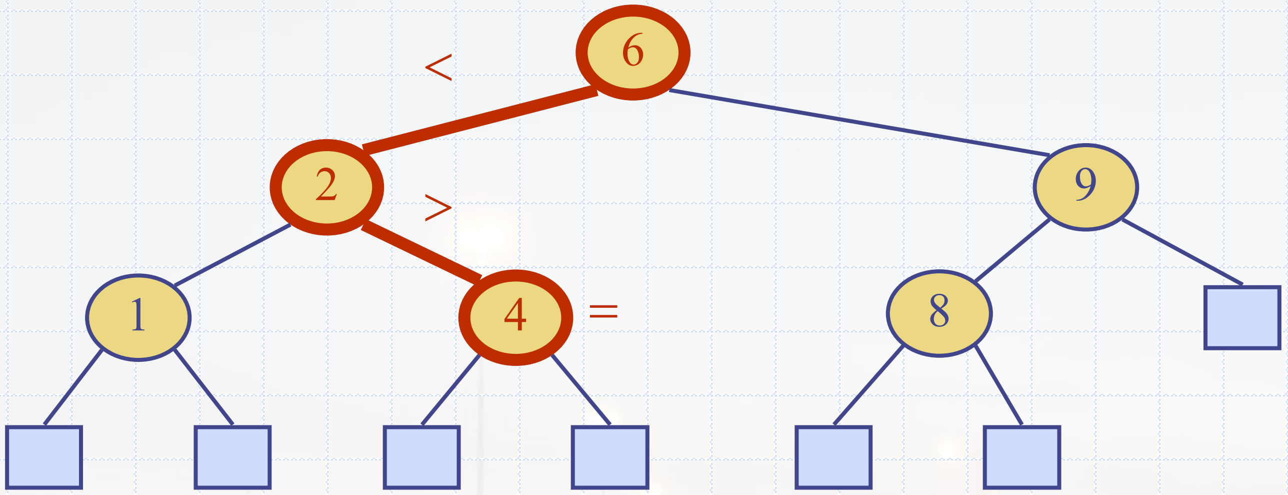
# Binary Search Trees

*Dr. Aiman Hanna*
**Department of Computer Science & Software Engineering**
**Concordia University, Montreal, Canada**

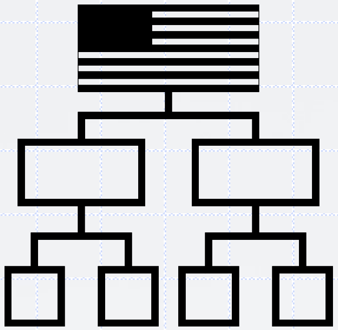
These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

# Coverage

- Section 10.1: Binary Search Trees



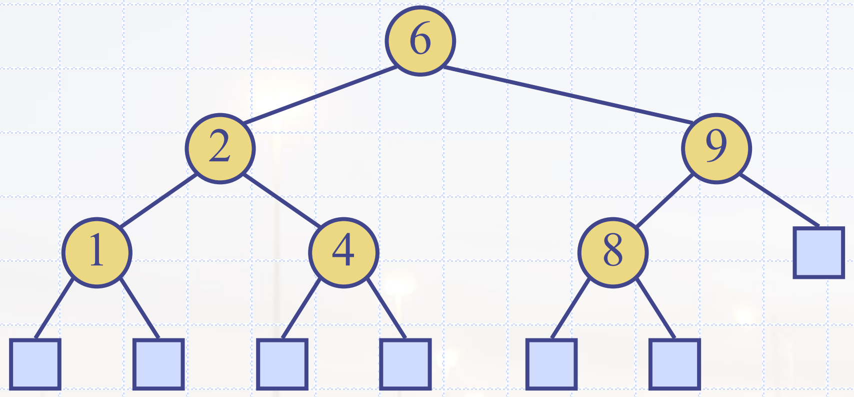
# Binary Search Trees



- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let $u$, $t$, and $w$ be three nodes such that $u$ is in the left subtree of $t$ and $w$ is in the right subtree of $t$. We have $key(u) \leq key(t) \leq key(w)$
- External nodes do not store items. They are there to guarantees that the tree is a proper binary tree, which simplifies other operations.

# Binary Search Trees

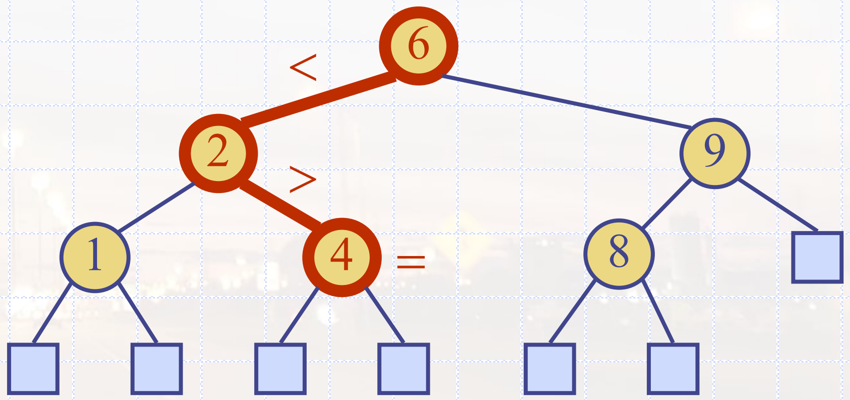
- An inorder traversal of a binary search trees visits the keys in increasing order



# Search

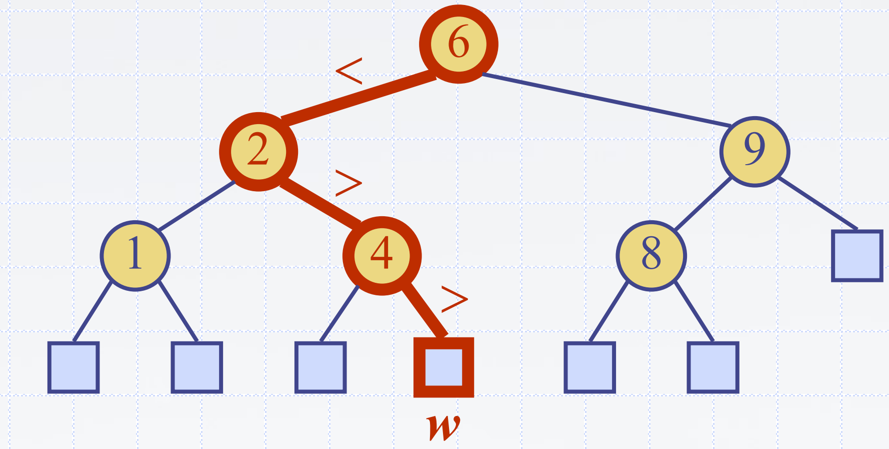
- To search for a key $k$, we trace a downward path starting at the root
- The next node visited depends on the comparison of $k$ with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry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

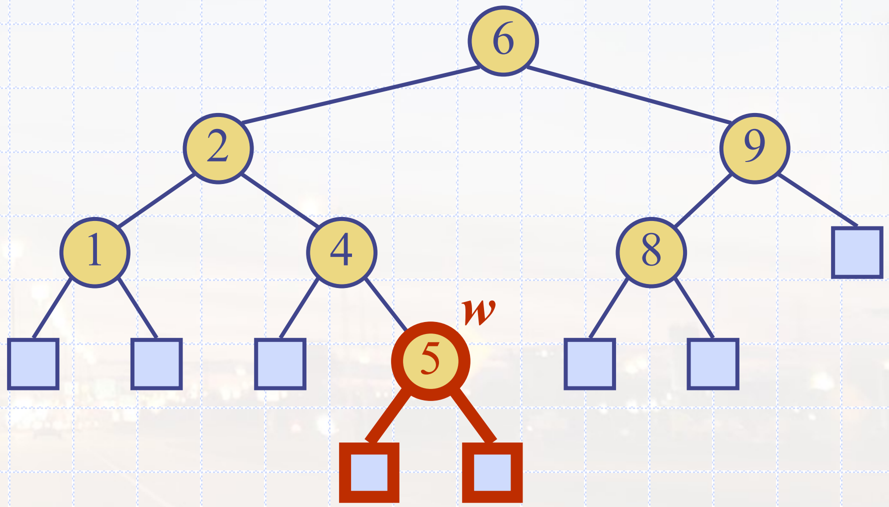
```
Algorithm TreeSearch(k, w)
    if T.isExternal (w)
        return null
    if k < key(w)
        return TreeSearch(k, T.left(w))
    else if k = key(w)
        return w
    else { k > key(w) }
        return TreeSearch(k, T.right(w))
```



# Insertion

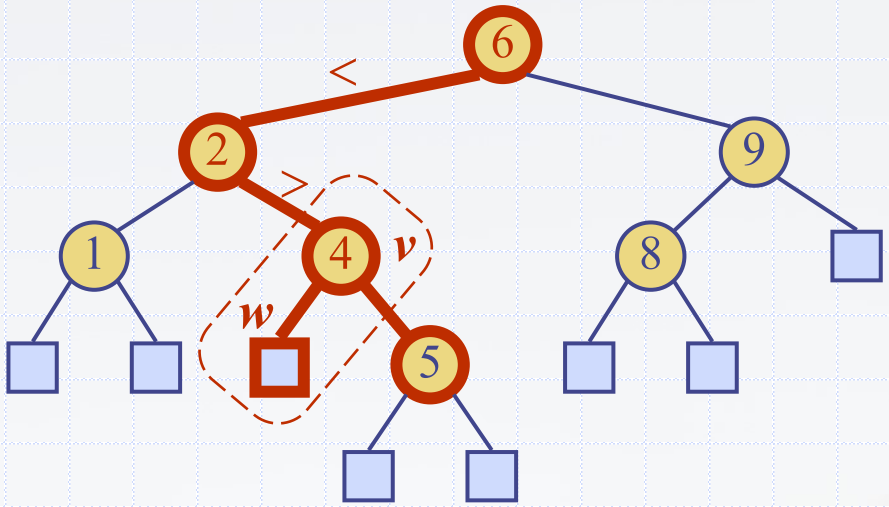
- To perform operation put(*k*, *o*), we search for key k (using TreeSearch)
- Assume *k* is not already in the tree, and let *w* be the leaf reached by the search
- We insert *k* at node *w* and expand *w* into an internal node
- Example: insert (5)

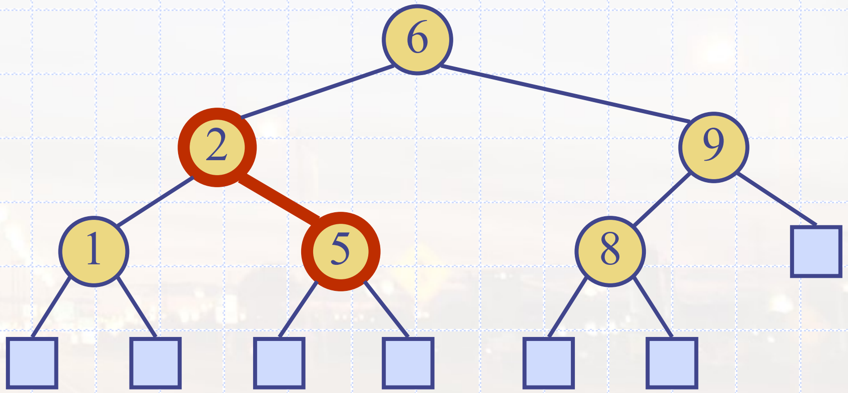




# Deletion

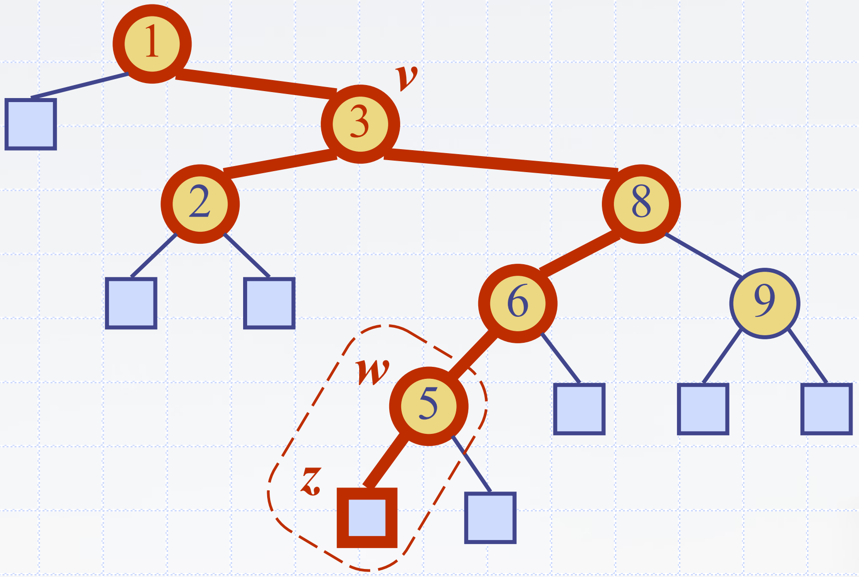
- To perform operation remove($k$), we search for key $k$
- Assume key $k$ is in the tree, and let $v$ be the node storing $k$
- If node $v$ has a leaf child $w$, we remove $v$ and $w$ from the tree with operation removeExternal($w$), which removes $w$ and its parent
- Example: remove (4)

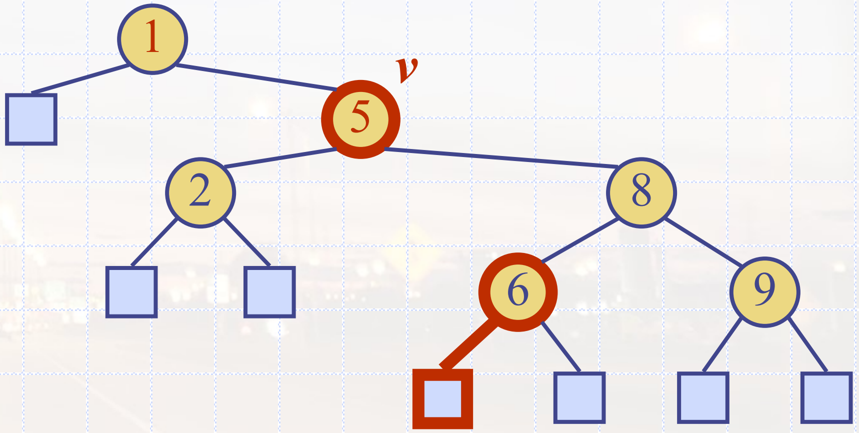




# Deletion (cont.)

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