B-Trees

CMSC 420: Lecture 9

Another way to achieve "balance"

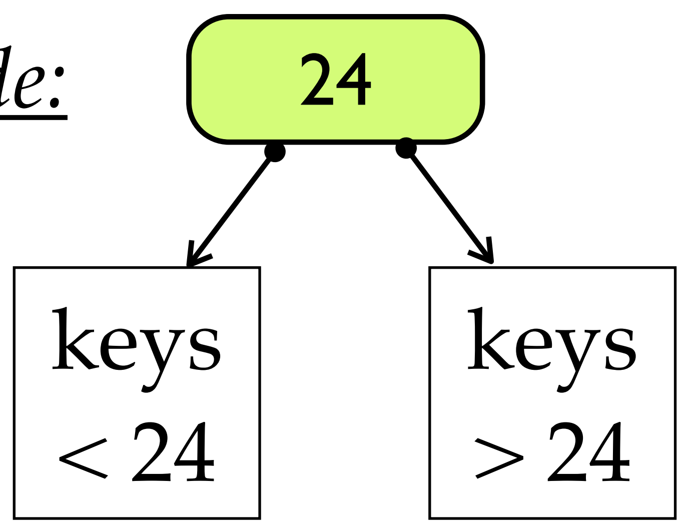
- Height of a perfect binary tree of n nodes is O(log n).
- Idea: Force the tree to be perfect.
  - Problem: can't have an arbitrary # of nodes.
  - Perfect binary trees only have 2^h-1 nodes
- So: relax the condition that the search tree be binary.
- As we'll see, this lets you have any number of nodes while keeping the leaves all at the same depth.

Global balance instead of the local balance of AVL trees.

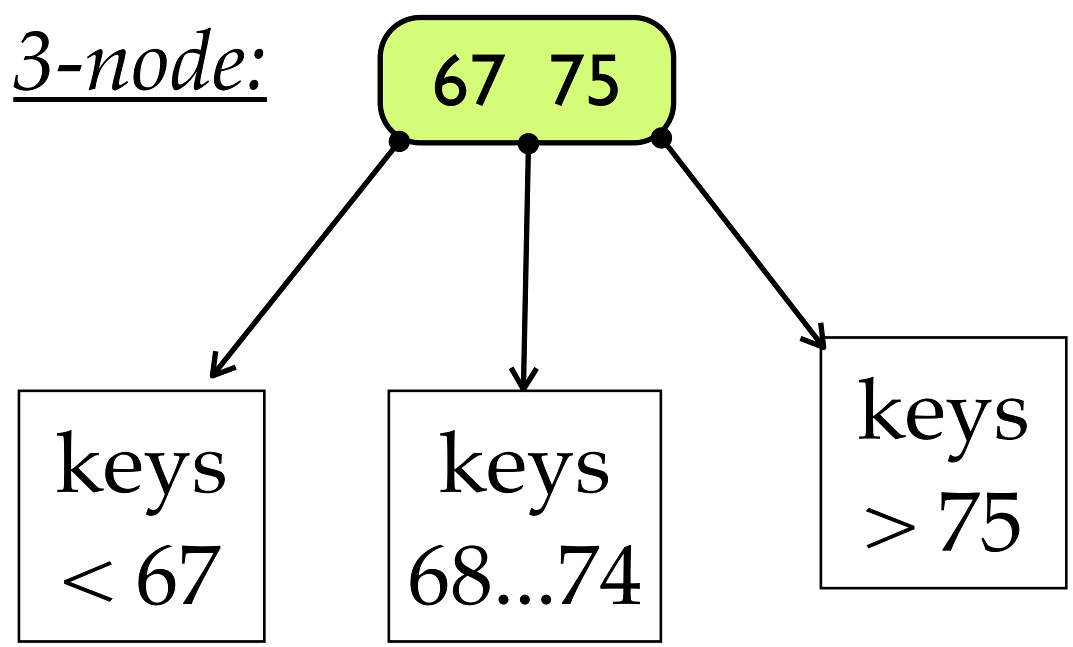
2,3 Trees

- All leaves are at the same level.
- Each internal node has either 2 or 3 children.
- If it has:
  - 2 children => it has 1 key
  - 3 children => it has 2 keys

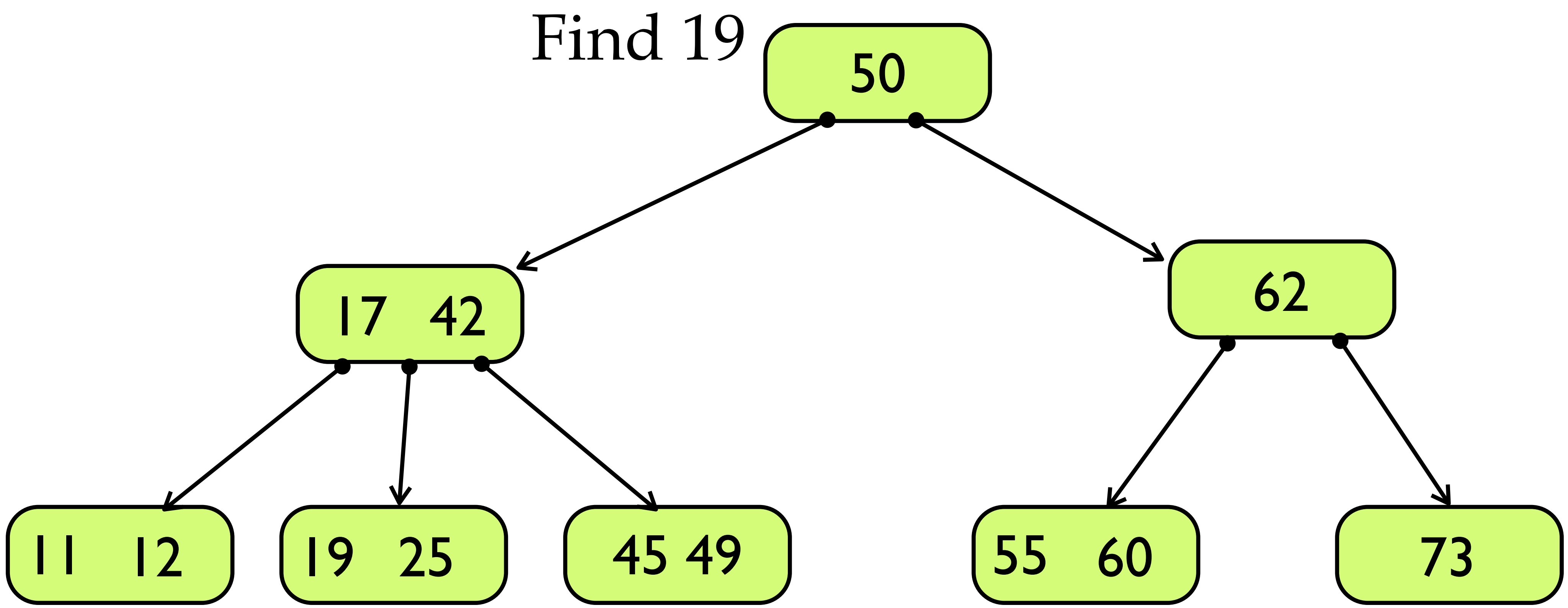
2-node:



3-node:

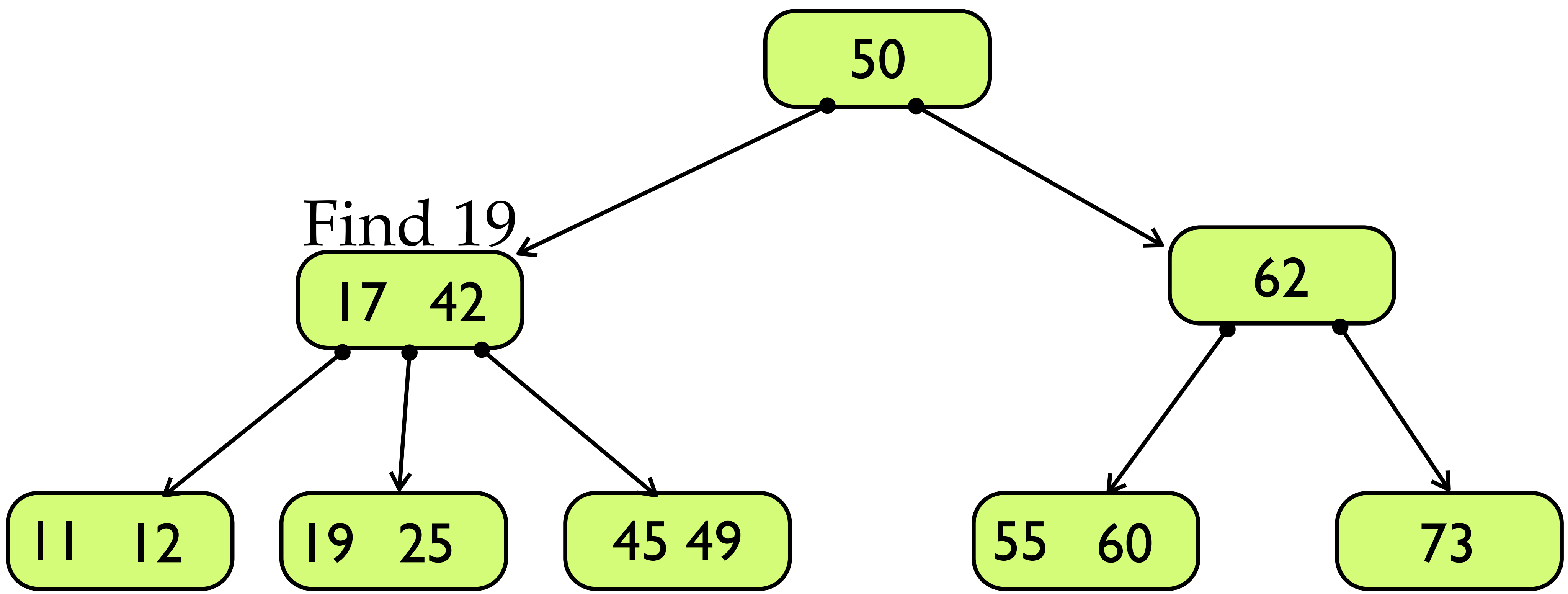


2,3 Tree Find (multiway searching)



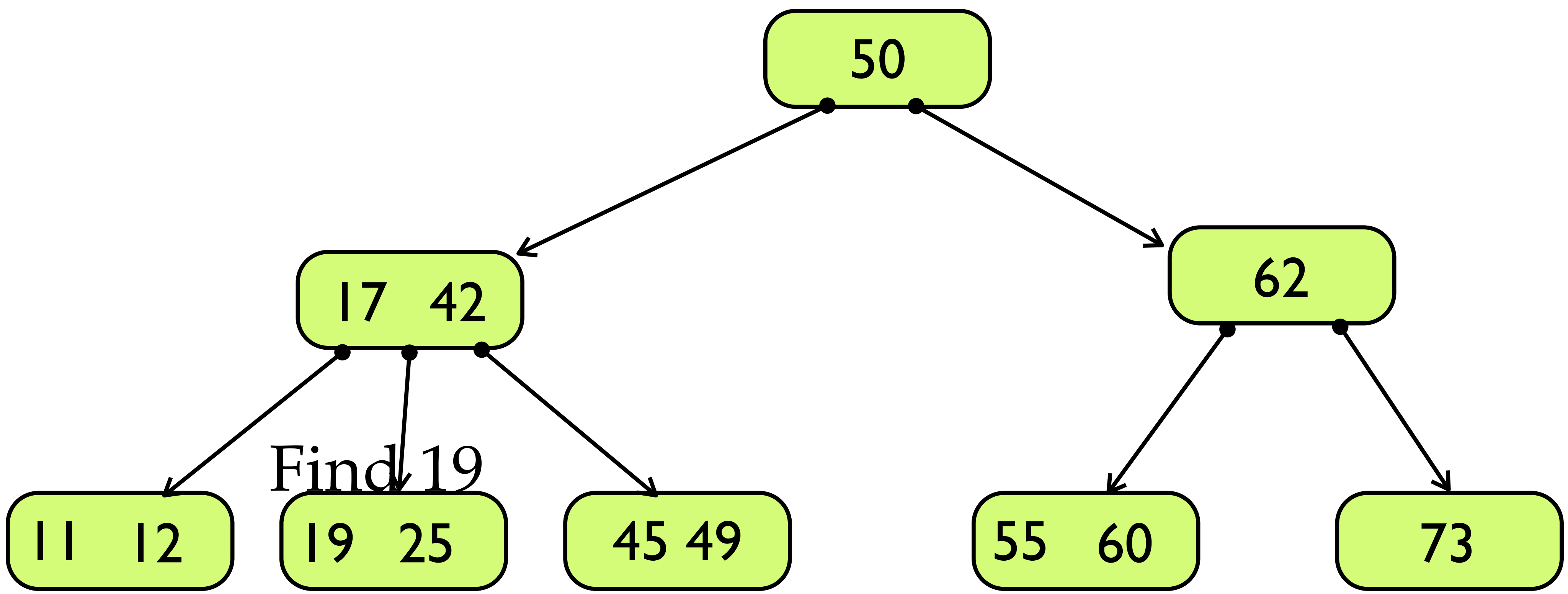
Standard BST-type walk down the tree.
At each node have to examine each key stored there.

2,3 Tree Find (multiway searching)



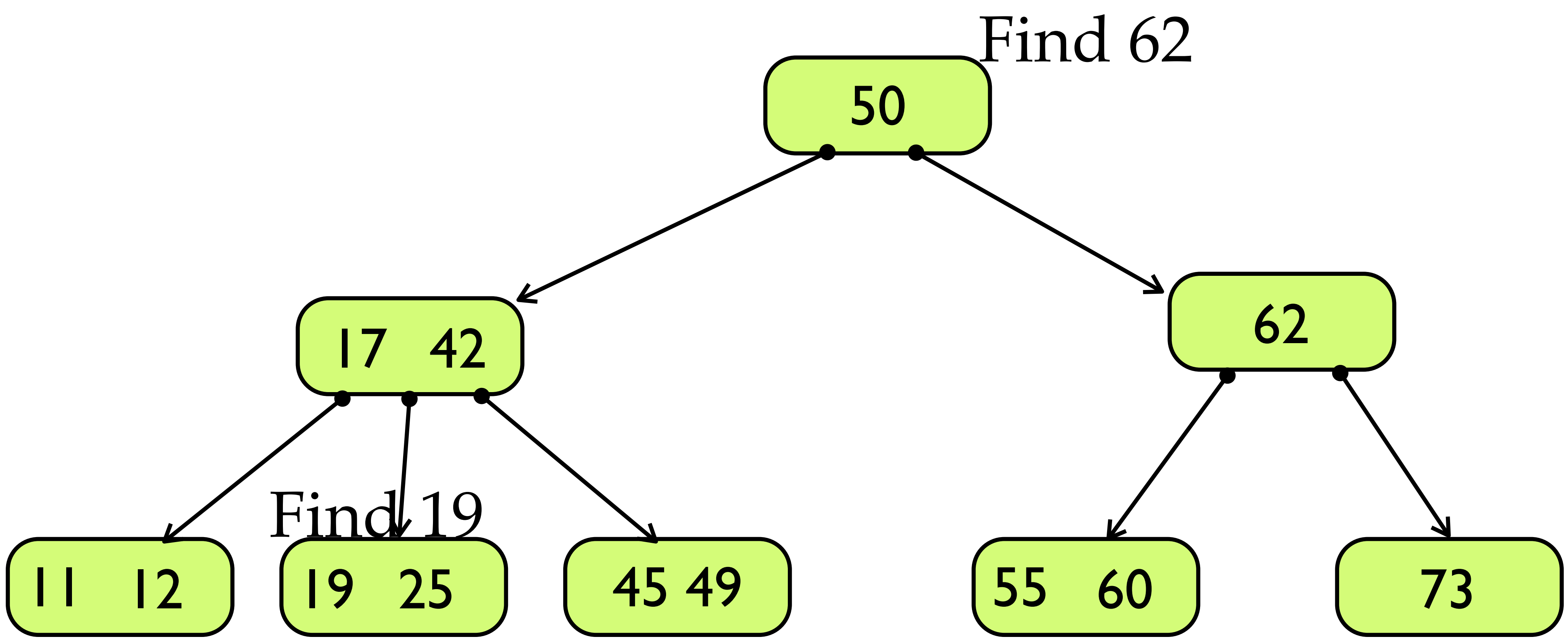
Standard BST-type walk down the tree.
At each node have to examine each key stored there.

2,3 Tree Find (multiway searching)



Standard BST-type walk down the tree.
At each node have to examine each key stored there.

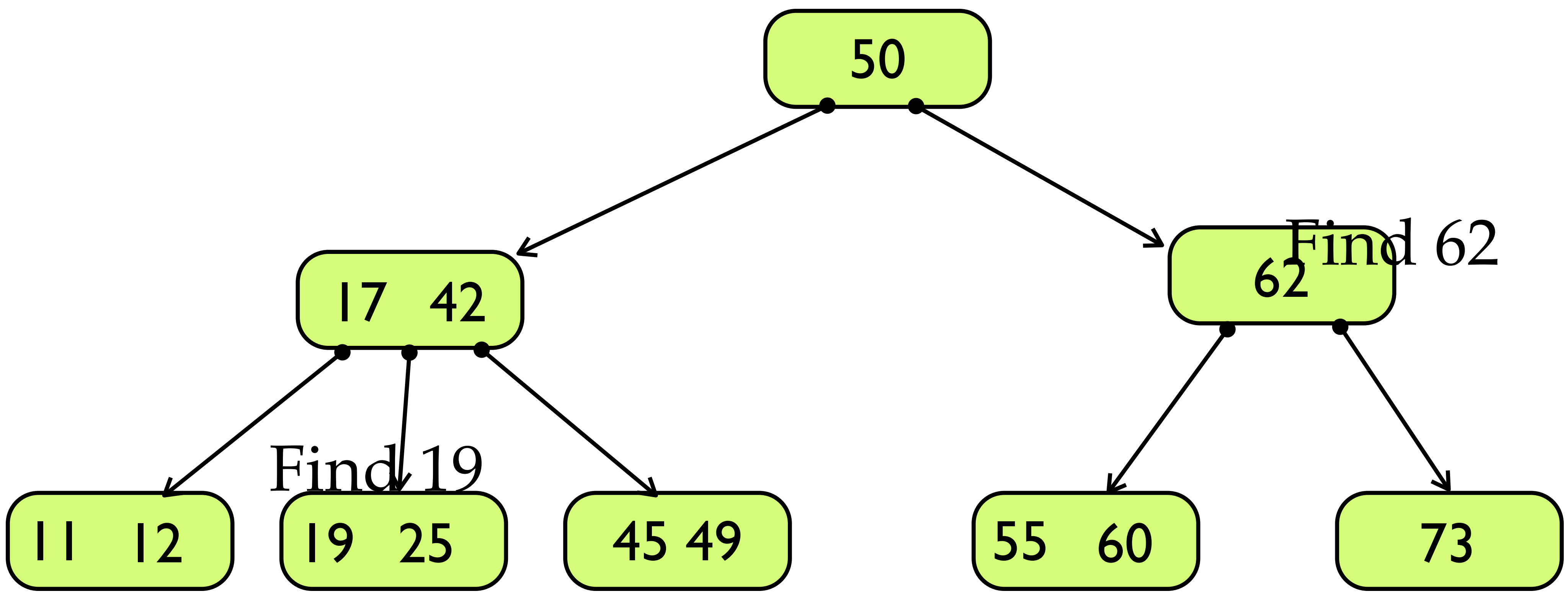
2,3 Tree Find (multiway searching)



Standard BST-type walk down the tree.

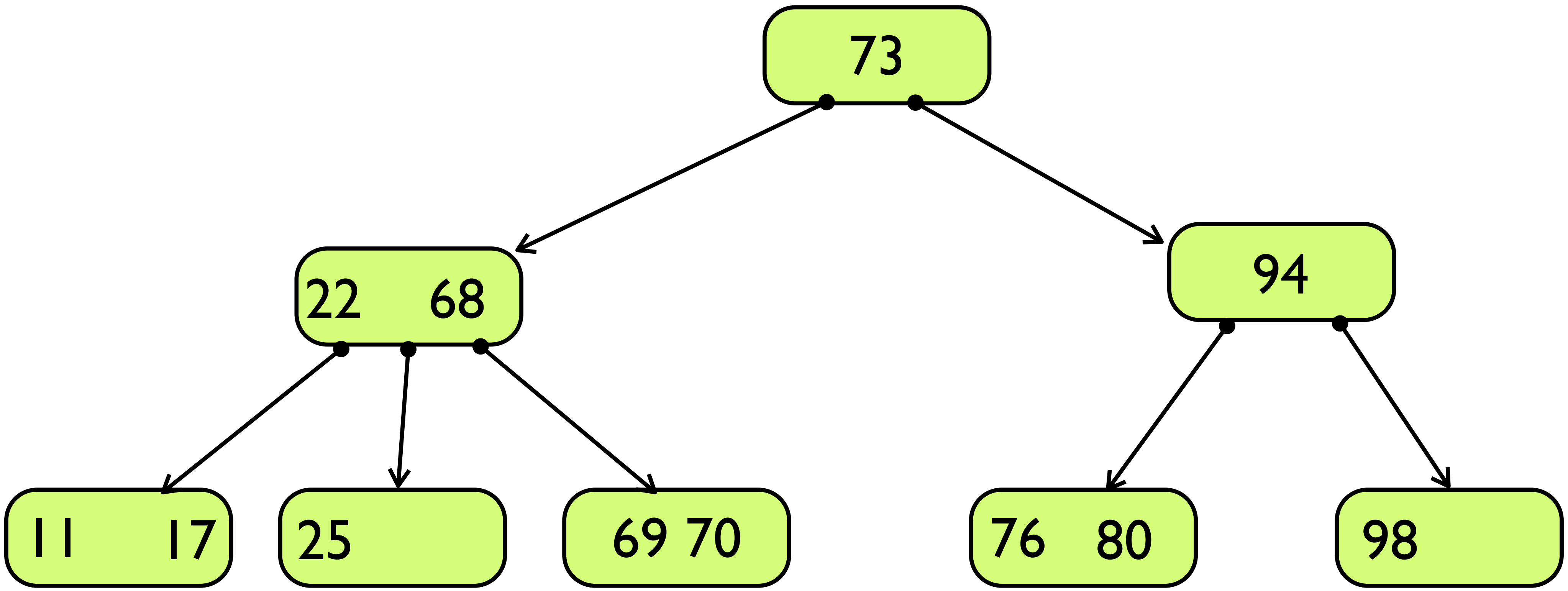
At each node have to examine each key stored there.

2,3 Tree Find (multiway searching)

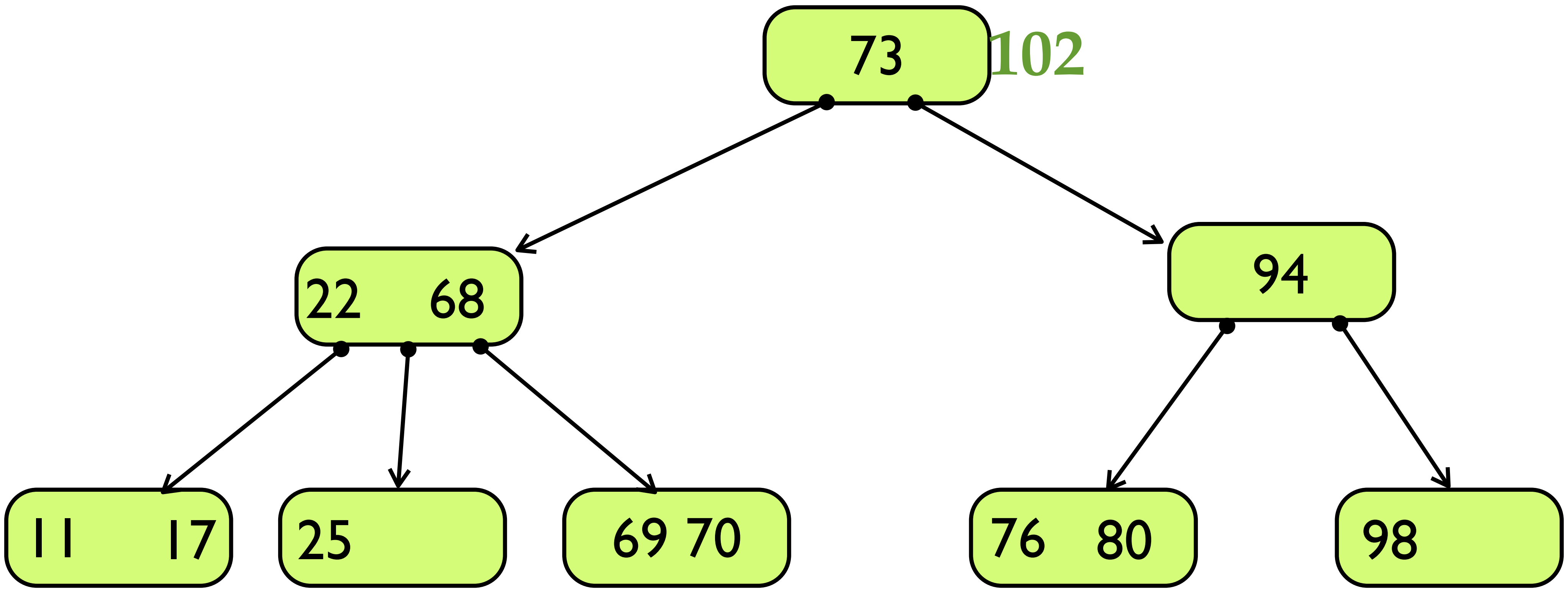


Standard BST-type walk down the tree.
At each node have to examine each key stored there.

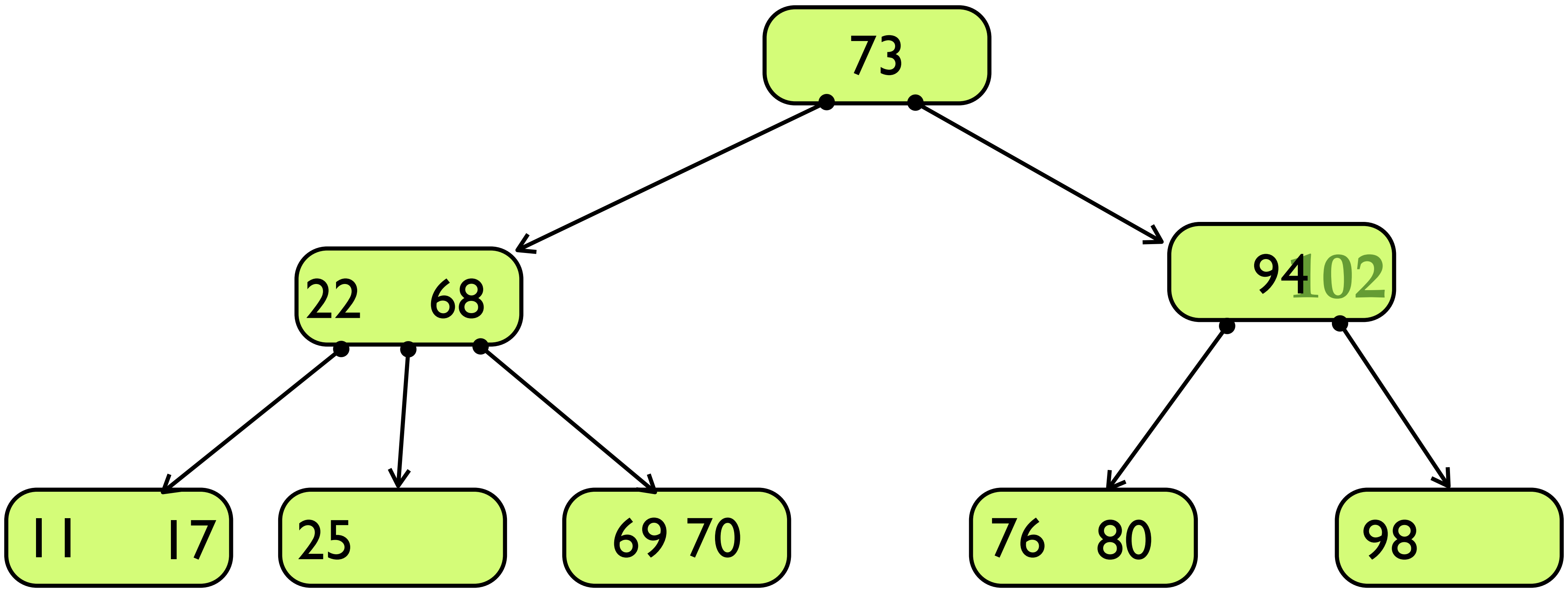
2,3 Tree Insertion



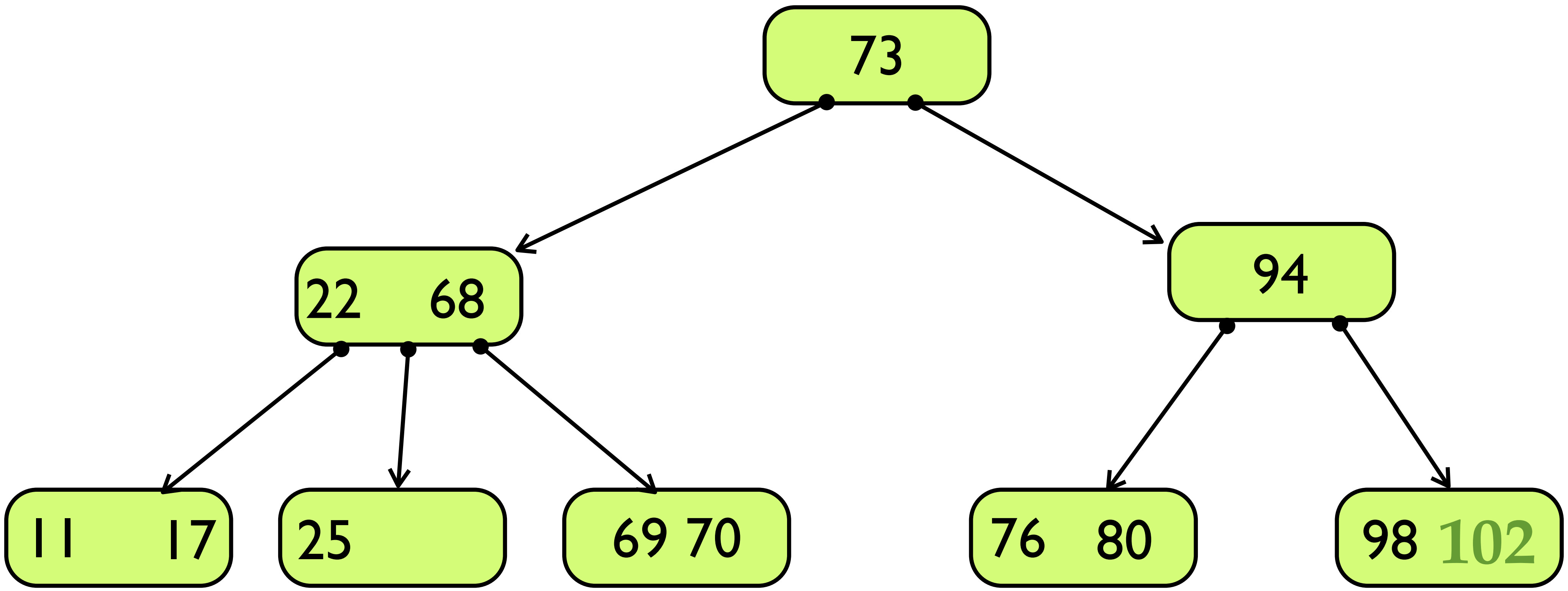
2,3 Tree Insertion



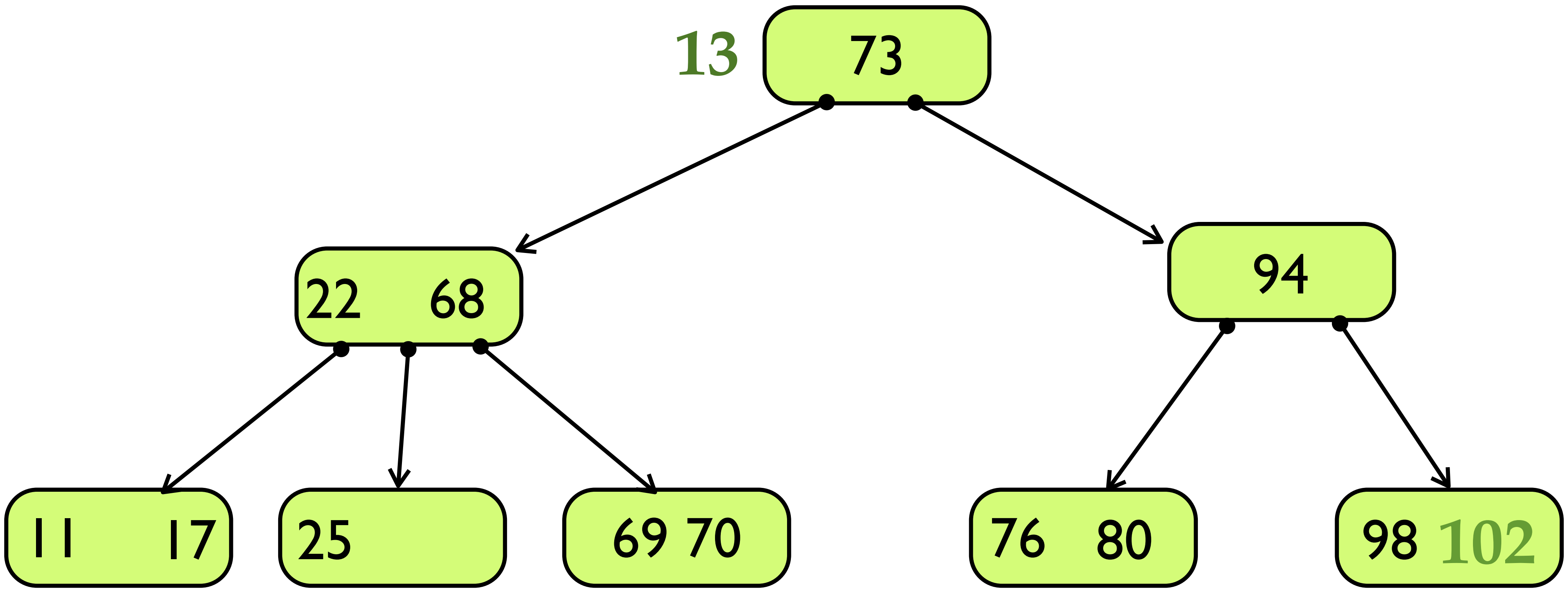
2,3 Tree Insertion



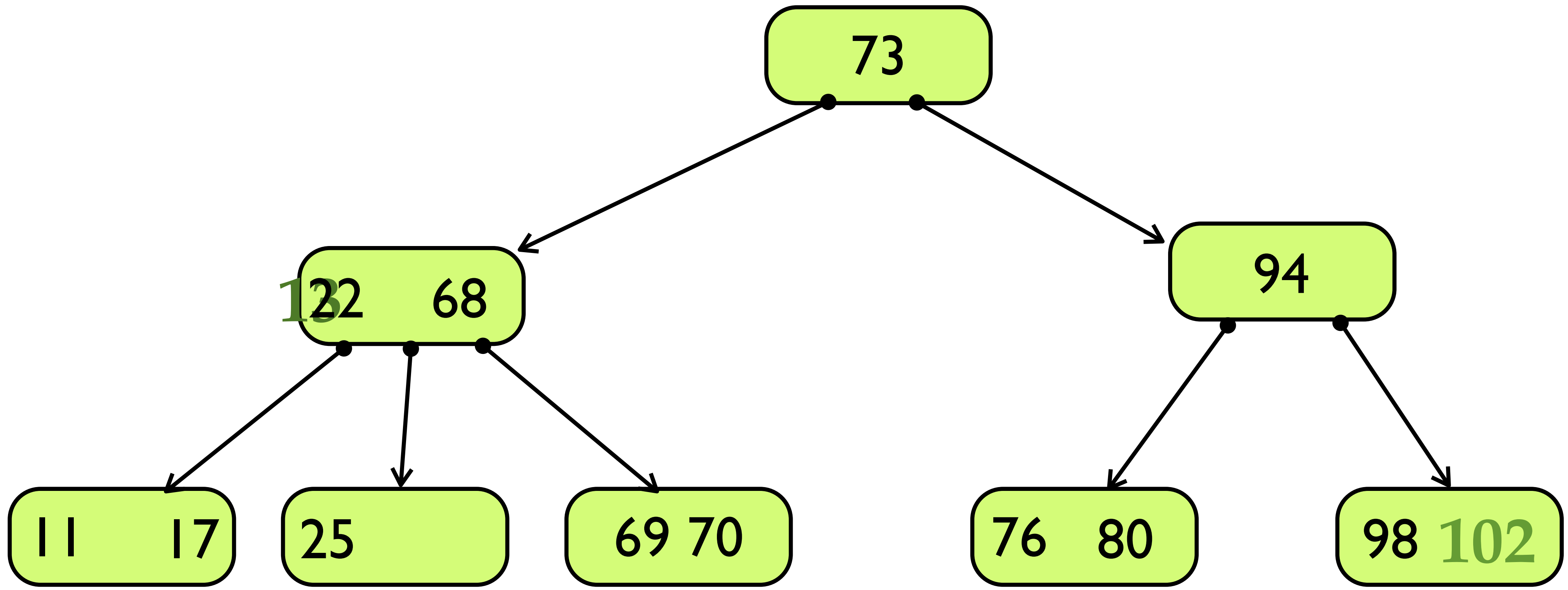
2,3 Tree Insertion



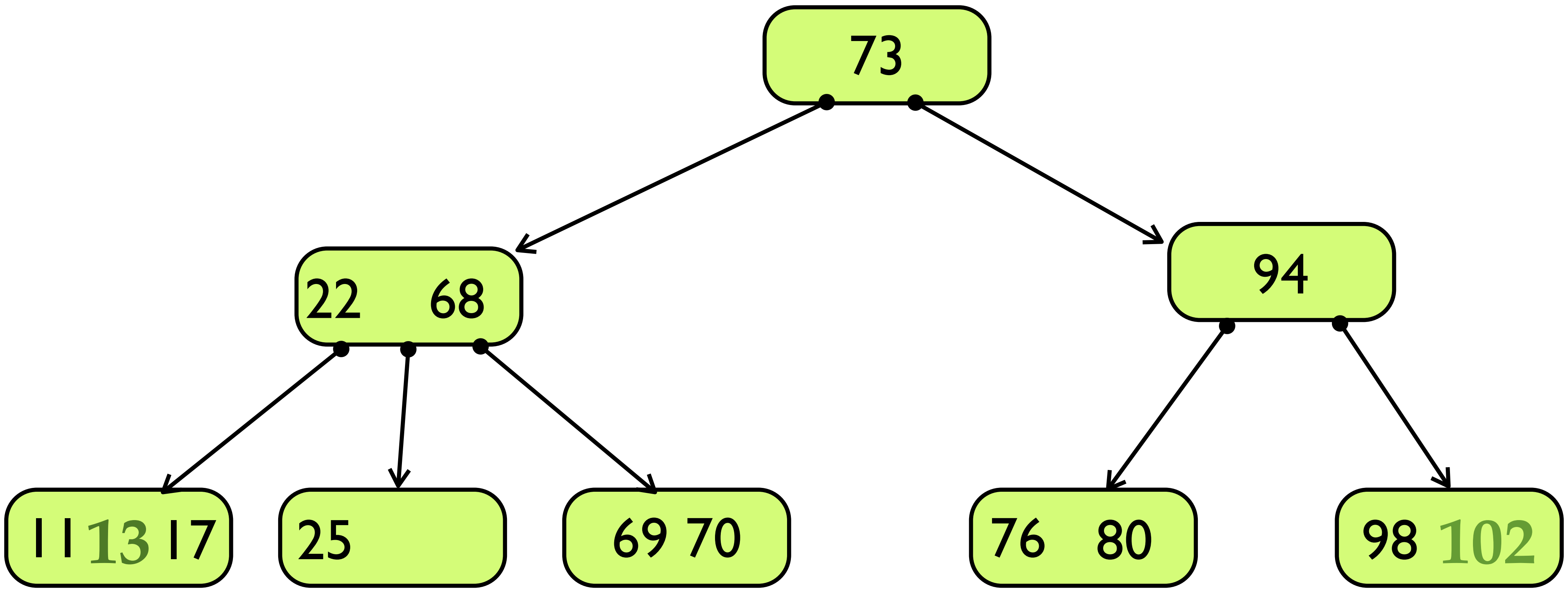
2,3 Tree Insertion



2,3 Tree Insertion

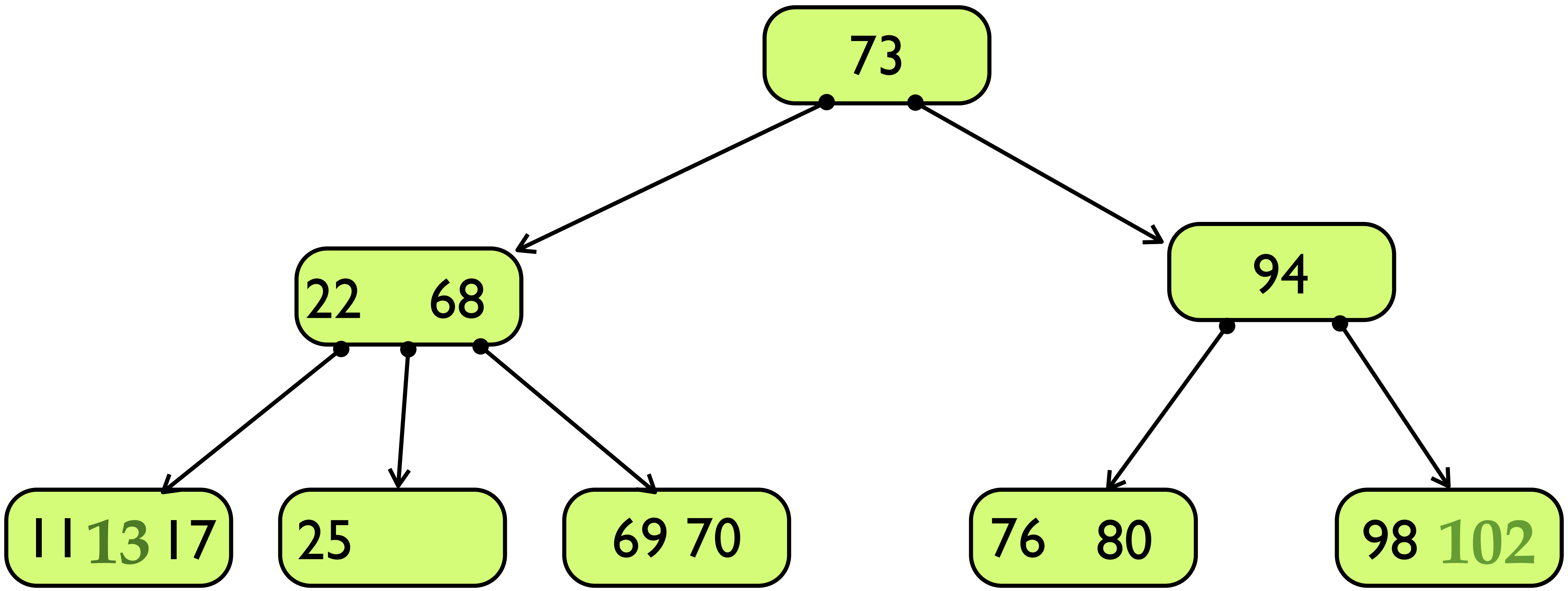


2,3 Tree Insertion



Overflow!

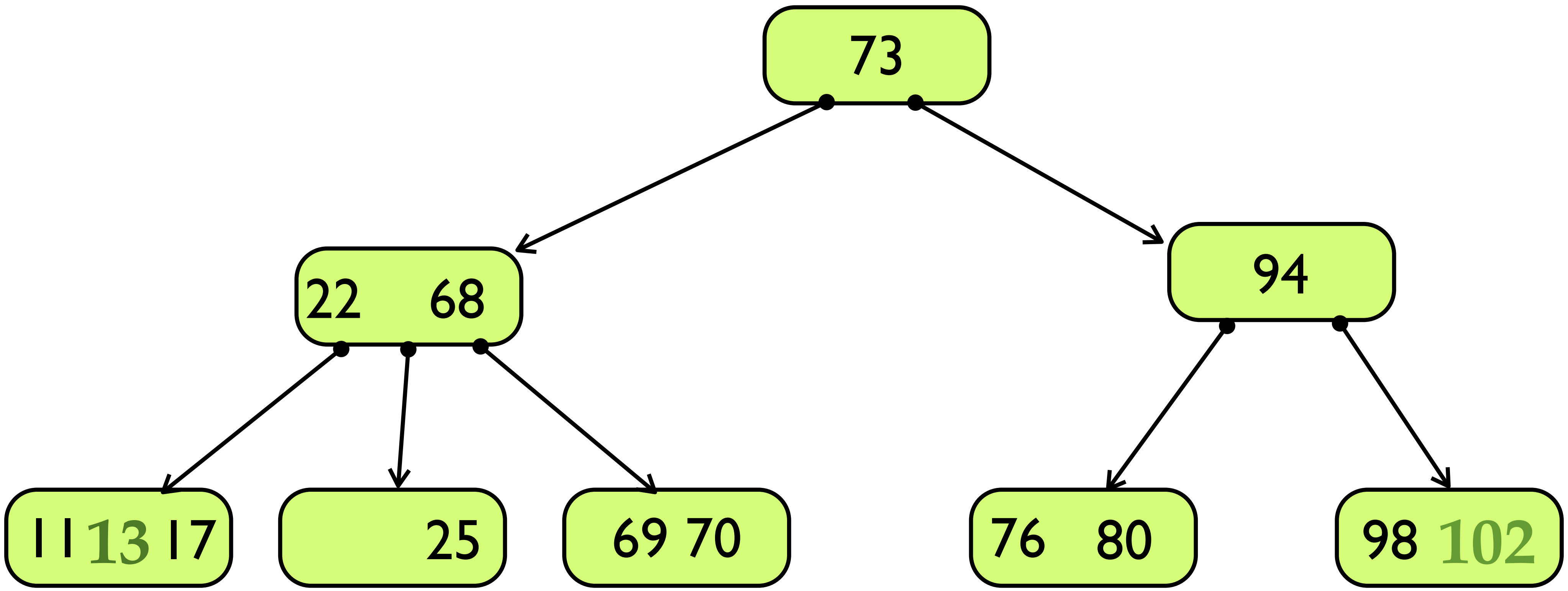
2,3 Tree Insertion



Overflow!

Try **Key Rotation:** Look for left or right sibling with some space, move a parent key into it, and a child key into the parent

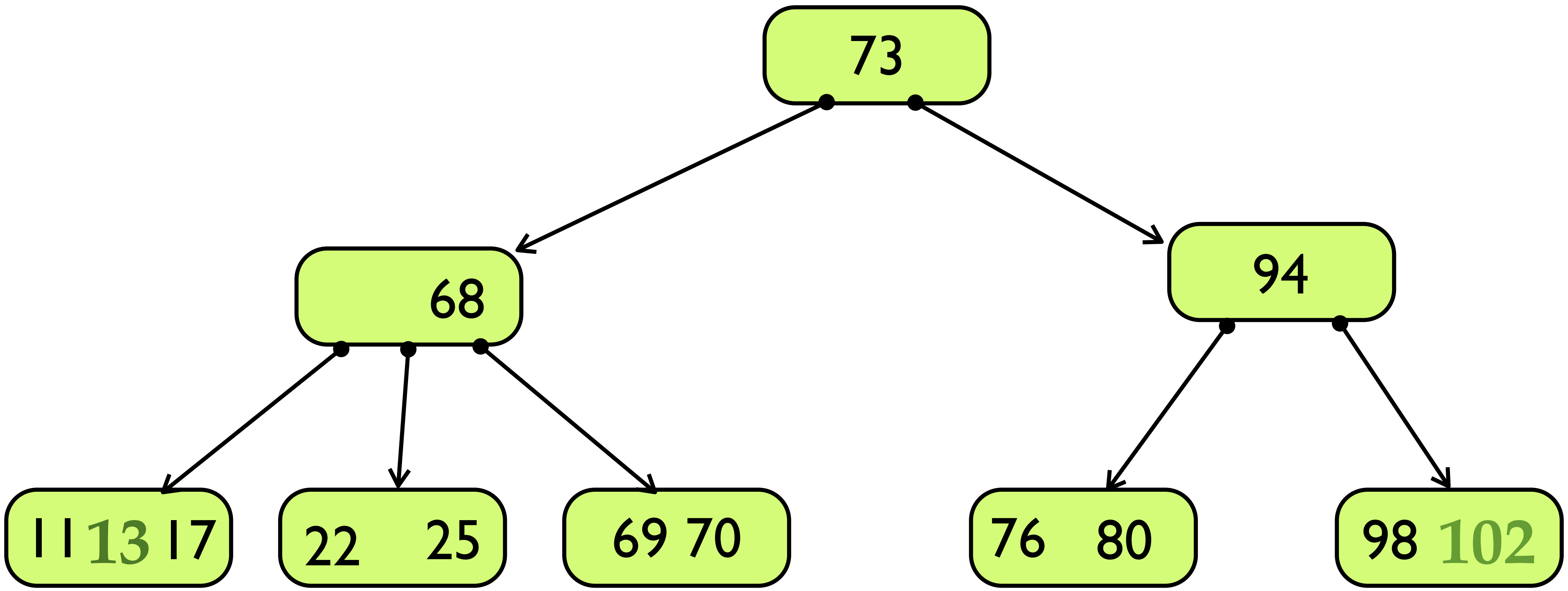
2,3 Tree Insertion



Overflow!

Try **Key Rotation:** Look for left or right sibling with some space, move a parent key into it, and a child key into the parent

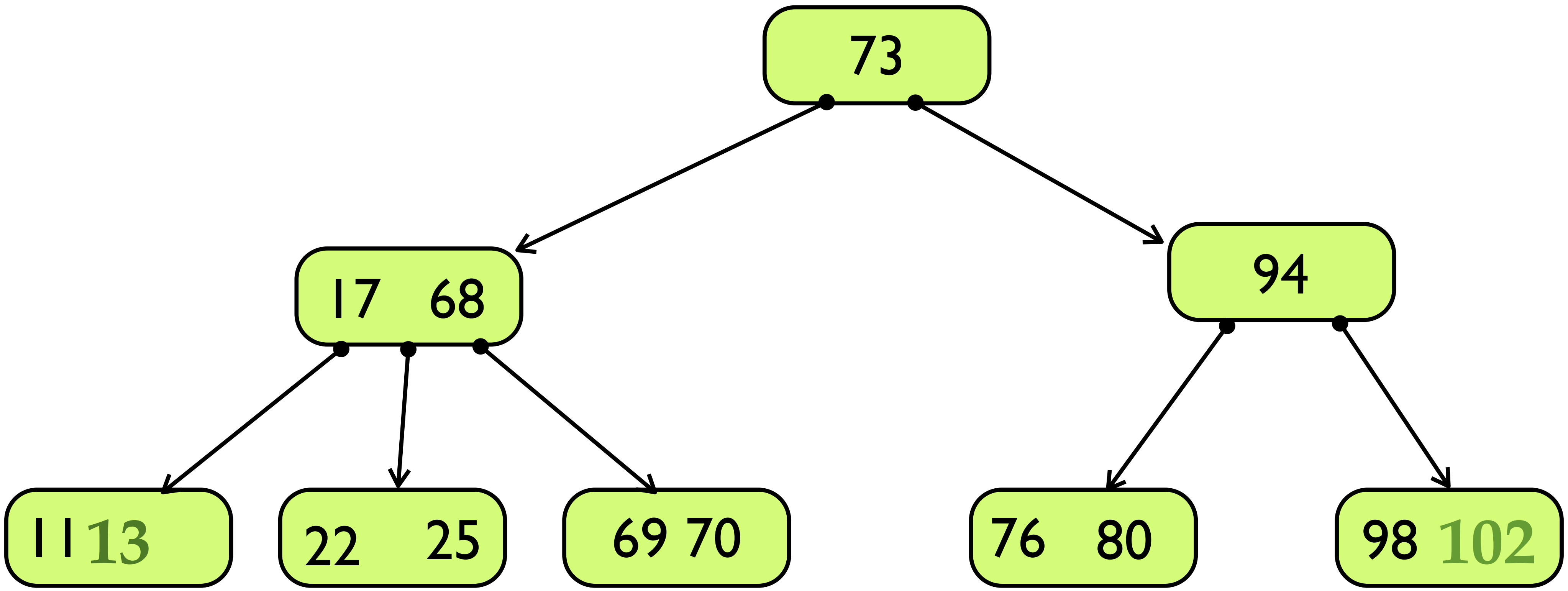
2,3 Tree Insertion



Overflow!

Try **Key Rotation:** Look for left or right sibling with some space, move a parent key into it, and a child key into the parent

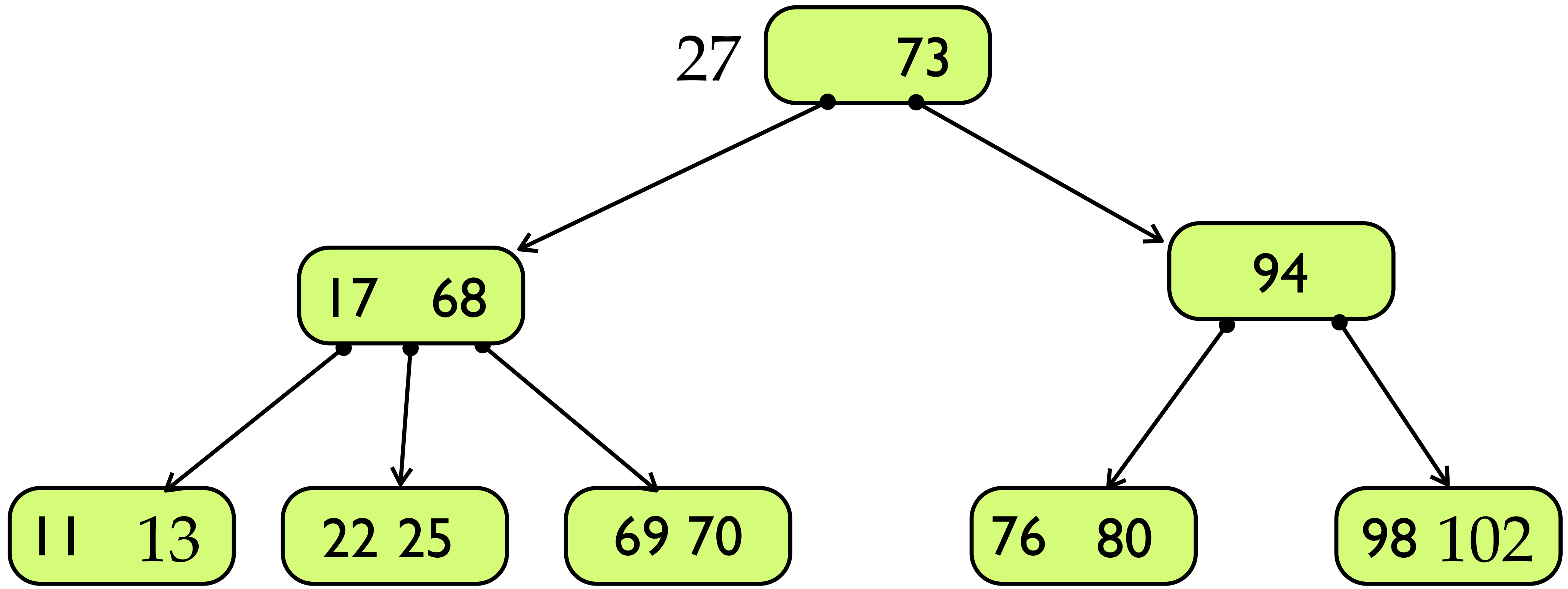
2,3 Tree Insertion



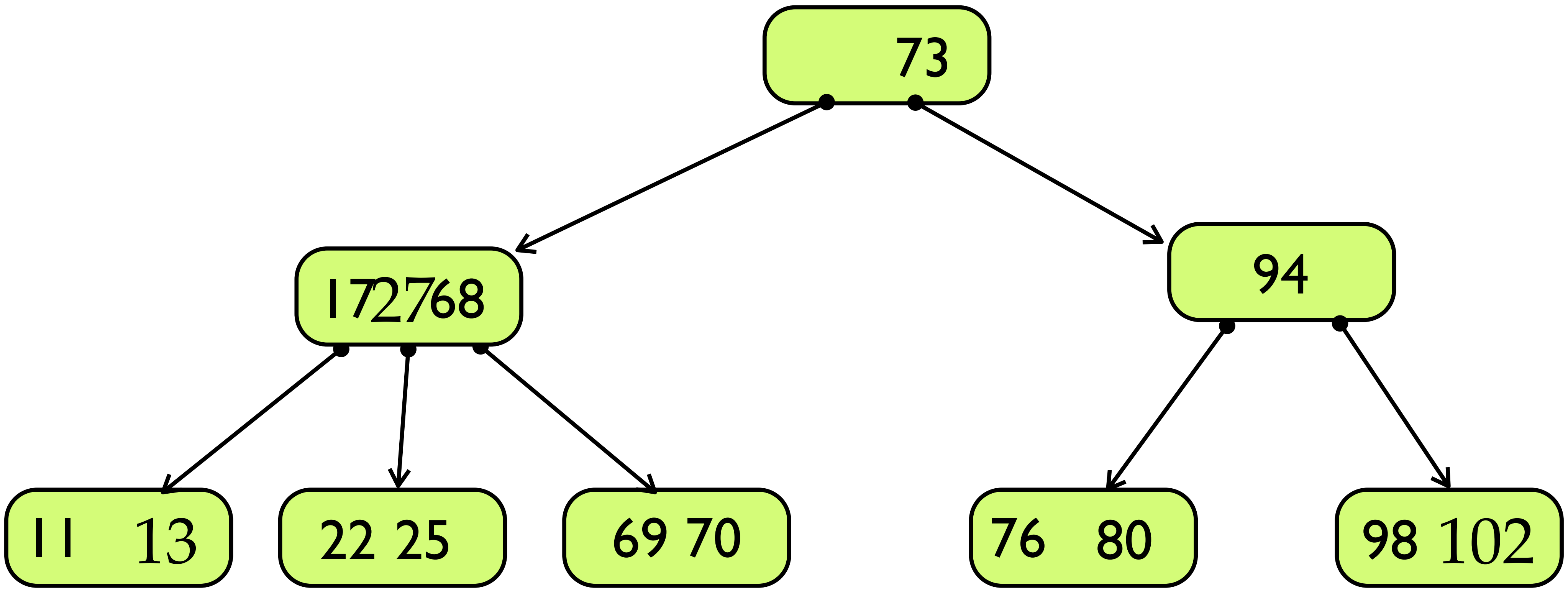
Overflow!

Try **Key Rotation:** Look for left or right sibling with some space, move a parent key into it, and a child key into the parent

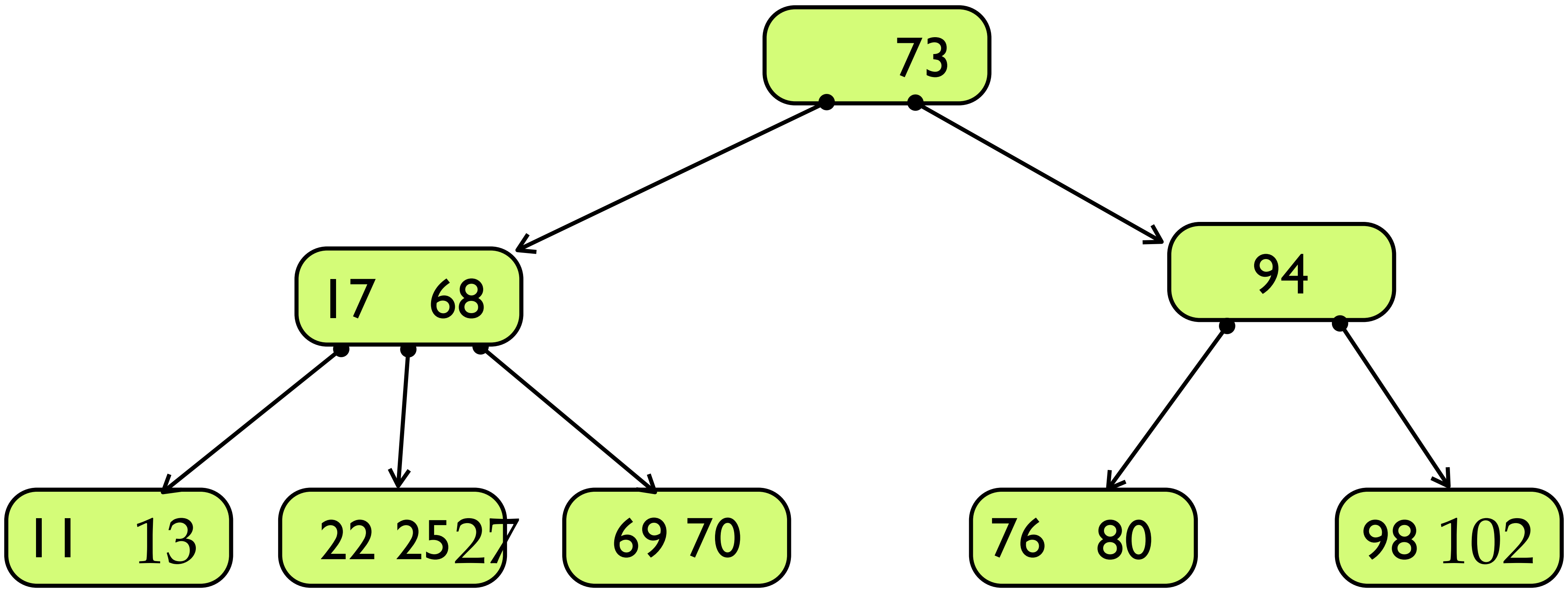
2,3 Tree Insertion – When key rotation fails



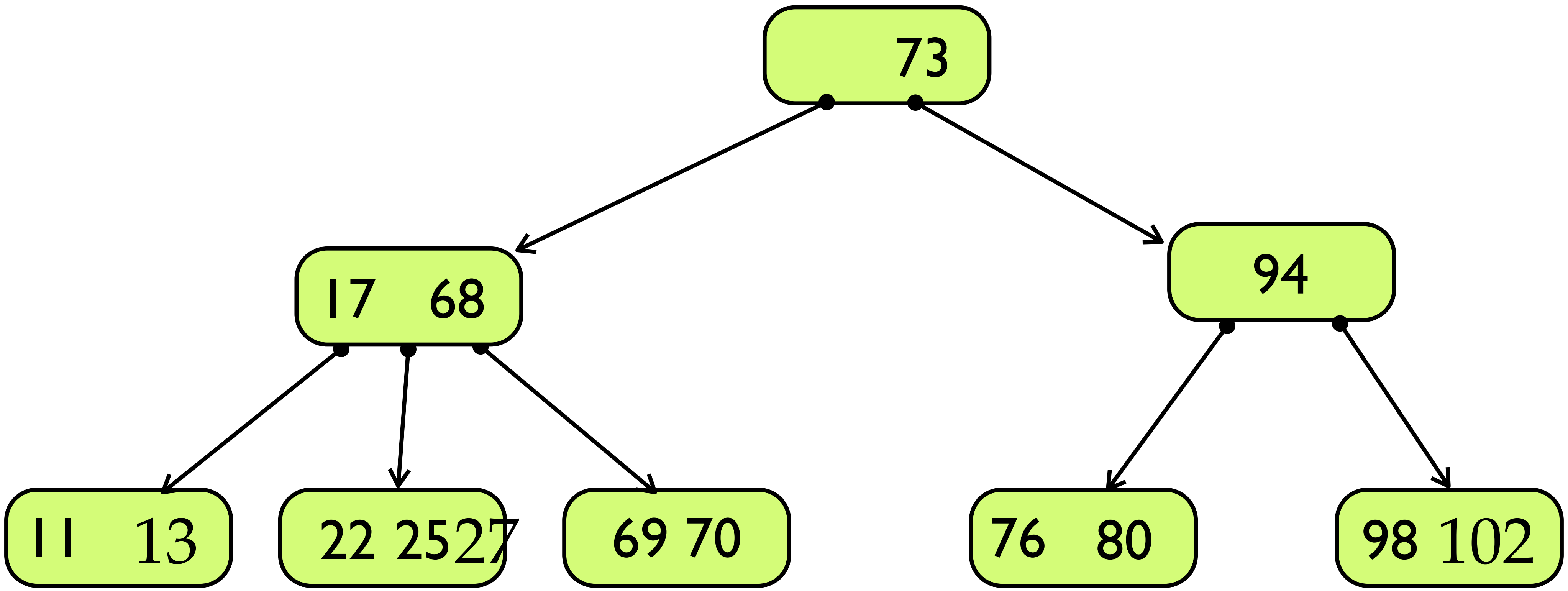
2,3 Tree Insertion – When key rotation fails



2,3 Tree Insertion – When key rotation fails



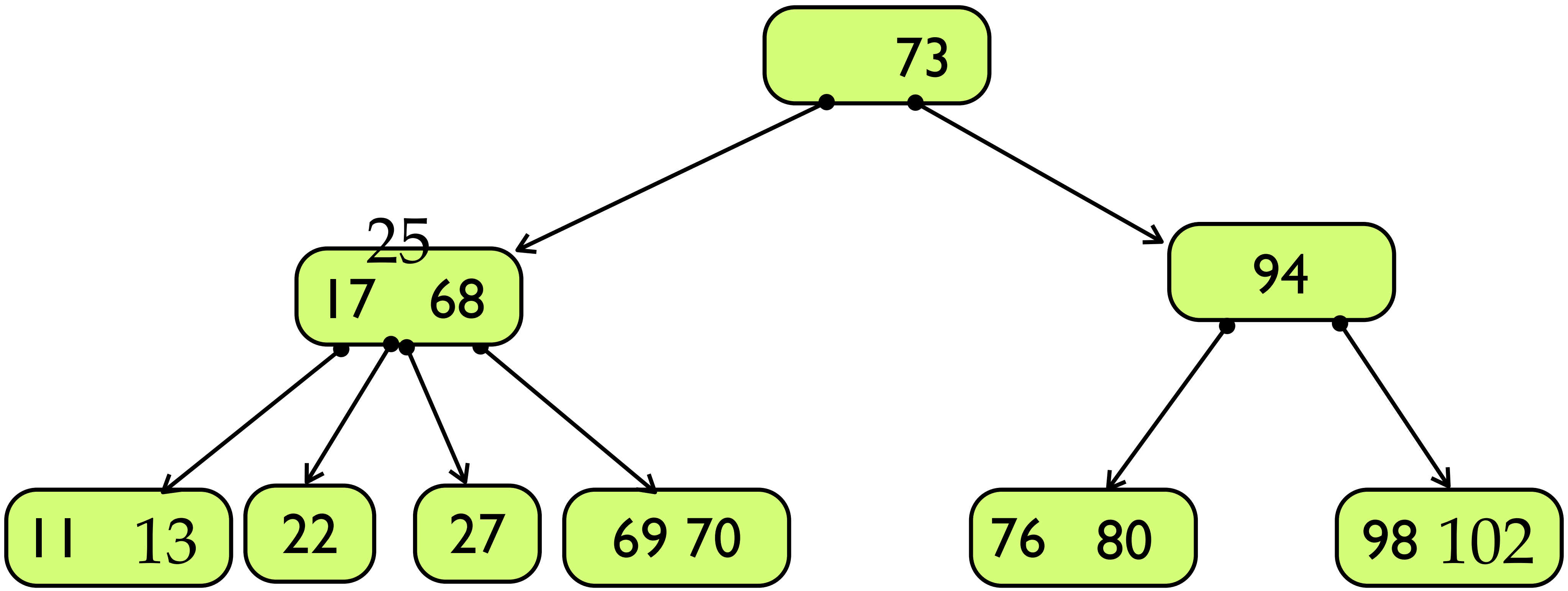
2,3 Tree Insertion – When key rotation fails



Overflow!

If both siblings are filled, you have to split the node.

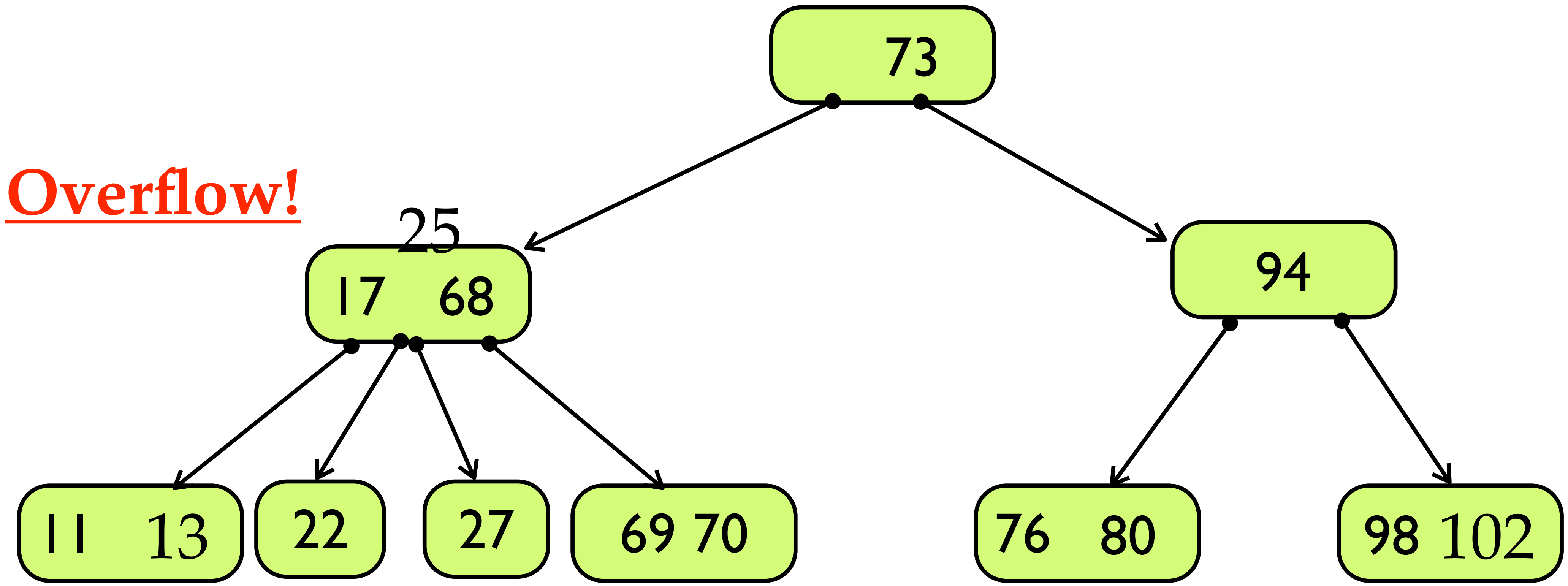
2,3 Tree Insertion – When key rotation fails



Overflow!

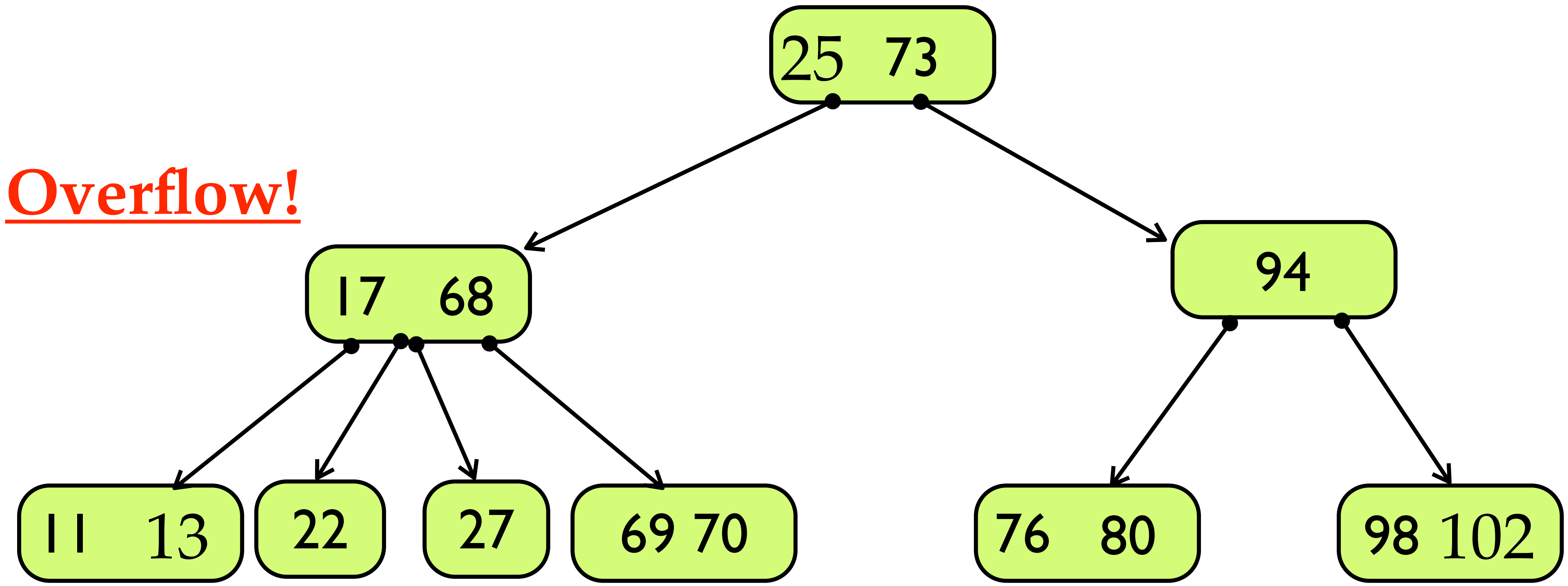
If both siblings are filled, you have to split the node.

2,3 Tree Insertion – When key rotation fails



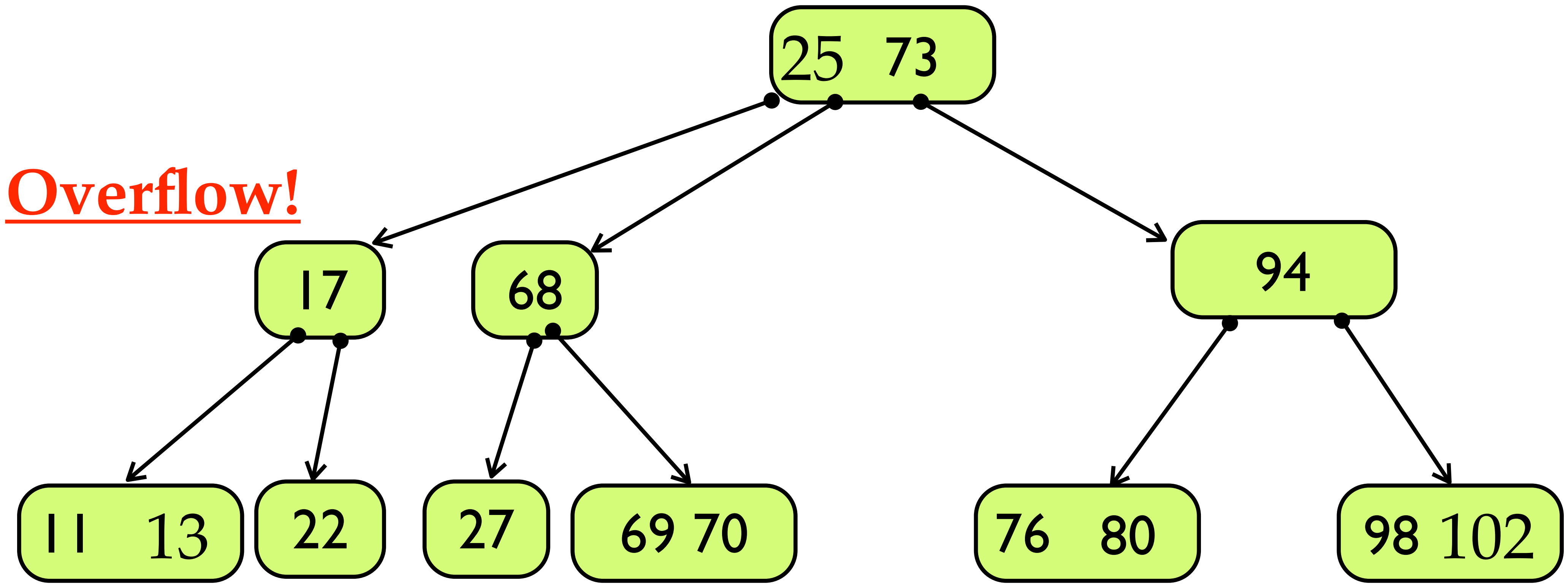
May have to recursively split nodes, working back to the root.

2,3 Tree Insertion – When key rotation fails



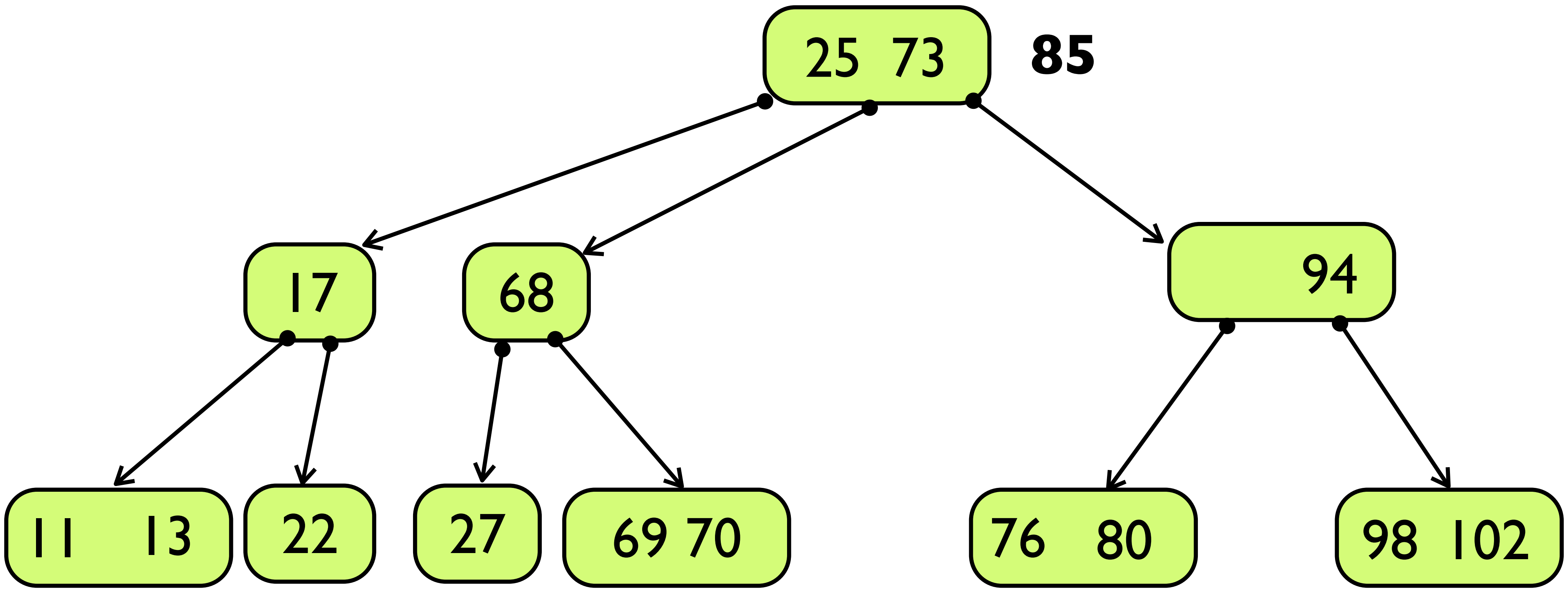
May have to recursively split nodes, working back to the root.

2,3 Tree Insertion – When key rotation fails

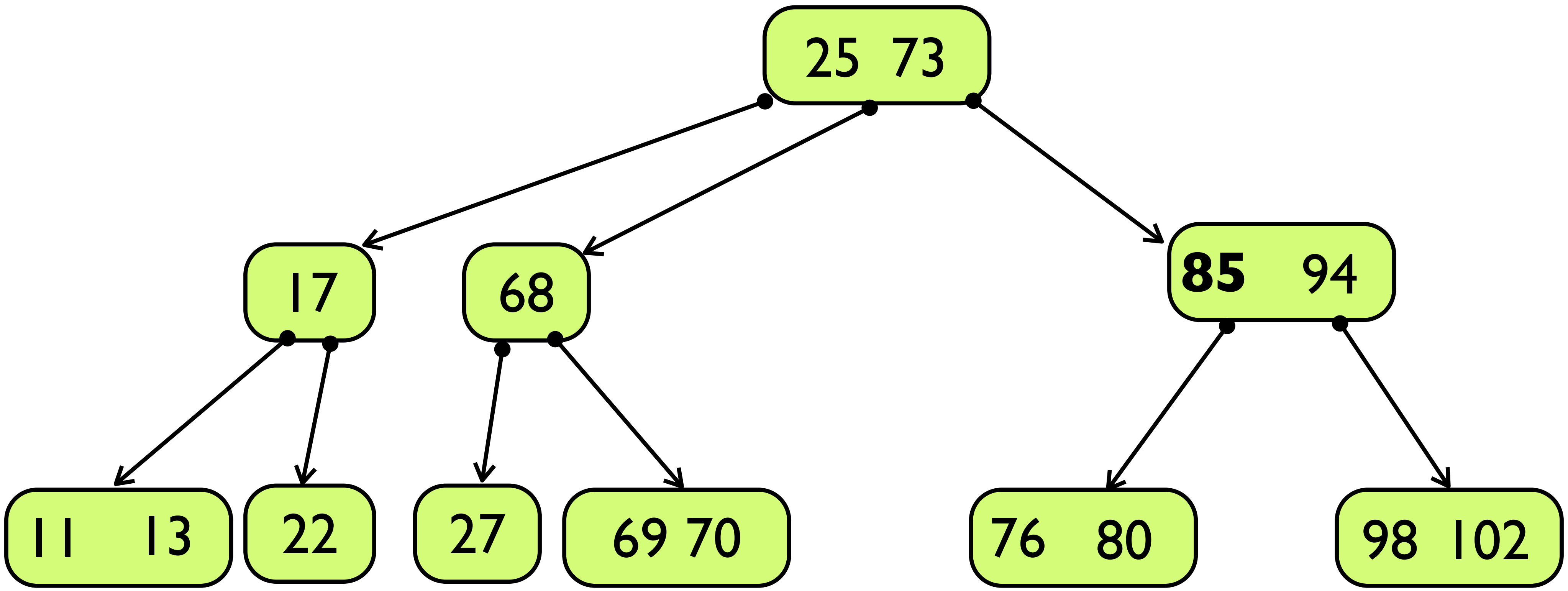


May have to recursively split nodes, working back to the root.

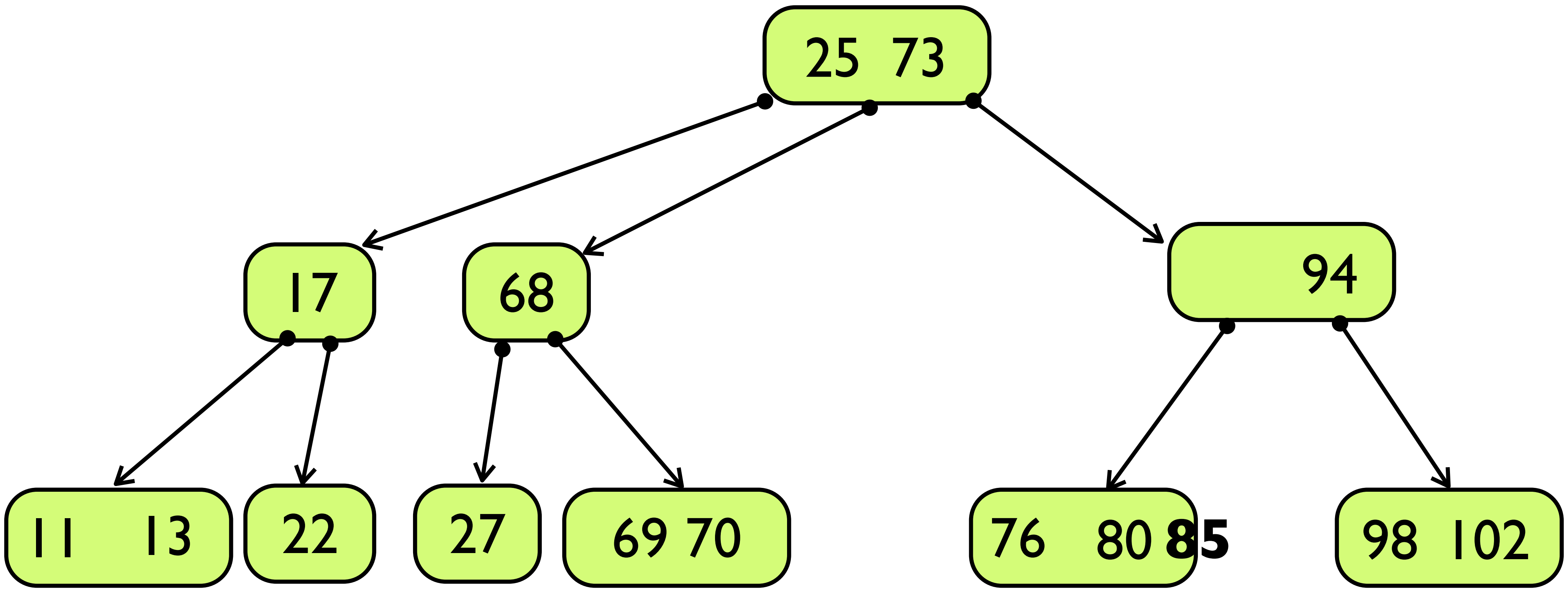
2,3 Tree Insertion – Another Splitting Example



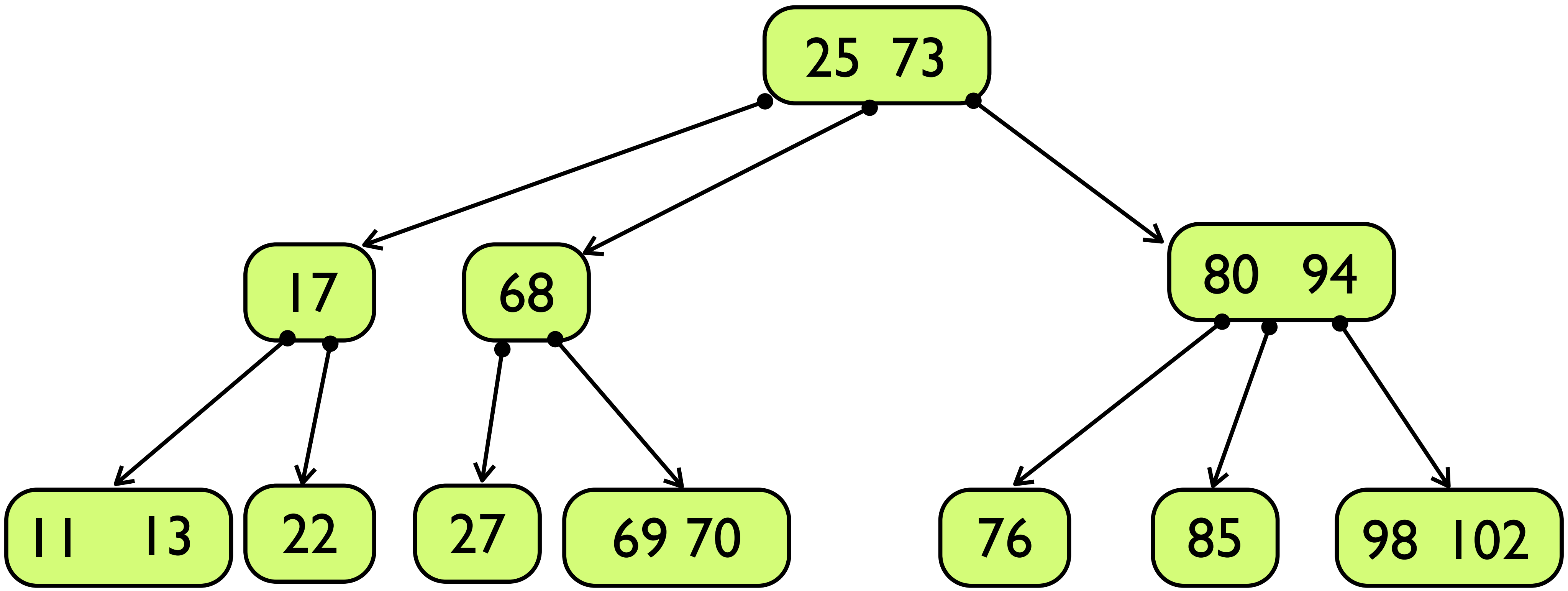
2,3 Tree Insertion – Another Splitting Example



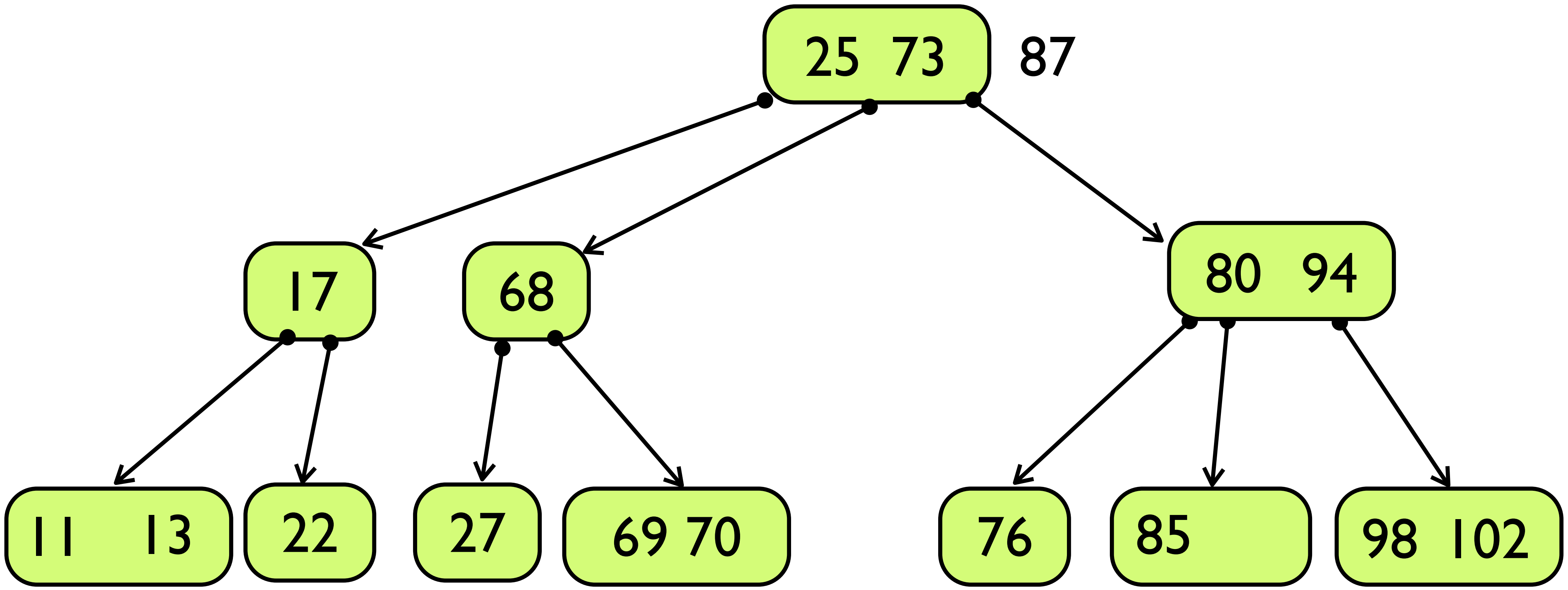
2,3 Tree Insertion – Another Splitting Example



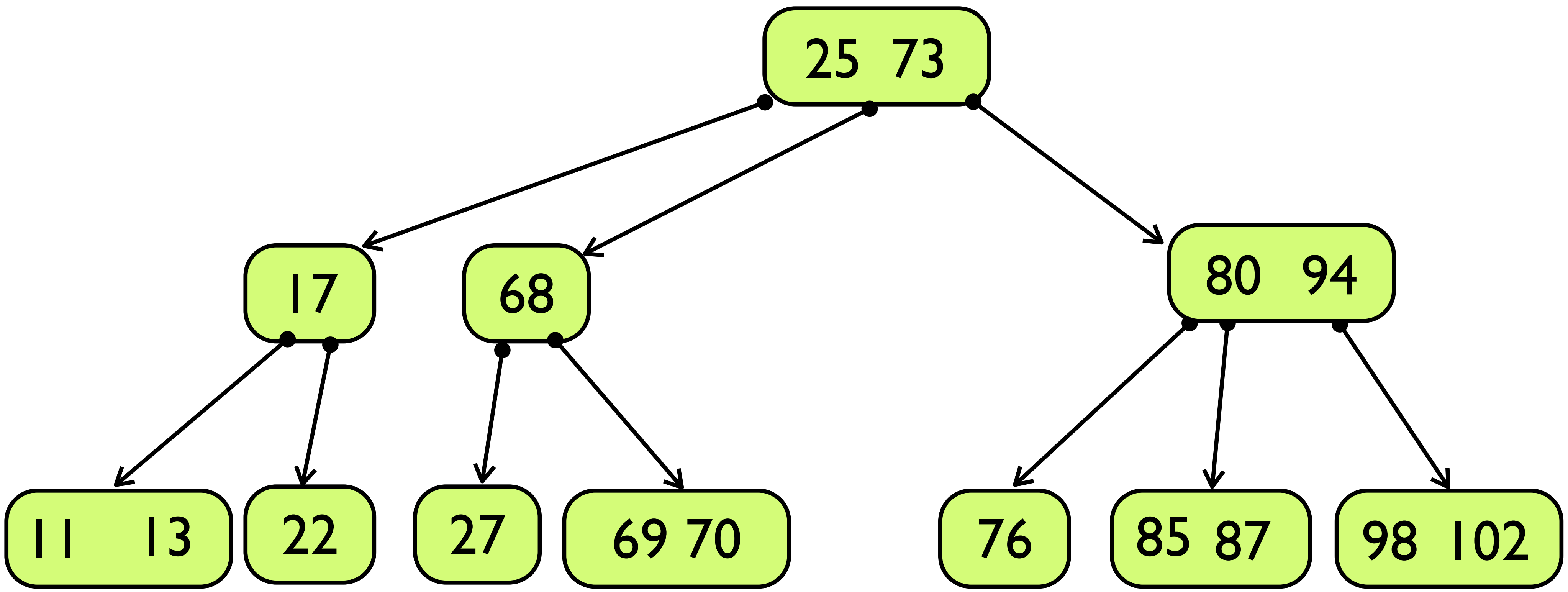
2,3 Tree Insertion – Another Splitting Example



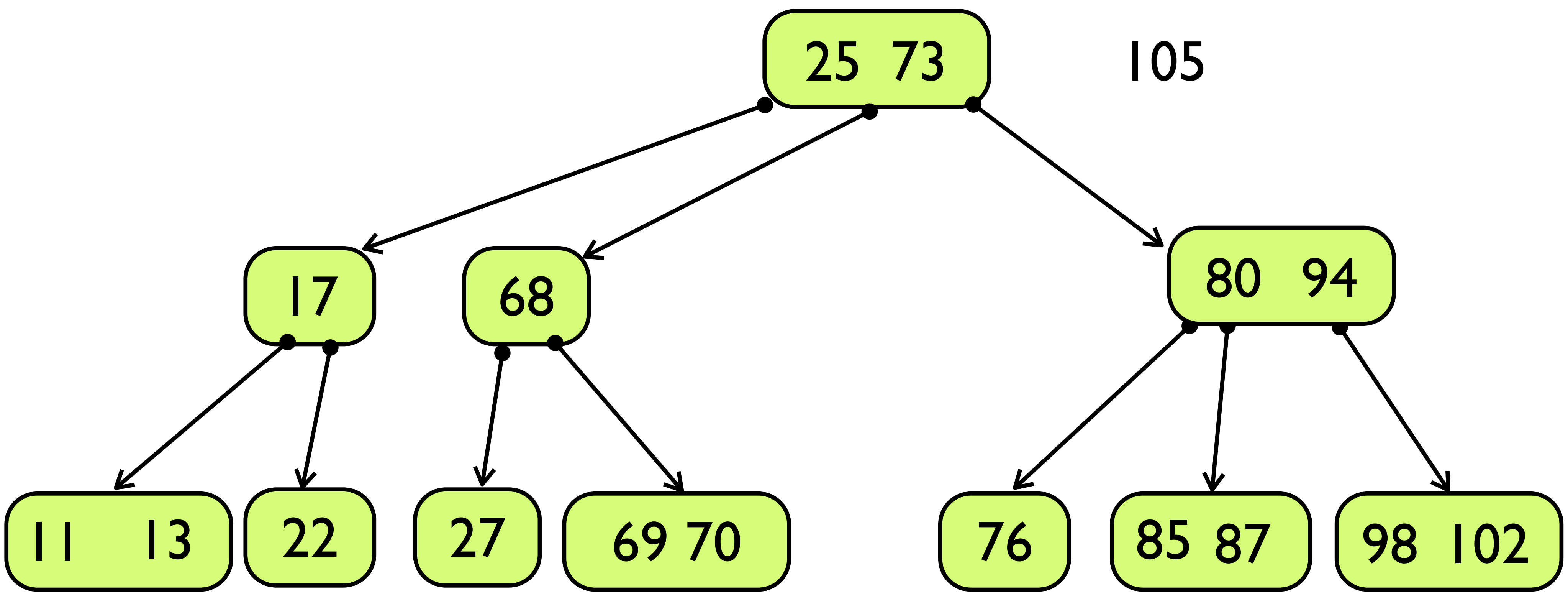
2,3 Tree Insertion – Splitting at the root



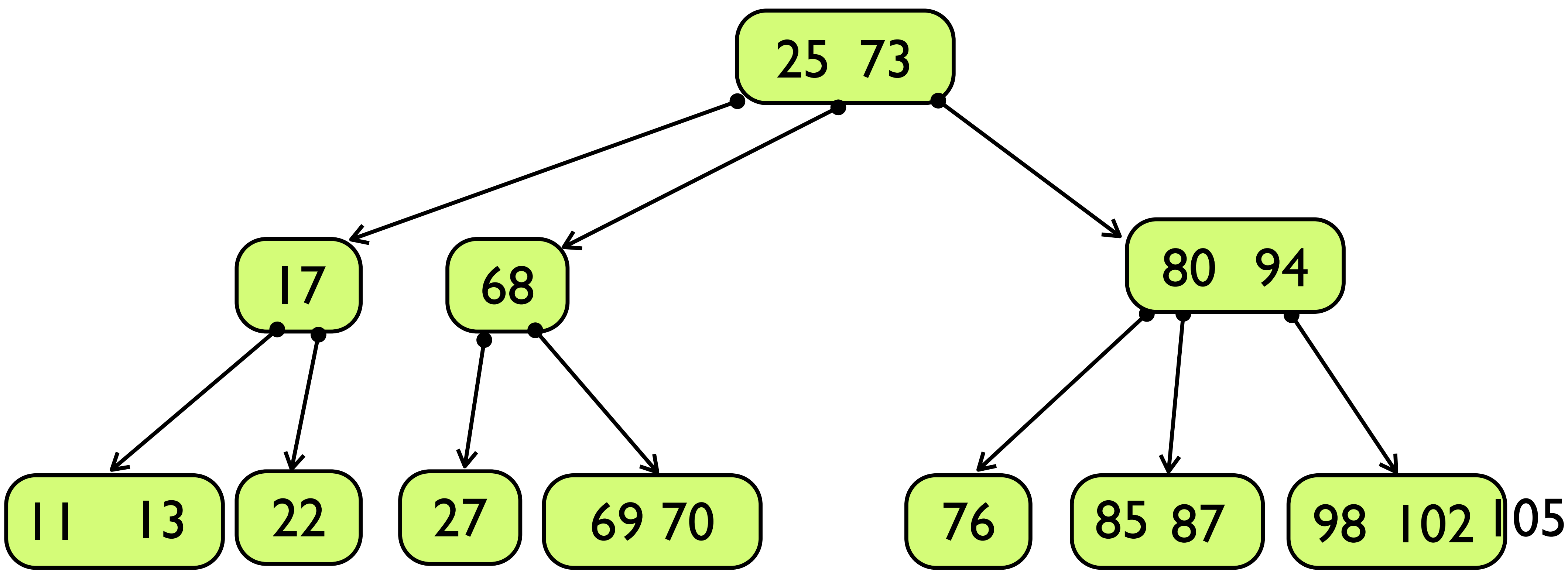
2,3 Tree Insertion – Splitting at the root



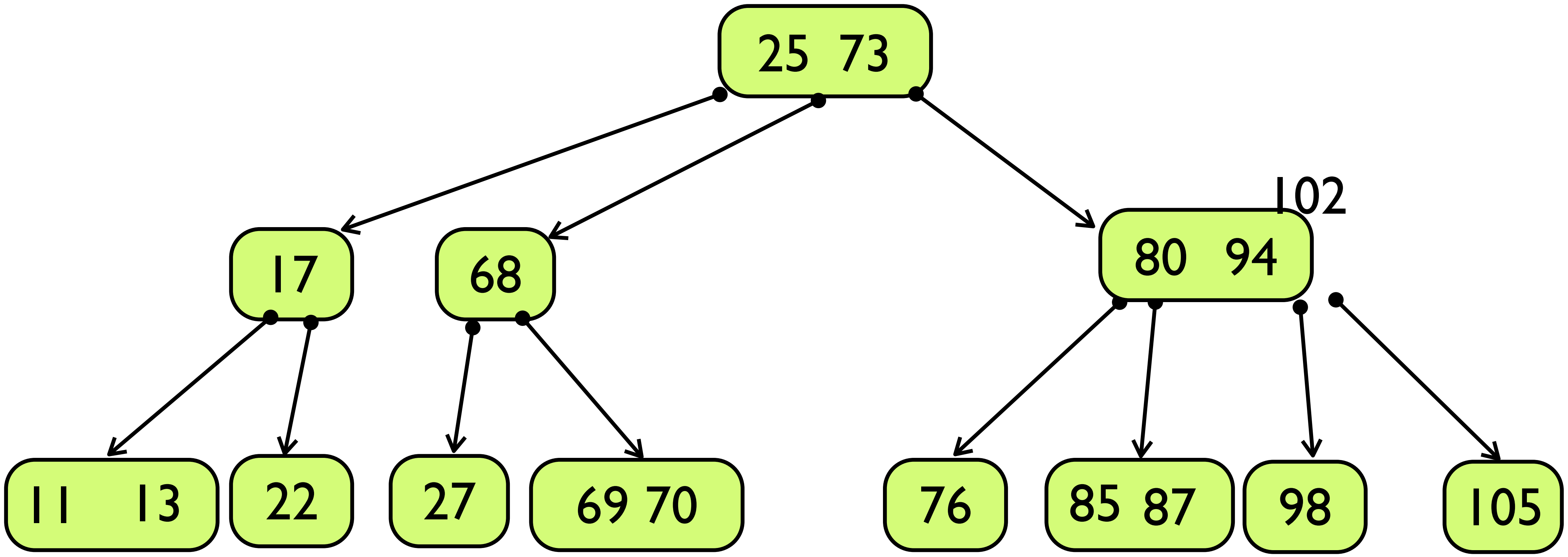
2,3 Tree Insertion – Splitting at the root



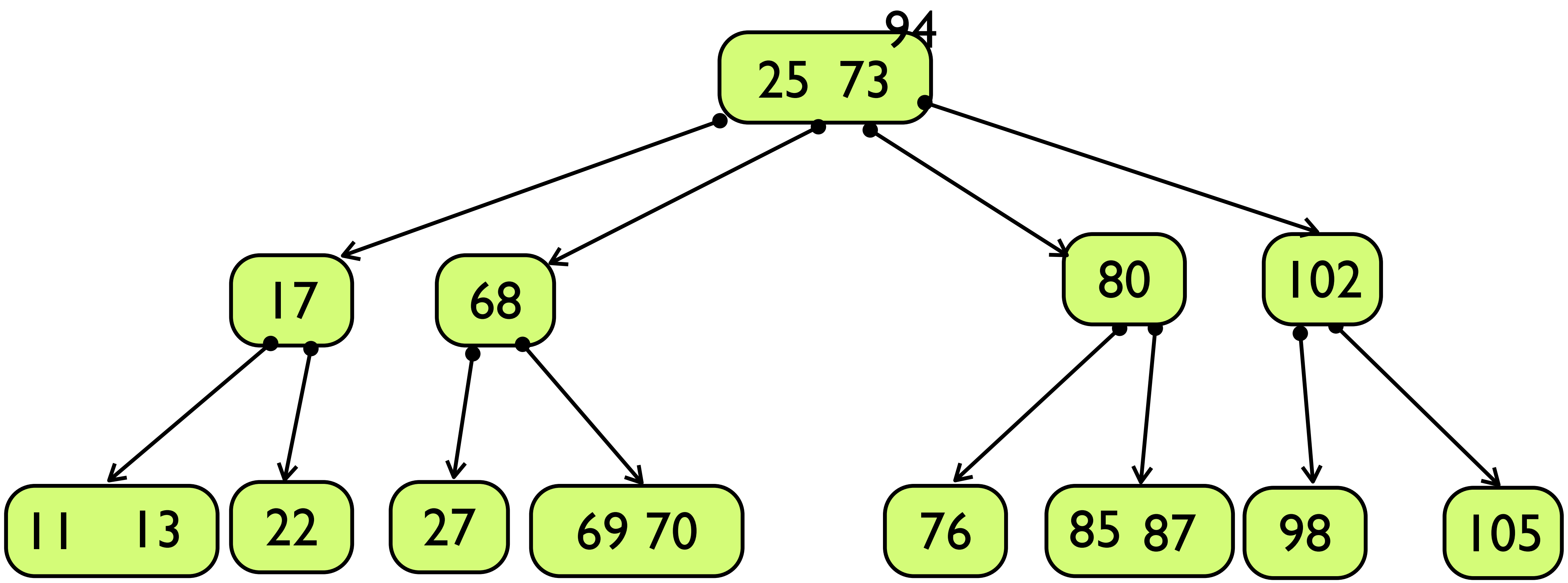
2,3 Tree Insertion – Splitting at the root



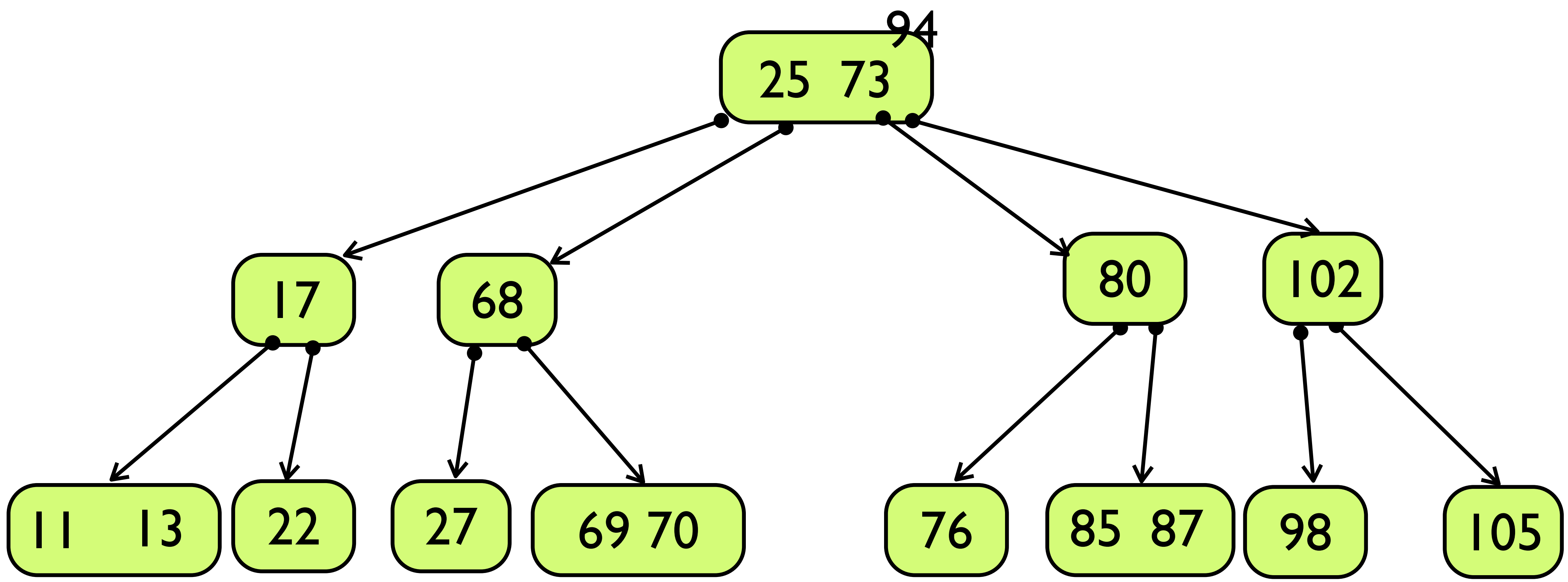
2,3 Tree Insertion – Splitting at the root



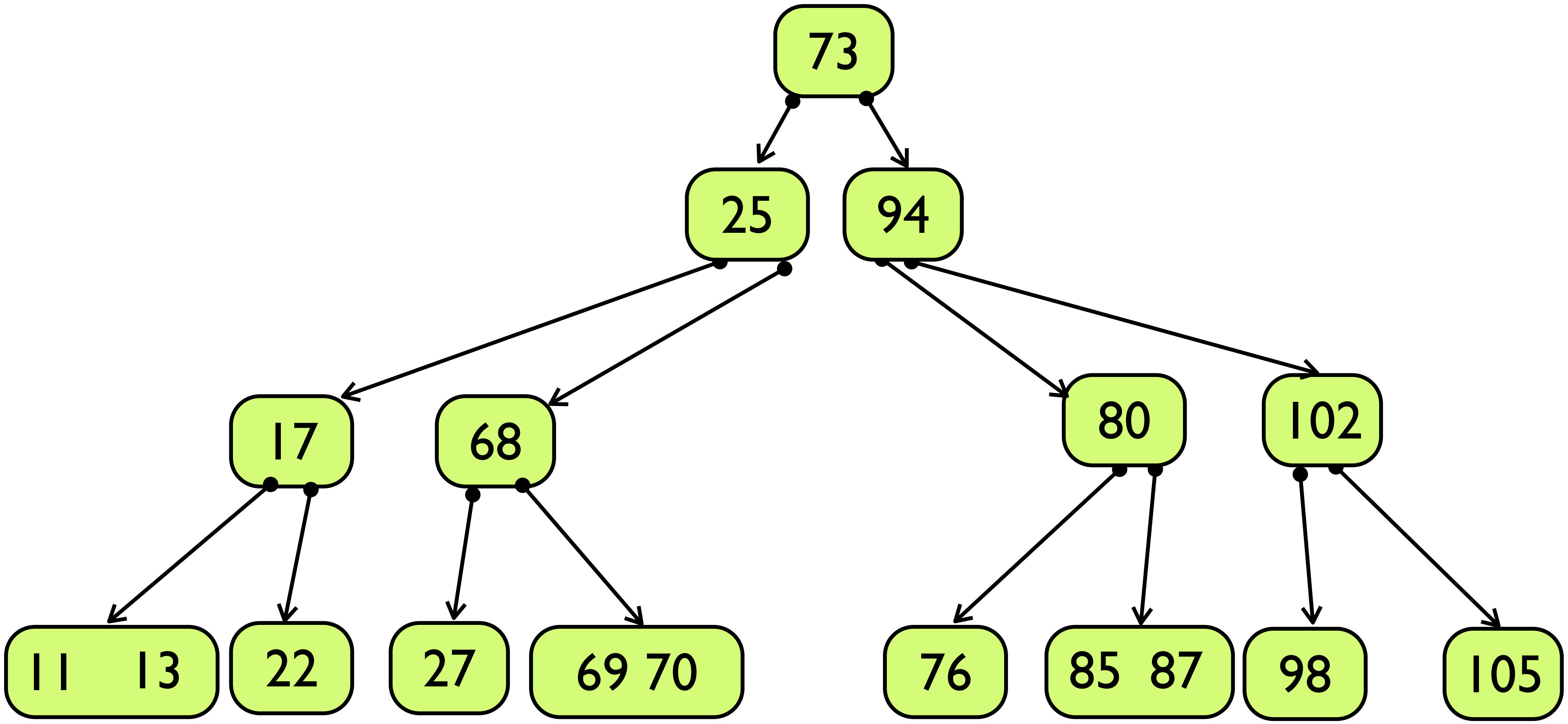
2,3 Tree Insertion – Splitting at the root



2,3 Tree Insertion – Splitting at the root

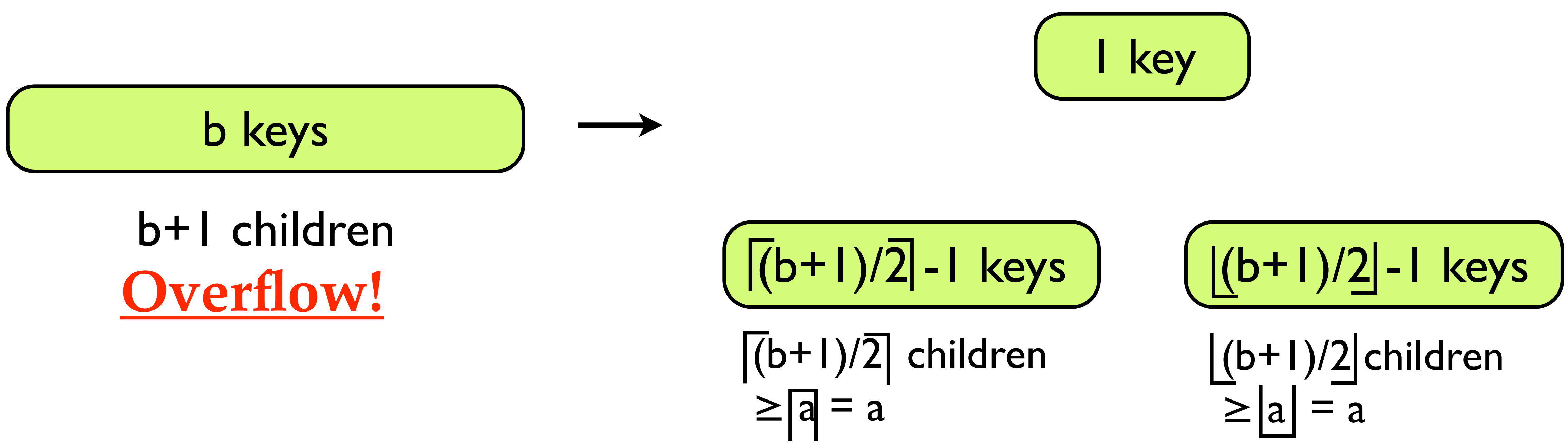


2,3 Tree Insertion – Splitting at the root

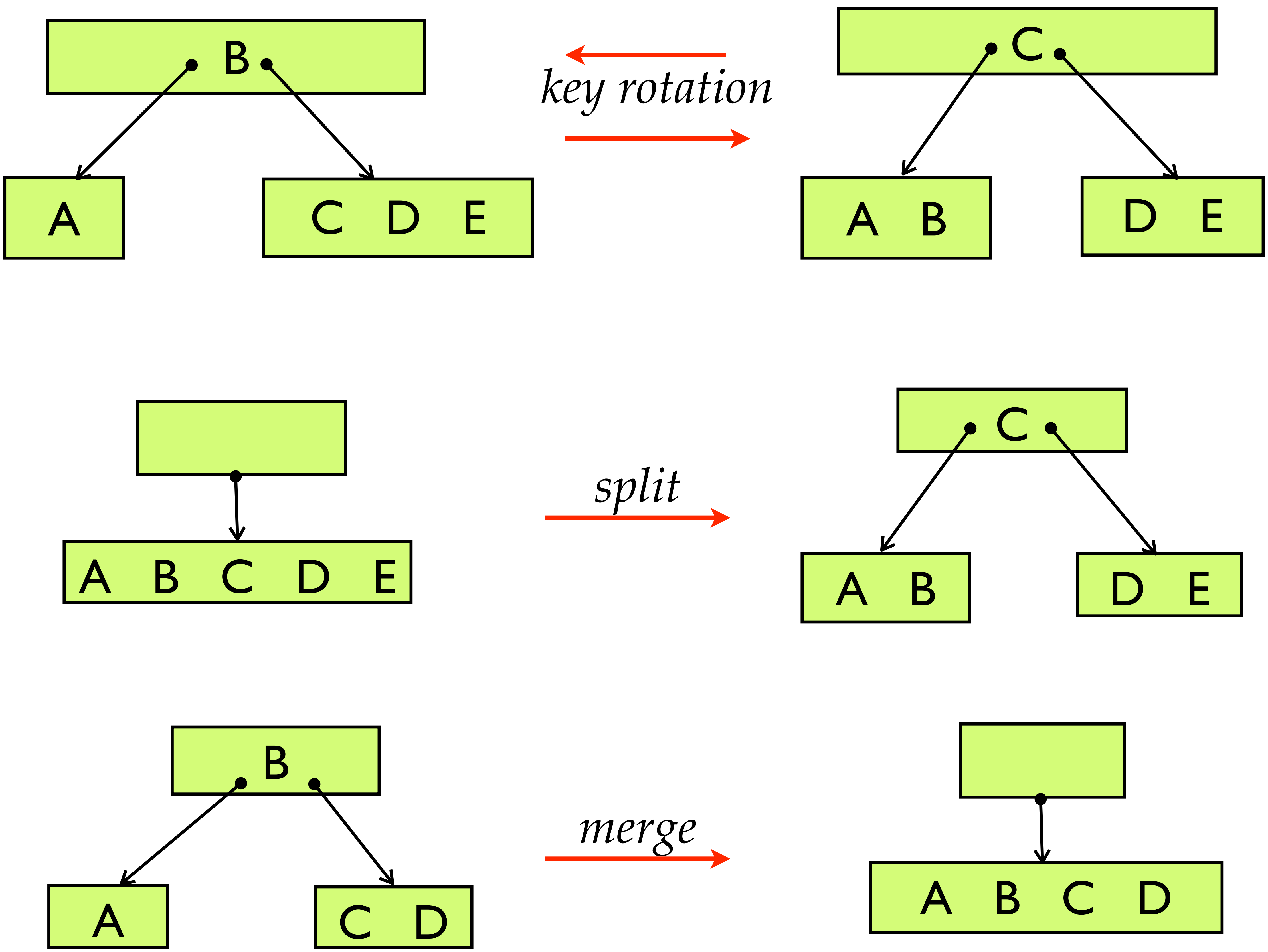


From 2,3-Trees to a,b-trees

- An a,b-tree is a generalization of a 2,3-tree, where each node (except the root) has between a and b children.
- Root can have between 2 and b children.
- We require that:
  - $a \geq 2$ (can't allow internal nodes to have 1 child)
  - $b \geq 2a - 1$ (need enough children to make split work)



a,b Insertions & Deletions



Deletion Details

- Try to borrow a key from a sibling if you have one that has an extra (≥ 1 more than the minimum)
- Otherwise, you have a sibling with exactly the minimum number a-1 of keys.
- Since you are underflowing, you must have one less than the minimum number of keys $= a$-2.
- Therefore, merging with your sibling produces a node with a-1 + a-2 = 2a-3 keys.
- This is one less than the maximum (2a-2 keys), so we have room to bring down the key that split us from our sibling.

B-trees

- A <u>*B-tree of order b*</u> is an a,b-tree with $b = 2a\text{-}1$
  - In other words, we choose the largest allowed a.

Each node (page) is at least 50% full.

- Want to have large b if bringing a node into memory is slow (say reading a disc block), but scanning the node once in memory is fast.

- b is usually chosen to match characteristics of the device.

- Ex. B-tree of order 1023 has $a = 512$.
  - If this B-tree stores n = 10 million records, its height no more than $O(\log_a n) \approx 2.58$. So only around 3 blocks need to be read from disk.

What if b is very large?

- Need to be able to find which subtree to traverse.

- Could linearly search through keys – technically constant time if b is a constant, but may be time consuming.

- Solution: Store a balanced tree (AVL or splay) at each node so that you can search for keys efficiently.