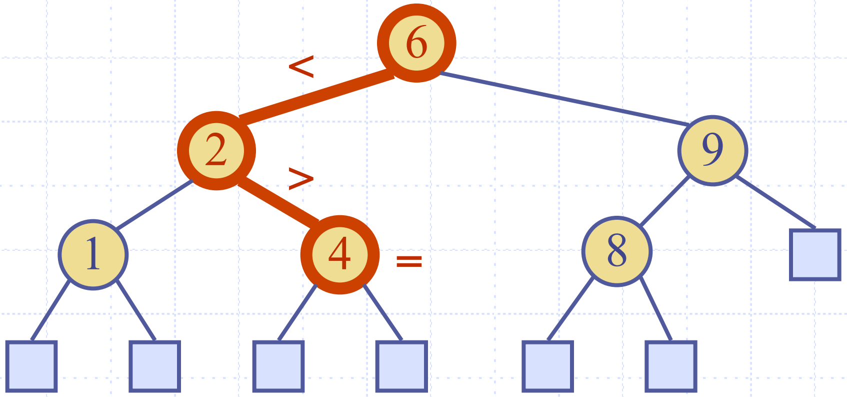
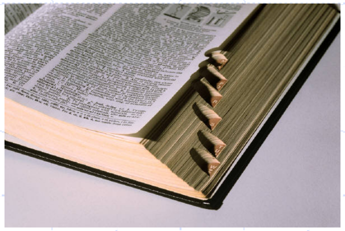
Binary Search Trees

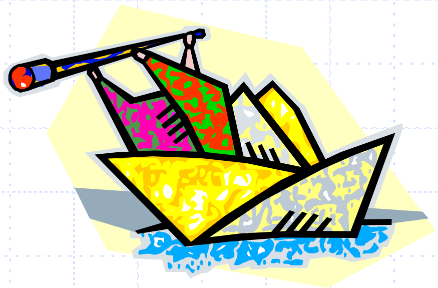


Ordered Maps

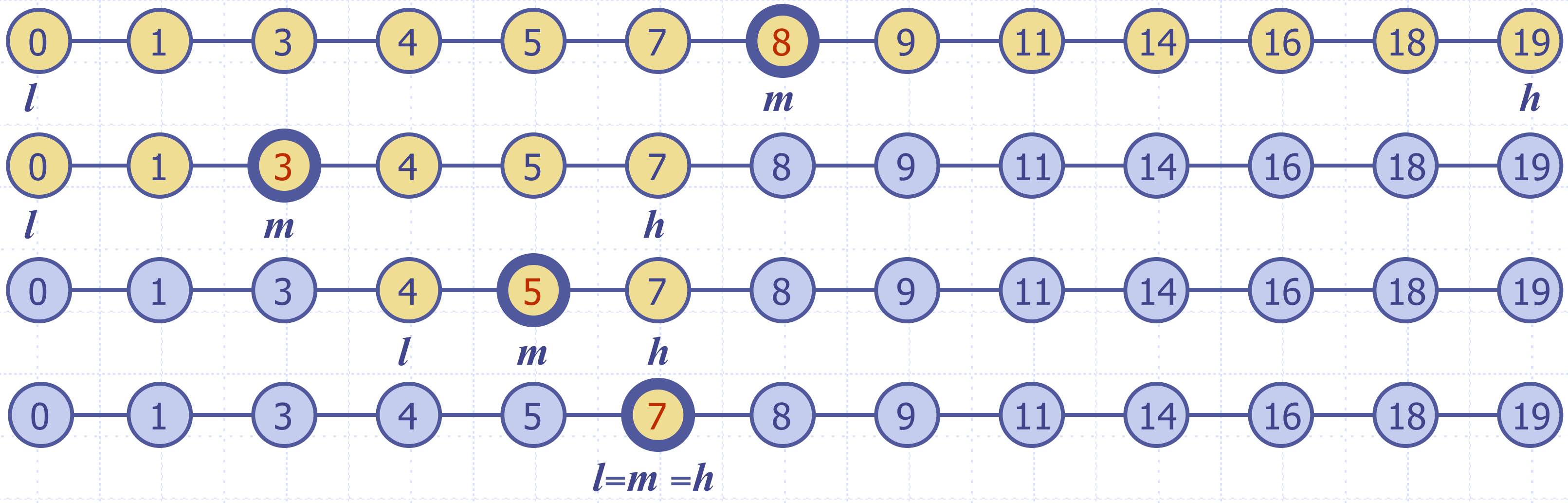


- Keys come from a total order
- New operations:
  - Each returns an iterator to an entry:
  - firstEntry(): smallest key in the map
  - lastEntry(): largest key in the map
  - floorEntry(k): largest key ≤ k
  - ceilingEntry(k): smallest key ≥ k
  - All return end if the map is empty

Binary Search



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

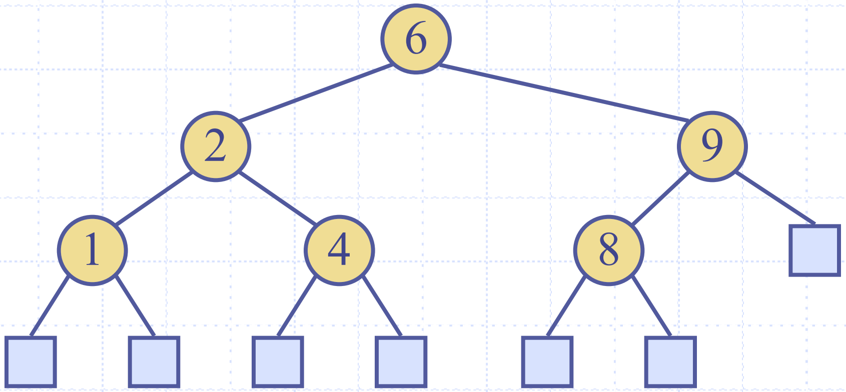


Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - get 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 lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

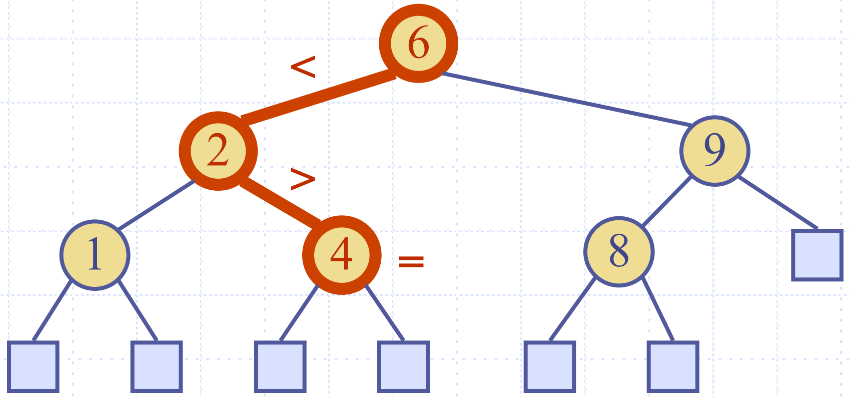
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

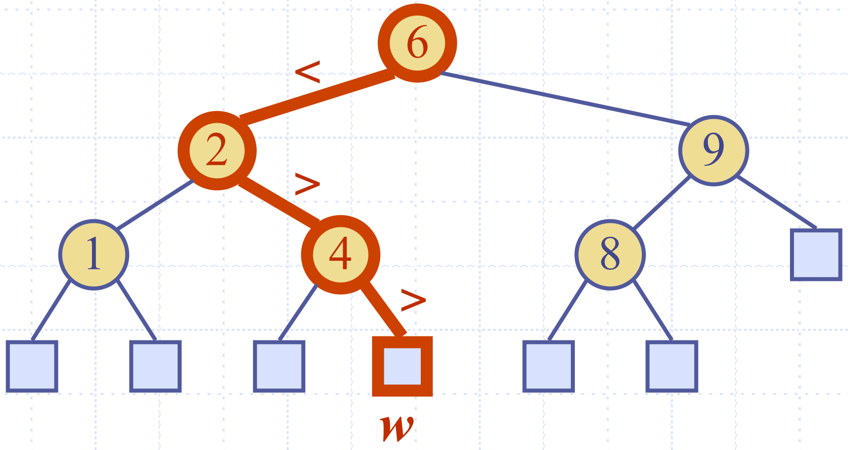
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry are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

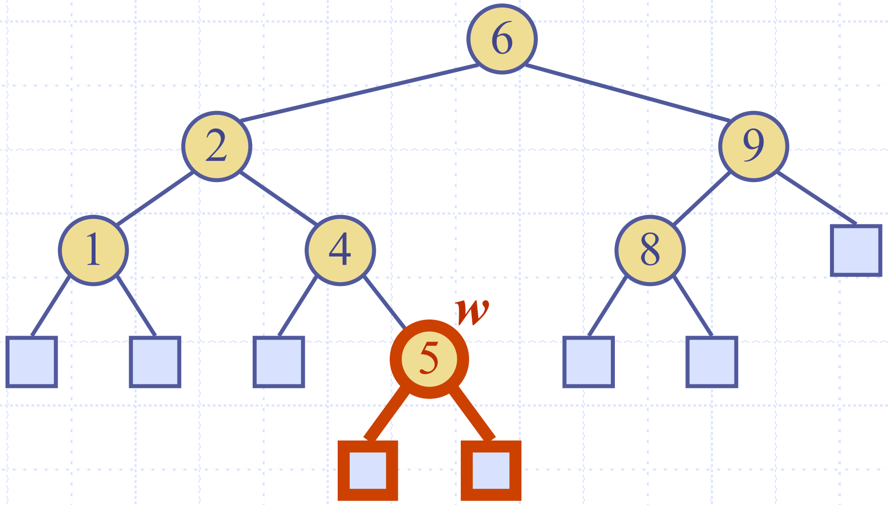
```
Algorithm TreeSearch(k, v)
    if v.isExternal ()
        return v
    if k < v.key()
        return TreeSearch(k, v.left())
    else if k = v.key()
        return v
    else { k > v.key() }
        return TreeSearch(k, v.right())
```



Insertion

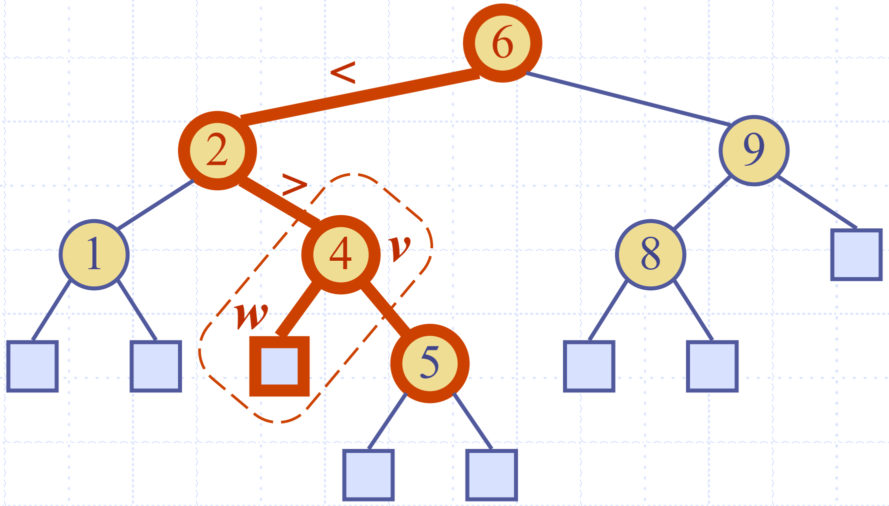
- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

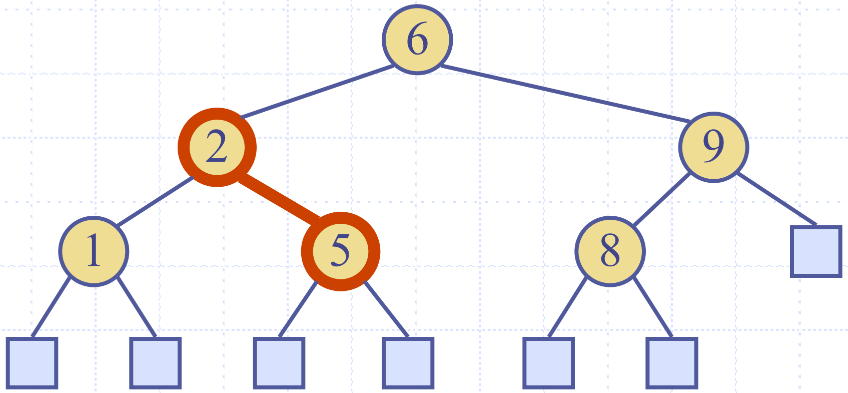




Deletion

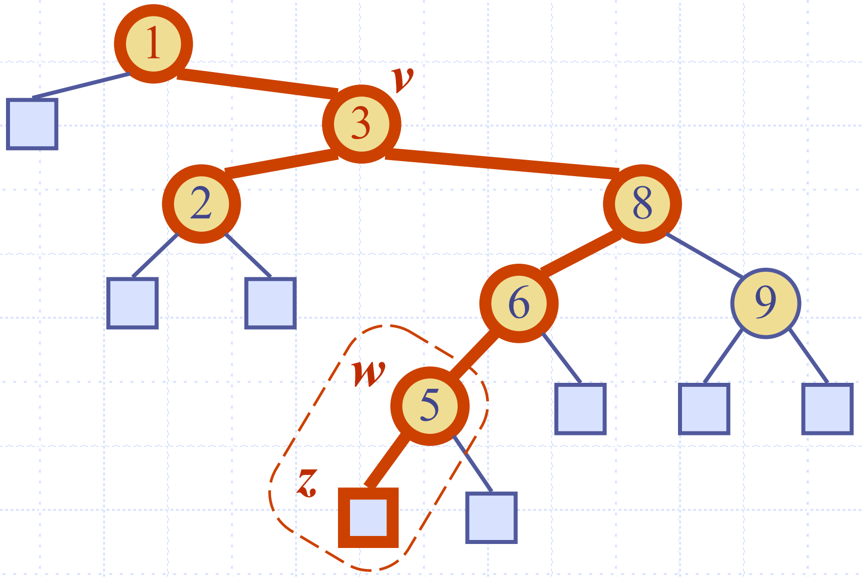
- To perform operation erase(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4

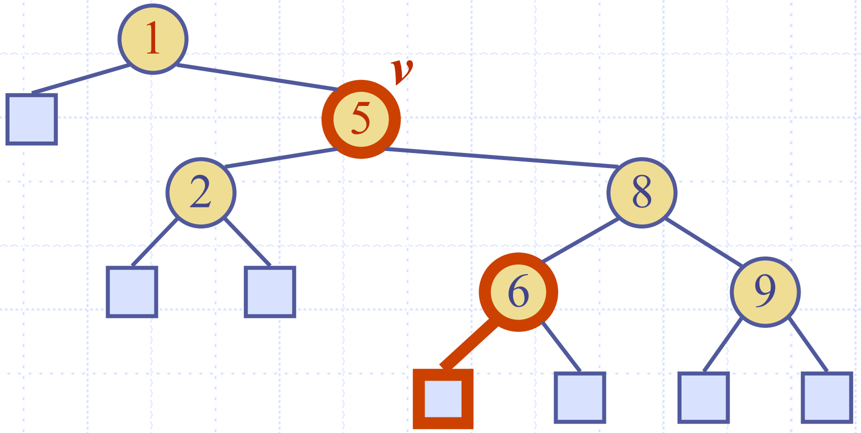




Deletion (cont.)

- 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





Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get, floorEntry, ceilingEntry, put and erase take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case