Unit #7: B^+-Trees

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Unit Outline

- Minimizing disk I/Os
- B^+-Tree properties
- Implementing B^+-Tree insert and delete
- Some final thoughts on B^+-Trees

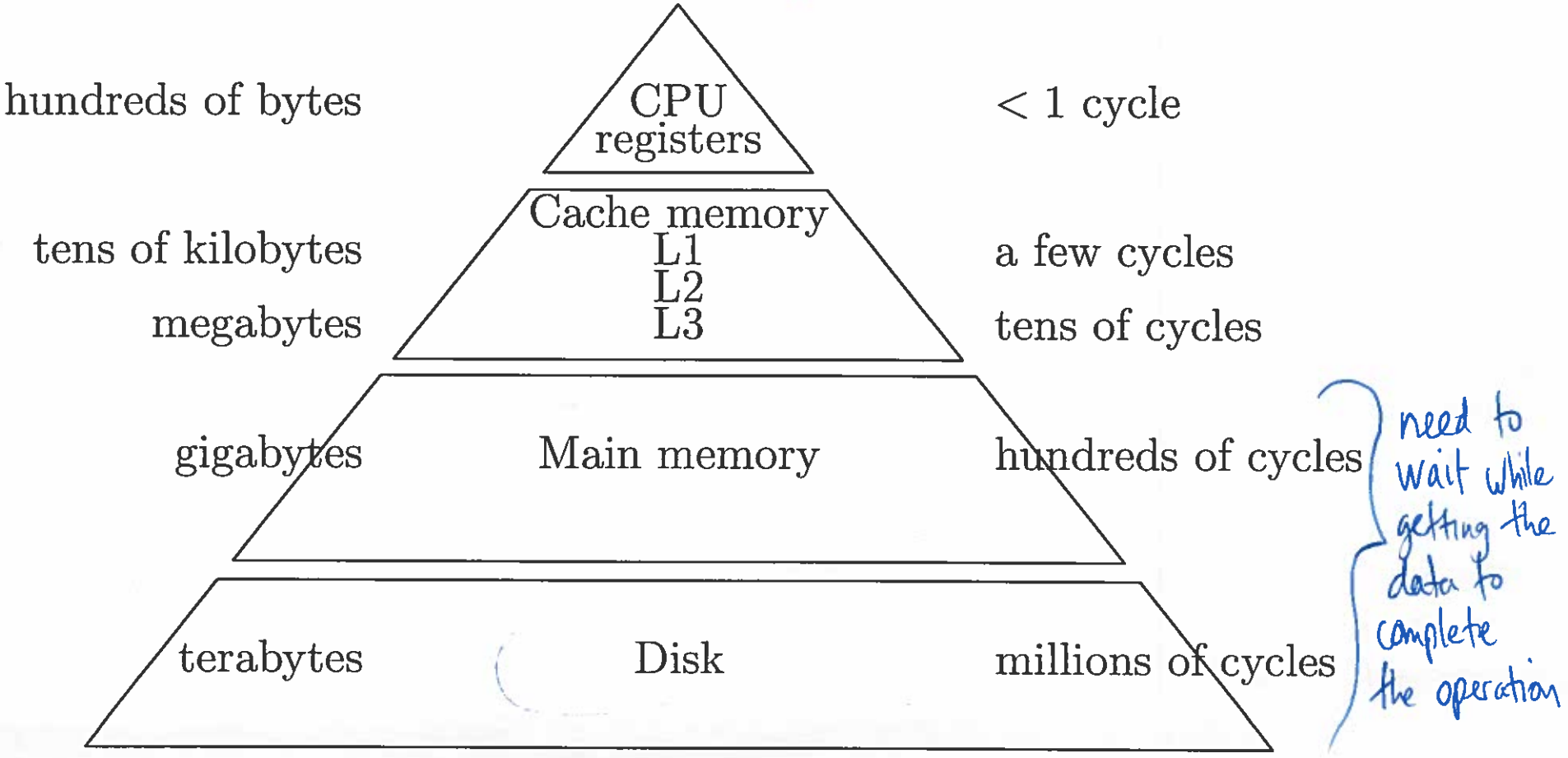
Learning Goals

- Describe the structure, navigation and time complexity of a B^+-Tree.
- Insert and delete keys from a B^+-Tree.
- Relate M, L, the number of nodes, and the height of a B^+-Tree.
- Compare and contrast B^+-Trees with other data structures.
- Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of CPU operations) when dealing with large datasets and their indexing structures (e.g., B^+-Trees).
- Explain the difference between a B-Tree and a B^+-Tree

Memory Hierarchy

Can be much more expensive (time) than many operations/cycles.

Why worry about the number of disk I/Os?



Time Cost: Processor to Disk

Processor

3.0-4

- Operates at a few GHz (gigahertz = billion cycles per second)
- Several instructions per cycle
- Average time per instruction < 1 ns (1 nanosecond $= 10^{-9}$ seconds)

0.000000001 seconds

Disk (HDD = Hard (spinning) Disk Drive)

finding the data

after there is a transfer time

- Seek time $\approx$ 10 ms (1 millisecond $= 10^{-3}$ seconds)
- Note: Solid State Drives (SSDs) have "seek time" $\approx$ 0.03 ms

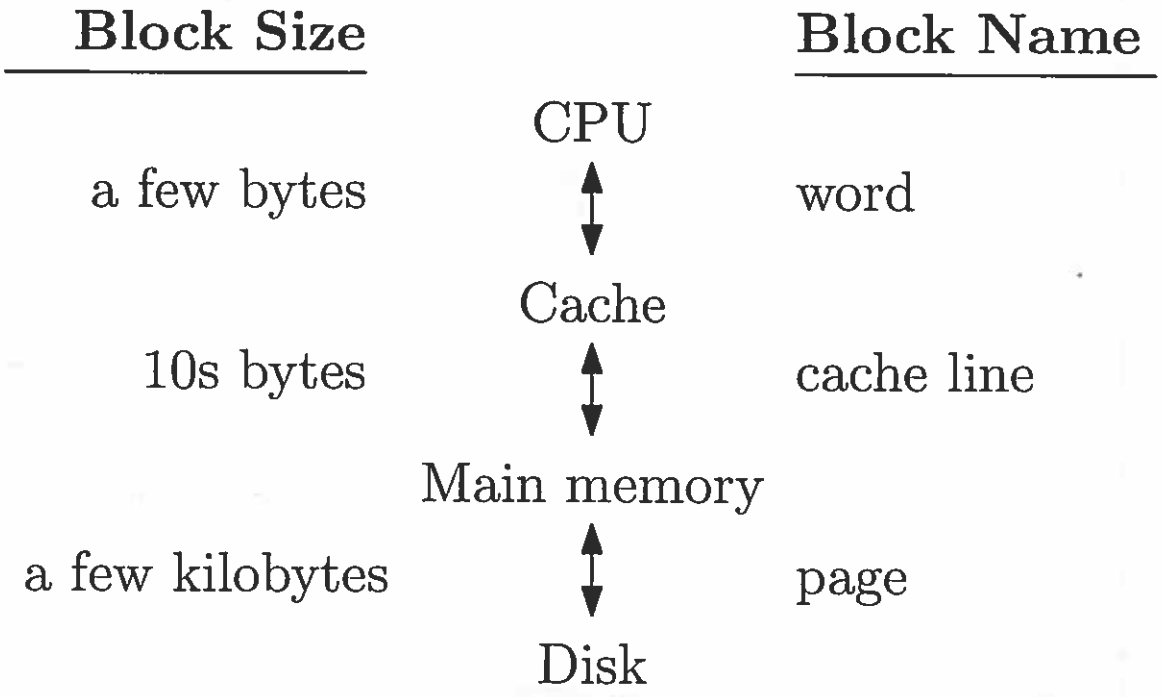
Result: $\approx$ 10 million instructions for each disk read!

instead of reading just 1 integer at a time usually whole "blocks" of data are read at a single time, to minimize our disk I/O.

Hold on... How long does it take to read a 1 TB (1 terabyte $= 10^{12}$ bytes) disk? 1 TB $\times$ 10 ms $=$ 10 billion seconds $>$ 300 years? What's wrong? Each disk read/write moves more than a byte (e.g., 4 KB, 8 KB, ... block sizes). Continuous HDD disk access is about the same speed as on an SSD.

Memory Blocks

Each memory access to a slower level of the hierarchy fetches a block of data.



A block is the contents of consecutive memory locations.

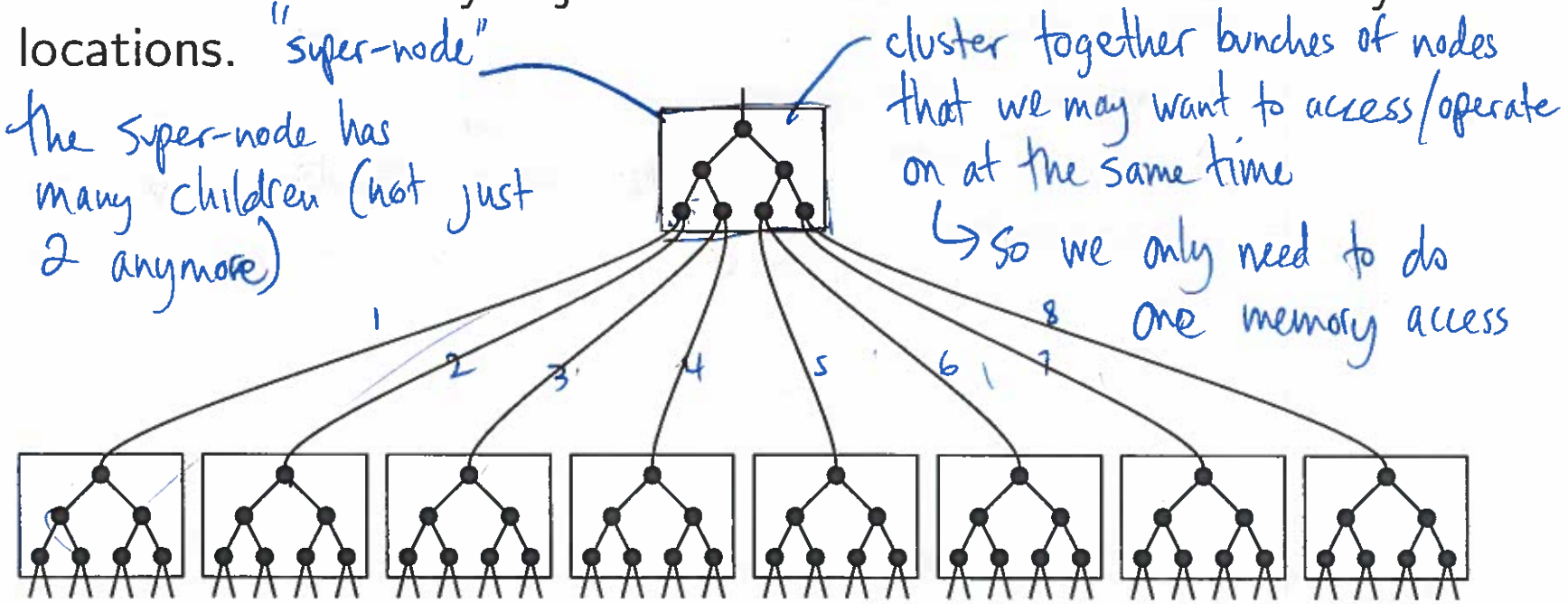
So random access between levels of the hierarchy is very slow.

We want all of data needed to solve the problem to be together in memory (in consecutive memory locations) to minimize the number of disk operations

Chopping Trees into Blocks

Idea

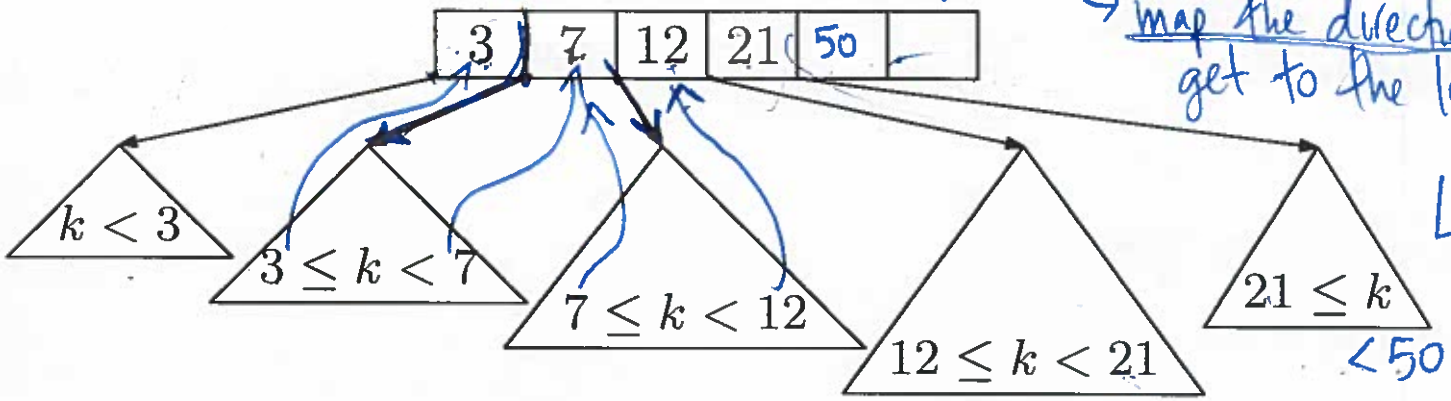
Store data for many adjacent nodes in consecutive memory locations.



Result

One memory block access provides keys to determine many (more than two) search directions.

M-ary Search Tree



"super-node"/cluster
↳ every internal node in a Bt-tree
↳ map the directions on how to get to the leaf-nodes.

Leaf-nodes are where all the data is stored.

- M-ary tree property
  - Each node has $\leq M$ children.

Result: Complete M-ary tree with n nodes has height $\Theta(\log_M n)$

Search tree property

- Each node has $\leq M - 1$ search keys: $k_1 < k_2 < k_3 \ldots$
- All keys k in ith subtree obey $k_i \leq k < k_{i+1}$ for $i = 0, 1, \ldots$.

Disk I/O's (runtime) for `find`:

A) $O(\log_m n)$ * → depends on keeping tree balanced
B) $O(\log n)$
C) $O(n)$

B$^+$-Trees

B$^+$-Trees of order M are specialized M-ary search trees:

ensure balance → shallow a tree as possible

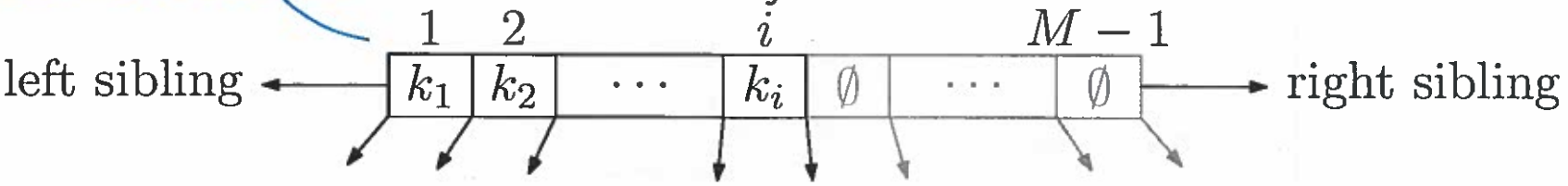
- ALL leaves are at the same depth!
- Internal nodes have between $\lceil M/2 \rceil$ and M children.
- Keys and values are stored only in the leaves. Search keys in internal nodes only direct traffic. B-Trees store (key, value) pairs at internal nodes. → more keys in interior nodes → higher branching factor → shallower tree
- Leaves hold between $\lceil L/2 \rceil$ and L (key, value) pairs.
- The root is special. If it is an internal page, it has between 2 and M children. If it is a leaf page, it holds at most L (key, value) pairs.

Result

- Height is $\Theta(\log_M n)$ — M is significant because it represents I/O (disk accesses) which are the major bottleneck.
- `insert`, `delete`, and `find` operations visit $\Theta(\log_M n)$ nodes.
- M and L are chosen so that each (full) node fills one page of memory. Each node visit (e.g., disk I/O operation) retrieves about $M/2$ to M keys or $L/2$ to L (key, value) pairs at a time.

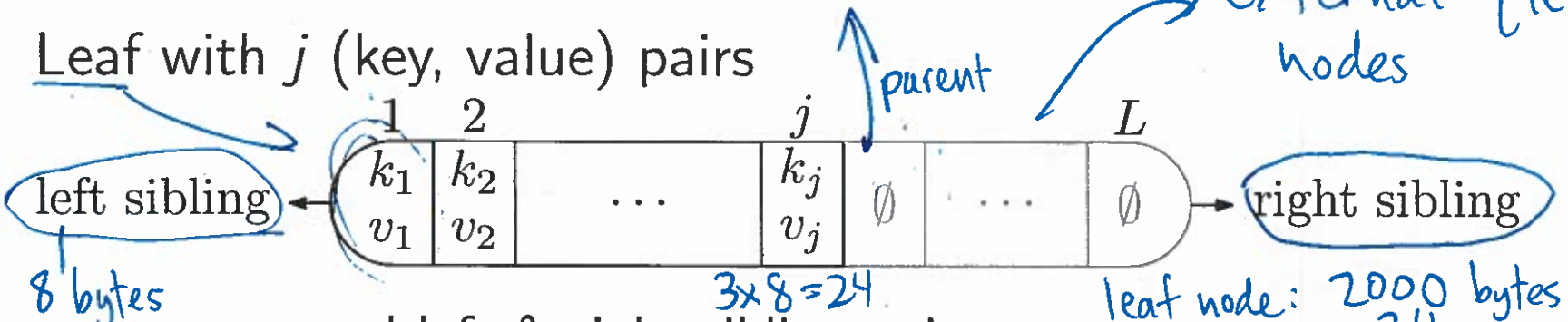
B$^+$-Tree Nodes

Internal node with i search keys



- $i + 1$ subtree pointers
- parent and left & right sibling pointers

Leaf with j (key, value) pairs



external (leaf) nodes

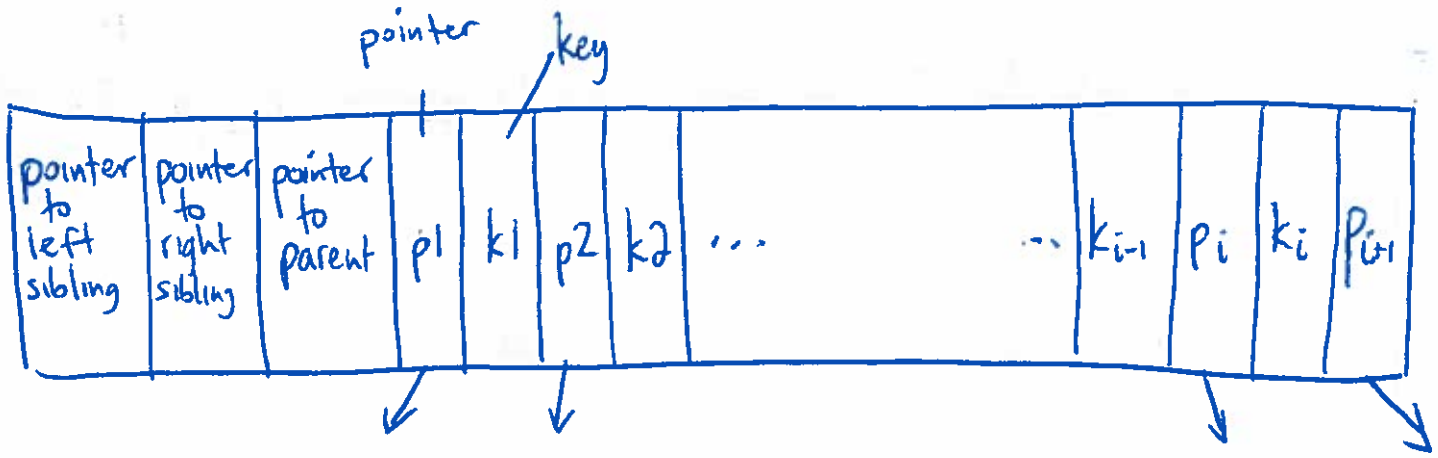
- parent and left & right sibling pointers
- values may be pointers to disk records

leaf node: 2000 bytes.
− 24
1976 bytes
size of k,v pair

Each node may hold a different number of items.

- leaf nodes contain key, value pairs
- internal nodes direct traffic and only contain keys.

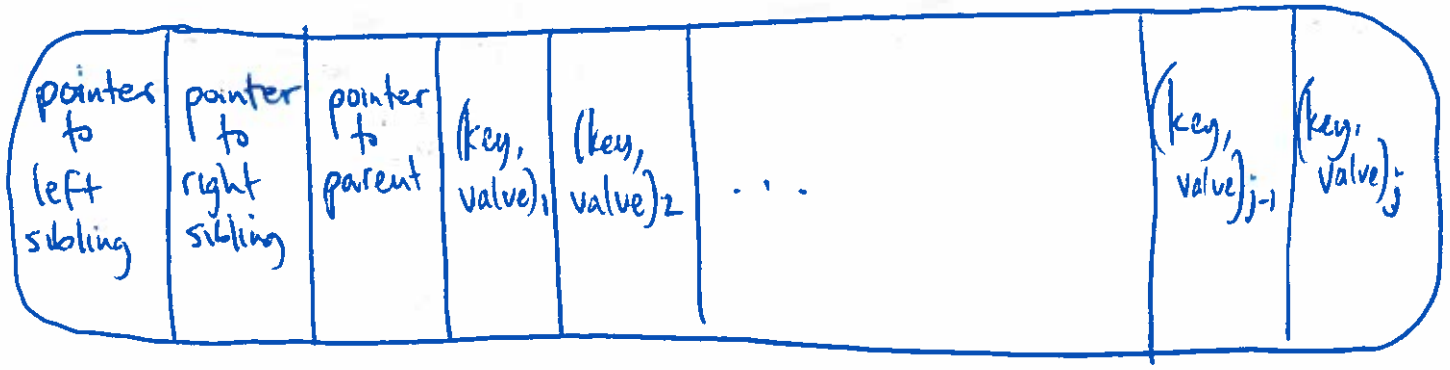
Internal node with i search keys:



~~if it~~

if an internal node contains n keys, it has $n+1$ children

Leaf node with j (key, value) pairs



Example B$^+$-Tree with $M = 4$ and $L = 4$

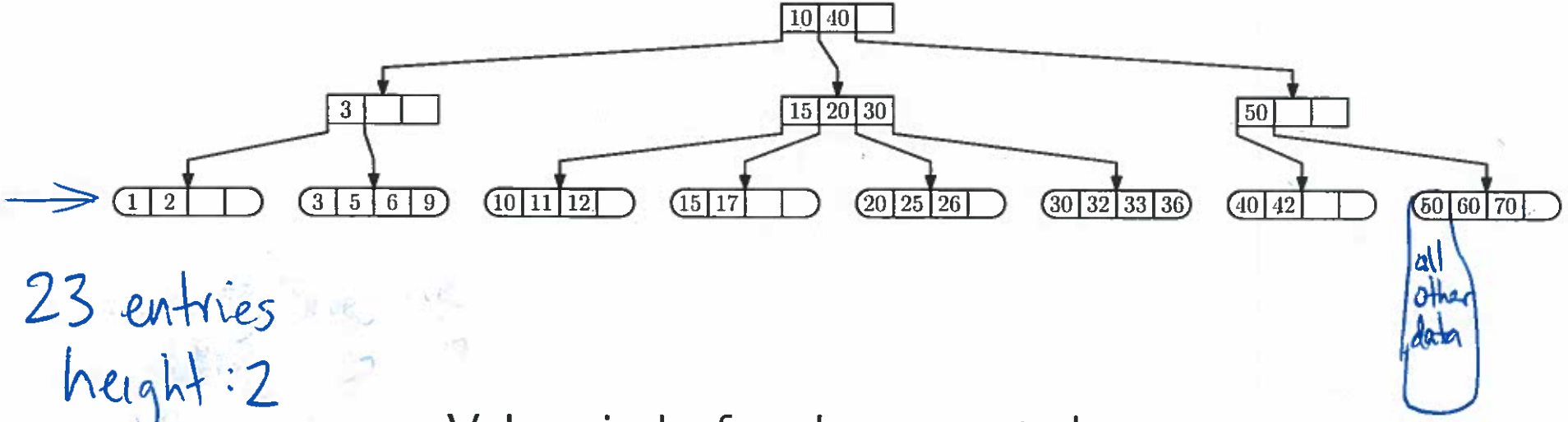
max # of k,v pairs in each leaf node

$2 \leq$ # children ≤ 4

$1 \leq$ keys ≤ 3

(max # of children for internal nodes)

these are very small values

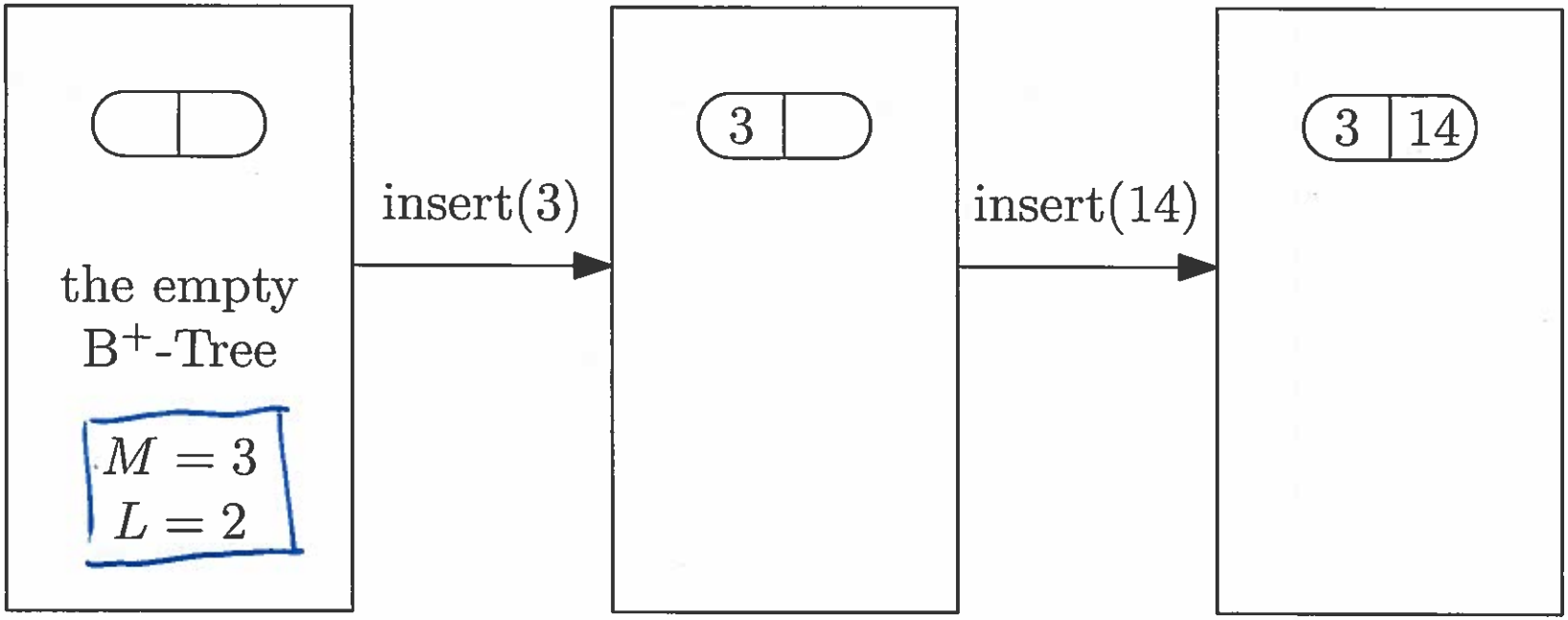


23 entries
height: 2

Values in leaf nodes are not shown.

With 23 entries, what is the minimum height of a BST? 4

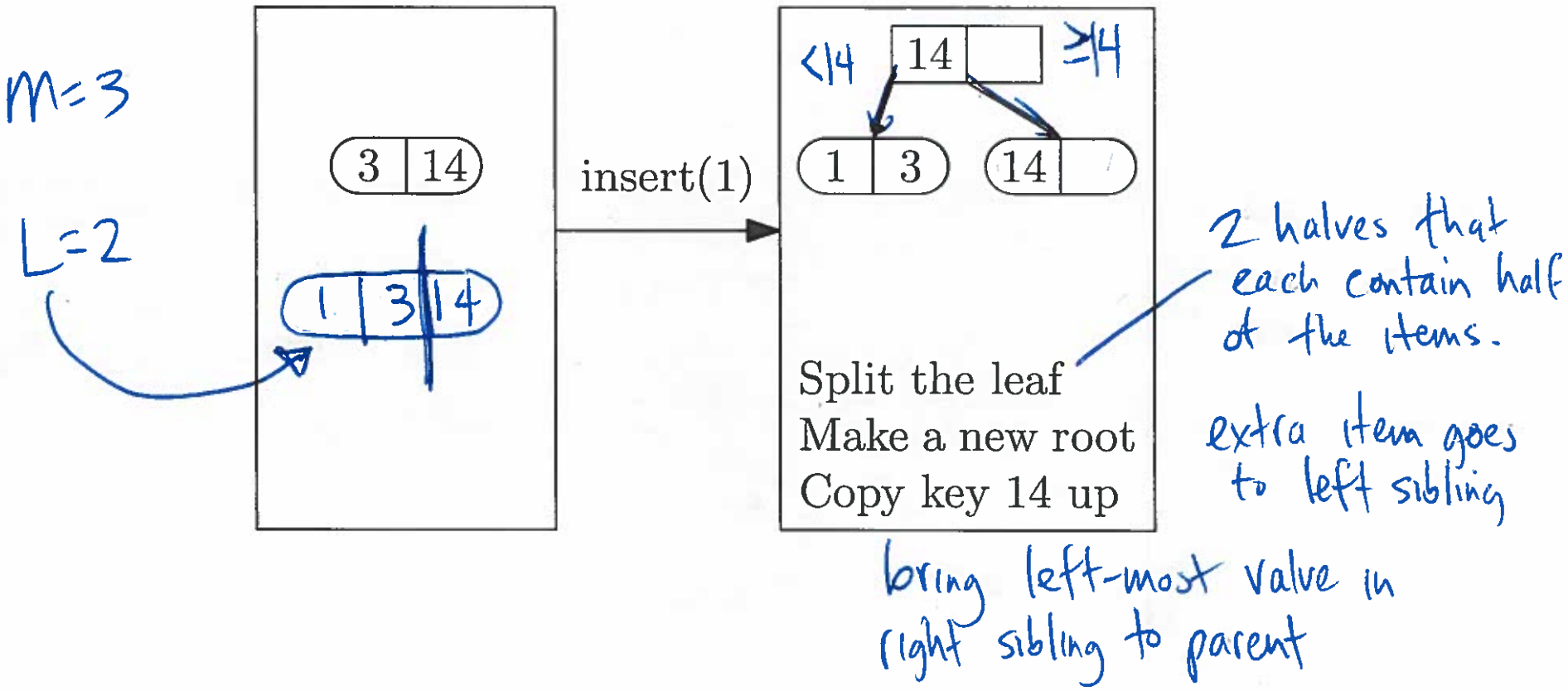
Making a B$^+$-Tree



The root is a leaf.

What happens when we now insert(1)?

Splitting the Root

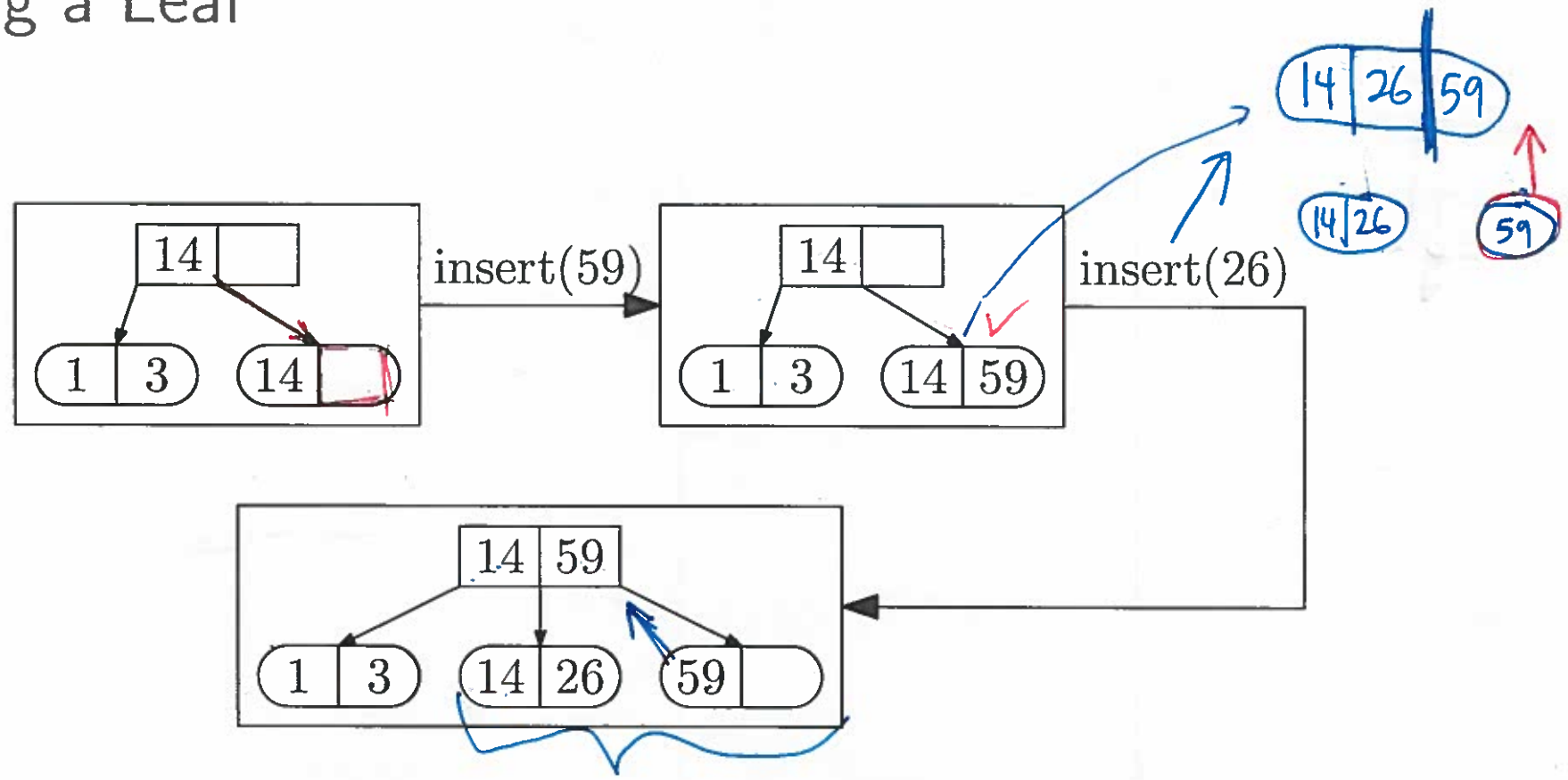


Too many keys for one leaf!

So, make a new leaf and create a parent (the new root) for both.

Why is key 14 duplicated?

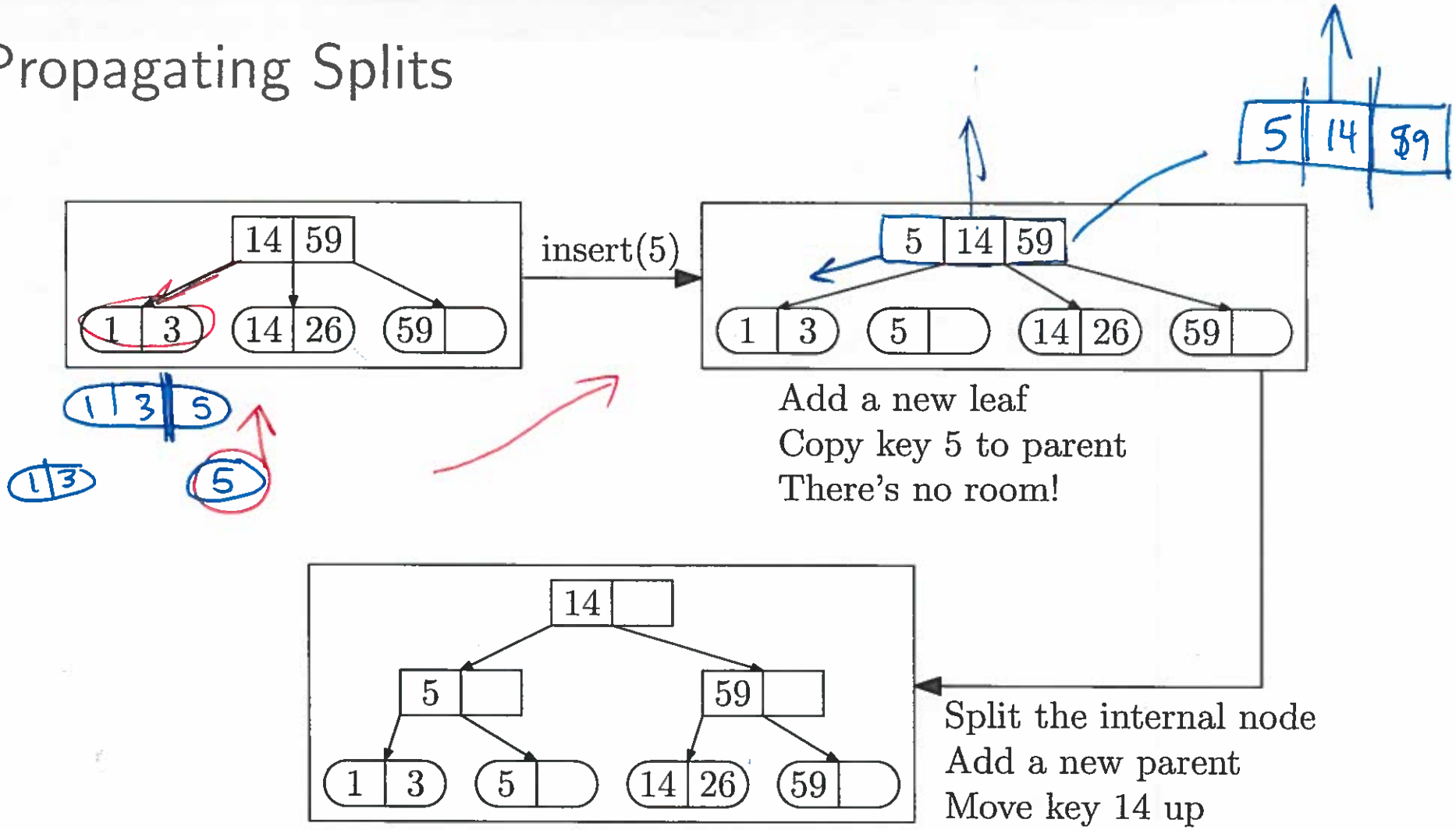
Splitting a Leaf



insert(26) causes too many keys for the (14 | 59) leaf to store.

So, make a new leaf and **copy** the "middle" key (the smallest key in the new leaf holding the larger keys) up to the common parent.

Propagating Splits



insert(5) causes too many keys for $\boxed{1 \mid 3}$ leaf

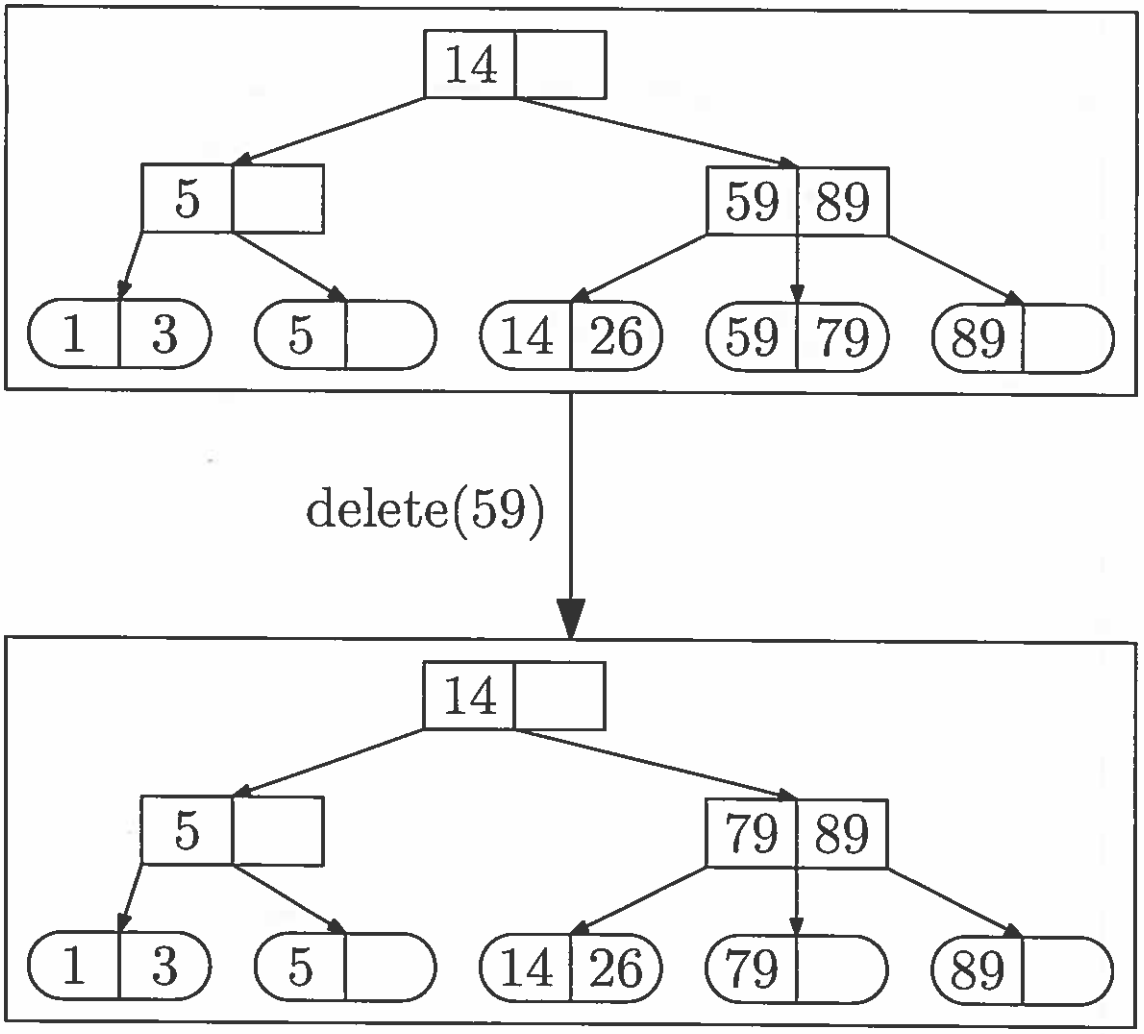
Copy up key 5 causes too many keys for $\boxed{14 \mid 59}$ node

So, make a new internal node and **move up** the middle key.

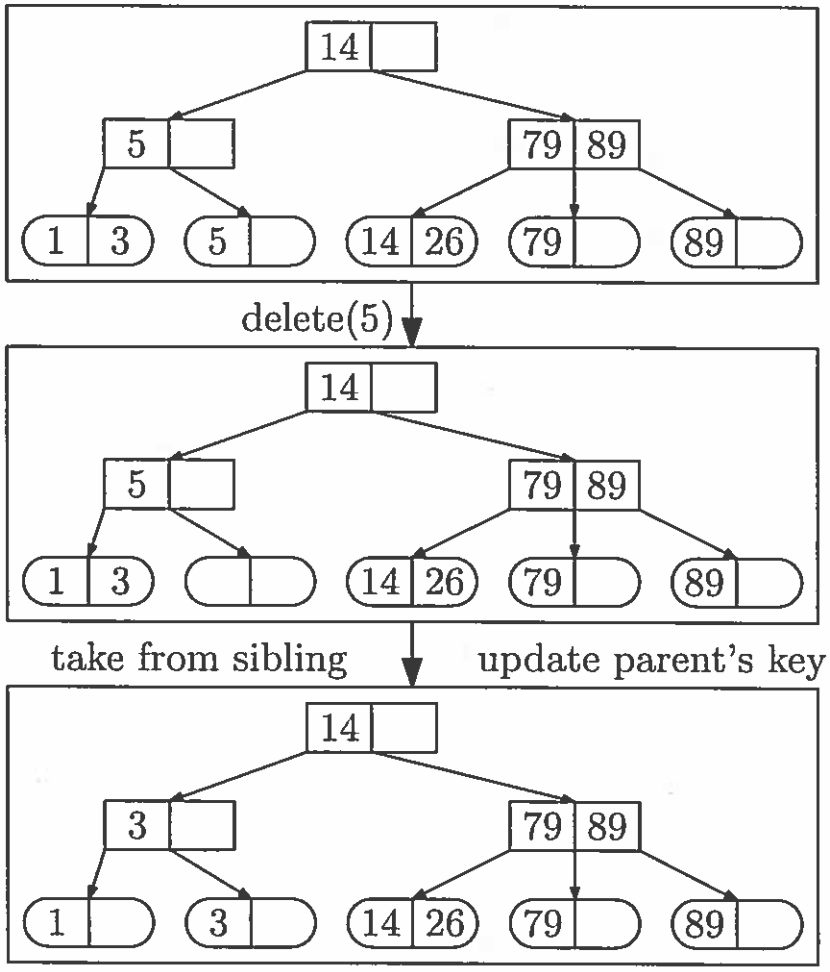
Insertion Algorithm

1. Insert (key, value) pair in the target leaf page
2. If the leaf now has $L+1$ pairs: // overflow
   - Split the leaf into two leaves:
     - Original holds the $\lceil (L+1)/2 \rceil$ small key pairs
     - New one holds the $\lfloor (L+1)/2 \rfloor$ large key pairs
   - **Copy** smallest key in new leaf (the middle key) up to parent
3. If an internal node now has M keys: // overflow
   - Split the node into two nodes:
     - Original holds the $\lceil (M-1)/2 \rceil$ small keys
     - New one holds the $\lfloor (M-1)/2 \rfloor$ large keys
   - If root, hang the new nodes under a new root. Done.
   - **Move** the remaining middle key up to parent & go to 3

Delete

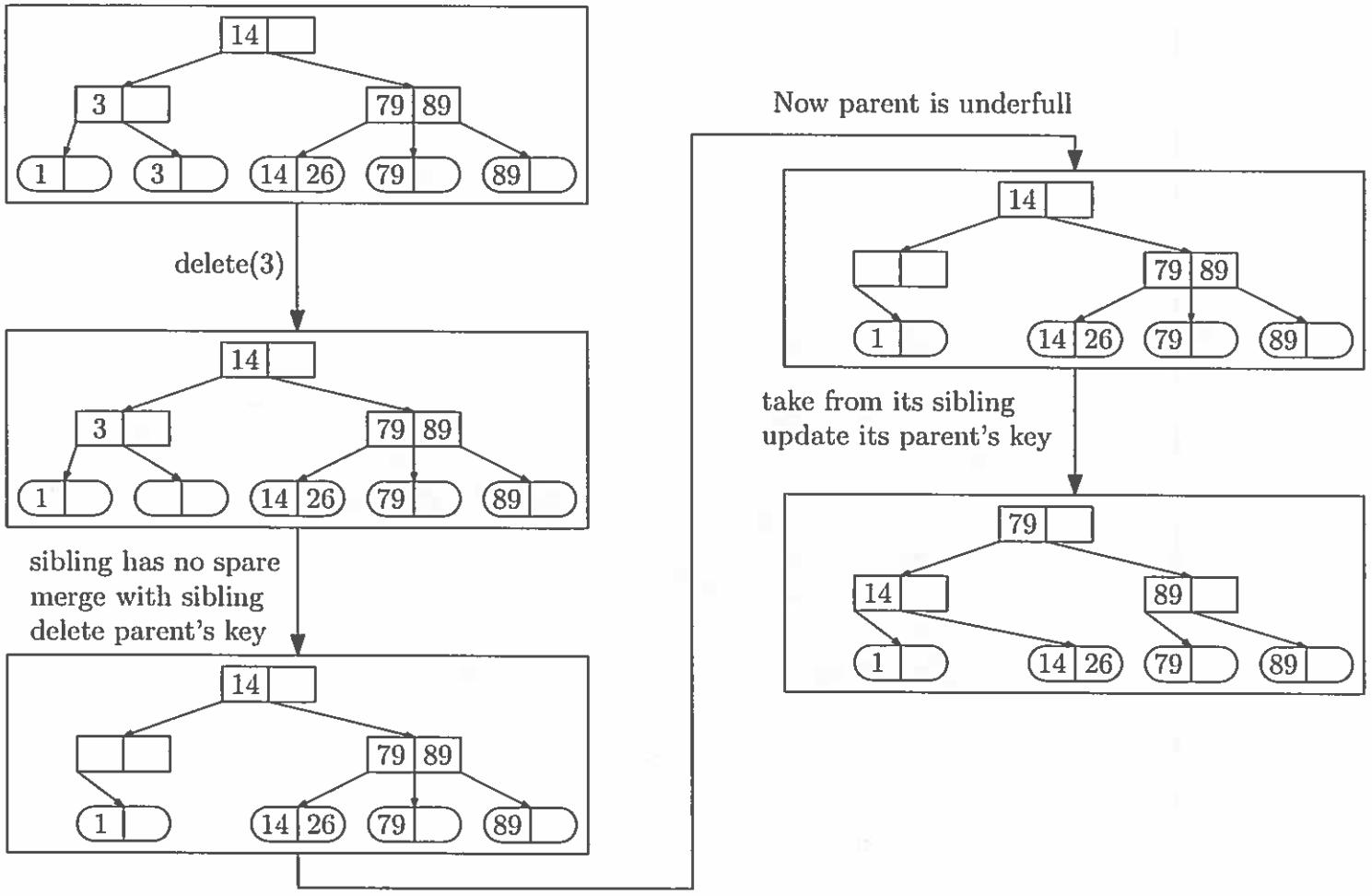


Delete: Take from a sibling



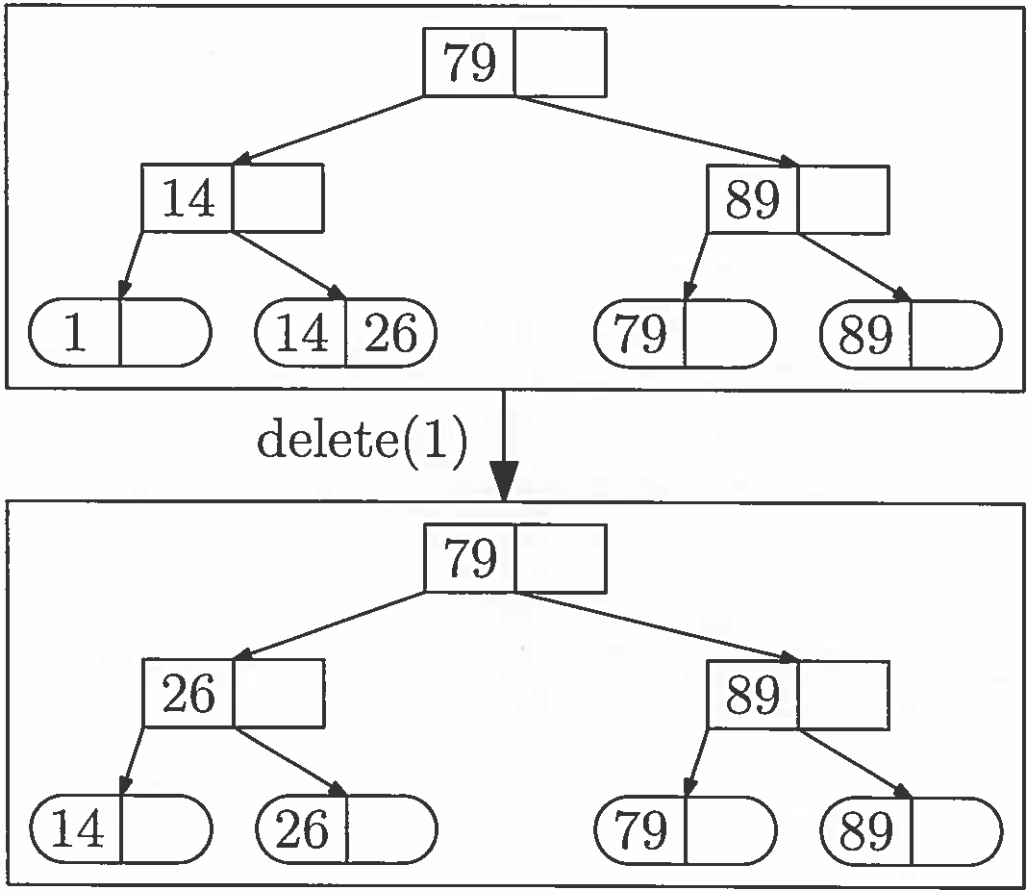
Take 3 from (1 | 3). It has enough items that it can spare one.
Update the parent's search key. This is not optional.

Delete: Merge

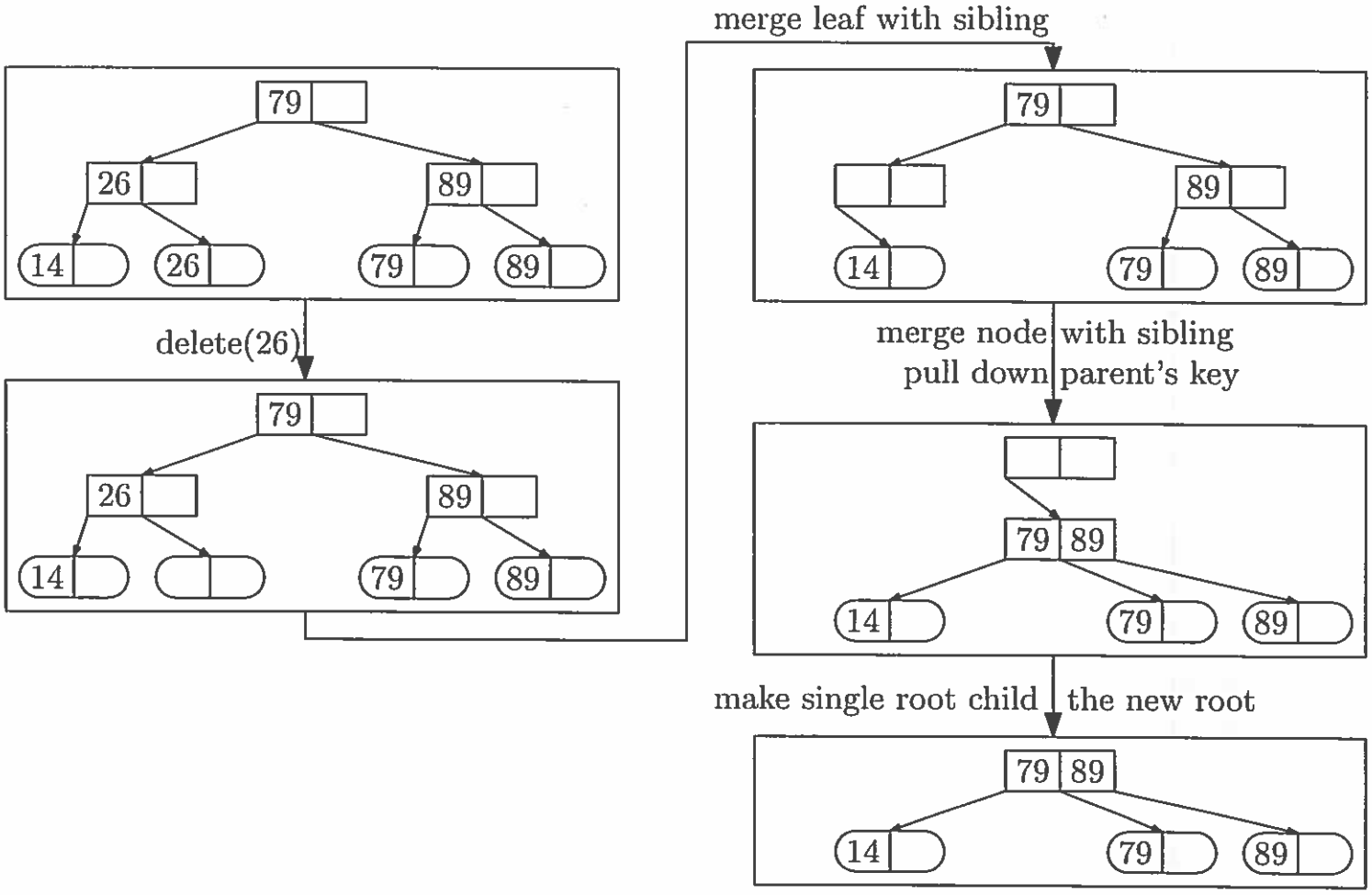


WARNING: A leaf is underfull if it holds fewer than $\lceil L/2 \rceil$ items.
For $L > 2$, an underfull leaf is not empty!

Delete: Take from a sibling



Deleting the Root



The root only gets deleted when it has just one subtree (no matter how big M is).

Deletion Algorithm

1. Remove the (key, value) pair from the correct leaf.
2. If the leaf now has $\lceil L/2 \rceil - 1$ items: // underflow
   - If a sibling has a spare item then take it (smallest from right sibling or largest from left sibling) & update parent's key
   - Else merge with a sibling & **delete** parent's key
3. If internal non-root node now has $\lceil M/2 \rceil - 2$ keys: // underflow
   - If a sibling has a spare child then take it (leftmost from right sibling or rightmost from left sibling) & update parent's key
   - Else merge with a sibling & **pull down** parent's key & go to 3
4. If the root now has only one child, make that child the new root.

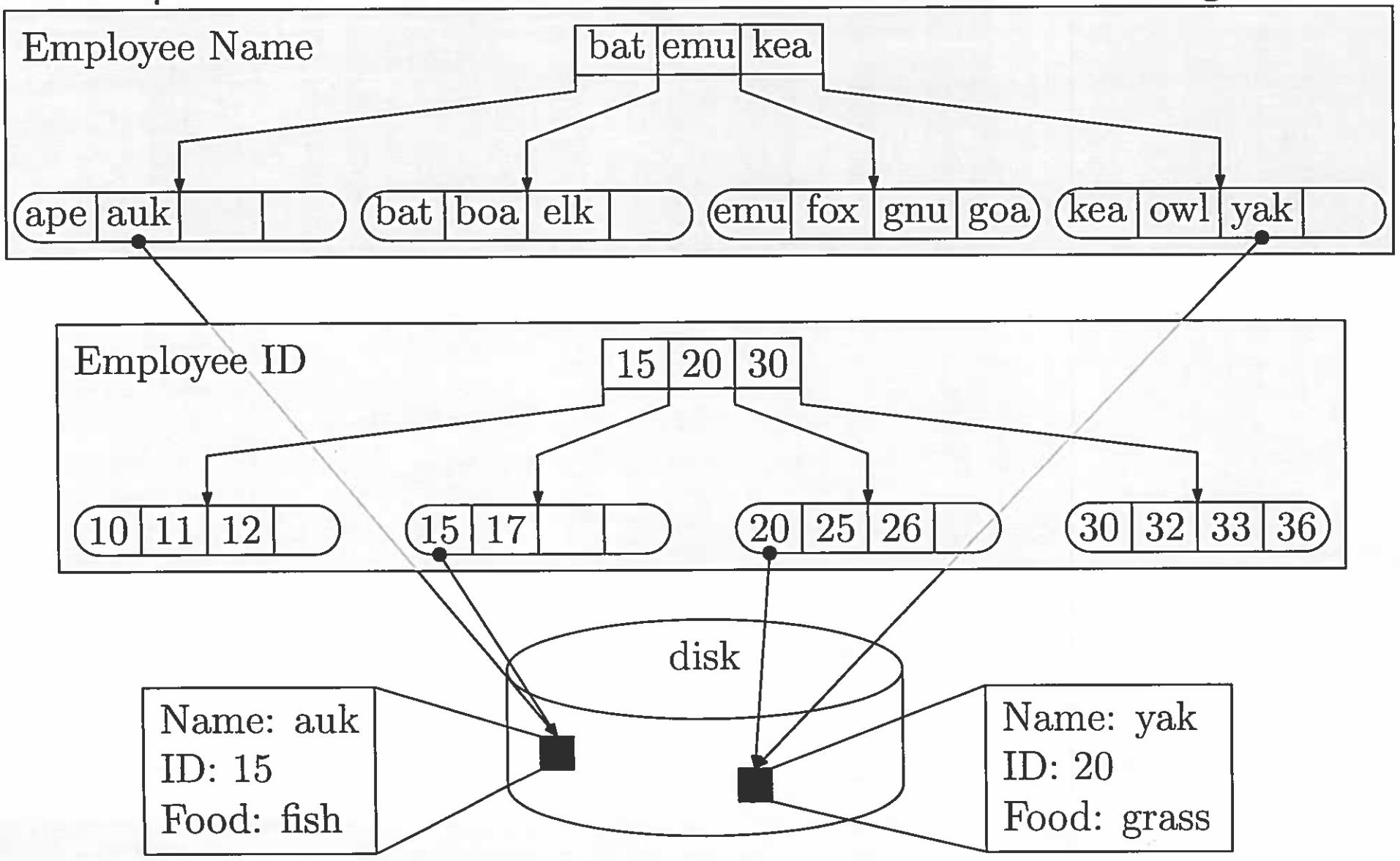
Note: Merge never creates a node with too many items. Why?

Thinking about B^+-Trees

- Deletion is fast if the leaf doesn't underflow or if we can take a (key, value) pair from a sibling. Merging and propagation take more time.
- Insertion is fast if the leaf doesn't overflow (could we give to a sibling?) Splitting and propagation take more time.
- Propagation is rare if M and L are large. (Why?)
- Repeated insertions and deletions can cause thrashing.
- If $M = L = 128$, then a B^+-Tree of height 4 will store at least 30,000,000 items.
- Range queries (i.e., `findBetween(key1, key2)`) are fast thanks to the sibling pointers in the leaves.

B$^+$-Trees in Practice

Multiple B$^+$-Trees can **index** the same data records. This is good.



A Tree by Any Other Name...

- B-Trees with $M = 3$ are called 2-3 trees
- B-Trees with $M = 4$ are called 2-3-4 trees