COMP2123/2823/9123
Data structures and Algorithms

Binary Search Trees

[GT 3.1-2] [GT 4.2]

Dr. Julian Mestre
School of Information Technologies

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

The Map ADT

- get(k): if the map M has an entry with key k, return its associated value
- put(k, v): if key k is not in M, then insert (k, v) into the map M; else, replace the existing value associated to k with v
- remove(k): if the map M has an entry with key k, remove it
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterable collection of the values in M

Example

Operation	*Output*	*Map*
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

Sorted map ADT (extra methods)

firstEntry() returns the entry with smallest key; if map is empty, returns null

lastEntry() returns the entry with largest key; if map is empty, returns null

ceilingEntry(k) returns the entry with least key that is greater than or equal to k (or null, if no such entry exists)

floorEntry(k) returns the entry with greatest key that is less than or equal to k (or null, if no such entry exists)

lowerEntry(k) returns the entry with greatest key that is strictly less than k (or null, if no such entry exists)

higherEntry(k) returns the entry with least key that is strictly greater than k (or null, if no such entry exists)

subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2

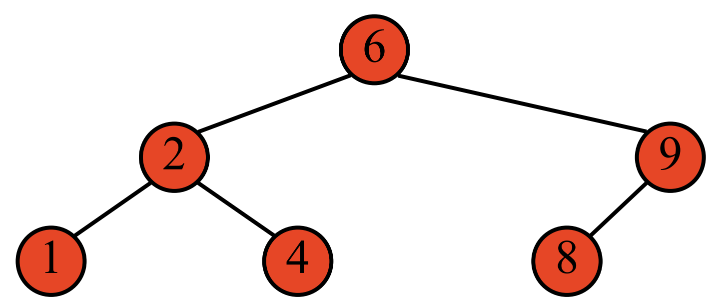
Binary Search Trees (BST)

A binary search tree is a binary tree storing keys (or key-value pairs) satisfying the following BST property

For any node v in the tree and
any node u in the left subtree of v and
any node w in the right subtree of v,

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

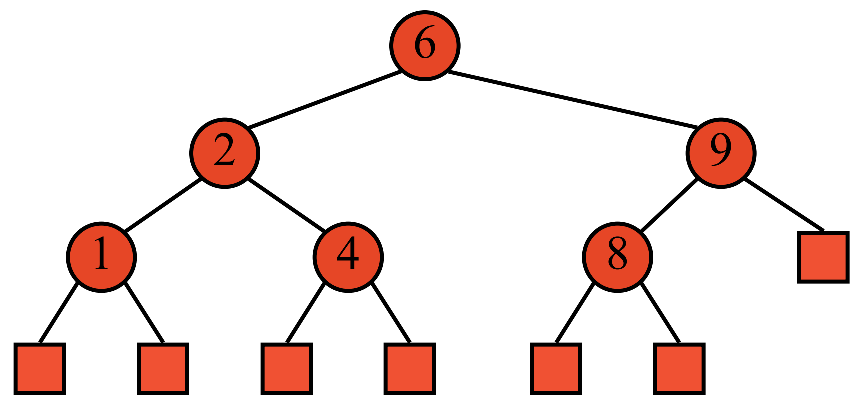
Note that an inorder traversal of a binary search tree visits the keys in increasing order.



BST Implementation

To simplify the presentation of our algorithms, we only store keys (or key-value pairs) at internal nodes

External nodes do not store items (and with careful coding, can be omitted, using null to refer to such)

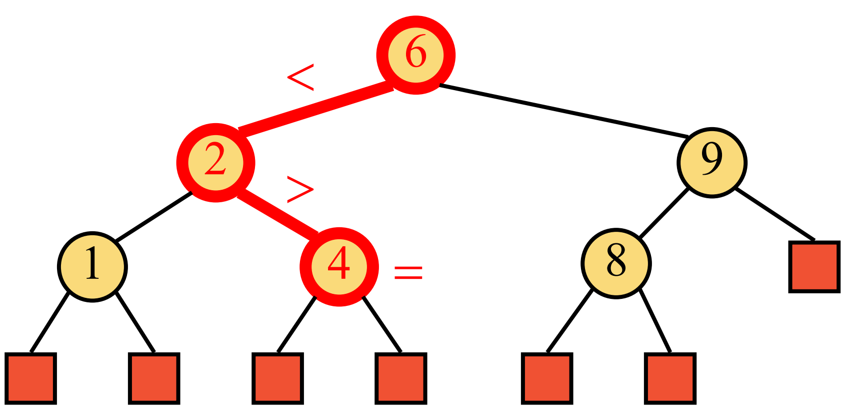


Searching with a Binary Search Tree

To search for a key k, we trace a downward path starting at the root

To decide whether to go left or right, we compare the current node with k

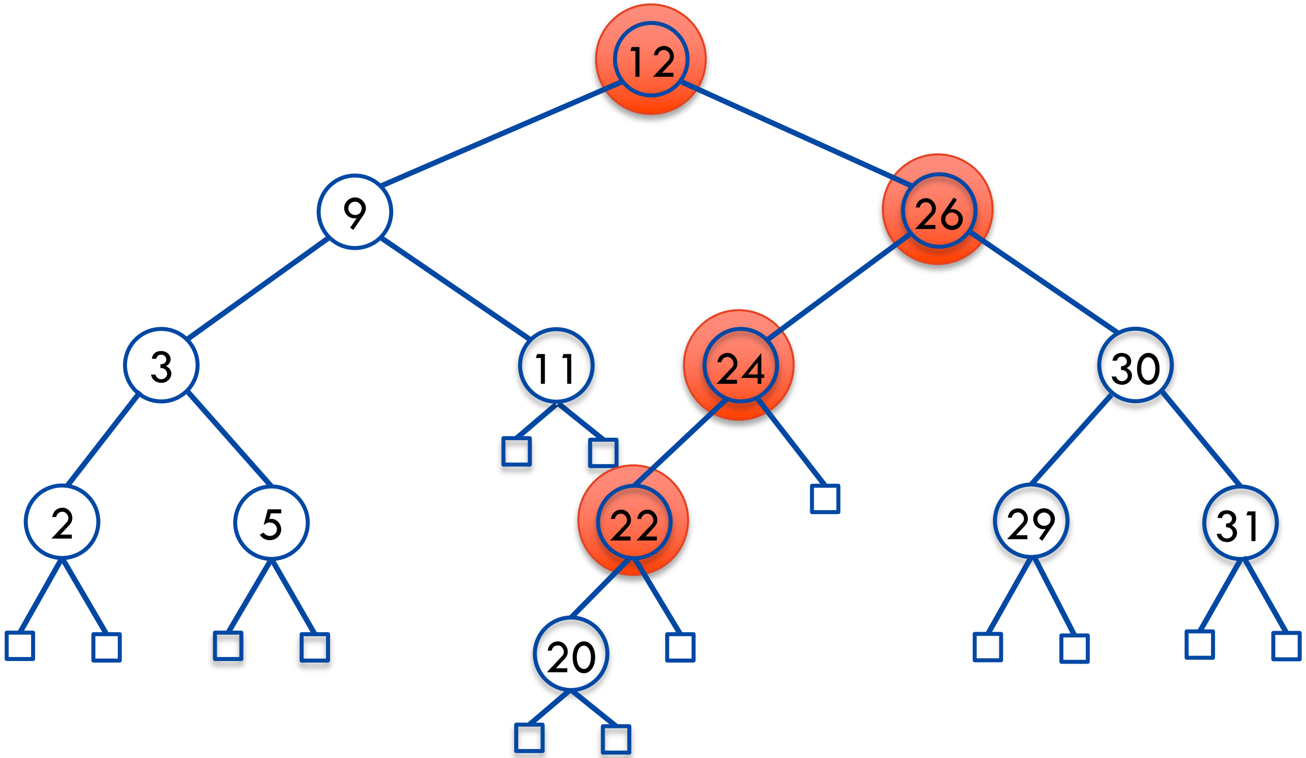
If we reach an external node, this means that the key is in the data structure




```
def search(k, v)
   if v is external then
      # unsuccessful search
      return v
   if k == key(v) then
      # successful search
      return v
   else if k < key(v) then
      # recur on left subtree
      return search (k, v.left)
   else
      # that is  k > key(v)
      # recur on right subtree
      return search(k, v.right)
```

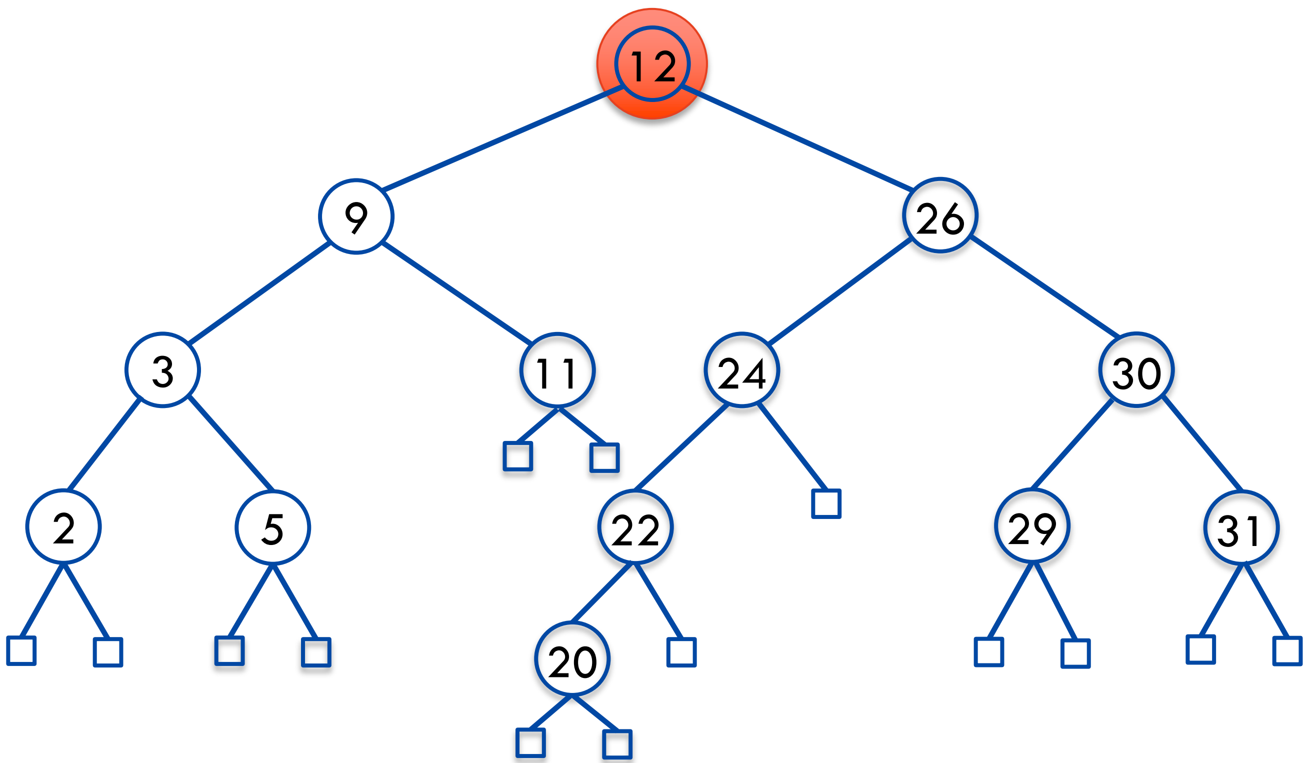
Example: Find 22

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



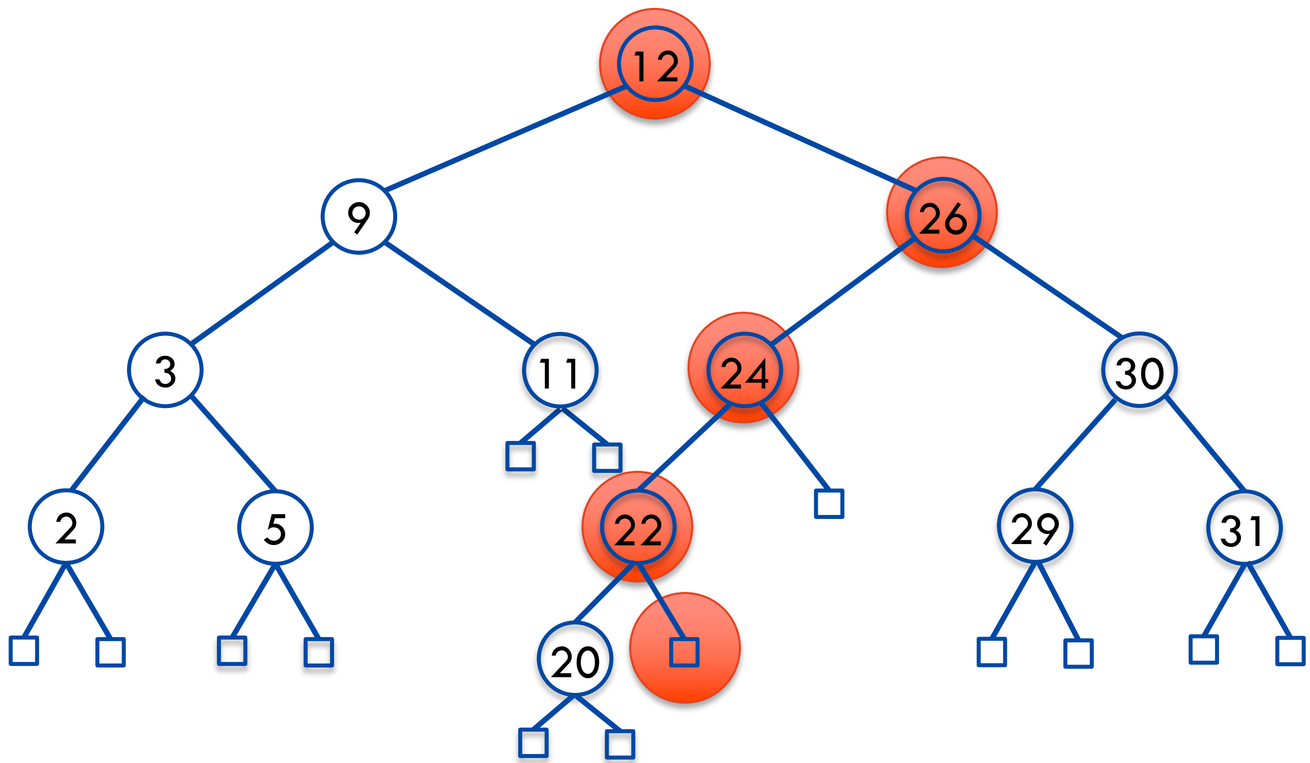
Example: Find 22

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



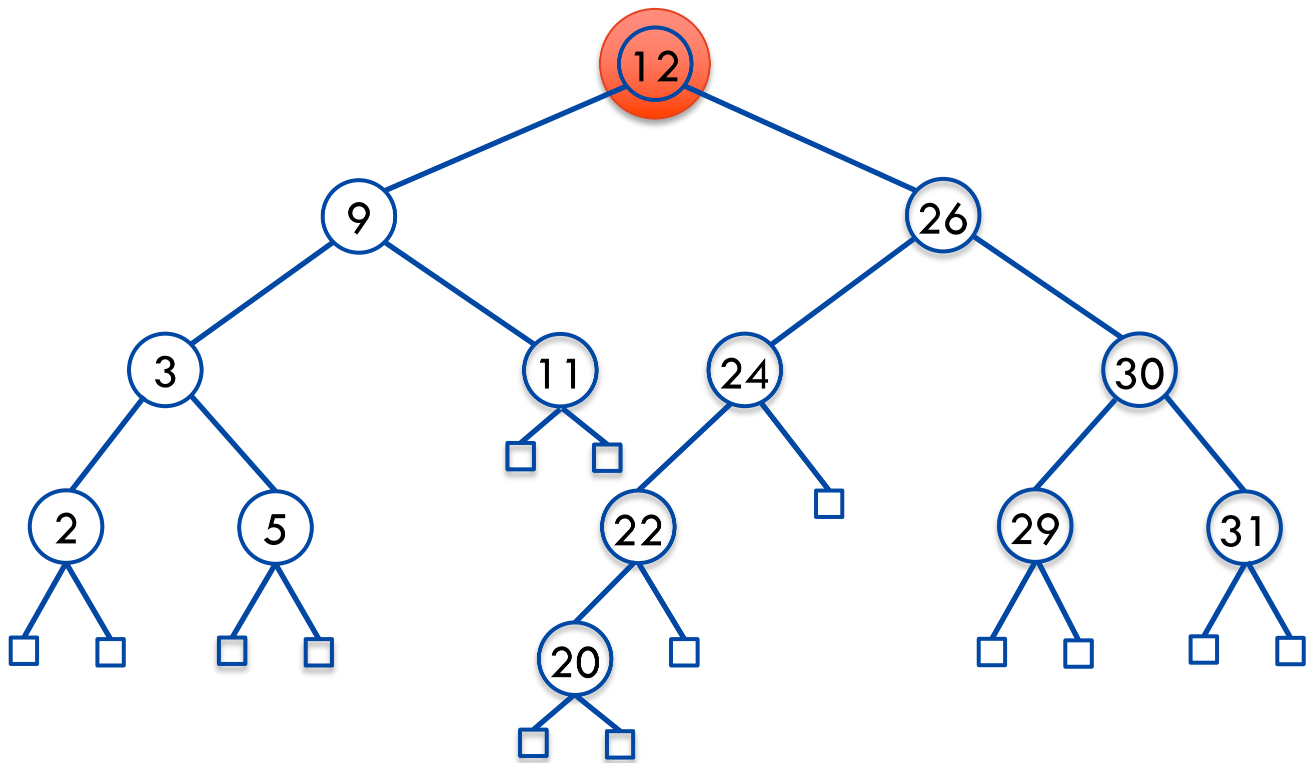
Example: Find 23

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



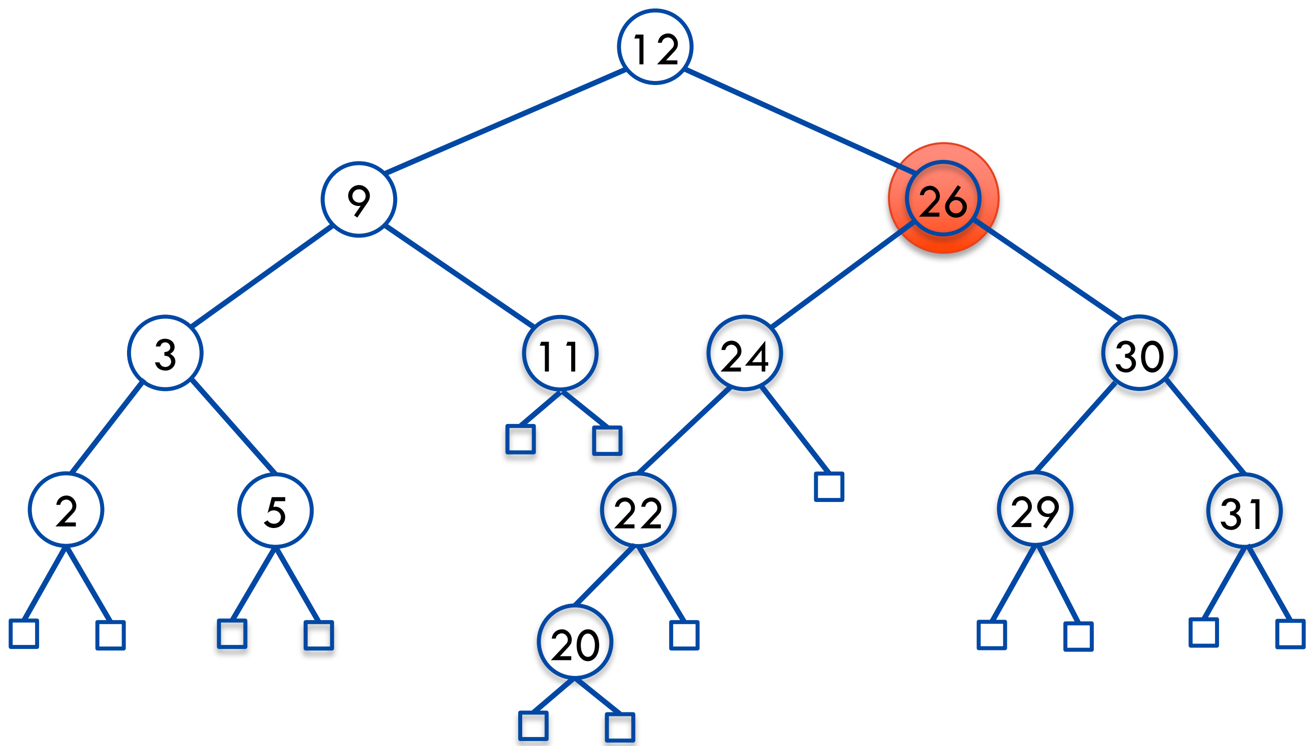
Example: Find 23

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



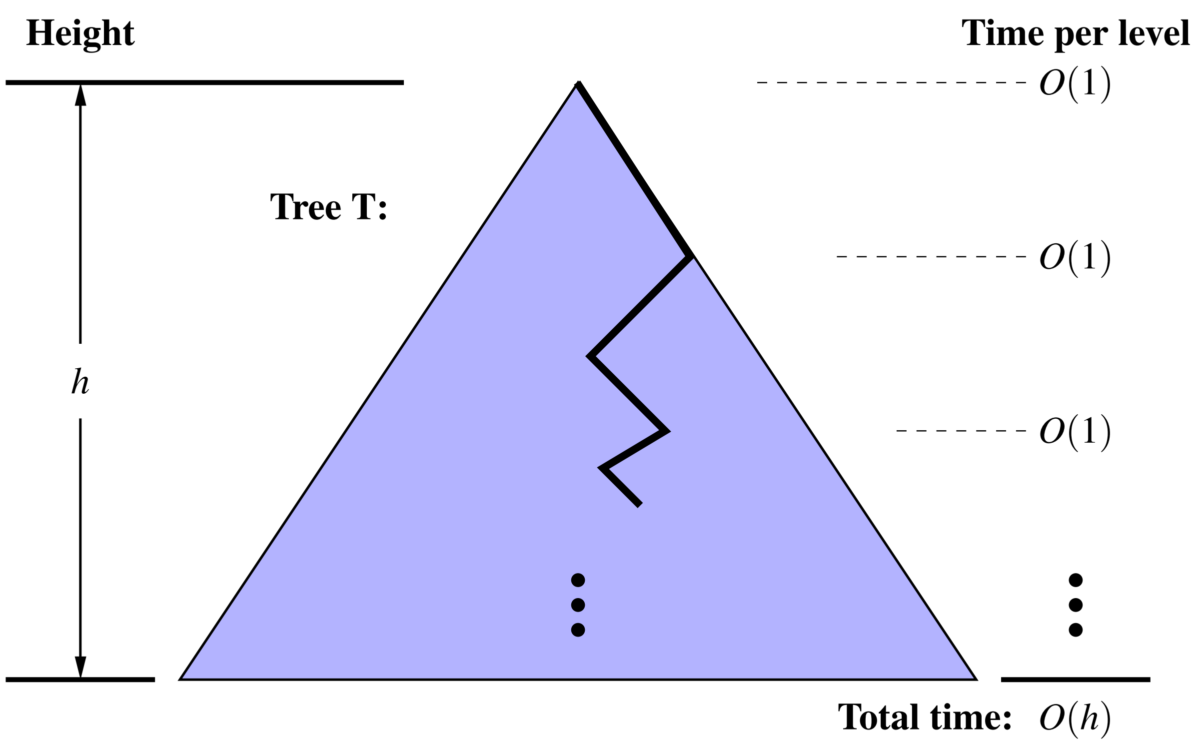
Example: Find 23

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



Analysis of Binary Tree Searching

- Runs in O(h) time, where h is the height of the tree
- Worst case is $h = O(n)$ but there are "balanced tree" strategies to maintain $h \le \log(n)$

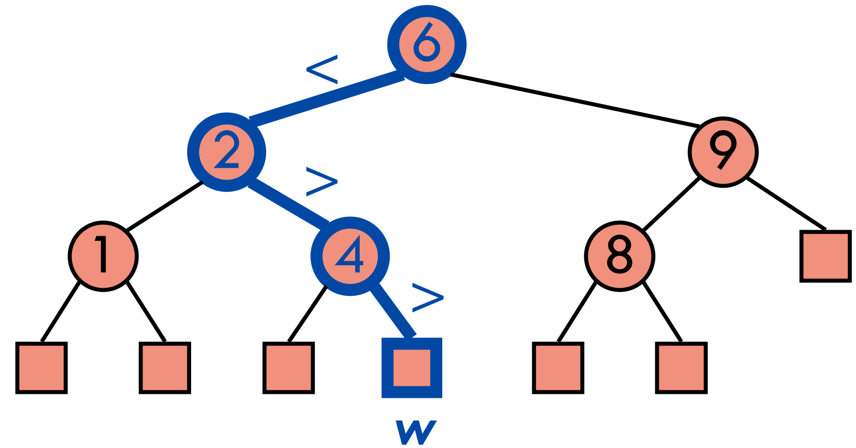


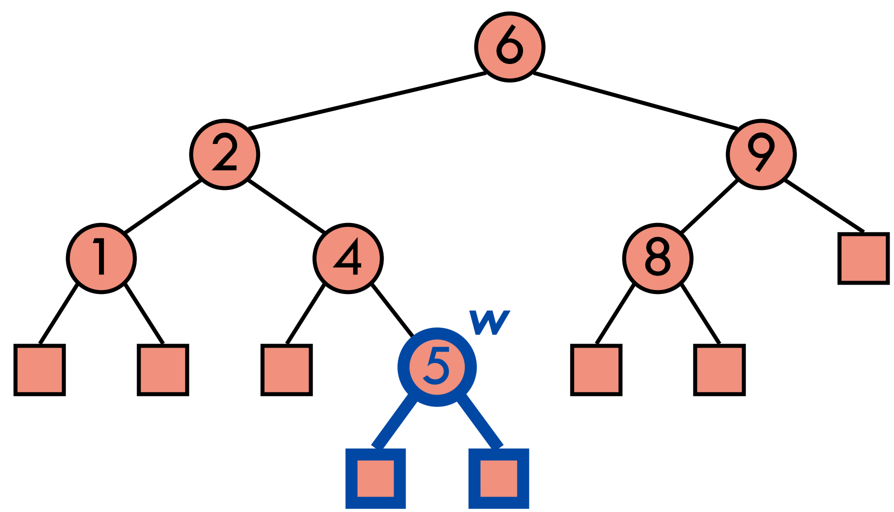
Insertion

To perform operation put(k, o), we search for key k (using search)

If k is found in the tree, replace the corresponding value by o

If k is not found, let w be the external node reached by the search. We replace w with an internal node holding (k, o)

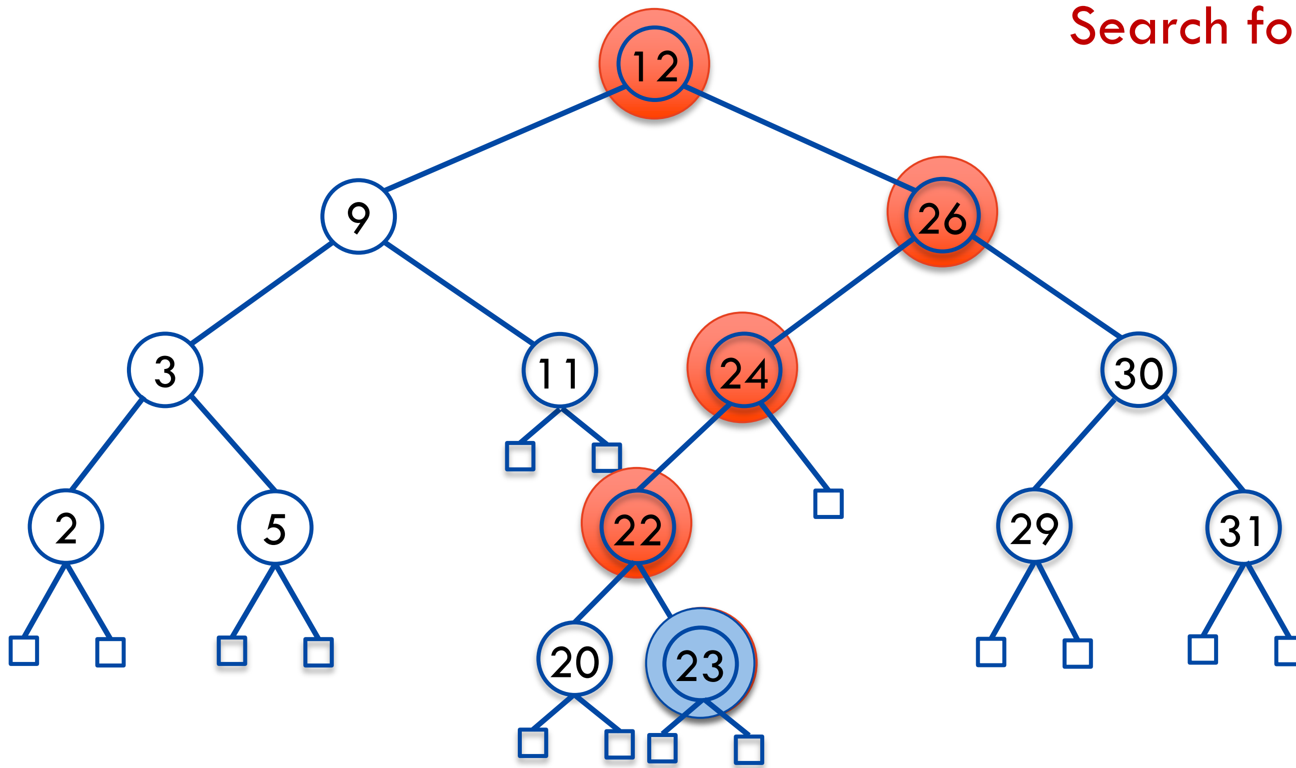




Example: Insert 23

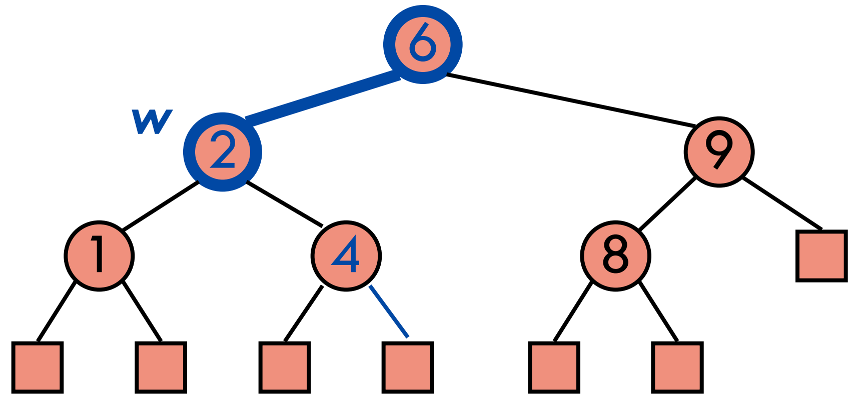
S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Search for 23



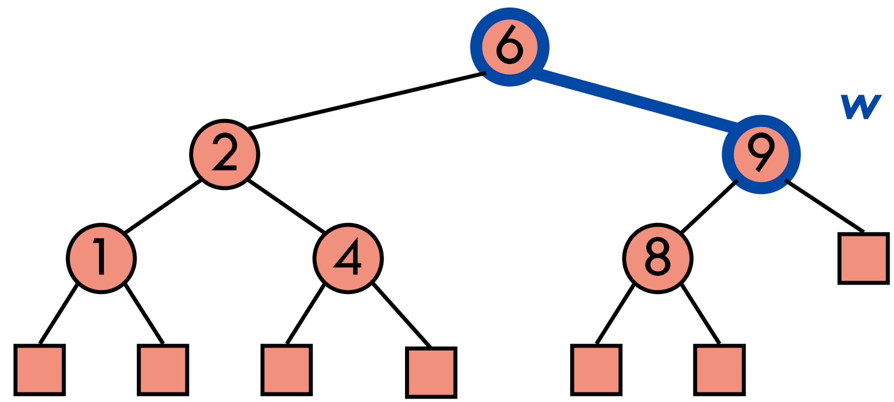
Delete

To perform operation remove(k), we search for key k (using search) to find the node w holding k



We distinguish between two cases

- w has one external child
- w has two internal children



If k is not in the tree we can either throw an exception or do nothing depending on the ADT specs

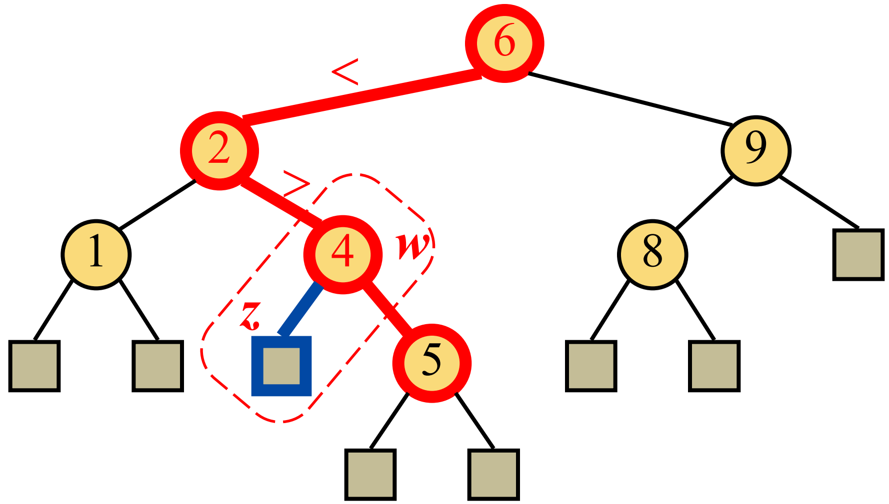
Deletion Case 1

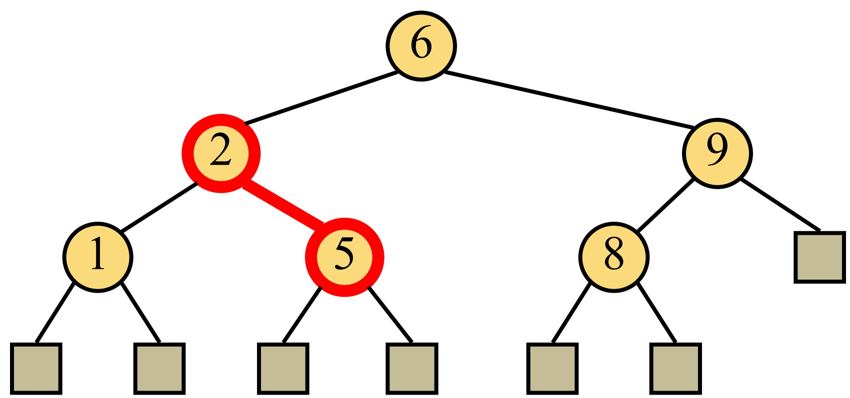
Suppose that the node w we want to remove has an external child, which we call z.

To remove w we

- remove w and z from the tree
- promote the other child of w to take w's place

This preserves the BST property

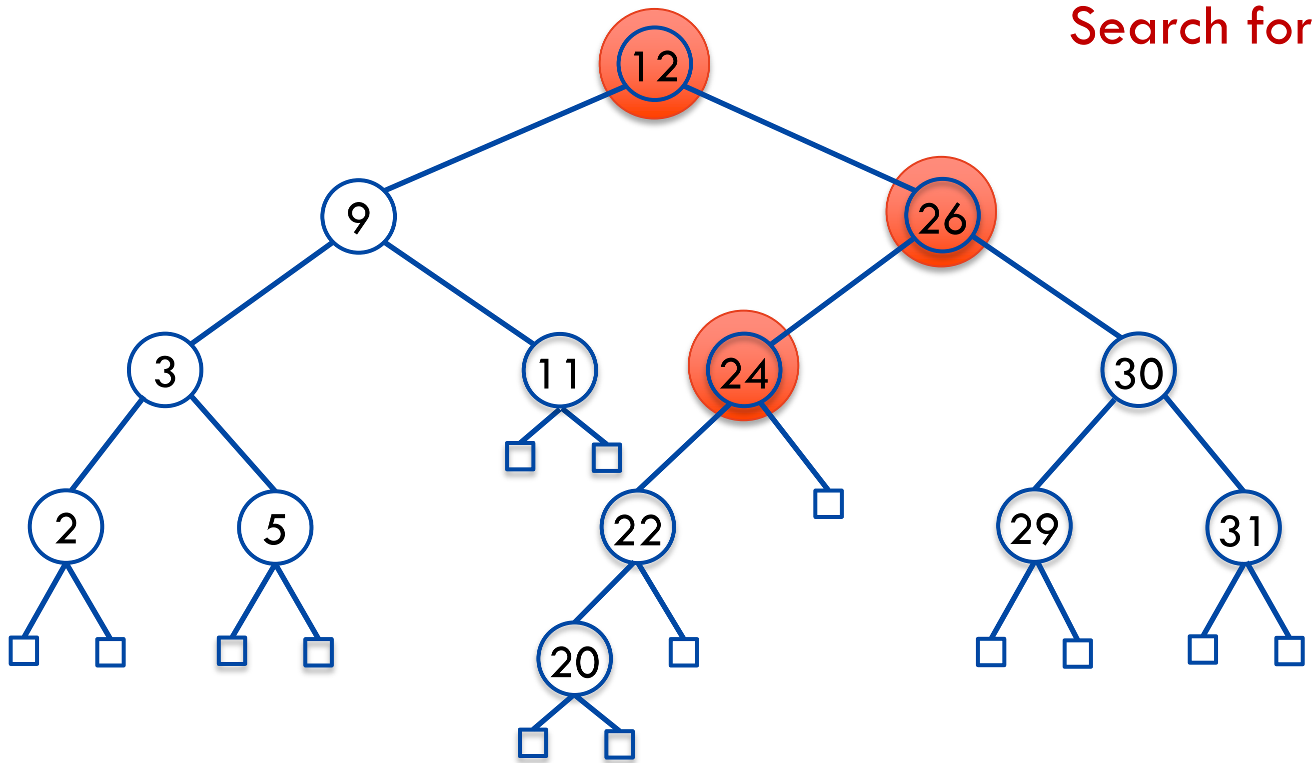




Example: Delete 24

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

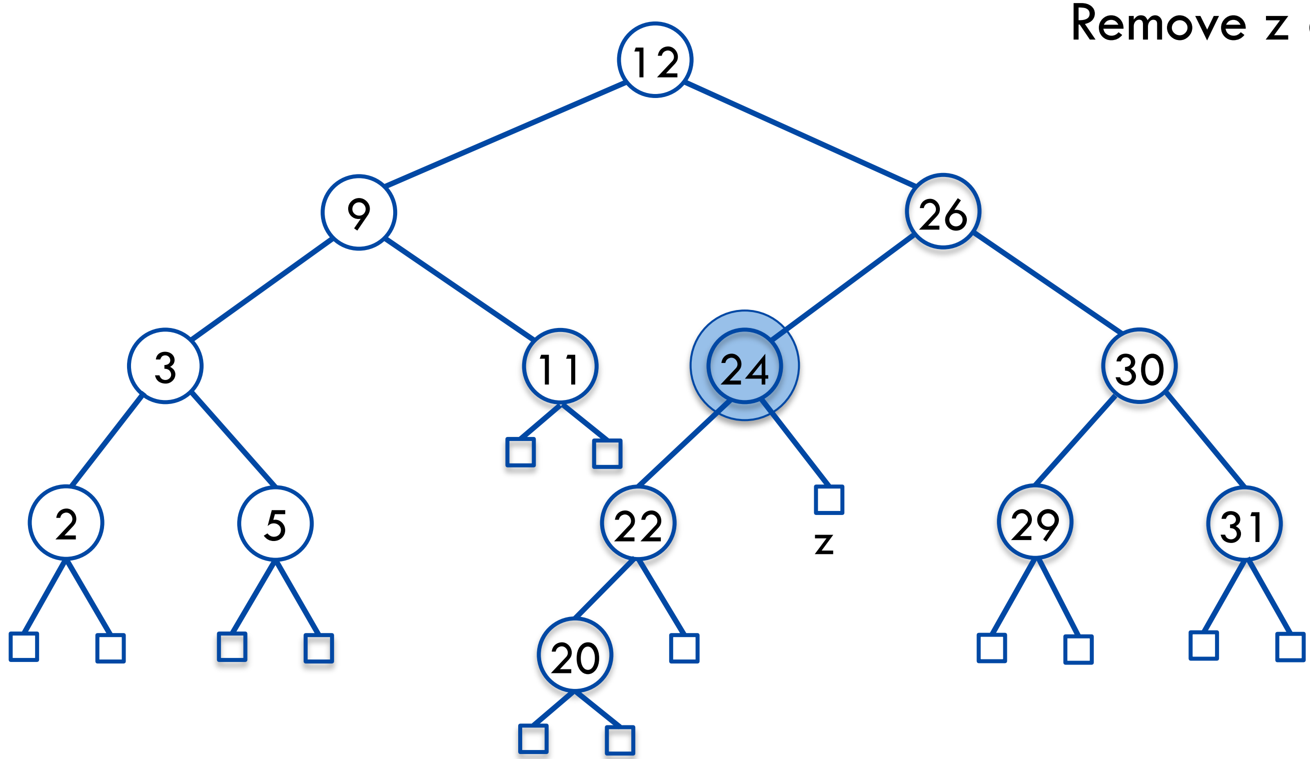
Search for 24



Example: Delete 24

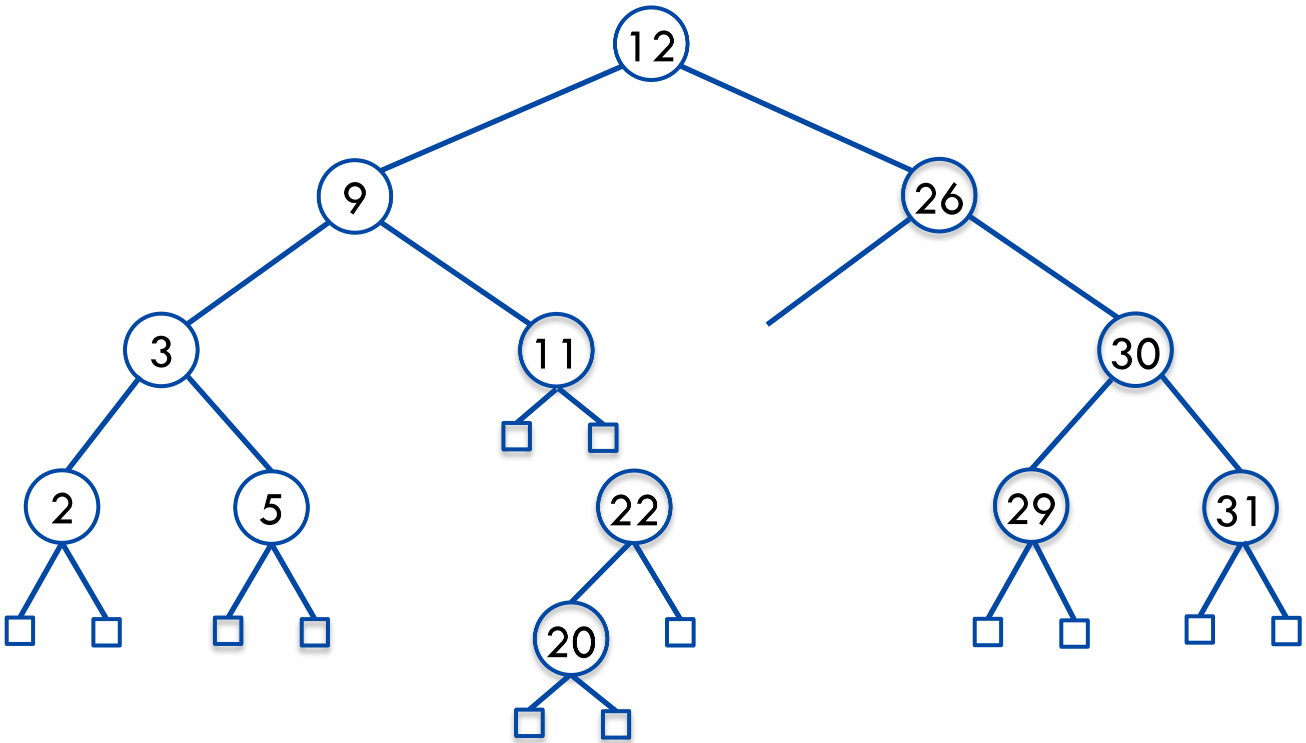
S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Remove z and w



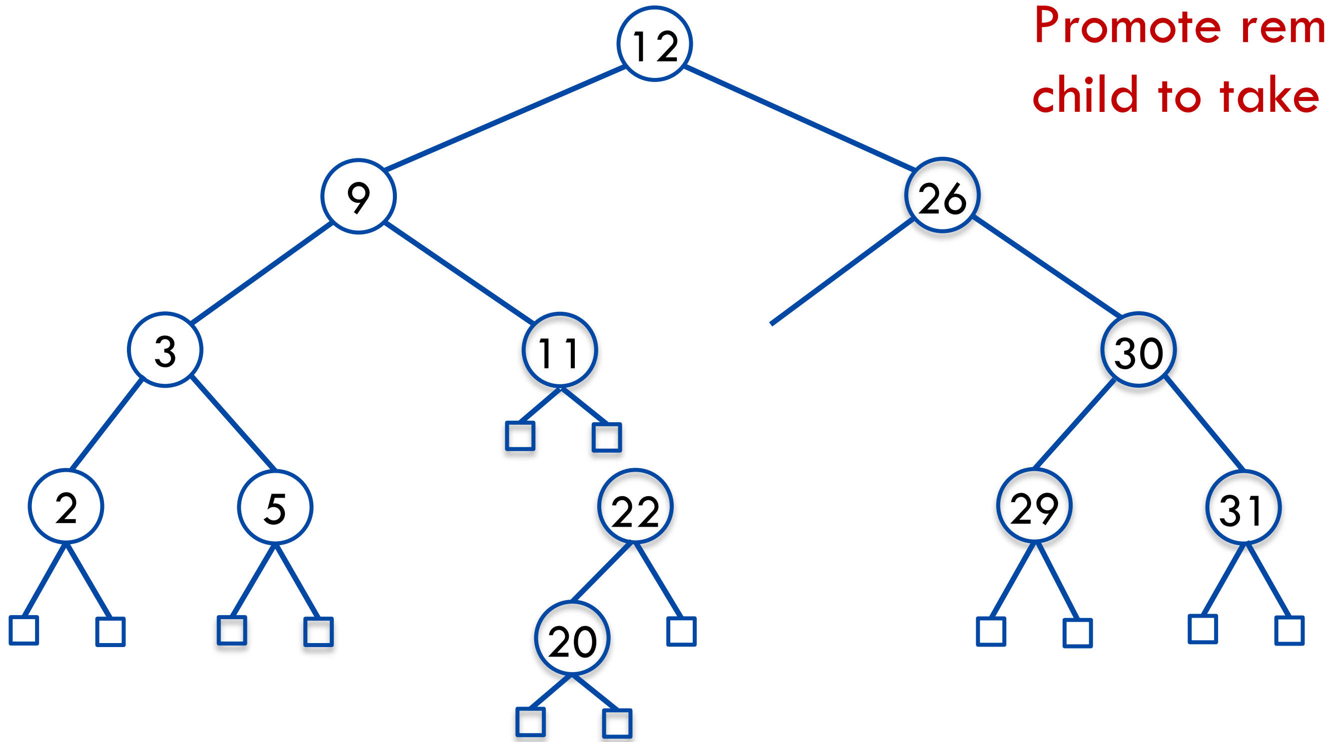
Example: Delete 24

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



Example: Delete 24

S={2,3,5,9,11,12,20,22,24,26,29,30,31}



Promote remaining child to take spot of w

Deletion : Case 2

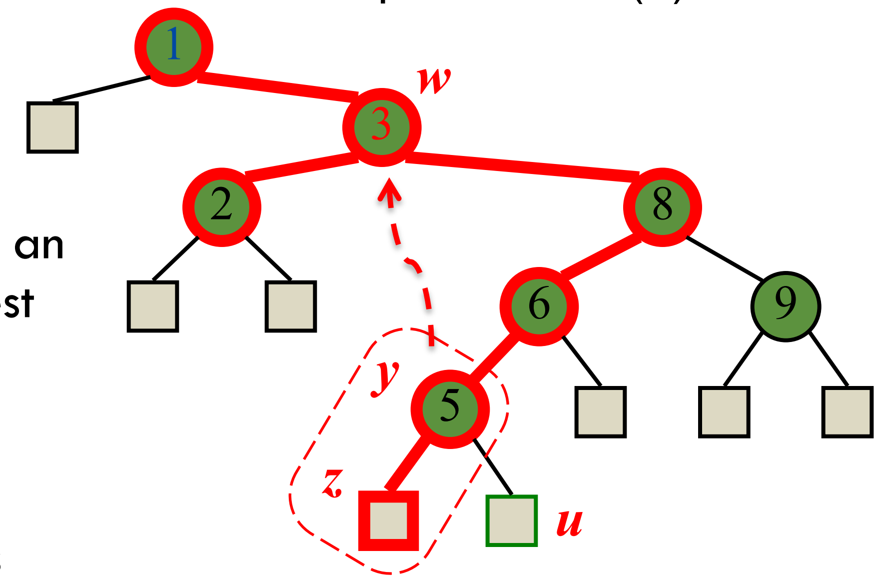
Suppose that the node w we want to remove has two internal children.

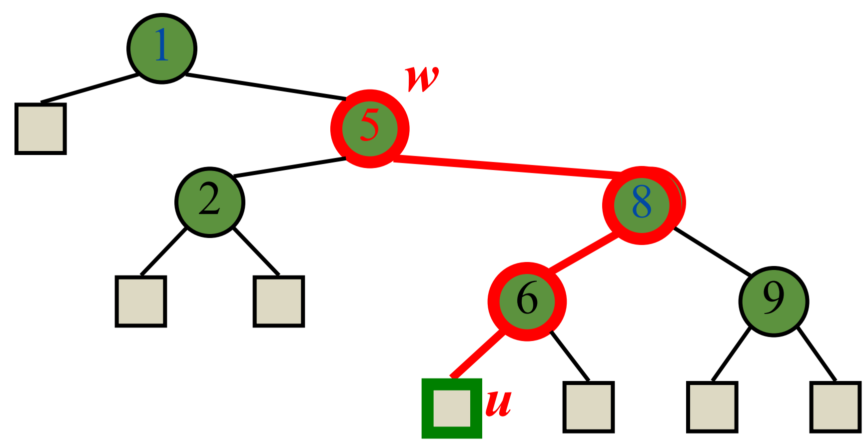
To remove w we

- find the internal node y following w in an inorder traversal (i.e., y has the smallest key among the right subtree under w)
- we copy the entry from y into node w
- we remove node y and its left child z, which must be external, using previous case

This preserves the BST property

Example: remove(3)





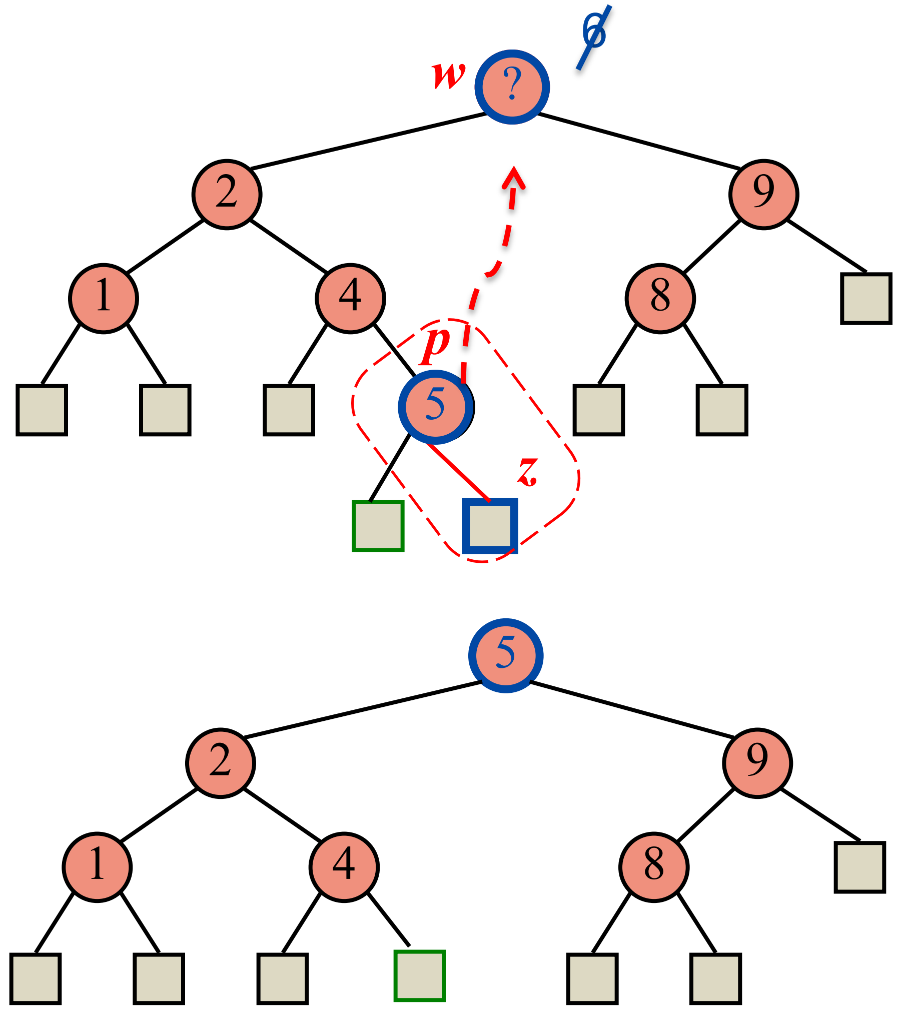
Deletion algorithm

```
def remove(k)
  w ← search(root, k)
  if w is external then
    # key not found
    return null
  else if w has at least one external child external z
    remove z
    promote the other child of w to take w's place
    remove w
  else
    # y is leftmost internal node in the right subtree of w
    y ← immediate successor of w
    replace contents of w with entry from y
    remove(y)
```

Deletion Case 2 variant

It is also possible to use the *largest key of the left subtree:*

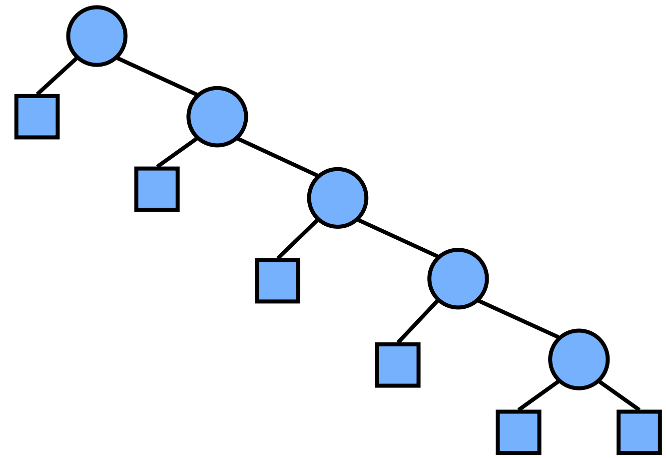
- find the internal node *p* that *immediately precedes w* in an inorder traversal
- copy *entry from p* into node *w*
- remove node *p* and its right child *z* (which must be external)



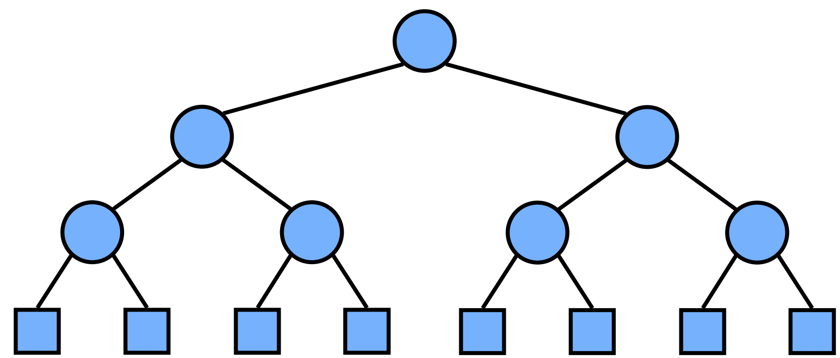
Complexity

Consider a map with n items implemented by means of a binary search tree of height h:

- the space used is O(n)
- get, put and remove take O(h) time



The height h can be n in the worst case and log n in the best case.



Therefore the best one can hope is that tree operations take O(log n) time but in general we can only guarantee O(n). But the former can be achieved with better insertion routines.

Duplicate key values in BST

Our definition says that keys are in strict increasing order

key(left descendant) < key(node) < key(right descendant)

This means that with this definition duplicate key values are not allowed (as needed when implementing Map)

However, it is possible to change it to allow duplicates. But that means additional complexity in the BST implementation:

- Allowing left (right) descendants to be equal to the parent
- Using a list to store duplicates

Range Queries

Range query is defined by two values k_1 and k_2. We are to find all keys k stored in T such that $k_1 \leq k \leq k_2$

E.g., find all cars on eBay priced between 10K and 15K.

The algorithm is a restricted version of inorder traversal. When at node v:

- if $key(v) < k_1$: Recursively search right subtree
- if $k_1 \leq key(v) \leq k_2$: Recursively search left subtree, add v to range output, search right subtree
- if $k_2 < key(v)$: Recursively search left subtree

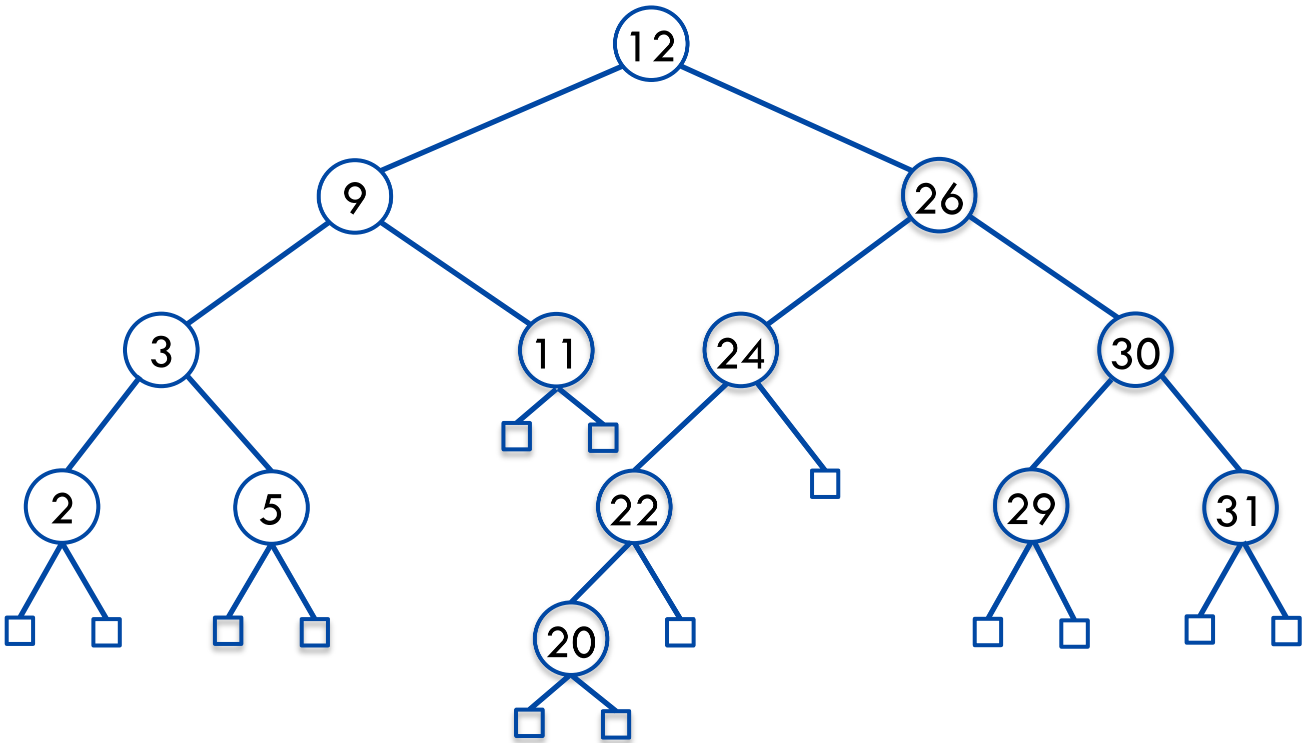
Pseudo-code

```
def range_search(T, k1, k2)
    output ← []
    range(T.root, k1, k2)
```

```
def range(v, k1, k2)
    if v is external then
        return
    if key(v) > k2 then
        range(v.left, k1, k2)
    else if key(v) < k1 then
        range(v.right, k1, k2)
    else
        range(v.left, k1, k2)
        output.append(v)
        range(v.right, k1, k2)
```

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

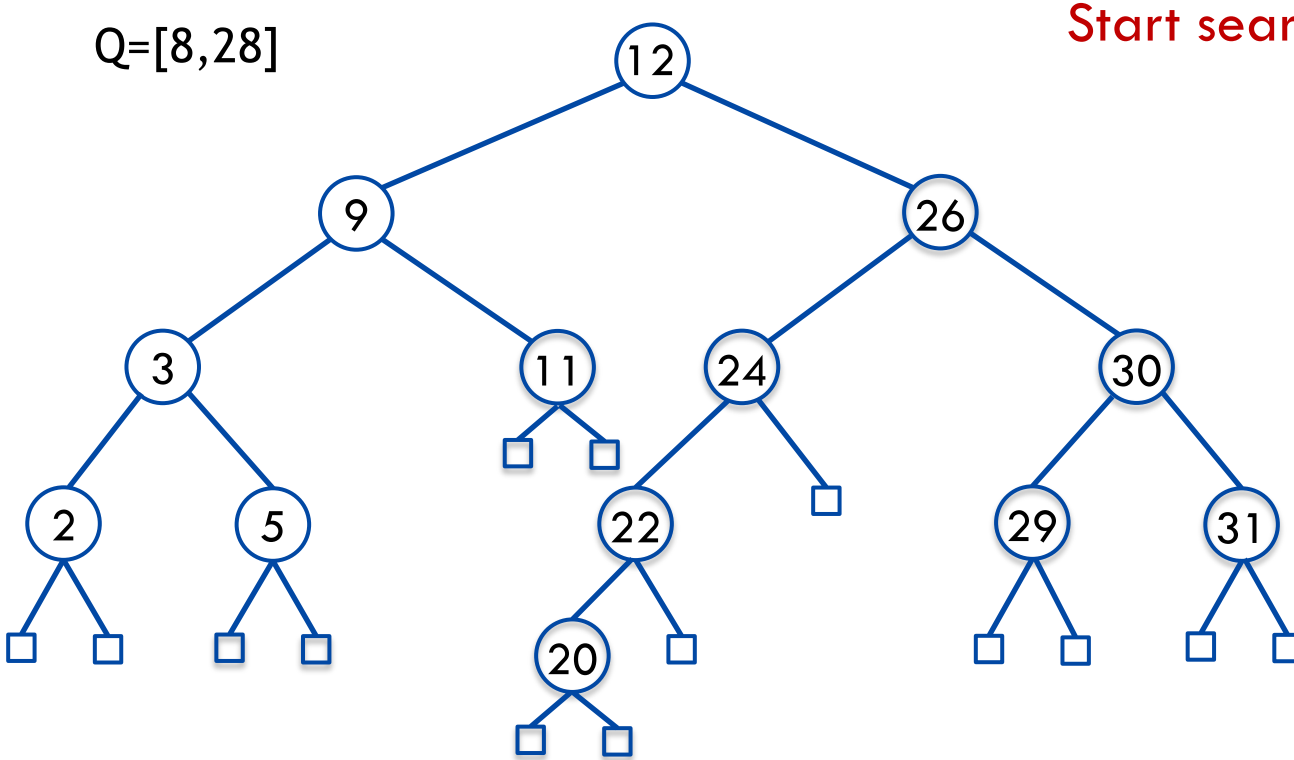


Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]

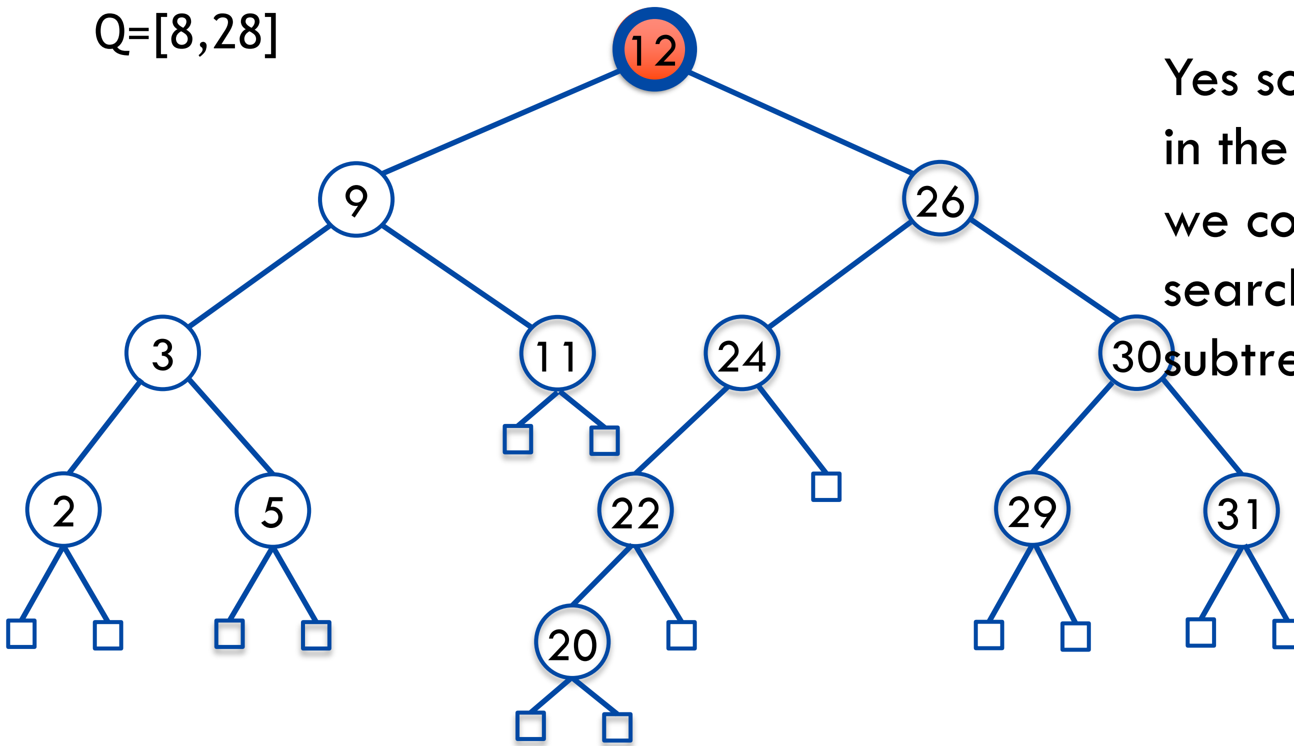
Start search



Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



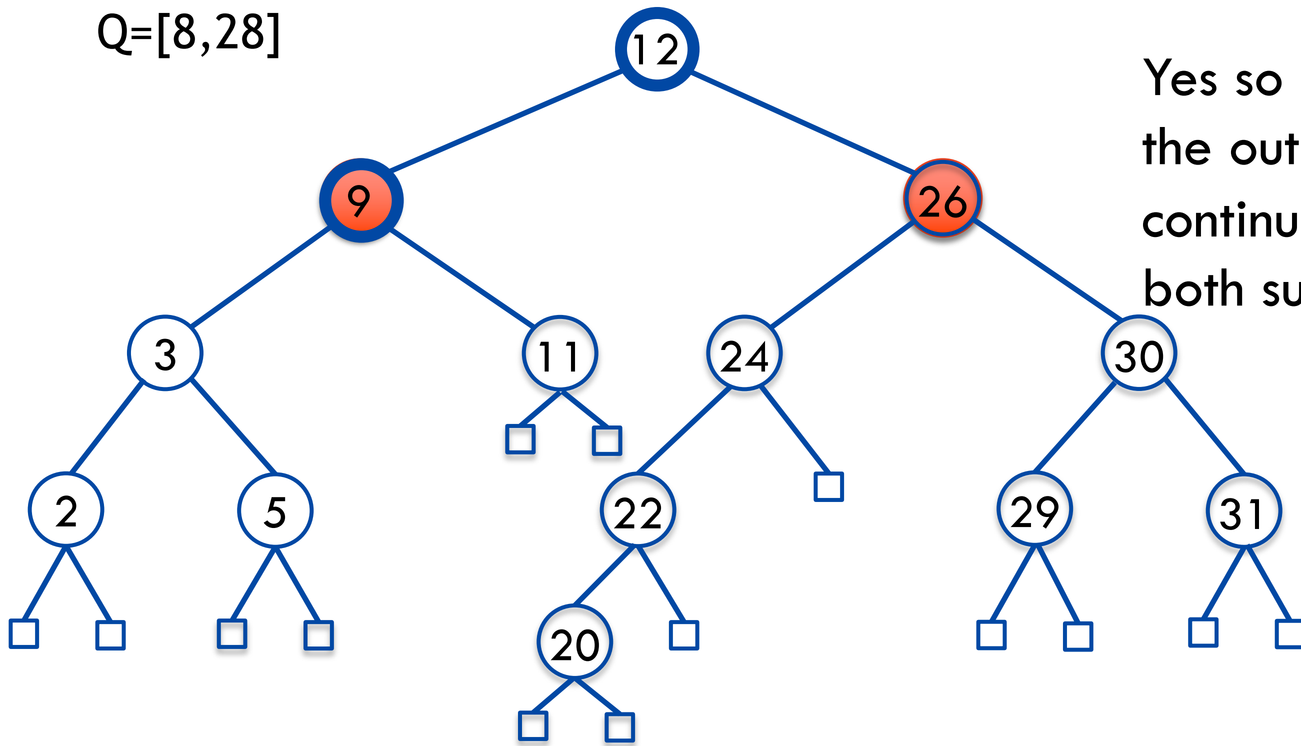
Is 12 in Q?

Yes so 12 will be in the output and we continue search in both subtrees

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



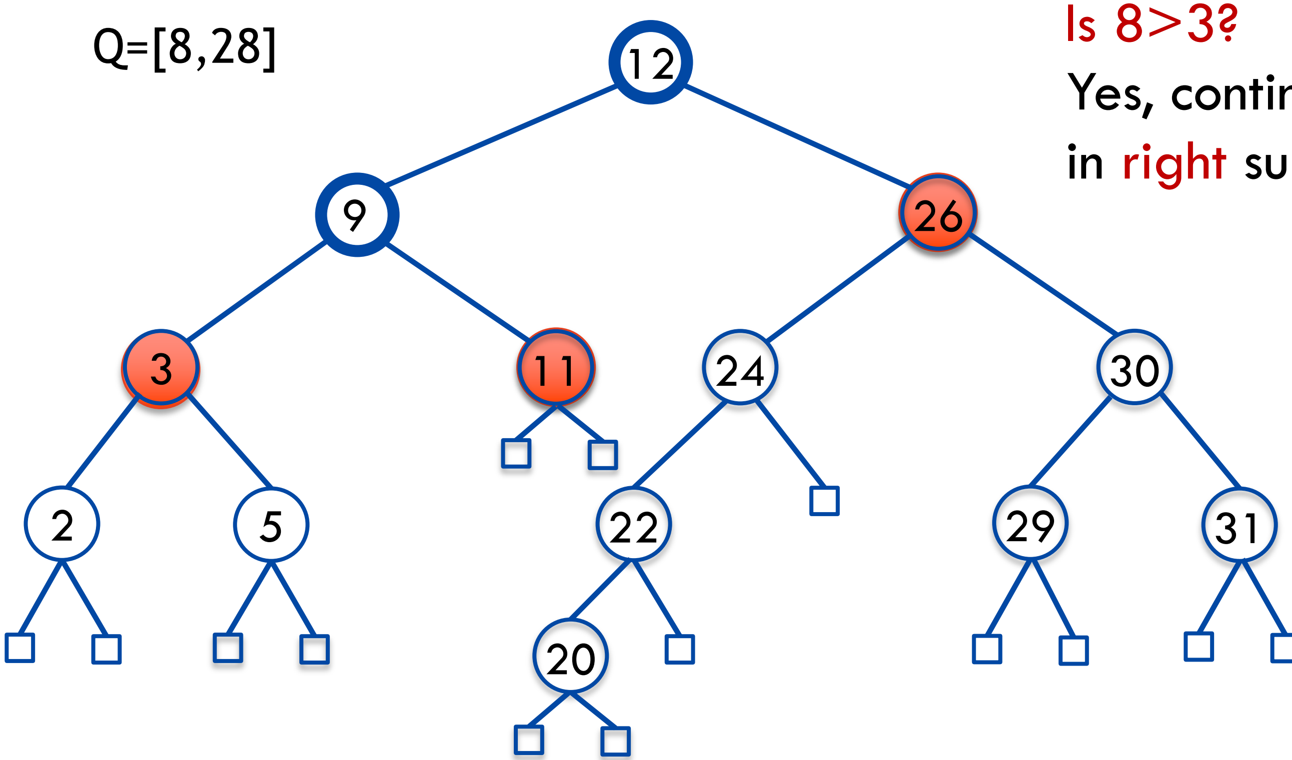
Is 9 in Q?

Yes so 9 will be in the output and we continue search in both subtrees

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



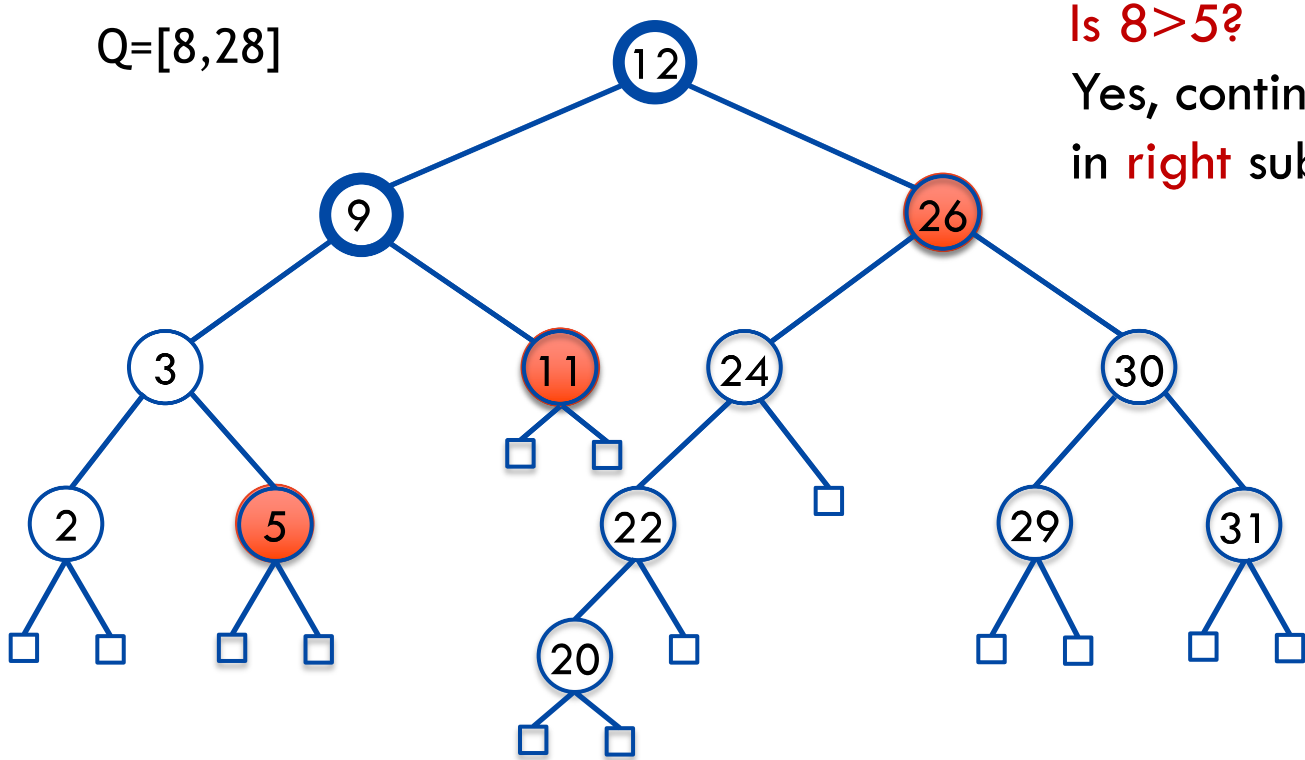
Is 8>3?

Yes, continue search in right subtree.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



Is 8>5?

Yes, continue search in right subtree.

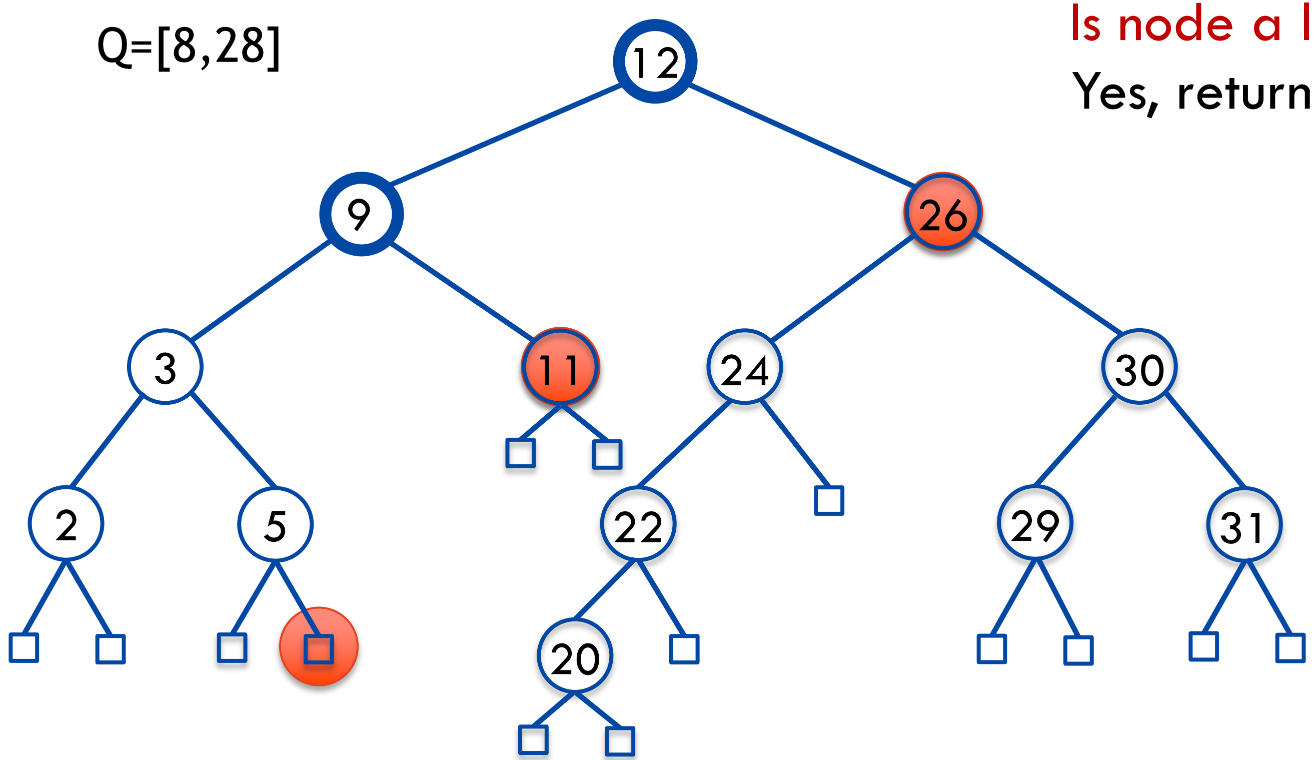
Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]

Is node a leaf?

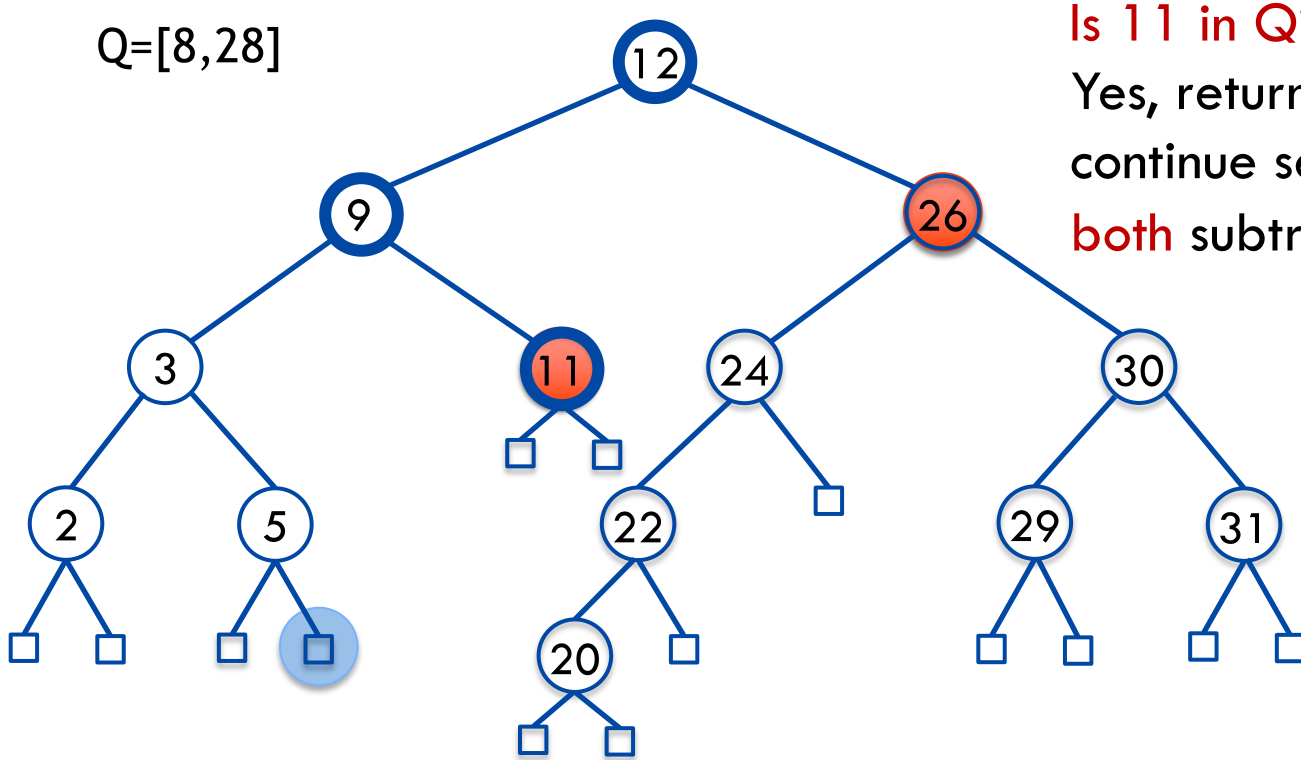
Yes, return ∅.



Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



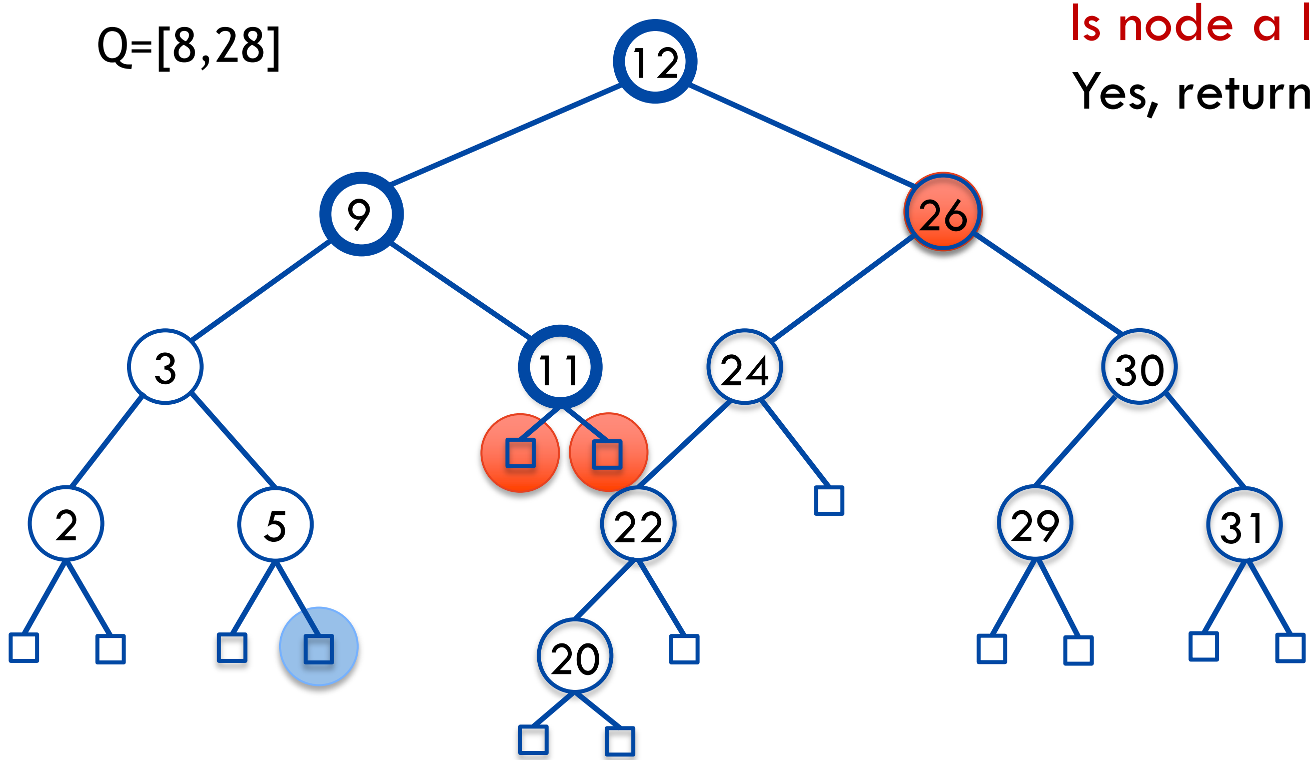
Is 11 in Q?

Yes, return 11 and continue search in both subtrees.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



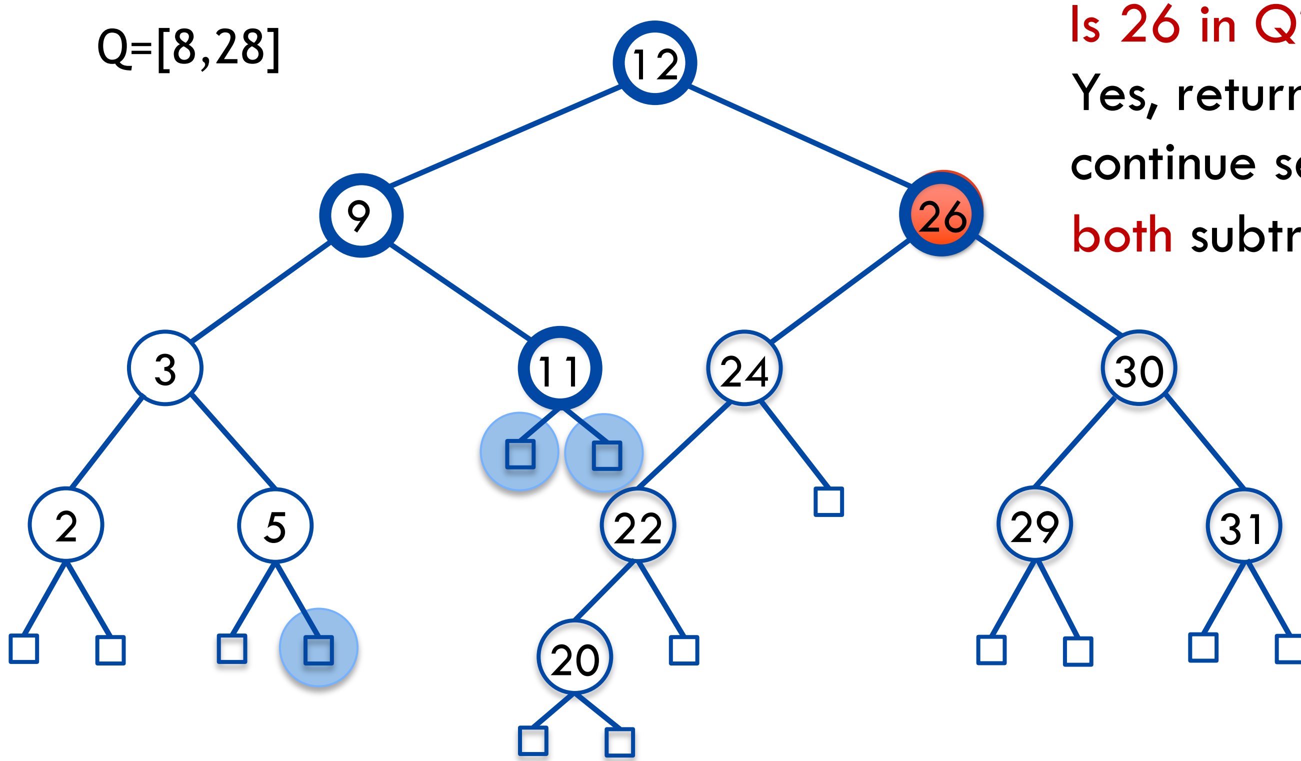
Is node a leaf (×2)?

Yes, return ∅.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



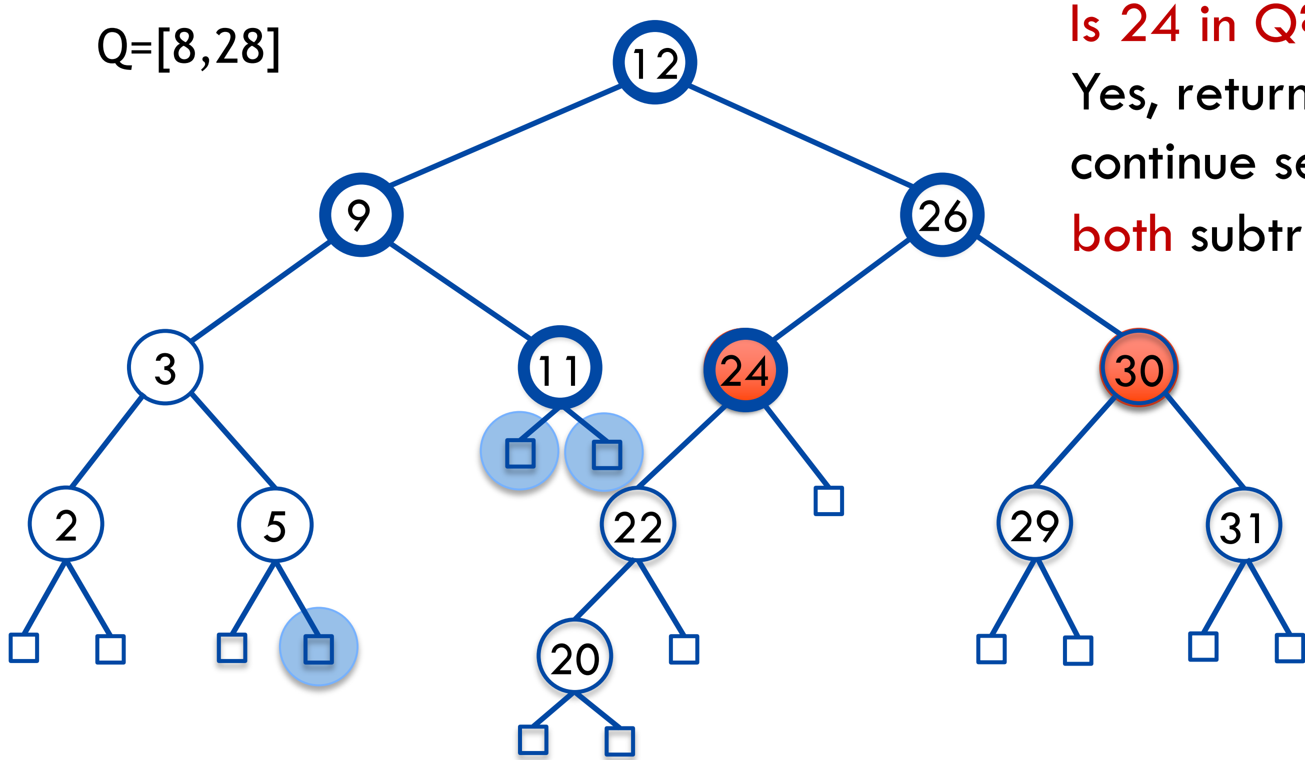
Is 26 in Q?

Yes, return 26 and continue search in both subtrees.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

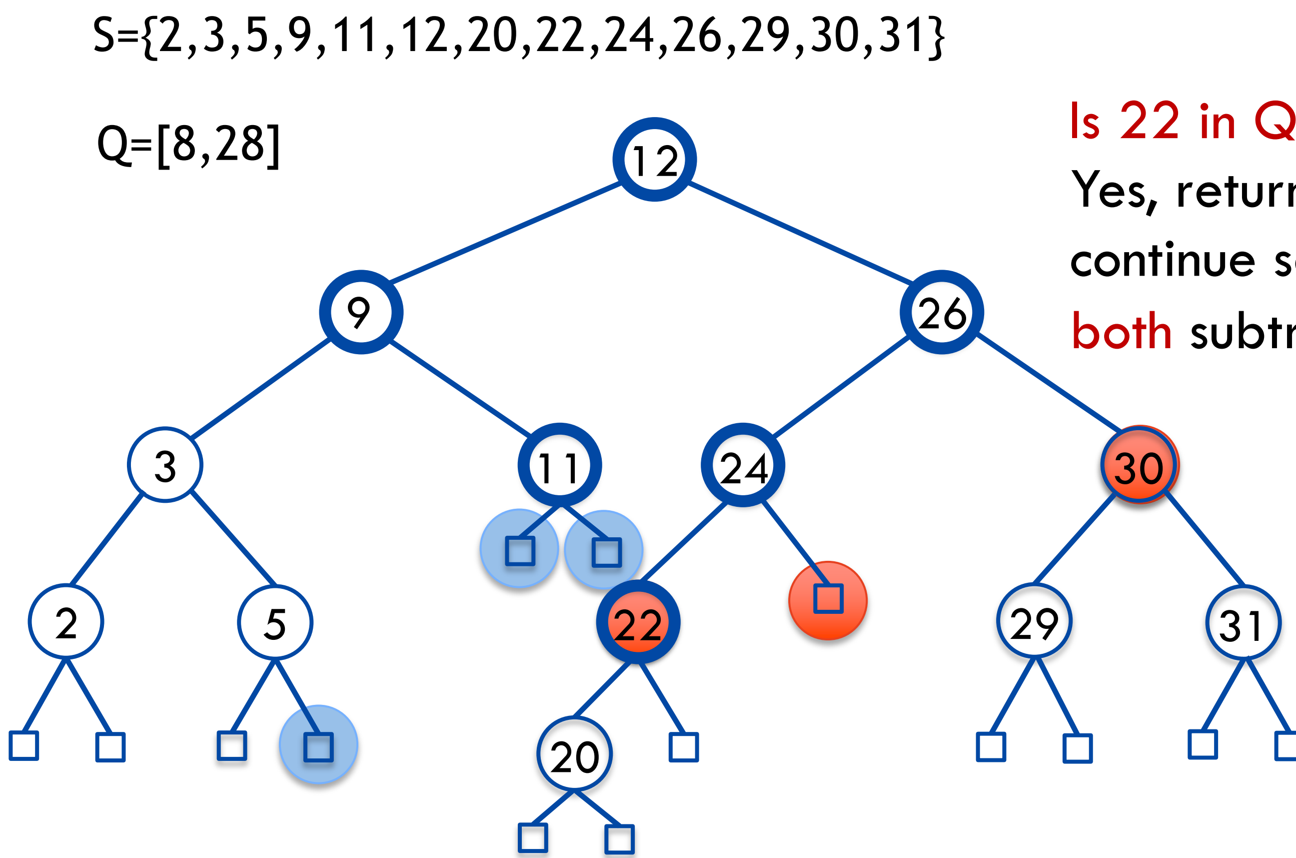
Q=[8,28]



Is 24 in Q?

Yes, return 24 and continue search in both subtrees.

Range queries



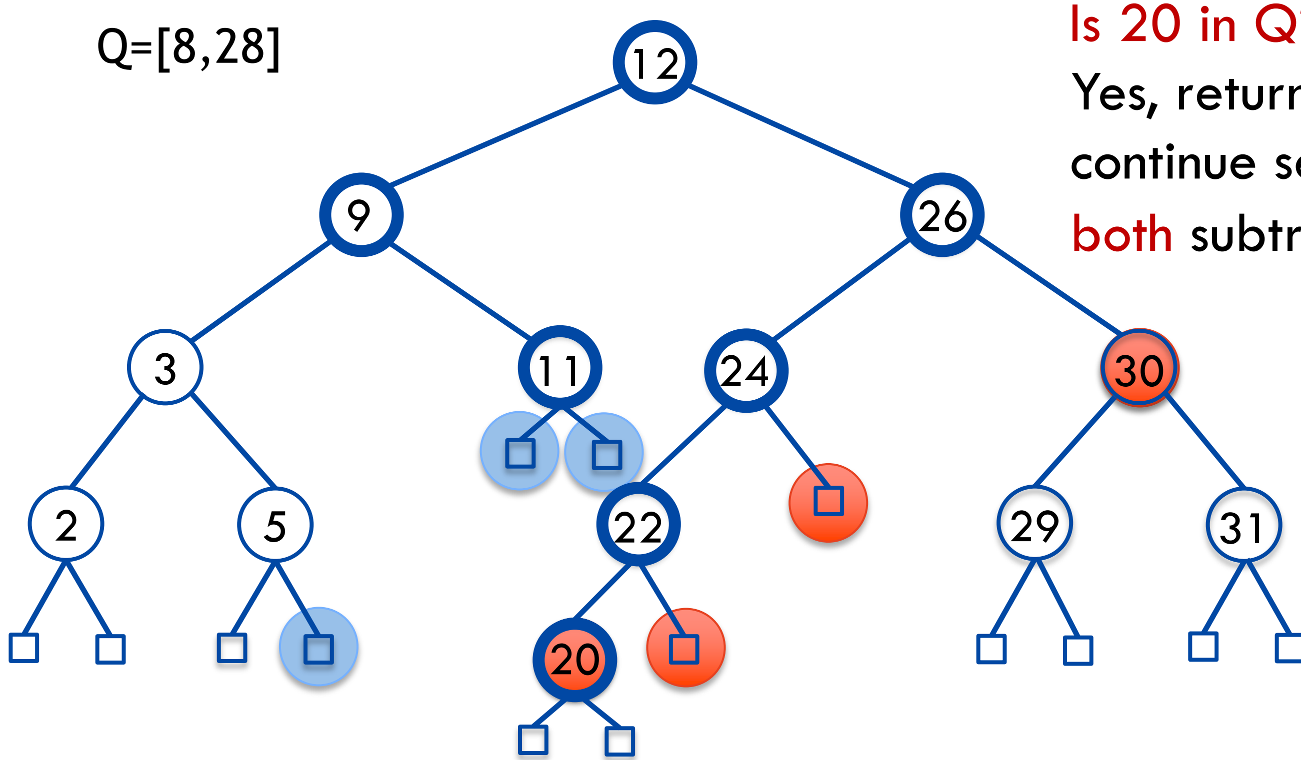
Is 22 in Q?

Yes, return 22 and continue search in both subtrees.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



Is 20 in Q?

Yes, return 20 and continue search in **both** subtrees.

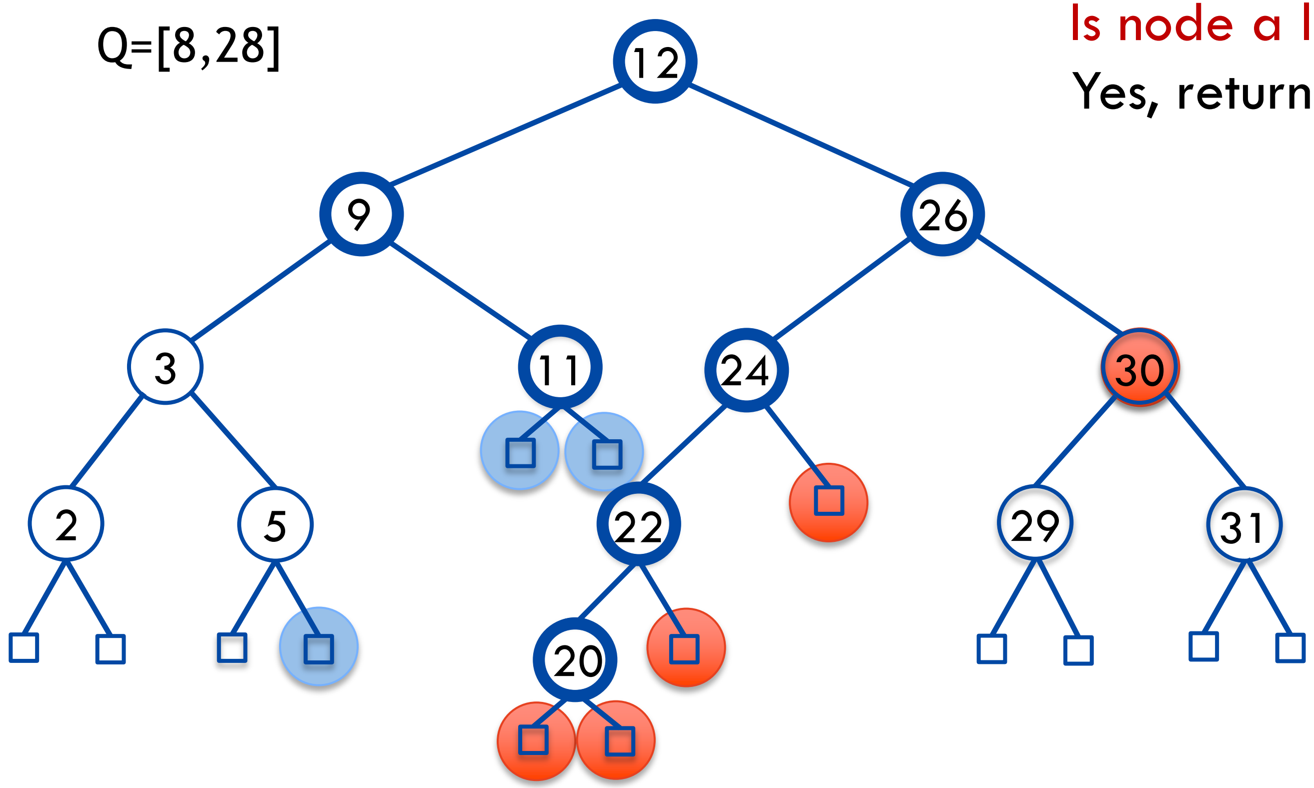
Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]

Is node a leaf (×4)?

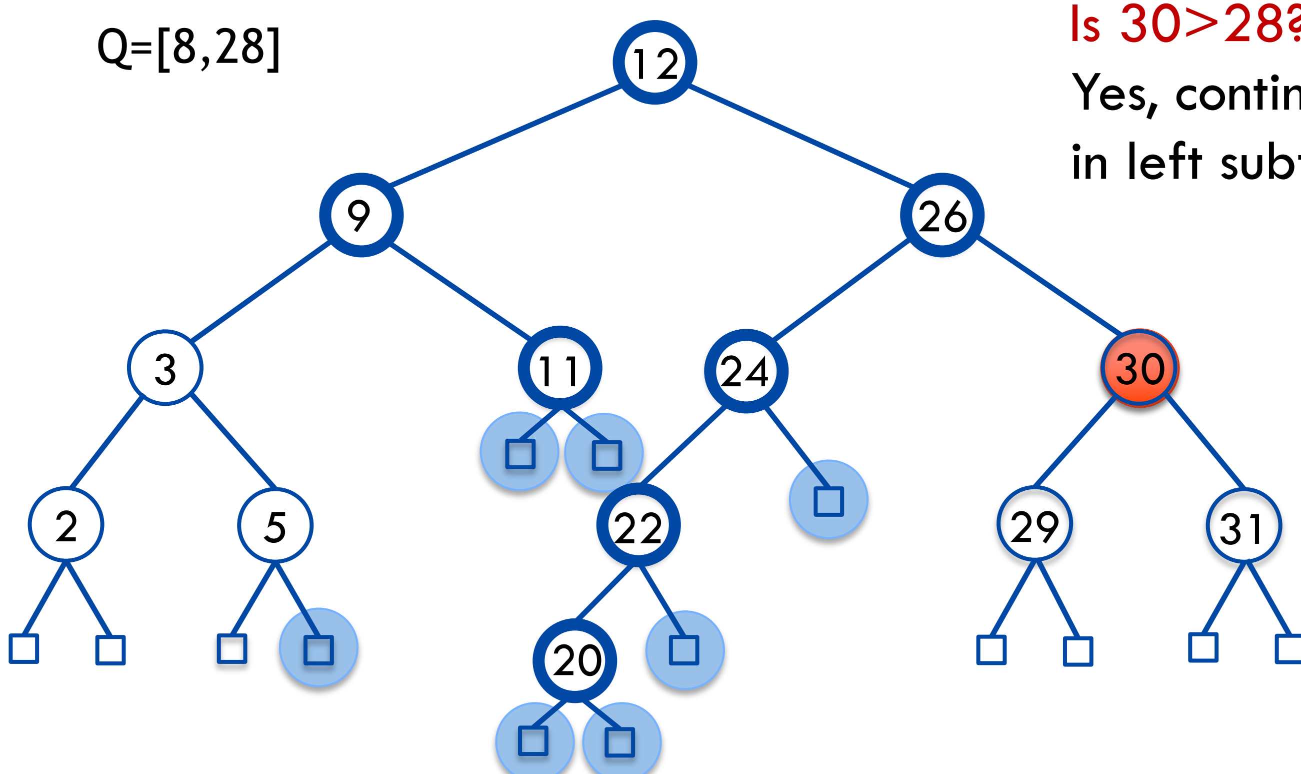
Yes, return ∅.



Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



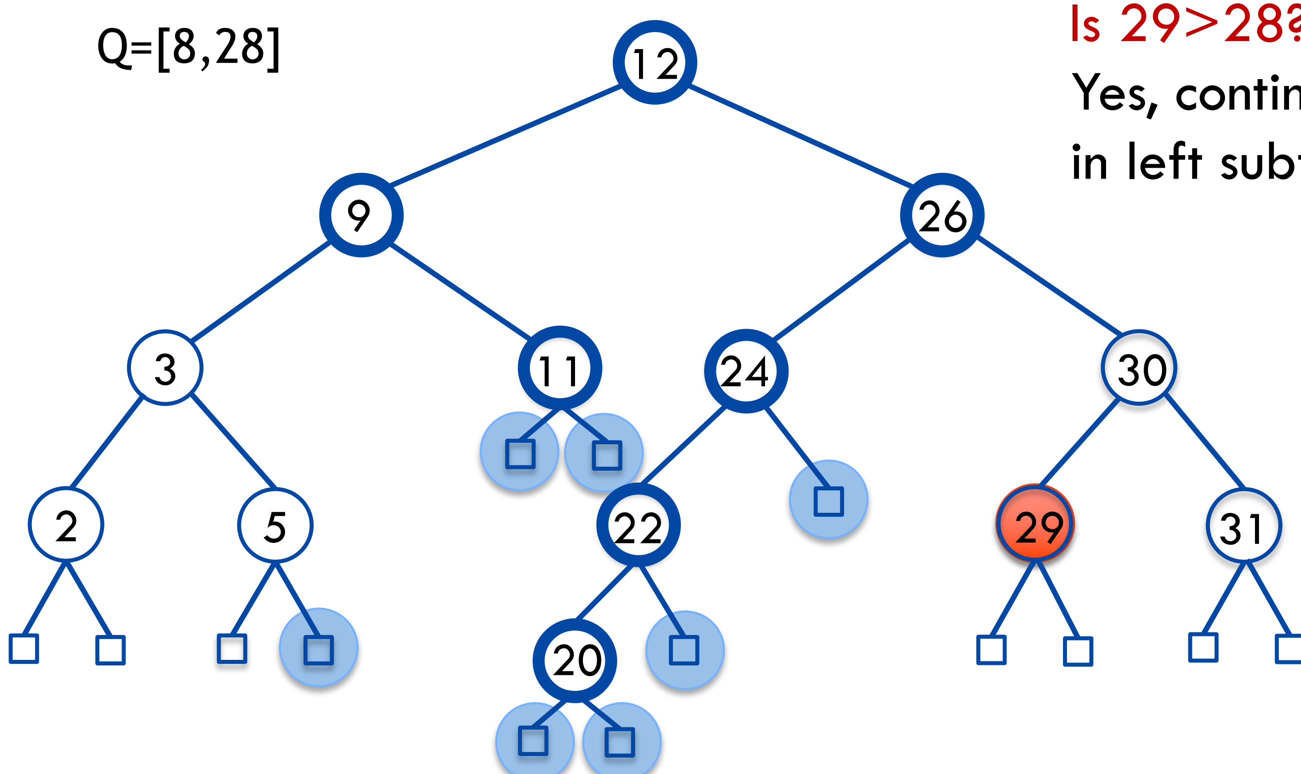
Is 30>28?

Yes, continue search in left subtree.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



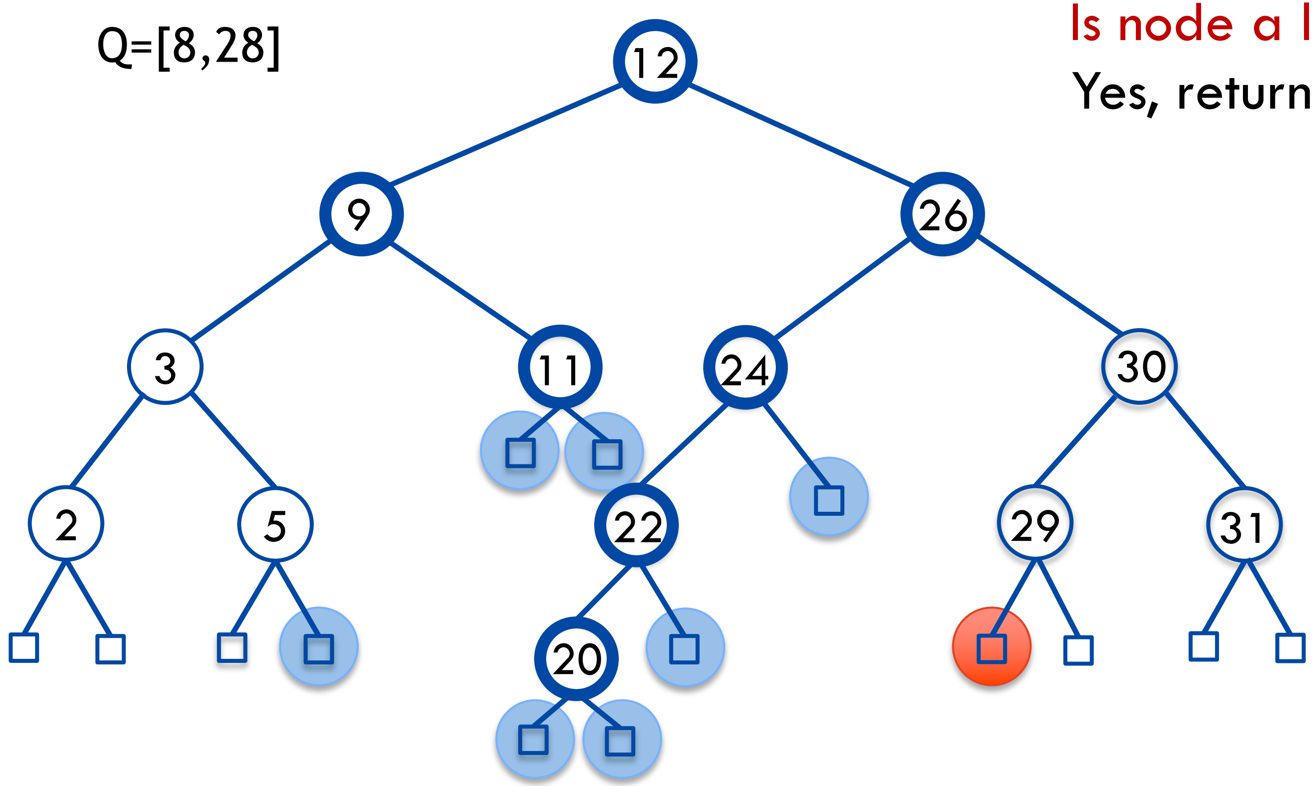
Is 29>28?

Yes, continue search in left subtree.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



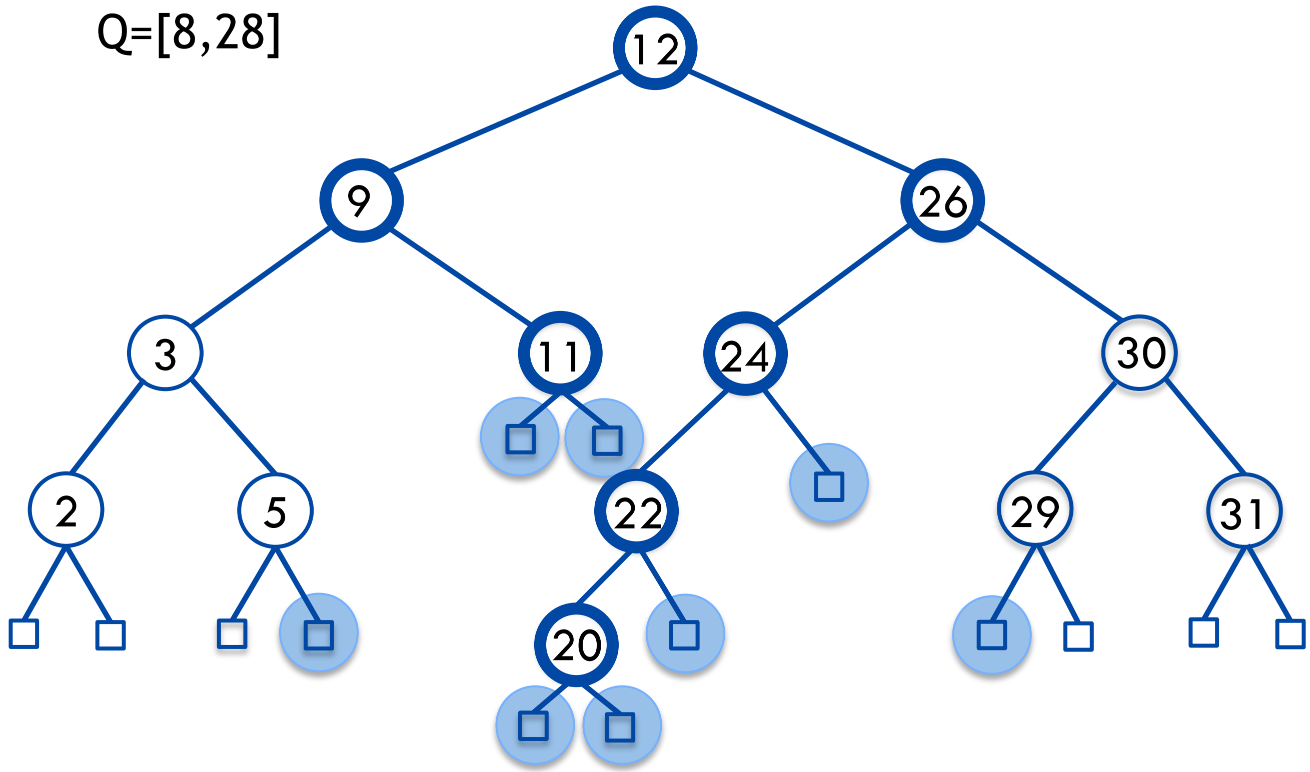
Is node a leaf?

Yes, return ∅.

Range queries

S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Q=[8,28]



Performance

Let P_1 and P_2 be the binary search paths to k_1 and k_2
We say a node v is a:

- boundary node if v in P_1 or P_2
- inside node if key(v) in $[k_1, k_2]$ but not in P_1 or P_2
- outside node if key(v) not in $[k_1, k_2]$ but not in P_1 or P_2

The algorithm only visits boundary and inside nodes and

- |inside nodes| ≤ |output|
- |boundary node| ≤ 2 * tree height

Therefore, since we only spend O(1) time per node we visit. The total running time of range search is O(|output| + tree height)

Maintaining a balanced BST

We have seen operations on BSTs that take O(height) time to run. Unfortunately, the standard insertion implementation can lead to a tree with height n-1 (e.g., if we insert in sorted order)

In the rest of today's lecture we will cover must more sophisticated algorithms that maintain a BST with height O(log n) at all times by rebalancing the tree with simple local transformations

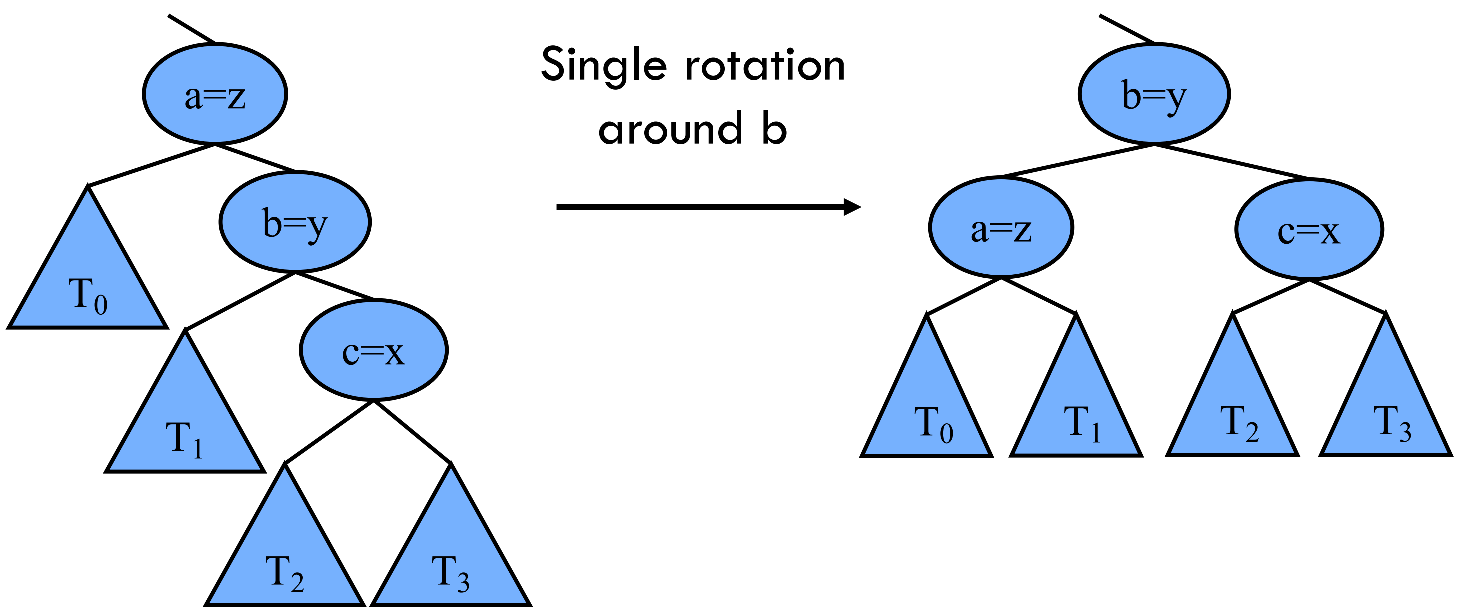
Directly translates into O(log n) performance for searching

Improving Balance: Trinode Restructuring

Let x, y, z be nodes such that x is child of y and y is a child of z.

Let a, b, c be the inorder listing of x, y, z

Perform the rotations so as to make b the topmost node of the three.

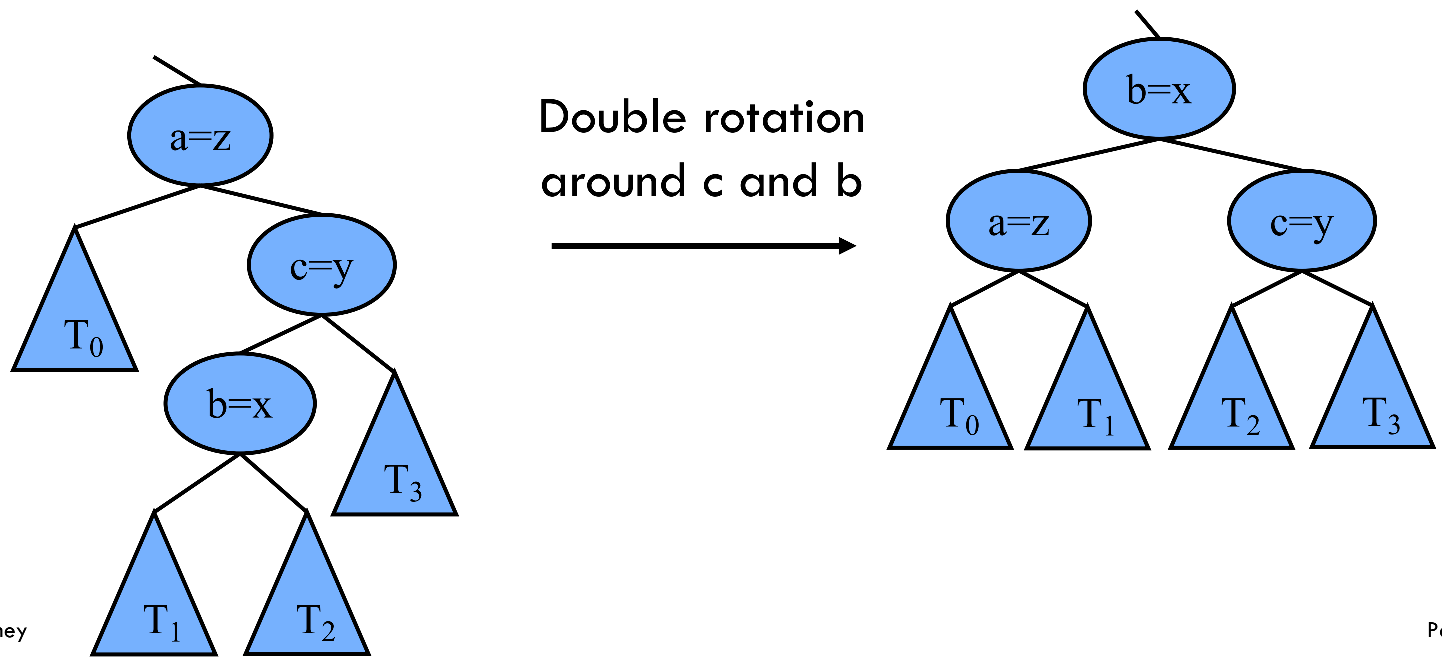


Improving Balance: Trinode Restructuring

Let x, y, z be nodes such that x is child of y and y is a child of z.

Let a, b, c be the inorder listing of x, y, z

Perform rotation so as to make *b* the topmost node of the three.



Pseudo-code

The algorithm for doing a trinode restructuring, which is used, possibly repeatedly, to restore balance after an insertion or deletion.

Algorithm restructure(x):

Input: A node x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z that are not rooted at x, y, or z.

2: Replace the subtree rooted at z with a new subtree rooted at b.

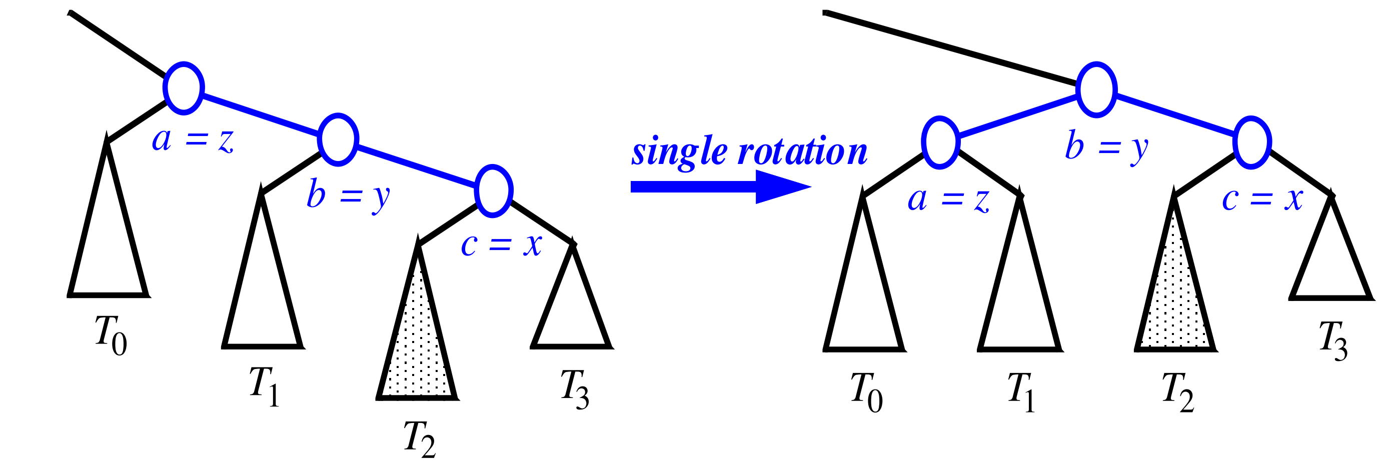
3: Let a be the left child of b and let T_0 and T_1 be the left and right subtrees of a, respectively.

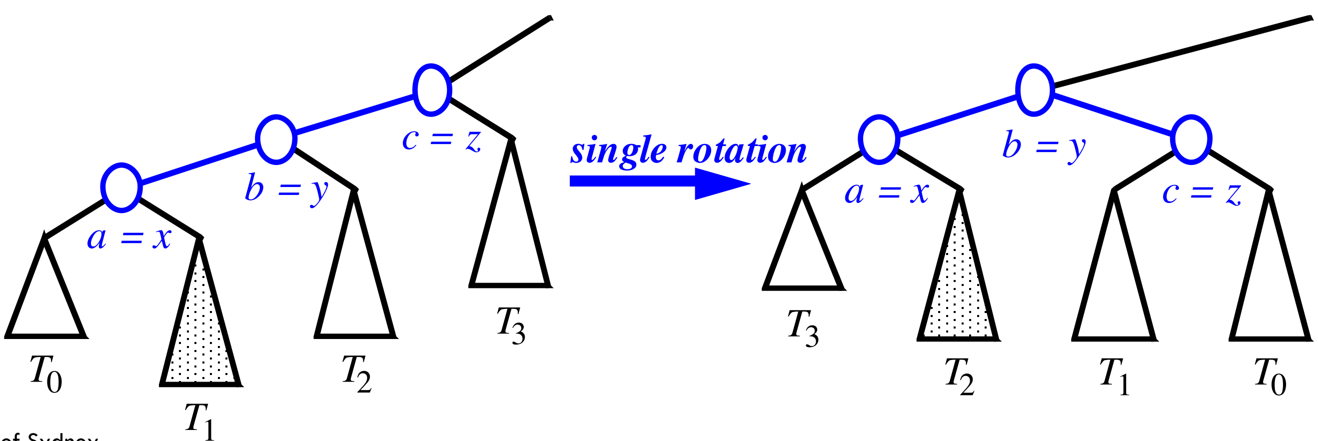
4: Let c be the right child of b and let T_2 and T_3 be the left and right subtrees of c, respectively.

5: Recalculate the heights of a, b, and c, (or a "standin" function for height), from the corresponding values stored at their children, and return b.

Trinode Restructuring (when done by Single Rotation)

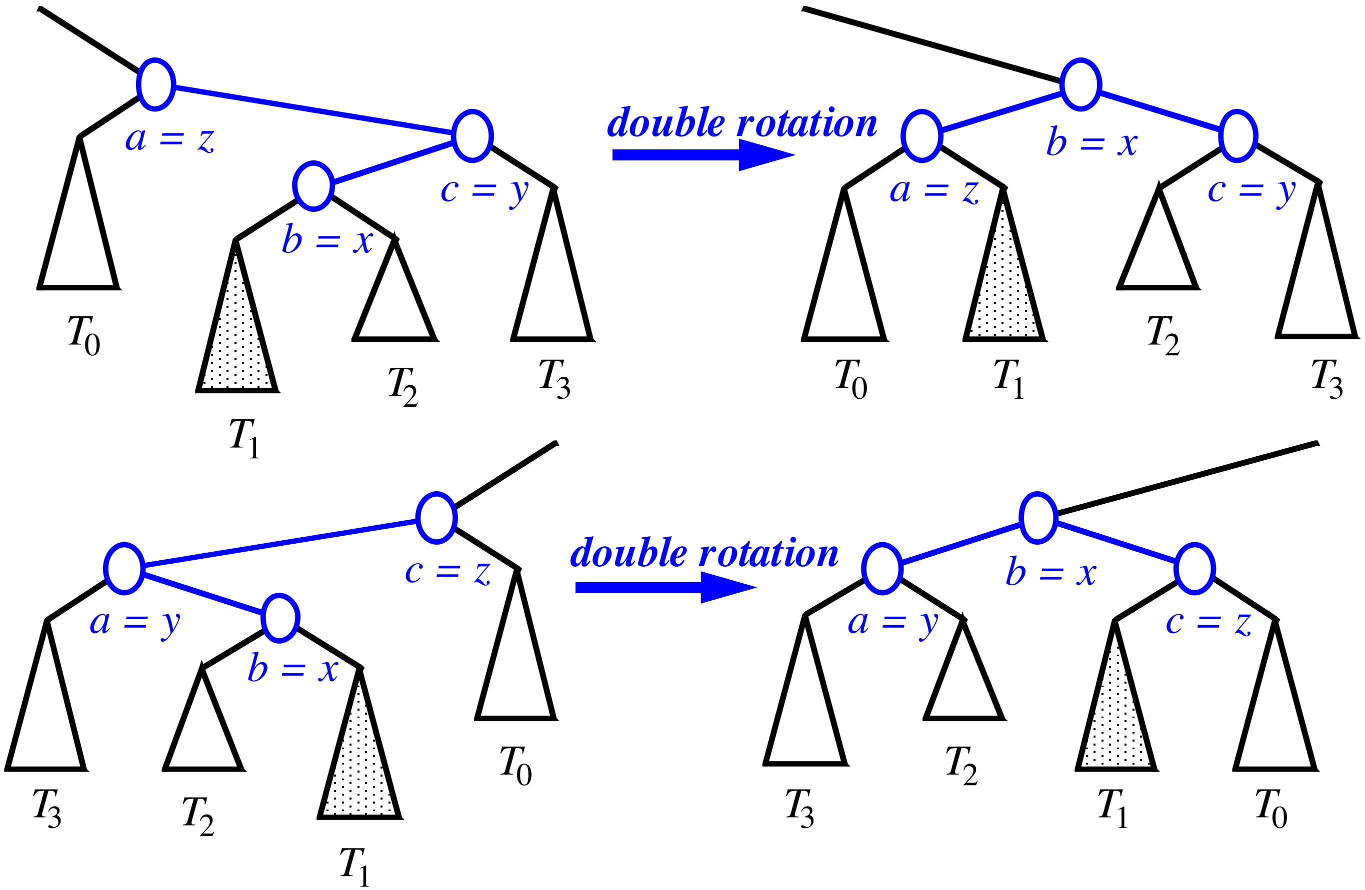
Single Rotations:





Trinode Restructuring (when done by Double Rotation)

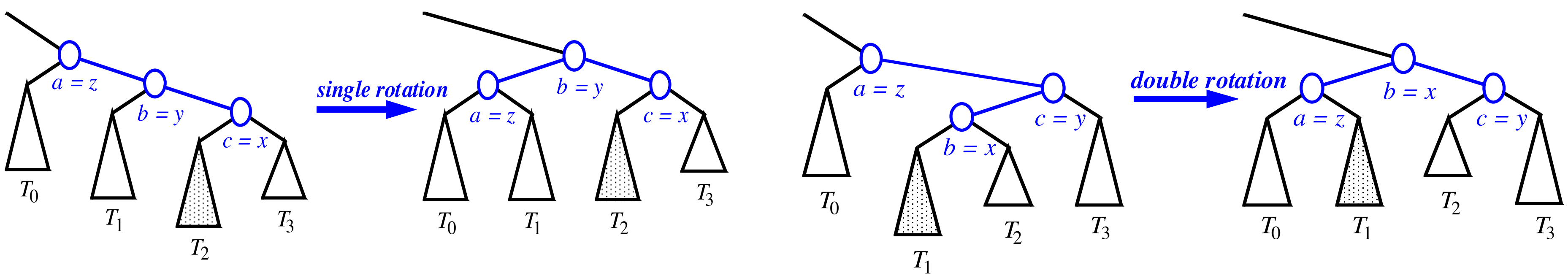
Double rotations:



Performance

Assume we are given a reference to the node x where we are performing a trinode restructure and that the binary search tree is represented using nodes and pointers to parent, left and right children

A single or double rotation takes O(1) time, because it involves updating O(1) pointers.



Rank-balanced Trees

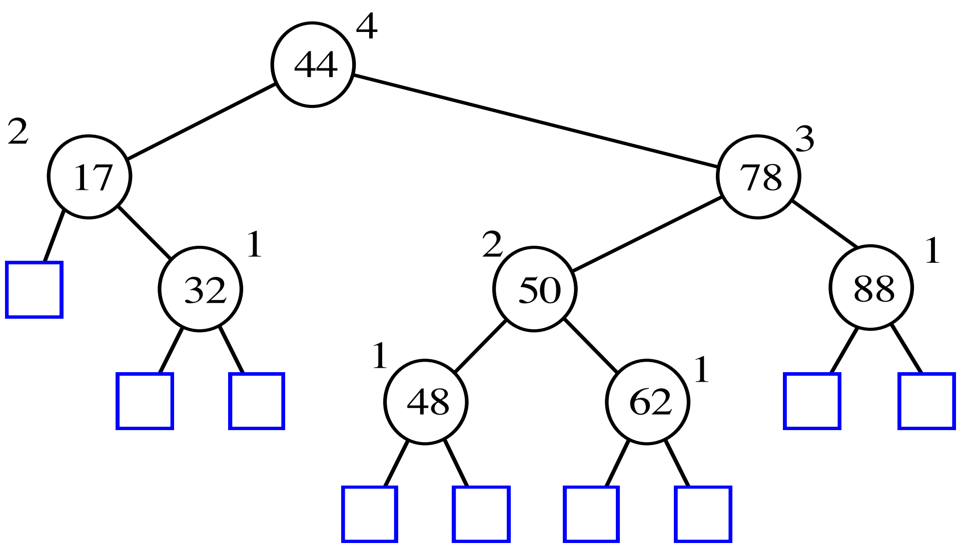
A family of balance BST implementations that use the idea of keeping a "rank" for every node, where $r(v)$ acts as a proxy measure of the size of the subtree rooted at v

Rank-balanced trees aim to reduce the discrepancy between the ranks of the left and right subtrees:

- AVL Trees (now)
- Red-Black Trees (adv)

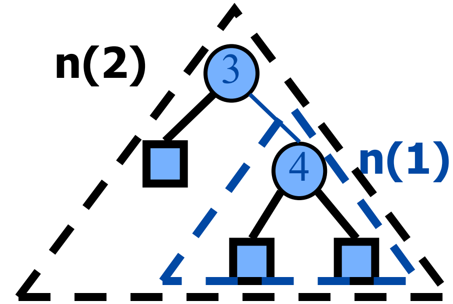
AVL Tree Definition

AVL trees are rank-balanced trees, where r(v) is its height of the subtree rooted at v



Balance constraint: The ranks of the the two children of every internal node differ by at most 1.

Height of an AVL Tree



Fact: The height of an AVL tree storing n keys is $O(\log n)$.

Proof (by induction):

- Let $N(h)$ be the minimum number of keys of an AVL tree of height h.
- We easily see that $N(1) = 1$ and $N(2) = 2$
- Clearly $N(h) > N(h-1)$ for any $h \geq 2$
- For $h > 2$, the AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and another of height $h-2$:

$$N(h) = 1 + N(h-1) + N(h-2) > 2\, N(h-2)$$

- By induction we can show that for h even

$$N(h) \geq 2^{h/2}$$

- Taking logarithms: $h < 2 \log N(h)$
- Thus the height of an AVL tree is $O(\log n)$

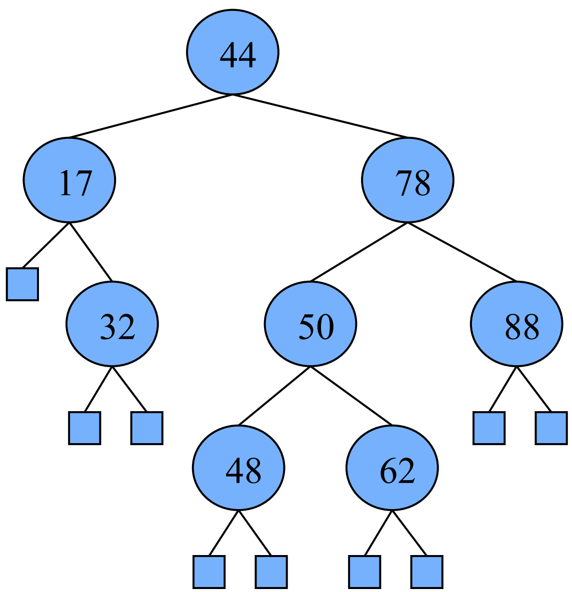
Insertion in AVL trees

Suppose we are to insert a key k into our tree:

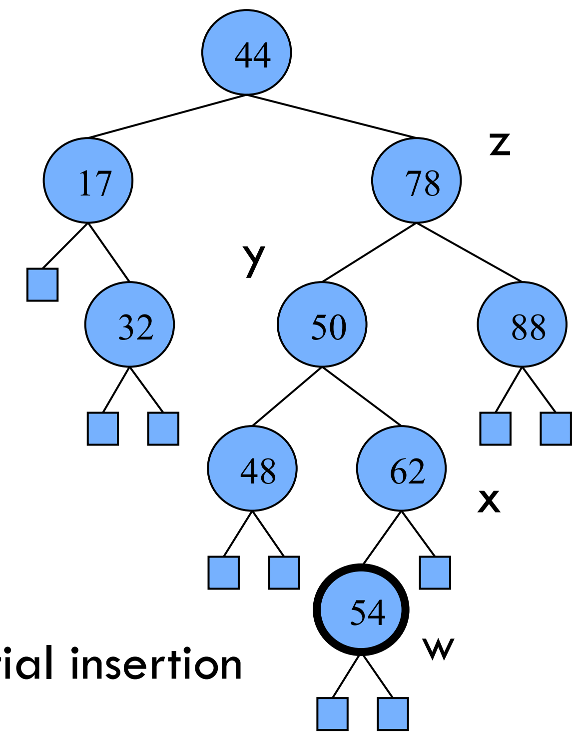
1. If k is in the tree, search for k ends at node holding k
   There is nothing to do so tree structure does not change
2. If k is not in the tree, search for k ends at external node w.
   Make this be a new internal node containing key k
3. The new tree has BST property, but it may not have AVL balance property at some ancestor of w since
   - some ancestors of w may have increased its height by 1
   - every node that is not an ancestor of w hasn't changed its heights
4. We use rotations to re-arrange tree to re-establish AVL property, while keeping BST property

Re-establishing AVL property

- Let w be location of newly inserted node
- Let z be *lowest* ancestor of w, whose children heights differ by 2
- Let y be the child of z that is ancestor of w (taller child of z)
- Let x be child of y that is ancestor of w

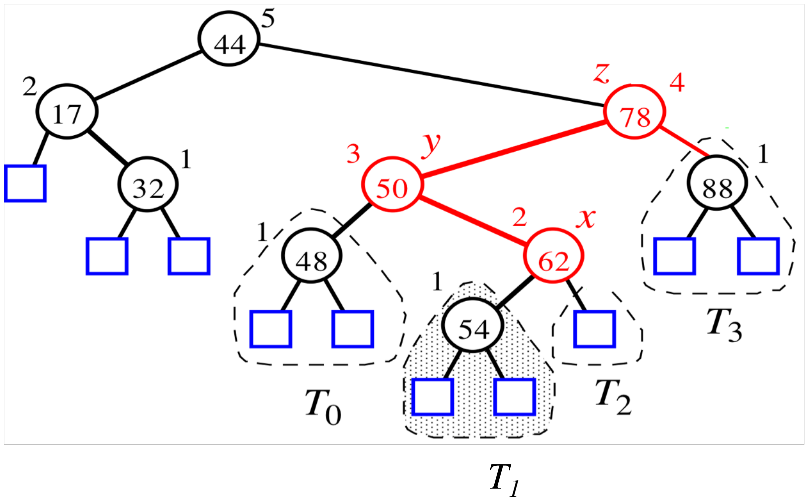


before inserting 54



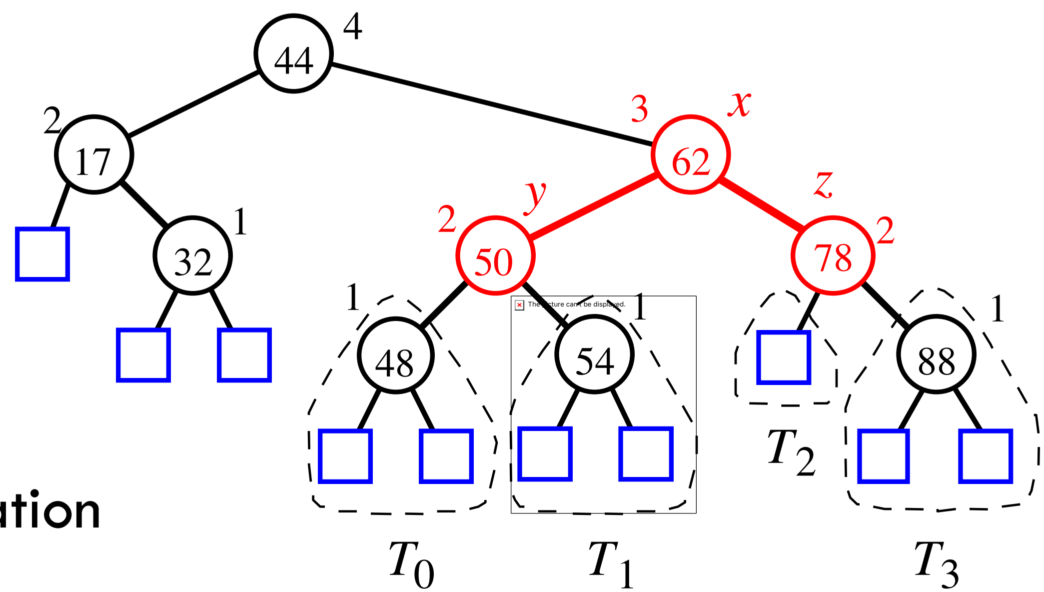
after initial insertion

Re-establishing AVL property



If tree does not have AVL property, do a trinode restructure at x, y, z

It can be argued that tree has AVL property after operation



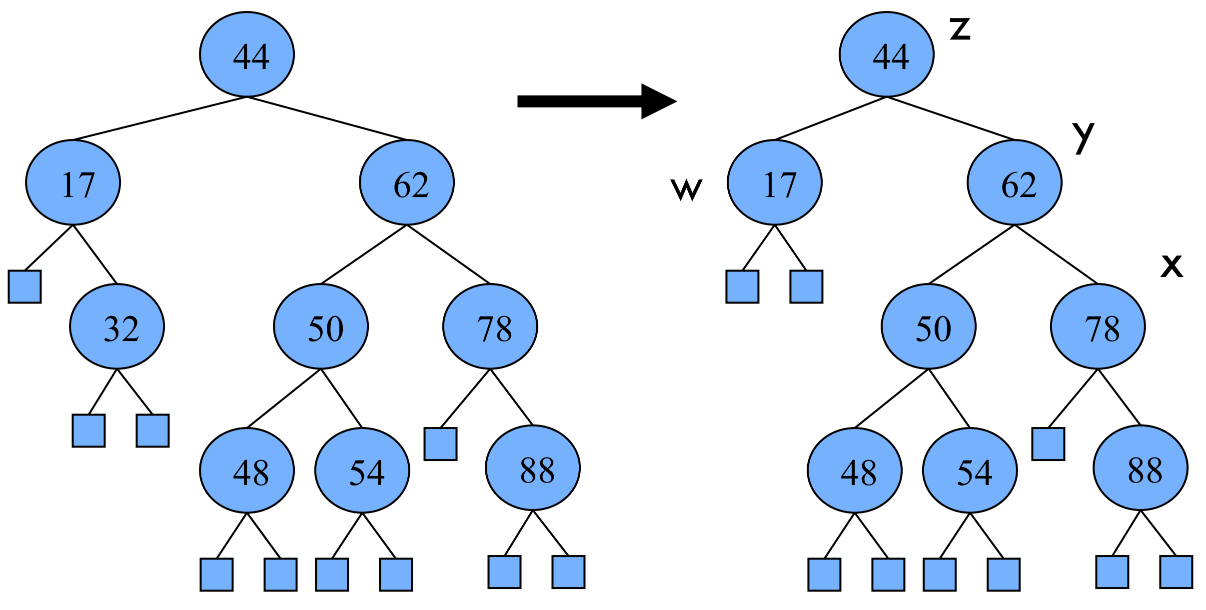
Removal in AVL trees

Suppose we are to remove a key k from our tree:

1. If k is not in the tree, search for k ends at external node
   There is nothing to do so tree structure does not change
2. If k is in the tree, search for k perform usual BST removal leading to removing a node with an external child and promoting its other child, which we call w
3. The new tree has BST property, but it may not have AVL balance property at some ancestor of w since
   - some ancestors of w may have decrease their height by 1
   - every node that is not an ancestor of w hasn't changed its heights
4. We use rotations to rearrange tree and re-establish AVL property, while keeping BST property

Re-establishing AVL property

- Let w be the parent of deleted node
- Let z be *lowest* ancestor of w, whose children heights differ by 2
- Let y be the child of z with larger height (y is not an ancestor of w)
- Let x be child of y with larger height

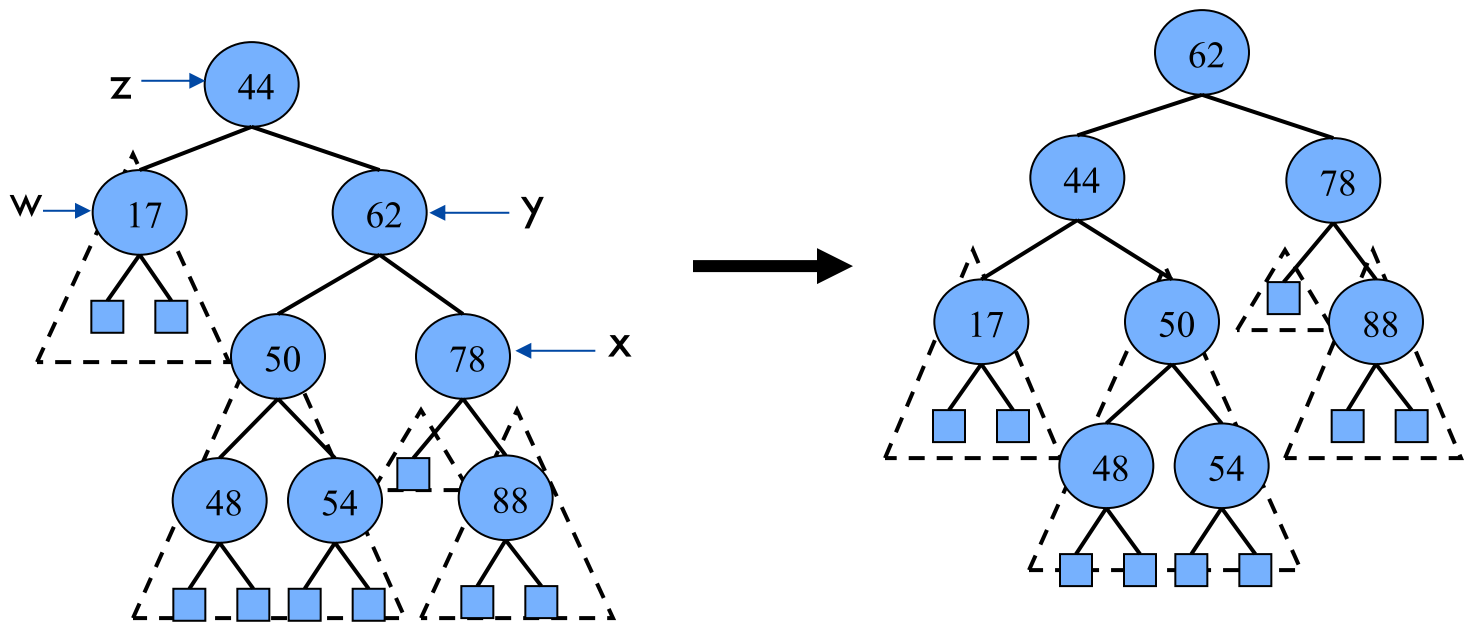


before deletion of 32

after initial deletion

Re-establishing AVL property

- If tree does not have AVL property, do a trinode restructure at x, y, z
- This restores the AVL property at z but it may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



AVL Tree Performance

Suppose we have an AVL tree storing n items then

- The data structure uses O(n) space
- Height of the tree O(log n)
- Searching takes O(log n) time
- Insertion takes O(log n) time
- Removal takes O(log n) time

Today we just saw a sketch of the how insertions and removals are performed. Working out all the details behind these operations is to heavy for the lecture, but I hope you got a flavor for what they entail and I encourage your to read the details on your own.