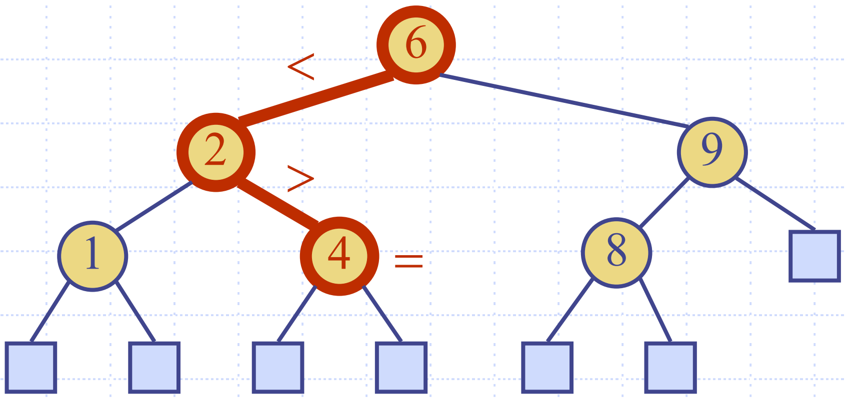
Dictionaries



Outline and Reading

- Dictionary ADT (§8.1.1)
- Log file (§8.1.1)
- Binary search (§8.3.3)
- Lookup table (§8.3.2)
- Binary search tree (§9.1)
  - Search (§9.1.1)
  - Insertion and deletion (§9.1.2)
  - Performance (§9.1.4)

Dictionary ADT

- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

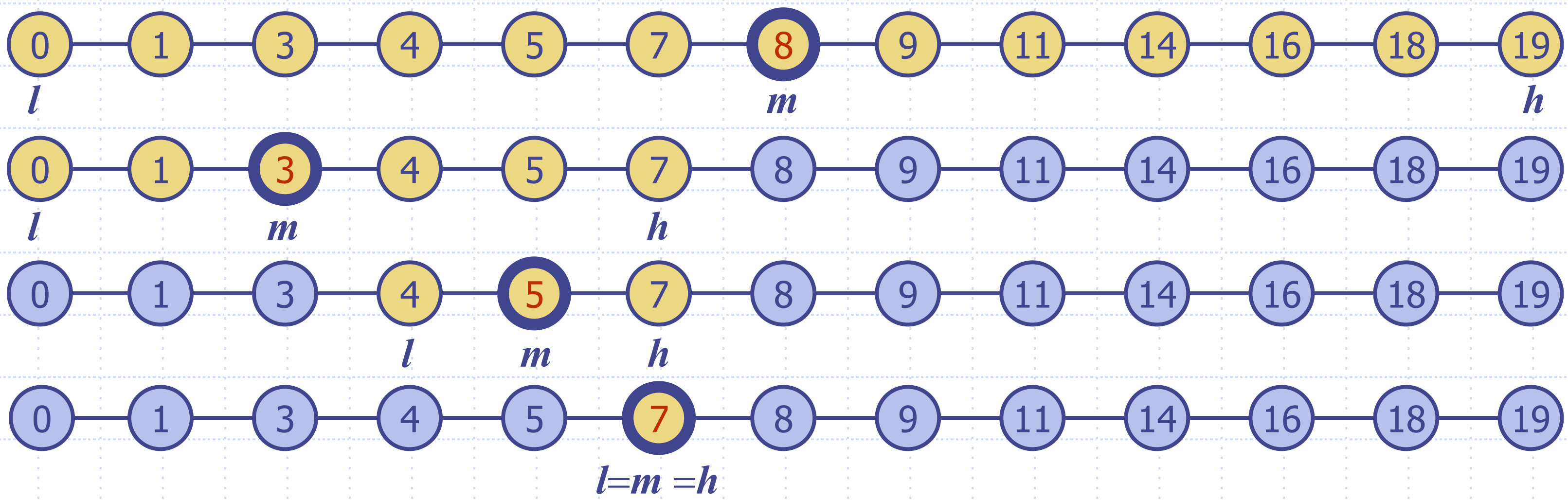
- Dictionary ADT methods:
  - find(k): if the dictionary has an item with key k, returns the position of this item, else, returns a null position.
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): removes the item with key k from the dictionary. If no such element exists, an error occurs.
  - size(), isEmpty()
  - keys(), Elements()

Log File

- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Binary Search

- Binary search performs operation find(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 a logarithmic number of steps
- Example: find(7)

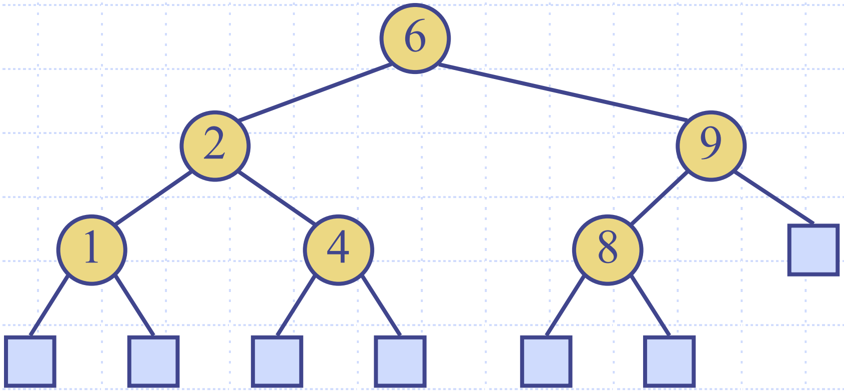


Lookup Table

- 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:
  - find 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

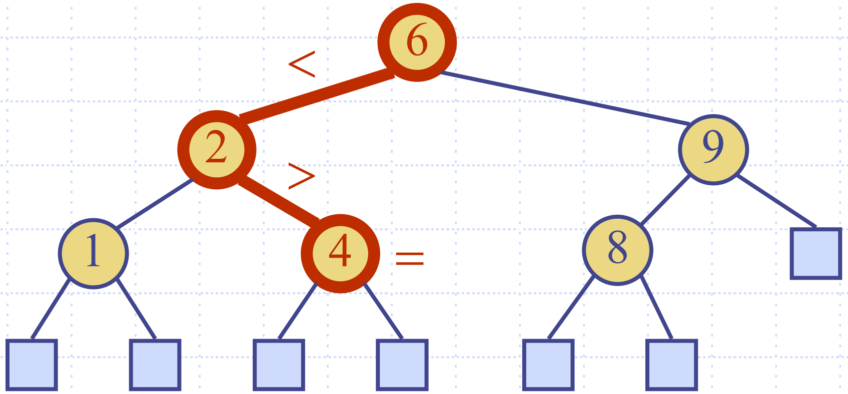
- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes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



Search

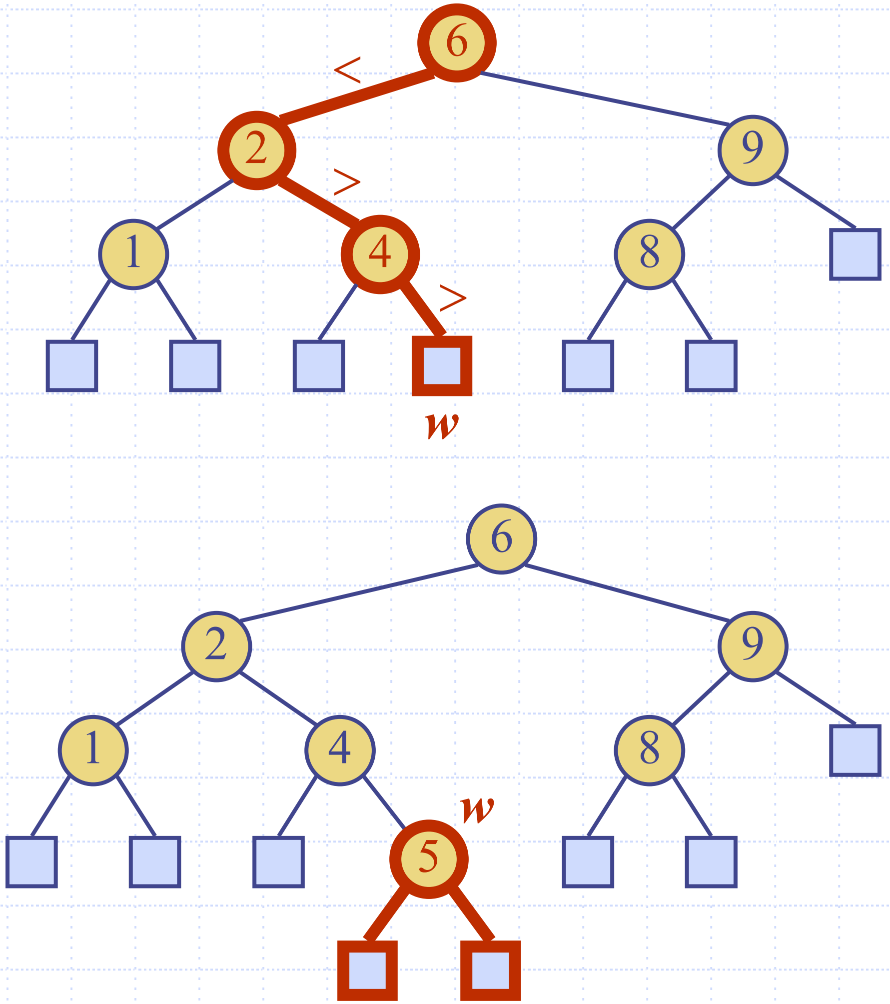
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return a null position
- Example: find(4)

```
Algorithm find (k, v)
    if T.isExternal (v)
        return Position(null)
    if k < key(v)
        return find (k, T.leftChild(v))
    else if k = key(v)
        return Position(v)
    else { k > key(v) }
        return find (k, T.rightChild(v))
```



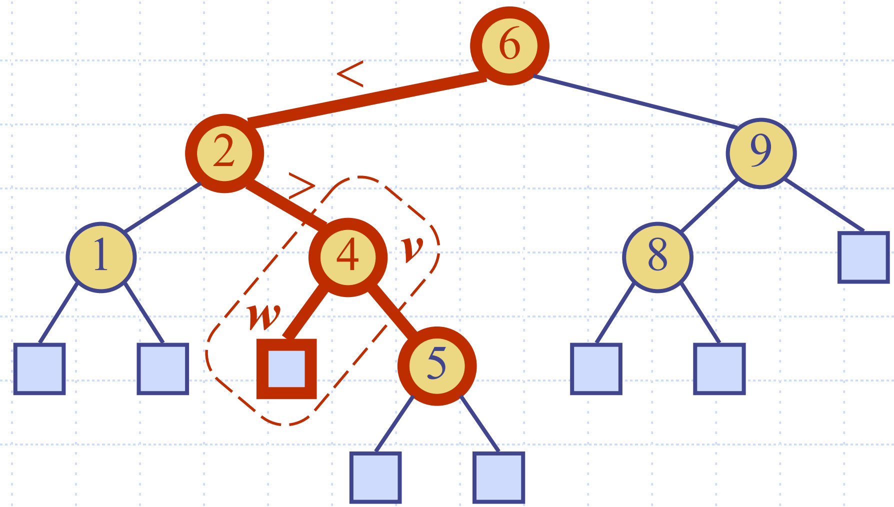
Insertion

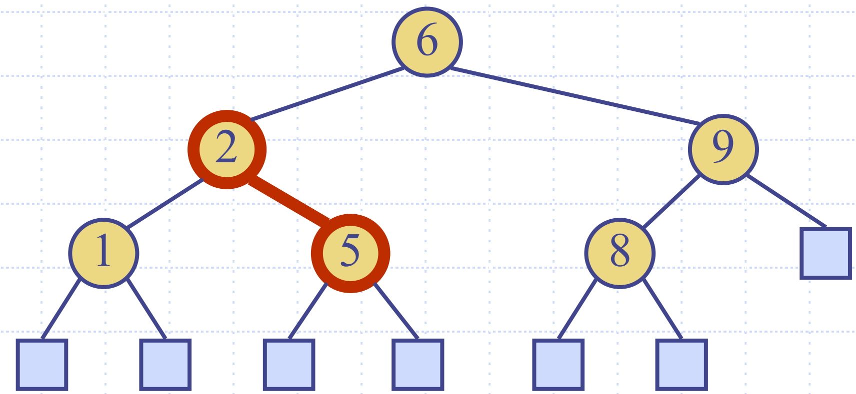
- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



Deletion

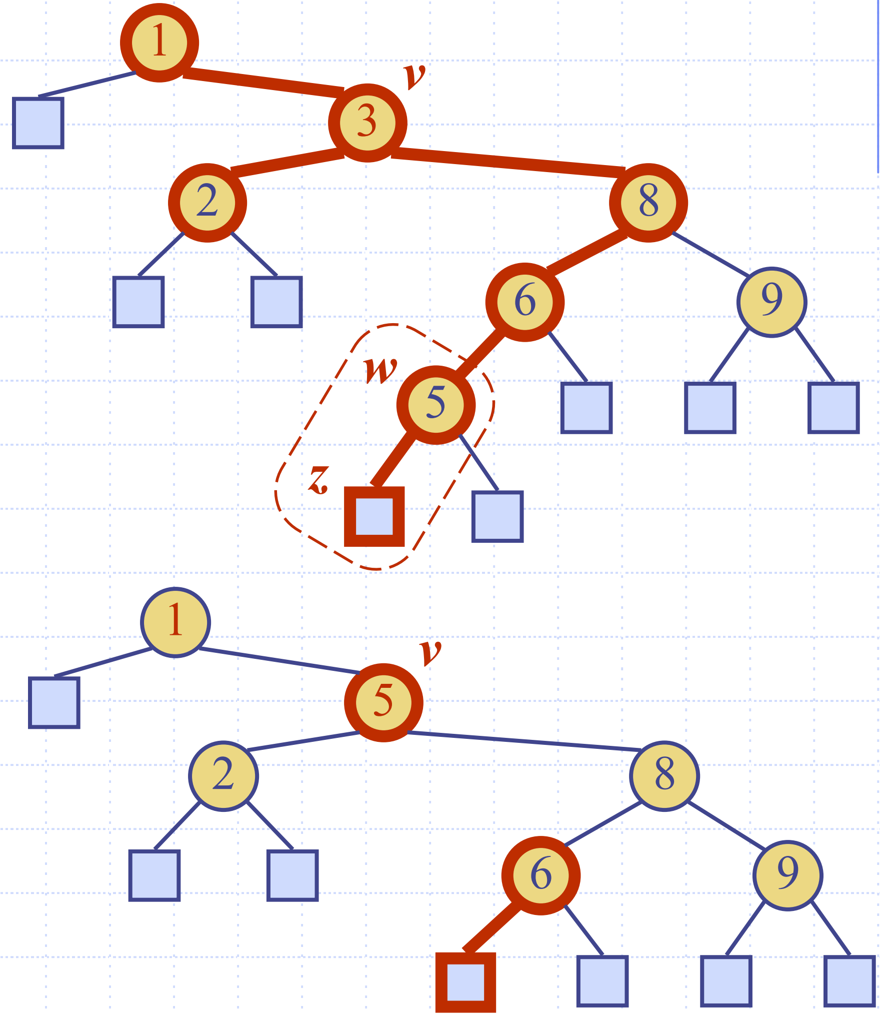
- To perform operation removeElement(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeAboveExternal(w)
- Example: remove 4





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

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods find(), 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