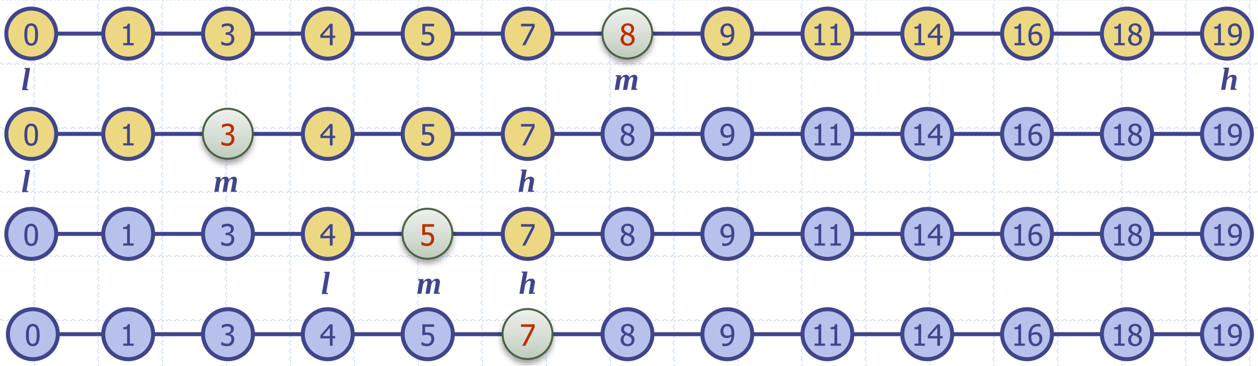
Dictionaries and Binary Search Trees

Sections 9.5 – 10.1



Dictionary

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

Entry ADT

- An entry stores a key-value pair (k,v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v

Dictionary ADT

- Dictionary ADT methods:
  - find(k): if there is an entry with key k, returns an iterator to it, else returns the special iterator end
  - findAll(k): returns iterators b and e such that all entries with key k are in the iterator range [b, e) starting at b and ending just prior to e
  - put(k, o): inserts and returns an iterator to it
  - erase(k): remove an entry with key k
  - begin(), end(): return iterators to the beginning and end of the dictionary
  - size(), empty()

Example

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(4)	**end**	(5,A),(7,B),(2,C),(8,D),(2,E)
find(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
findAll(2)	{(2,C),(2,E)}	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
erase(5)	—	(7,B),(2,C),(8,D),(2,E)
find(5)	**end**	(7,B),(2,C),(8,D),(2,E)

A List-Based Dictionary

- A log file or audit trail 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 list or array), in arbitrary order
- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find and erase 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)

The find, put, erase Algorithms

```
Algorithm find(k)
for each p in [S.begin(), S.end()) do
    if p.key() = k  then
        return p

Algorithm put(k, v)
Create a new entry e = (k, v)
p = S.insertBack(e)        {S is unordered}
return p

Algorithm erase(k):
for each p in [S.begin(), S.end()) do
    if p.key() = k then
        S.erase(p)
```

Hash Table Implementation

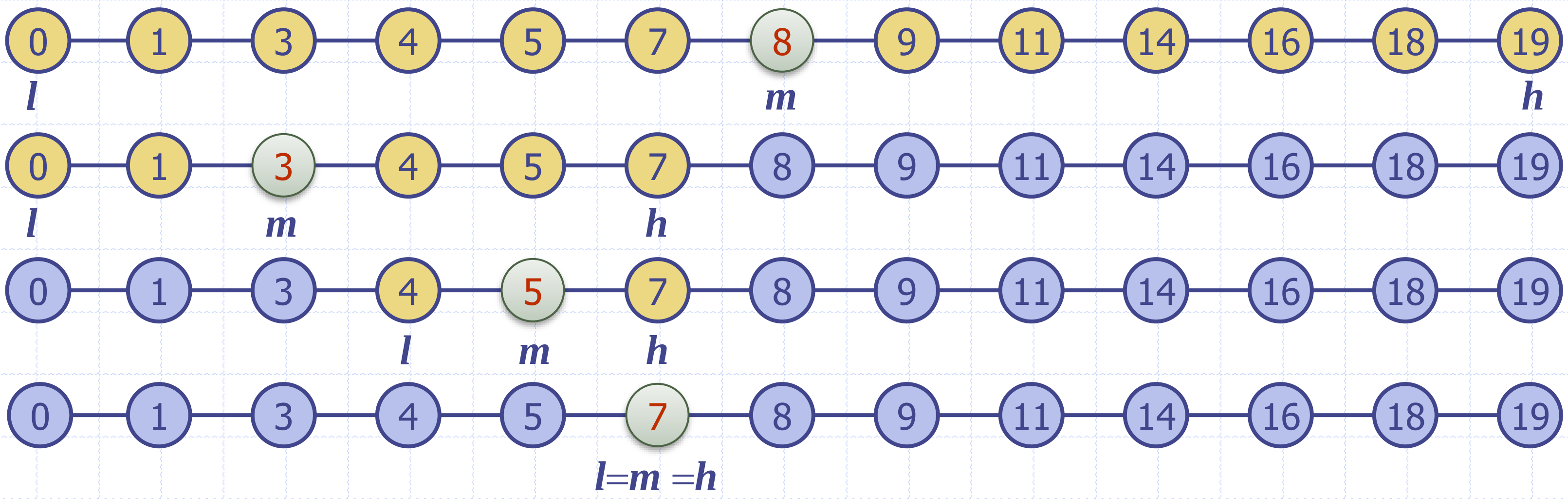
- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Search Table

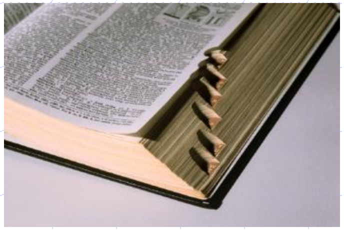
- A search table is a dictionary implemented by means of a sorted array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - find takes $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since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

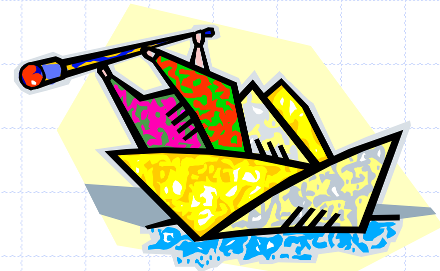
- Binary search performs operation find(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example: find(7)



Ordered Maps

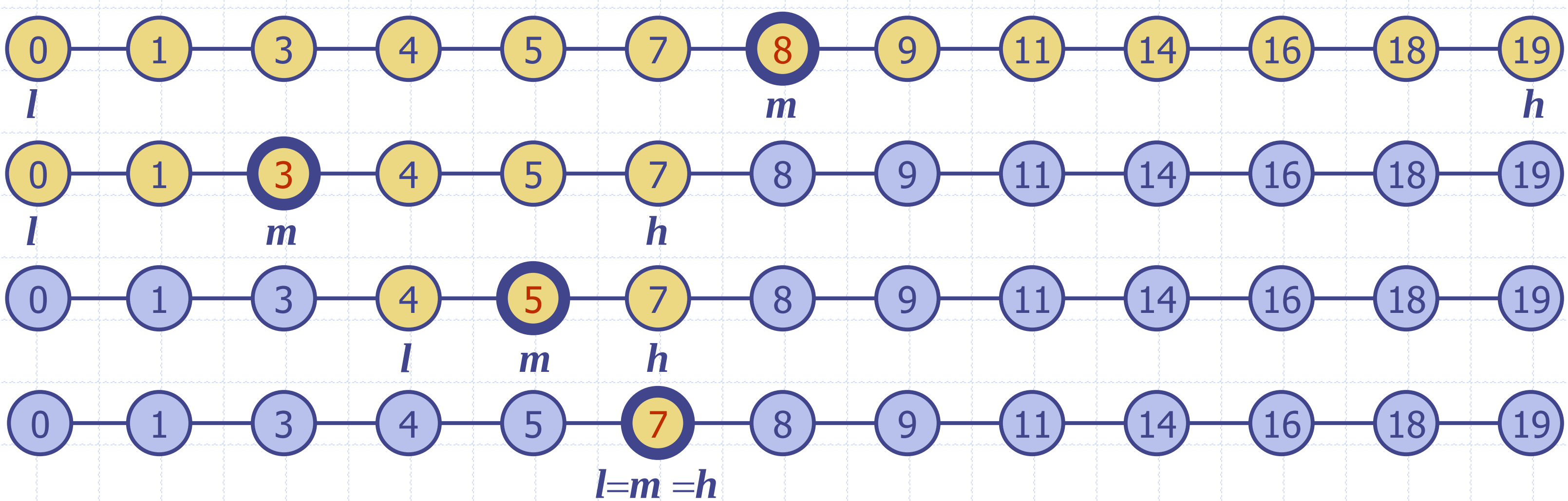


- Keys come from a total order
- Extension of Map ADT
- New functions are (each returns an iterator to an entry):
  - firstEntry(): smallest key in the map
  - lastEntry(): largest key in the map
  - floorEntry(k): largest key $\leq$ k
  - ceilingEntry(k): smallest key $\geq$ k
  - lowerEntry(k): largest key < k
  - higherEntry(k): smallest key > k
  - All return end if the map is empty



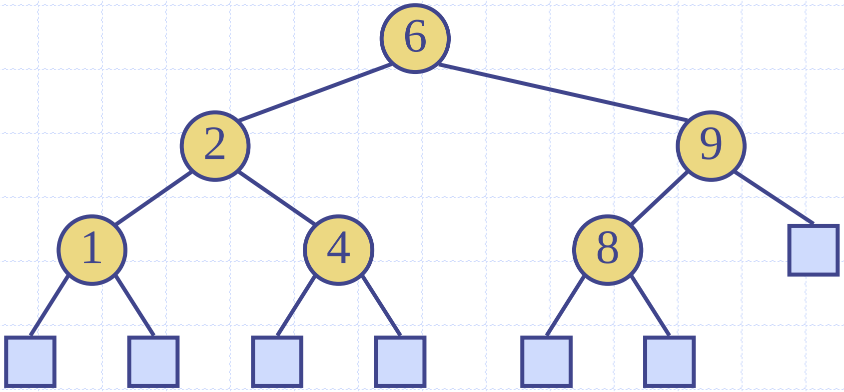
Binary Search

- Binary search can perform operations get, floorEntry, ceilingEntry, lowerEntry, and higherEntry on an ordered map implemented by means of an array-based sequence, sorted by key.
  - terminates after O(log n) steps
- Example: find(7)



Binary Search Trees

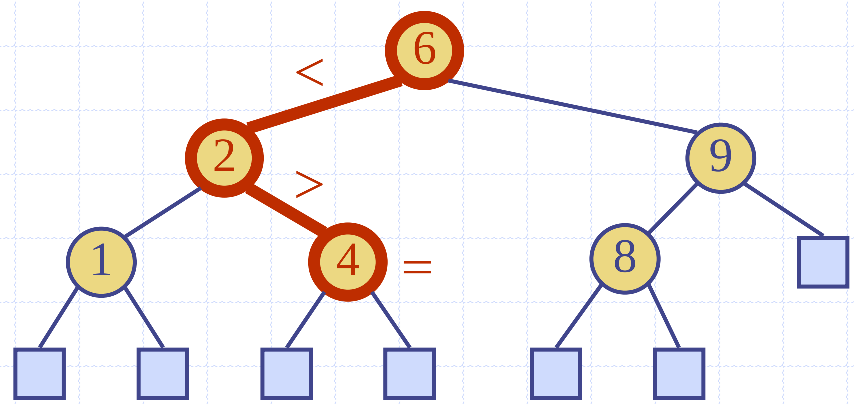
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes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



Search

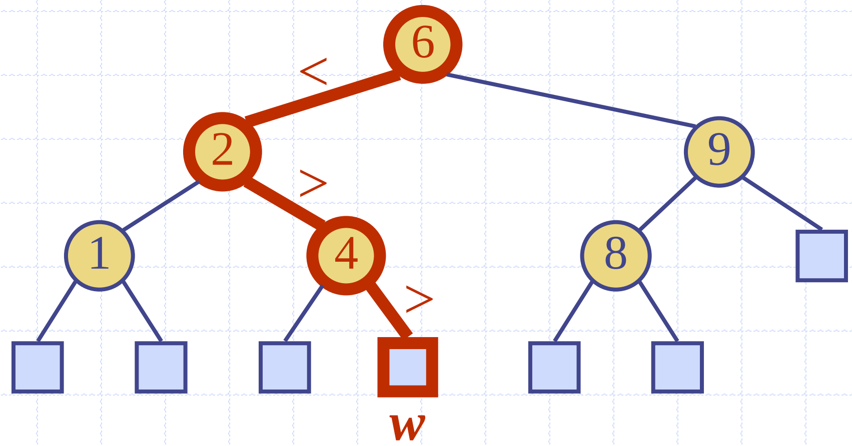
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry are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

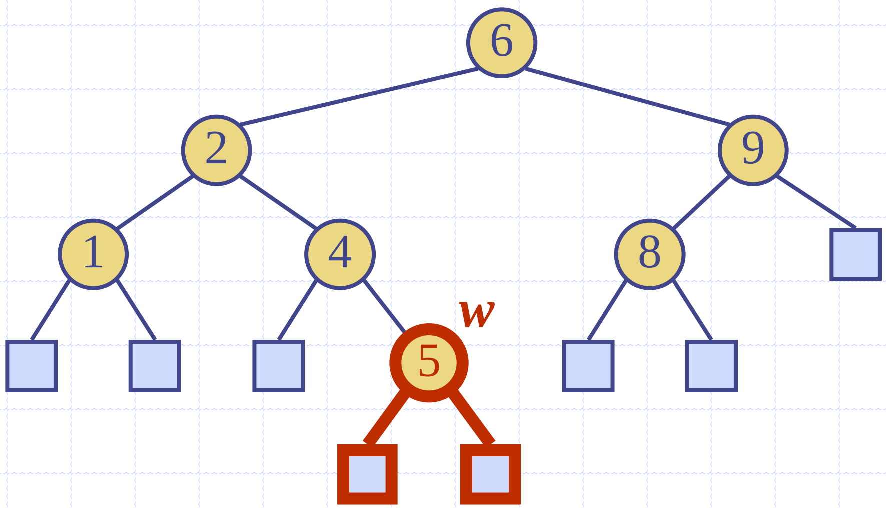
```
Algorithm TreeSearch(k, v)
    if v.isExternal ()
        return v
    if k < v.key()
        return TreeSearch(k, v.left())
    else if k = v.key()
        return v
    else { k > v.key() }
        return TreeSearch(k, v.right())
```



Insertion

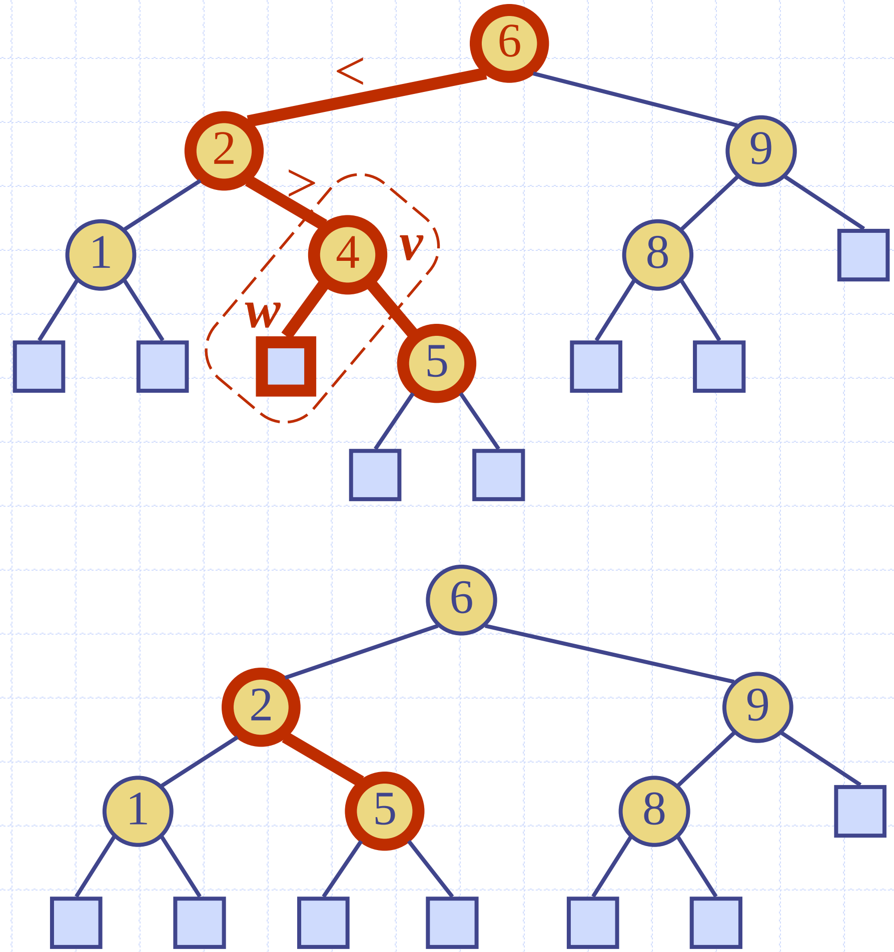
- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5





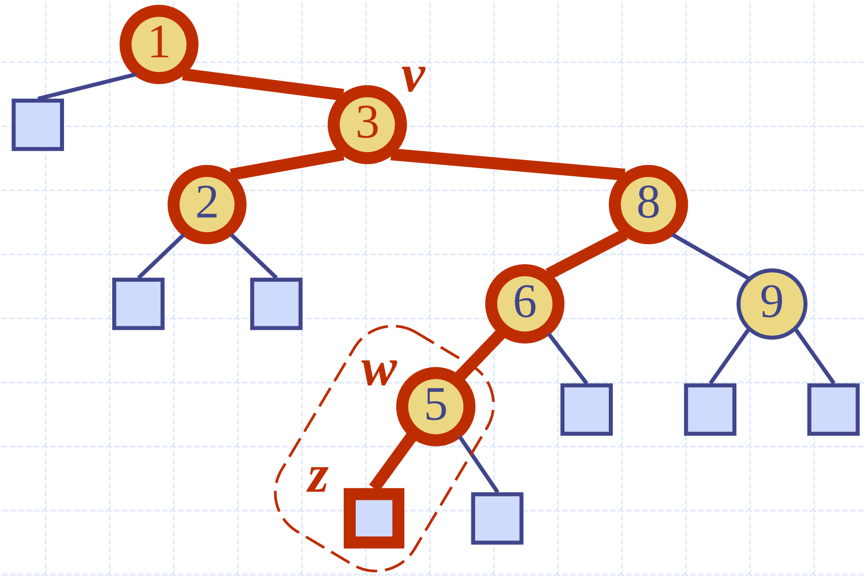
Deletion

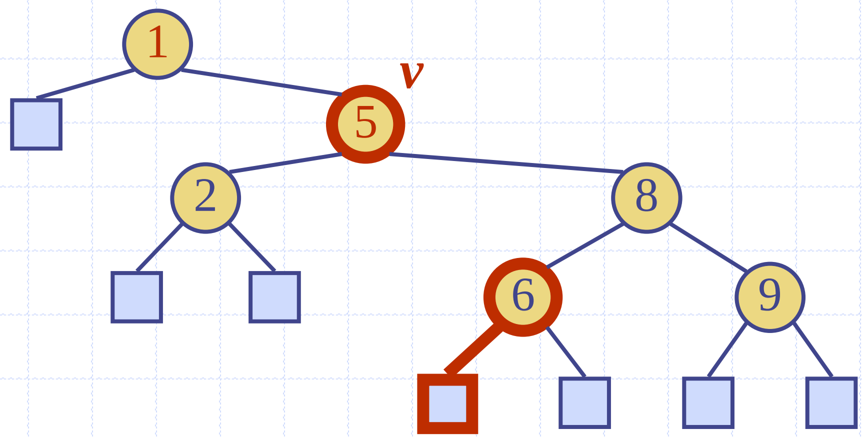
- To perform operation erase(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4



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