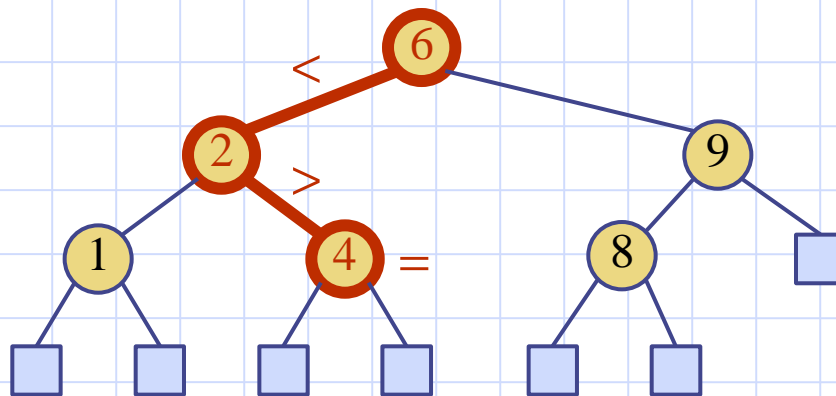


Dictionaries



Outline and Reading

- ◆ Dictionary ADT (§9.3)
 - ◆ Log file (§9.3.1)
 - ◆ Binary search (§9.3.3)
 - ◆ Lookup table (§9.3.3)
 - ◆ Binary search tree (§10.1)
 - Search (§10.1.1)
 - Insertion (§10.1.2)
 - Deletion (§10.1.2)
 - Performance (§10.1.2)

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), findAll(K)**: if the dictionary has an item with key K, returns its element, else, returns null
 - **insert(K, V)**: inserts item (K, V) into the dictionary
 - **remove(Entry<K,V>)**: if the dictionary has an item with Entry<K,V>, removes it from the dictionary and returns its element, else returns null
 - **size(), isEmpty()**
 - **entries()** returns an iterator containing all the entries in the dictionary

Map and Dictionary ADTs

Map ADT methods:

- **get(K)**: if the map has an entry with key K, returns its value, else, returns null
- **put(K, V)**: if map does not have an entry with key equal to K, then add entry to map and return null; else, replace with v the existing value of the entry with key equal to K and return the old value
- **remove(K)**: if the map has an entry with key K, removes it from the map and returns its value, else returns null
- **size(), isEmpty()**
- **entries()**
- **values()**: return an iterable collection containing all the values associated with keys stored in map

Dictionary ADT methods:

- **find(K), findAll(K)**: if the dictionary has an item with key K, returns its element, else, returns null
- **insert(K, V)**: inserts item (K, V) into the dictionary
- **remove(Entry<K,V>)**: if the dictionary has an item with entry `Entry<K,V>`, removes it from the dictionary and returns its element, else returns null
- **size(), isEmpty()**
- **entries()** returns an iterator containing all the entries in the dictionary

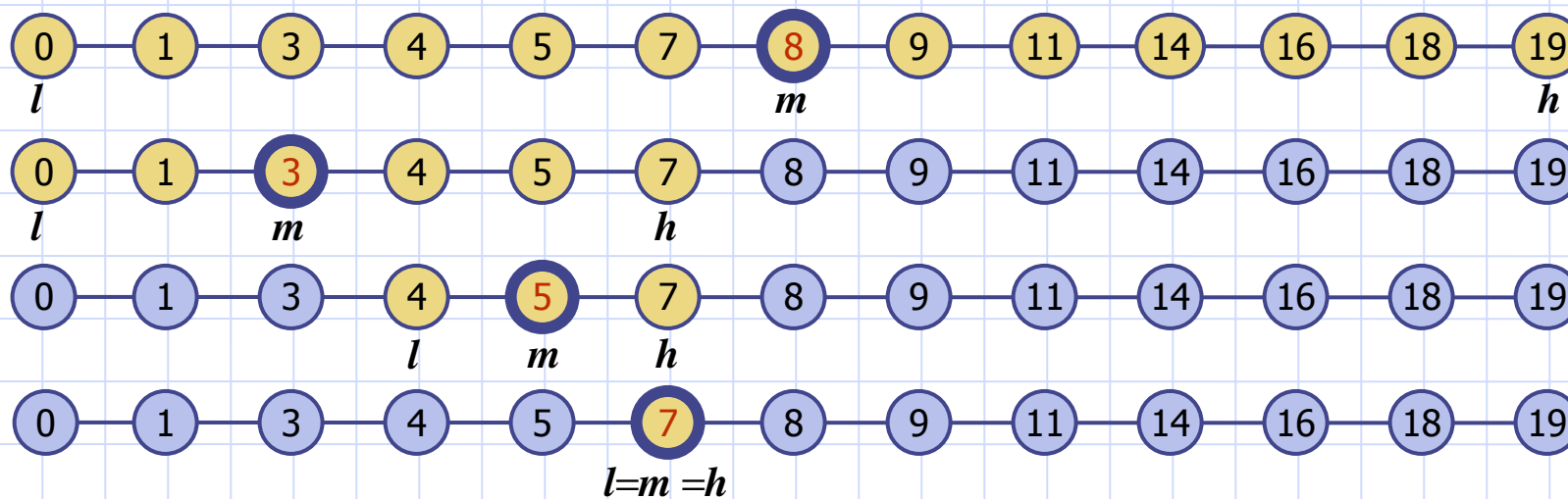
Log File

- ◆ 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), arranged in arbitrary order
- ◆ Performance:
 - **insert** takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
 - **find** and **remove** 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
 - **insert** takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
 - **remove** takes $O(n)$ 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)

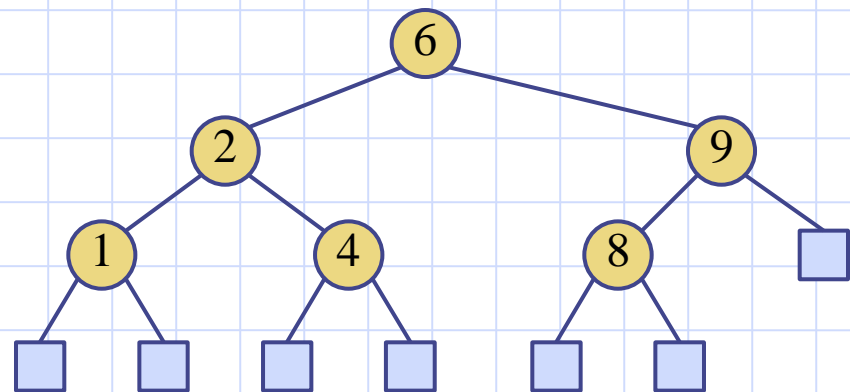
Binary Search Tree

◆ A binary search tree is a binary tree storing key-element pairs at its internal nodes and satisfying the following order 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

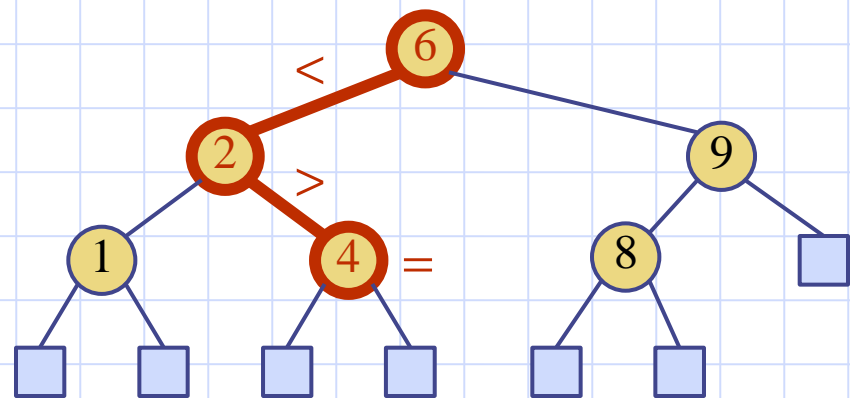
◆ An inorder traversal of a binary search tree visits the keys in non-decreasing order



Search

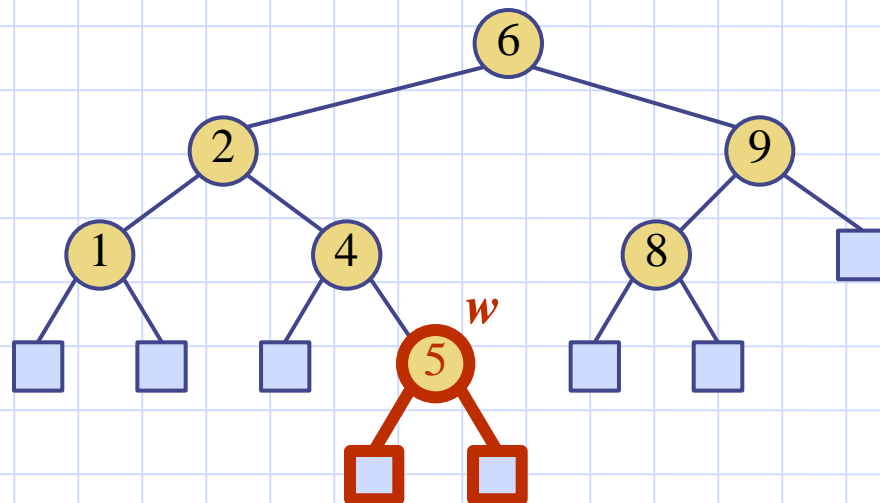
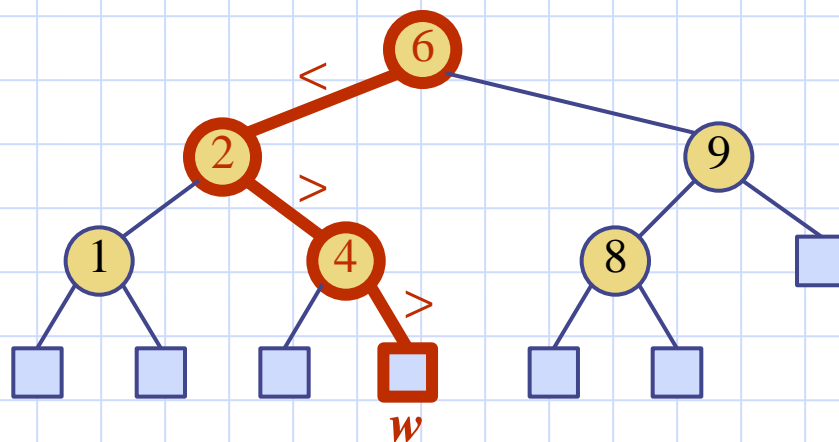
- ◆ To search for a key K , we trace a downward path starting at the root
- ◆ The next node visited depends on the outcome of the comparison of K with the key of the current node
- ◆ If we reach a leaf, the key is not found and we return null
- ◆ Example: $\text{find}(4)$

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



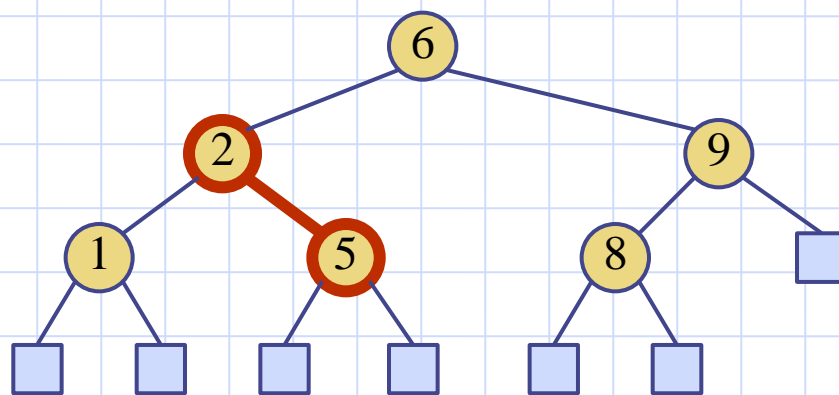
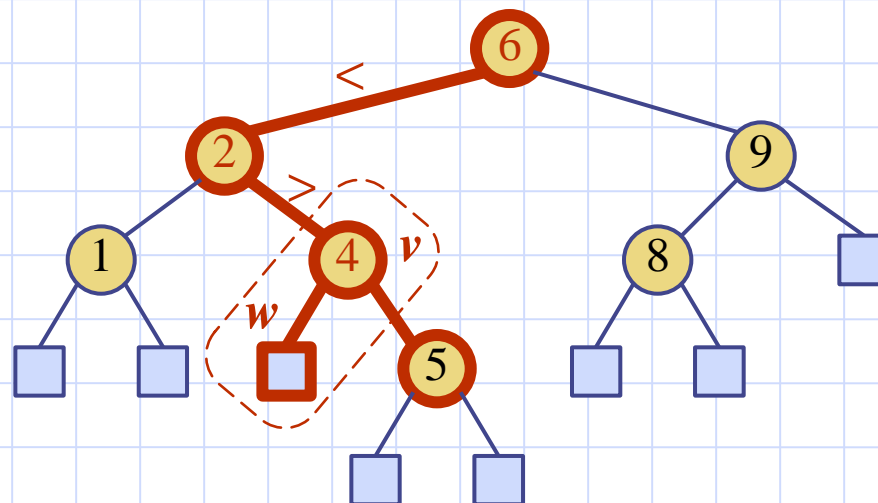
Insertion

- ◆ To perform operation **insert**(K, V), we search for key K
- ◆ 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, V)



Deletion

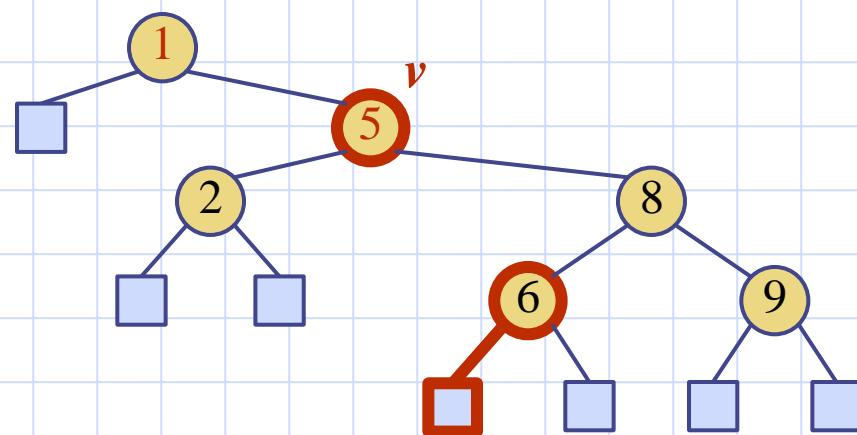
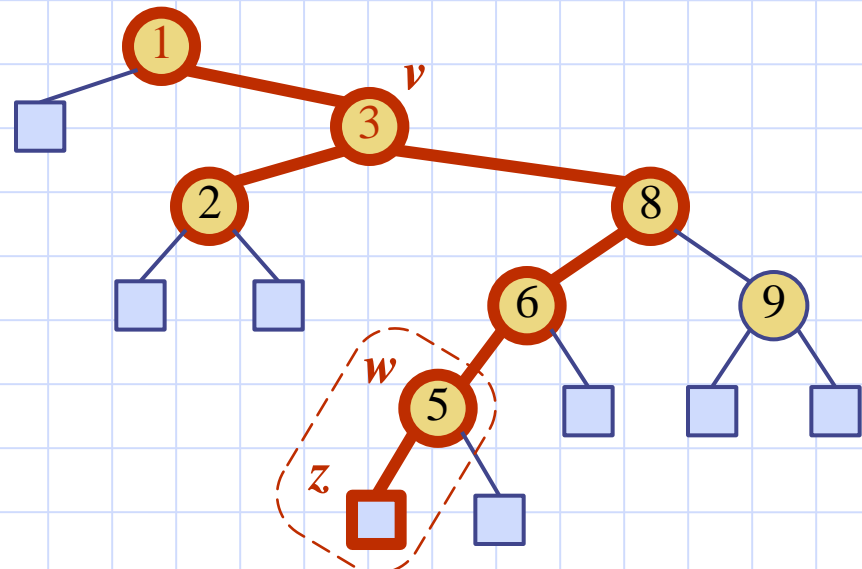
- ◆ To perform operation **remove**(K), we search for key K
- ◆ Assume key K is in the tree, and let v be the node storing K
- ◆ Case 1: node v has a leaf child w
 - We remove v and w from the tree with operation **removeAbove**(w)
 - Example: **remove**(4)



Deletion (cont.)

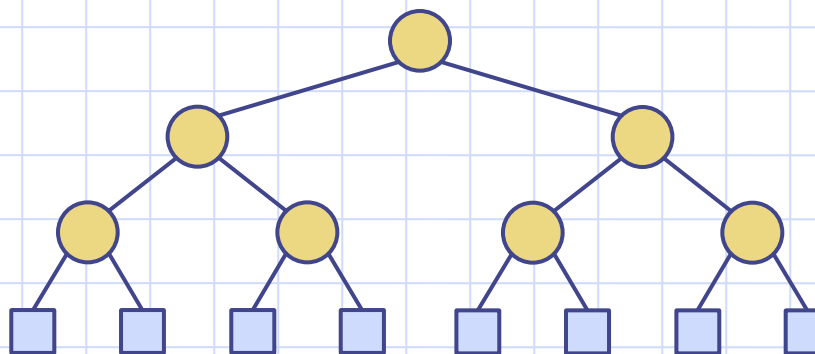
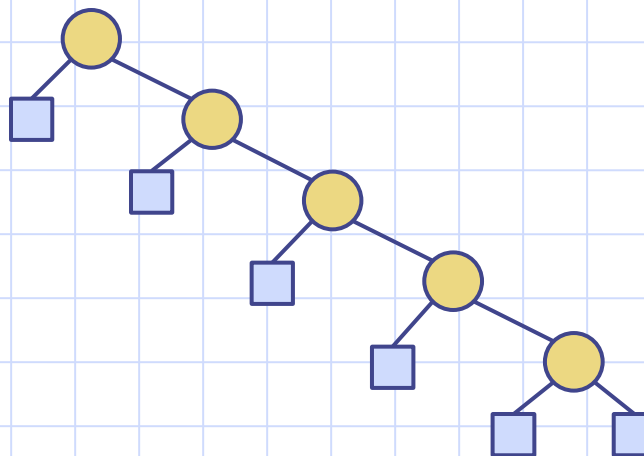
◆ Case 2: both children of node v are internal

- We find the external node z that follows v in an inorder traversal and let w be the parent of z
- We copy the item stored at w into v
- We remove nodes w and z by means of operation `removeAboveExternal(z)`
- Example:
`removeElement(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**, **insert** and **remove** take $O(h)$ time
- ◆ The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



Everybody loves a demo

- ◆ Check [this](#) out! (Animation of binary search, insert, and delete)
- ◆ A good applet to play with: [here](#)