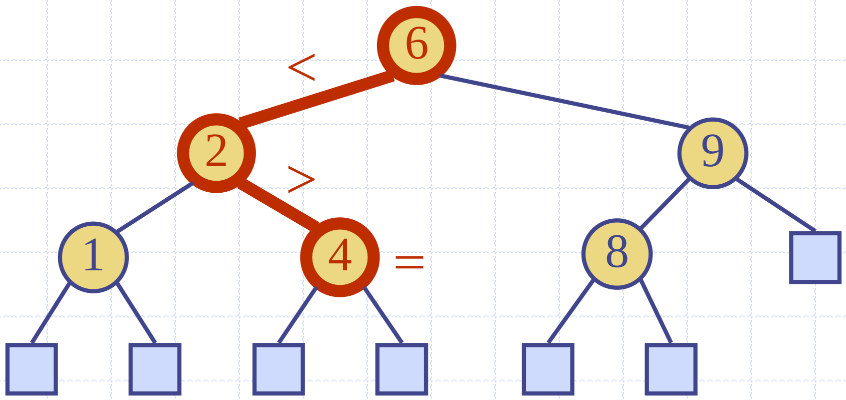
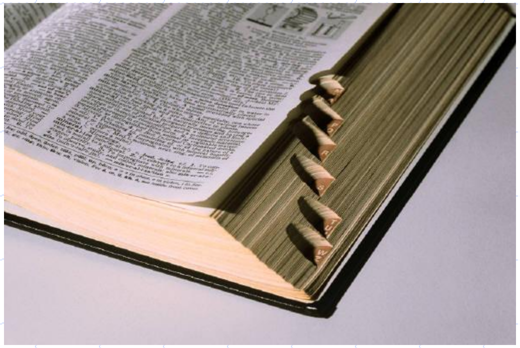
Binary Search Trees

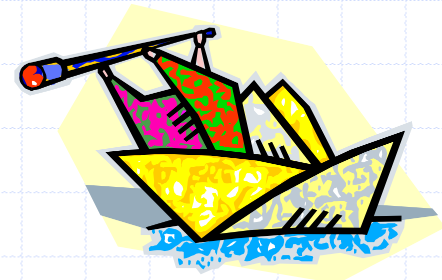


Ordered Dictionaries

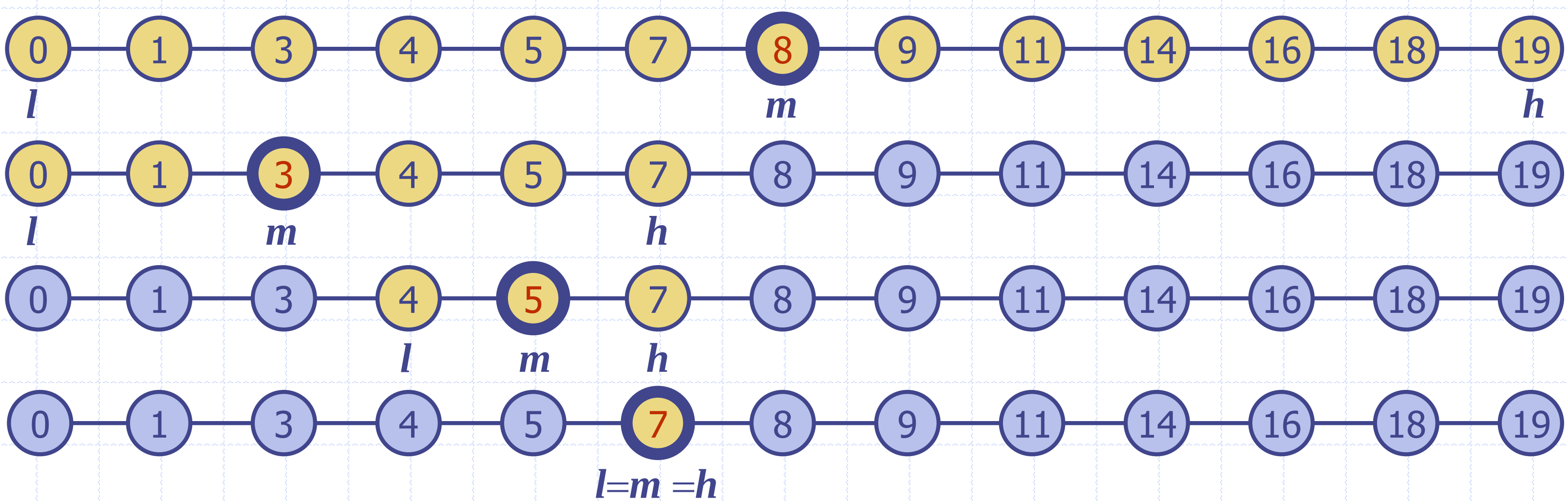


- Keys are assumed to come from a total order.
- New operations:
  - closestKeyBefore(k)
  - closestElemBefore(k)
  - closestKeyAfter(k)
  - closestElemAfter(k)

Binary Search (§3.1.1)



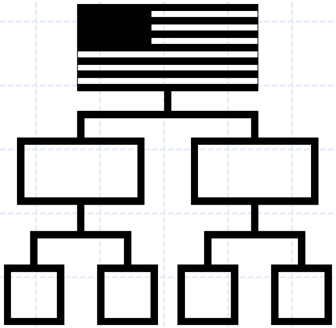
- Binary search performs operation findElement(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: findElement(7)



Lookup Table (§3.1.1)

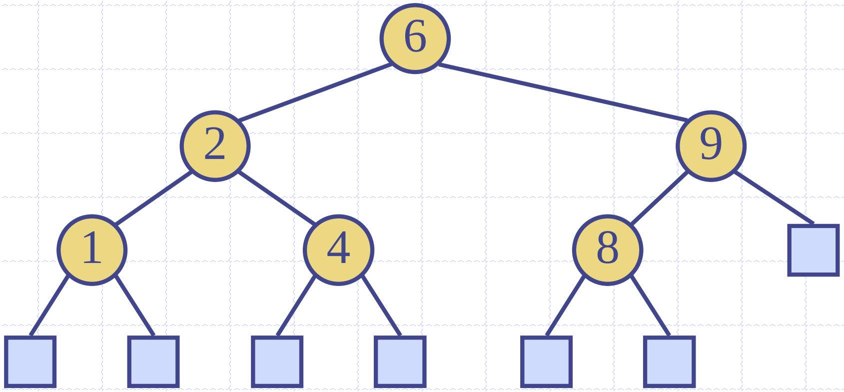
- A lookup table is a dictionary implemented by means of a sorted sequence
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - findElement takes $O(\log n)$ time, using binary search
  - insertItem takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - removeElement 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 Tree (§3.1.2)



- A binary search tree is a binary tree storing keys (or key-element pairs) 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

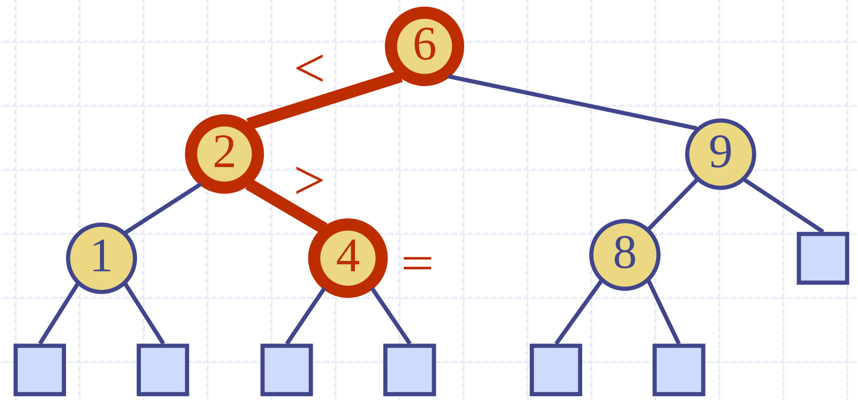
- An inorder traversal of a binary search trees visits the keys in increasing order



Search (§3.1.3)

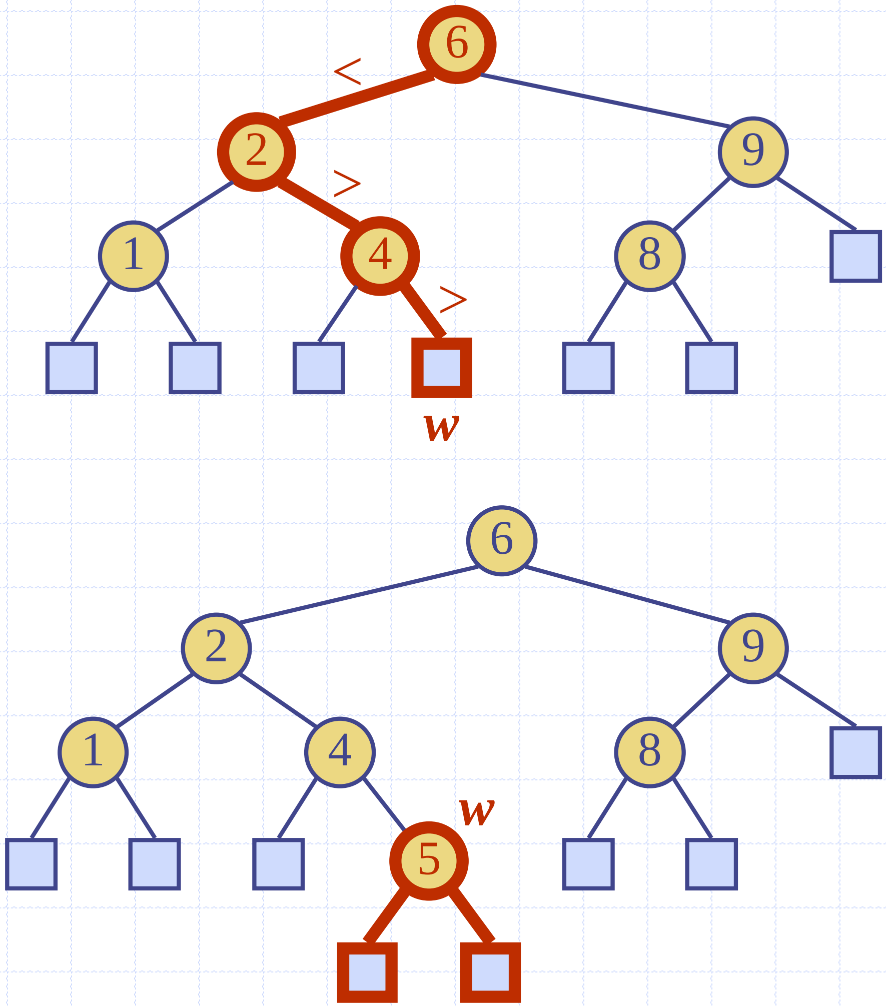
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4)

Algorithm ***findElement(k, v)***
    if *T.isExternal* (*v*)
        return *NO_SUCH_KEY*
    if $k < key(v)$
        return *findElement*(*k*, *T.leftChild*(*v*))
    else if $k = key(v)$
        return *element*(*v*)
    else { $k > key(v)$ }
        return *findElement*(*k*, *T.rightChild*(*v*))



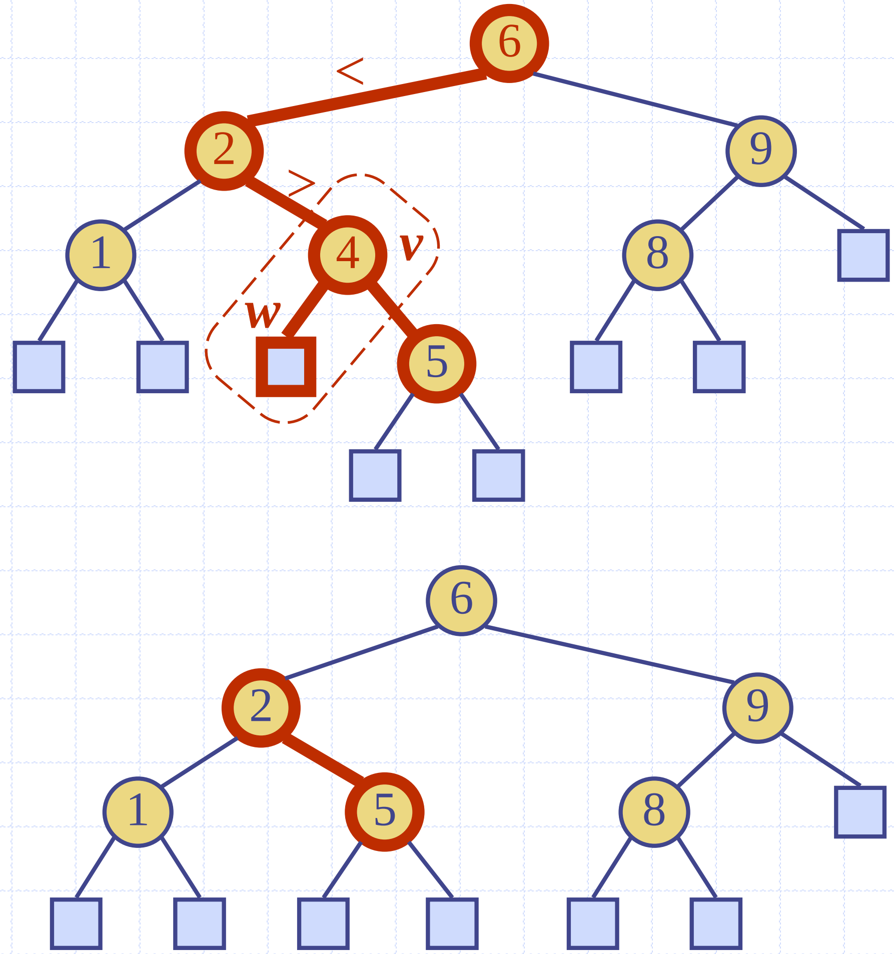
Insertion (§3.1.4)

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let 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



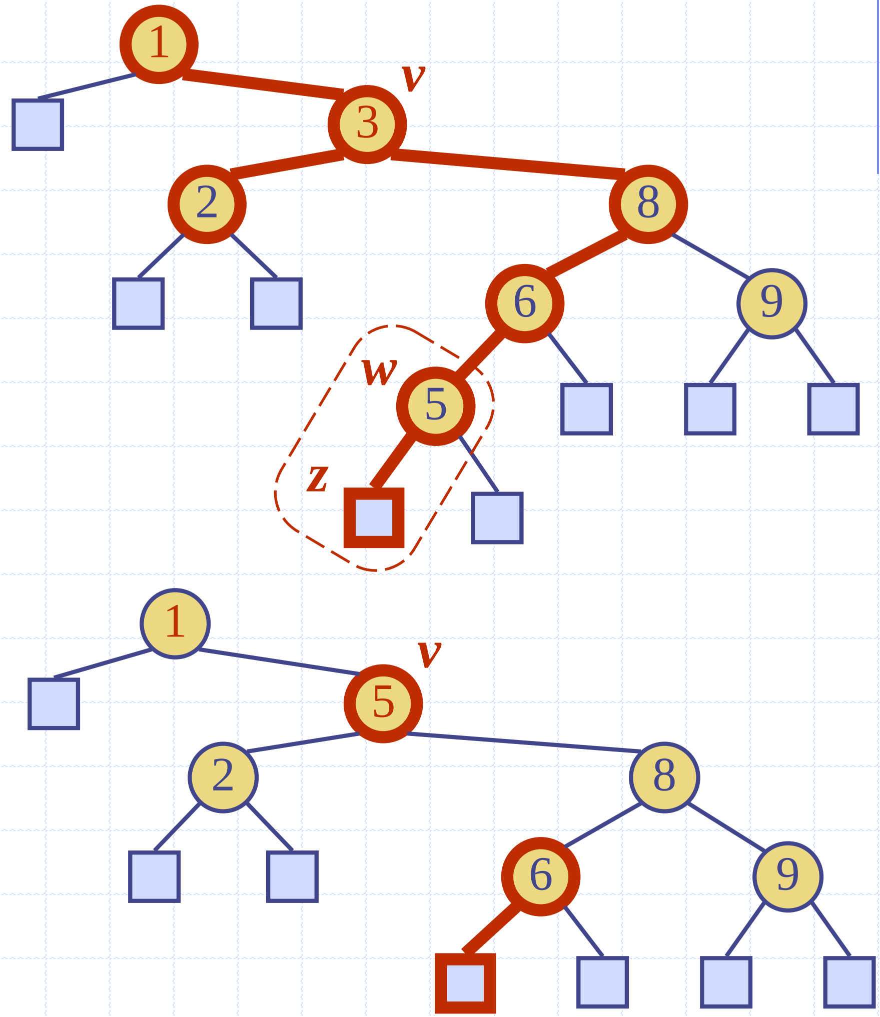
Deletion (§3.1.5)

- To perform operation removeElement(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 removeAboveExternal(w)
- 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 removeAboveExternal(z)
- Example: remove 3



Performance (§3.1.6)

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods findElement , insertItem and removeElement take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case