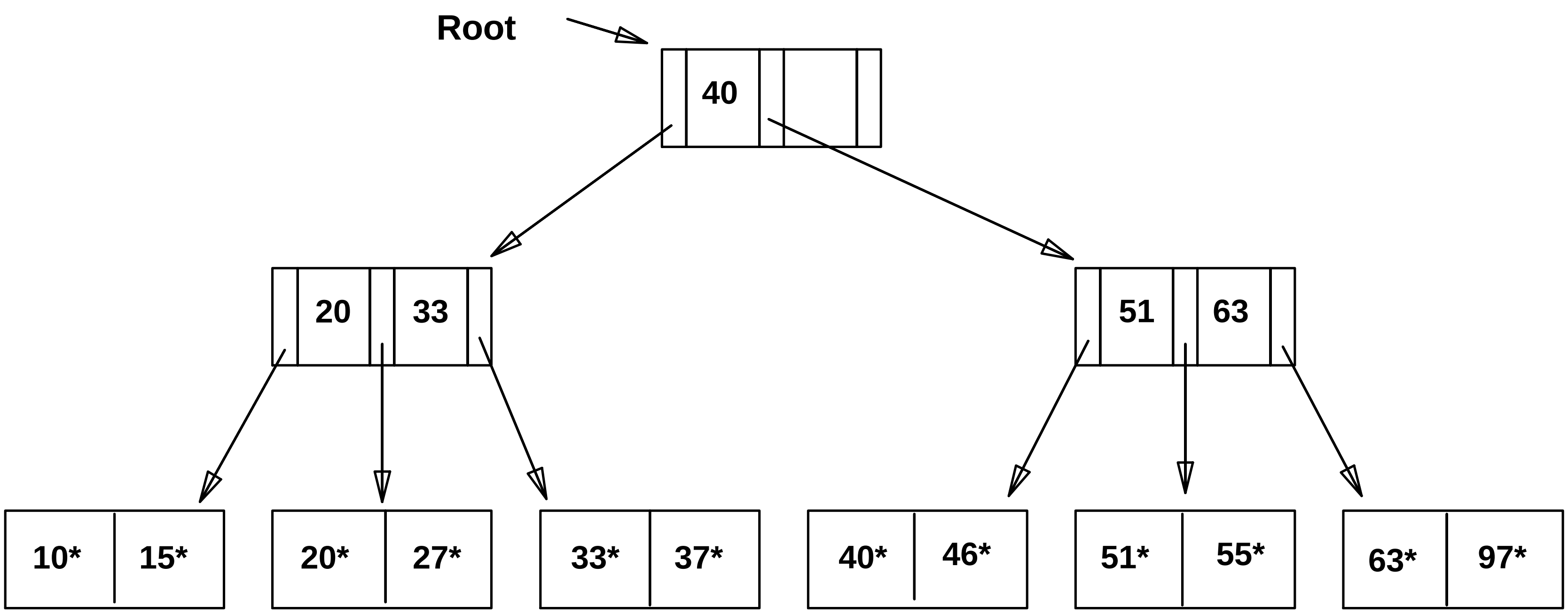


- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key.
  Then index pages allocated.
  Then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf.
  Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf where data entry belongs, put it there. (Could be on an overflow page).
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

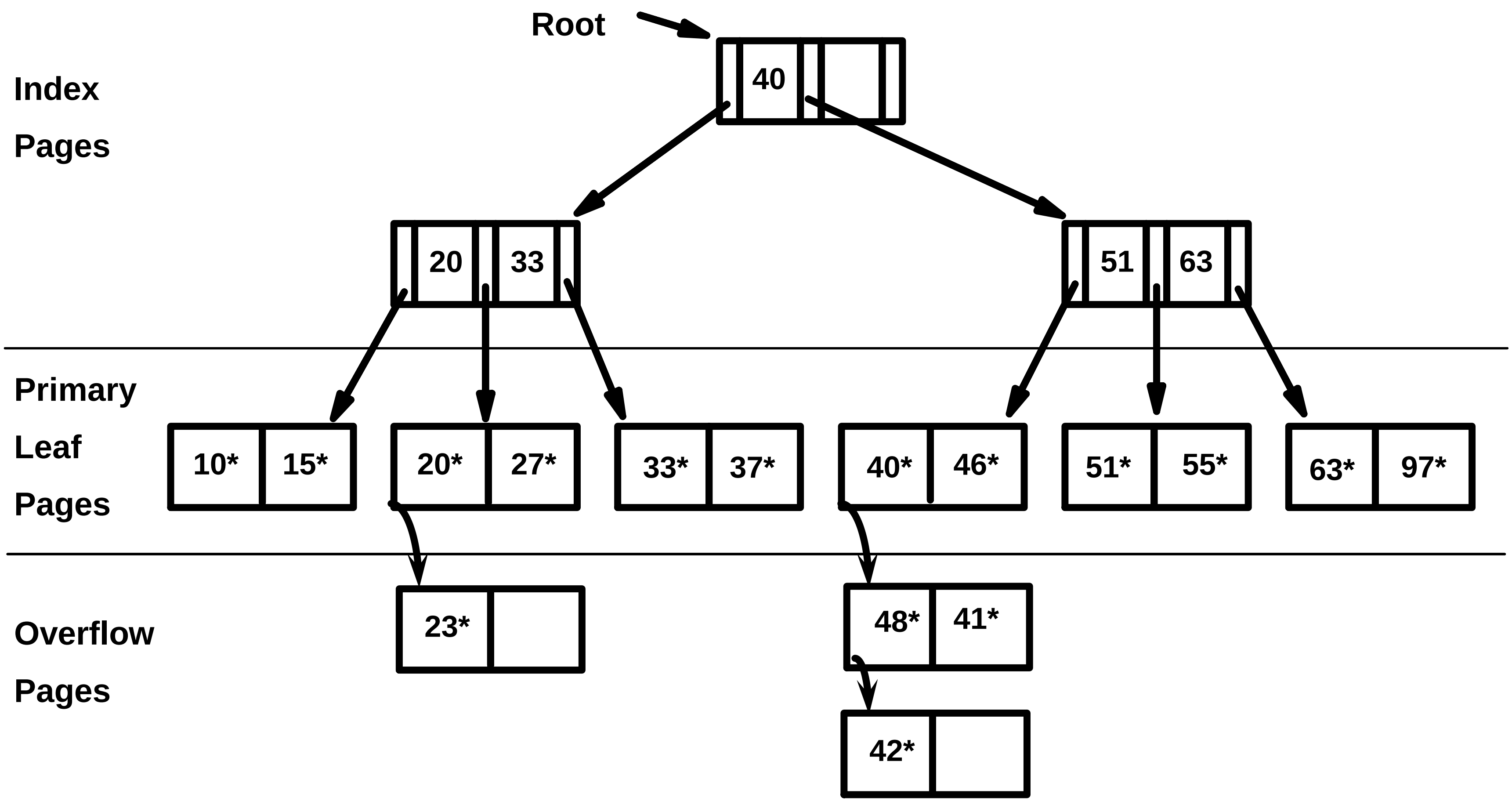
Static tree structure: *inserts/deletes affect only leaf pages*.

Example ISAM Tree

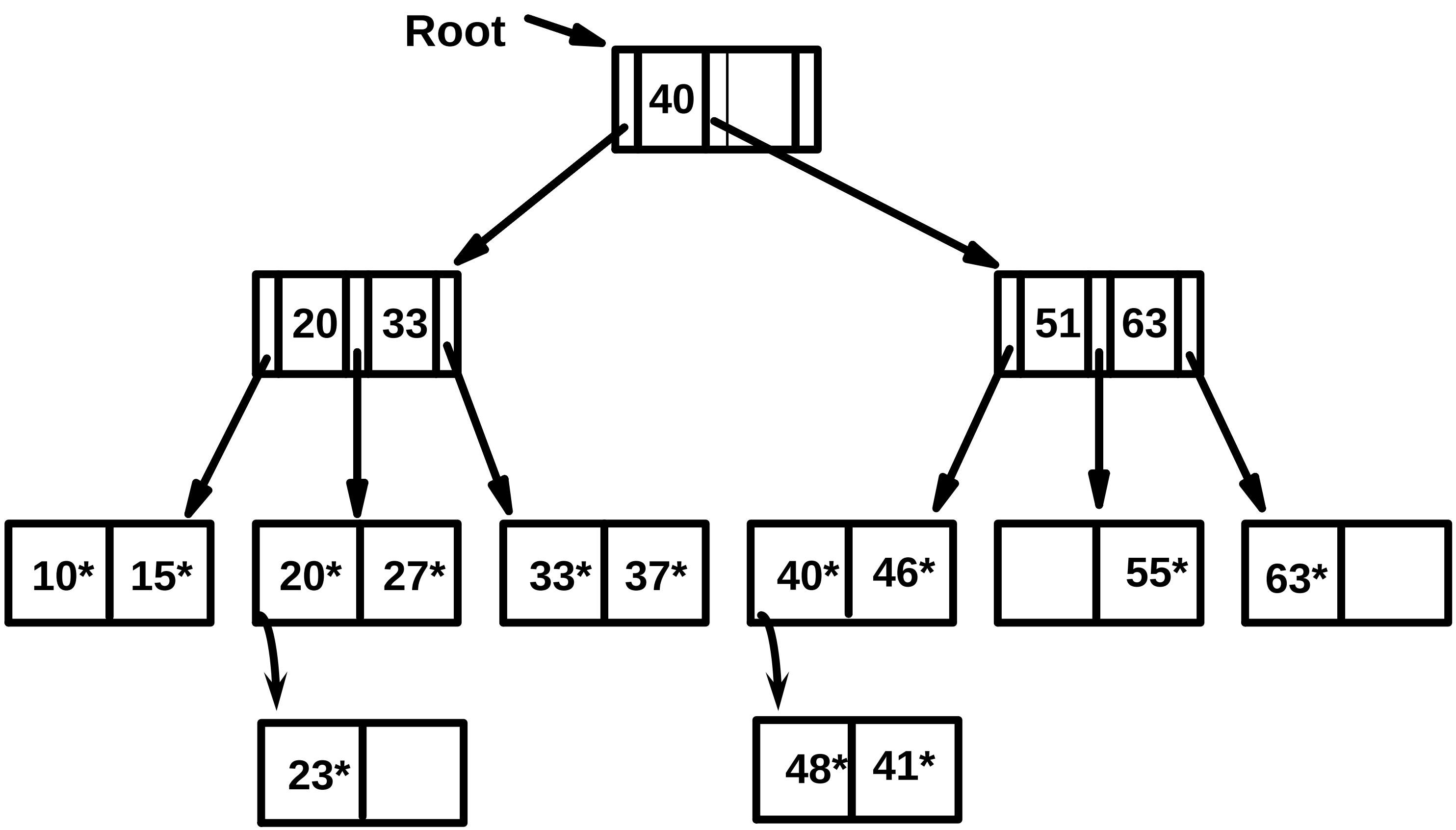
- Each node can hold 2 entries; no need for `next-leaf-page' pointers.



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



... then Deleting 42*, 51*, 97*



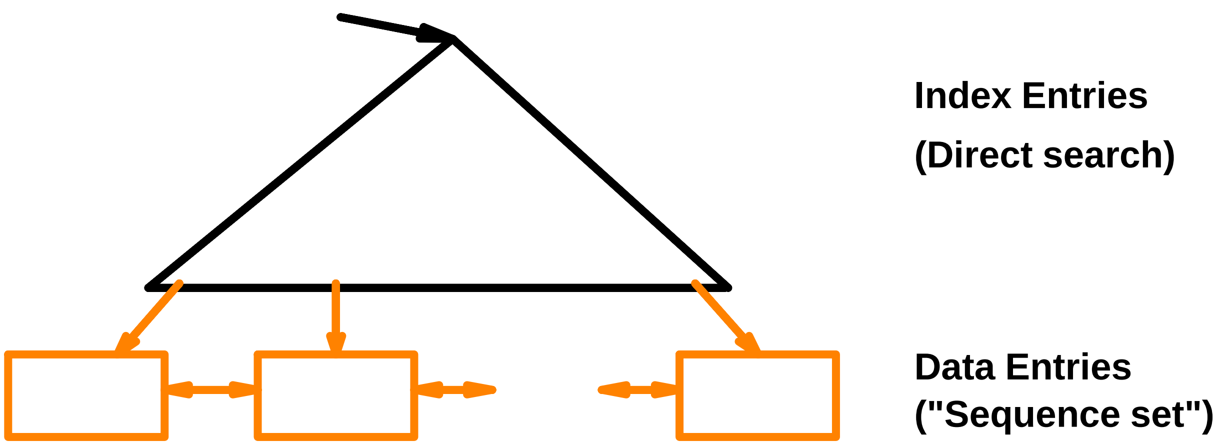
Note that 51 appears in index levels , but 51 not in leaf!*

Pros, Cons & Usage

- Pros
  - Simple and easy to implement
- Cons
  - Unbalanced overflow pages
  - Index redistribution
- Usage
  - MS Access
  - Berkeley DB
  - MySQL (before 3.23) → MyISAM (not real ISAM)

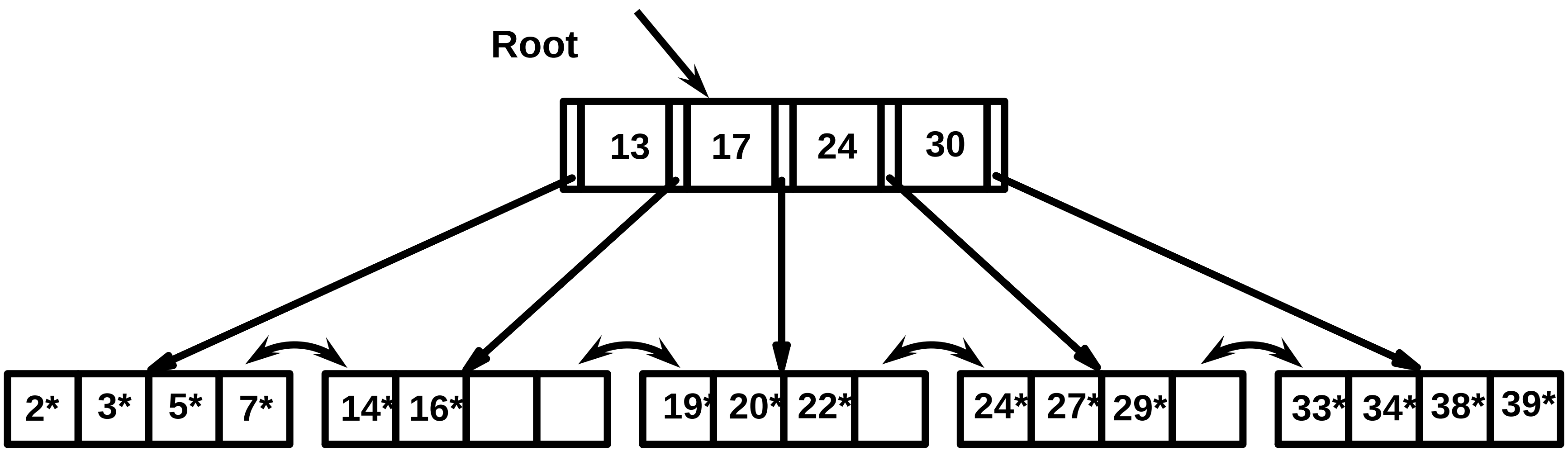
B+ Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains d <= *m* <= 2d entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

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



Based on the search for 15, we <u>know</u> it is not in the tree!*

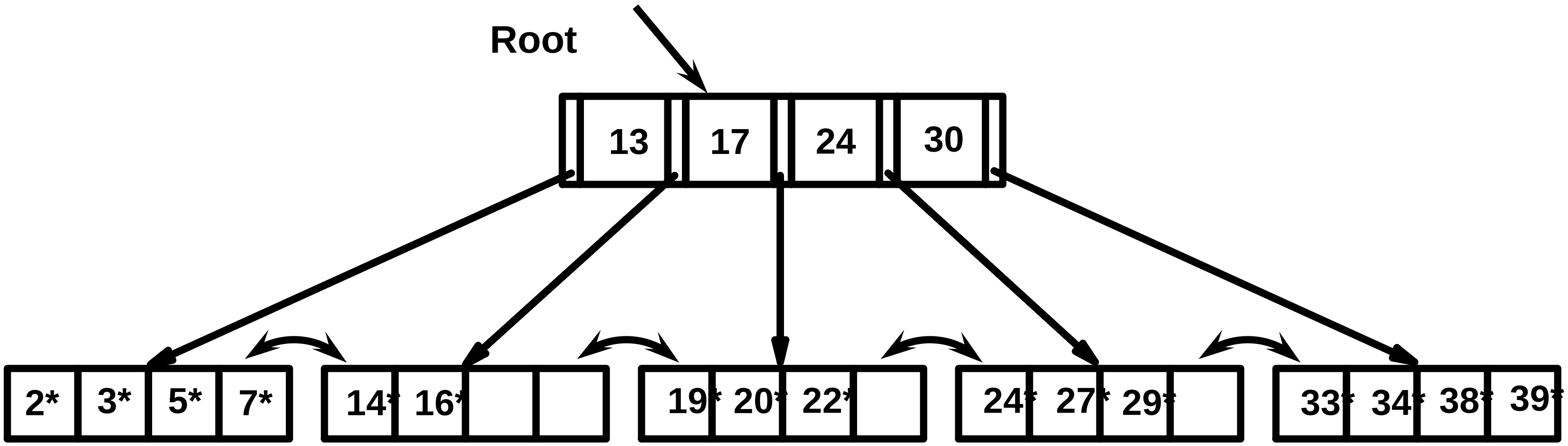
B+ Tree in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average 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

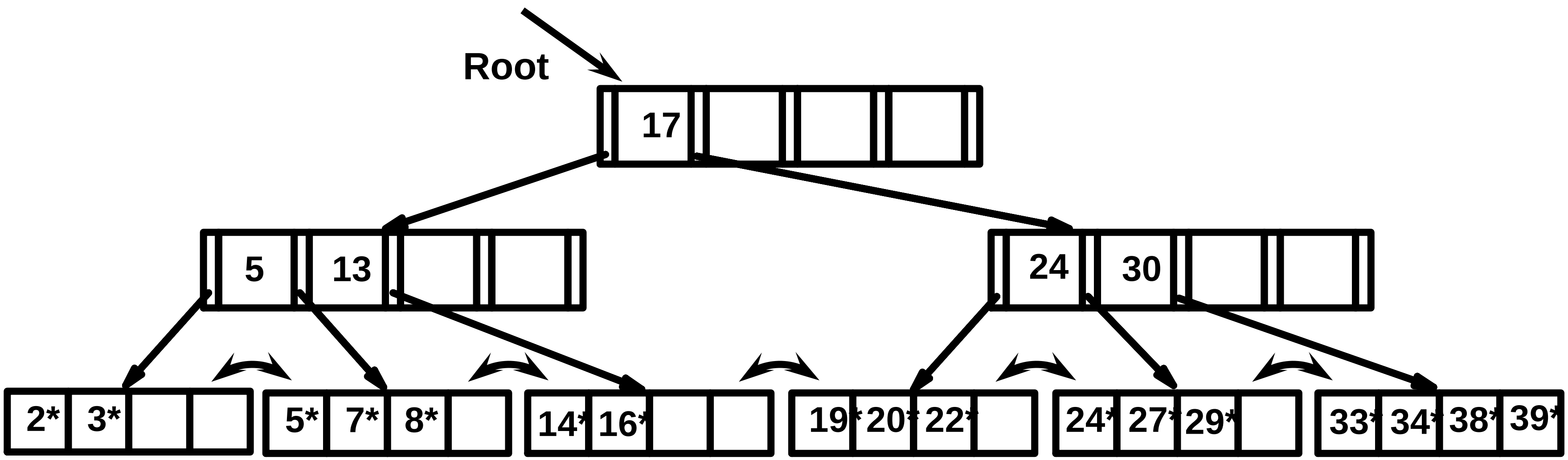
Inserting a Data Entry into a B+ Tree

- 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 B+ Tree - Inserting 8*



Example B+ Tree - Inserting 8*

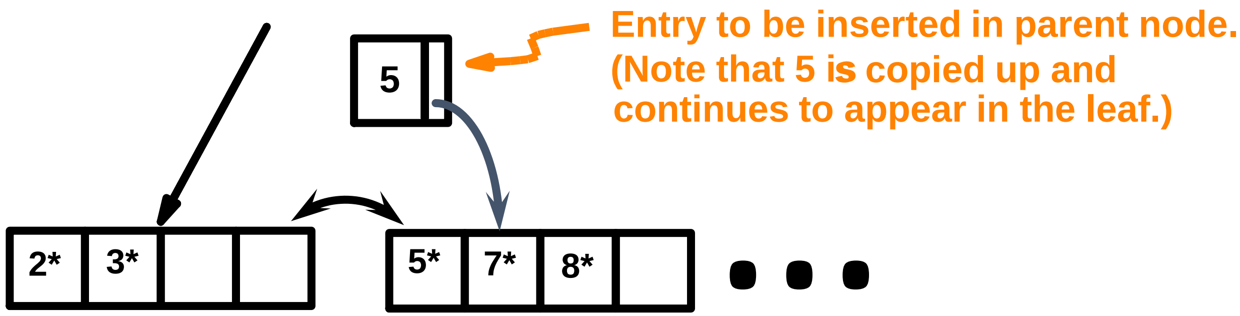


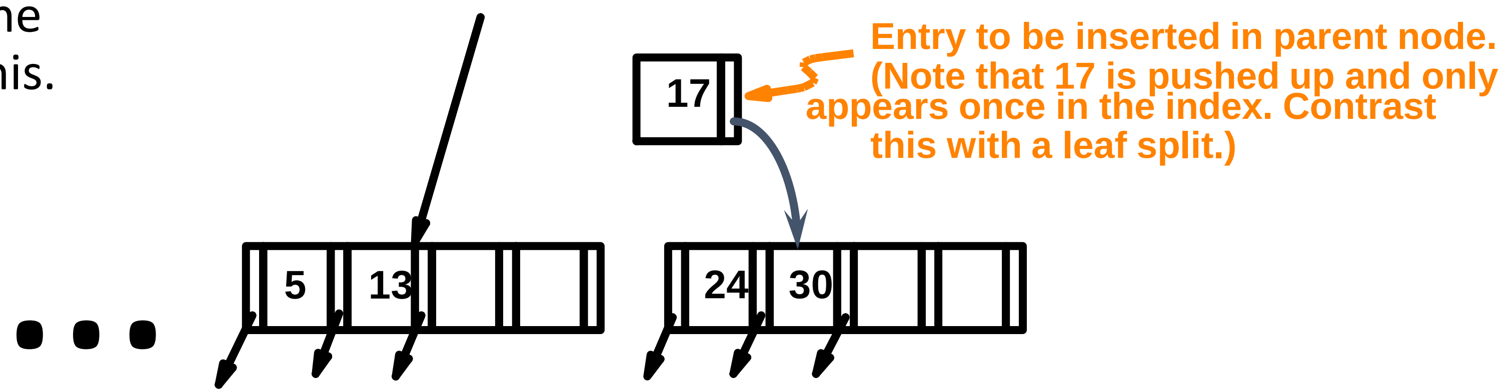
❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

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.

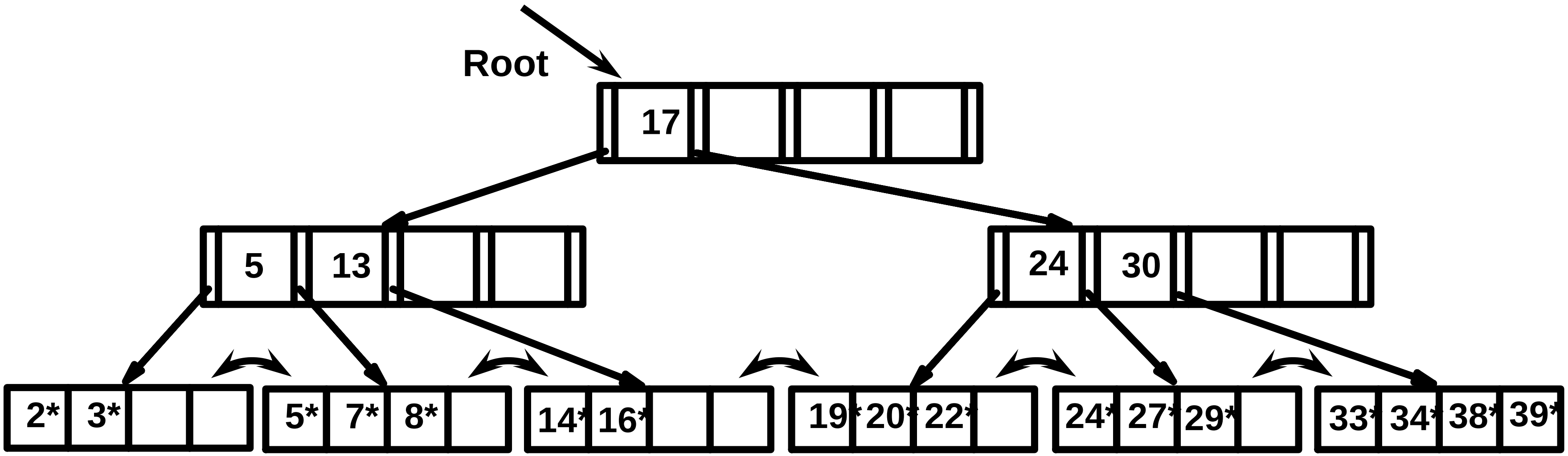




Deleting a Data Entry from a B+ Tree

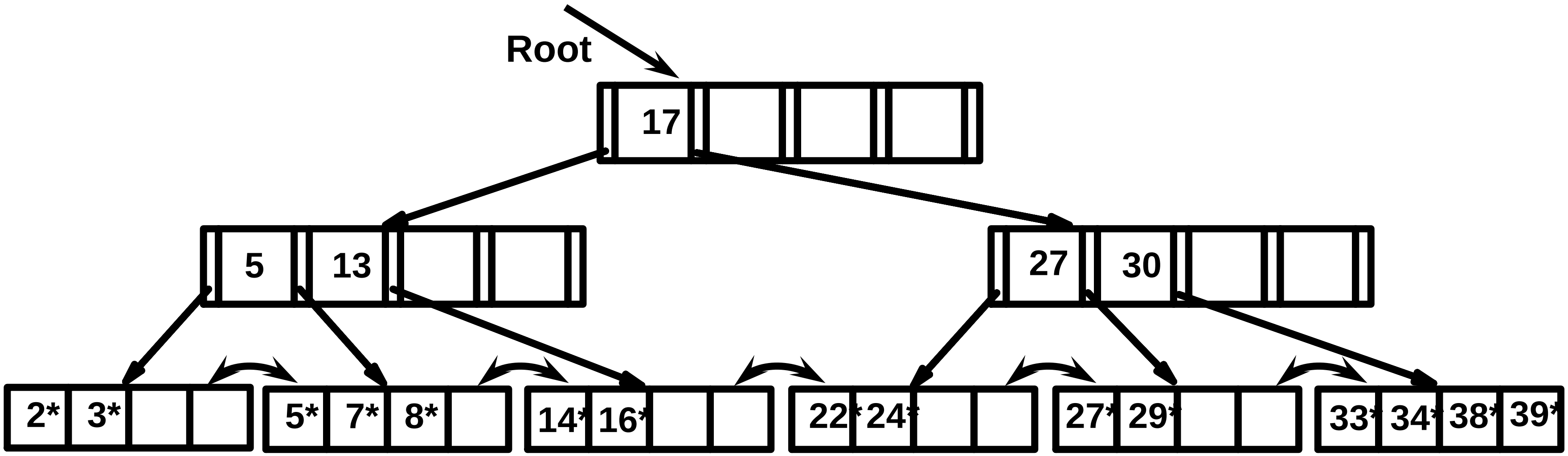
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Example Tree (including 8*) Delete 19* and 20* ...



- Deleting 19* is easy.

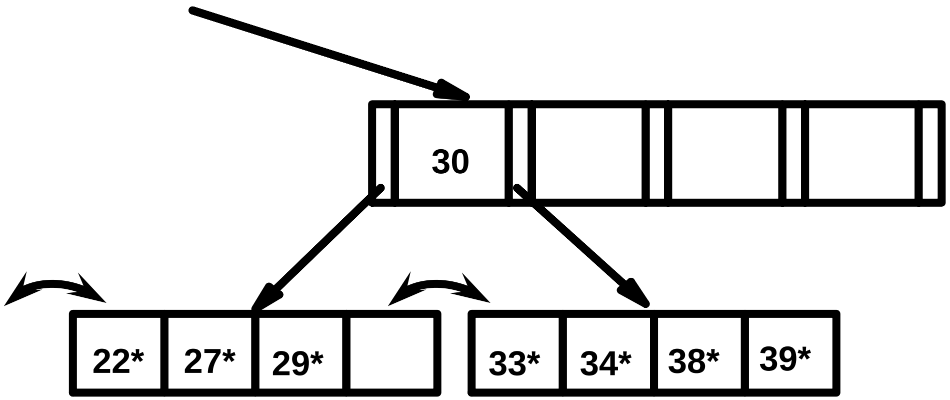
Example Tree (including 8*)
Delete 19* and 20* ...

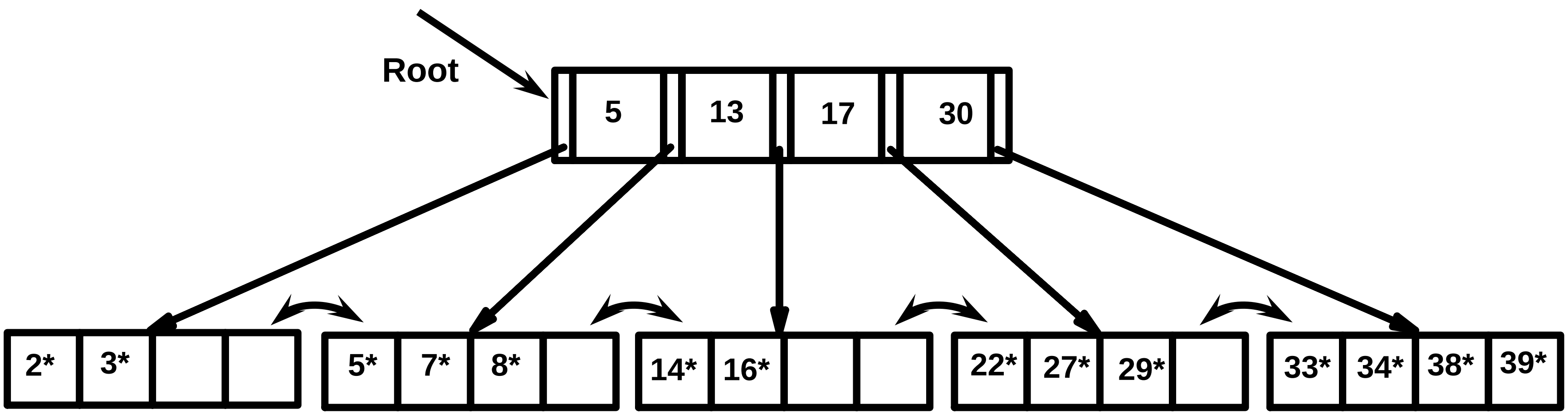


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

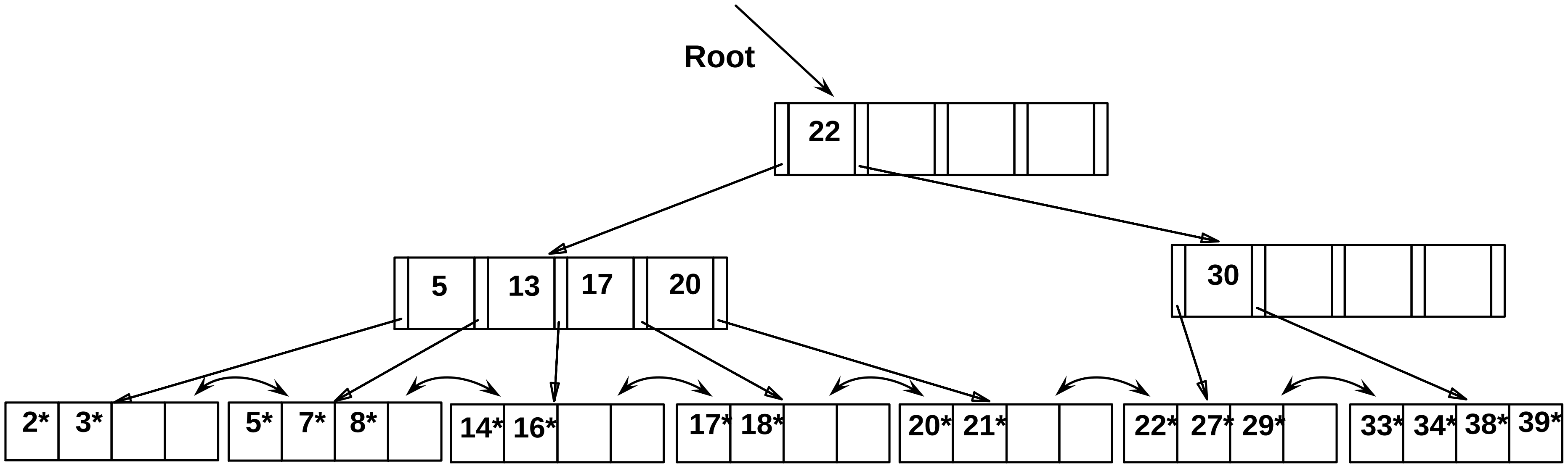
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





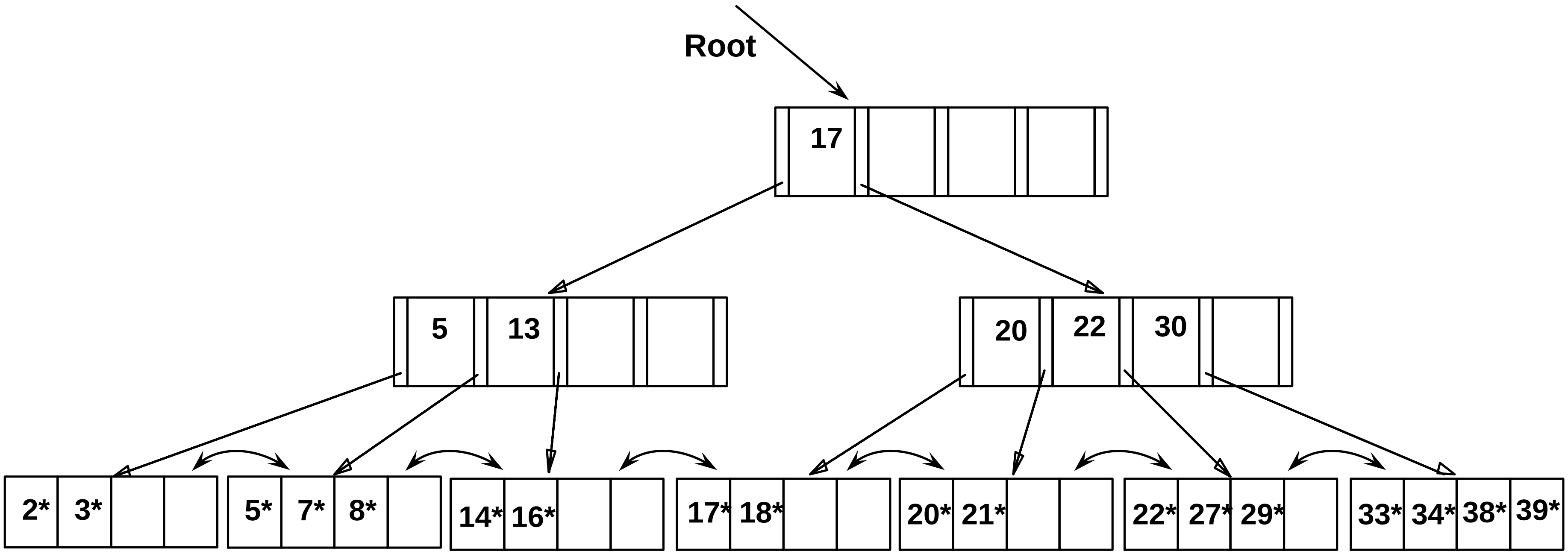
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by *`pushing through'* the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

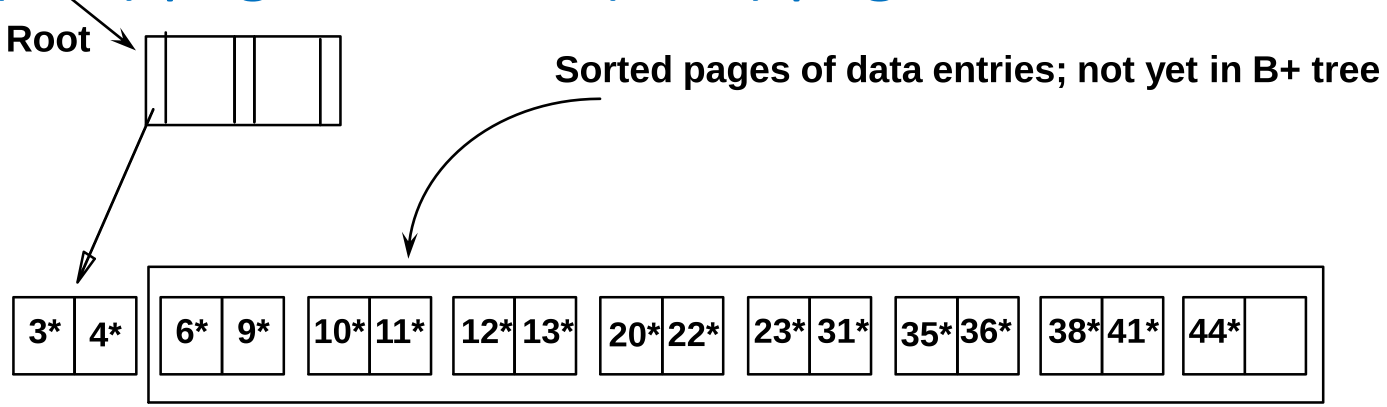


Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

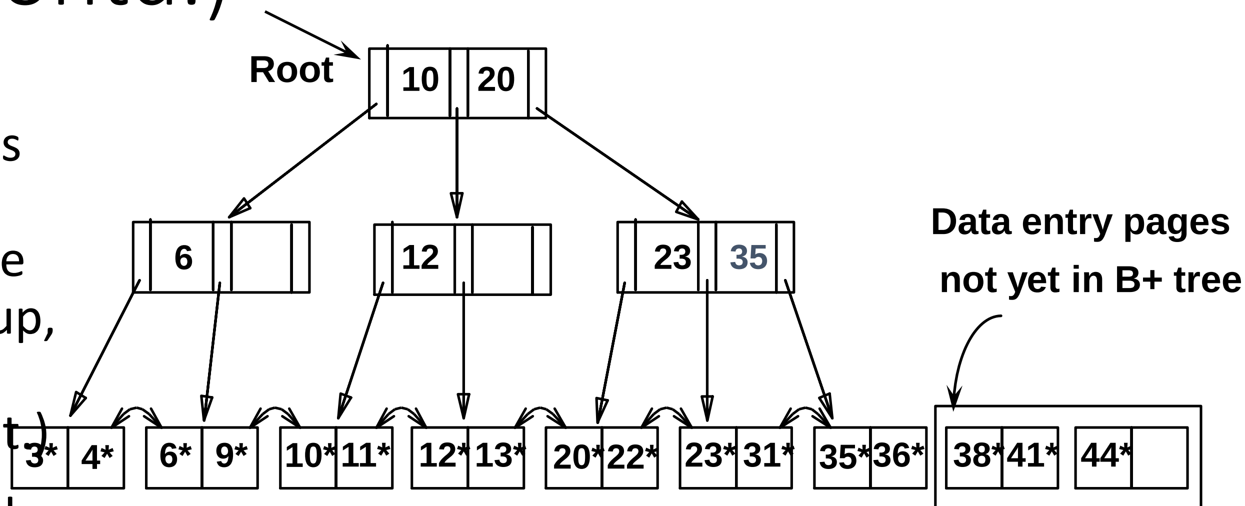
Bulk Loading of a B+ Tree

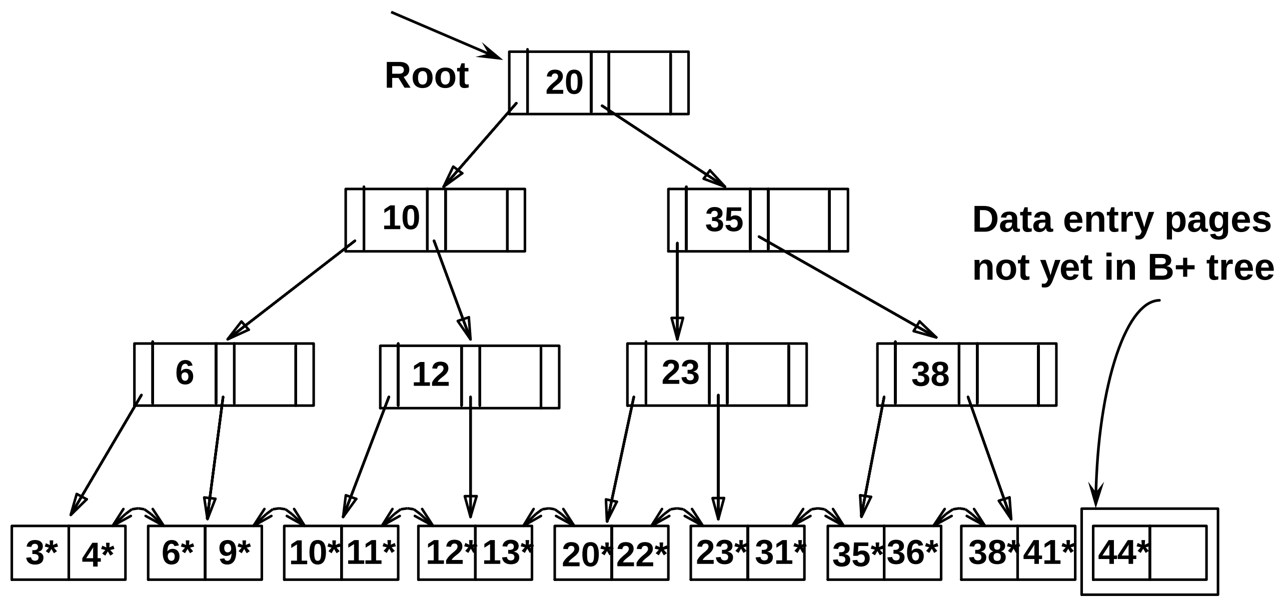
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
    - Also leads to minimal leaf utilization --- why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!





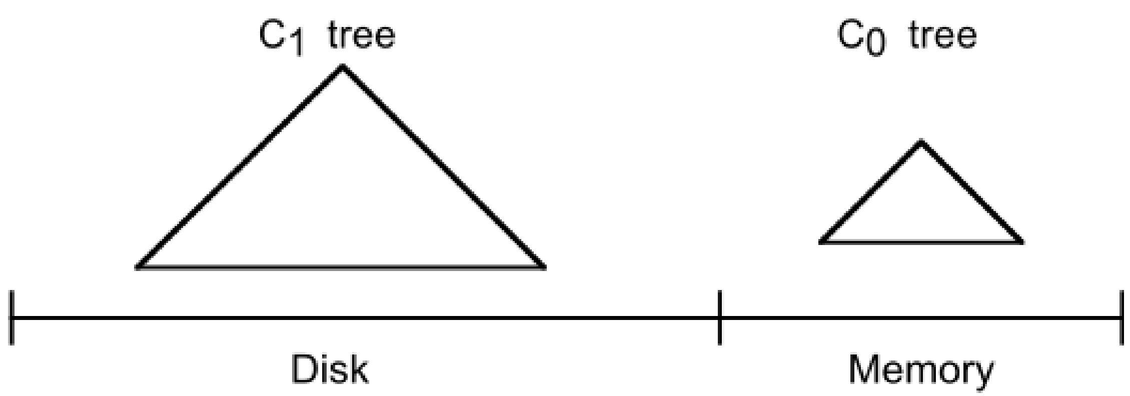
Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

3 Log Structured Merge (LSM) Tree

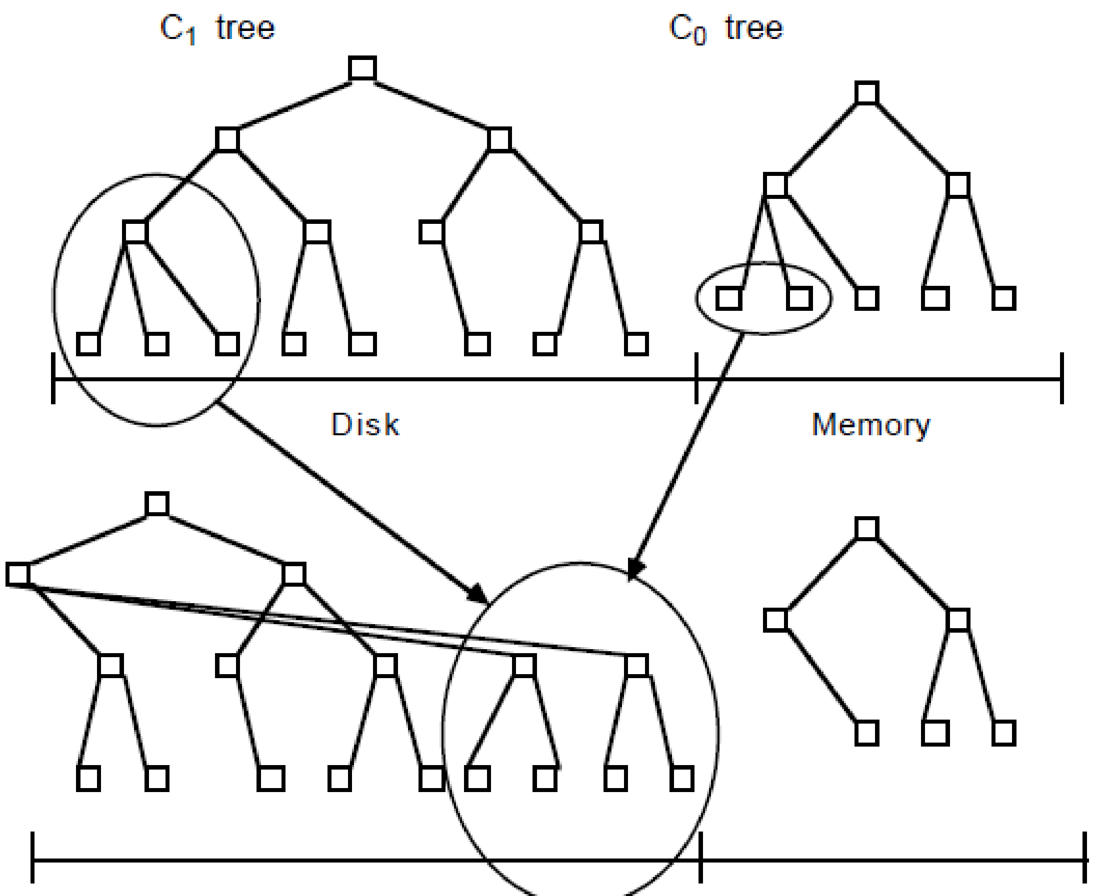
Structure of LSM Tree

- Two trees
  - C_0 tree: memory resident (smaller part)
  - C_1 tree: disk resident (whole part)



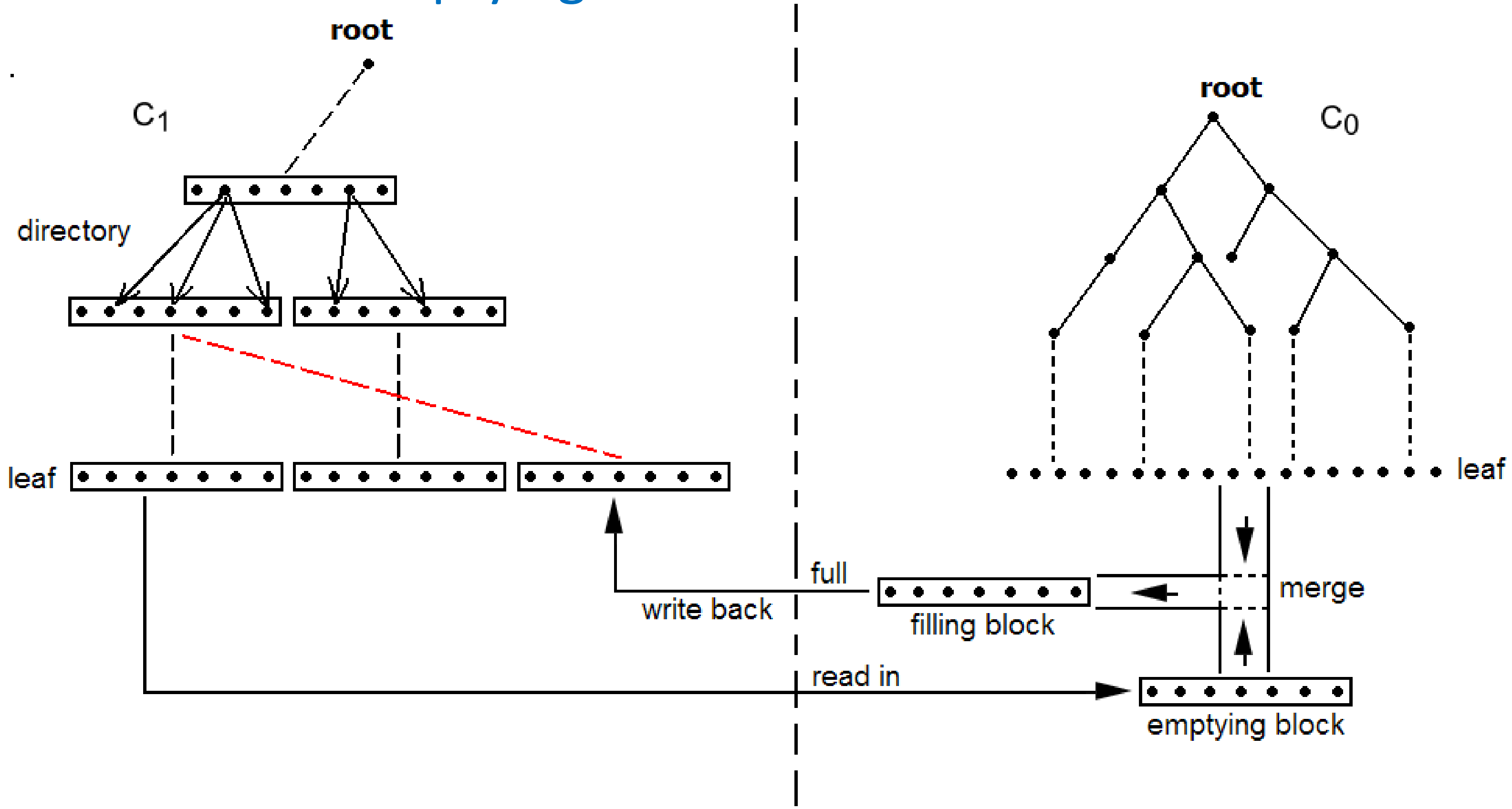
Rolling Merge (1)

- Merge new leaf nodes in C_0 tree and C_1 tree



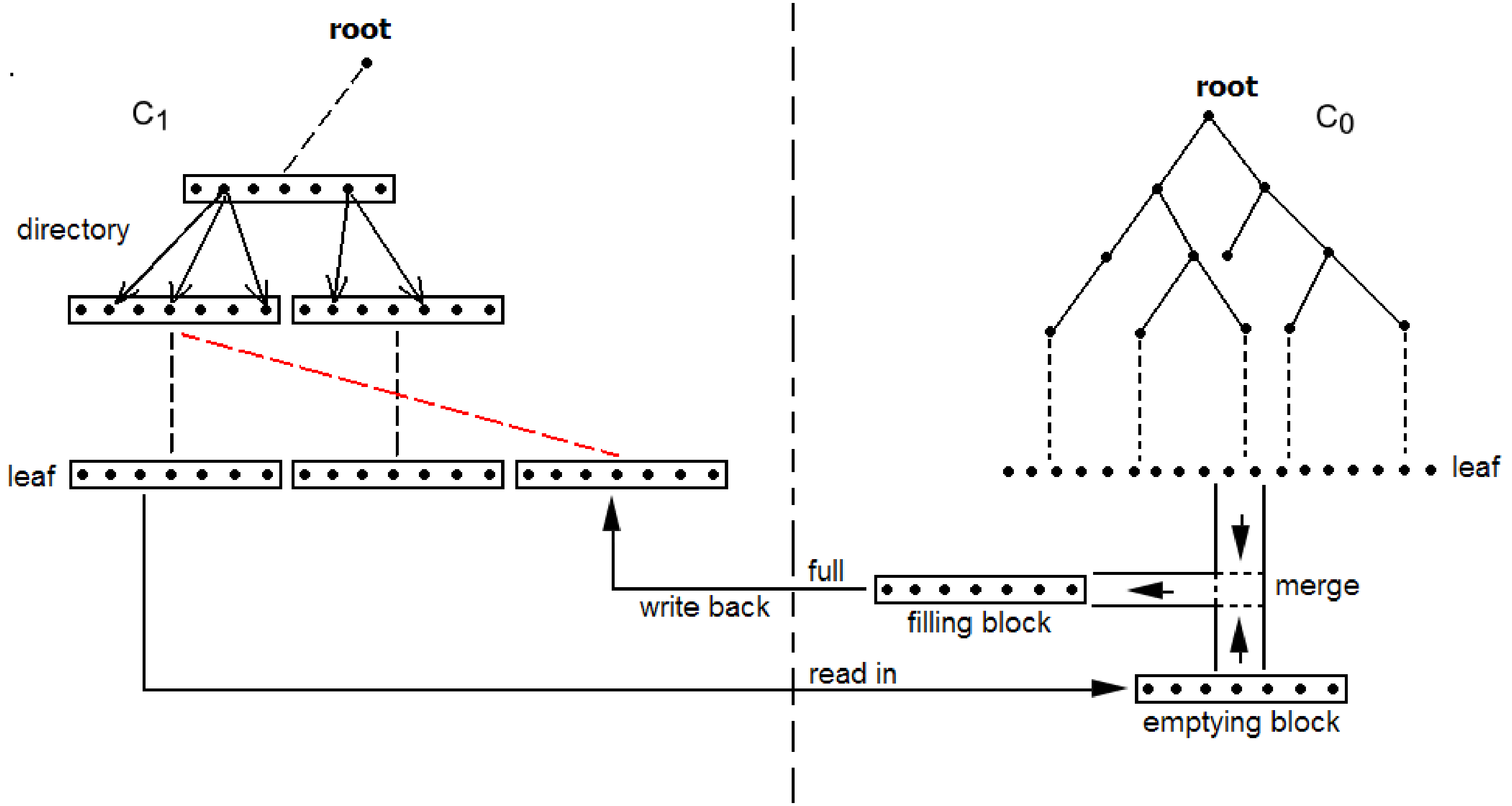
Rolling Merge (2)

- Step 1: read the new leaf nodes from C_1 tree, and store them as emptying block in memory
- Step 2: read the new leaf nodes from C_0 tree, and make merge sort with the emptying block



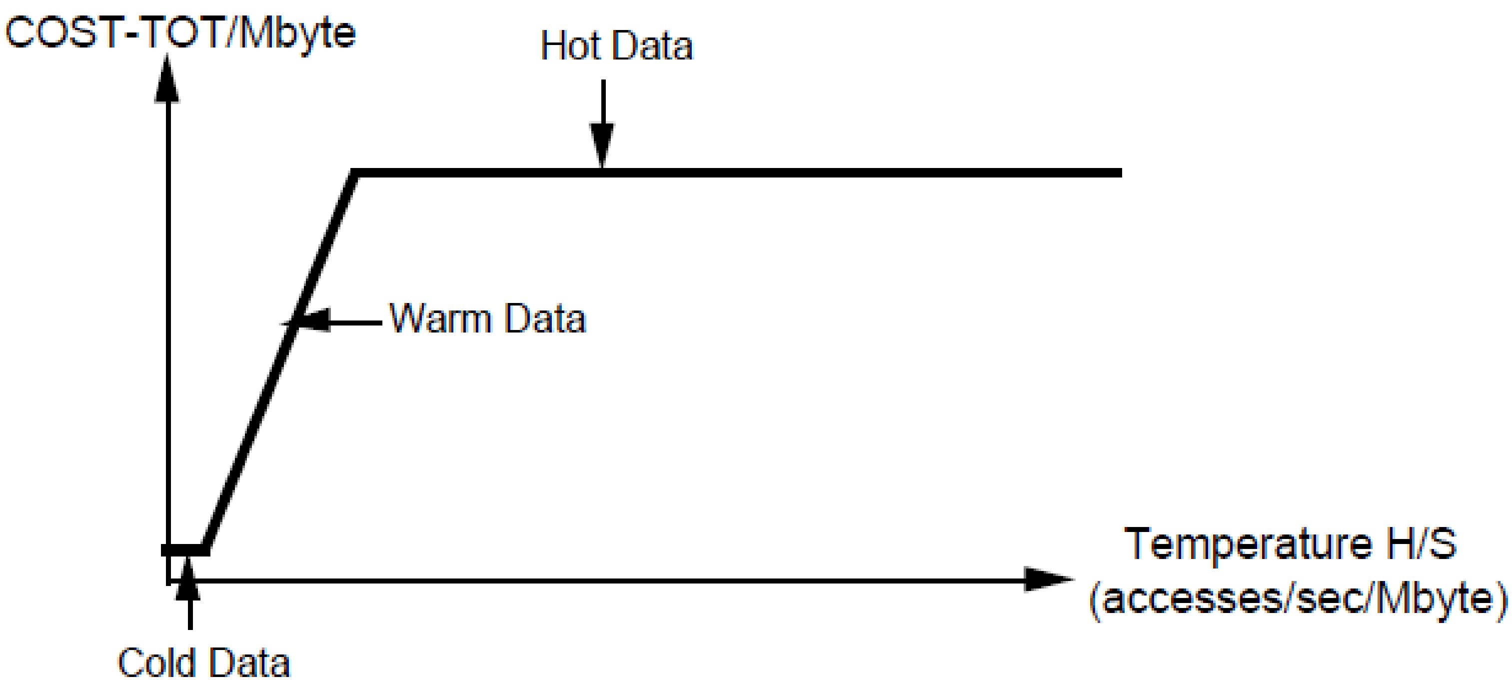
Rolling Merge (3)

- Step 3: write the merge results into filling block, and delete the new leaf nodes in C_0.
- Step 4: repeat step 2 and 3. When the filling block is full, write the filling block into C_1 tree, and delete the corresponding leaf nodes.
- Step 5: after all new leaf nodes in C_0 and C_1 are merged, finish the rolling merge process.



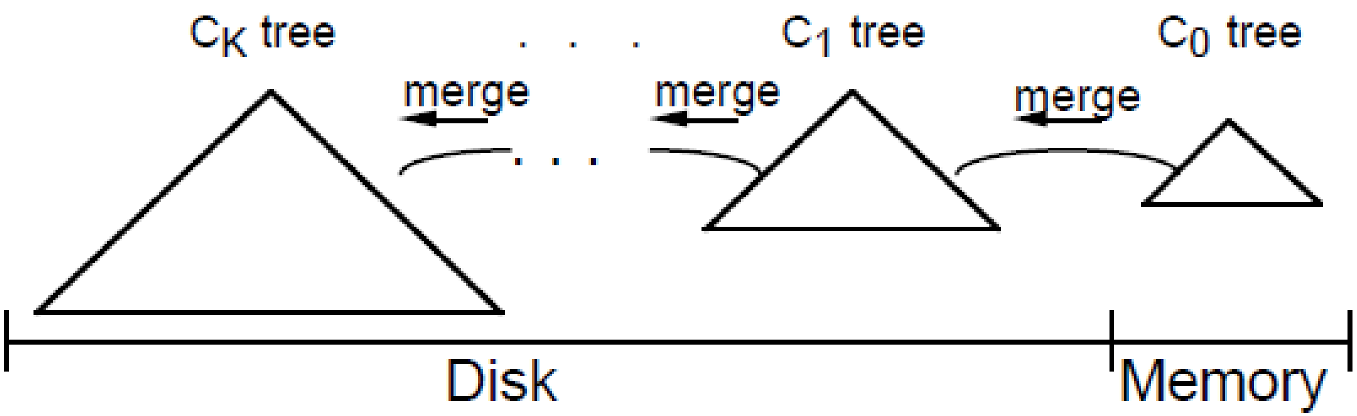
Data temperature

- Data Type
  - Hot/Warm/Cold Data → different trees



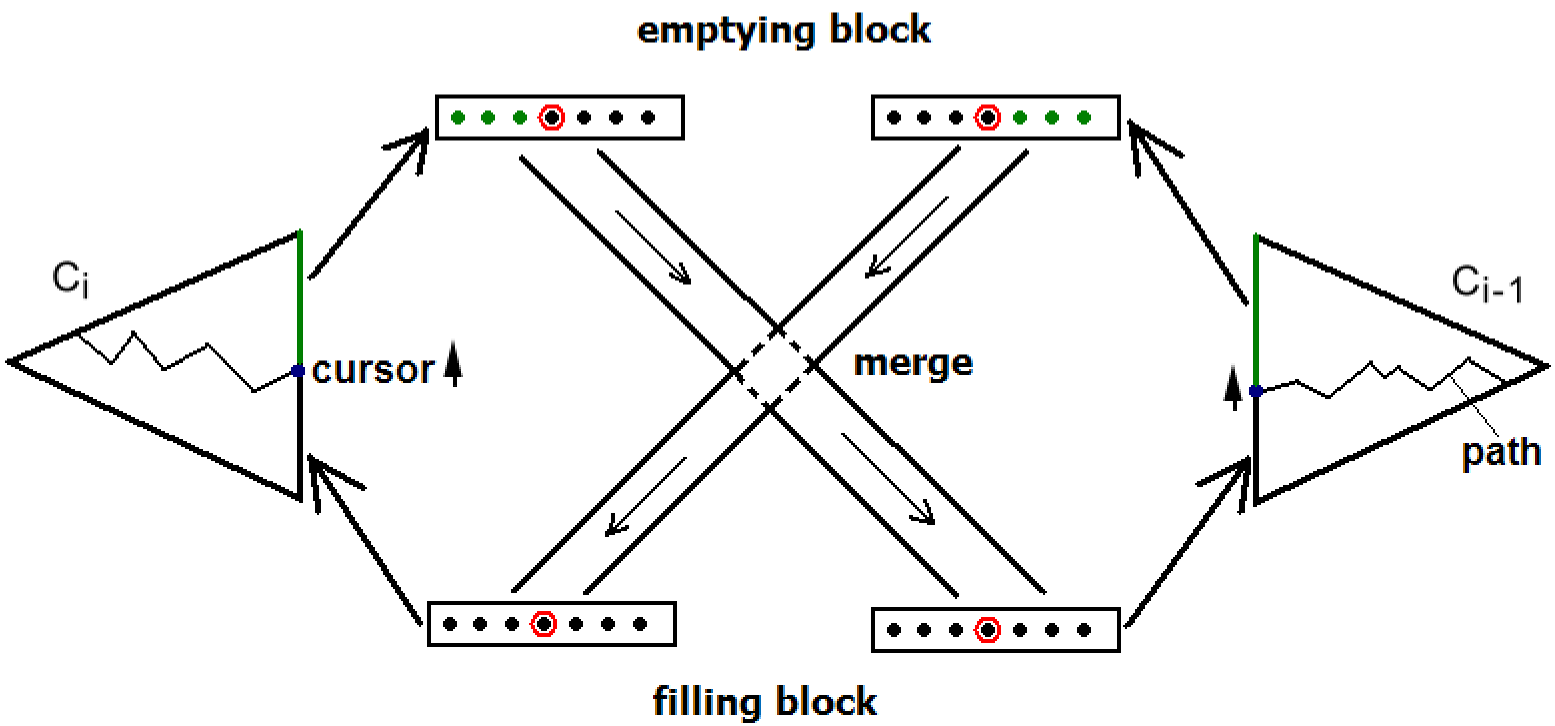
A LSM tree with multiple components

- Data Type
  - Hottest data → C_0 tree
  - Hotter data → C_1 tree
  - ……
  - Coldest data → C_K tree



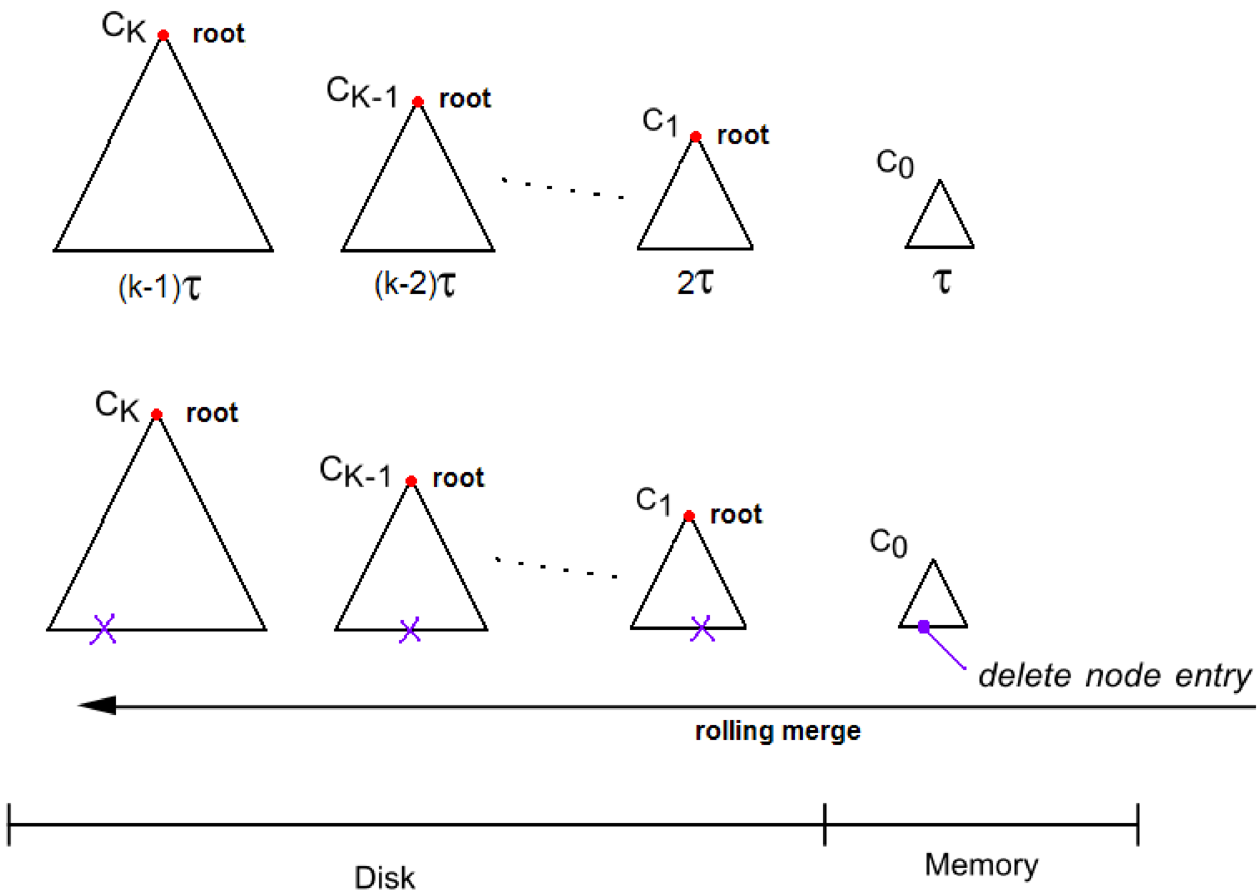
Rolling Merge among Disks

- Two emptying blocks and filling blocks
- New leaf nodes should be locked (write lock)



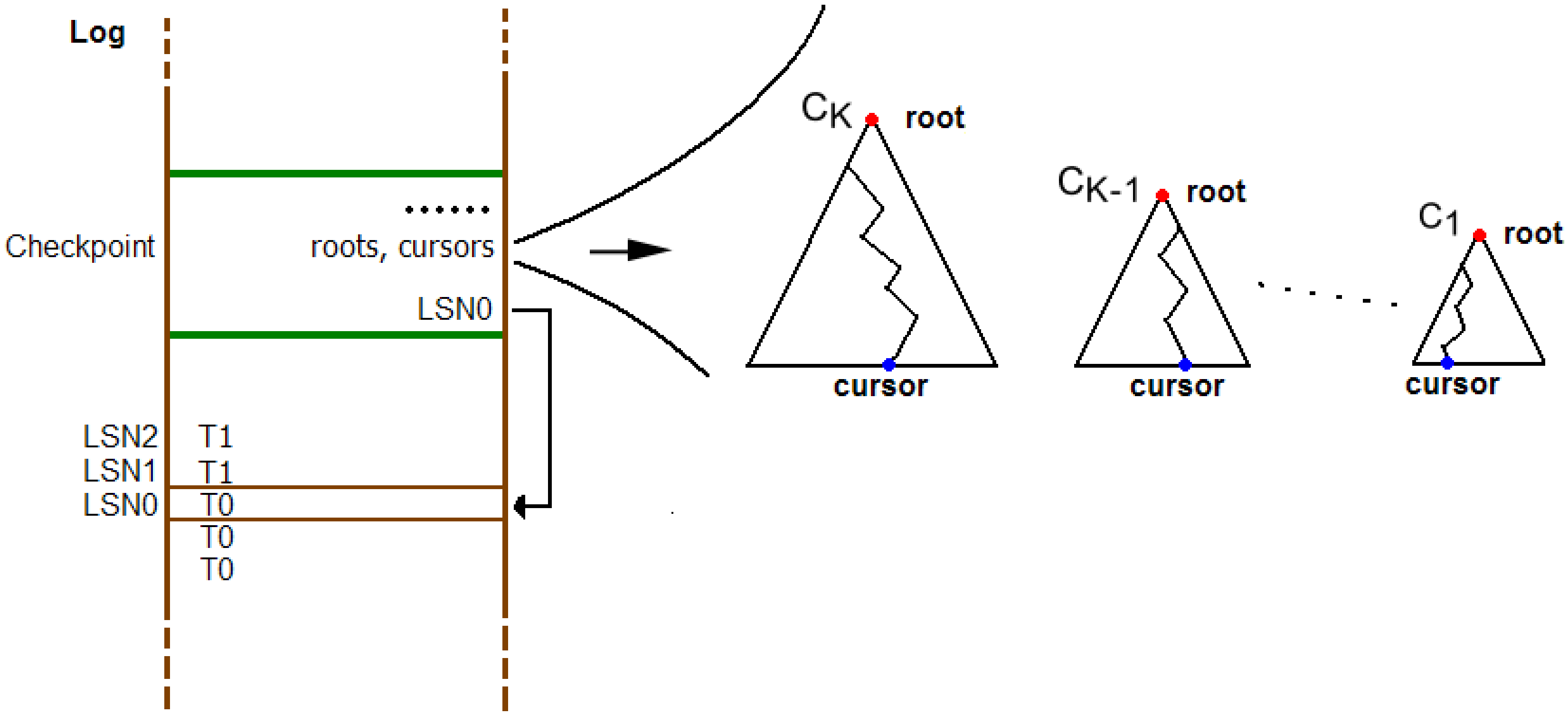
Search and deletion (based on temporal locality)

- Lastest T (0- T) accesses are in C_0 tree
- T - 2T accesses are in C_1 tree
- ……



Checkpointing

- Log Sequence Number (LSN0) of last insertion at Time T_0
- Root addresses
- Merge cursor for each component
- Allocation information



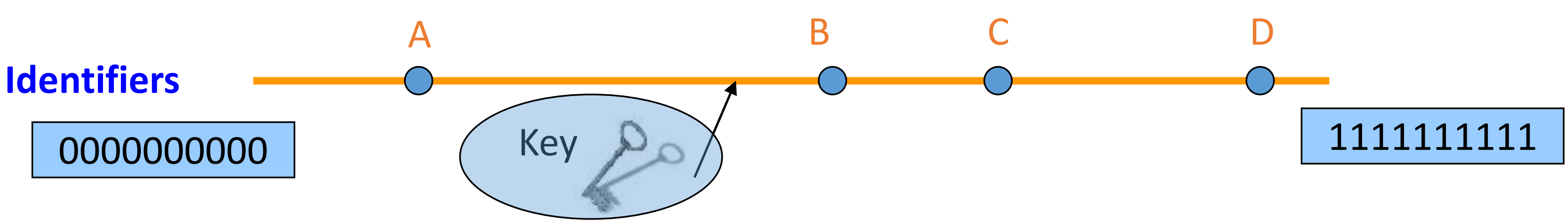
4 Distributed Hash & DHT

Definition of a DHT

- Hash table → supports two operations
  - `insert(key, value)`
  - `value = lookup(key)`
- Distributed
  - Map hash-buckets to nodes
- Requirements
  - Uniform distribution of buckets
  - Cost of `insert` and `lookup` should *scale* well
  - Amount of local state (routing table size) should *scale* well

Fundamental Design Idea - I

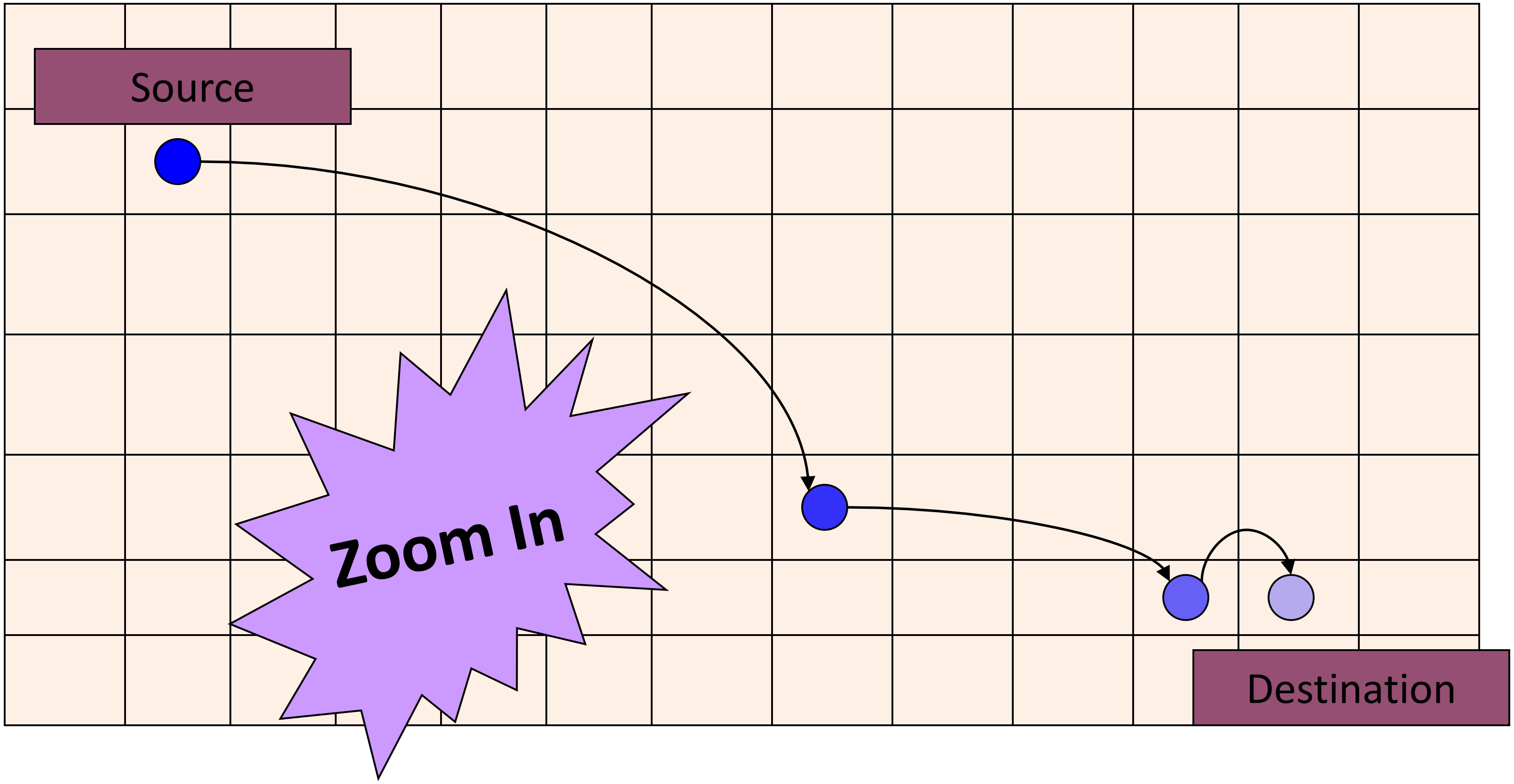
- Consistent Hashing
  - Map keys *and* nodes to an *identifier* space; implicit assignment of responsibility



- Mapping performed using hash functions (e.g., SHA-1)
  - Spread nodes and keys *uniformly* throughout

Fundamental Design Idea - II

- Prefix / Hypercube routing



But, there are so many of them!

- Scalability trade-offs
  - Routing table size at each node vs.
  - Cost of lookup and insert operations
- Simplicity
  - Routing operations
  - Join-leave mechanisms
- Robustness
- DHT Designs
  - Plaxton Trees, Pastry/Tapestry
  - Chord
  - Overview: CAN, Symphony, Koorde, Viceroy, etc.
  - SkipNet

Plaxton Trees Algorithm (1)

1. Assign labels to objects and nodes
   - using randomizing hash functions

9	A	E	4

Object

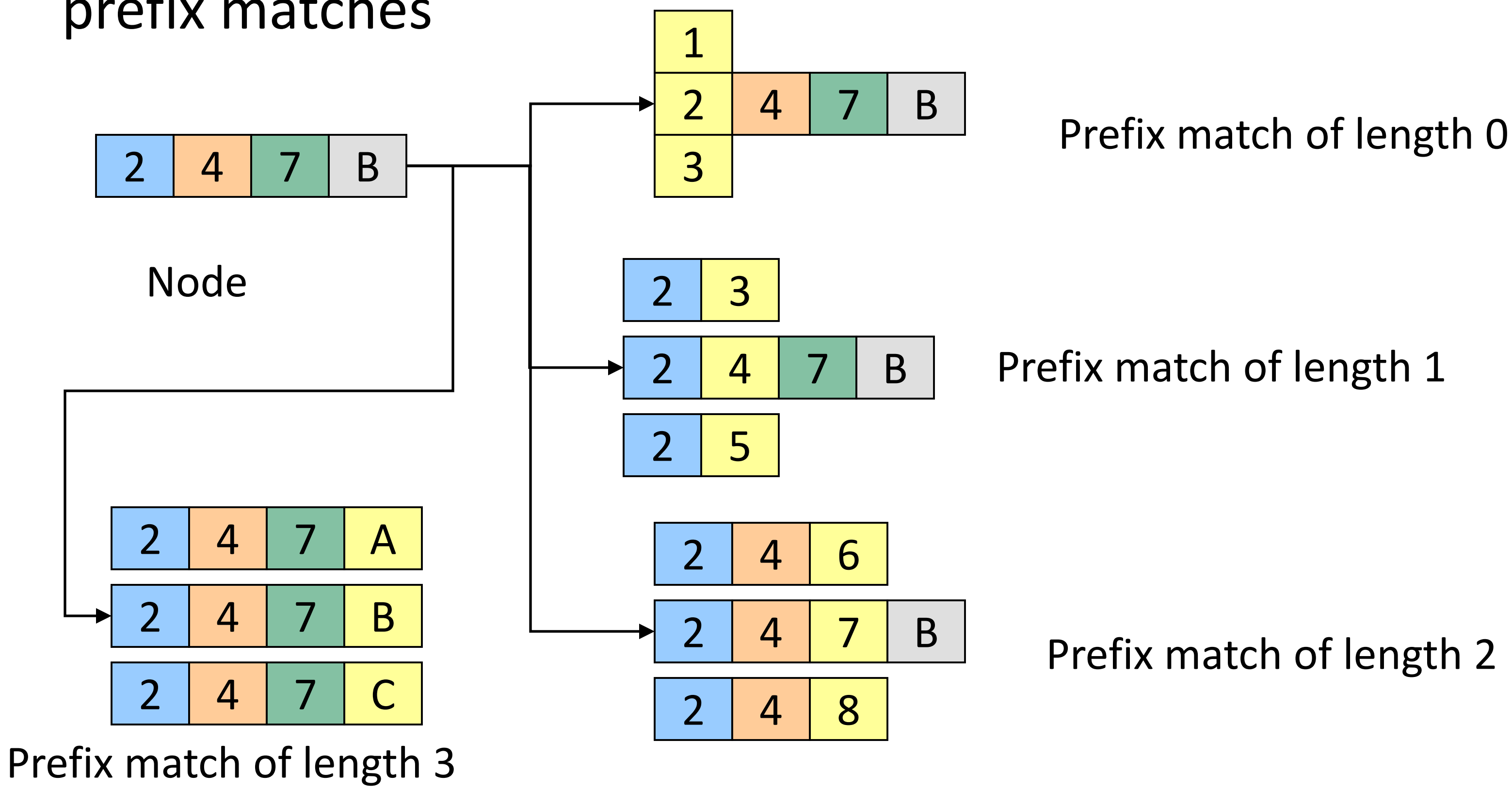
2	4	7	B

Node

Each label is of $\log_2^b n$ digits

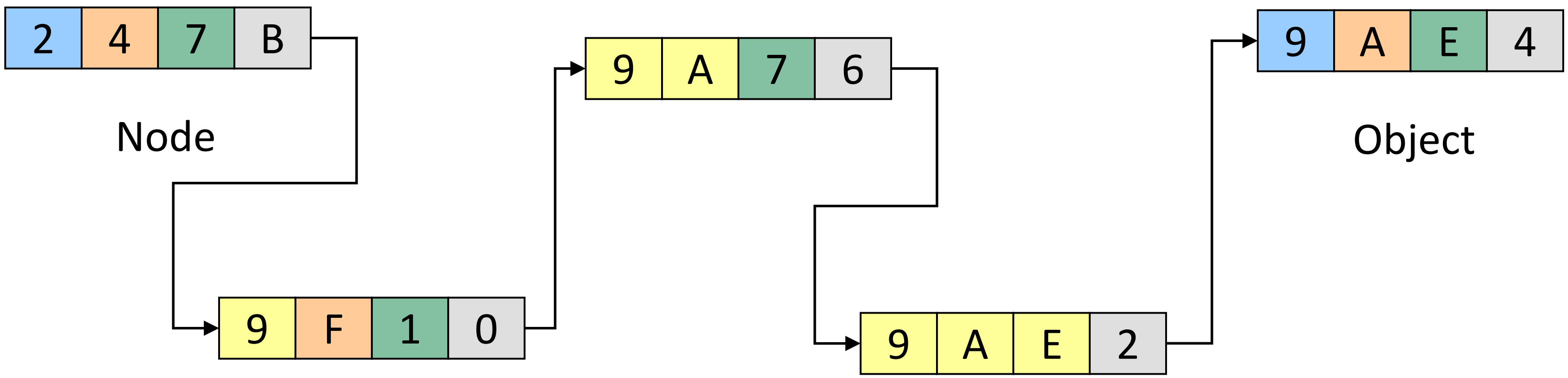
Plaxton Trees Algorithm (2)

2. Each node knows about other nodes with varying prefix matches



Plaxton Trees Algorithm (3) Object Insertion and Lookup

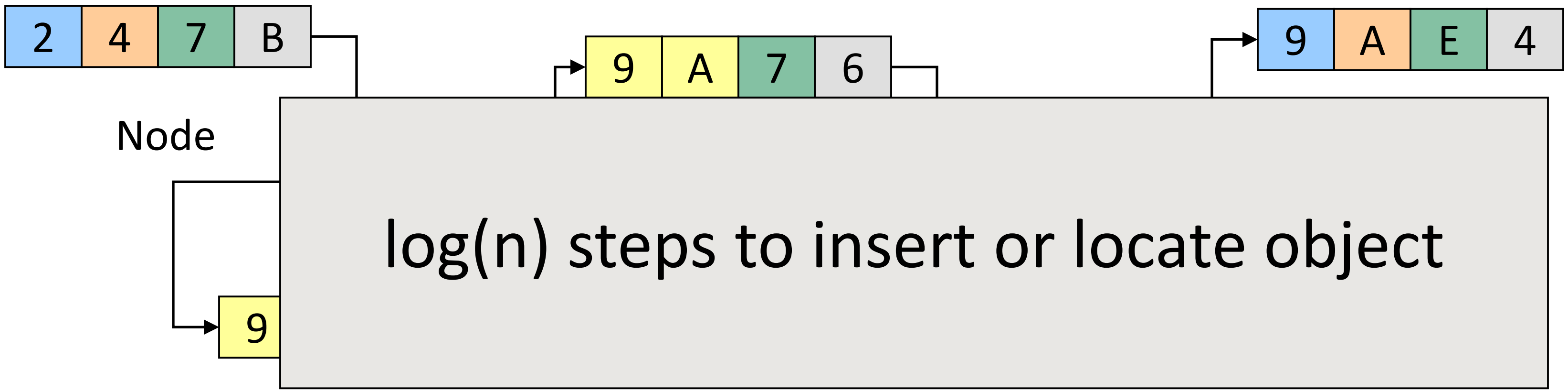
Given an object, route successively towards nodes with greater prefix matches



Store the object at each of these locations

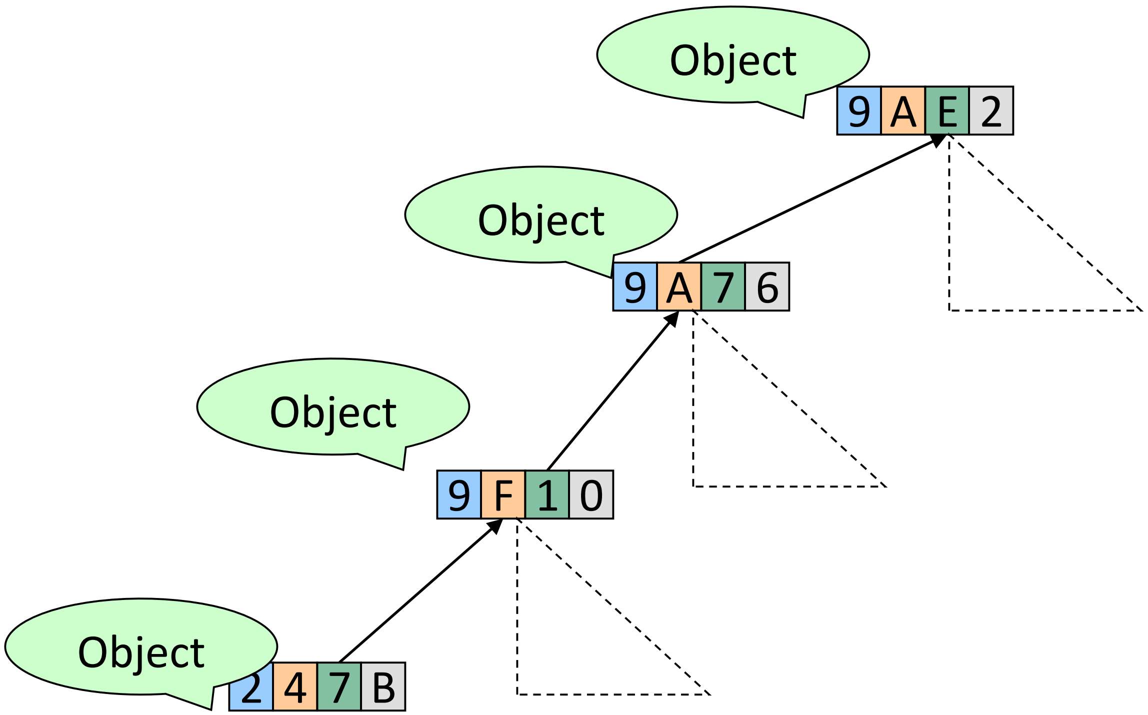
Plaxton Trees Algorithm (4)
Object Insertion and Lookup

Given an object, route successively towards nodes with greater prefix matches



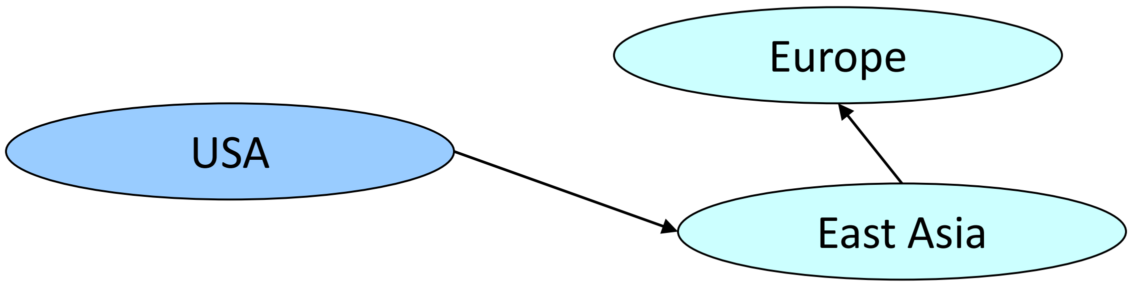
Store the object at each of these locations

Plaxton Trees Algorithm (5)
Why is it a tree?



Plaxton Trees Algorithm (6)
Network Proximity

- Overlay tree hops could be totally unrelated to the underlying network hops

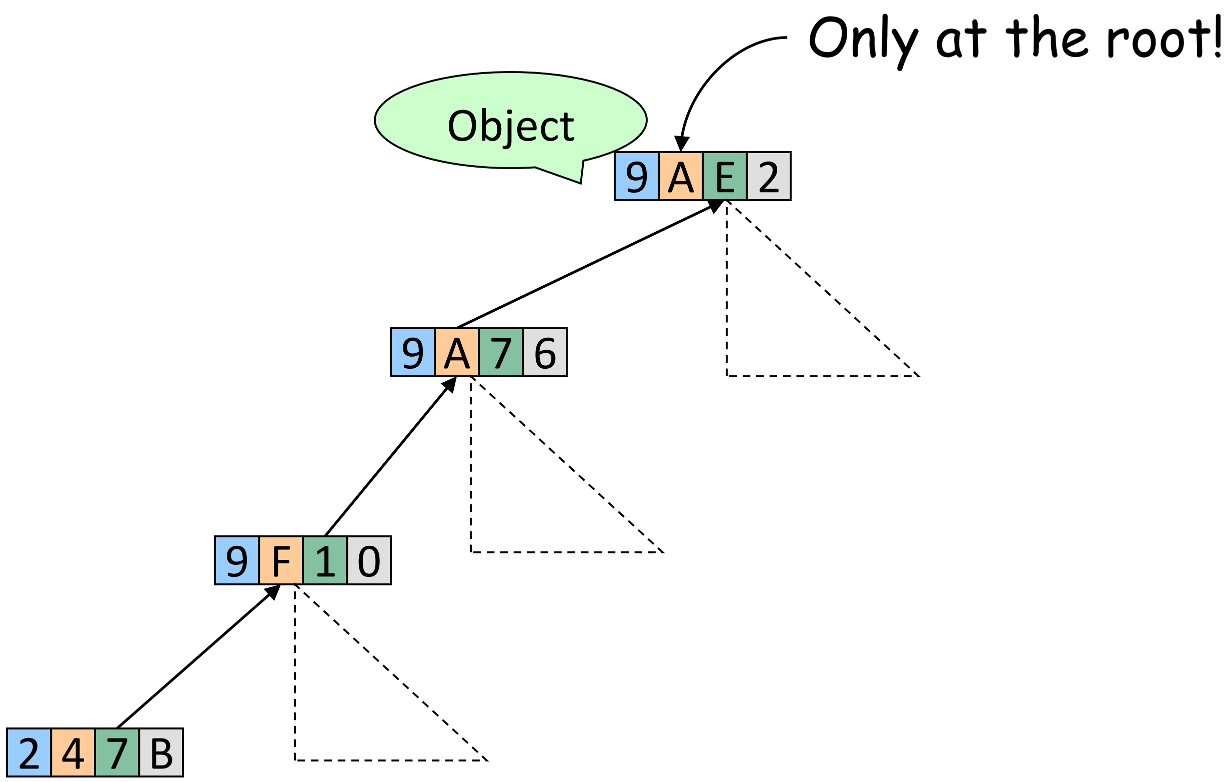


- Plaxton trees guarantee constant factor approximation!
  - Only when the topology is *uniform* in some sense

Pastry (1)

- Based directly upon Plaxton Trees
- Exports a DHT interface
- Stores an object only at a node whose ID is *closest* to the object ID
- In addition to main routing table
  - Maintains *leaf set* of nodes
  - Closest L nodes (in ID space)
    - $L = 2^{(b+1)}$,typically -- one digit to left and right

Pastry (2)



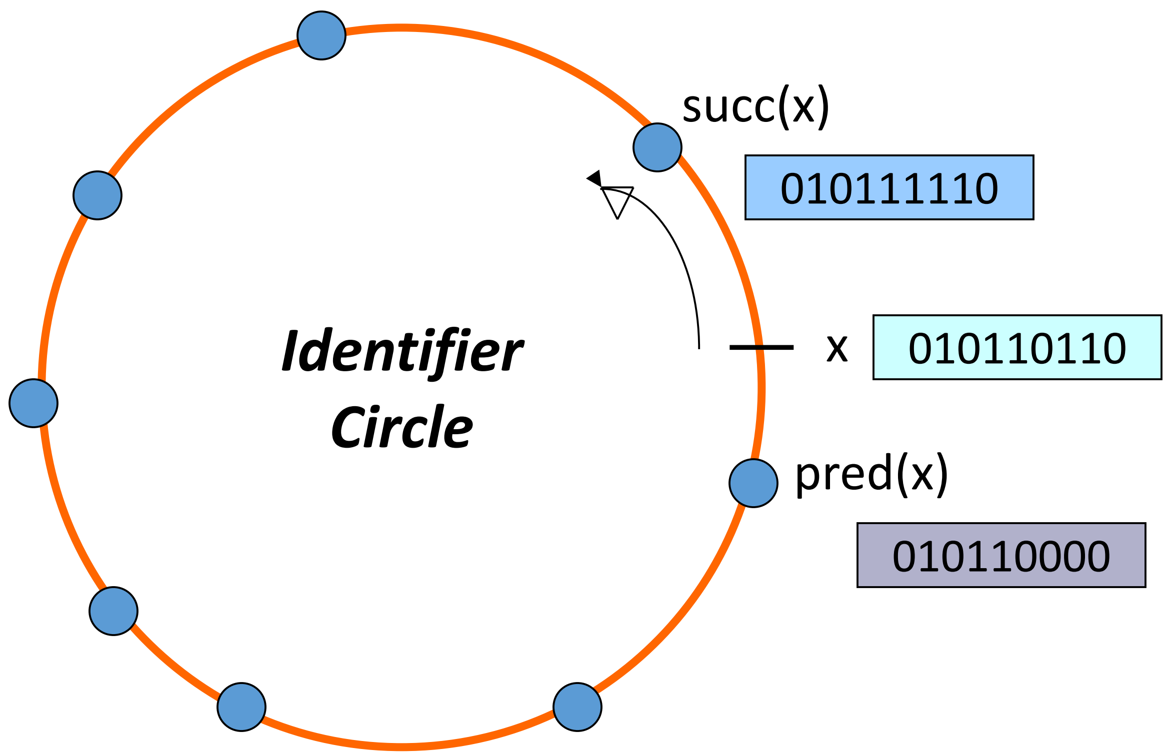
Key Insertion and Lookup = Routing to Root
➔ Takes O(log n) steps

Pastry (3)
Self Organization

- Node join
  - Start with a node "close" to the joining node
  - Route a message to nodeID of new node
  - Take union of routing tables of the nodes on the path
- Joining cost: O(log n)
- Node leave
  - Update routing table
    - Query nearby members in the routing table
  - Update leaf set

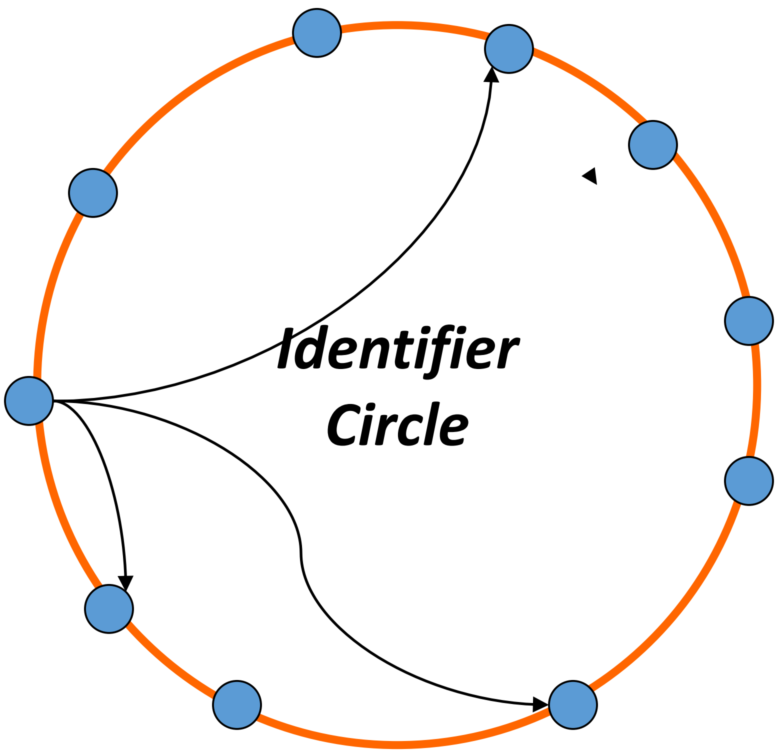
Chord [Karger, et al] (1)

- Map nodes and keys to identifiers
  - Using randomizing hash functions
- Arrange them on a circle



Chord (2)
Efficient routing

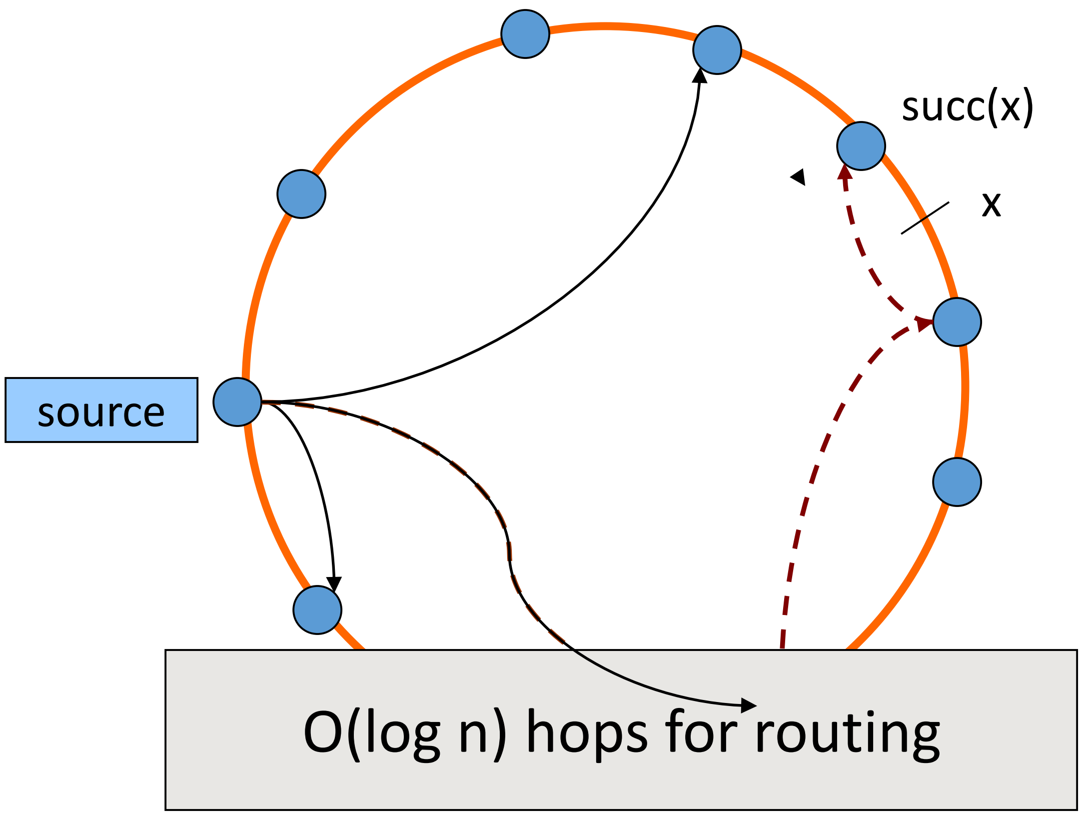
- Routing table
  - i^{th} entry = succ(n + 2^i)
  - log(n) *finger pointers*



Exponentially spaced pointers!

Chord (3)
Key Insertion and Lookup

To insert or lookup a key ‘x’,
route to succ(x)

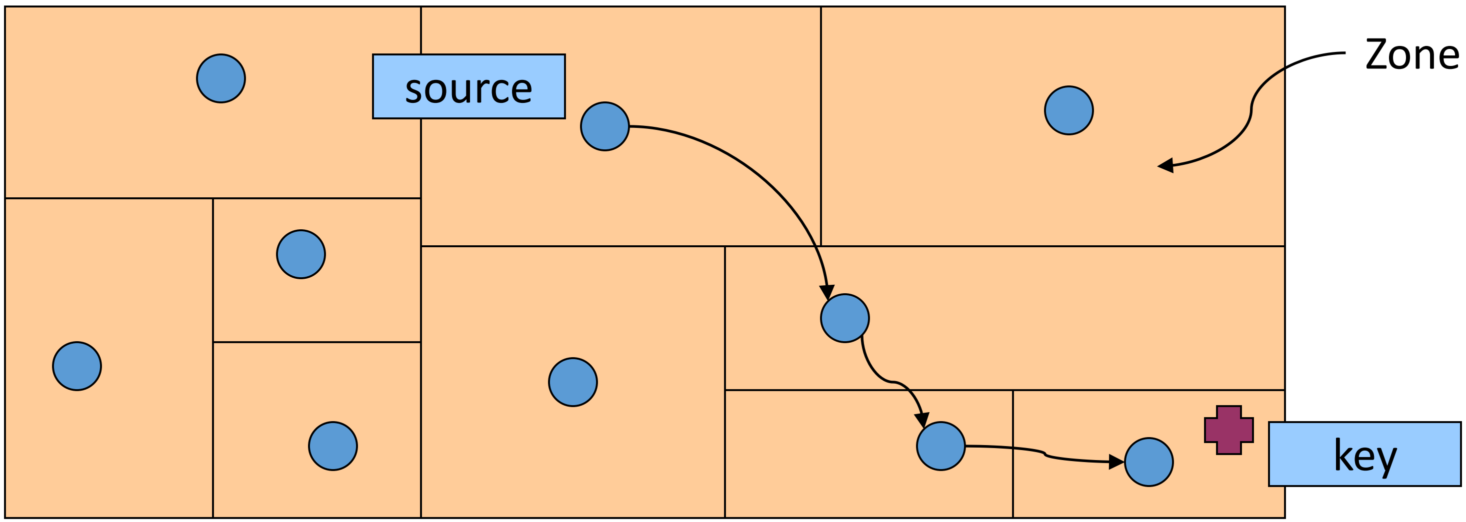


Chord (4)
Self Organization

- Node join
  - Set up finger *i:* route to *succ(n + 2^i^)*
  - log(n) fingers) O(log² n) cost
- Node leave
  - Maintain successor list for ring connectivity
  - Update successor list and finger pointers

CAN [Ratnasamy, et al]

- Map nodes and keys to *coordinates* in a multi-dimensional cartesian space

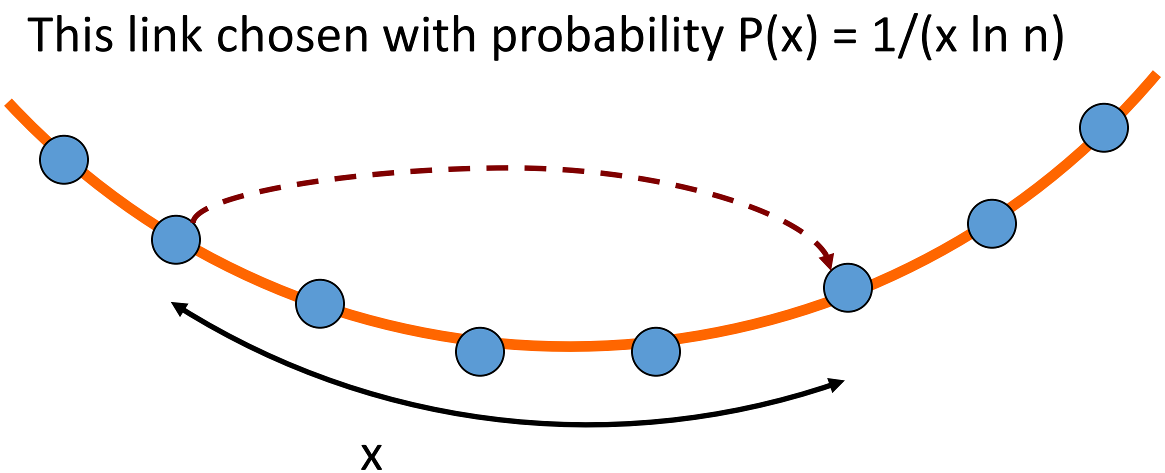


Routing through shortest Euclidean path

For d dimensions, routing takes $O(dn^{1/d})$ hops

Symphony [Manku, et al]

- Similar to Chord – mapping of nodes, keys
  - ‘k’ links are constructed *probabilistically!*



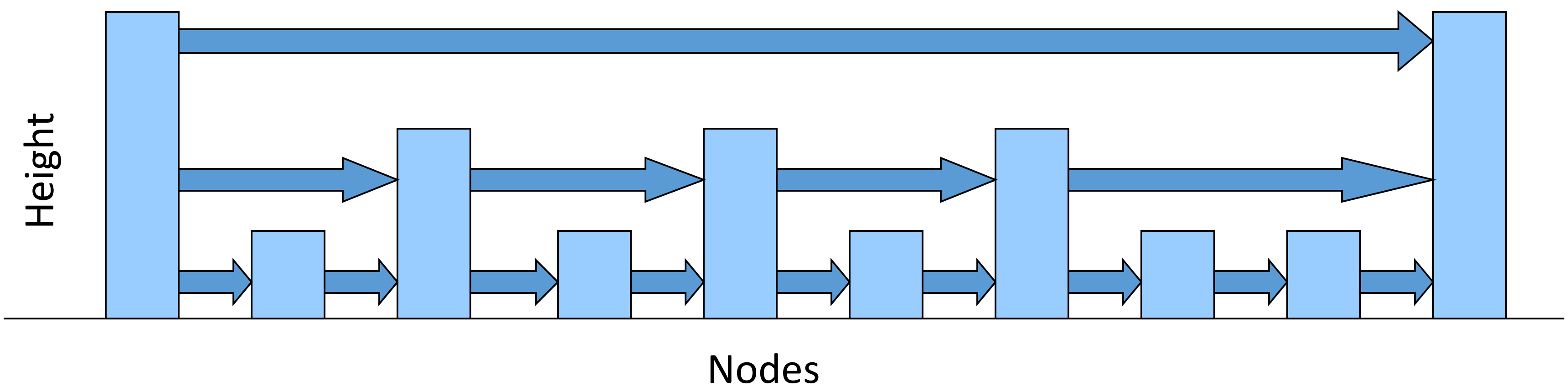
Expected routing guarantee: $O(1/k (\log^2 n))$ hops

SkipNet [Harvey, et al] (1)

- Previous designs distribute data uniformly throughout the system
  - Good for load balancing
  - But, my data can be stored in Timbuktu!
  - Many organizations want stricter control over data placement
  - What about the routing path?
    - Should a Microsoft → Microsoft end-to-end path pass through Sun?

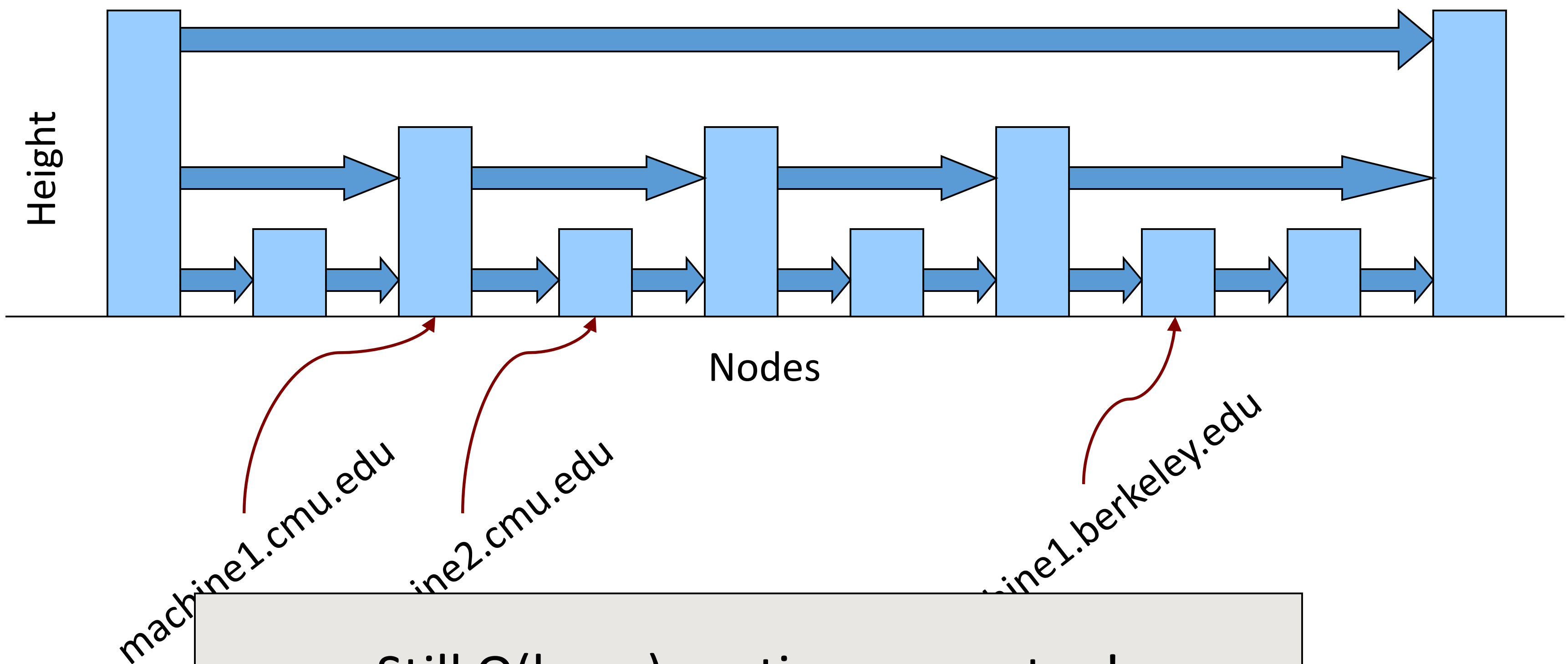
SkipNet (2)
Content and Path Locality

Basic Idea: Probabilistic skip lists



- Each node choose a height at random
  - Choose height ‘h’ with probability $1/2^h$

SkipNet (3)
Content and Path Locality



Still $O(\log n)$ routing guarantee!

- Nodes are *lexicographically sorted*

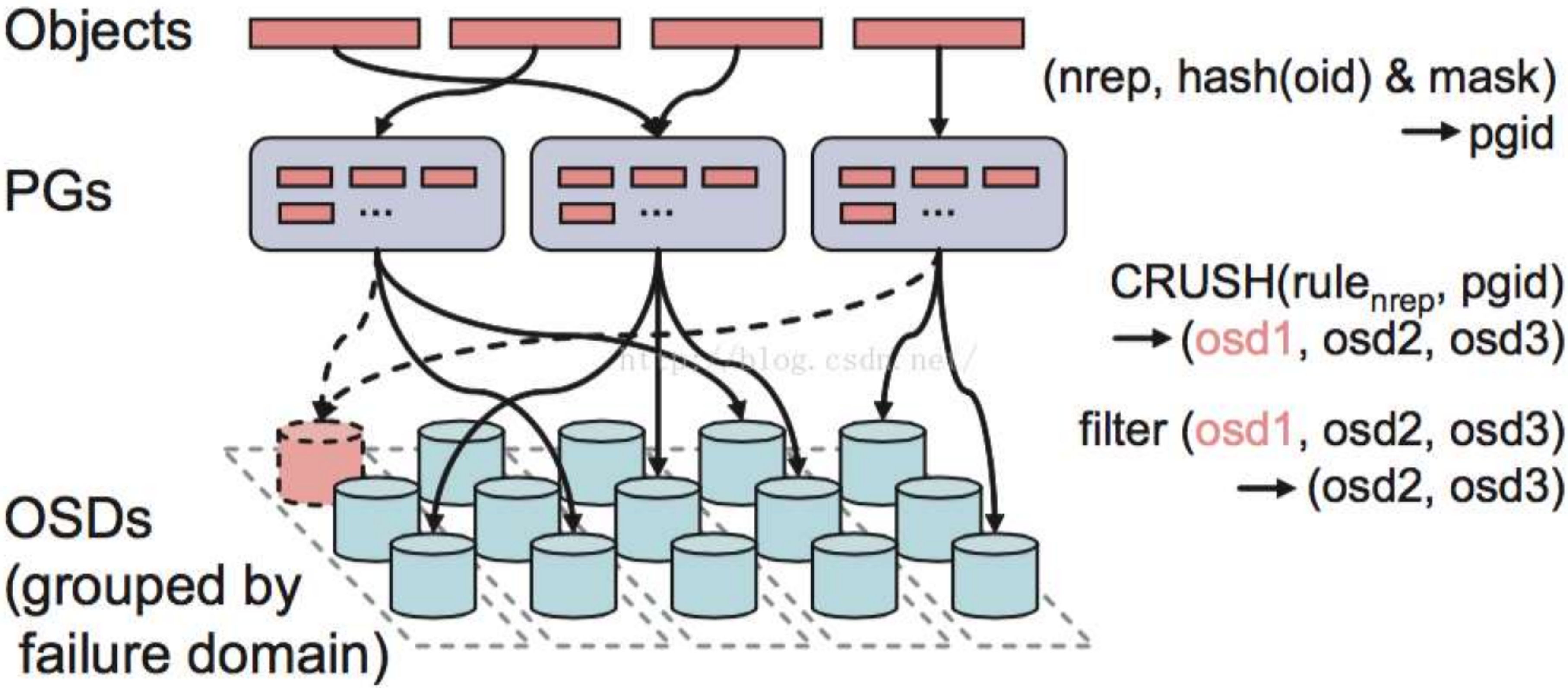
Summary

	# Links per node	Routing hops
Pastry/Tapestry	$O(2^b \log_{2^b} n)$	$O(\log_{2^b} n)$
Chord	$\log n$	$O(\log n)$
CAN	d	$dn^{1/d}$
SkipNet	$O(\log n)$	$O(\log n)$
Symphony	k	$O((1/k) \log^2 n)$
Koorde	d	$\log_d n$
Viceroy	7	$O(\log n)$

Optimal (= lower bound)

Ceph Controlled Replication Under Scalable Hashing (CRUSH) (1)

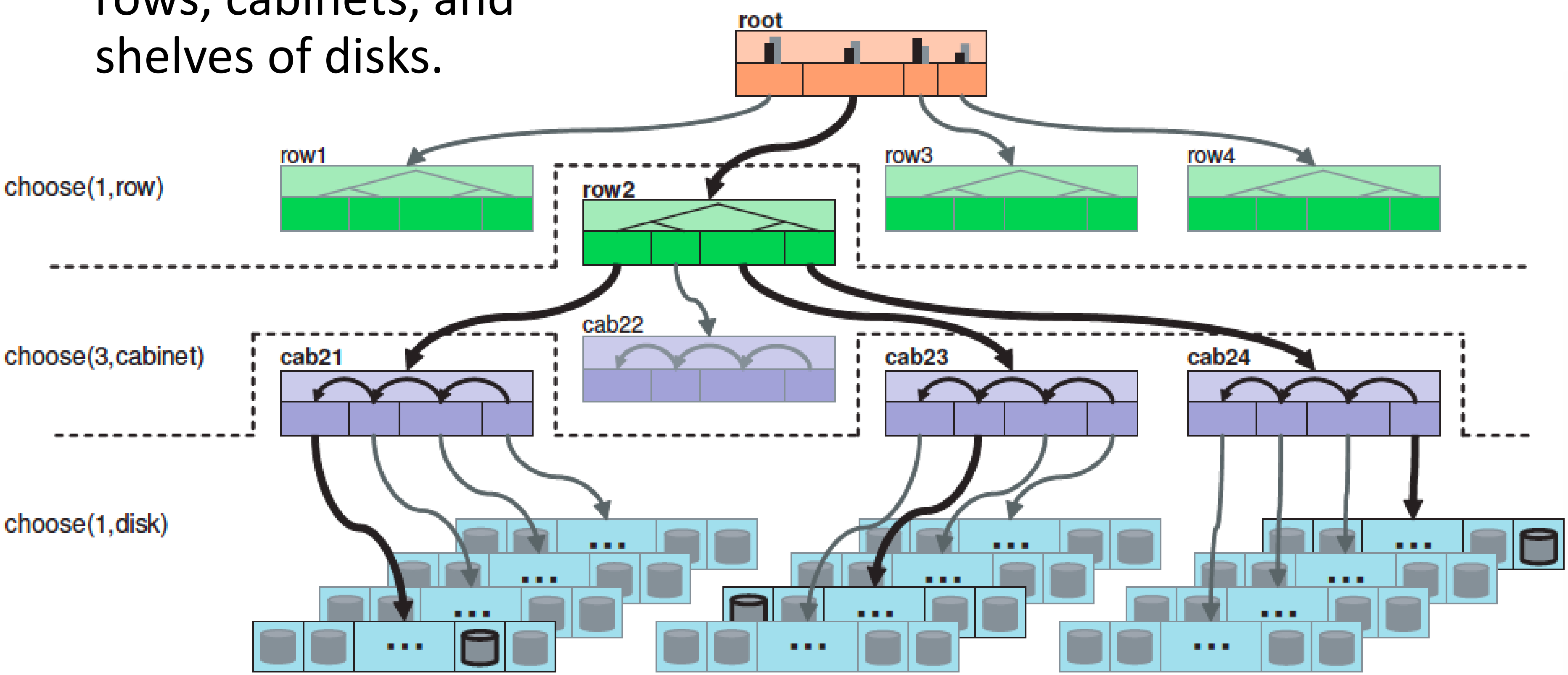
- CRUSH algorithm: pgid → OSD ID?
- Devices: leaf nodes (weighted)
- Buckets: non-leaf nodes (weighted, contain any number of devices/buckets)



CRUSH (2)

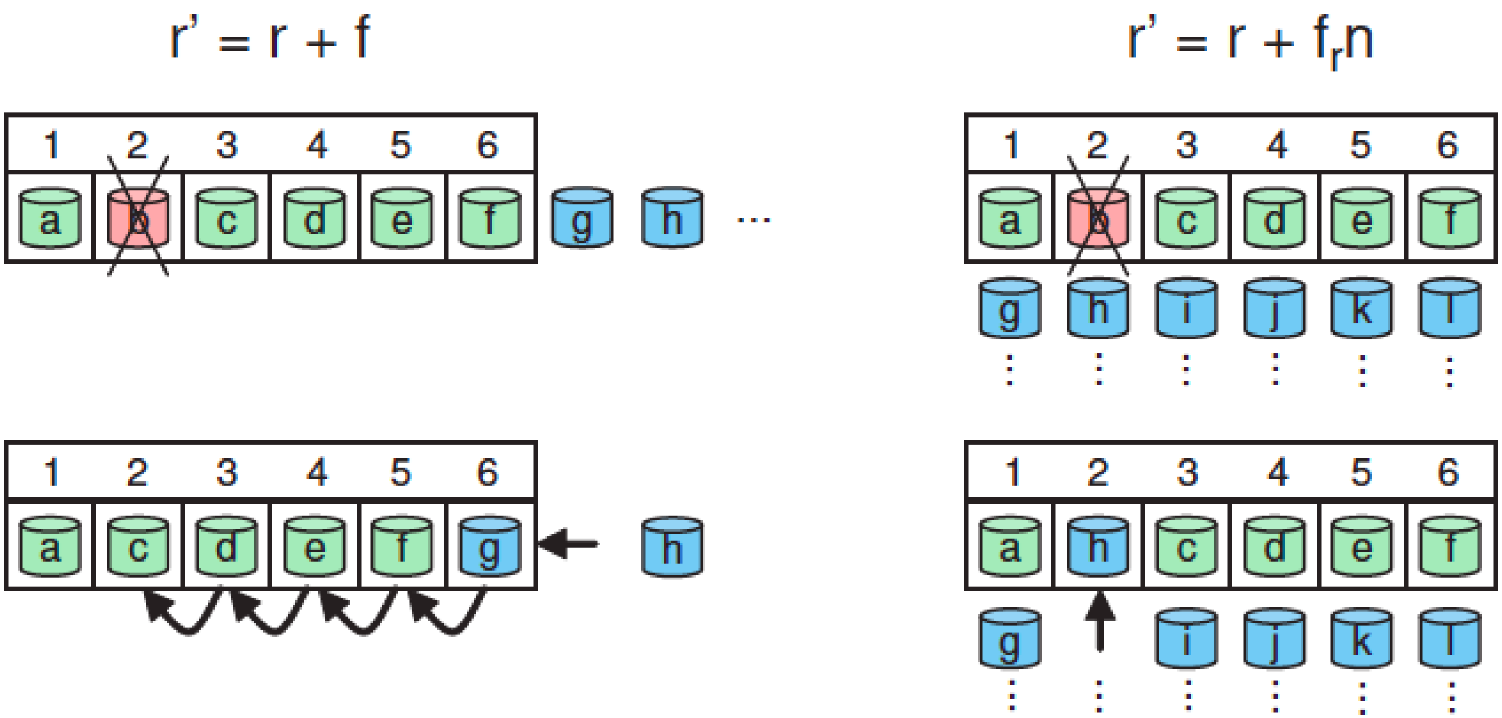
- A partial view of a four-level cluster map hierarchy consisting of rows, cabinets, and shelves of disks.

Action	Resulting $\vec{i}$
take(root)	root
select(1,row)	row2
select(3,cabinet)	cab21 cab23 cab24
select(1,disk)	disk2107 disk2313 disk2437
emit	



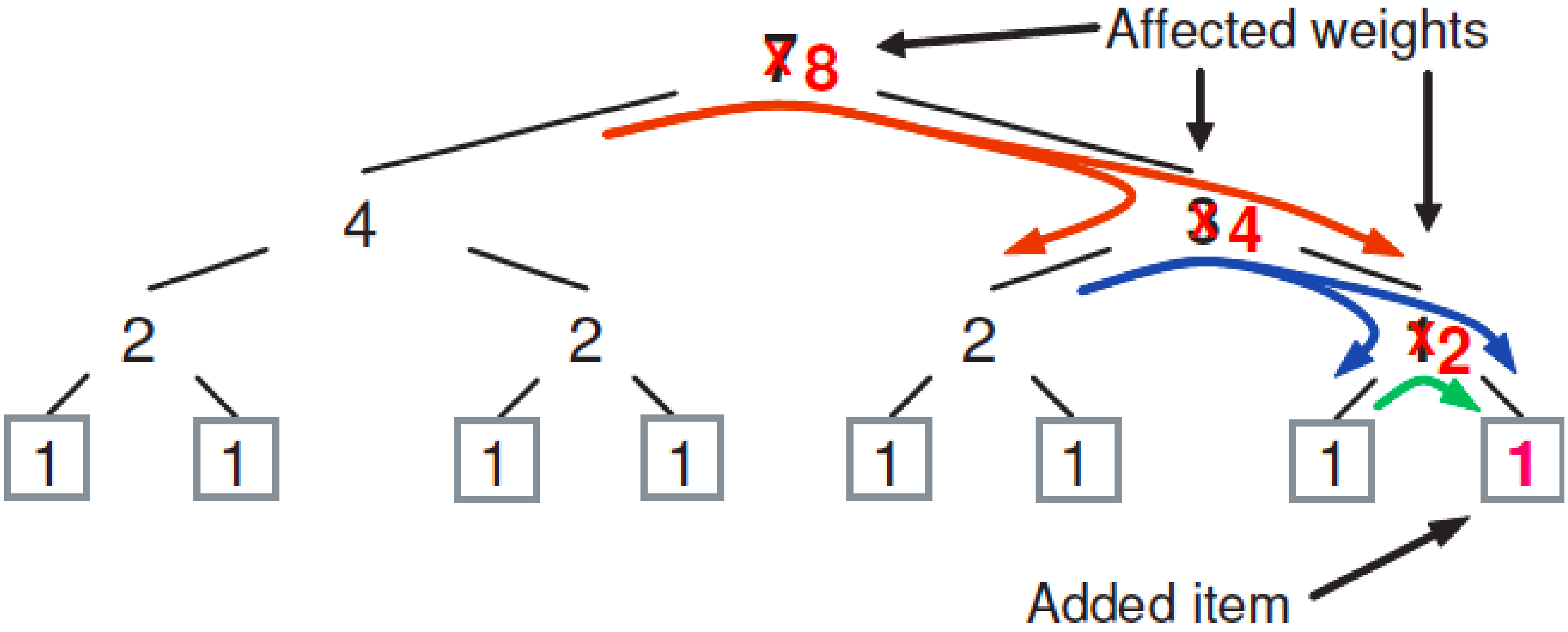
CRUSH (3)

- Reselection behavior of select(6,disk) when device r = 2 (b) is rejected, where the boxes contain the CRUSH output R of n = 6 devices numbered by rank. The left shows the "first n" approach in which device ranks of existing devices (c,d,e,f) may shift. On the right, each rank has a probabilistically independent sequence of potential targets; here $f_r = 1$, and $r' = r + f_r n = 8$ (device h).



CRUSH (4)

- Data movement in a binary hierarchy due to a node addition and the subsequent weight changes.



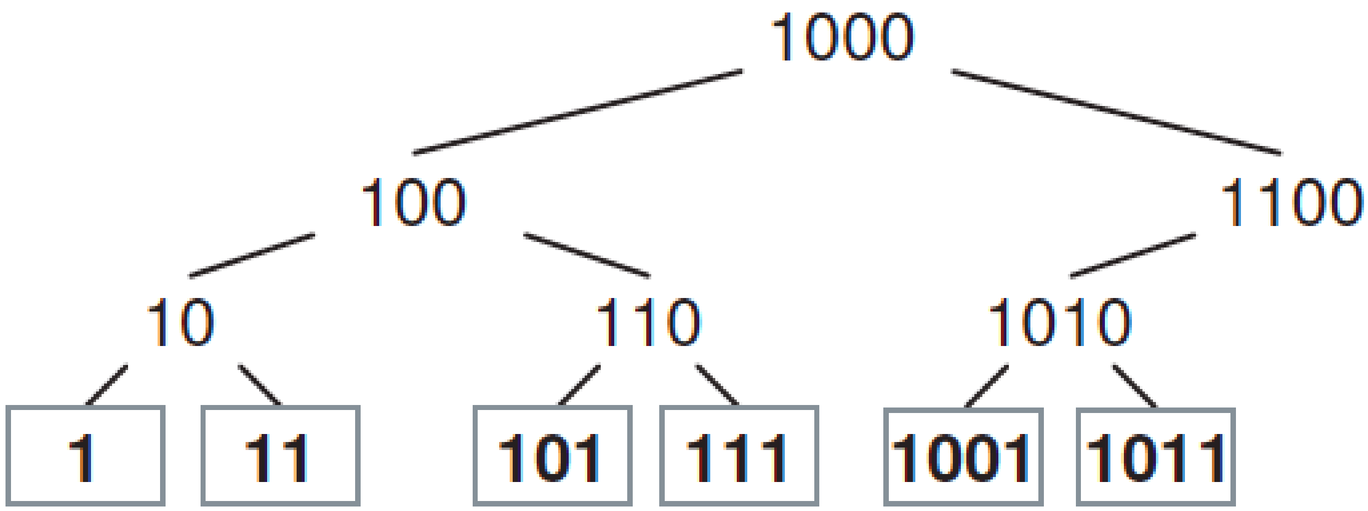
CRUSH (5)

- Four types of Buckets
  - Uniform buckets
  - List buckets
  - Tree buckets
  - Straw buckets
- Summary of mapping speed and data reorganization efficiency of different bucket types when items are added to or removed from a bucket.

Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

CRUSH (6)

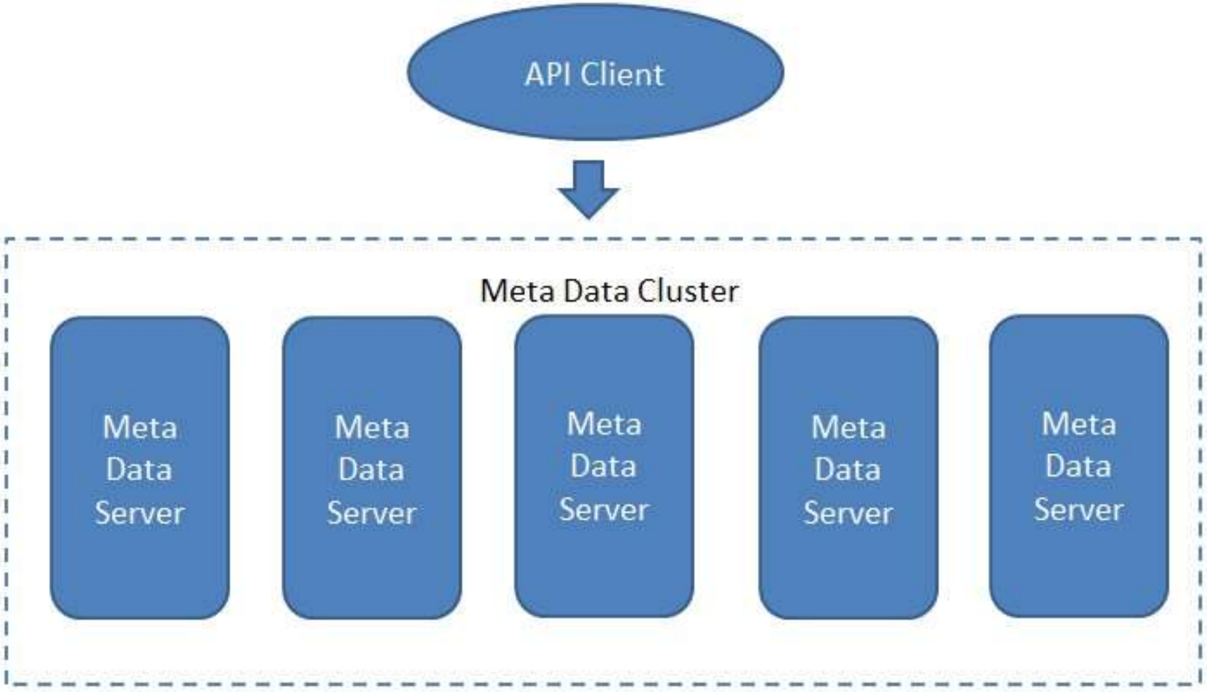
- Node labeling strategy used for the binary tree comprising each tree bucket



5 Project 4

Metadata Management in DFS (1)

- Design a simple metadata management module for a distributed file system. Establish a distributed metadata cluster and a POSIX API based client.



Metadata Management in DFS (2)

- The metadata management has the following functions,
  - Basic command set: support metadata operations via POSIX-based API
    - i.e., mkdir, create file, readdir, rm file, stat, etc.
    - file handle can be ignored
  - Distribution of metadata
    - Metadata are distributed among various metadata servers

Metadata Management in DFS (3)

- Tests on the metadata management functions,
  - Input: Input the specified files & directories by client
  - Output:
    - Traverse the files via readdir command
    - List the status of a file via stat command
    - Etc.
  - Write the metadata of these file operations into the metadata server
    - Give the data distribution information of the whole cluster
    - Consistent with other metadata servers

Metadata Management in DFS (4)

- Additional scores
  - Support metadata server failover (process level)
  - Support metadata server failure
    - No metadata lost in the failure
  - Implementation on the read/write operations of a file

Thank you!