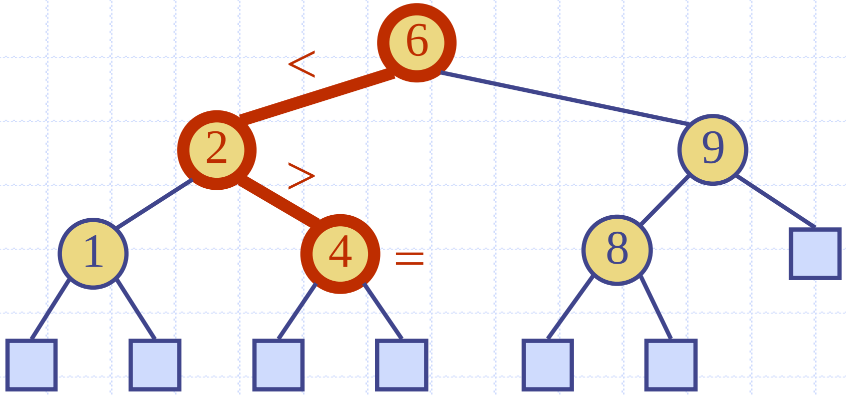
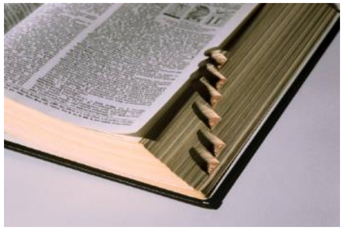
Binary Search Trees



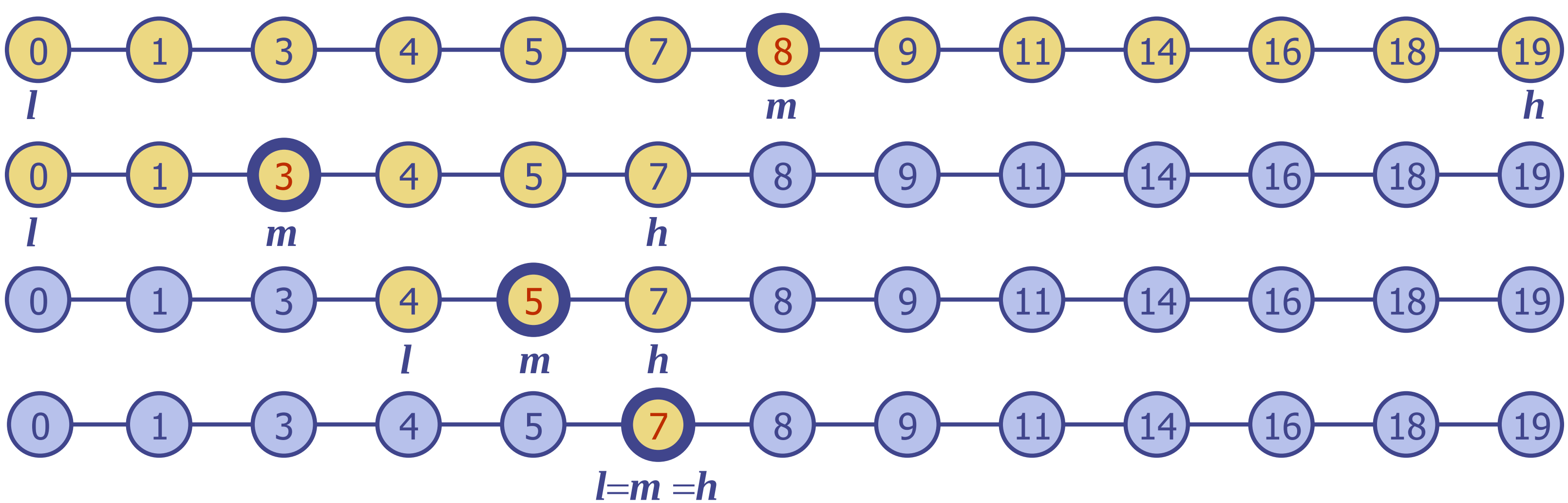
Ordered Maps



- Keys come from a total order
- New operations:
  - Each returns an iterator to an entry:
  - firstEntry(): smallest key in the map
  - lastEntry(): largest key in the map
  - floorEntry(k): largest key $\leq$ k
  - ceilingEntry(k): smallest key $\geq$ k
  - All return end if the map is empty

Binary Search

- Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map implemented via an array, sorted by key
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



Search Tables

- A search table is an ordered map implemented via an array
- Performance:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - put (or insert) takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - erase take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The table is effective only for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., English dictionaries, student records)

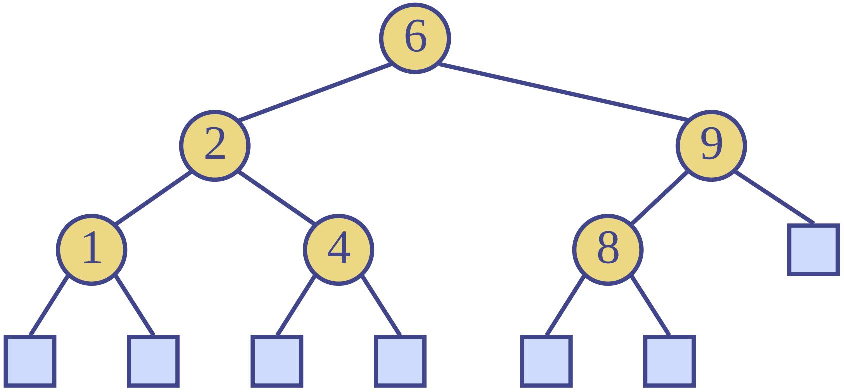
Bad!

Bad!

Binary Search Trees (BST)

- A binary search tree stores keys at internal nodes, satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items (for easy coding only)

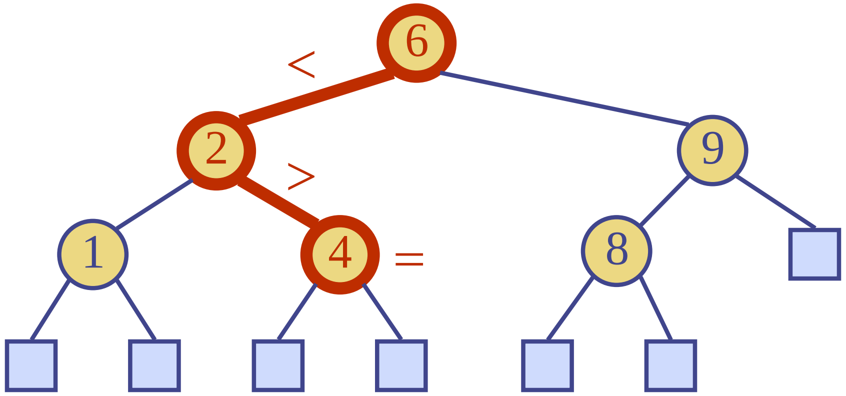
- An inorder traversal of a binary search trees visits the keys in increasing order



Search in BST

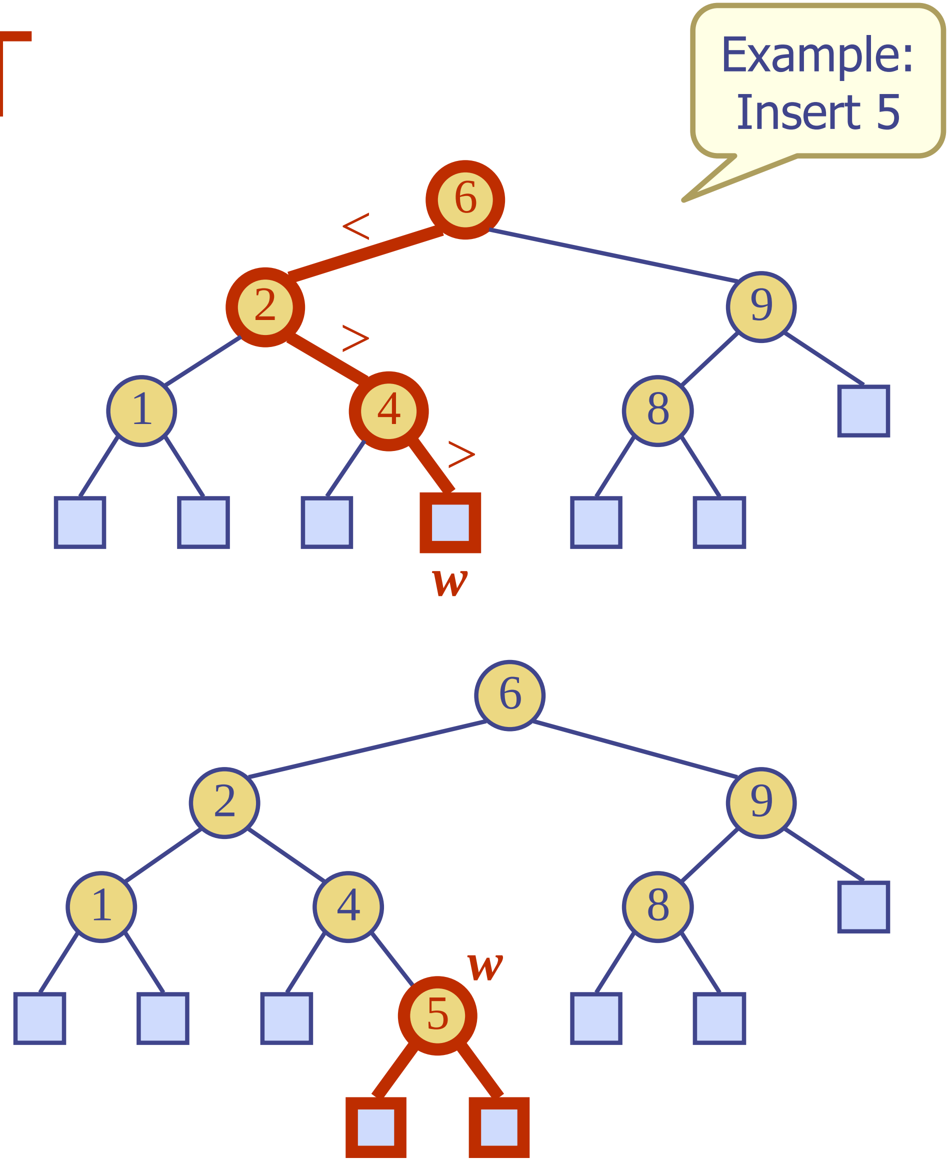
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry are similar

```
Algorithm TreeSearch(k, v)
    if v.isExternal ()
        return v
    if k < v.key()
        return TreeSearch(k, v.left())
    else if k = v.key()
        return v
    else { k > v.key() }
        return TreeSearch(k, v.right())
```



Insertion in BST

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- To keep the height small ➔ AVL trees, splay trees, (2, 4) trees, red-black trees, etc.



Deletion in BST (1/5)

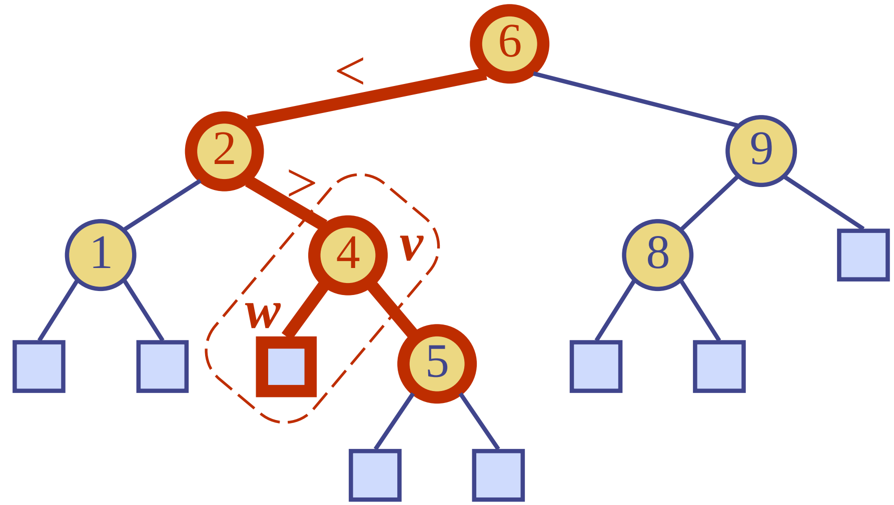
- Rules for Deletion in BST
  - Node with 2 leaves → Delete directly
  - Node with 1 leaf → Replace it with the non-leaf subtree
  - Node with 0 leaf → Replace it with its inorder successor (or predecessor), which is then replaced by its non-leaf subtree

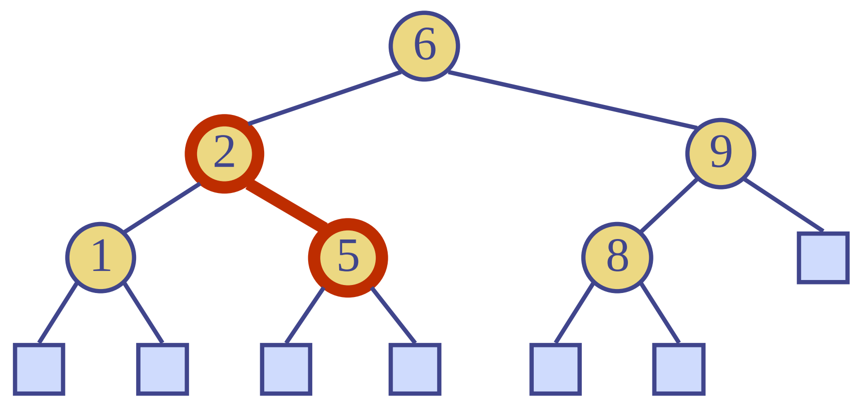
Only one!

- Key point: To keep inorder sequence (with no consideration on tree height)

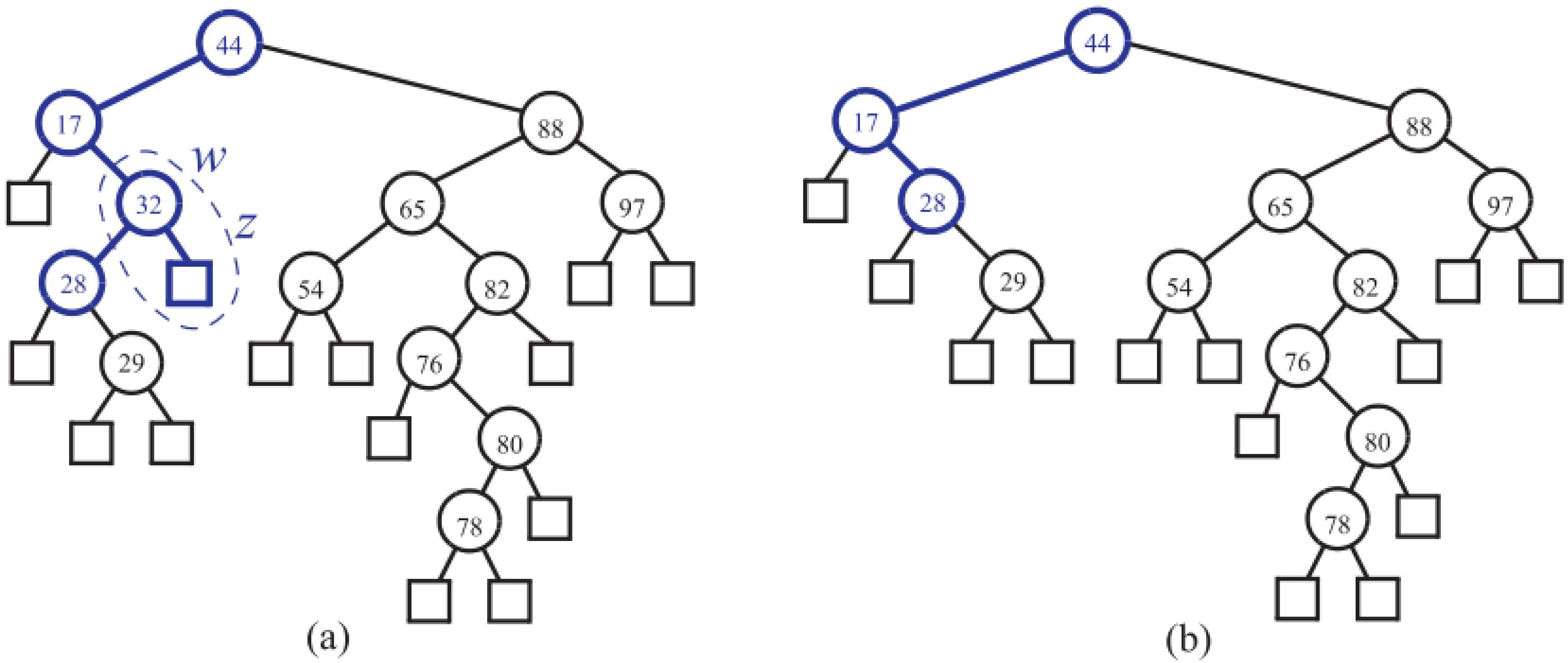
Deletion in BST (2/5)

- To perform erase(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4



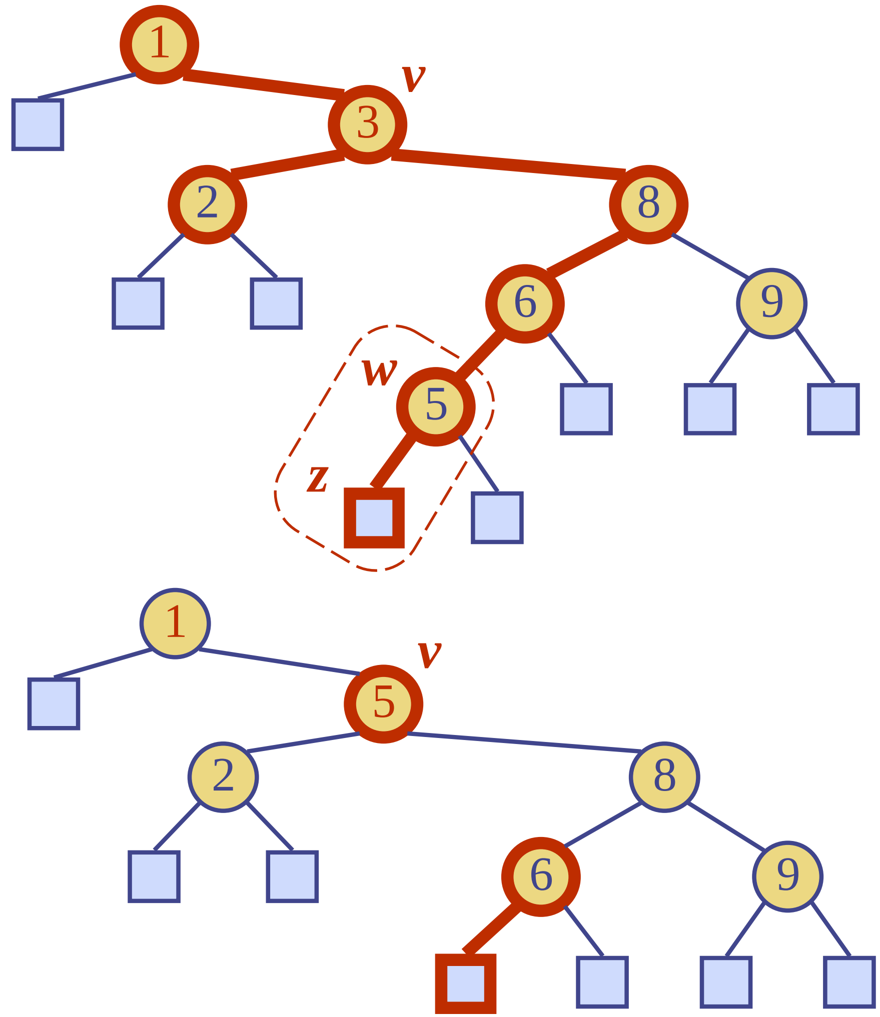


Deletion in BST (3/5)

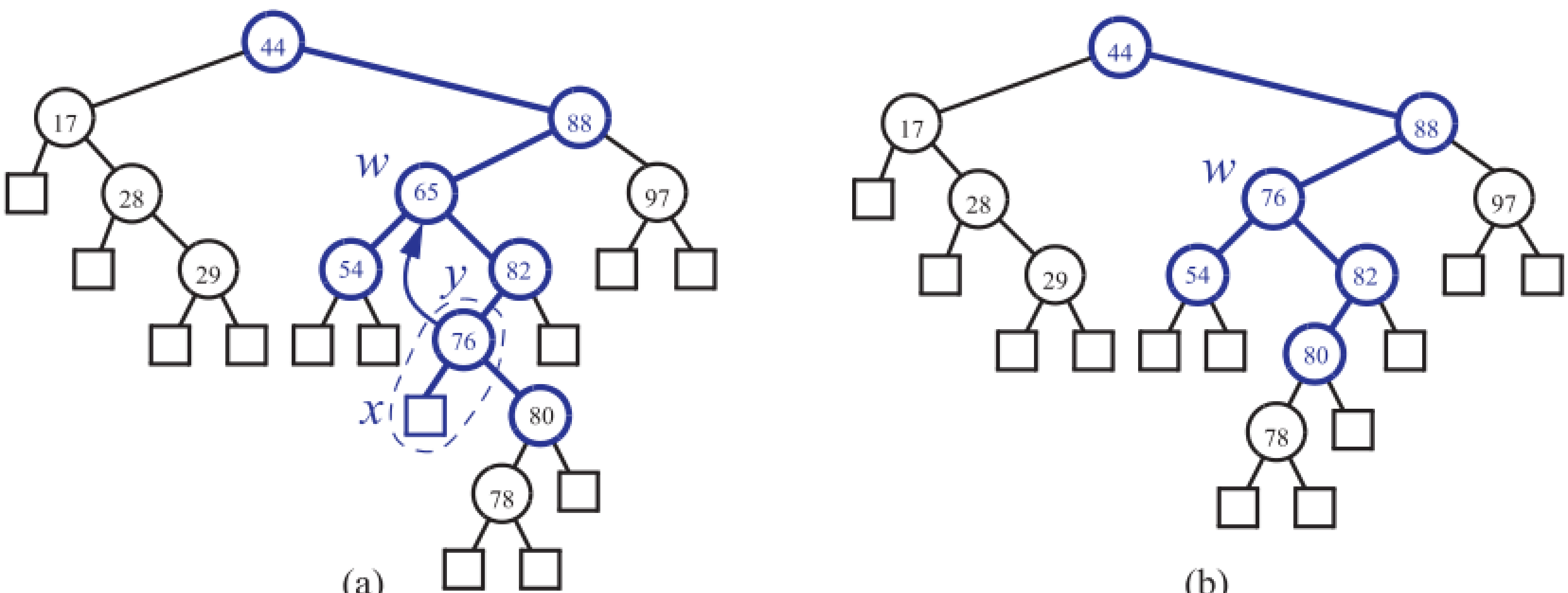


Deletion in BST (4/5)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3



Deletion in BST (5/5)



Hint: Instead, you can choose 54 to replace 65 too.

Performance

- Consider an ordered map with n items implemented by a BST of height h
  - the space used is $O(n)$
  - methods get, floorEntry, ceilingEntry, put and erase take $O(h)$ time
- The height h
  - Worse case $h=O(n)$
  - Best case $h=O(\log n)$
- How to keep a shallow tree is our next topic.