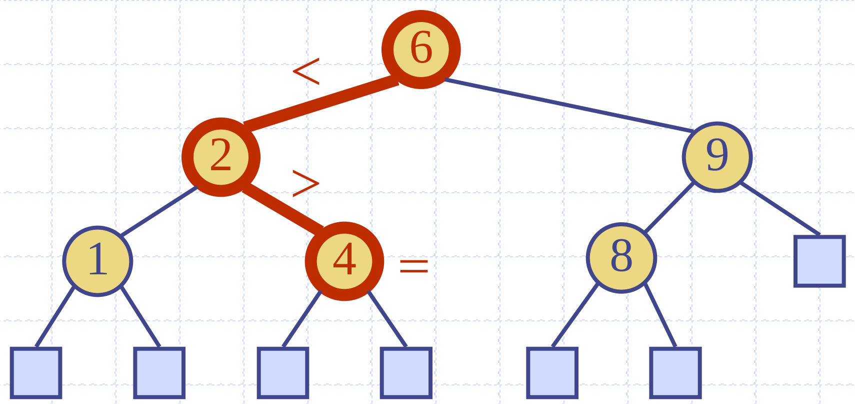
Lecture 9: Search Tree Structures

(Chapter 10, Section 10.1 from the book)

Agenda

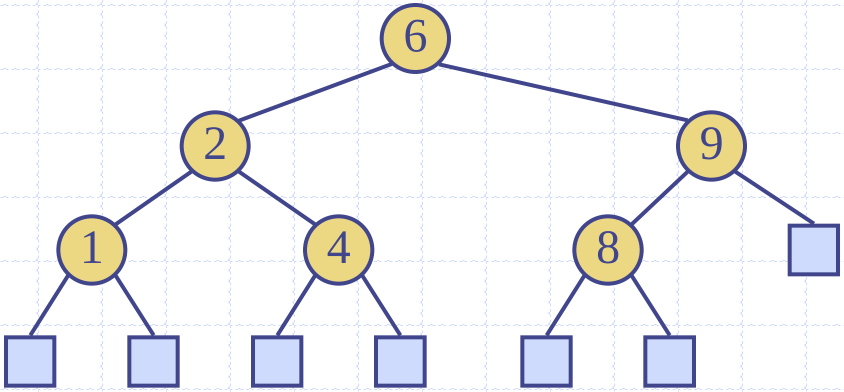
- Binary Search Trees
  - Search
  - Insertion
  - Deletion
- Binary Search

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, 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



Binary Search Tree for Implementing a Map

- The map ADT (get, put, remove)
  - get(k): if the map M has an entry $e=(k,o)$ with key k, return its associated value o; else, return **null**;
  - put(k,o): If M does not have an entry (k, o) then add it to the map M and return **null**; else, replace with o the existing value of the entry with key equal to k and return old value associated with k;
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return **null**;

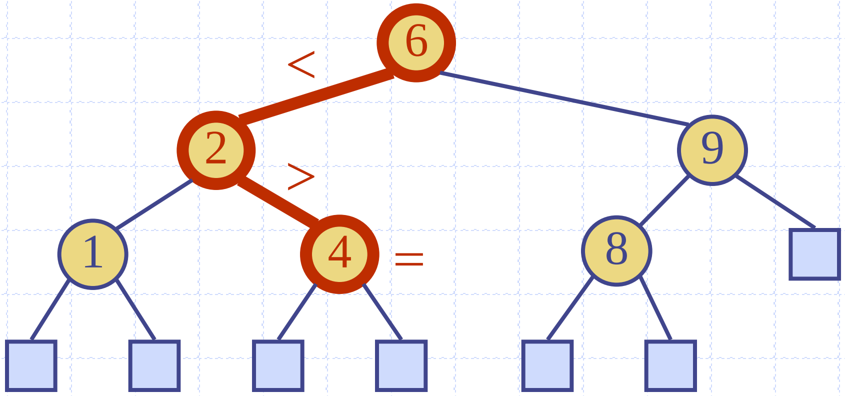
Insert and Remove External Tree Nodes

- For the future reference assume that a binary tree supports the following update operation:
- insertAtExternal(w,(k,o)) – insert the element (k,o) at the external node w, and expand w to be internal, having new (empty) external node children
- removeExternal(w) – remove an external node w and its parent, replacing w's parent with w's sibling

Search

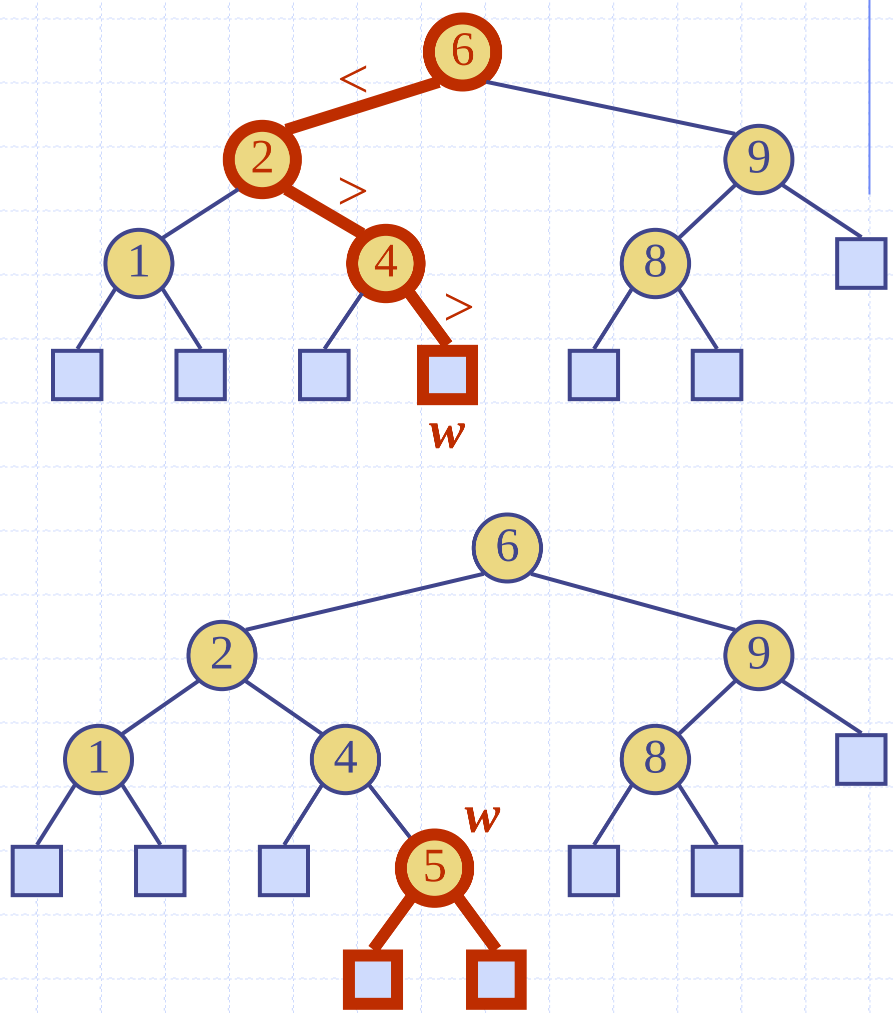
- To perform operation get(k) in a map M i.e. to search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: TreeSearch(4,root)

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



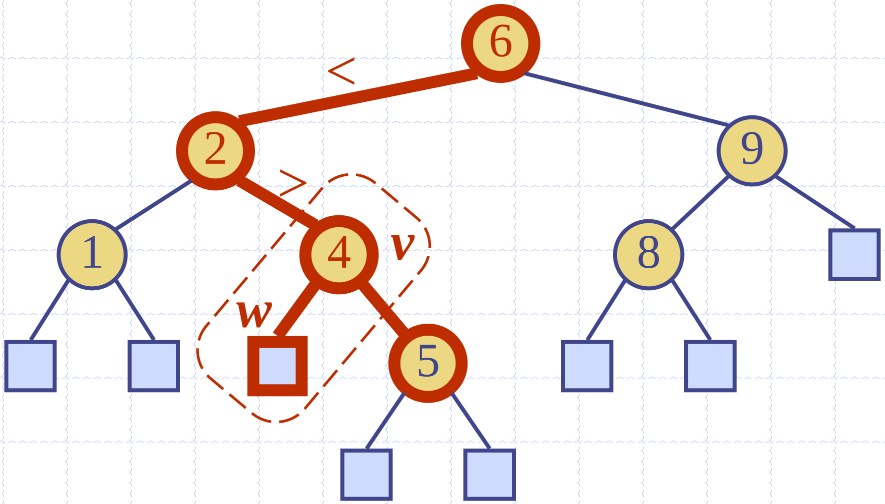
Insertion

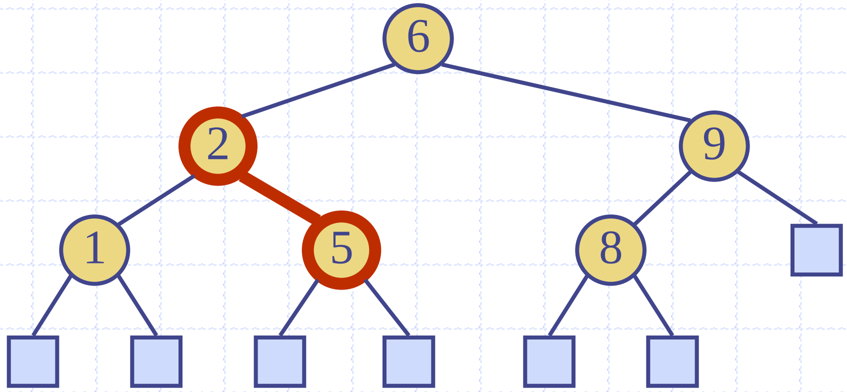
- To perform operation put(k,o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node using insertAtExternal(w,(k,o))
- Example: insert 5



Deletion

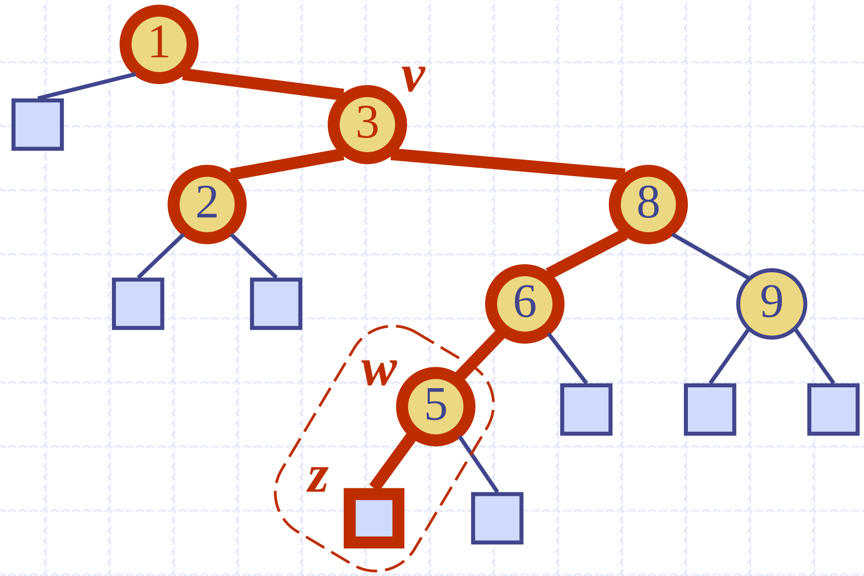
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4

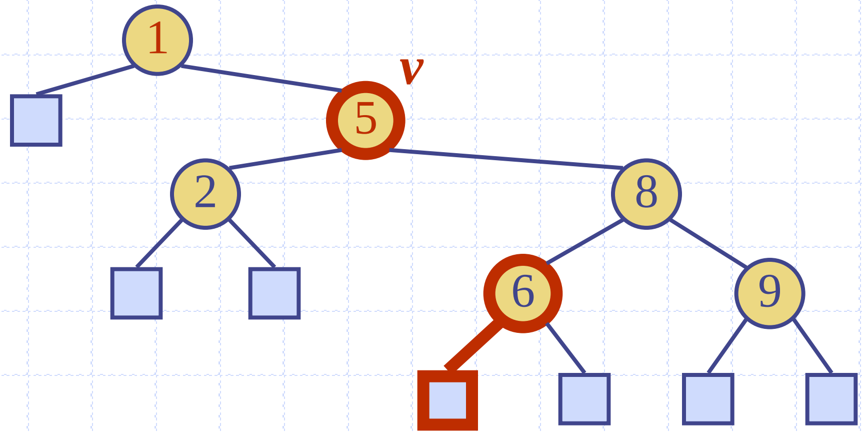




Deletion (cont.)

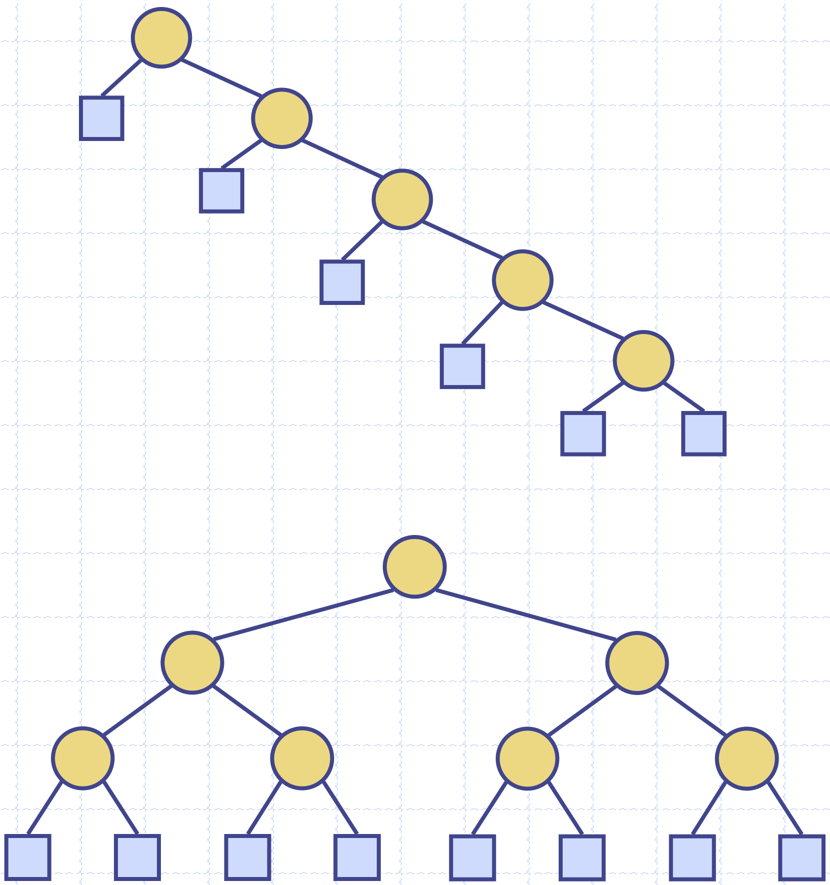
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3





Performance of a Binary Search Tree

- Consider a 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 (assuming we spend $O(1)$ at each node)
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



Exercise 1 – Binary Search Tree

- Insert into an empty binary search tree, entries with keys 50, 32, 19, 40, 62, 28, 69, 55, 65, 100 (in this order). Draw the tree after each insertion
- Describe step by step the execution of operation: TreeSearch(28,root) on tree T
- Describe step by step the execution of operations:
  - remove(28) on tree T,
  - Remove(69) on tree T.

Binary Search

Binary Search

- Assume that we have an ordered array A of size N i.e. $[0..N]$ of integers and we want to find key k
- Search for k=12

```
Algorithm BinarySearch(k, A, N)
    min := 1;
    max := N;
repeat
      mid := (min+max) div 2;
      if k > A[mid] then min := mid + 1;
            else max := mid- 1;
until (A[mid] = k) or (min > max);
```

0	1	2	3	4	5	6	7	8
	12	15	18	21	25	29	37	40

STEP 3:
mid=1
12=12
STOP

STEP 2:
mid=2
12<15
max=2

STEP 1:
mid=4
12<21
max=3

Binary search in a sorted sequence

- **Linear search** for the key **41** in a sequence with elements in arbitrary order:

  46 9 11 27 59 14 17 3 33 63 37 41 52 7 53

- **Binary search** for the key **41** in the sorted sequence:

  3 7 9 11 14 17 27 33 37 41 46 52 53 59 63

  Compare with the middle element:

  3 7 9 11 14 17 27 $\boxed{33}$ 37 41 46 52 53 59 63

  Search recursively in that half (either the lower half or the upper half) which may contain the search key:

  3 7 9 11 14 17 27 $\boxed{33}$ 37 41 46 52 53 59 63

  3 7 9 11 14 17 27 $\boxed{33}$ 37 41 46 $\boxed{52}$ 53 59 63

  3 7 9 11 14 17 27 $\boxed{33}$ 37 $\boxed{41}$ 46 $\boxed{52}$ 53 59 63

- Number of comparisons in the worst case, for a sequence of n elements:

  linear search: n   **binary search**: [log n] + 1 (all log's with base 2).