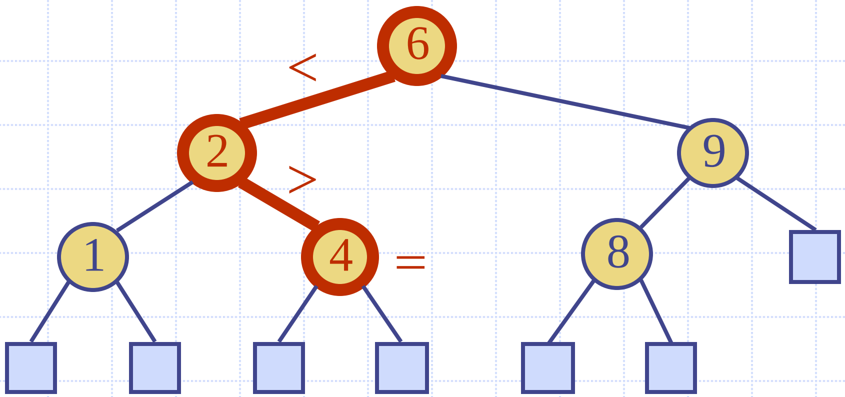
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Binary Search Trees



Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 10. Hash Tables, Maps, and Skip Lists**
- **Section 10.3 The Sorted Map ADT**
- **pp. 396-401**

- **Chapter 11. Search Tree Structures**
- **Sections 11.1-11.2**
- **pp. 423-442**

Learning Objectives

- To be able to understand and describe the Sorted Map ADT;
- To be able to analyze the complexity of the Sorted Map ADT methods;
- To be able to implement the Sorted Map ADT with a binary search tree;
- To be able to explain the update operations for a binary search tree;
- To be able to apply the Sorted Map ADT and binary search tree.

Motivation

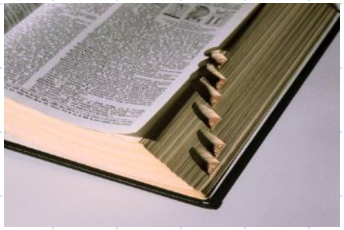
Suppose you have an array of integers and want to search for a particular integer k, how will you do it?

Motivation

- If the array is not sorted, then we need to scan all elements, hence $O(n)$.

- If the array is sorted, then we can do better. How?

Ordered Maps



- Keys are assumed to come from a total order.
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries:
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

The Sorted Map ADT (Sec.10.3)

The Sorted Map ADT includes all methods of the Map ADT, plus the following.

- firstEntry( ): Returns the entry with smallest key value (or null, if the map is empty).
- lastEntry( ): Returns the entry with largest key value (or null, if the map is empty).
- ceilingEntry(k): Returns the entry with the least key value greater than or equal to k (or null, if no such entry exists).
- floorEntry(k): Returns the entry with the greatest key value less than or equal to k (or null, if no such entry exists).

The Sorted Map ADT

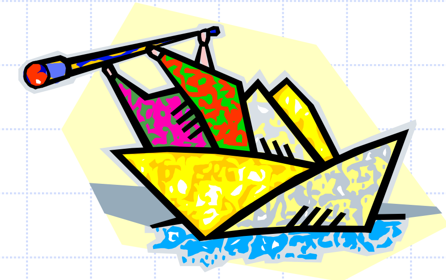
- lowerEntry(k): Returns the entry with the greatest key value strictly less than k (or null, if no such entry exists).
- higherEntry(k): Returns the entry with the least key value strictly greater than k (or null if no such entry exists).
- subMap(k_1, k_2): Returns an iteration of all entries with key greater than or equal to k_1, but strictly less than k_2.

Sorted Search Tables

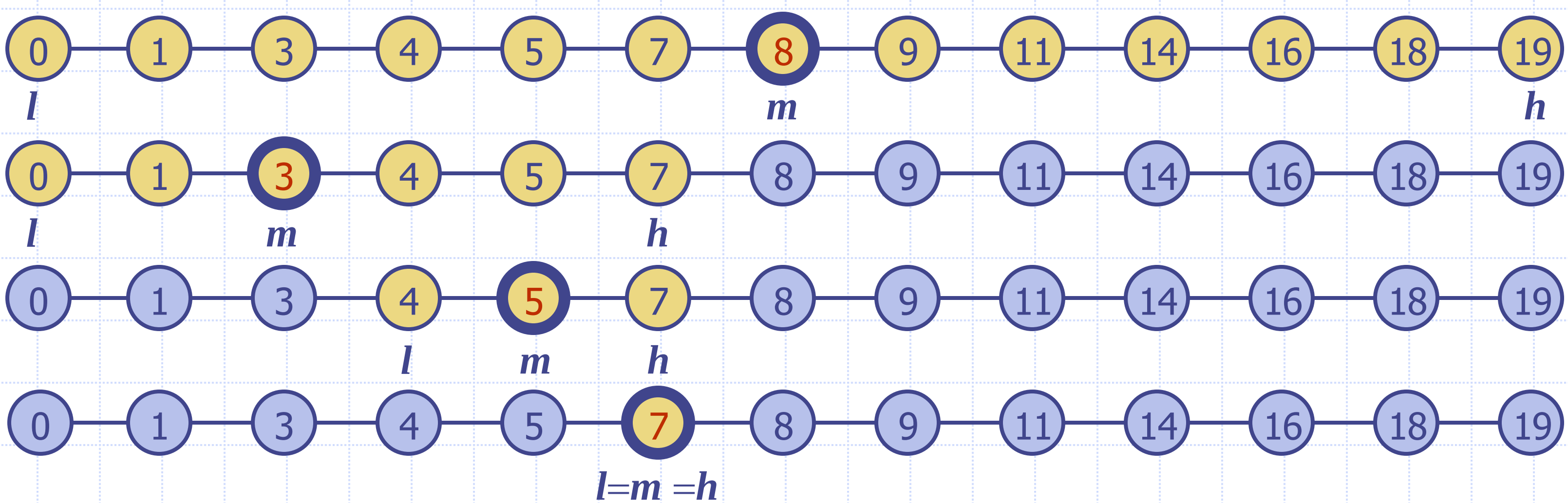
- We store the map's entries in an array list *A* so that they are in increasing order of their keys.
- We refer to this implementation as a ***sorted search table***.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Binary Search



- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after $O(\log n)$ steps
- Example: find(7)



Sorted 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:
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift n items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift n items to compact the items after the removal
- The lookup table is effective only for ordered maps of *small size* or for maps on which *searches* are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Performance of a sorted map
(implemented using a sorted search table)

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

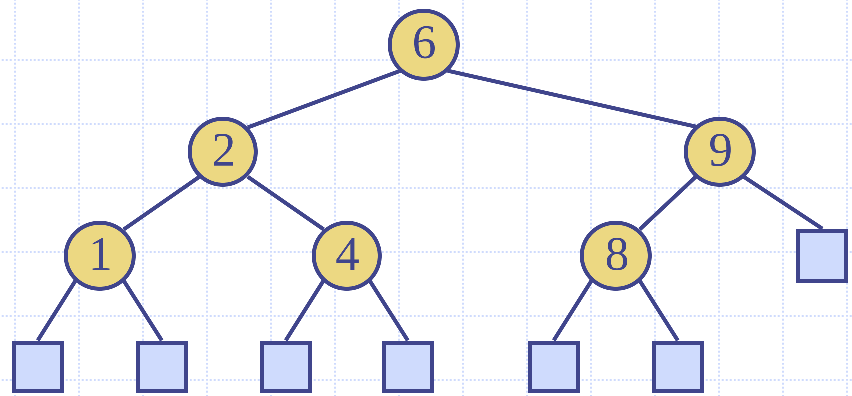
Motivation

- Binary search on ordered arrays is efficient: $O(\log_2 n)$
- However, insertion or removal of an item in an ordered array is slow: $O(n)$
- Ordered arrays are best suited for *static* searching, where search space does not change.
- Binary search trees can be used for *efficient dynamic searching*.

Binary Search Trees

A binary search tree is a *proper* binary tree storing keys (or key-value entries) at its *internal nodes* and satisfying the following property:

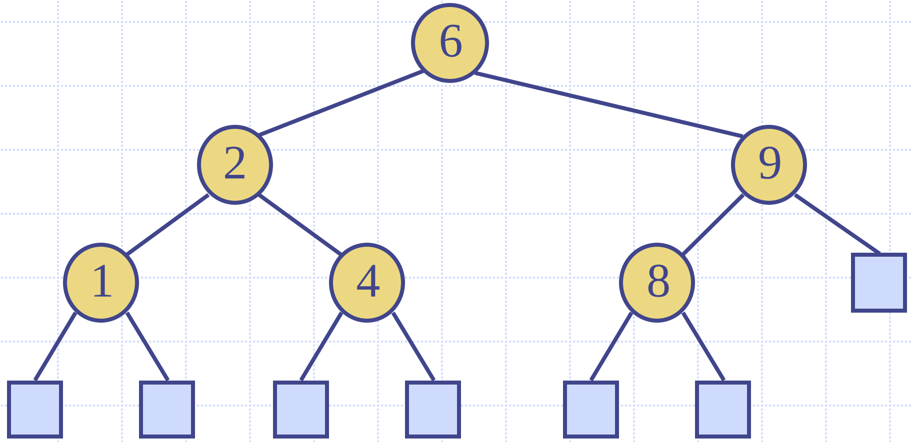
- Let u, v, and w be three nodes such that u is in the left *subtree* of v and w is in the right *subtree* of v. We have $key(u) \leq key(v) \leq key(w)$
- Assuming there are no duplicate keys, we have $key(u) < key(v) < key(w)$



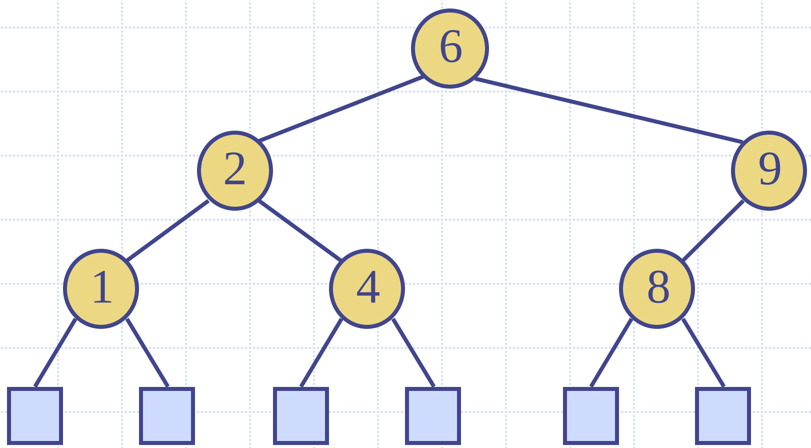
Binary Search Trees

External nodes do not store items

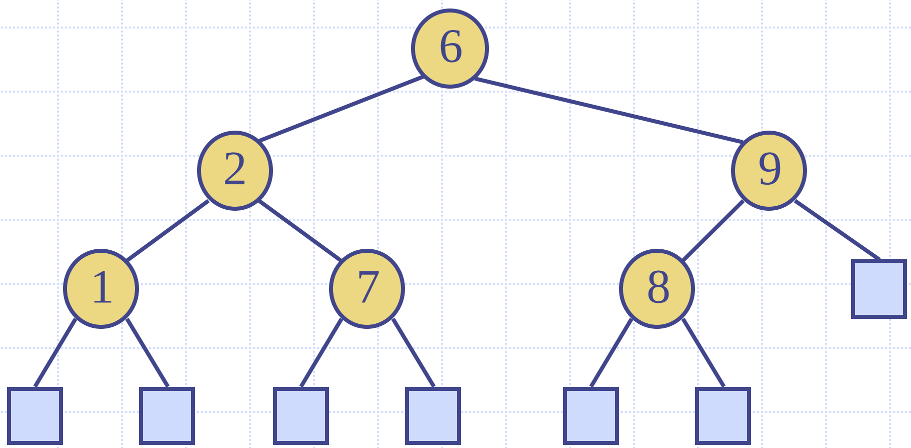
- and likely are not actually implemented, but are just null links from the parent



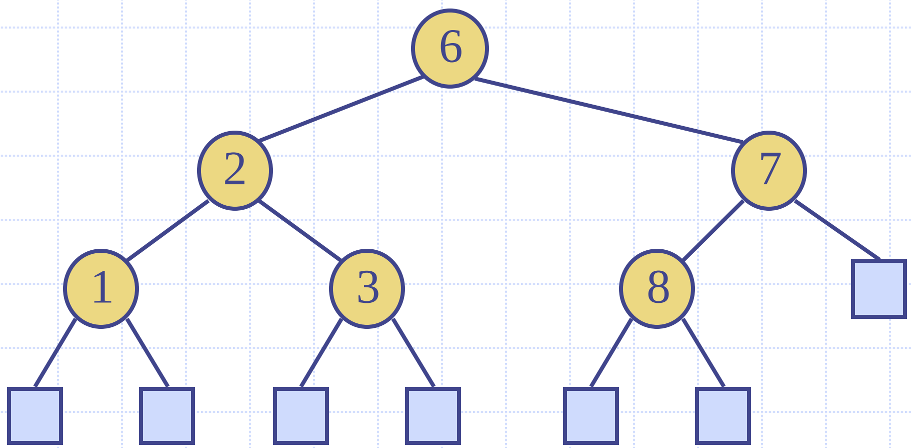
Is this a binary search tree?



Is this a binary search tree?



Is this a binary search tree?

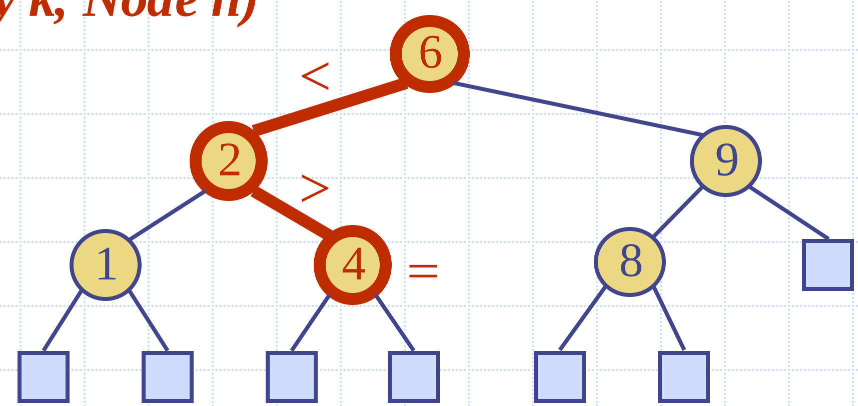


Search

- 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

- Exercise: Write down the pseudo-code of the Search method.

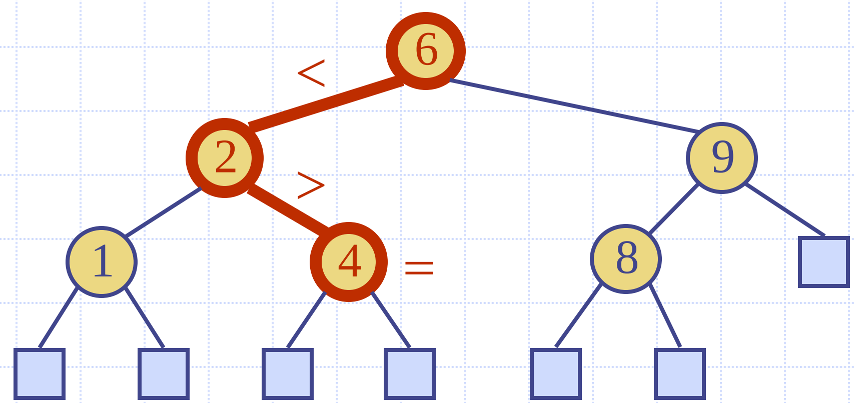
Algorithm ***Node TreeSearch(Key k, Node n)***



Search

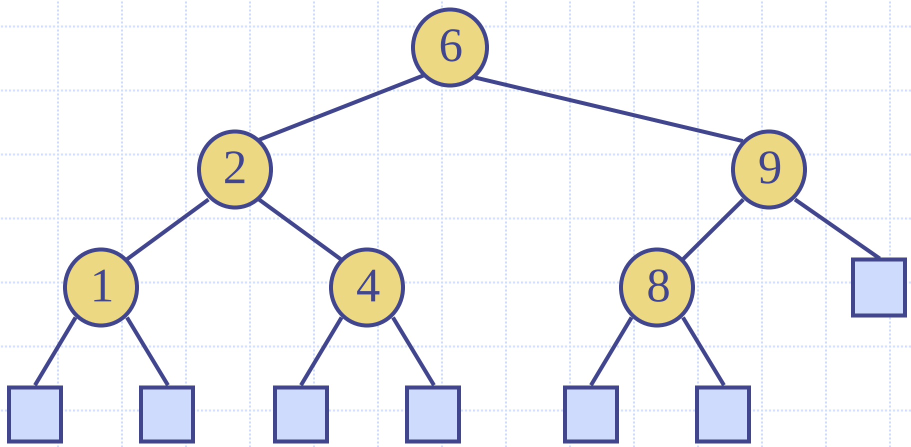
```
Algorithm Node TreeSearch(Key k, Node n)
    if n.isExternal () // or "if n == null"
        return null
    if k < n.key()
        return TreeSearch(k, n.left())
    else if k = n.key()
        return n
    else // k > n.key()
        return TreeSearch(k, n.right())
```

- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar



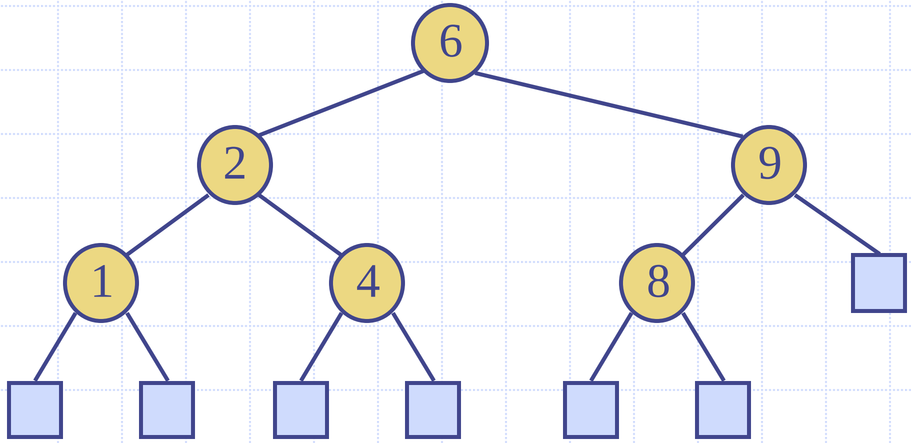
Fundamental Property of Binary Search Trees

- What is an inorder traversal of a tree?
- Exercise: what does an inorder traversal of the following search tree produce?



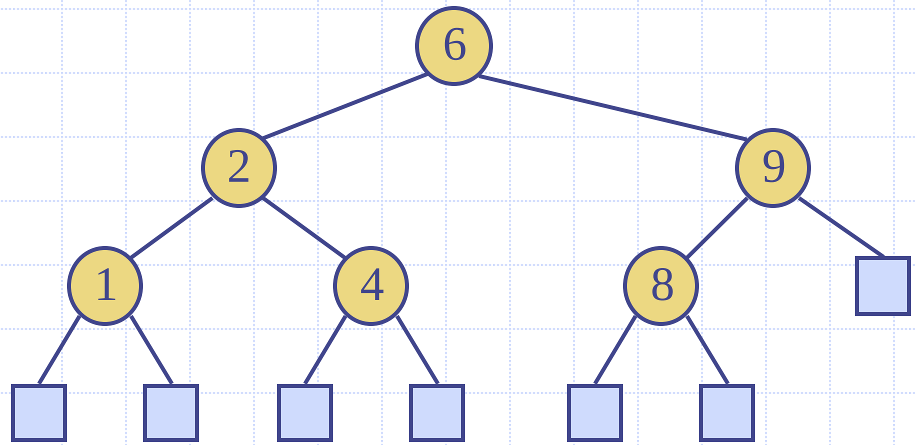
Fundamental Property of Binary Search Trees

An inorder traversal of a binary search tree visits the keys in increasing order.



Fundamental Property of Binary Search Trees

- How to access the minimal key?
- How to access the maximal key?
- How to access the largest key less than or equal to k?
- How to access the smallest key greater than or equal to k?



Insertion

- Exercise: How to do insertion, $put(k, o)$?

- Have to insert k where a $get(k)$ would find it.
- So natural that $put(k, o)$ starts with $get(k)$

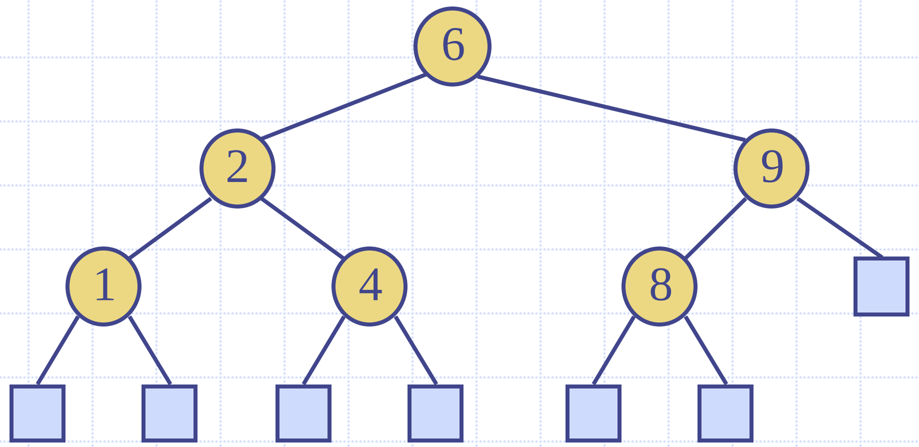
Insertion

- Exercise: How to do insertion, $put(k, o)$?

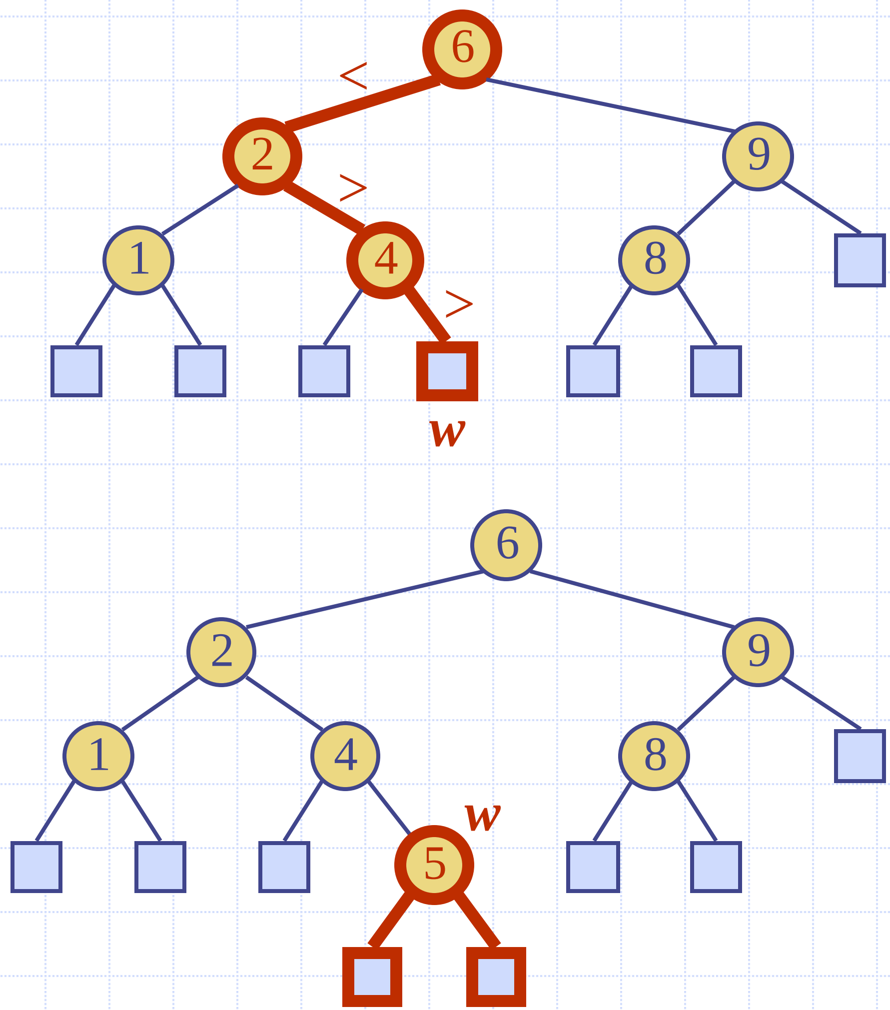
- We search for key k (using TreeSearch)
- Assume k is already in the tree then just replace the value.
- Otherwise, let w be the leaf reached by the search, we insert k at node w and expand w into an internal node

Insertion

- Example: insert 5

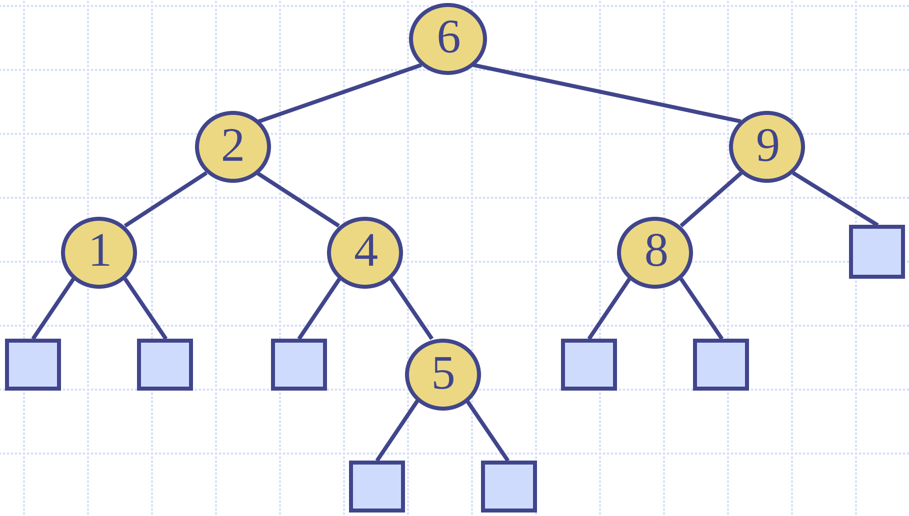


Insertion



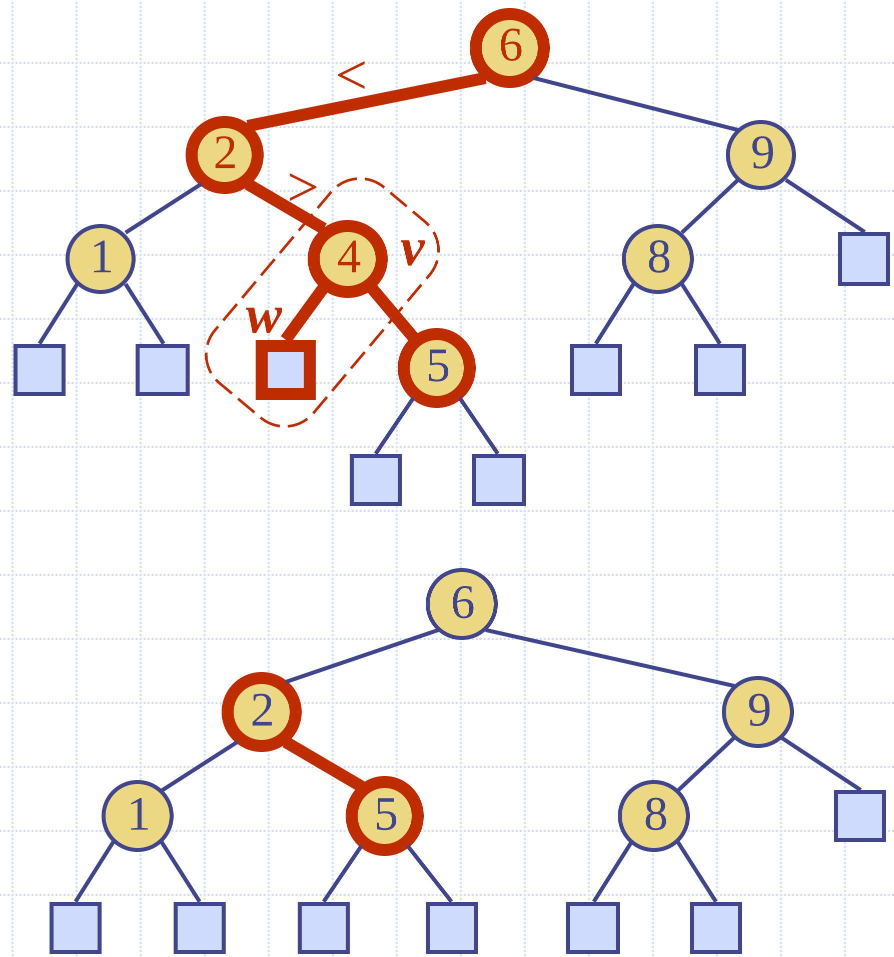
Deletion

- How can we perform the operation $remove(k)$?
  - E.g., remove 7, 5, 4, or 2



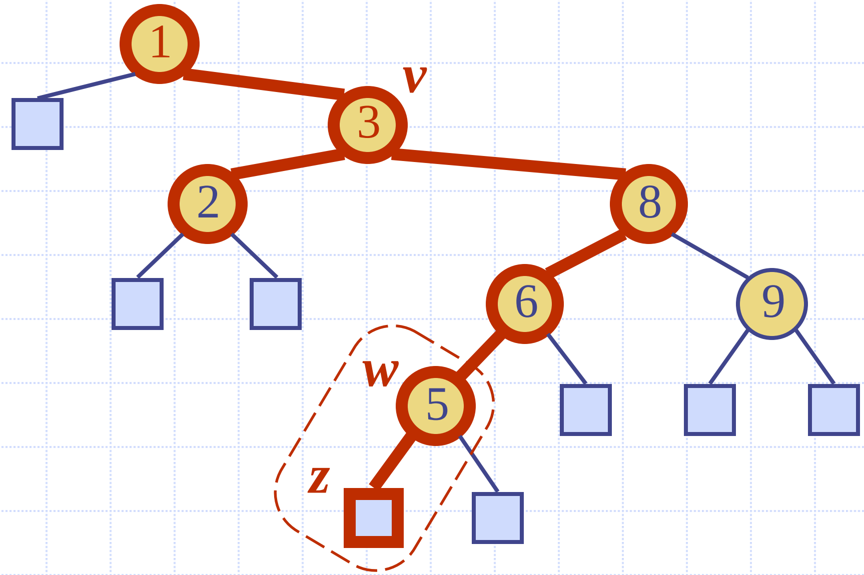
Deletion

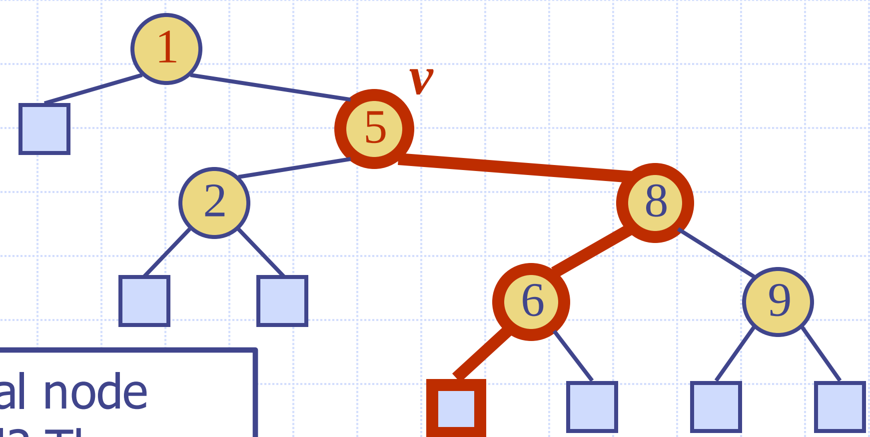
- To perform operation *remove*($\boldsymbol{k}$), we search for key $\boldsymbol{k}$
- Assume key $\boldsymbol{k}$ is in the tree, and let $\boldsymbol{v}$ be the node storing $\boldsymbol{k}$
- If node $\boldsymbol{v}$ has a leaf child $\boldsymbol{w}$, we remove $\boldsymbol{v}$ and $\boldsymbol{w}$ from the tree with operation *removeExternal*($\boldsymbol{w}$), which removes $\boldsymbol{w}$ 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





Self-study: can we let w be the internal node that precedes v in an inorder traversal? Then how to perform "delete"? pp. 428-429

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, put and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case