Recall how we progressed to the Heap.

a) We were interested in the priority queue (PQ) data structure with the following two operations: (i) finding the "minimum" key, (ii) removing the "minimum" key and (iii) inserting a new element into the PQ

b) We started with unsorted list implementation of a PQ and then a sorted implementation. (i) unsorted took O(n) to find/remove min and O(1) to insert (ii) sorted took O(1) to find/remove min and O(n) to insert.

c) And then we realised we could do better by interpreting the list as a specific kind of tree called heap (with two properties).

Similarly, let us say

a) We are interested in a data structure with the following three operations: (i) searching for an element, (ii) removing an element and (iii) inserting a new element into the data structure.

b) We have already seen unsorted and sorted implementations for such a data structure. (i) unsorted took O(n) to search and O(1) to insert (ii) sorted took O(log n) to search and O(n) to insert

c) Can you now think of a tree based implementation of such a data structure, which gives you O(log n) kind of search and better insertion time?

Recall how we progressed to the Heap.

a) We were interested in the priority queue (PQ) data structure with the following two operations: (i) finding the "minimum" key, (ii) removing the "minimum" key and (iii) inserting a new element into the PQ

THINK

b) We started with the unsorted list implementation of a PQ and the sorted implementation. (i) unsorted took O(n) to find/remove min and O(1) to insert (ii) sorted took O(1) to find/...

Find tree based implementation for search (inspired by binary search) s.t insertion takes less time

heap (with two properties)

Similarly, let us say

a) We are interested in a data structure with the following three operations: (i) searching for an element, (ii) removing an element and (iii) inserting a new element into the data structure.

b) We have already seen unsorted and sorted implementations for such a data structure. (i) unsorted took O(n) to search and O(1) to insert (ii) sorted took O(log n) ✓ to search and O(n) ✗ to insert

c) Can you now think of a tree based implementation of such a data structure, which gives you O(log n) kind of search and better insertion time?

Recall how we progressed to the Heap.

a) We were interested in the priority queue (PQ) data structure with the following two operations: (i) finding the "minimum" key, (ii) removing the "minimum" key and (iii) inserting a new element into the PQ

PAIR

b) We started with unsorted list implementation of a PQ and then a sorted implementation. (i) unsorted took O(n) to find/remove min and O(1) to insert (ii) sorted took O(1) to find/remove min and O(n) to insert.

c) And then we realised we could do better by interpreting the list as a specific kind of tree called heap (with two properties).

Similarly, let us say

a) We are interested in a data structure with the following three operations: (i) searching for an element, (ii) removing an element and (iii) inserting a new element into the data structure.

b) We have already seen unsorted and sorted implementations for such a data structure. (i) unsorted took O(n) to search and O(1) to insert (ii) sorted took O(log n) to search and O(n) to insert

c) Can you now think of a tree based implementation of such a data structure, which gives you O(log n) kind of search and better insertion time?

Recall how we progressed to the Heap.

a) We were interested in the priority queue (PQ) data structure with the following two operations: (i) finding the "minimum" key, (ii) removing the "minimum" key and (iii) inserting a new element into the PQ.

b) We started with unsorted list implementation of a PQ and then a sorted implementation. (i) unsorted took O(n) to find/remove min and O(1) to insert (ii) sorted took O(1) to find/remove min and O(n) to insert.

c) And then we realised we could do better by interpreting the list as a specific kind of tree called heap (with two properties).

SHARE

Similarly, let us say

a) We are interested in a data structure with the following three operations: (i) searching for an element, (ii) removing an element and (iii) inserting a new element into the data structure.

b) We have already seen unsorted and sorted implementations for such a data structure. (i) unsorted took O(n) to search and O(1) to insert (ii) sorted took O(log n) to search and O(n) to insert

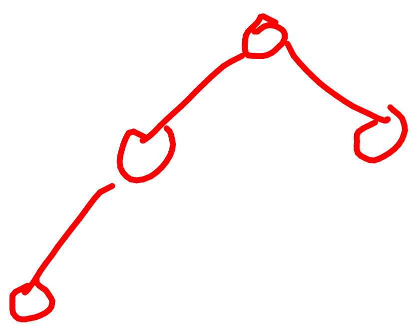
c) Can you now think of a tree based implementation of such a data structure, which gives you O(log n) kind of search and better insertion time?

Desirable properties of a binary search tree:

① Why not heap property?

- Will not help in "searching"

② Balanced tree: At every node, height of left subtree = height of right subtree



} Not possible with 4 elements!

Modification: Height of 2 subtrees differ by atmost 1

[of course, heap had this property. Hence possible]

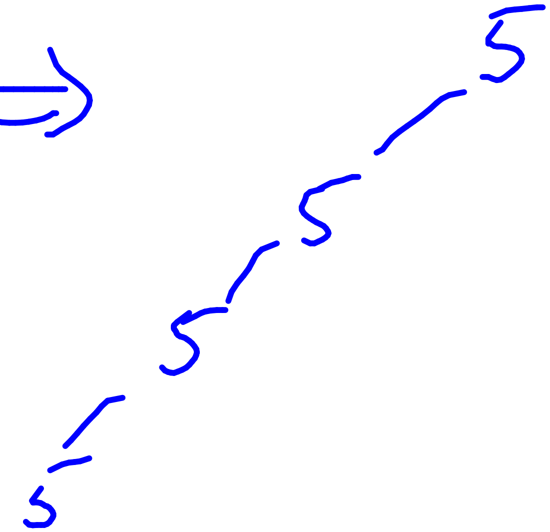
③ Key(node.leftdescendents) $\leq$ Key(node) $<$ key(node.rightdescendents).

for each node

strict inequality to avoid ambiguity

Q: Does ③ conflict with ②?

5 5 5 5 $\Rightarrow$



Modification:

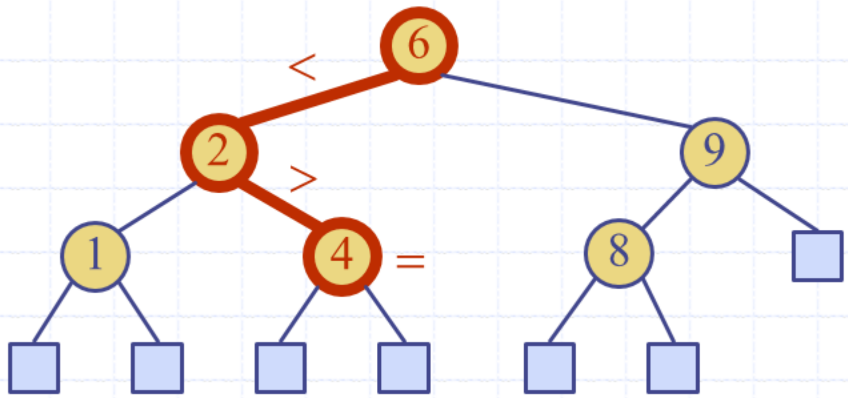
key(node.leftdescendents) < key(node) < key(node.rightdescendents)

Incase of duplicate keys, you can use a linkedlist at each node or use "$\leq$" & search both branches!

3 alone is binary search tree

2 & 3 together $\Rightarrow$ AVL tree

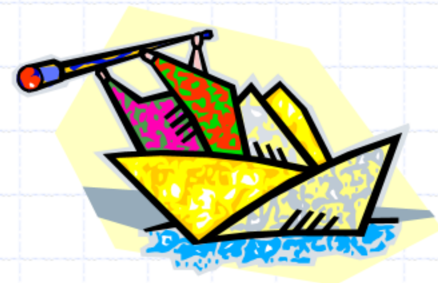
Binary Search Trees



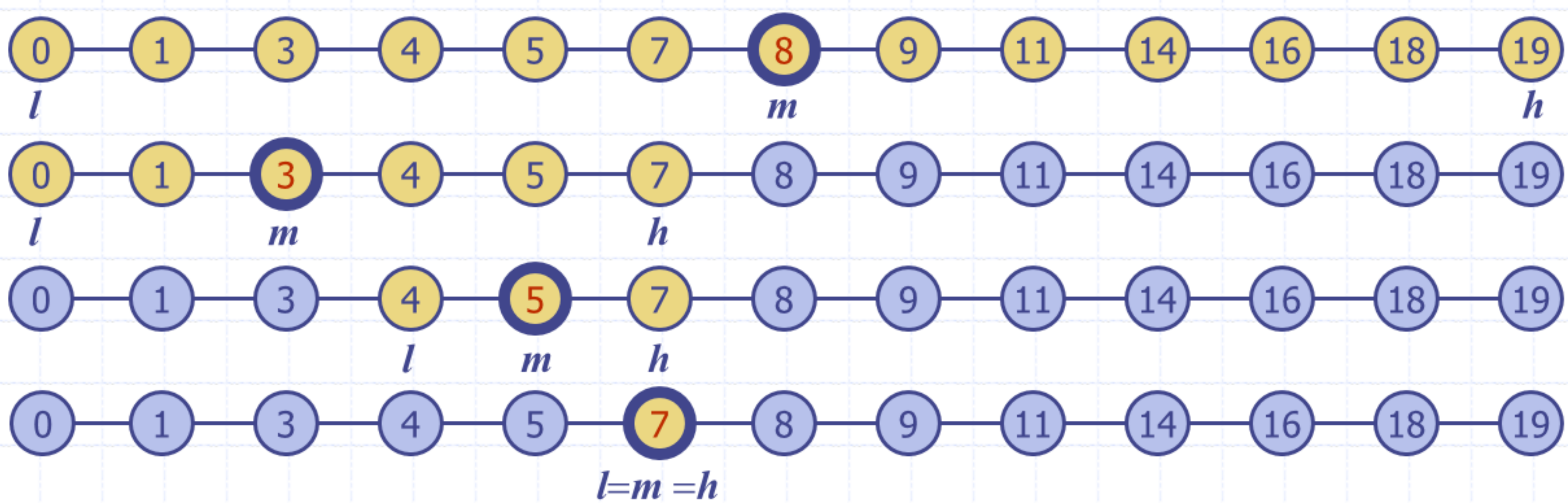
Ordered Dictionaries

- Keys are assumed to come from a total order.
- New operations:
  - first(): first entry in the dictionary ordering
  - last(): last entry in the dictionary ordering
  - successors(k): iterator of entries with keys greater than or equal to k; increasing order
  - predecessors(k): iterator of entries with keys less than or equal to k; decreasing order

Binary Search (§ 8.3.3)



- Binary search can perform operation find(k) on a dictionary 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 : O(n) if you need all entries of k
- Example: find(7)



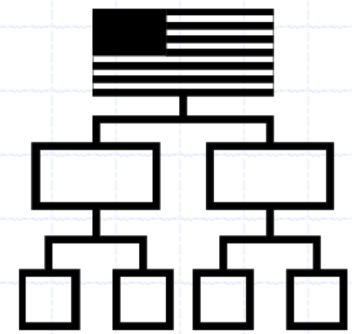
Note: O(log n) assumes that even if key "k" is repeated, you find only One Instance

Search Tables

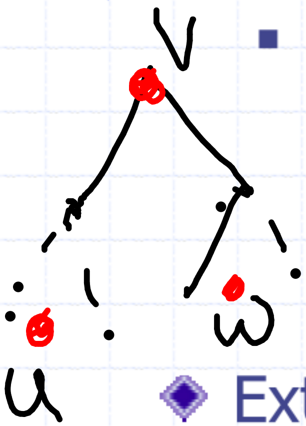
- A search table is a dictionary implemented by means of a sorted sequence
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - find takes $O(\log n)$ time, using binary search → assuming only one instance of k returned
  - insert takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal

  we hope to reduce these 2 using tree
- 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 Trees (§ 9.1)

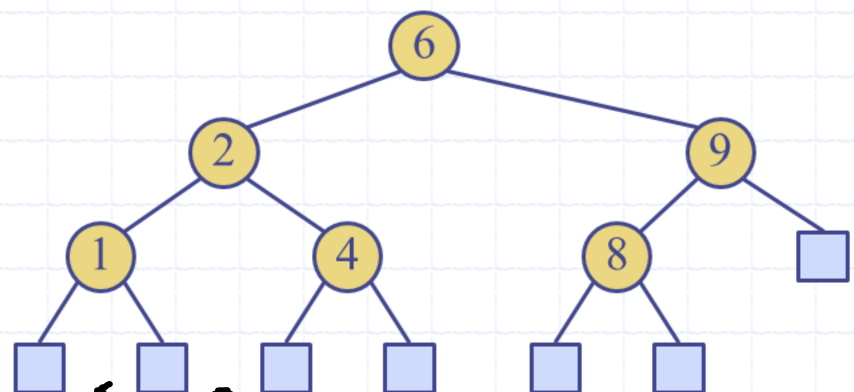


- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes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



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

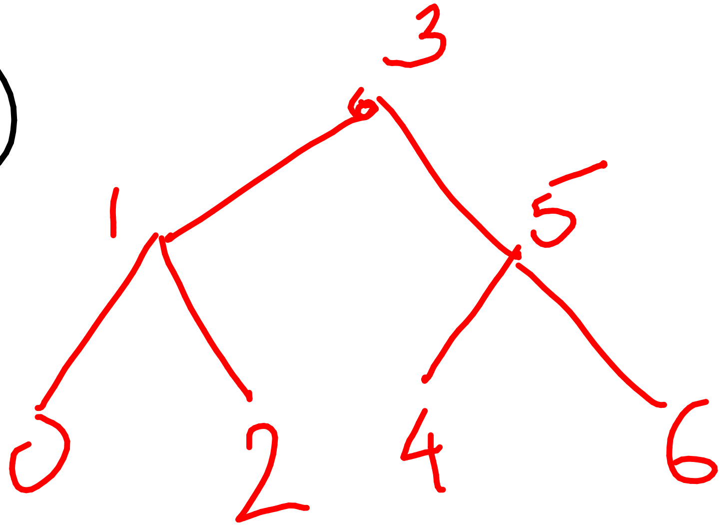
O(n) traversal time



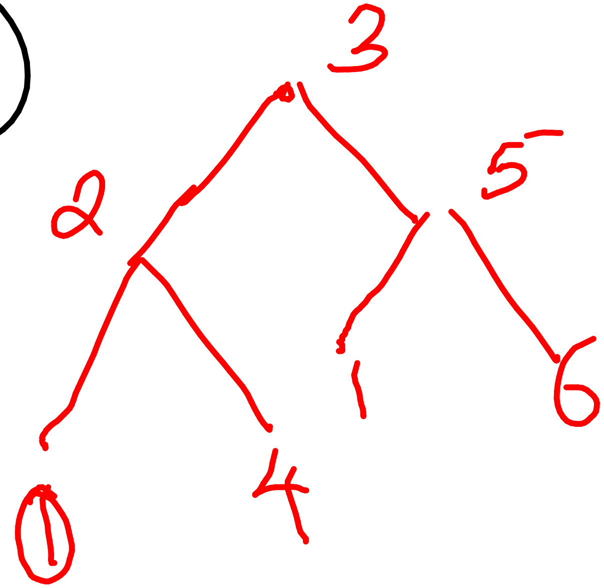
(1) u & w are arbitrary left & right descendent (see next side)

(2) if keys were allowed to repeat, $key(u) \leq key(v) \leq key(w)$ Linked list at each node (like Hashmaps)

1



2



Tighter constraint!

a: key(n.leftdesc) < key(n) < key(n.rightdesc)

Only using

b: key(n.left) < key(n) < key(n.right)

1 satisfies a & b

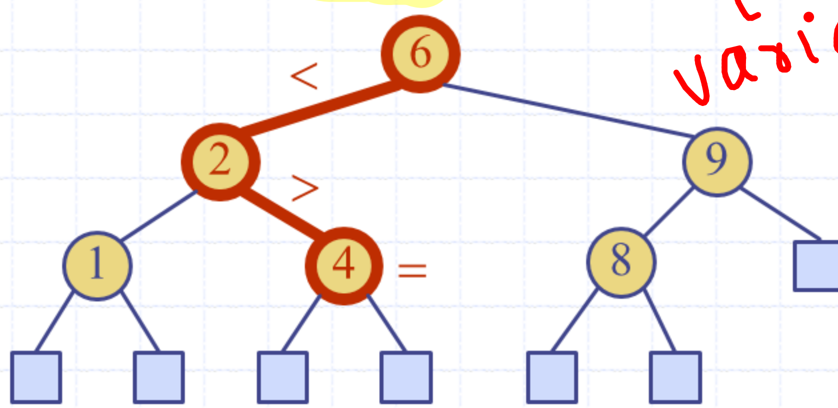
2 satisfies a

Search (§ 9.1.1)

- 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 nukk
- Example: find(4):
  - Call TreeSearch(4,root)

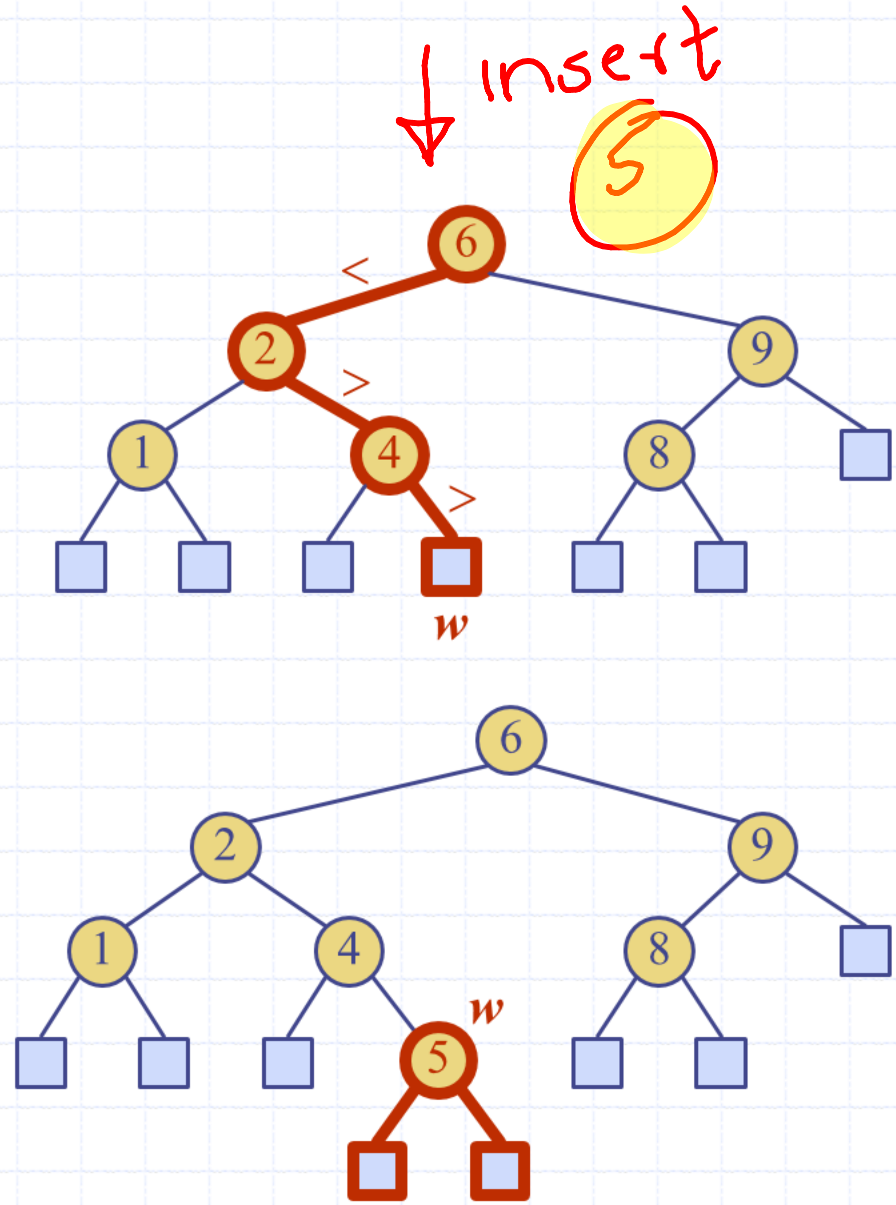
Algorithm *TreeSearch*(k, v)
    if *T.isExternal* (v)
        return v
    if $k < key(v)$
        return *TreeSearch*(k, *T.left*(v))
    else if $k = key(v)$
        return v
    else $\{ k > key(v) \}$
        return *TreeSearch*(k, *T.right*(v))

Recursion
(Iterative variant is simple)



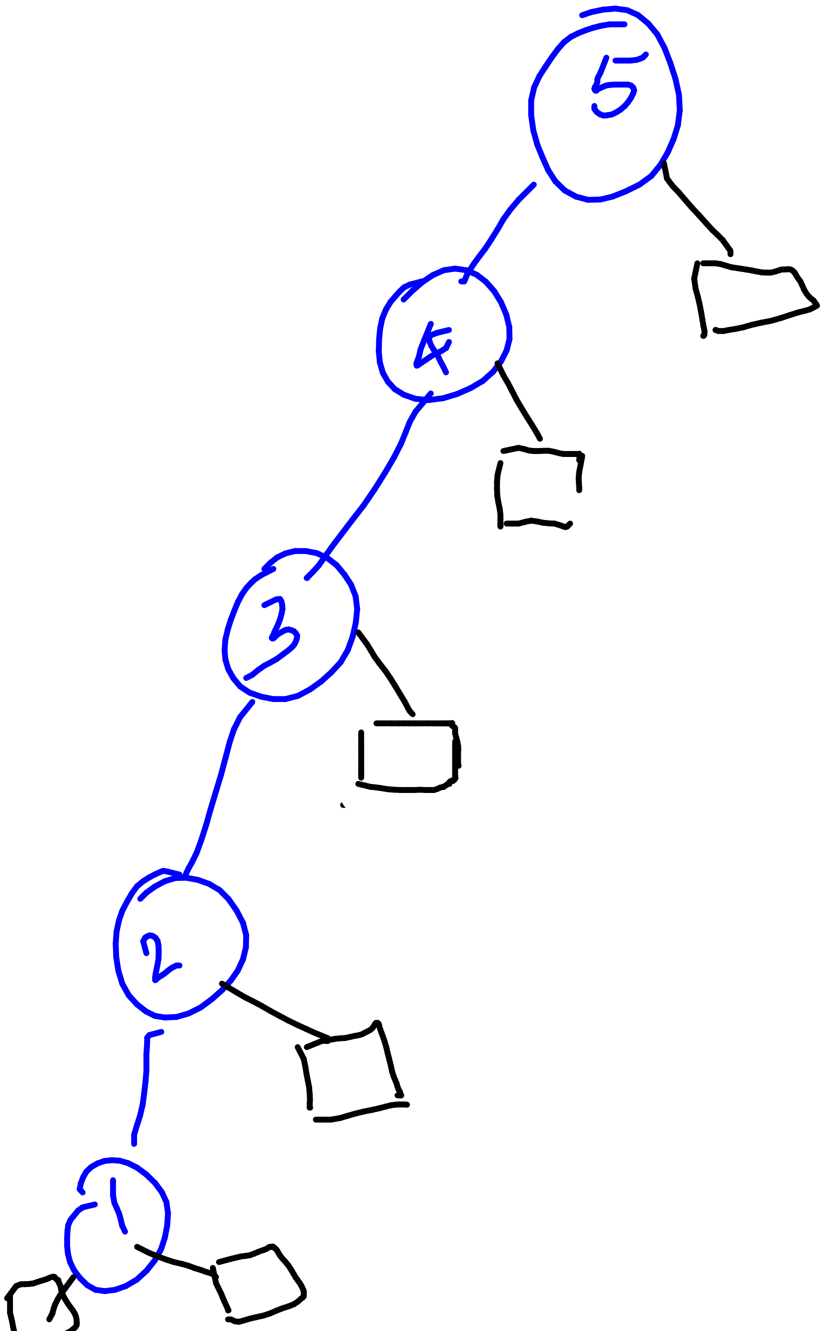
Insertion

- To perform operation inser(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let 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



5 4 3 2 1

Note: $h \le n$



Highly skewed

height = 5

n = 5

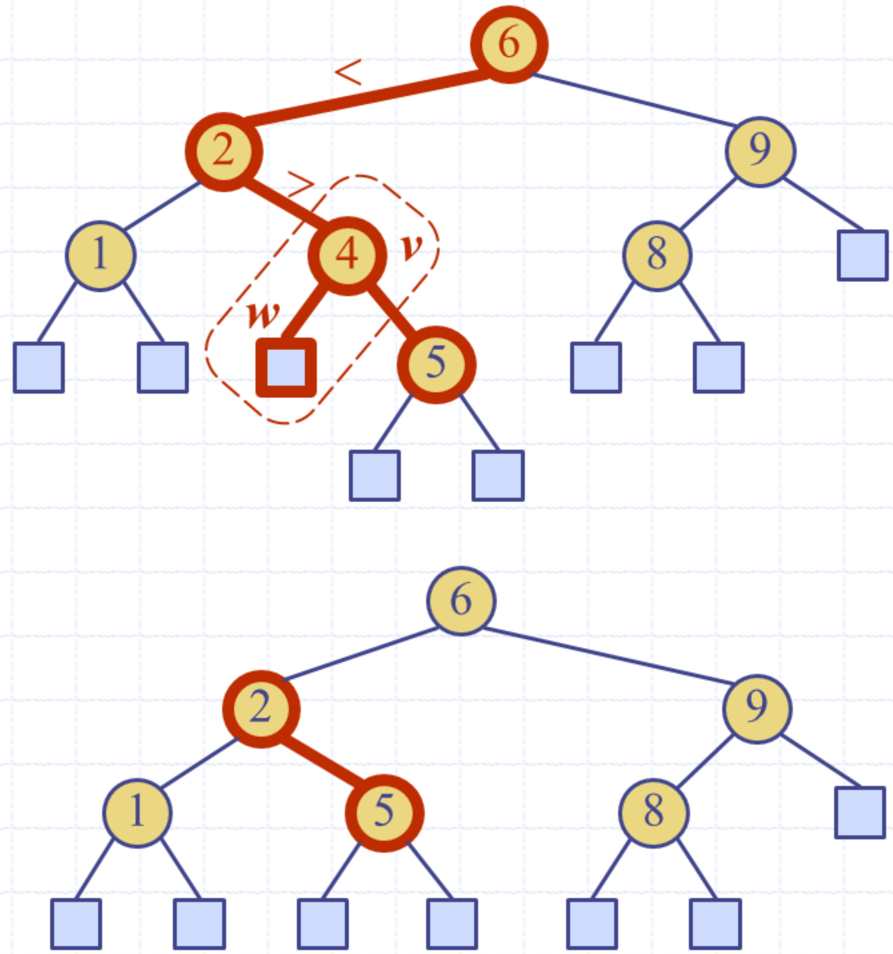
$\boxed{h = n \text{ in worst case!}}$

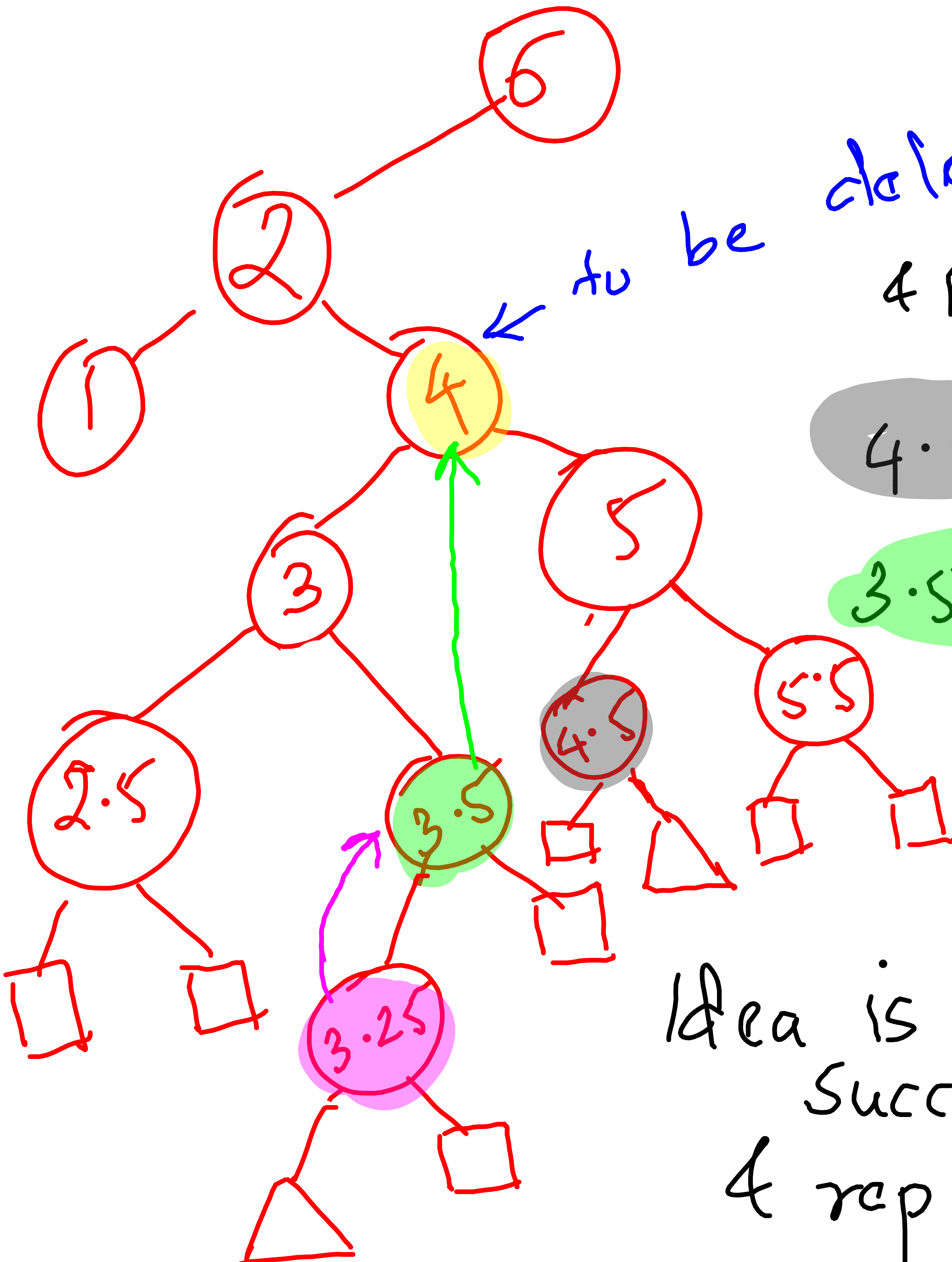
Q: Can I put a lower bnd on h?

Ans: $n \leq 2^h - 1 \Rightarrow h \geq \log_2(n+1)$

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





to be deleted

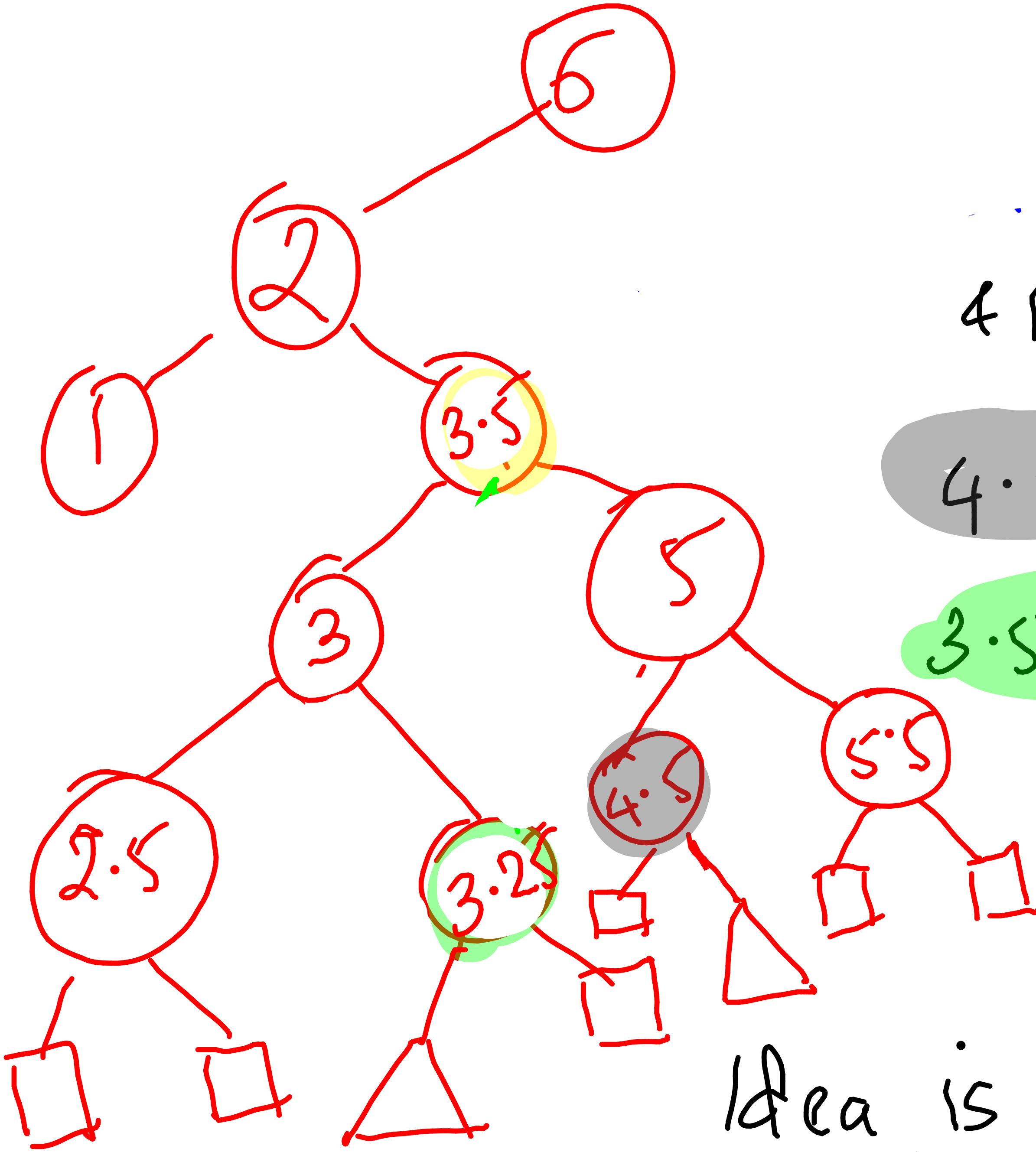
Go to right child & keep moving left

$4.5 = succ(4)$

$3.5 = pred(4)$

Go to left child & keep moving right

Idea is: Find pred or successor of k & replace k with that



Go to right child & keep moving left

$4.5 = succ(4)$

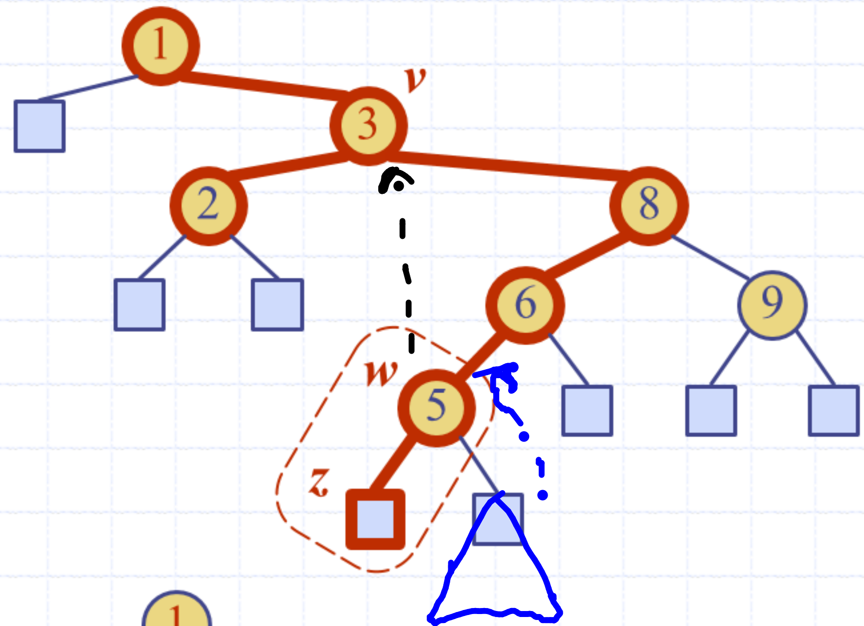
$3.5 = pred(4)$

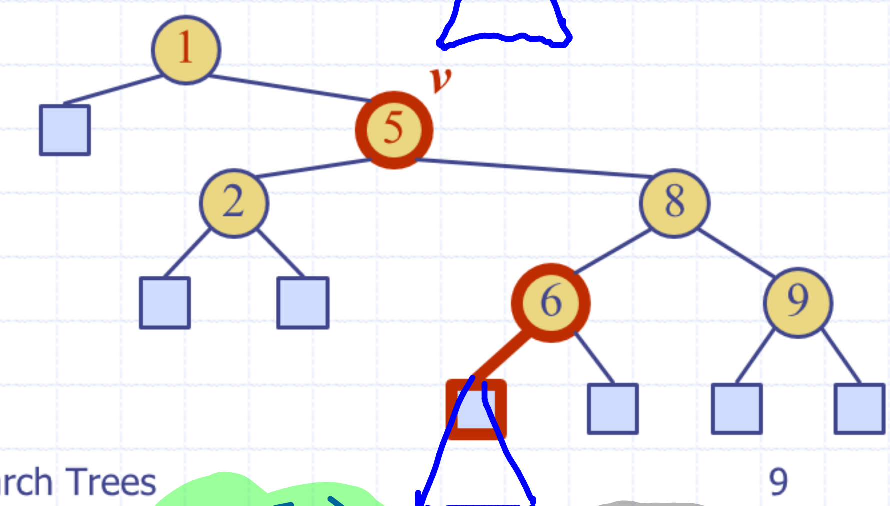
Go to left child & keep moving right

Idea is: Find pred or Successor of k & replace k with that

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





Complexity = $\underbrace{O(h)}_{\text{search}} + \underbrace{O(h)}_{\text{To find pred/succ}} + O(1)$ → moving

Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods find, insert and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case