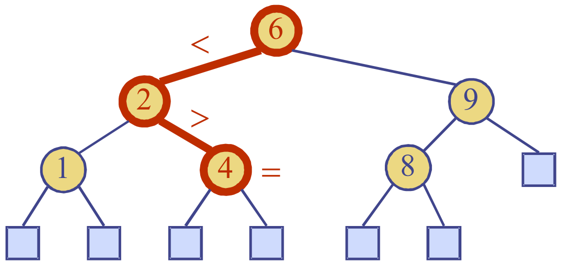
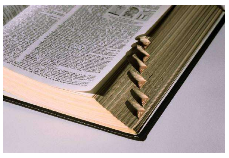
Binary Search Trees

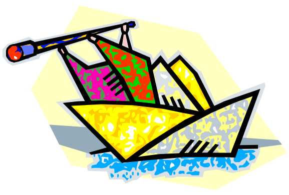


Ordered Maps

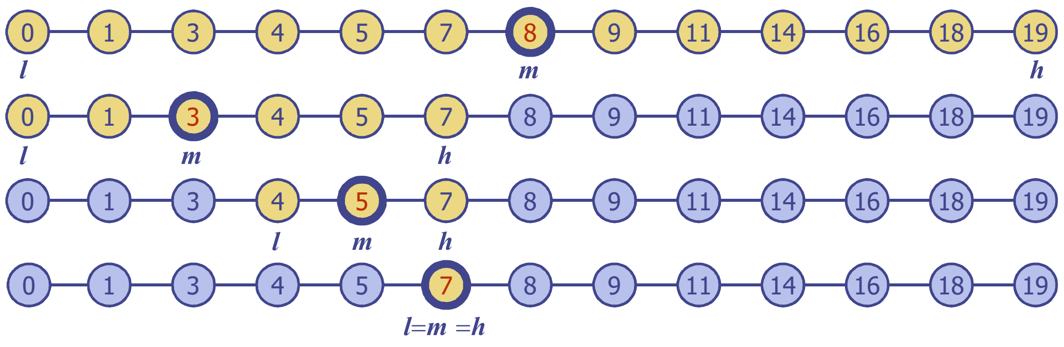


- Keys are assumed to come from a total order.
- New operations:
  - firstEntry(): entry with smallest key value
  - lastEntry(): entry with largest key value
  - floorEntry(k):entry with largest key ≤ k
  - ceilingEntry(k): entry with smallest key ≥ k
  - These operations return null if the map is empty

Binary Search



- Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)

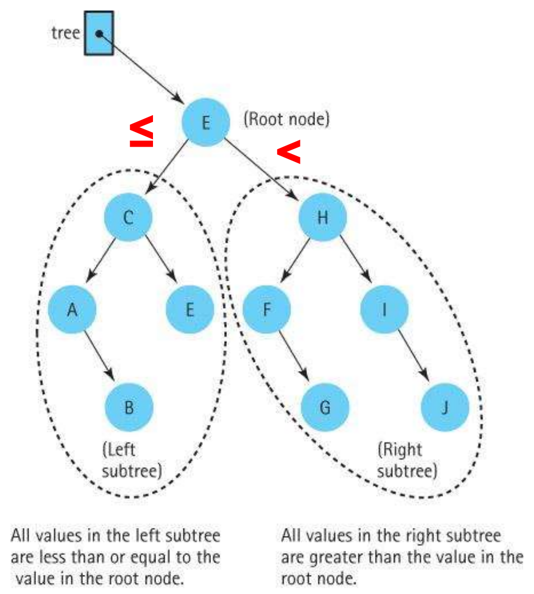


Search Tables

- 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:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since in the worst case we have to shift $n-1$ items to make room for the new item
  - remove take $O(n)$ time since in the worst case we have to shift $n-1$ items to compact the items after the removal
- The lookup 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)

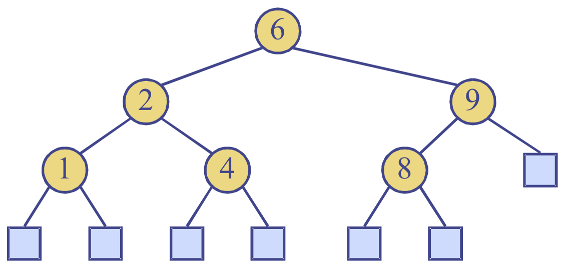
Binary Search Tree

- A binary tree in which the key value in any node is greater than or equal to the key values in its left subtree and less than the key values in its right subtree


tree
E
(Root node)
≤
<
C
A
E
B
(Left subtree)
H
F
I
G
J
(Right subtree)
All values in the left subtree are less than or equal to the value in the root node.
All values in the right subtree are greater than the value in the root node.

Binary Search Trees (cont'd)

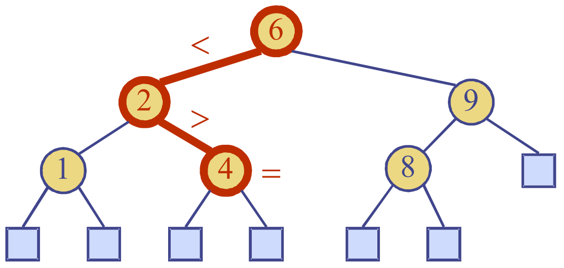
- A binary search tree stores keys (or key-value entries) at its internal nodes
- External nodes do not store items - placeholder
- An inorder traversal of a binary search trees visits the keys in non-decreasing order



Search

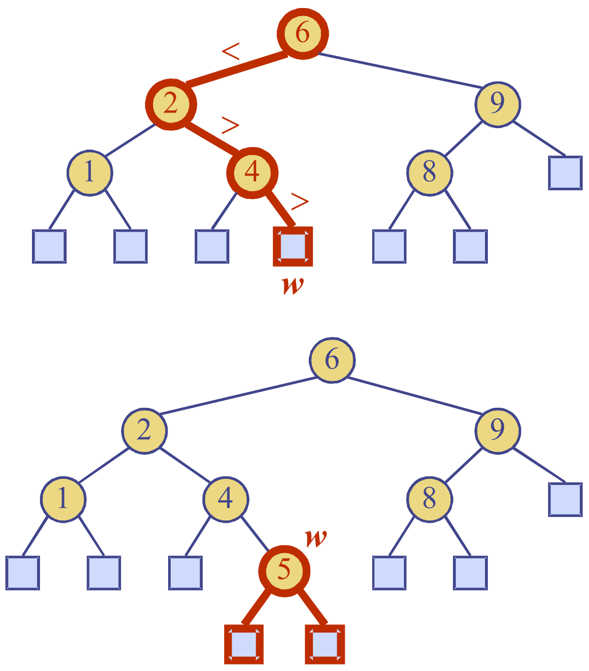
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for floorEntry and ceilingEntry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

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

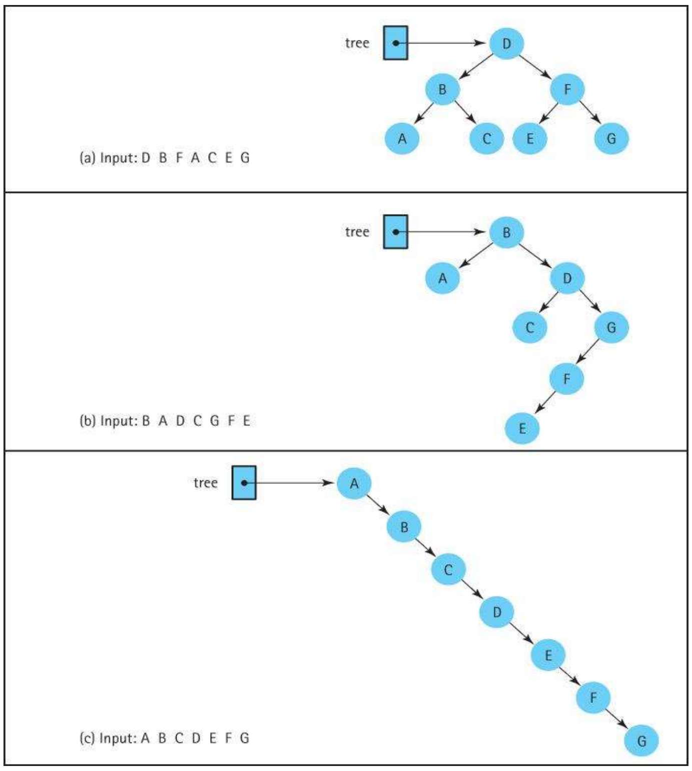


Insertion

- 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
- Example: insert 5



Insertion Order and Tree Shape

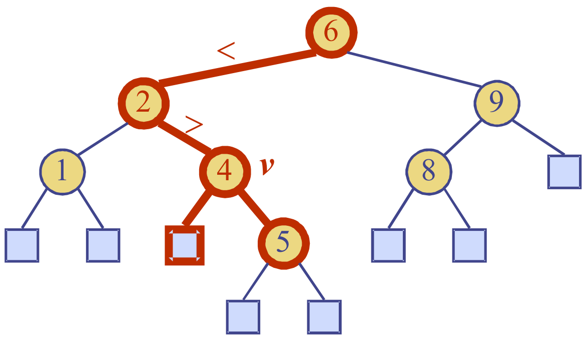


Deletion

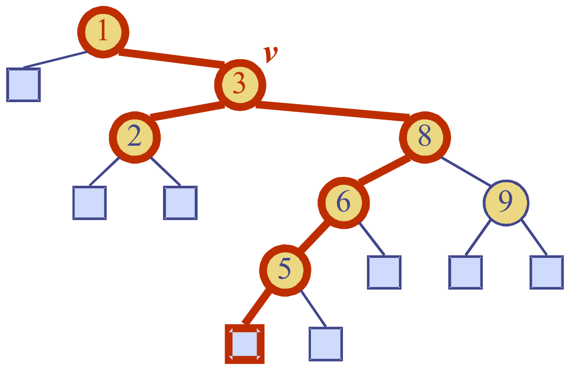
- The most complicated of the binary search tree operations.
- We must ensure when we remove an element we maintain the binary search tree property.

Two cases for the Deletion operation

- Case I: Removing a node with only one child (Removing a node with two external nodes is the same)

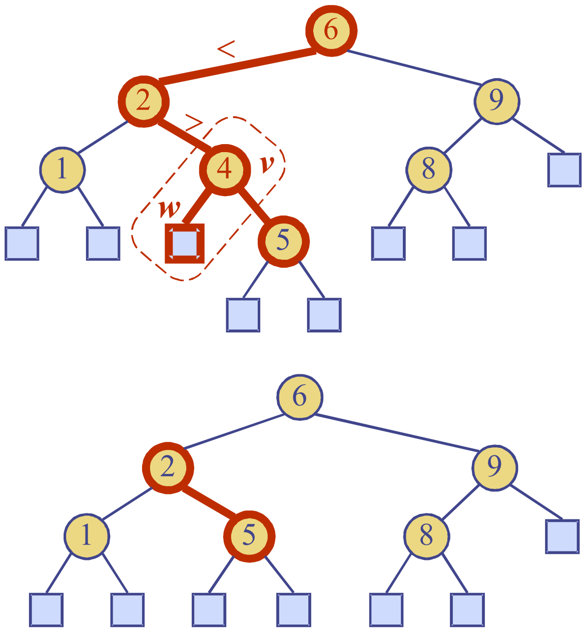


- Case II: Removing a node with two children



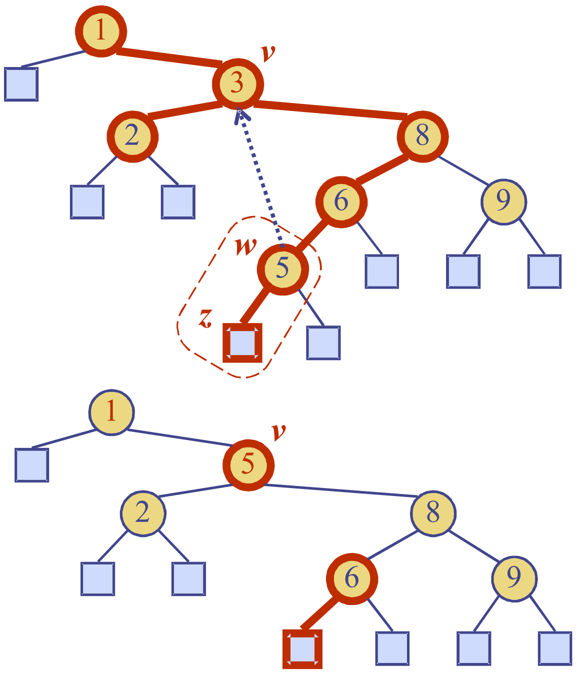
Deletion Case I.

- 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



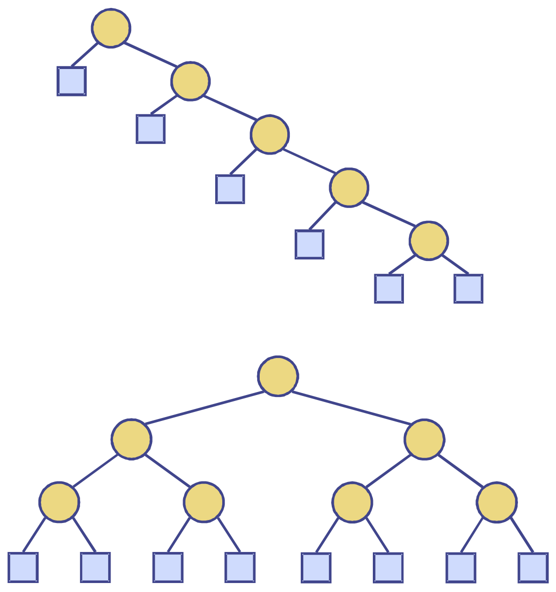
Deletion Case II

- 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



Balancing a Binary Search Tree

- A beneficial addition to our Binary Search Tree ADT operations is a **`balance`** operation
- The specification of the operation is:

```
public void balance();
// Restructures this BST to be optimally balanced
```

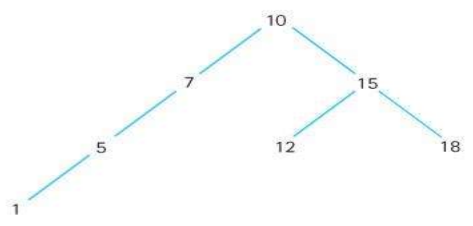
- It is up to the client program to use the **`balance`** method appropriately

Our Approach

- Basic algorithm:
  - Save the tree information in an array
  - Insert the information from the array back into the tree
- The structure of the new tree depends on the order that we save the information into the array, or the order in which we insert the information back into the tree, or both

Using inOrder traversal

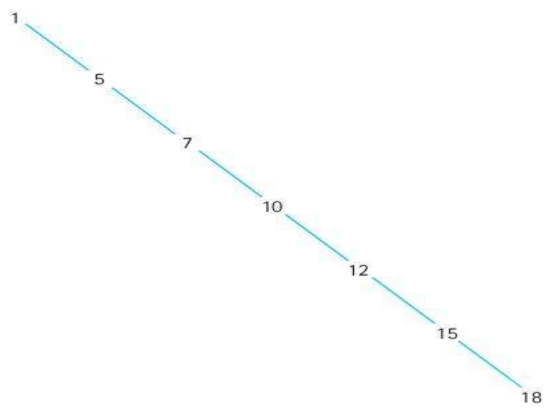
(a) The original tree



(b) The inorder traversal

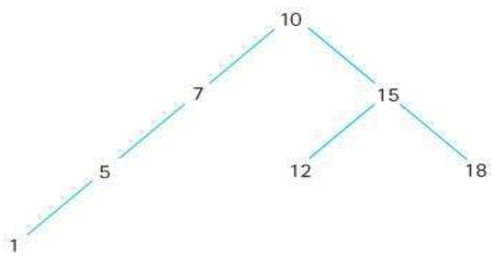
	0	1	2	3	4	5	6
array:	1	5	7	10	12	15	18

(c) The resultant tree if linear traversal of array is used



Using preOrder traversal

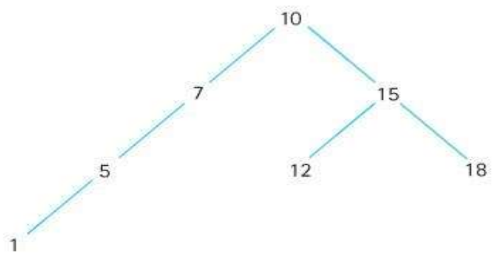
(a) The original tree



(b) The preorder traversal

	0	1	2	3	4	5	6
array:	10	7	5	1	15	12	18

(c) The resultant tree if linear traversal of array is used



To Ensure a Balanced Tree

- Even out as much as possible, the number of descendants in each node’s left and right subtrees
- First insert the “middle” item of the inOrder array
  - Then insert the left half of the array using the same approach
  - Then insert the right half of the array using the same approach

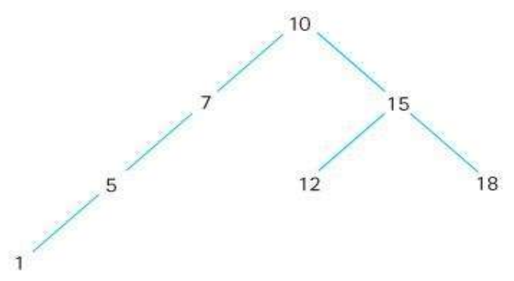
Our Balance Tree Algorithm

```
Balance
For (int index = 0; index < count; index++)
    Set array[index] = tree.getNext(INORDER).
tree = new BinarySearchTree().
tree.InsertTree(0, count - 1)

InsertTree(low, high)
if (low == high)                // Base case 1
    tree.add(nodes[low]).
else if ((low + 1) == high)     // Base case 2
    tree.add(nodes[low]).
    tree.add(nodes[high]).
else
    mid = (low + high) / 2.
    tree.add(mid).
    tree.InsertTree(low, mid – 1).
    tree.InsertTree(mid + 1, high).
```

Using recursive `insertTree`

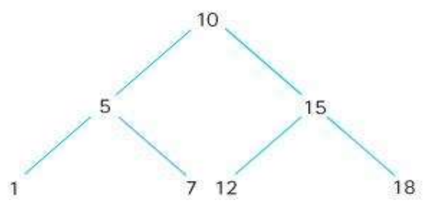
(a) The original tree



(b) The inorder traversal

	0	1	2	3	4	5	6
array:	1	5	7	10	12	15	18

(c) The resultant tree if InsertTree (0,6) is used


10
5
15
1
7
12
18