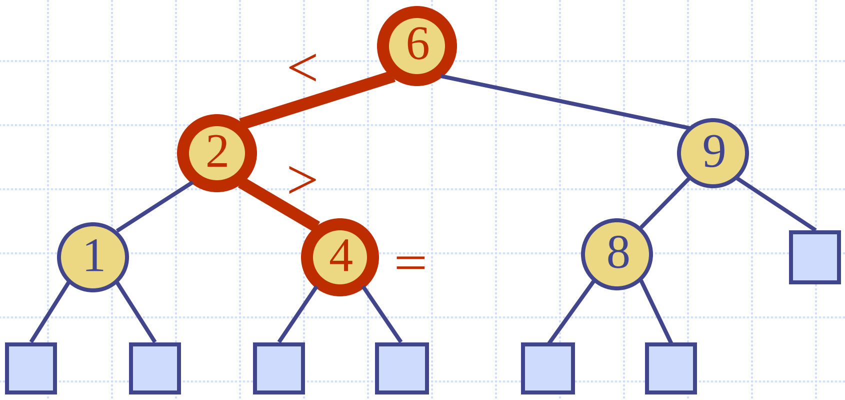
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

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?

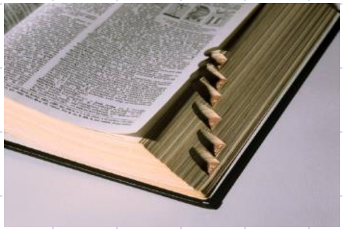
线性查找（Linear Search）对无序数组适用时间复杂度：O(n)
二分查找（Binary Search）要求数组是**有序的**时间复杂度：O(log n)
使用哈希表（Hash Table）如果提前建立映射，查找效率可以达到 O(1)
构建搜索树（如 BST / AVL / 红黑树）插入与查找都能达到 O(log n)（平衡状态下）

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: 最近邻查询
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

找到小于等于某个键 k 的最大键项
找到大于等于某个键 k 的最小键项

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)返回最小的键 k 的项（若不存在则返回 null）
例如：map = {3, 7, 10}, 查询 ceilingEntry(8) 10

- 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).

floorEntry(k)返回最大的键 k 的项（若不存在则返回 null）
例如：map = {3, 7, 10}, 查询 floorEntry(8) 7

The Sorted Map ADT

返回**严格小于**k 的**最大键项**（如果不存在，则返回 null）
例：若 map 为 {3, 7, 10}，查询 lowerEntry(7) 得到 3

- lowerEntry(k): Returns the entry with the greatest key value strictly less than k (or null, if no such entry exists).

返回**严格大于**k 的**最小键项**（如果不存在，则返回 null）
例：若 map 为 {3, 7, 10}，查询 higherEntry(7) 得到 10

- 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.

返回一个迭代器，遍历所有键值 **k 且 < k** 的项即：[k, k)
例：map 为 {2, 4, 6, 8, 10}，查询 subMap(4, 9) 得到 {4, 6, 8}

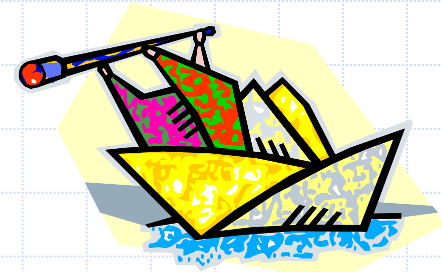
我们将映射的条目（entries）存储在数组列表 A 中，按照键（key）递增顺序排列。
这种实现方式被称为“ 有序查找表 ”（sorted search table）。

Sorted Search Tables

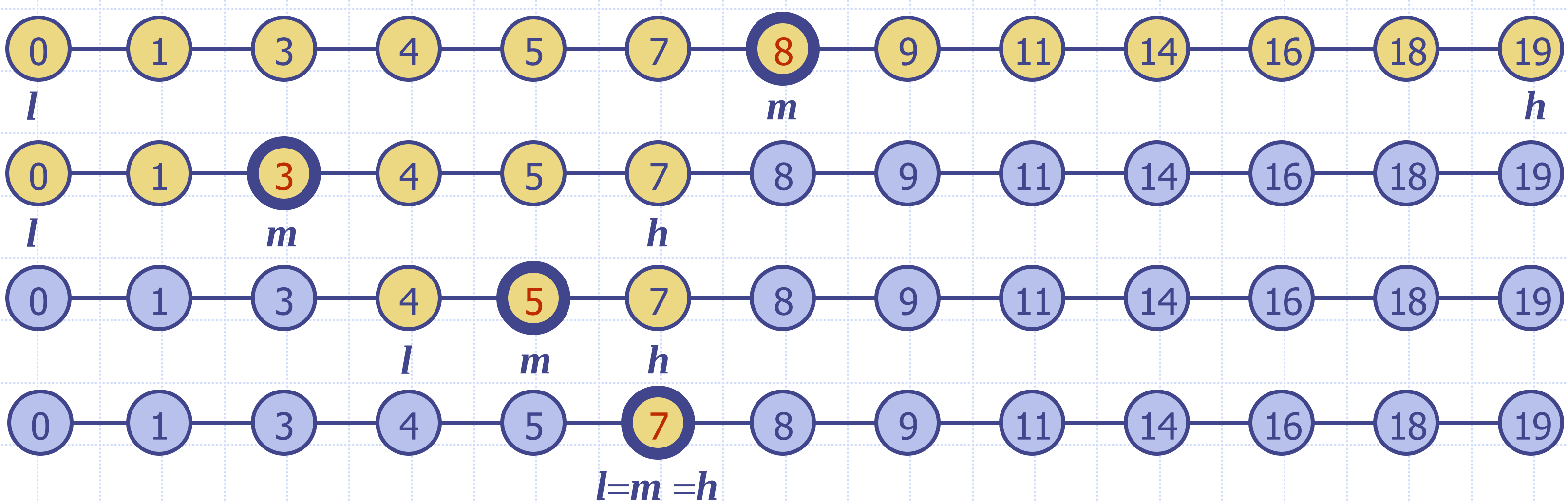
- We store the map’s entries in an array list *A* so that they are in increasing order of their keys.
- We refer to this implementation as a ***sorted search table***.

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
  $key(u) < key(v) < key(w)$

- 对于任意节点 v：
  - 所有在其**左子树**的节点 u，满足：

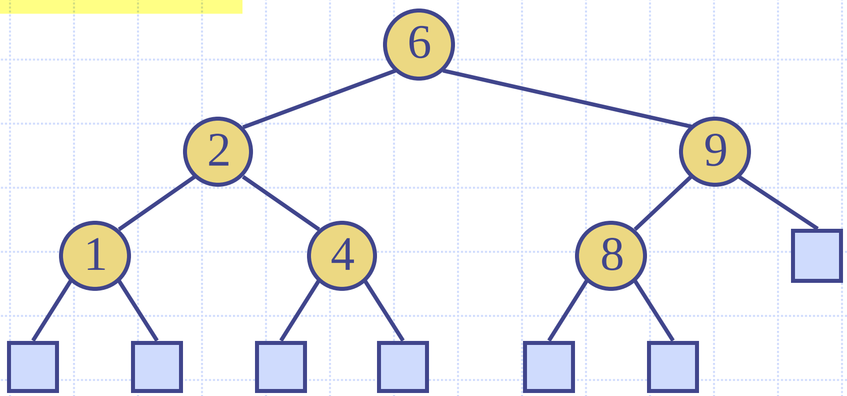
$\mathrm{key}(u) \leq \mathrm{key}(v)$

  - 所有在其**右子树**的节点 w，满足：

$\mathrm{key}(v) \leq \mathrm{key}(w)$

- 如果不允许重复键值（常见设定），则满足严格不等：

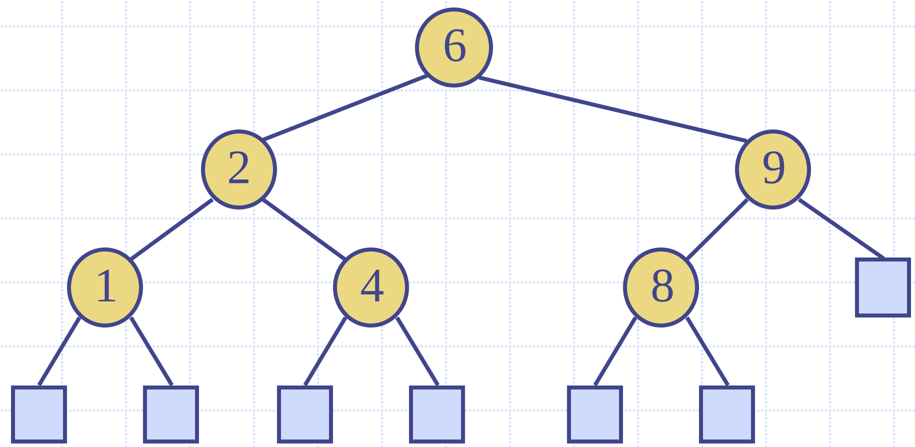
$\mathrm{key}(u) < \mathrm{key}(v) < \mathrm{key}(w)$



Binary Search Trees

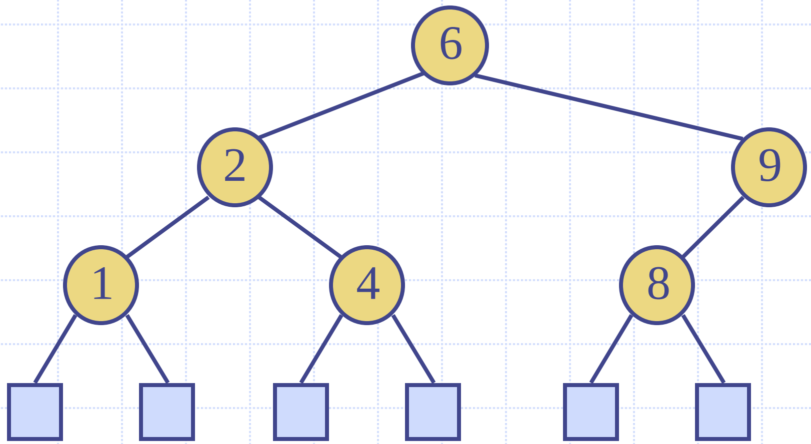
External nodes do not store items

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



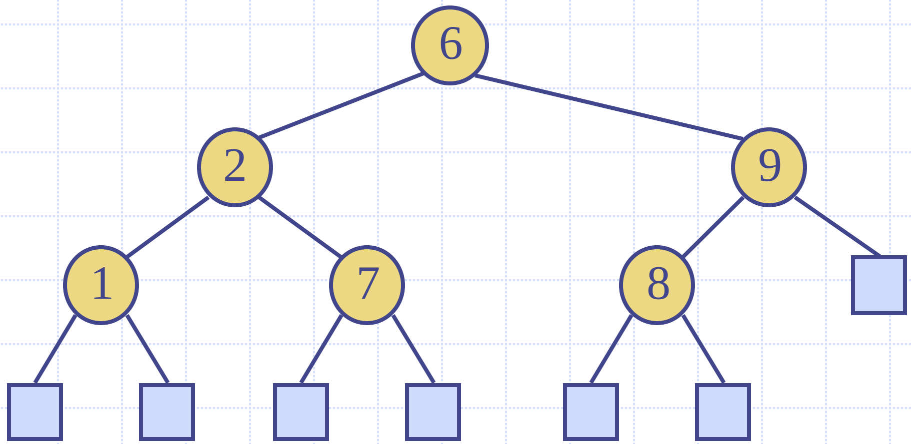
Is this a binary search tree?

一个二叉搜索树需要满足：
对于任意节点 v：
所有左子树节点的值 < v
所有右子树节点的值 > v



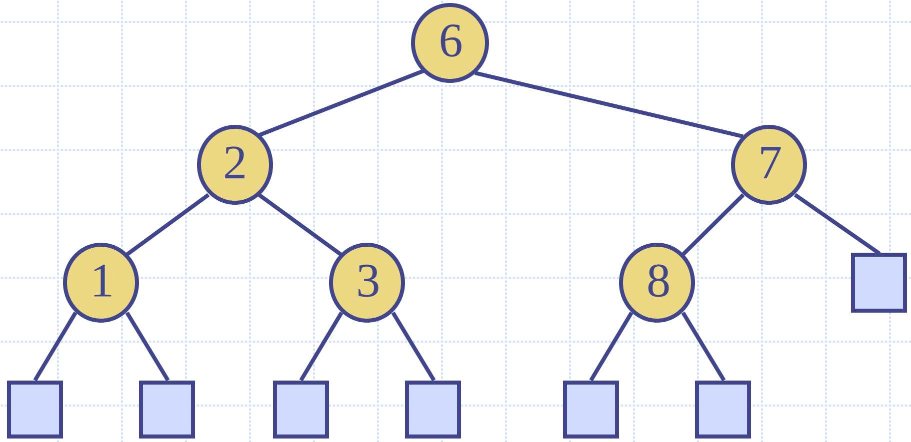
Is this a binary search tree?

No



Is this a binary search tree?

No



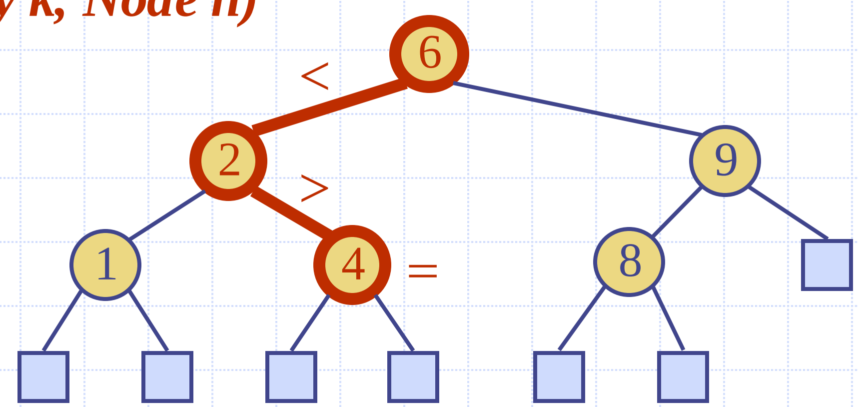
从根节点开始，沿着树向下查找。
每一步：
如果 k < 当前节点值 去左子树
如果 k > 当前节点值 去右子树如果 k == 当前节点值 找到目标
如果走到底仍未找到（即遇到 null 节点），则目标不存在。

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.

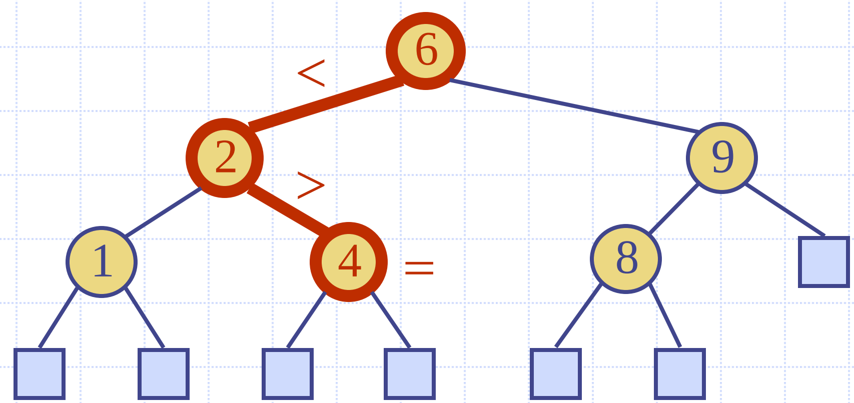
Algorithm ***Node TreeSearch(Key k, Node n)***



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***())

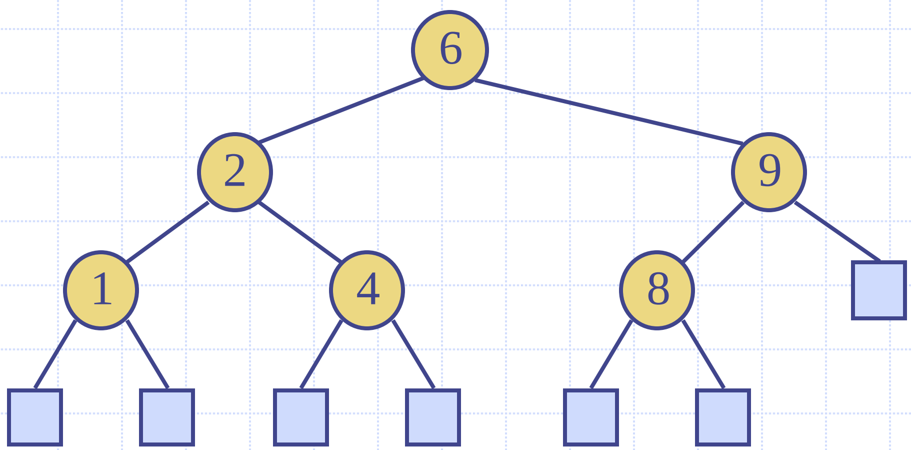
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar



Fundamental Property of Binary Search Trees

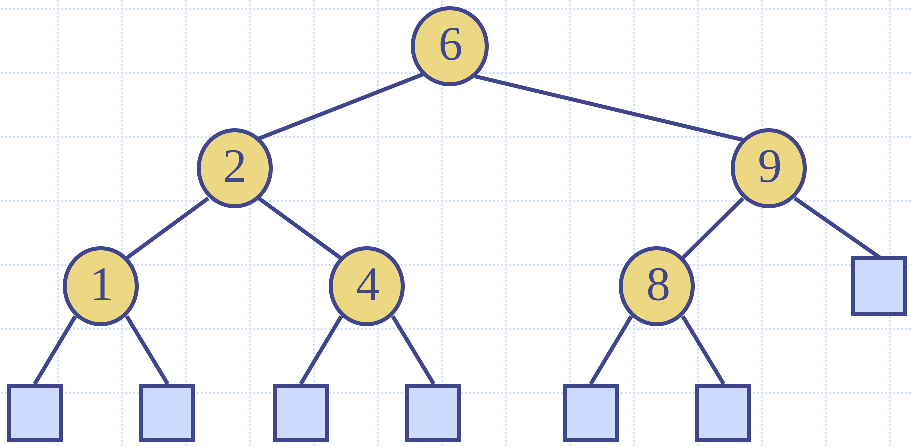
中序遍历的顺序是：**左子树　当前节点　右子树**
对每个节点递归执行此顺序。1，2，4，6，8，9

- 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.

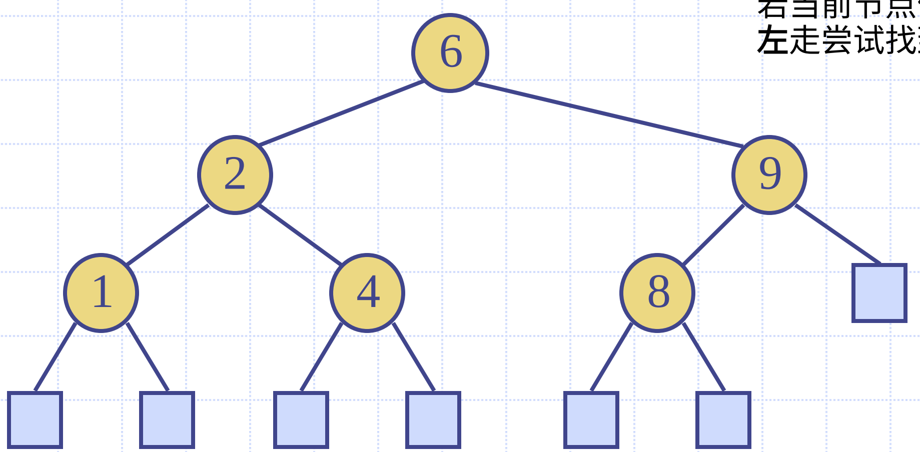


Fundamental Property of Binary Search Trees

- How to access the minimal key? 从根开始，不断往**左子节点**走，直到没有左子节点。
- How to access the maximal key? 从根开始，不断往**右子节点**走，直到没有右子节点。
- How to access the largest key less than or equal to k?
- How to access the smallest key greater than or equal to k?

3.从根向下搜索：若当前节点值 > k，往**左**走；
若当前节点值 k，当前节点可能是候选解，还需去**右子树**看看有没有更大但仍 k 的值。

4.从根向下搜索：若当前节点值 < k，往**右**走
；
若当前节点值 k，当前可能是候选解，继续向**左**走尝试找到更小的。



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 $get(k)$

从根节点开始；
如果 k < 当前节点 去左子树；
如果 k > 当前节点 去右子树；
如果遇到空位置（即 None），就在那里插入新的节点；
如果遇到相同的 key 选择更新值（可选逻辑）。

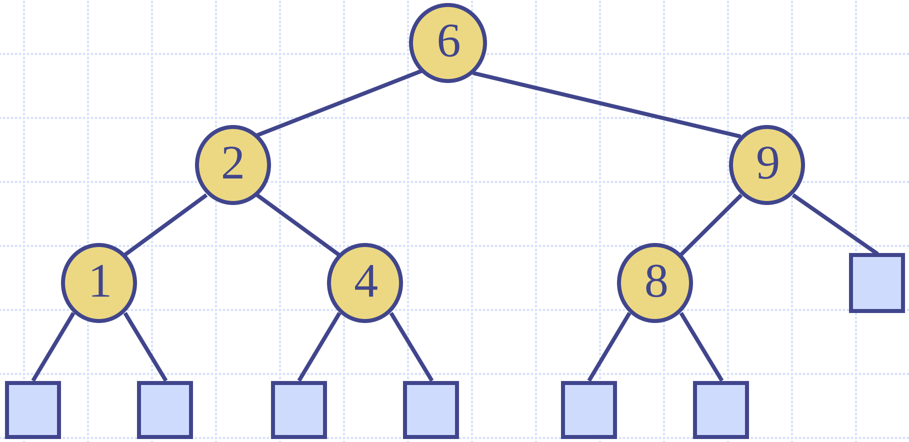
Insertion

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

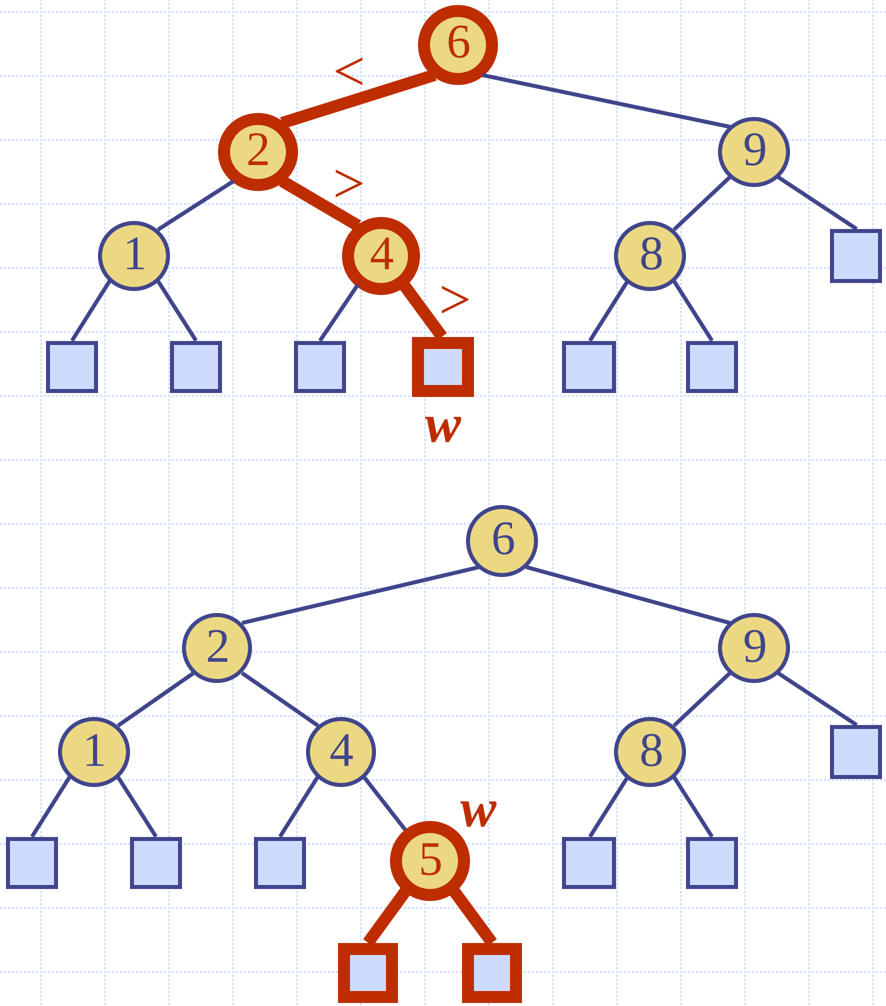
- We search for key k (using TreeSearch)
- Assume k is already in the tree then just replace the value.
- Otherwise, let w be the leaf reached by the search, we insert k at node w and expand w into an internal node

Insertion

- Example: insert 5

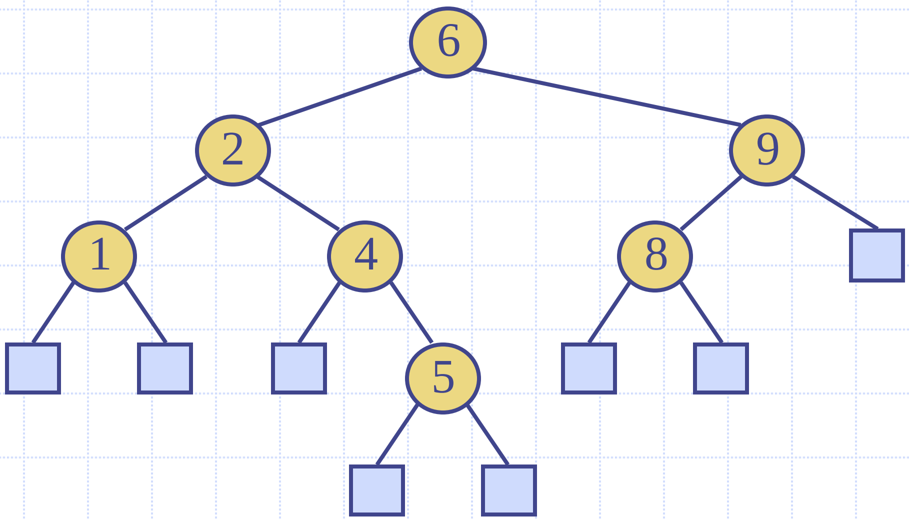


Insertion



Deletion

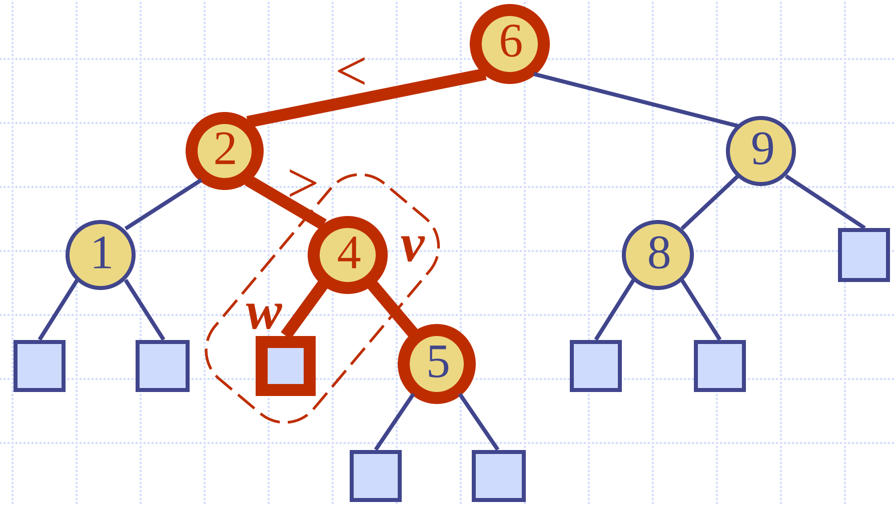
- How can we perform the operation $remove(k)$?
  - E.g., remove 7, 5, 4, or 2

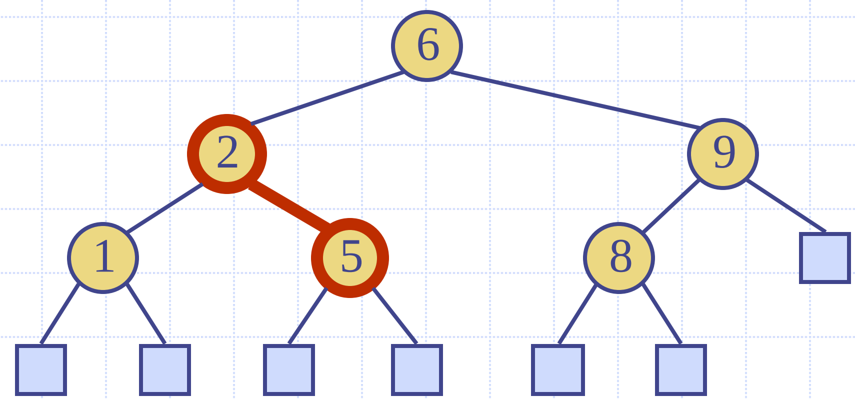


上图（中间）：
4 是 2 的右子节点，5 是 4 的右子；
4 的左子节点为空（外部节点）；
所以可以将 4 和它的左“空子节点”一起删除；
然后将 5 提上，直接连接到 2 的右子上。

Deletion

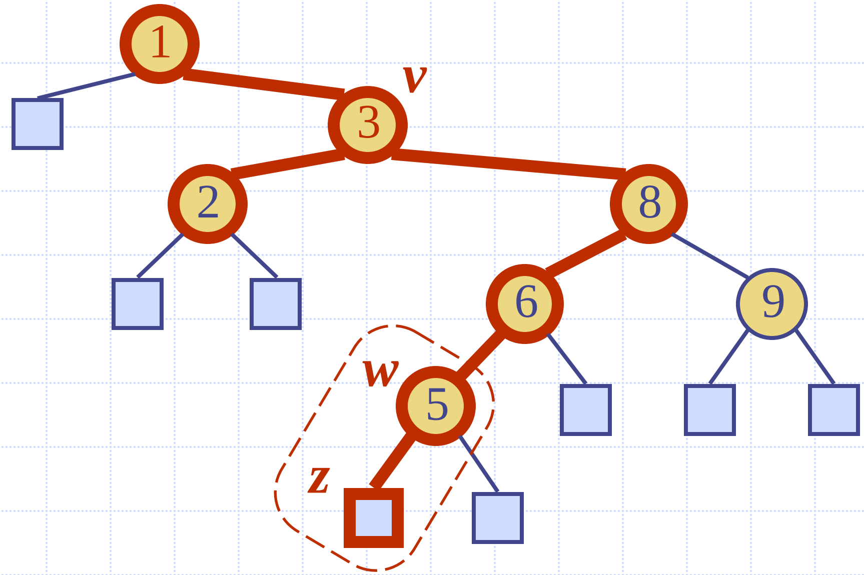
- To perform operation $remove(k)$, we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation $removeExternal(w)$, which removes w and its parent
- Example: remove 4

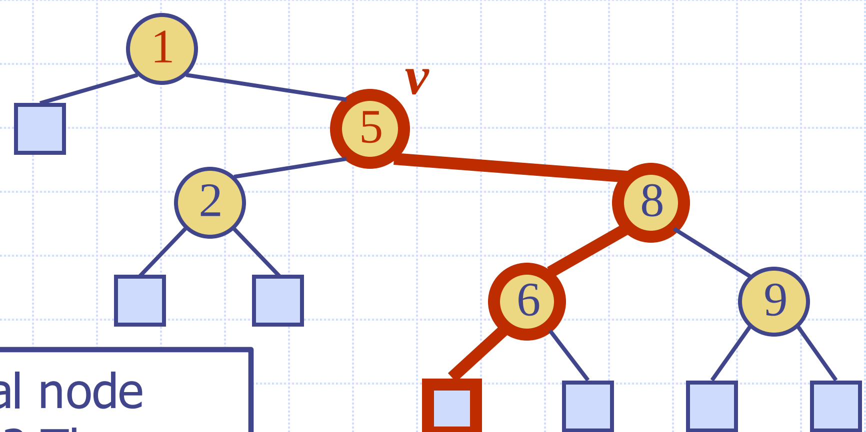




Deletion (cont.)

- 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





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