Comparison of data structs and algos so far…

	Find/Search Any	Insert	Delete Any	Notes
Stack and Standard Queue	O(n)	O(1)	O(n)	
Unsorted List Priority Queue	O(n)	O(1)	O(n)	
Sorted List Priority Queue	O(n)	O(n)	O(n)	
Heap Priority Queue	O(n)	O(log n)	O(n)	Designed for operations on "min" in O(log n)
Skip List	log n high prob.	log n high prob.	log n high prob.	Randomized insertion algorithm
Hash Table	1 expected	1 expected	1 expected	O(n) worst case

Comparison of data structs and algos so far…

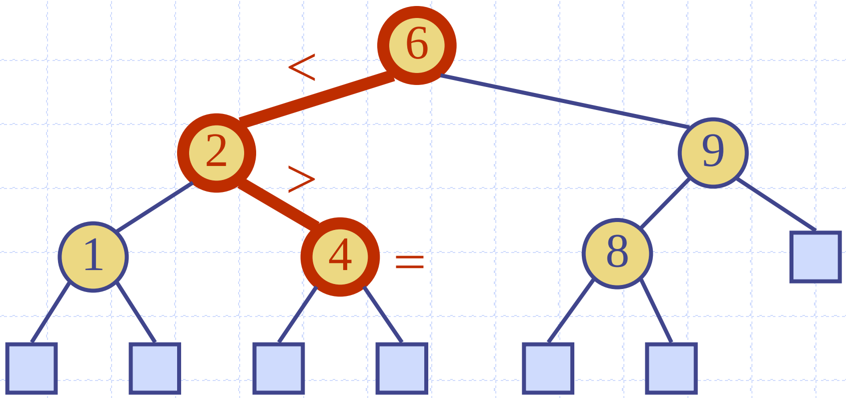
	Find/Search Any	Insert	Delete Any	Notes
Stack and Standard Queue	O(n)	O(1)	O(n)	
Unsorted List Priority Queue	O(n)	O(1)	O(n)	
Sorted List Priority Queue	O(n)	O(n)	O(n)	
Heap Priority Queue	O(n)	O(log n)	O(n)	Designed for operations on "min" in O(log n)
Skip List	log ***n*** high prob.	log ***n*** high prob.	log ***n*** high prob.	Randomized insertion algorithm
Hash Table	1 expected	1 expected	1 expected	O(n) worst case

Find/Search

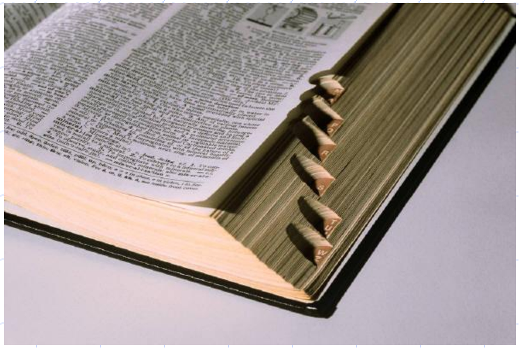
- Performance of delete depends on find/search
  - (performance of insert might as well…)
- Thus, we focus on improving find/search
- This brings us to…

Binary Search Trees

(and several others methods eventually…)

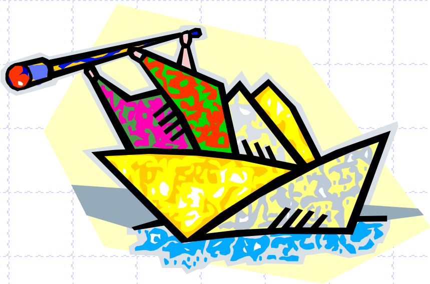


Assumptions

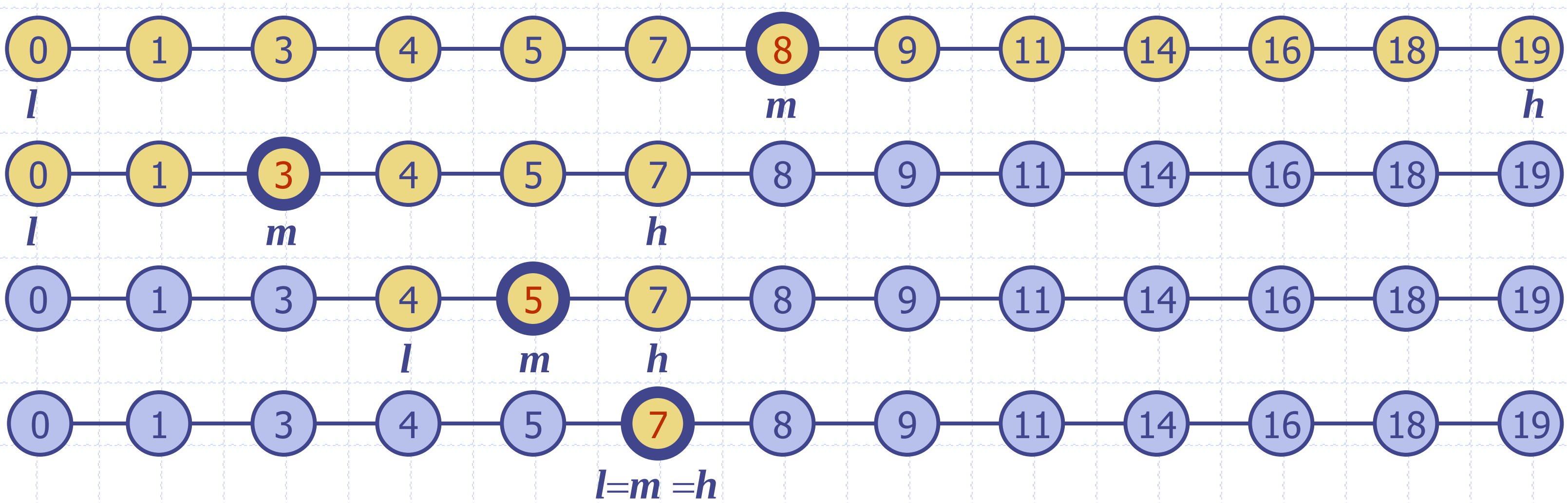


- We have an "ordered dictionary"
  - Keys are assumed to come from a total order
    - E.g., you can compare any key to any key and get a proper ordering
      - (this is as opposed to a partial ordering, where only adjacent keys can be compared with other keys)
  - New operations:
    - closestBefore(k)
    - closestAfter(k)

Binary Search (array-based)



- Perform 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 O(log n) steps
- Example: find(7)



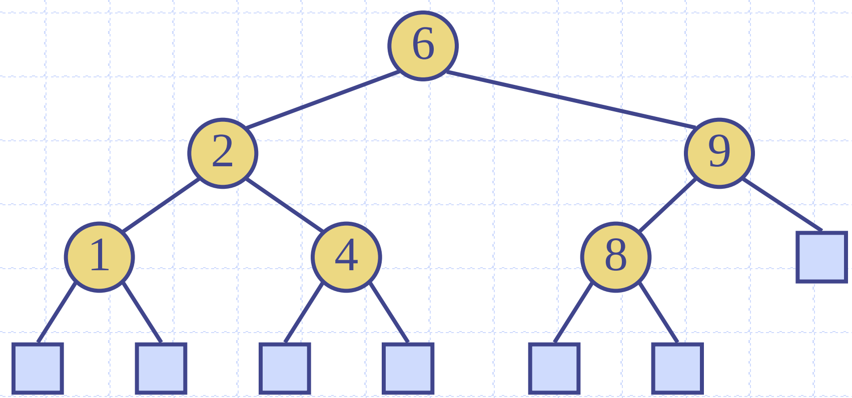
Lookup Table

- A lookup table is a dictionary implemented by means of a sorted sequence and uses binary search
  - 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:
    - $O(\log n)$ time (using binary search)
  - insertItem:
    - $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - removeElement:
    - $O(n)$ 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 for small dictionaries on which searches are the most common operations, while insertions and removals are rare

Binary Search Tree

- A binary search tree is a binary tree storing 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

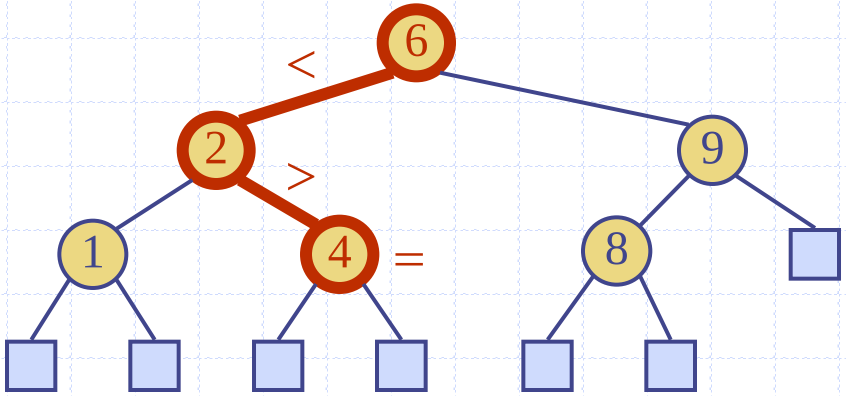
- Thus, how do you visit all keys in increasing order?
  - inorder traversal…



Find/Search

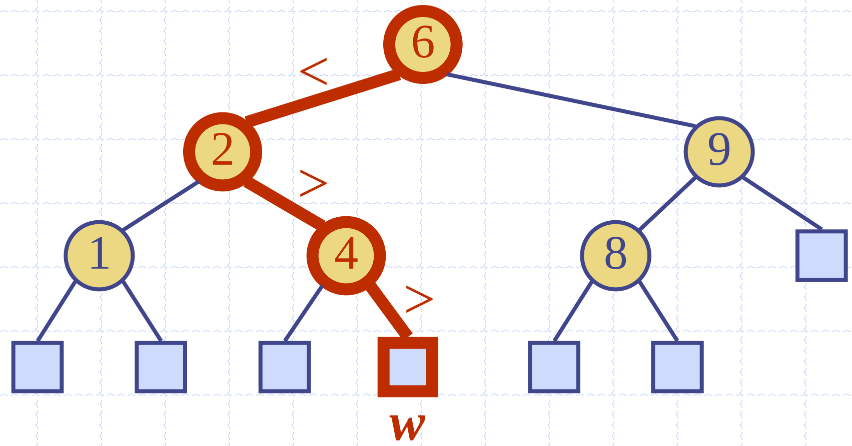
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return a null position
- Example: find(4)

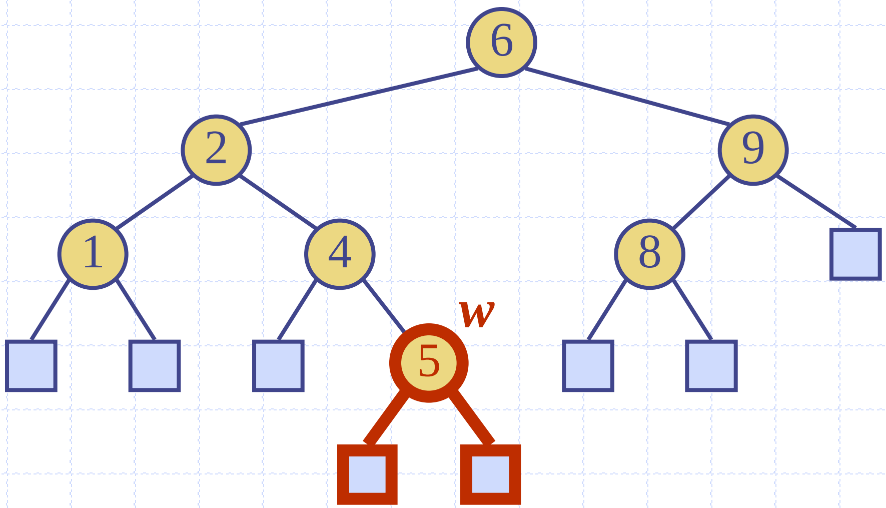
```
Algorithm find (k, v)
    if T.isExternal (v)
        return Position(null)
    if k < key(v)
        return find(k, T.leftChild(v))
    else if k = key(v)
        return Position(v)
    else { k > key(v) }
        return find(k, T.rightChild(v))
```



Insertion

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



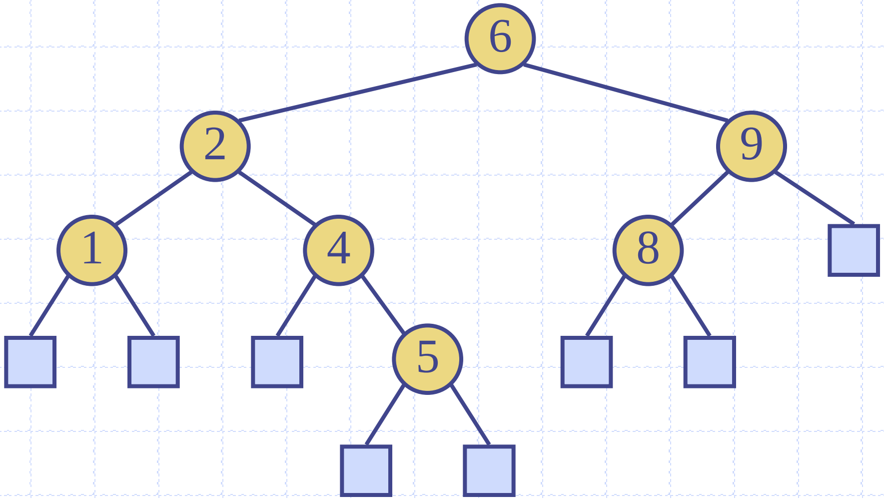


Deletion

- Three cases:
  - Zero children
  - One child
  - Two children

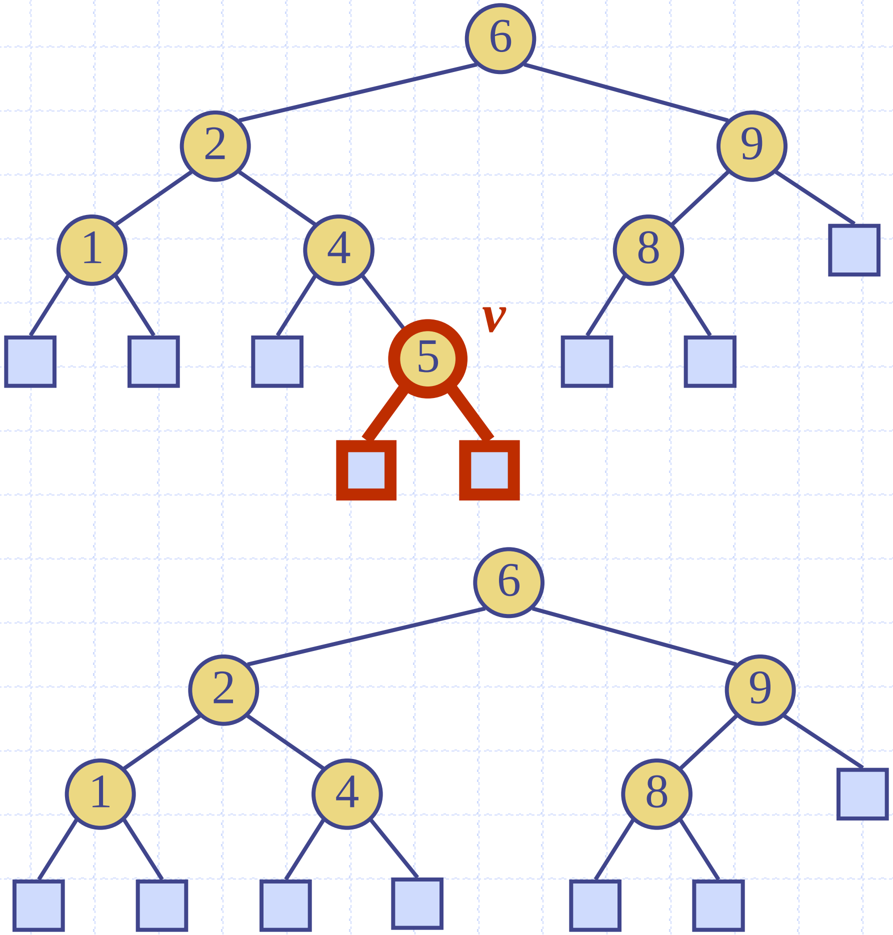
Deletion: zero children

- Must be a leaf node – simple (e.g., remove 5)



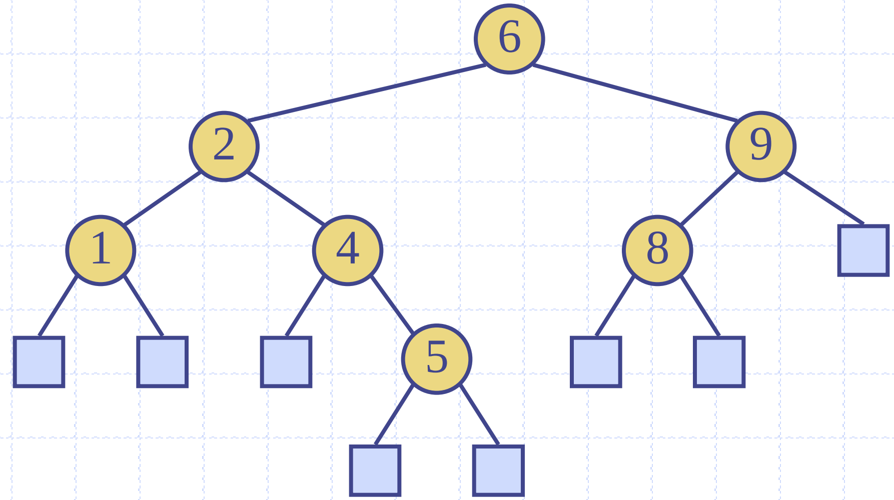
Deletion: zero children

- Must be a leaf node – simple (e.g., remove 5)
  - Assume key k is in tree, and let v be the node storing k
  - We search for key k
  - Remove node



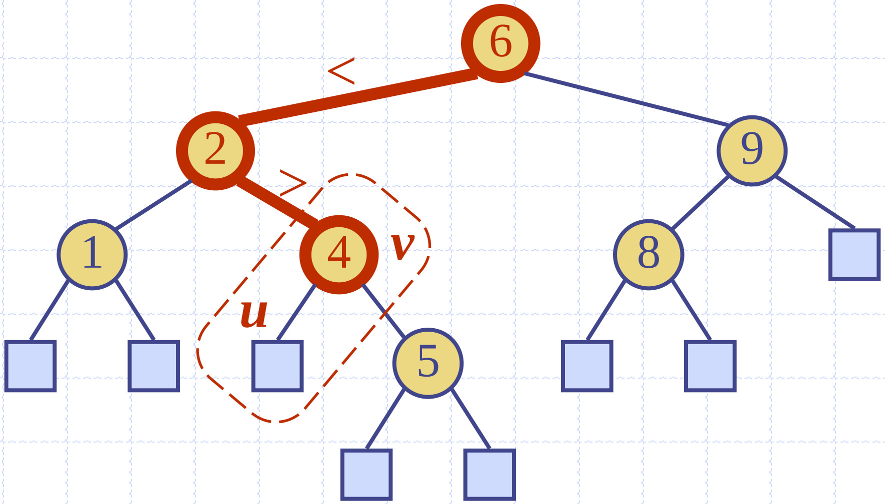
Deletion: one child

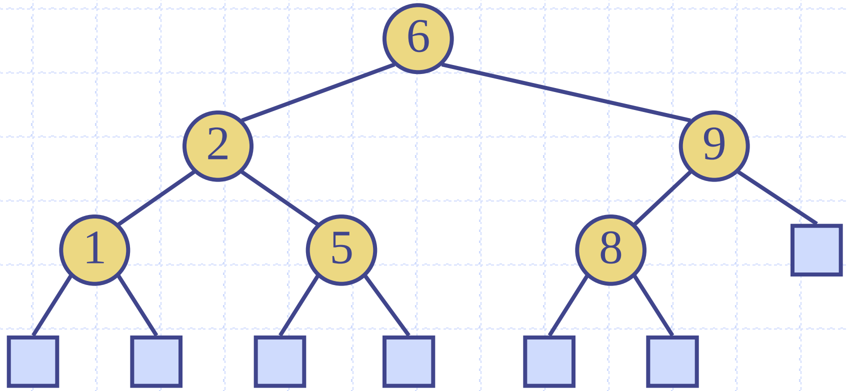
- To perform operation, we search for key ***k*** (e.g., remove 4)



Deletion: one child

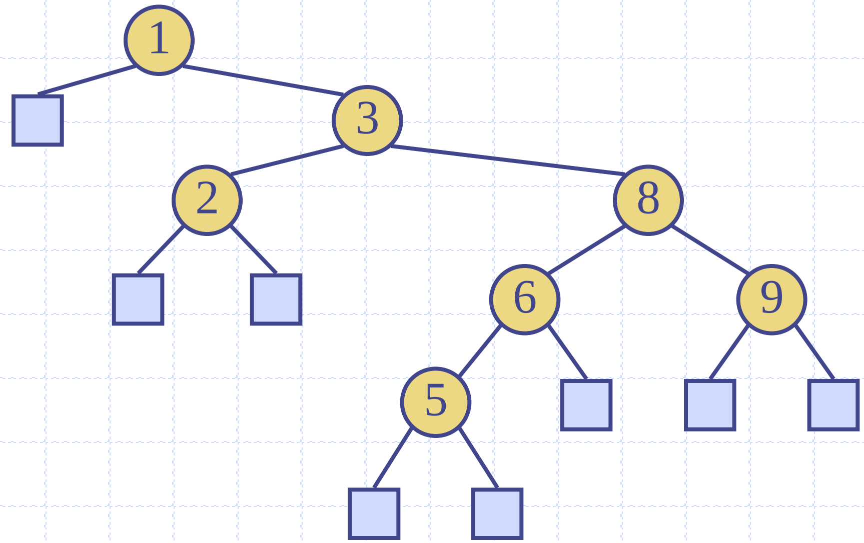
- To perform operation, we search for key k (e.g., remove 4)
- Assume key k is in tree, and let v be the node storing k
- If node v has **one** leaf child u, we remove v and u from the tree with operation removeAboveExternal(u)





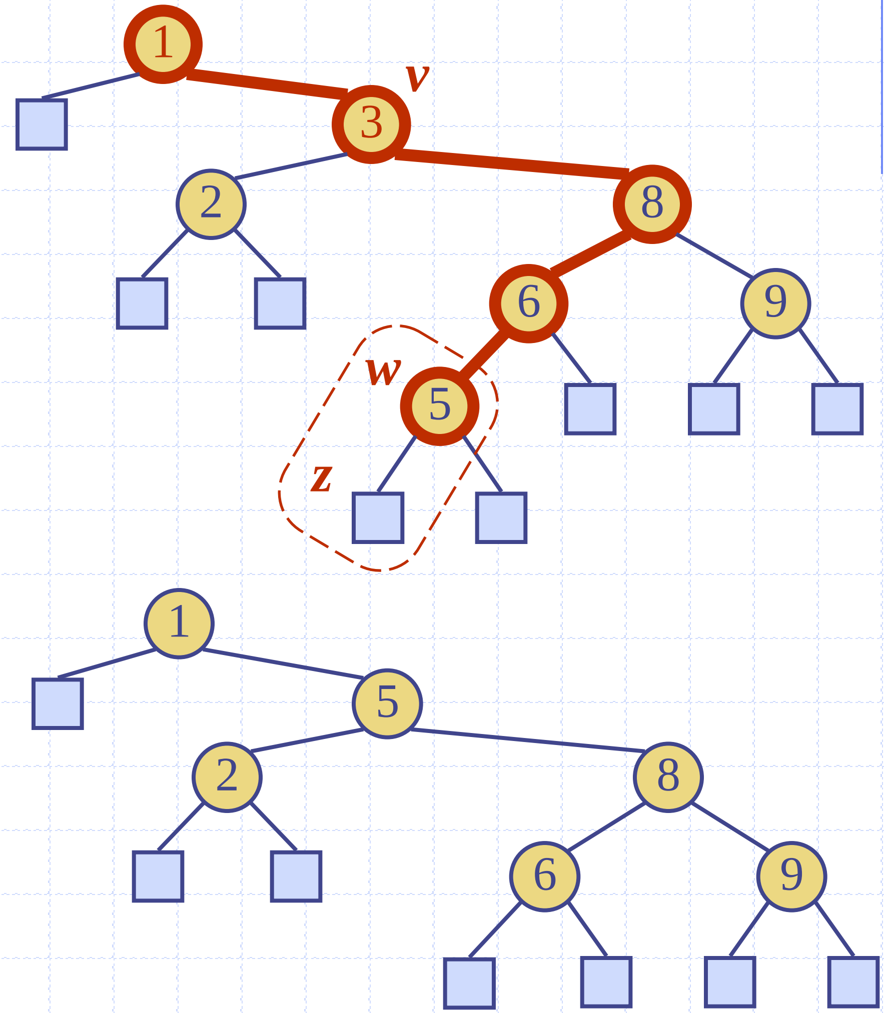
Deletion: two children

- What if the key k to be removed has **two** internal nodes as children, e.g. "remove 3"
  - we find the internal node w that follows v in an inorder traversal



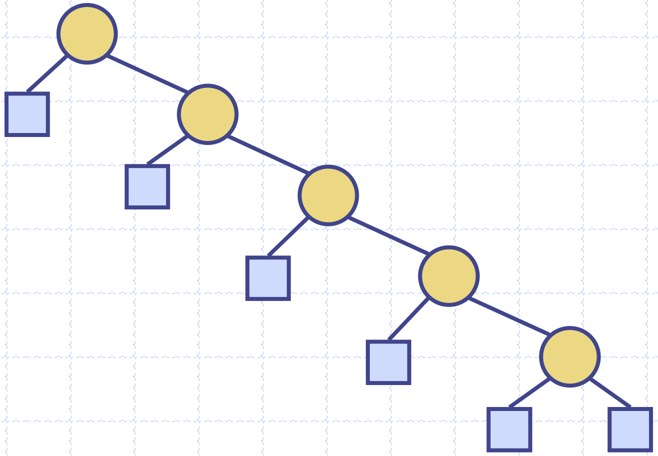
Deletion: two children

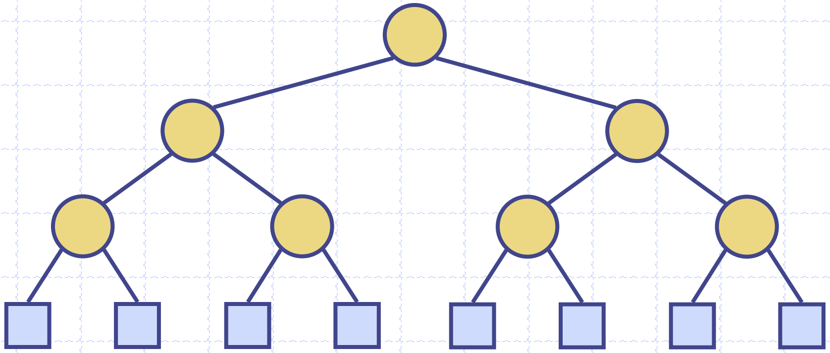
- What if the key k to be removed has **two** internal nodes as children, e.g. "remove 3"
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)



Performance

- A dictionary with n items implemented with a binary search tree of height h
  - Space is:
    - $O(n)$
  - Time findElement() , insertItem() and removeElement() is:
    - $O(h)$ time
- Height h is $O(n)$ in the worst case and $O(\log n)$ in the best case
- How can we "prevent" $O(n)$ height?





Searching++

- That brings us to our next (more complex) set of searching algorithms and data structures:
  - 2-3-4 trees
  - (AVL trees)
  - Red-black trees