CPSC 221: Data Structures
B+-Trees

Alan J. Hu

(Using mainly Steve Wolfman's Slides)

Learning Goals

After this unit, you should be able to:

- Describe the structure, navigation and complexity of an order m B-tree.
- Insert and delete elements from a B+-tree, maintaining the half-full principle.
- Explain the relationship among the order of a B+-tree, the number of nodes, and the minimum and maximum elements of internal and external nodes.
- 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 operations/steps) when dealing with larger datasets and their indexing structures (e.g., B+-trees).
- Describe a B+-Tree and explain the difference between a B-tree and a B+ Tree

B-Tree Motivation

- We’ve got balanced BSTs (e.g. AVL trees):
  - Guaranteed worst case O(log n) performance for insert, find, delete
- We’ll get hash tables:
  - Expected O(1) insert, find, delete
- Why in the world do we need **another** dictionary data structure???

Answer: Because constant factors matter in practice!

Memory Hierarchy

- Computers are built with different kinds of memory, because it's impossibly expensive (and physically impossible) to build all memory to be incredibly fast:
  - Processor Registers: 100s of locations, <1 cycle access time
  - L1 Cache: 1000s of locations, a few cycles to access
  - L2/L3 Cache: Millions of locations, tens of cycles to access
  - Main Memory: Billions of locations, hundreds of cycles to access
  - Disk: Trillions of locations (or more), millions of cycles to access

Coping with the Memory Hierarchy

- Wait! I can go to Future Shop and buy a 1TB disk for less than a hundred bucks. If average seek time is 10ms for a disk read, it should take me about 1TB * 10ms to read all the data off the disk.
- 1 tera * 10 ms = 10 billion seconds > 300 years
- Either that disk is VERY slow, or your numbers are wrong. What's going on?

Answer: You don't read/write one byte at a time.

Coping with the Memory Hierarchy

- At every level of the memory hierarchy, the slow access to the lower level is amortized by getting a whole bunch of data at once.
  - For cache, these are called “cache lines” or “blocks”, 16, 32, 64, 128 bytes, etc. common
  - For main memory, typically called “pages”, 1k, 2k, 4k, 8k, 16k, etc. common
  - For disk, typically called “blocks”, 1k, 2k, 4k, 8k, etc. common

Coping with the Memory Hierarchy

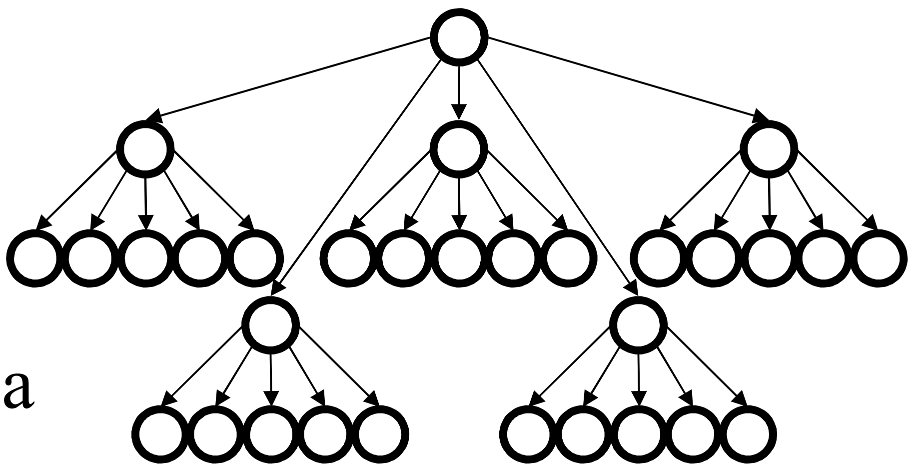
- Therefore, *random* accesses are very slow.
- Sequential access, or lots of access to a single block of data, are much much faster.

- What do hash tables do?
- What do AVL trees do?

M-ary Search Tree

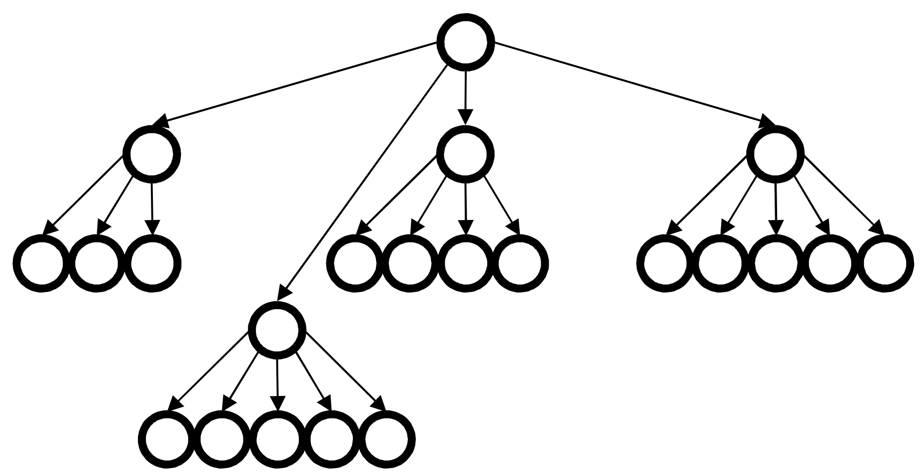
- Maximum branching factor of M
- Complete tree has depth = $\log_M N$
- Each internal node in a complete tree has $M - 1$ keys

runtime:



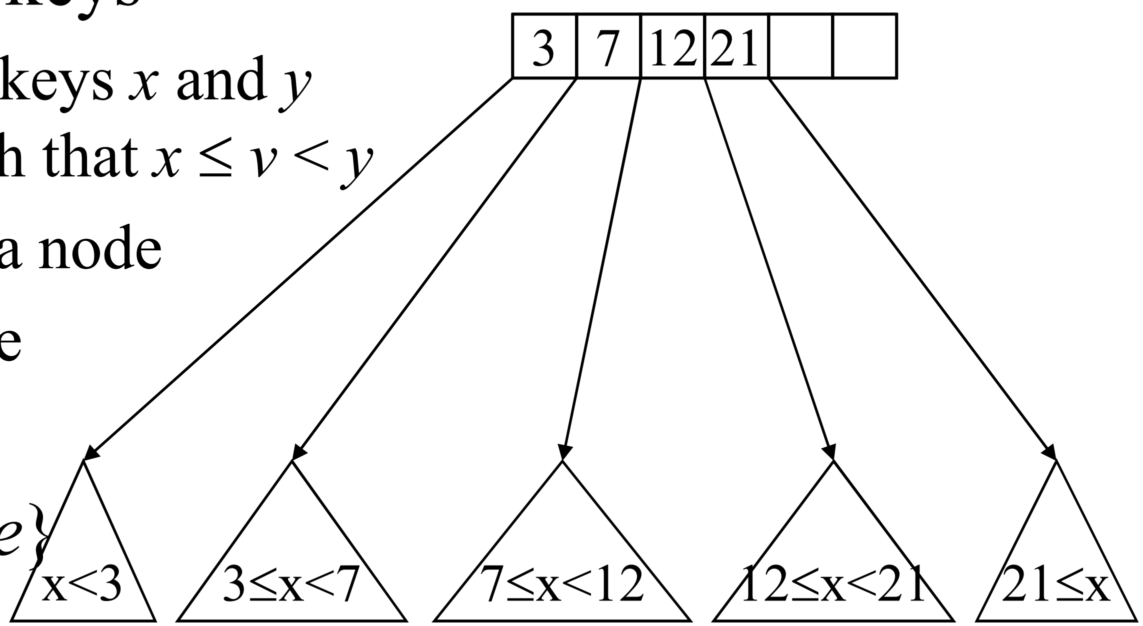
Incomplete *M*-ary Search Tree ☹

- Just like a binary tree, though, complete m-ary trees has m^0 nodes, $m^0 + m^1$ nodes, $m^0 + m^1 + m^2$ nodes, …
- What about numbers in between??



B-Trees

- B-Trees are specialized *M*-ary search trees
- Each node has many keys
  - subtree between two keys x and y contains values v such that $x \le v < y$
  - binary search within a node to find correct subtree
- Each node takes one full {*page, block, line*} of memory
- ALL the leaves are at the same depth!



Today’s Outline

- B-tree motivation
- B+-tree properties
- Implementing B+-tree insertion and deletion
- Some final thoughts on B+-trees

B+Tree Properties

- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *L* keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only *search* keys (no data)
  - smallest datum between search keys *x* and *y* equals *x*
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and *L* keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

B+Tree Properties

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys x and y equals x
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

B+Tree Properties

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys x and y equals x
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

B+Tree Properties

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys x and y equals x
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

Aside: B-Tree Properties

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - ~~internal nodes contain only search keys (no data)~~
  - ~~smallest datum between search keys x and y equals x~~
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

Aside: B-Tree Properties

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes do contain data $\boxed{\text{Just like BSTs!}}$
  - data in subtrees between keys x and y strictly between x and y
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - tree is $\Theta(\log_M n)$ deep (between $\log_{M/2} n$ and $\log_M n$)
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

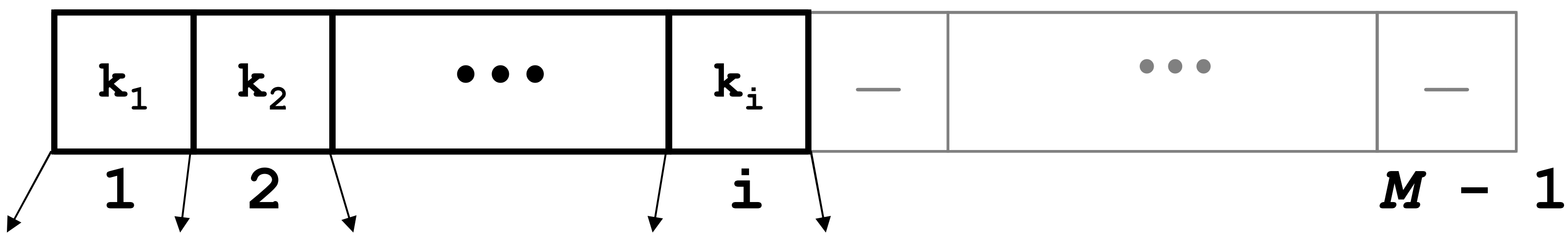
Today’s Outline

- Addressing our other problem
- B+-tree properties
- Implementing B+-tree insertion and deletion
- Some final thoughts on B+-trees

B+Tree Nodes

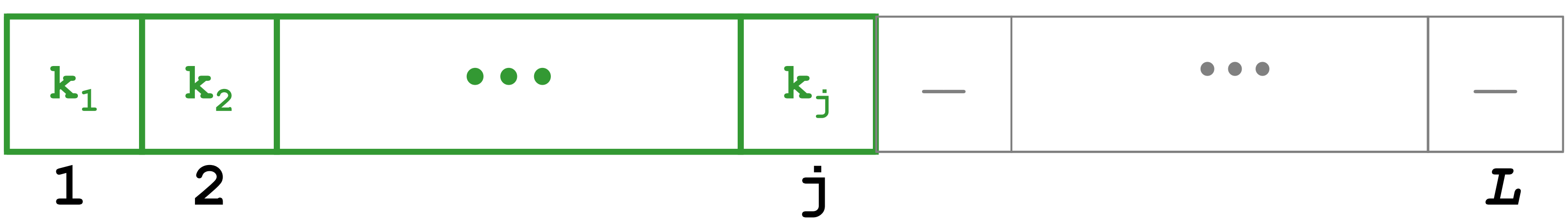
- Internal node

  `i` search keys; `i+1` children; ***M* − 1 −i** inactive keys



- Leaf

  `j` data keys; ***L* − j** inactive entries



Alan's Aside: B+Tree Nodes

```
struct btree_node {
    bool is_leaf;
    int key_count;
    int key[max(M-1, L)]; // some key_type in reality
    int child_count;
    union {  // uses same memory space
        btree_node *child[M];
        data_type *leaf_data[L];
    }
}
```

child[i] between key[i-1] and key[i]

Alan's Aside: B+Tree Nodes

```
struct btree_node {
    bool is_leaf;
    int key_count;
    int key[max(M-1, L)]; // some key_type in reality
    int child_count;
    union {  // uses same memory space
        btree_node *child[M];
        data_type *leaf_data[L];
    }
}
```

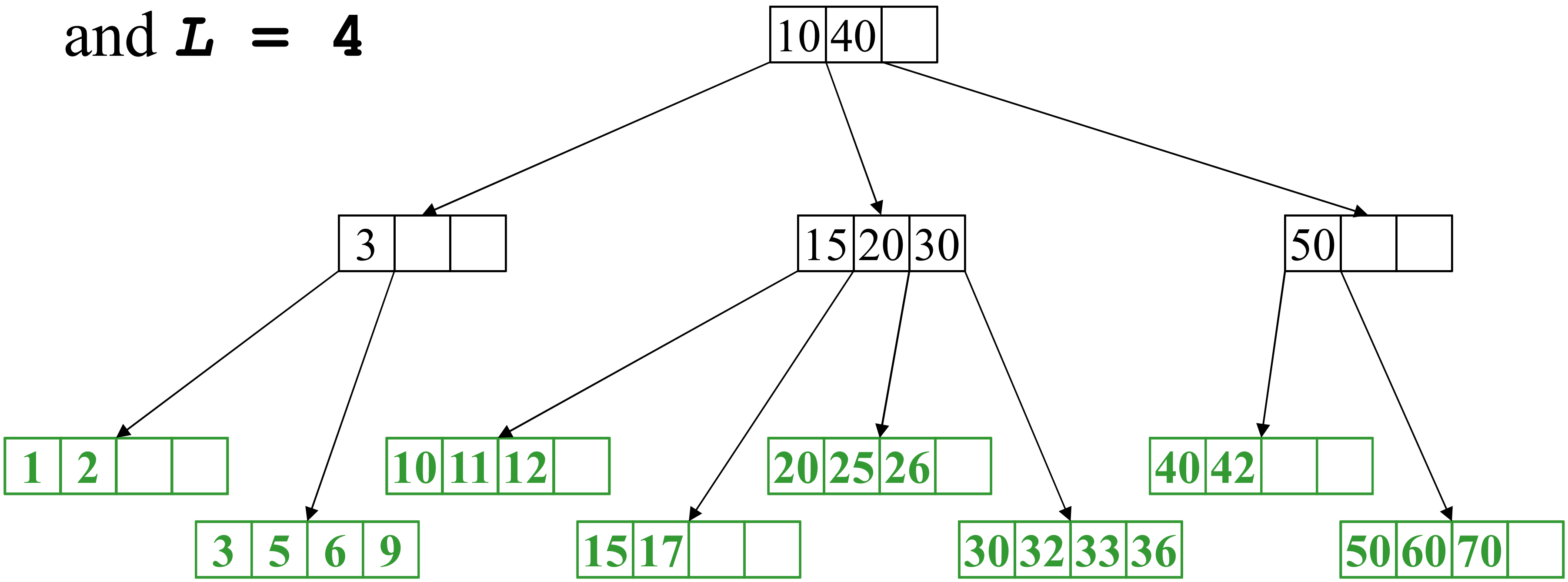
child[i] between key[i-1] and key[i]

The smallest key in subtree rooted at child[i] is exactly equal to key[i-1]

Example

B+Tree with `M = 4`

and `L = 4`

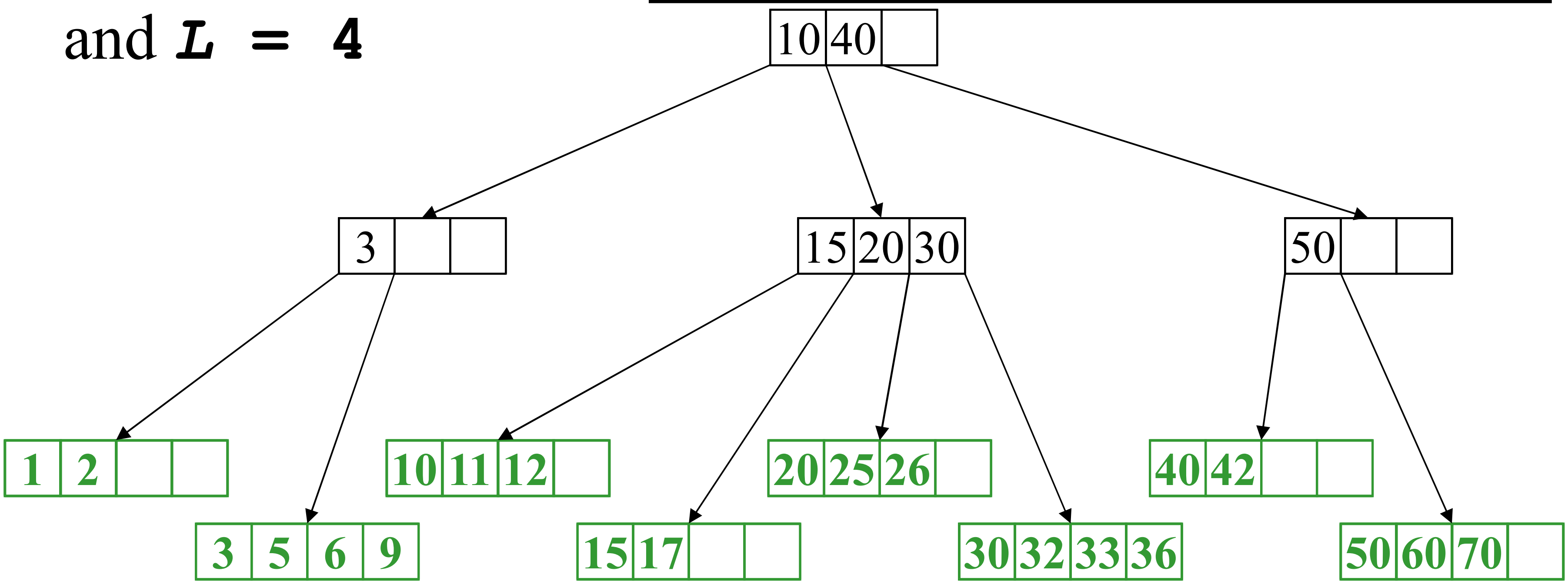


Example

Notice in these pictures that we are drawing the keys, but not the pointers, so there are 3 boxes, but M=4

B+Tree with ***M* = 4**

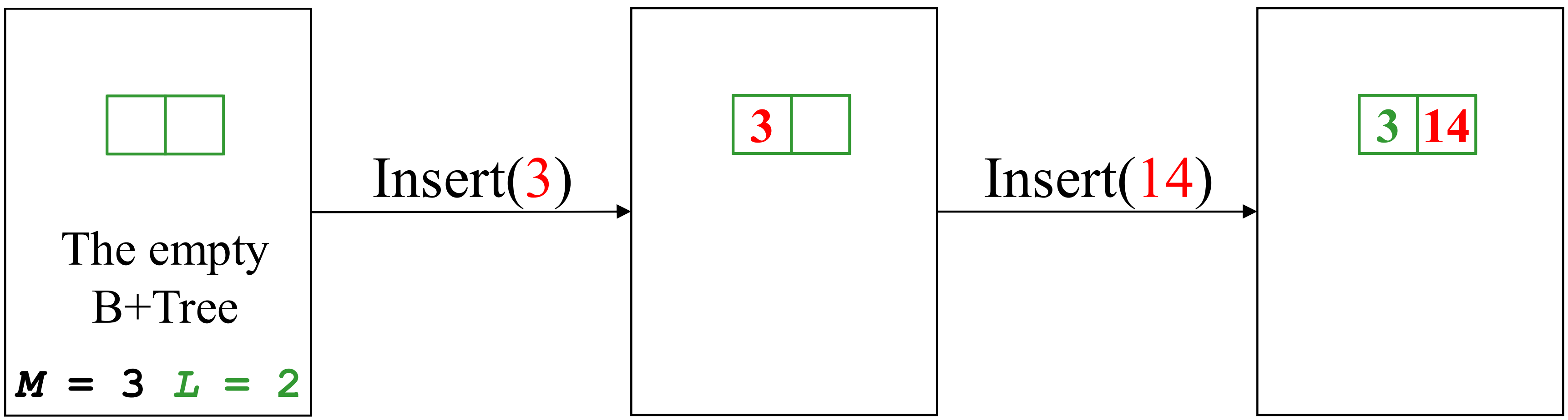
and ***L* = 4**



B+Tree Find Pseudo-Code

```
data_type * find(btree_node *root, int target) {
    if (root->is_leaf) {
        binary search on root->key array for target
        if (found at location i) return root->leaf_data[i];
        else return null;
    }
    binary search on root->key array for target
    let i be the correct subtree
    return find(root->child[i], target)
}
```

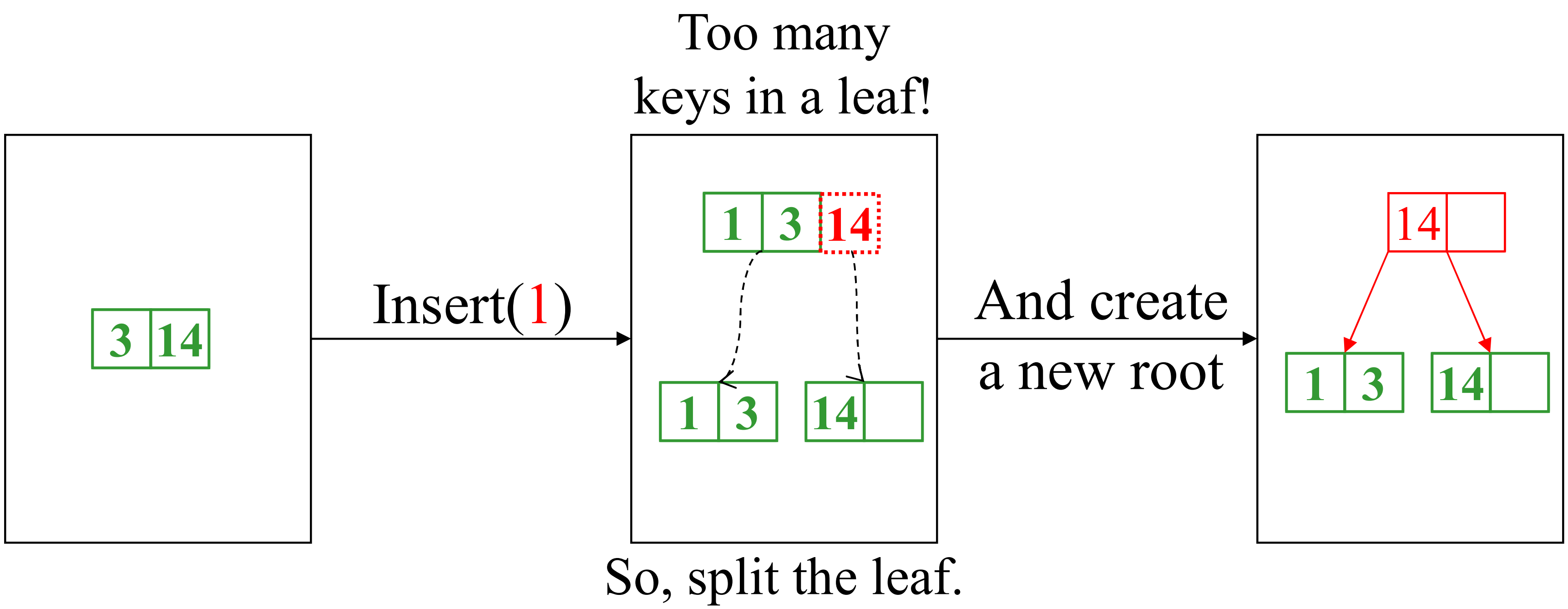
Making a B+Tree



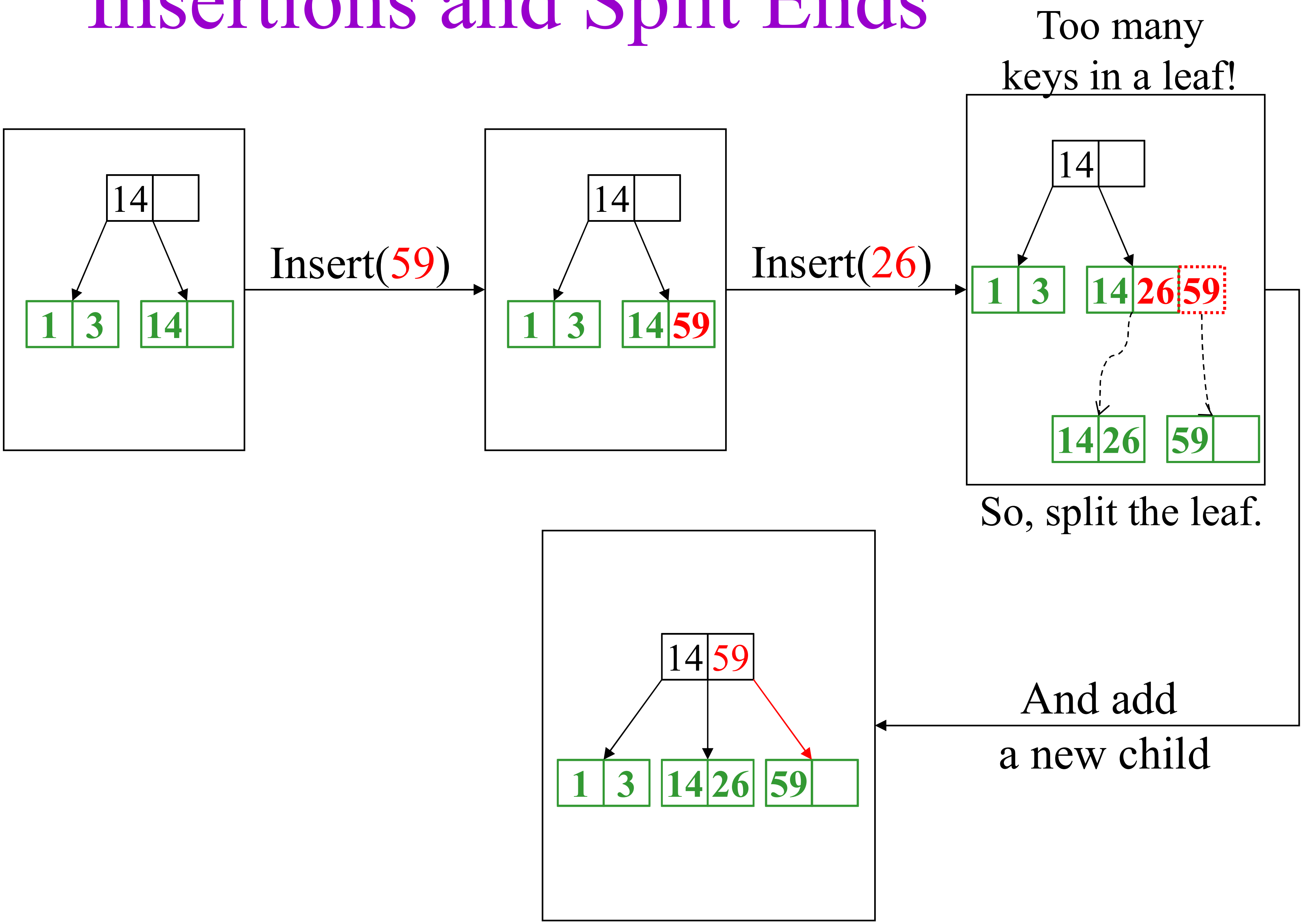
B-Tree with **M = 3**
and **L = 2**

Now, Insert(1)?

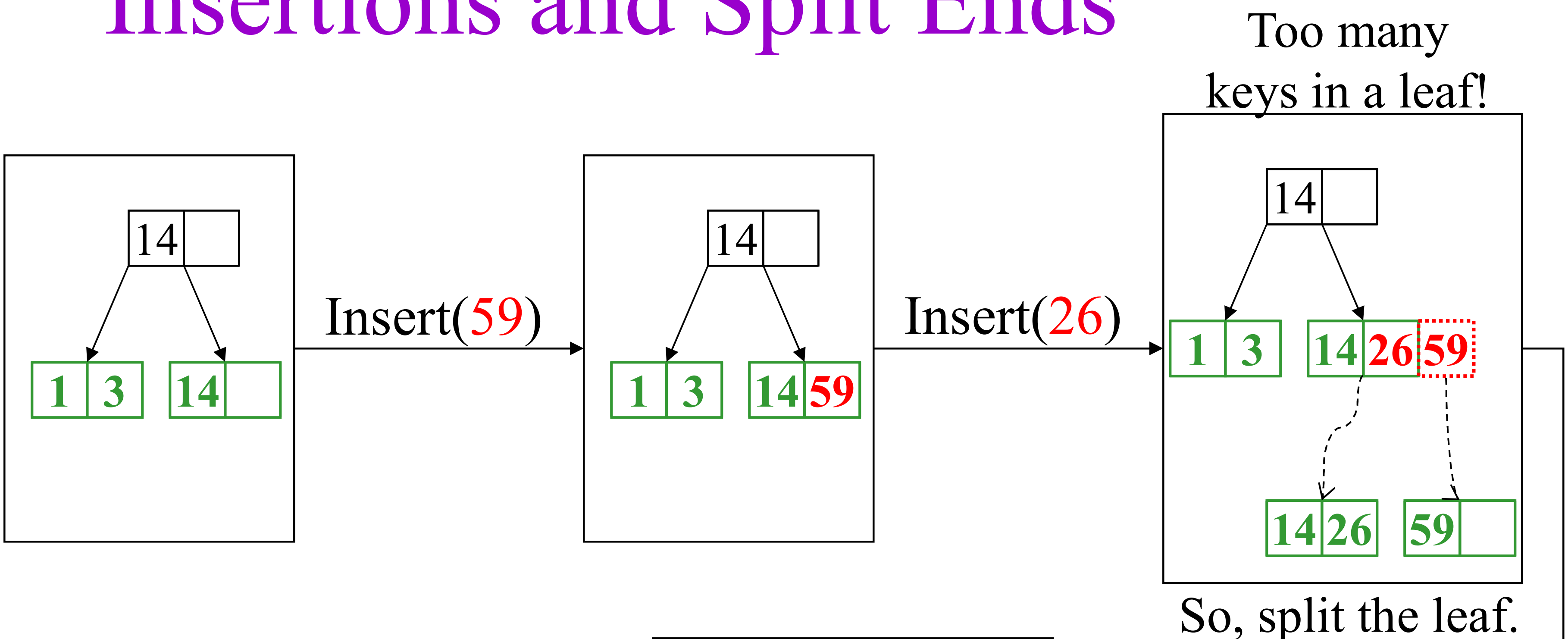
Splitting the Root

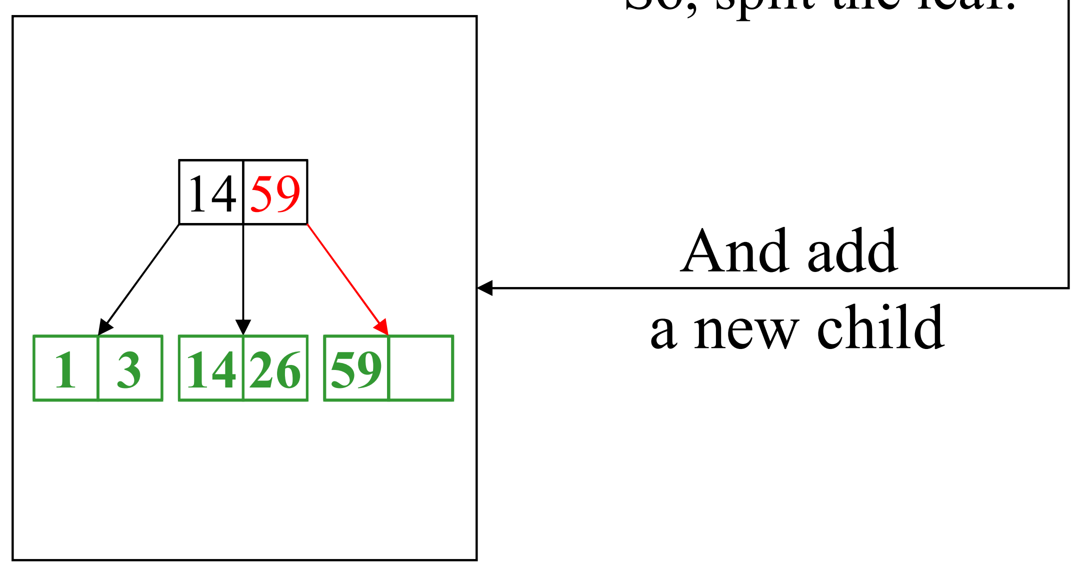


Insertions and Split Ends



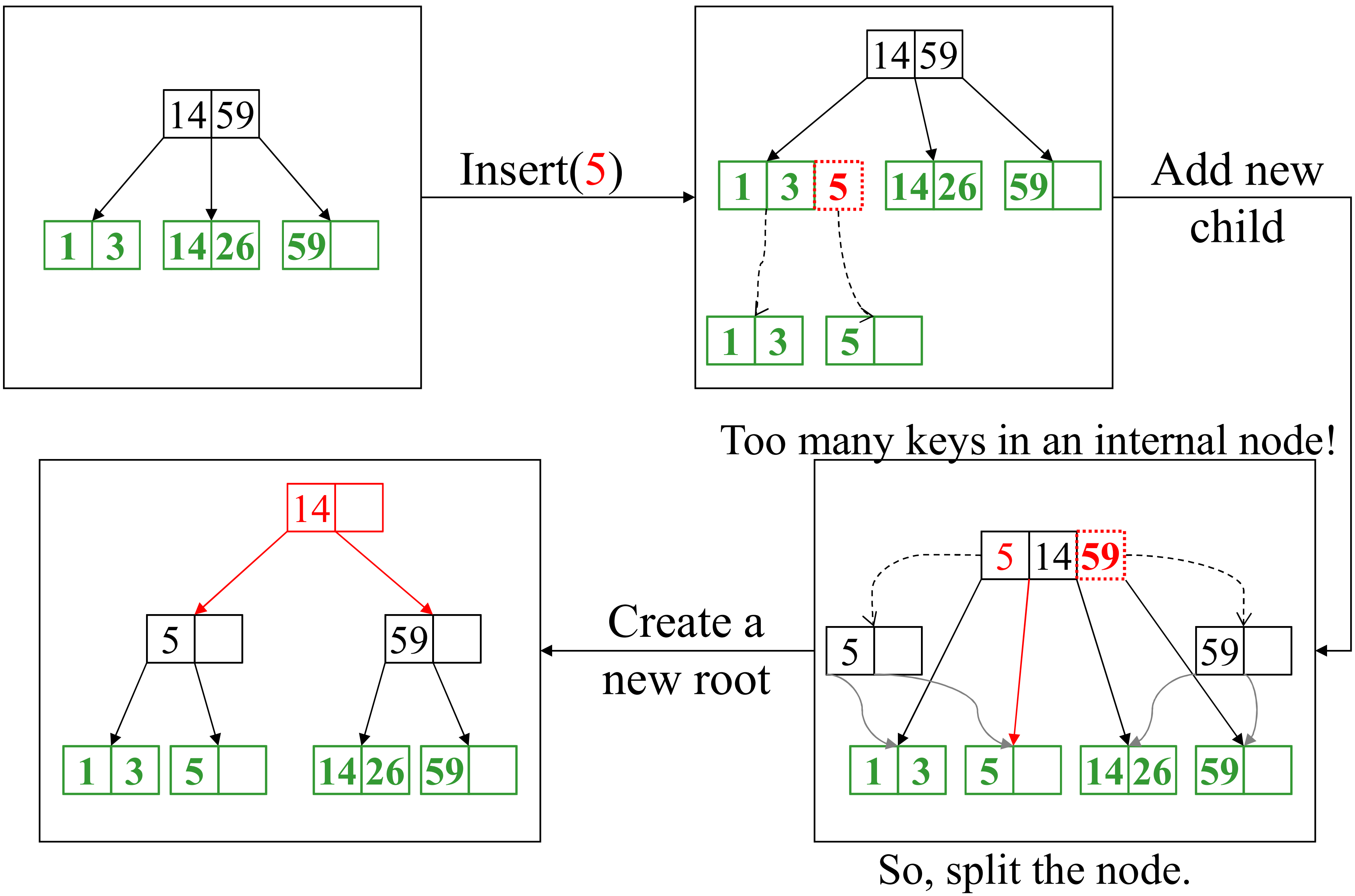
Insertions and Split Ends



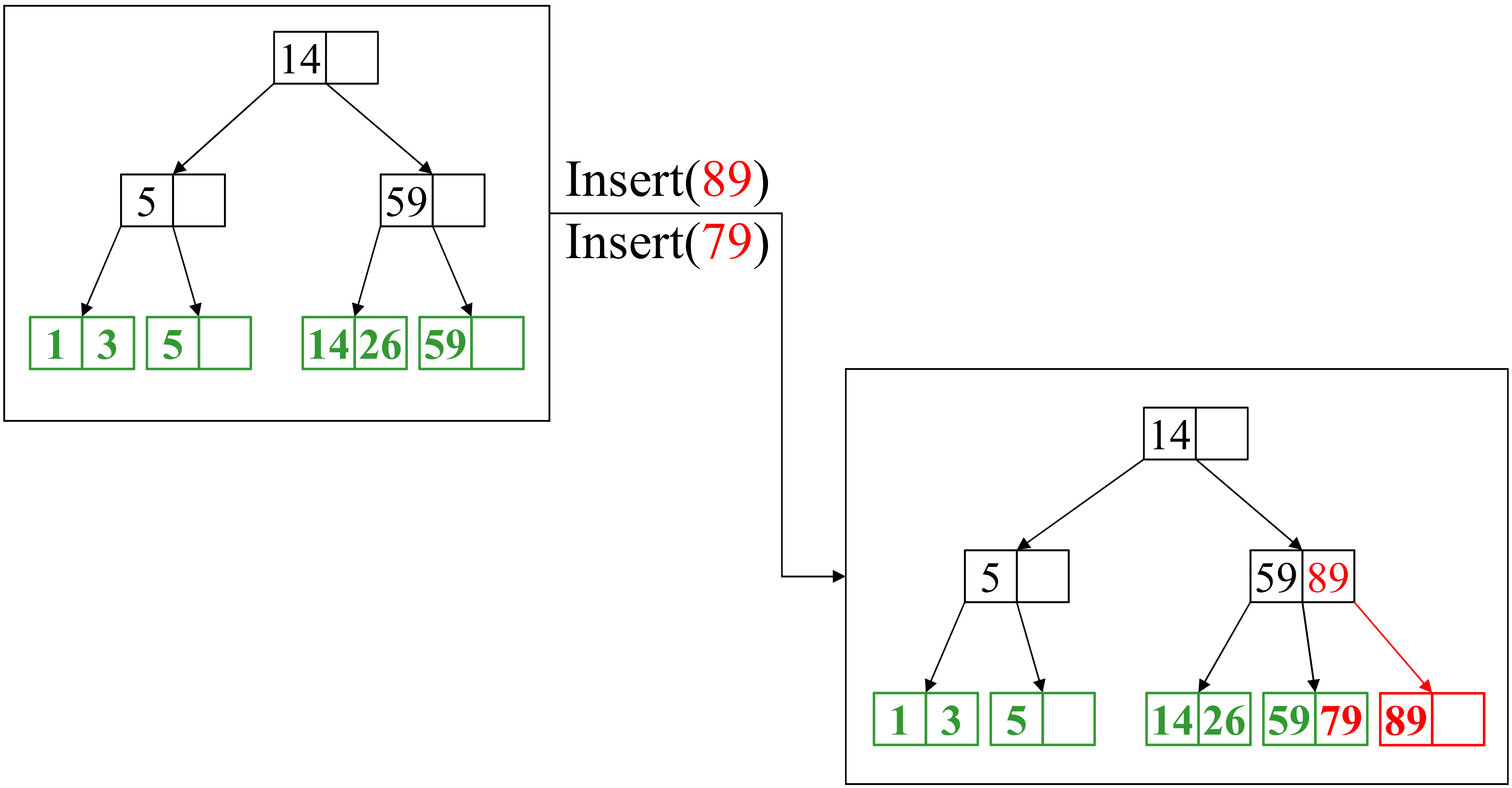


Alan's Aside:
I don't really like this picture. Leaves are always at same level. Tree grows from the root!

Propagating Splits



After More Routine Inserts



Insertion in Boring Text

- Insert the key in its leaf
- If the leaf ends up with L+1 items, **overflow**!
  - Split the leaf into two nodes:
    - original with $\lceil \mathbf{(}\boldsymbol{L}\mathbf{+1)/2} \rceil$ items
    - new one with $\lfloor \mathbf{(}\boldsymbol{L}\mathbf{+1)/2} \rfloor$ items
  - Add the new child to the parent
  - (If the parent ends up with $\boldsymbol{M}\mathbf{+1}$ items, **overflow**!)

- If an internal node ends up with M+1 items, **overflow**!
  - Split the node into two nodes:
    - original with $\lceil \mathbf{(}\boldsymbol{M}\mathbf{+1)/2} \rceil$ items
    - new one with $\lfloor \mathbf{(}\boldsymbol{M}\mathbf{+1)/2} \rfloor$ items
  - Add the new child to the parent
  - (If the parent ends up with $\boldsymbol{M}\mathbf{+1}$ items, **overflow**!)

- Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Insertion Recursion in English

- If key is in my key array, return. It's already in the dictionary.
- If this node is a leaf,
  - insert the new key/data into the leaf.
  - If the leaf is too big, split into two leaves, and return, notifying my parent of the overflow, the new leaf, and the key value for the new leaf.
- If this node is not a leaf,
  - recurse down the correct child.
  - If the child returns no overflow, then just return.
  - If the child returns overflow, then insert new key/child into my arrays.
  - If preceding step makes me overflow, split myself into two nodes, and return, notifying my parents of the overflow, the new node, and key value for new node.

B+Tree Insert Pseudo-Code

```
void insert(btree_node *root, int target, data_type * data,
            bool &overflow, int &new_key, btree_node *&new_node)
{
    // Assuming no duplicate keys inserted…
    if (root->is_leaf) {
        if (child_count<L) {
                insert new key and data into arrays
                overflow = false;
                return;
        } else {
                create a new node and move half of keys/data over
                overflow = true; new_key = smallest key of new node;
                return;
        }
    }
```

B+Tree Insert Pseudo-Code 2

```
void insert(btree_node *root, int target, data_type * data,
        bool &overflow, int &new_key, btree_node
    *&new_node)
{
    …
    // Recursive case
    binary search on root->key array for target
    let i be the correct subtree
    insert (root->child[i], target, data, overflow, …);
```

B+Tree Insert Pseudo-Code 3

```
    …
    // Recursive case
    …
    insert (root->child[i], target, data, overflow, …);
    if (overflow) {
          ?
    }
}
```

B+Tree Insert Pseudo-Code 3

```
…
if (overflow) {
	if (key_count<M-1) {
		insert new key and child into arrays
		overflow = false;
		return;
	} else {
		create a new node and move half of the children over
		overflow = true;
		new_key = the key that used to be at the split;
		return;
	}
}
```

B+Tree Insert Pseudo-Code 3

```
…
if (overflow) {
	if (key_count<M-1) {
		insert new key and child into arrays
		overflow = false;
		return;
	} else {
		create a new node and move half of the children over
		overflow = true;
		new_key = the key that used to be at the split;
		return;
	}
}
```

This is where B+Tree property is very handy!

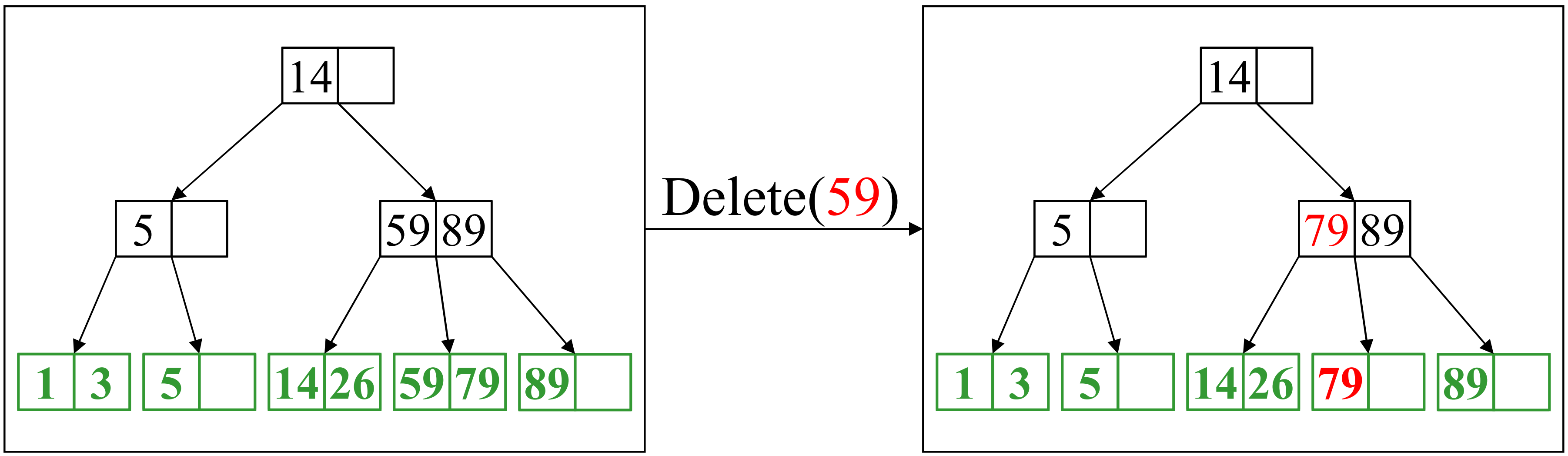
B+Tree Insert: Wrapper

- Our insert function has prototype:

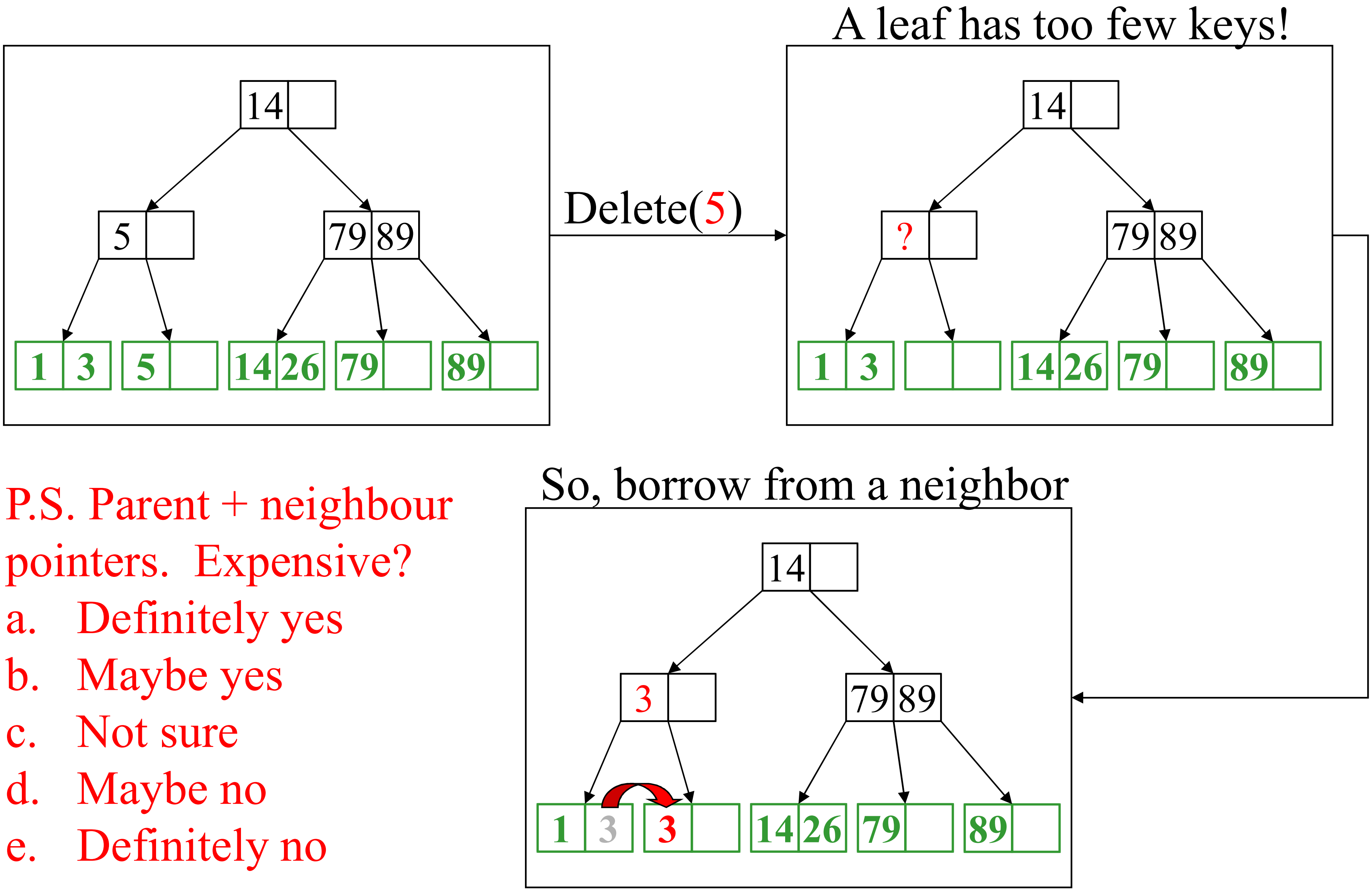
void insert(btree_node *root, int target, data_type * data, bool &overflow, int &new_key, btree_node *&new_node)

- Dictionary ADT insert doesn't!
- We've actually written an insert_helper. Must write an insert function that has proper prototype.
- This insert function will also take care of creating new nodes when root splits.

Deletion



Deletion and Adoption

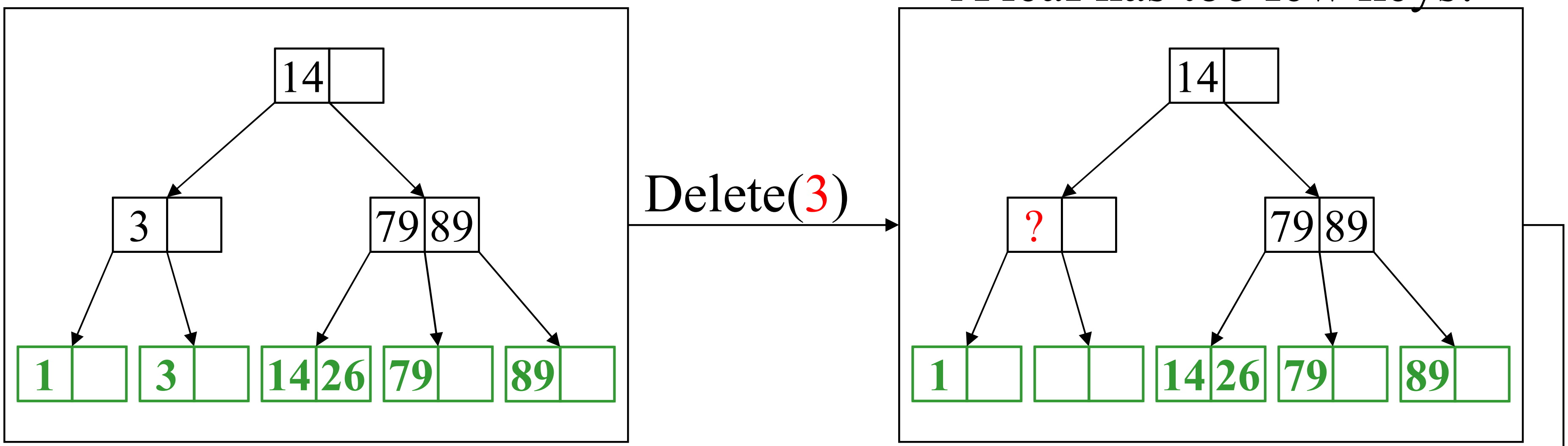


P.S. Parent + neighbour pointers. Expensive?

a. Definitely yes
b. Maybe yes
c. Not sure
d. Maybe no
e. Definitely no

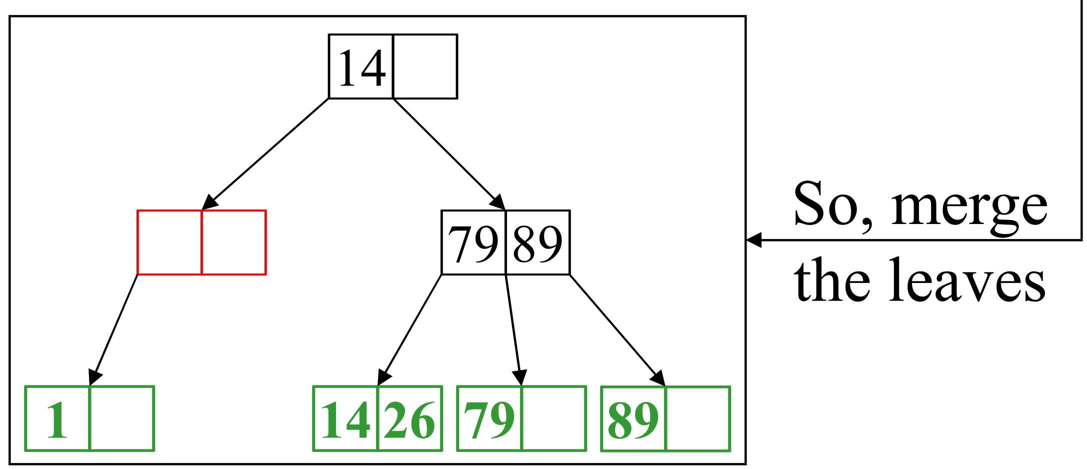
Deletion with Propagation

A leaf has too few keys!



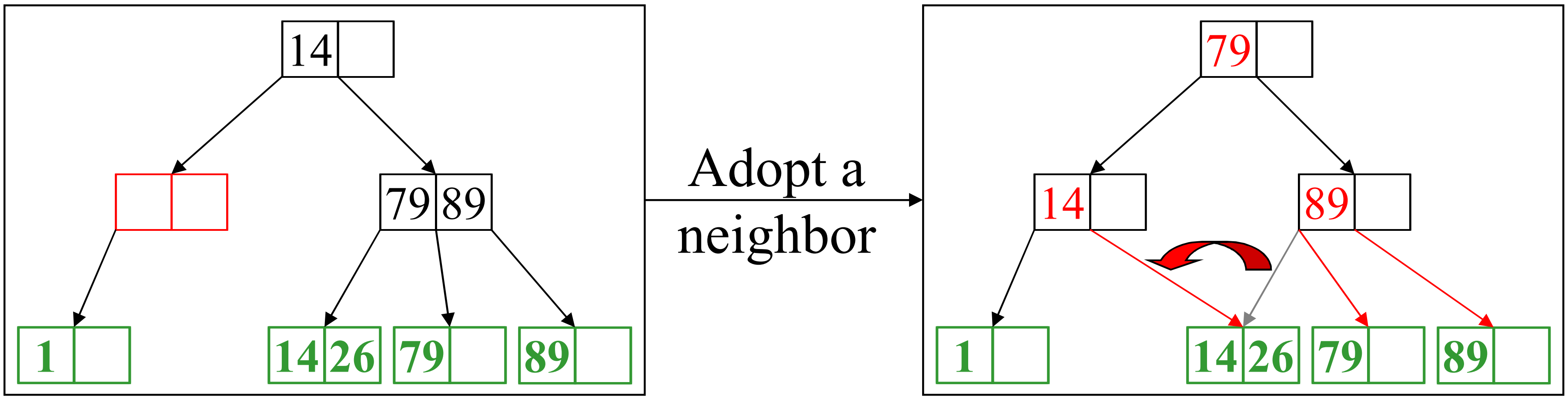
And no neighbor with surplus!

But now a node
has too few subtrees!

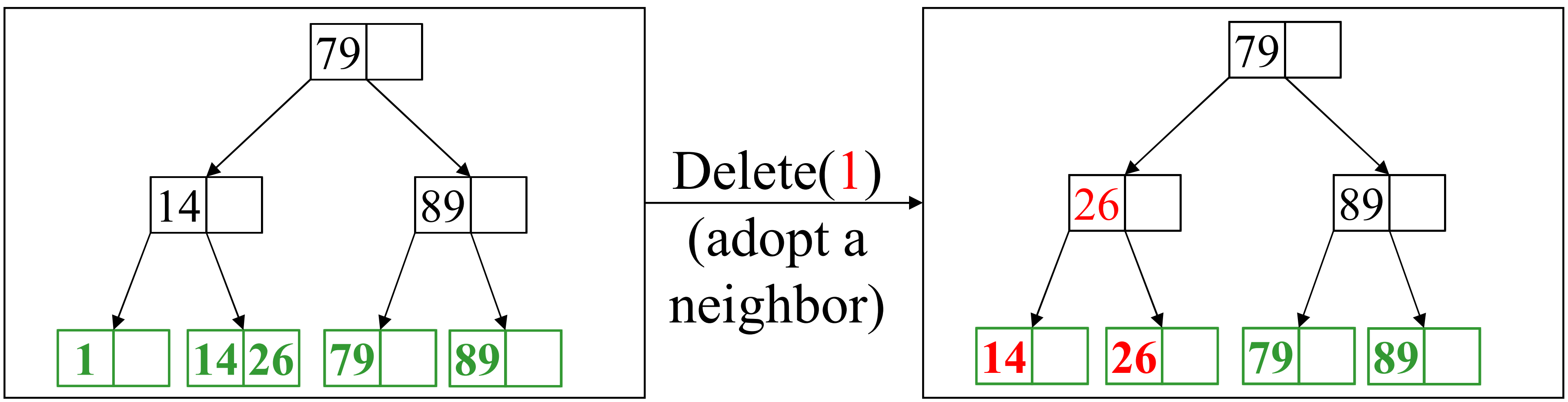


WARNING: with larger L,
can drop below L/2
without being empty!
(Ditto for M.)

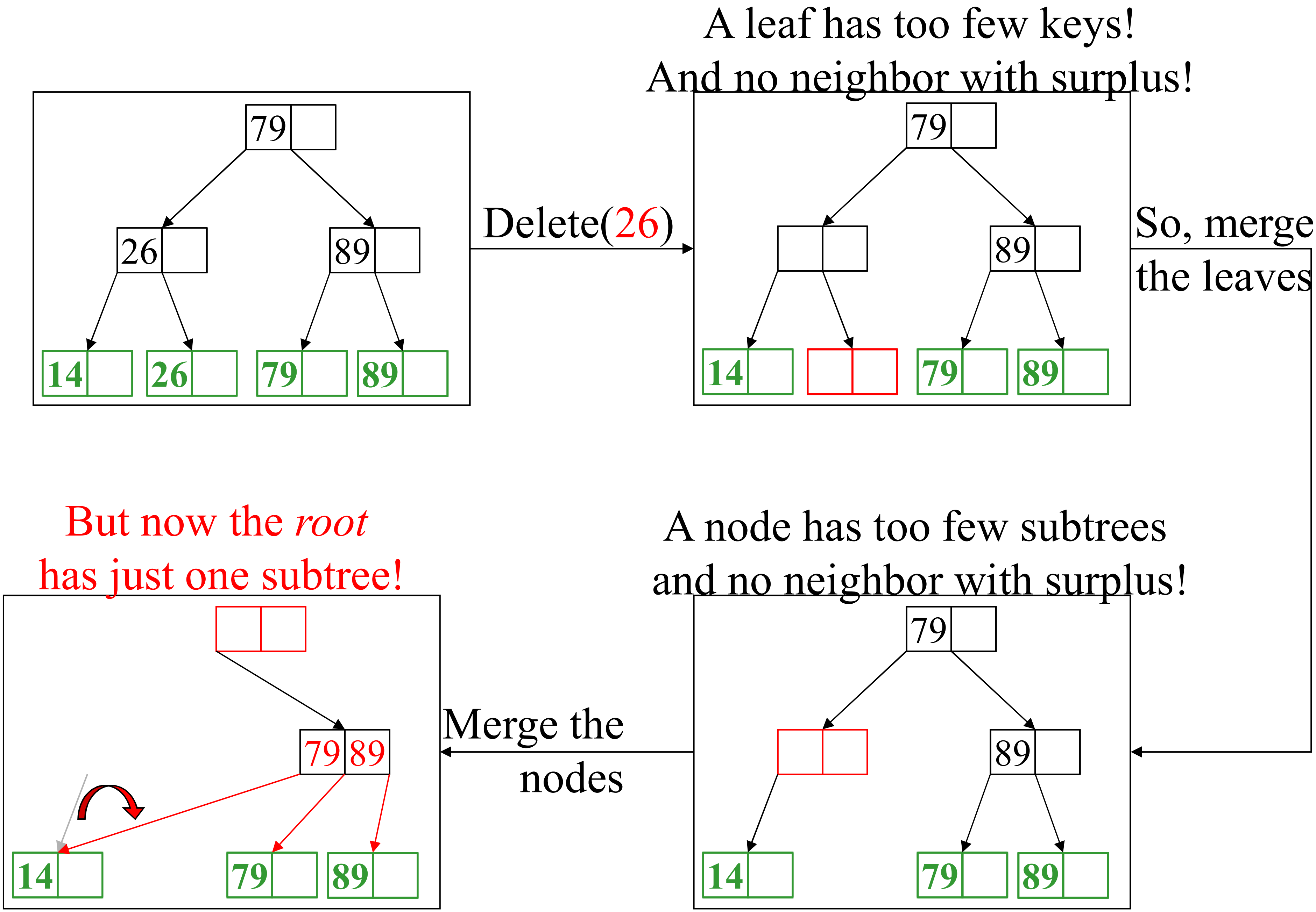
Finishing the Propagation (More Adoption)



A Bit More Adoption

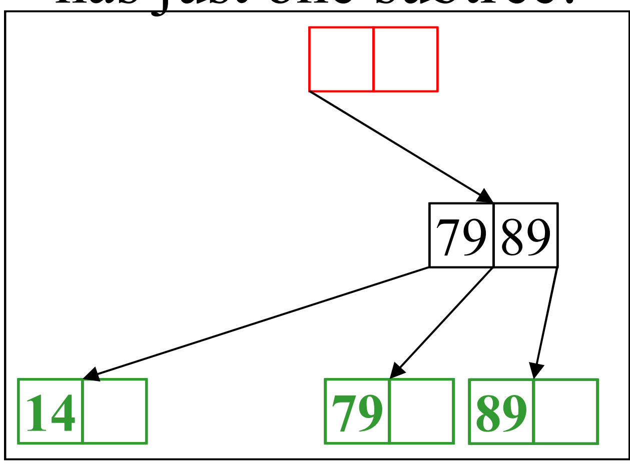


Pulling out the Root



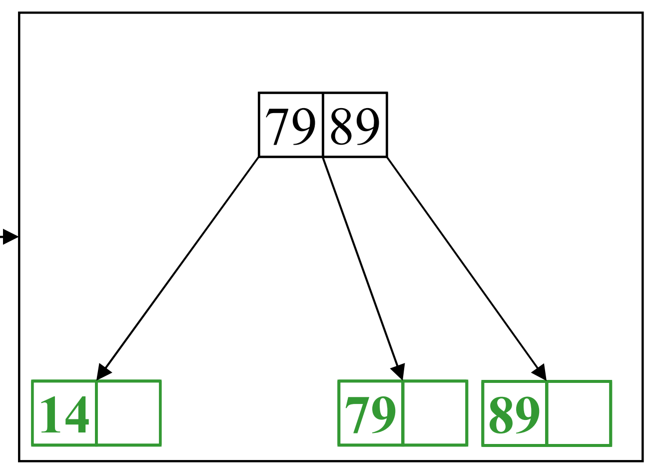
Pulling out the Root (continued)

The *root*
has just one subtree!



But that's silly!

Just make
the one child
the new root!



Note: The root really does only get deleted when it has just one subtree (no matter what M is).

Deletion in *Two* Boring Slides of Text

- Remove the key from its leaf
- If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
  - Adopt data from a neighbor; update the parent
  - If borrowing won't work, delete node and divide keys between neighbors
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!

Will dumping keys always work if adoption does not?

a. Yes
b. It depends
c. No

Deletion Slide Two

- If a node ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!
  - Adopt subtrees from a neighbor; update the parent
  - If borrowing won't work, merge with neighbor and update the parent
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!
- If the root ends up with only one child, make that child the new root of the tree

This reduces the height of the tree!

Deletion Recursion in English 1

- This is the big picture. We'll have to fix some details later:
- Base Case: If node is a leaf, search the leaf for key.
  - If not found, then nothing to do. Return.
  - If found, delete the key/data from the leaf.
  - Return, notifying parent if we underflowed.
- If node isn't a leaf:
  - Recurse down correct child.
  - If it returns without underflow, nothing more to do. Return.
  - If child underflowed, try to borrow from child's sibling(s).
  - If that fails, merge child with a sibling.
  - Return, notifying parent if **we** underflowed.

Deletion Recursion in English 1

- This is the big picture. We'll have to fix some details later:
- Base Case: If node is a leaf, search the leaf for key.
  - If not found, then nothing to do. Return.
  - If found, delete the key/data from the leaf.
  - Return, notifying parent if we underflowed.
- If node isn't a leaf:
  - Recurse down correct child.
  - If it returns without underflow, nothing more to do. Return.
  - If child underflowed, try to borrow from child's sibling(s).
  - If that fails, merge child with a sibling.
  - Return, notifying parent if **we** underflowed.

Borrowing from Left Sibling

- root->key[i-1] separates root->child[i-1] from root->child[i]
- Suppose I want to borrow a key/subtree from child[i-1] for child[i]. How do I do this?
  - Just remove from one array and insert into the other.
  - But, what are the new keys???
    - root->key[i-1]?
    - new root->child[i]->key[0]?
    - Anything else?
    - (Draw this out. Aha! Thanks to B+Tree property, keys are there!)

Borrowing from Right Sibling

- root->key[i] separates root->child[i] from root->child[i+1]
- Suppose I want to borrow a key/subtree from child[i+1] for child[i]. How do I do this?
  - Just remove from one array and insert into the other.
  - But, what are the new keys???
    - root->key[i]?
    - new root->child[i]->key[key_count]?
    - Anything else?
    - (Draw this out. Aha! Thanks to B+Tree property, keys are there!)

Deletion Recursion in English 1

- This is the big picture. We'll have to fix some details later:
- Base Case: If node is a leaf, search the leaf for key.
  - If not found, then nothing to do. Return.
  - If found, delete the key/data from the leaf.
  - Return, notifying parent if we underflowed.
- If node isn't a leaf:
  - Recurse down correct child.
  - If it returns without underflow, nothing more to do. Return.
  - If child underflowed, try to borrow from child's sibling(s).
  - If that fails, merge child with a sibling.
  - Return, notifying parent if **we** underflowed.

Merging with Left Sibling

- root->key[i-1] separates root->child[i-1] from root->child[i]
- Suppose we want to merge child[i-1] and child[i]. How do we do this?
  - Just merge keys/children/data arrays!
  - Delete root->key[i-1] from root->key[] array
  - But, before you do that, use root->key[i-1] as key to separate largest of child[i-1]'s children from smallest of child[i]'s children.
    - (Draw this out. Aha! Thanks to B+Tree property, keys are there!)

Merging with Right Sibling

- root->key[i] separates root->child[i] from root->child[i+1]
- Suppose we want to merge child[i] and child[i+1]. How do we do this?
  - Just merge keys/children/data arrays!
  - Delete root->key[i] from root->key[] array
  - But, before you do that, use root->key[i] as key to separate largest of child[i]'s children from smallest of child[i+1]'s children.
    - (Draw this out. Aha! Thanks to B+Tree property, keys are there!)

Wait! What if smallest value is the one deleted?!?

- Then the B+Tree property that key[i] is smallest value in child[i+1] doesn't hold temporarily.
- Therefore, preceding code is slightly wrong.
- Easy fix: Have the recursive calls return the value of the smallest item in their subtree, if it changed:
  - Base Case: In a leaf, if smallest value deleted, notify parent of new smallest value.
  - Recursion: If a recursive call on my child returns a new smallest value:
    - Update it's key, if it's not a leftmost child.
    - Notify my parent that **MY** smallest value has changed if it was my leftmost child.

Deletion Recursion in English -- Fixed

- Base Case: If node is a leaf, search the leaf for key.
  - If not found, then nothing to do. Return.
  - If found, delete the key/data from the leaf.
  - Return, notifying parent if we underflowed **and new smallest value if it changed.**
- If node isn't a leaf:
  - Recurse down correct child i.
  - **If child i tells me it changed smallest value, update key[i-1], or if i=0, save value to notify my parent that my smallest value changed.**
  - If it returns without underflow, nothing more to do. Return.
  - If child underflowed, try to borrow from child's sibling(s).
  - If that fails, merge child with a sibling.
  - Return, notifying parent if we underflowed **and new smallest value if it changed.**

Today's Outline

- Addressing our other problem
- B+-tree properties
- Implementing B+-tree insertion and deletion
- Some final thoughts on B+-trees

Thinking about B+Trees

- B+Tree insertion can cause (expensive) splitting and propagation (could we do something like borrowing?)
- B+Tree deletion can cause (cheap) borrowing or (expensive) deletion and propagation
- Propagation is rare if ***M*** and ***L*** are large *(Why?)*
- Repeated insertions and deletion can cause thrashing
- If **`M = L = 128`**, then a B-Tree of height 4 will store at least 30,000,000 items

Aside: B-Trees vs. B+Trees

- B-Trees were the original
  - Closer in structure to BSTs
  - Same asymptotic complexity as B+Trees
- B+Trees are more common in practice
  - Leaves are typically also linked together in a linked list
    - Makes it easy to do range queries
  - Leaves can be optimized for storing data
  - Easier to implement and explain operations
    - E.g., consider general case of merging nodes during deletion

A Tree by Any Other Name

FYI:

- B-Trees with `M = 3`, `L = x` are called 2-3 trees
- B-Trees with `M = 4`, `L = x` are called 2-3-4 trees
- 2-3-4 trees are basically the same as "Red-Black trees"

Why would we ever use these?