COMP2823

Binary Search Trees
[GT 3.1-2, 4.2, 19.6]

A/Prof Julian Mestre
School of Computer Science

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

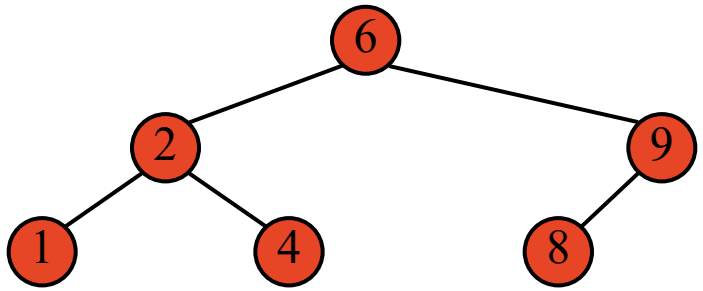
THE UNIVERSITY OF SYDNEY

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)

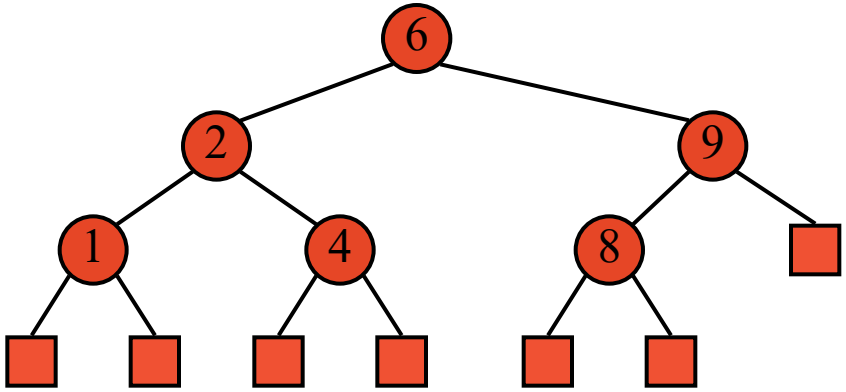
Note that an inorder traversal of a binary search tree visits the keys in increasing order.



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)



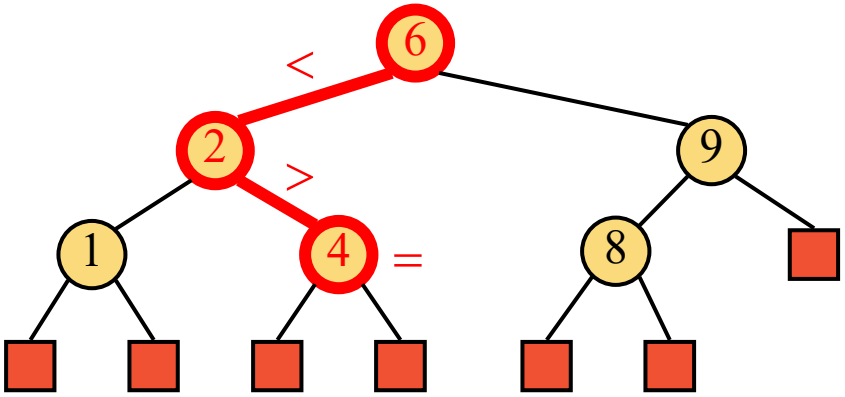
Searching with a Binary Search Tree

This should be fairly straight forward

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 key of the current node v with k

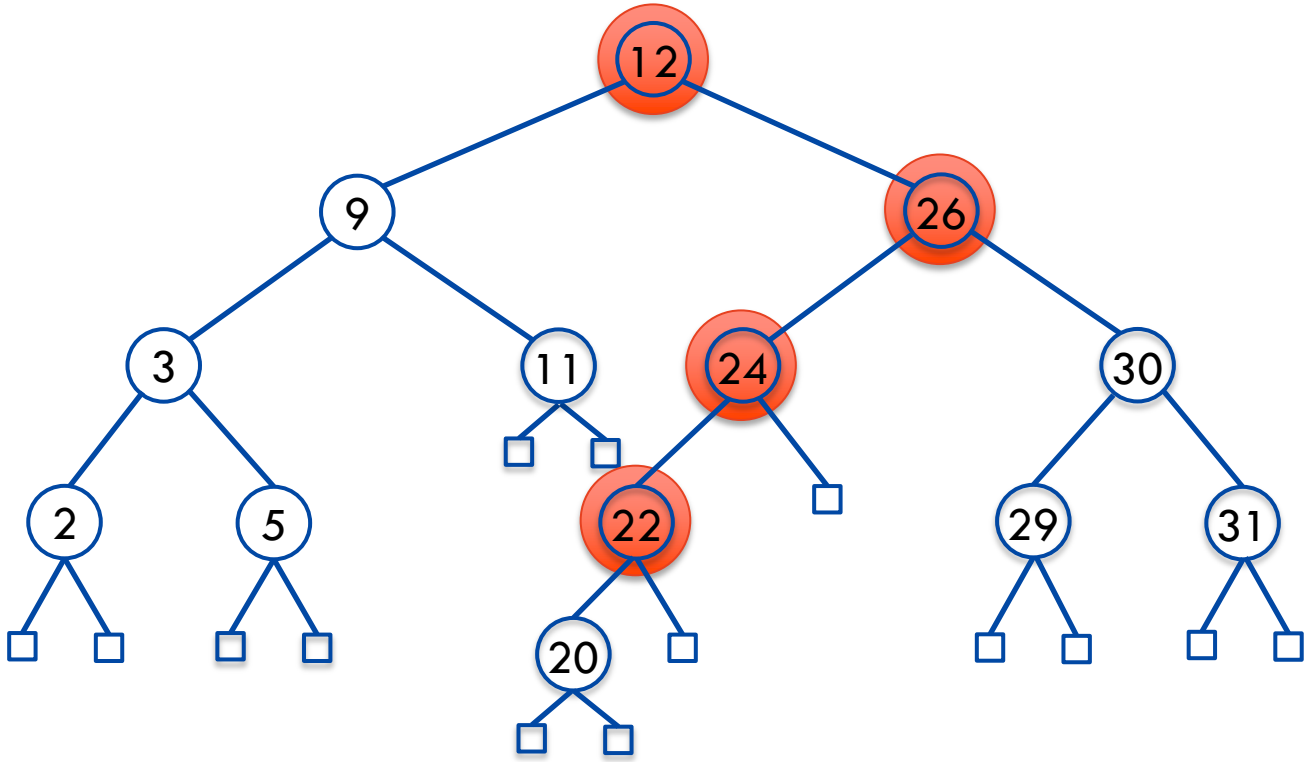
If we reach an external node, this means that the key is not in the data structure




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

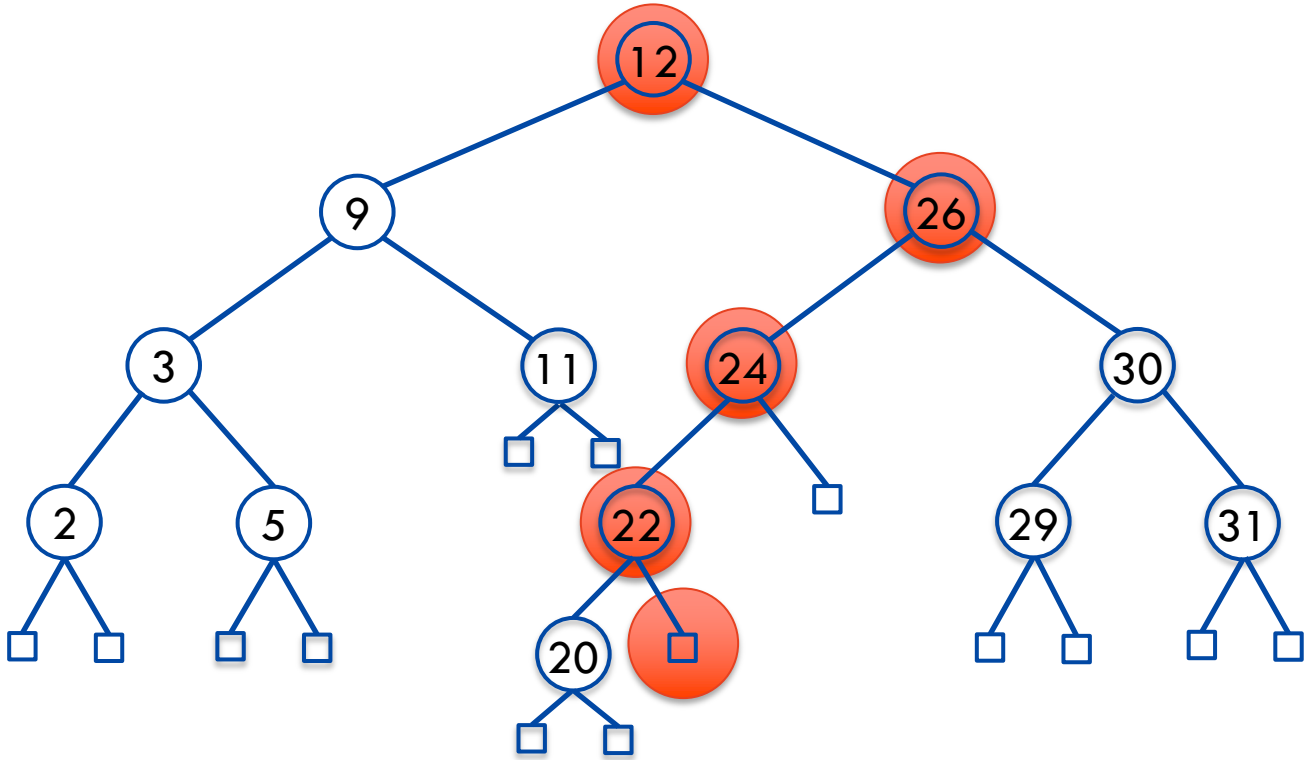
Example: Find 22

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



Example: Find 23

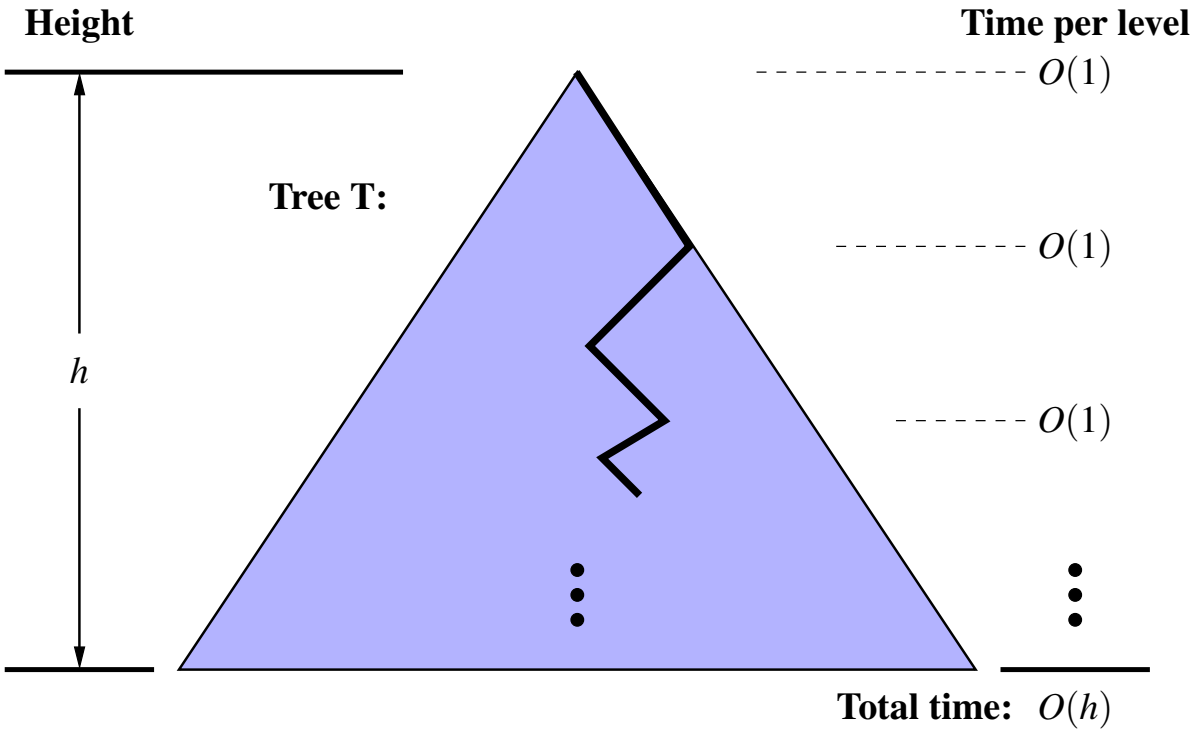
S={2,3,5,9,11,12,20,22,24,26,29,30,31}



Analysis of Binary Tree Searching

Runs in O(h) time, where h is the height of the tree

- worst case is $h = n - 1$
- best case is $h \leq \log_2 n$

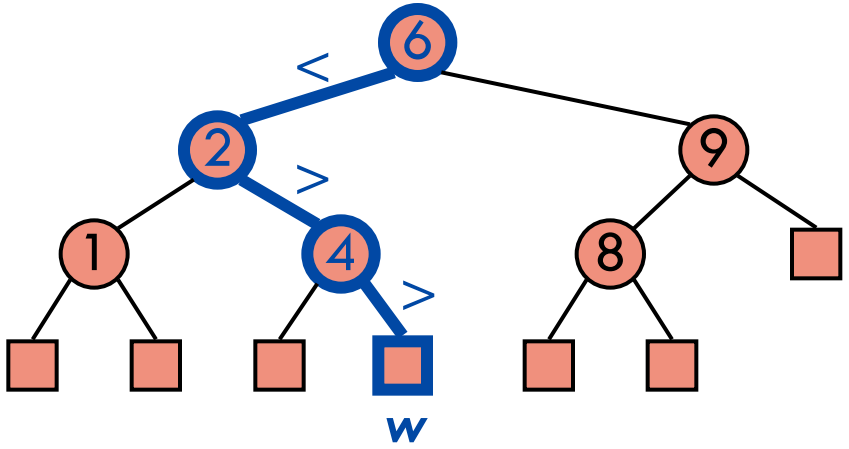


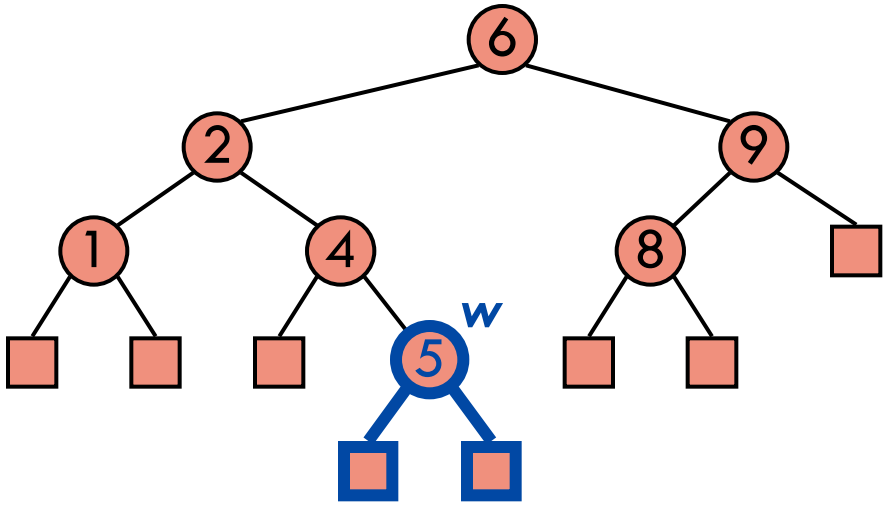
Insertion

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

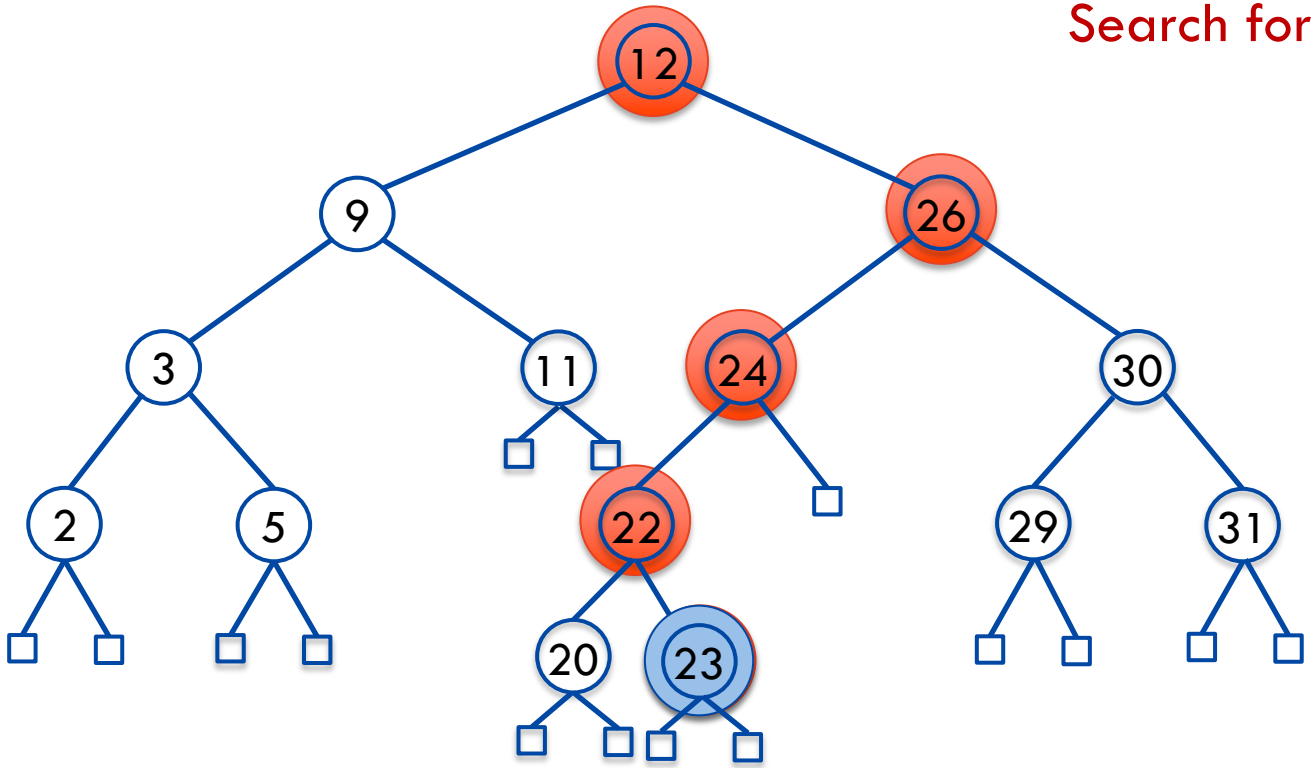
If k is not found, let w be the external node reached by the search. We replace w with an internal node holding (k, o)





Example: Insert 23

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



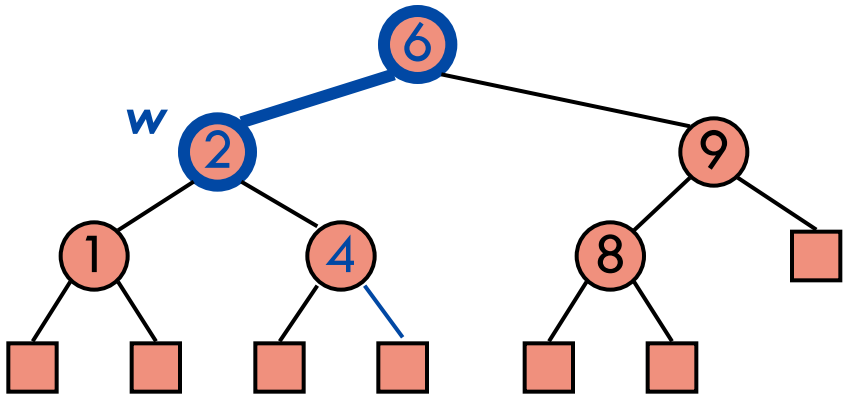
Delete

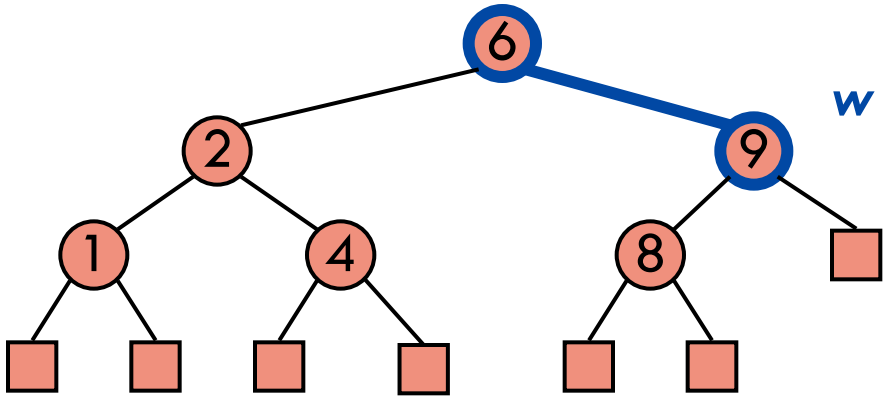
To perform operation remove(k), we search for key k (using search) to find the node w holding k

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

I found that my later thought on this is wrong:

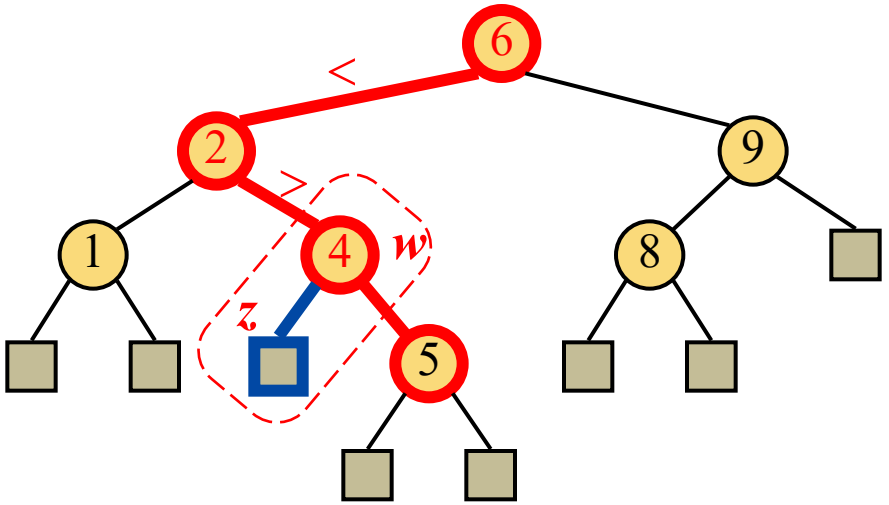
We're removing the external child of "4" w also, we're just promoting the node "5"

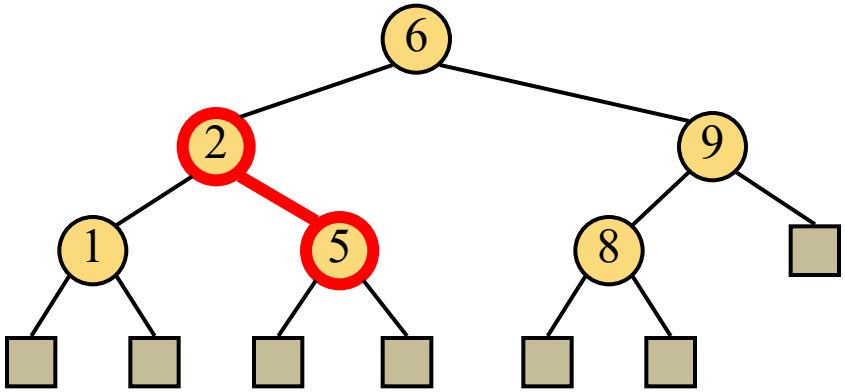
Suppose that the node w we want to remove has an external child, which we call z.

To remove w we

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

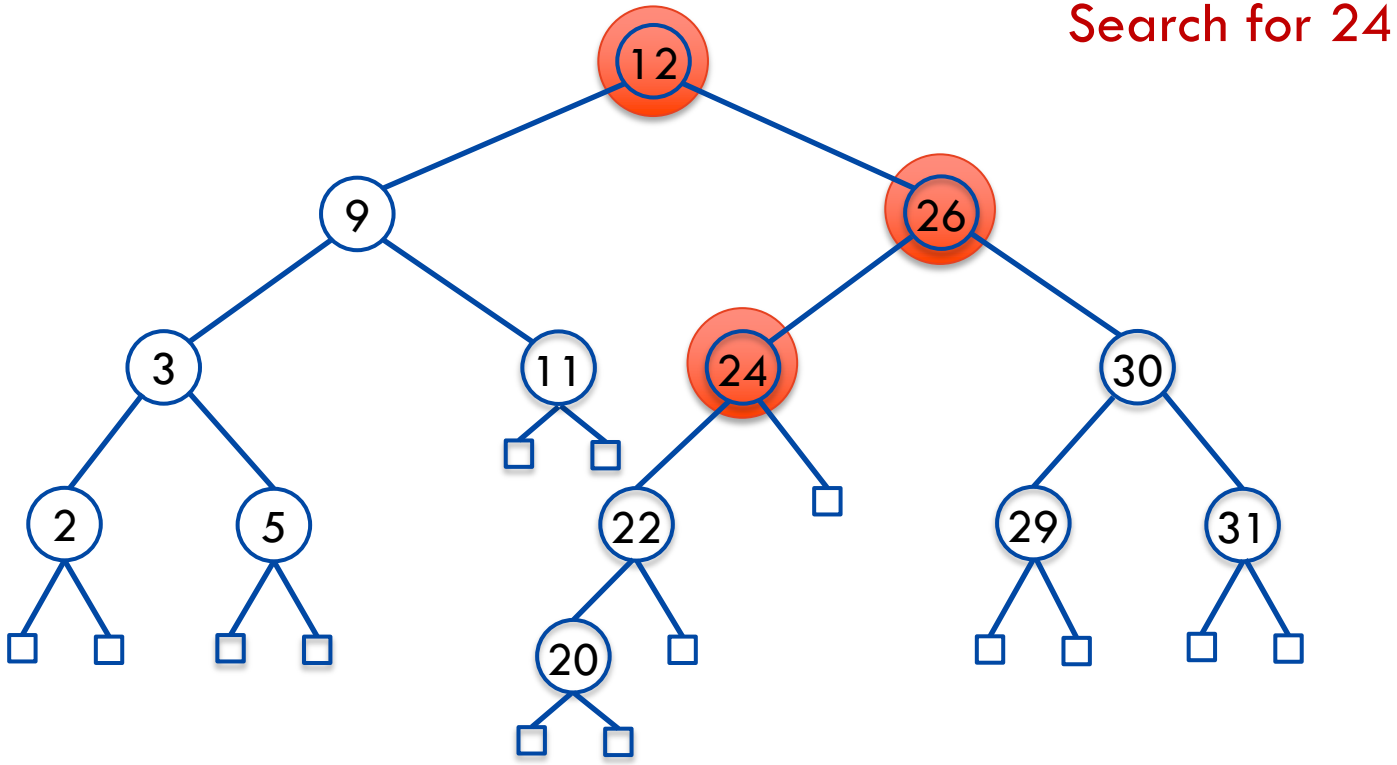
This preserves the BST property





Example: Delete 24

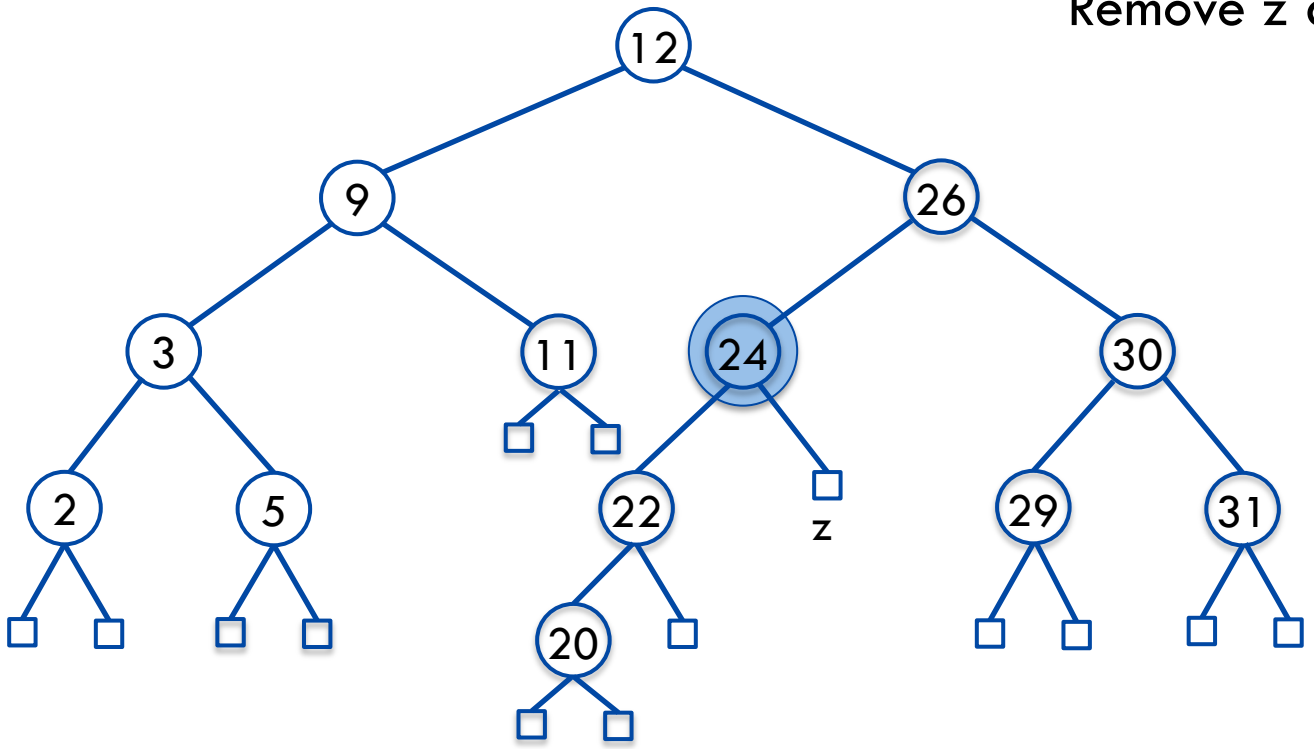
S={2,3,5,9,11,12,20,22,24,26,29,30,31}



Example: Delete 24

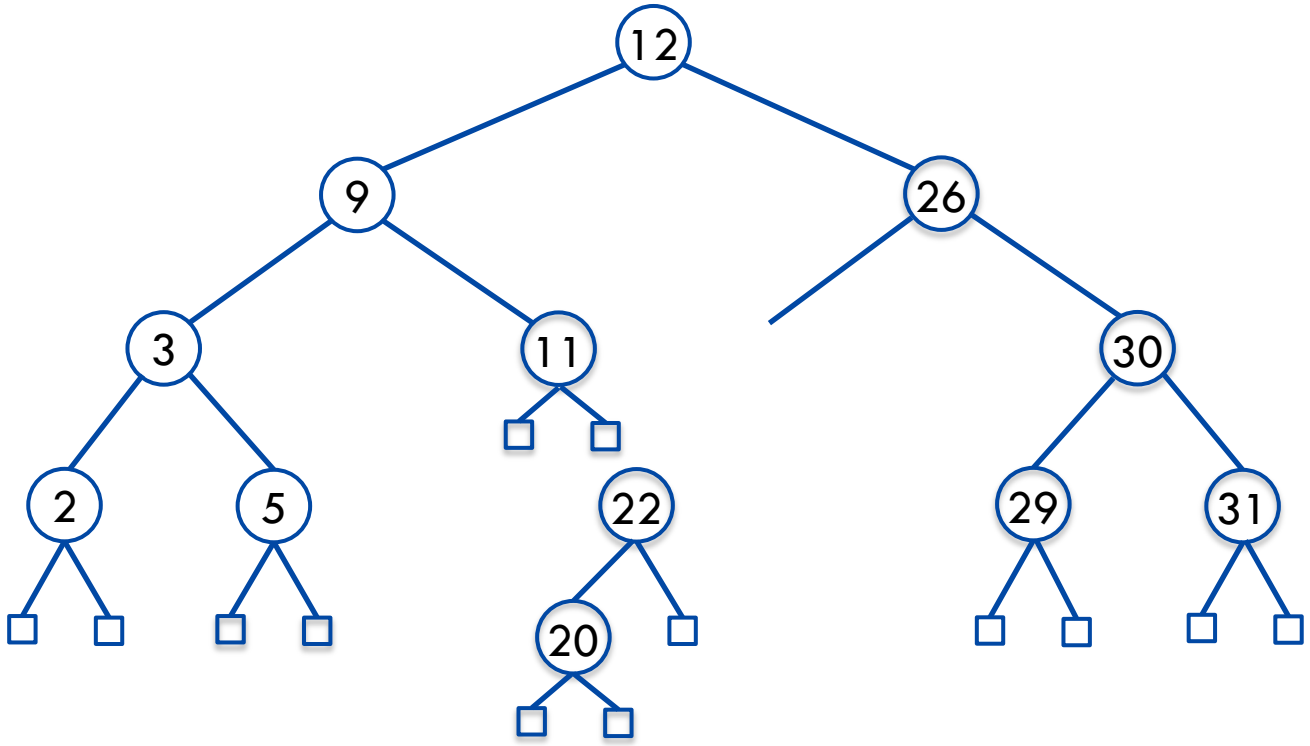
S={2,3,5,9,11,12,20,22,24,26,29,30,31}

Remove z and w



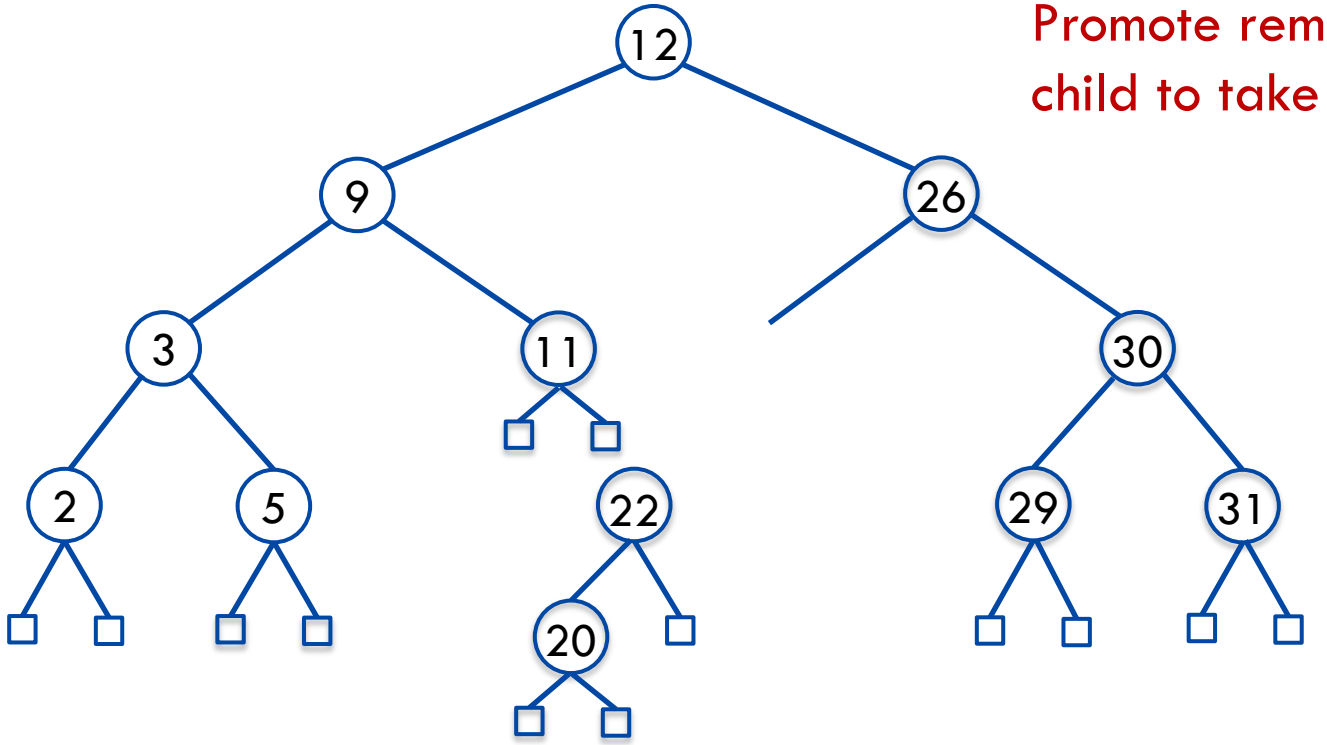
Example: Delete 24

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

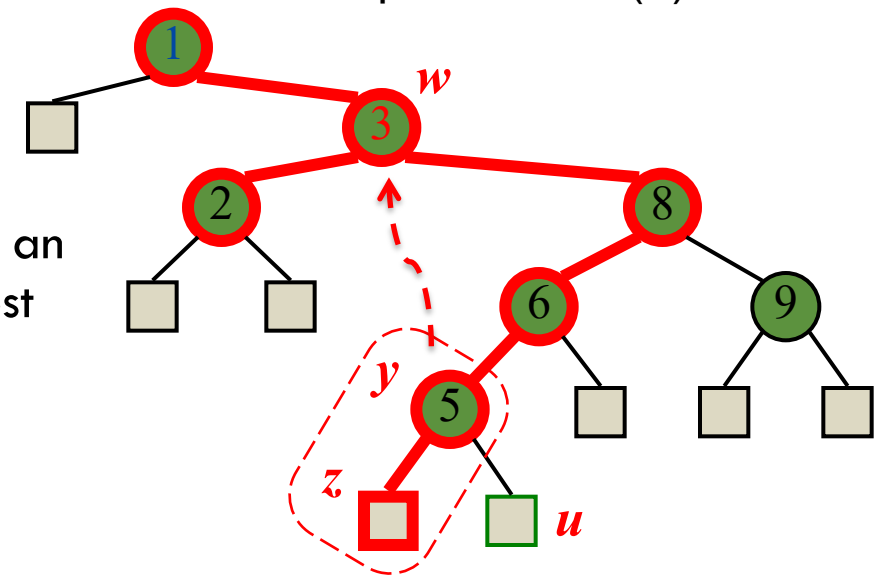
Note that naturally this can work in both ways: left and right

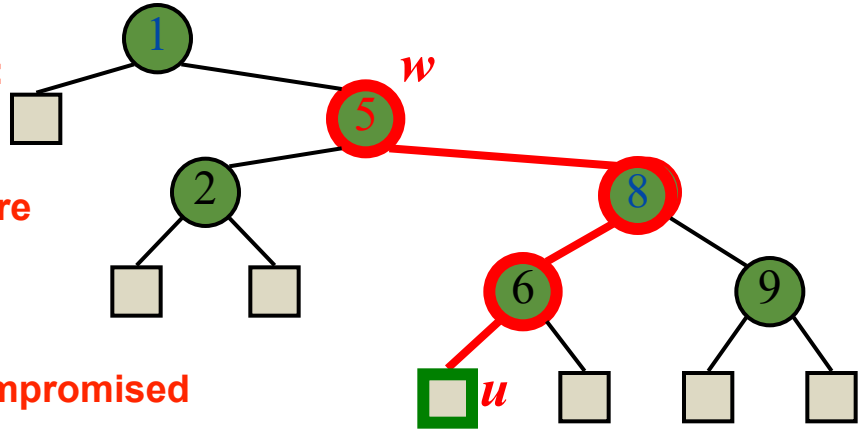
This preserves the BST property

Note also that if we have this picture, then we're forced to cut an incision in the right muscle, for the left one only has pone child.

Cutting that one will cause the artery to be compromised

Example: remove(3)





Deletion algorithm

This is a very plain text English description.

One can imagine the real code to be much more complicated

```
def remove(k)
  w ← search(k, root)
  if w.isExternal() then
    # key not found
    return null
  else if w has at least one external child z then
    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 as above
```

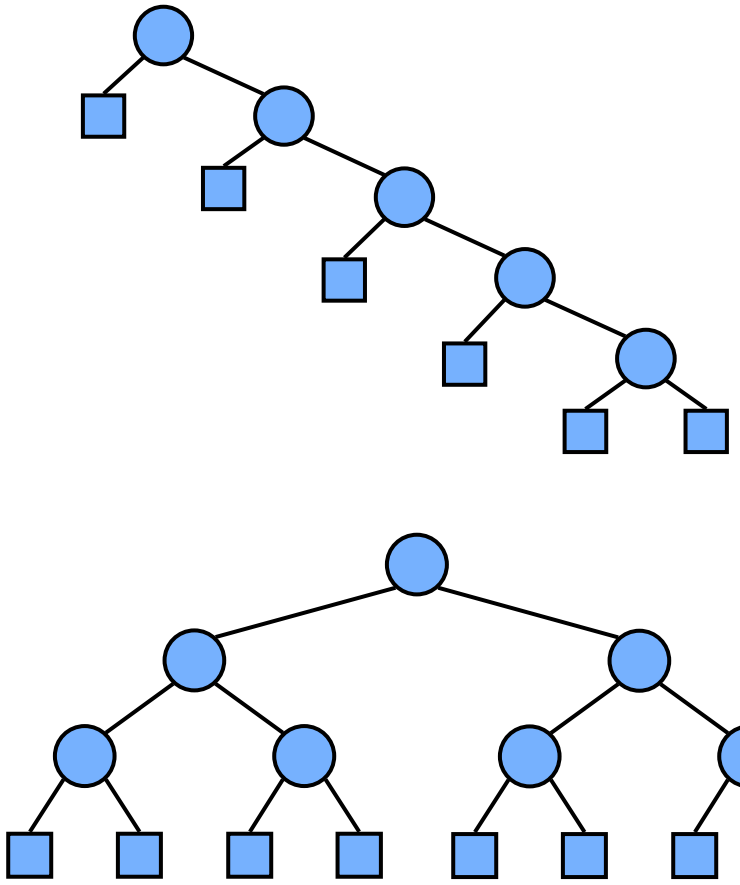
Complexity

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 descendants to be equal to the parent

  key(left descendant) ≤ key(node) < key(right descendant)
- Using a list to store duplicates

Range Queries

A 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 null
   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)
```

The most important line is this:

output.append(v)

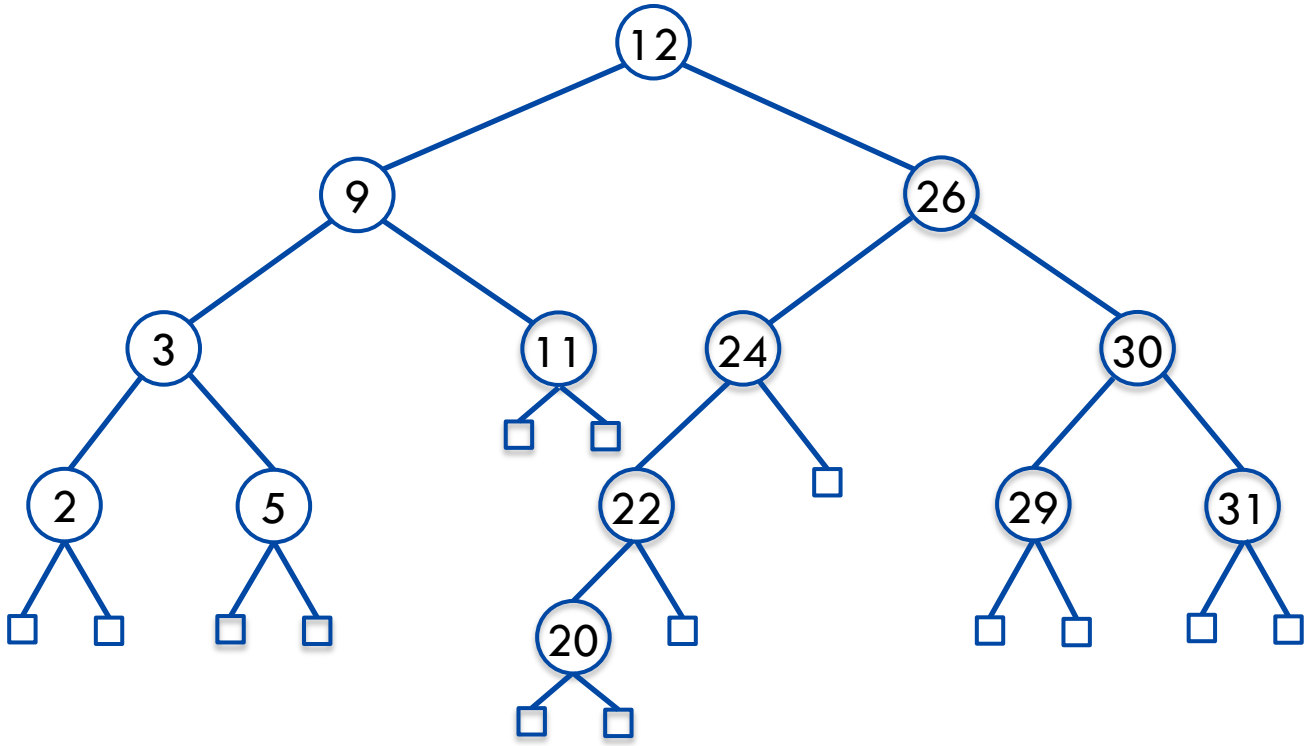
ALSO NOTICE THE ORDER OF THE THREE LINES;
RANGE(LEFT)
OUTPUT.APPEND()
RANGE(RIGHT)

Note that we also need to look out for external nodes:

If they're leaf nodes, they we need to return NULL and go up one level

Range queries

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

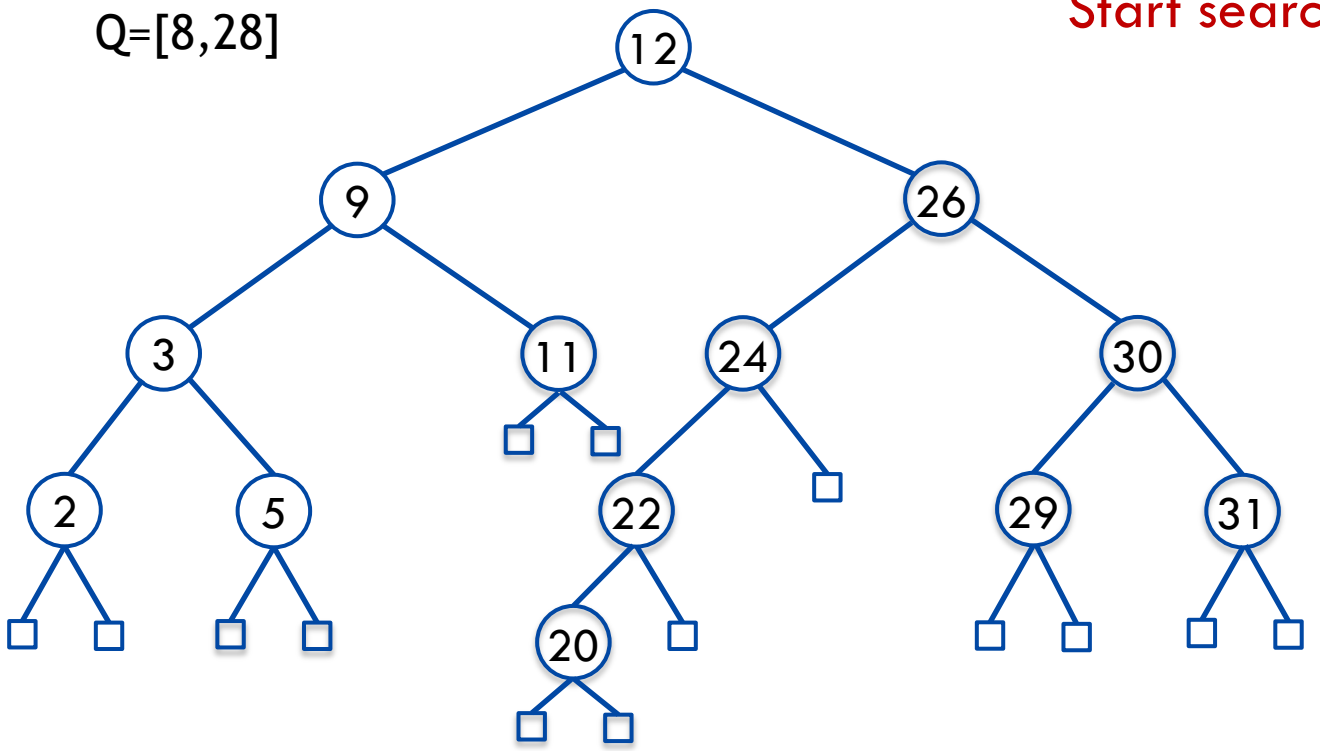


Range queries

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

Q=[8,28]

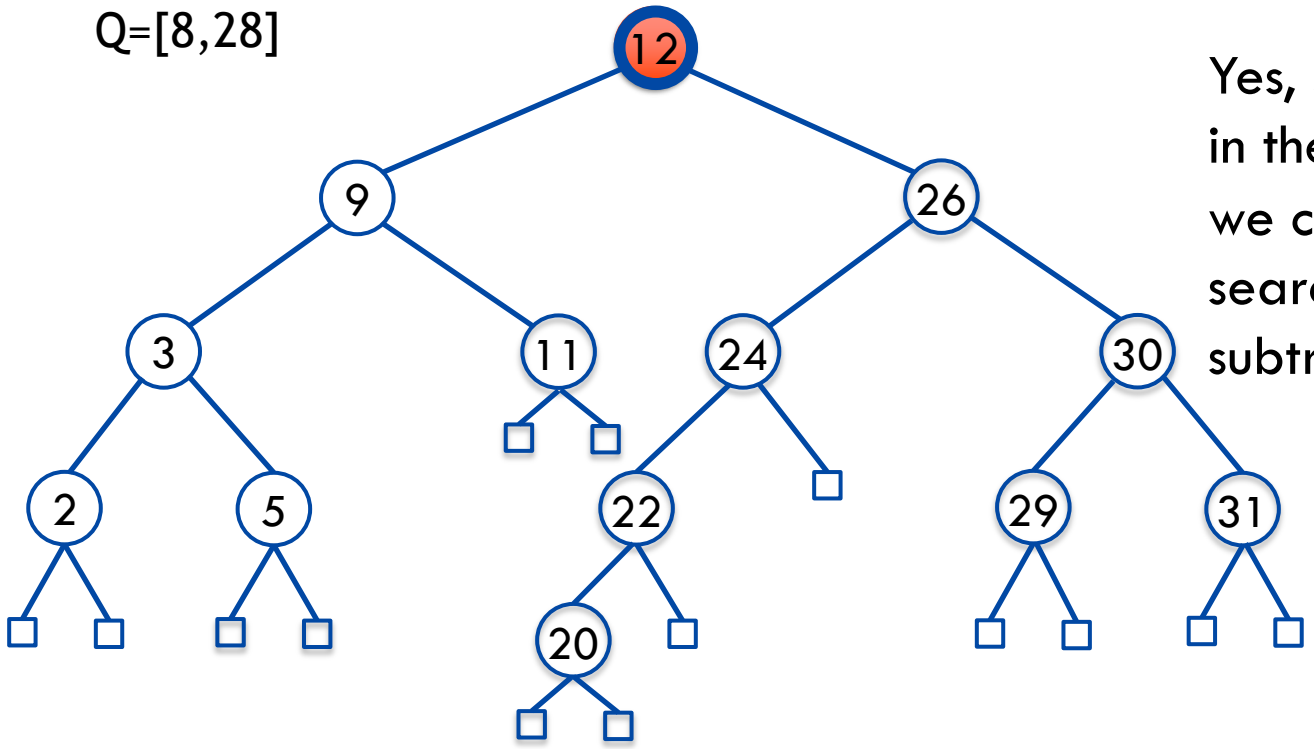
Start search



Range queries

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

Q=[8,28]



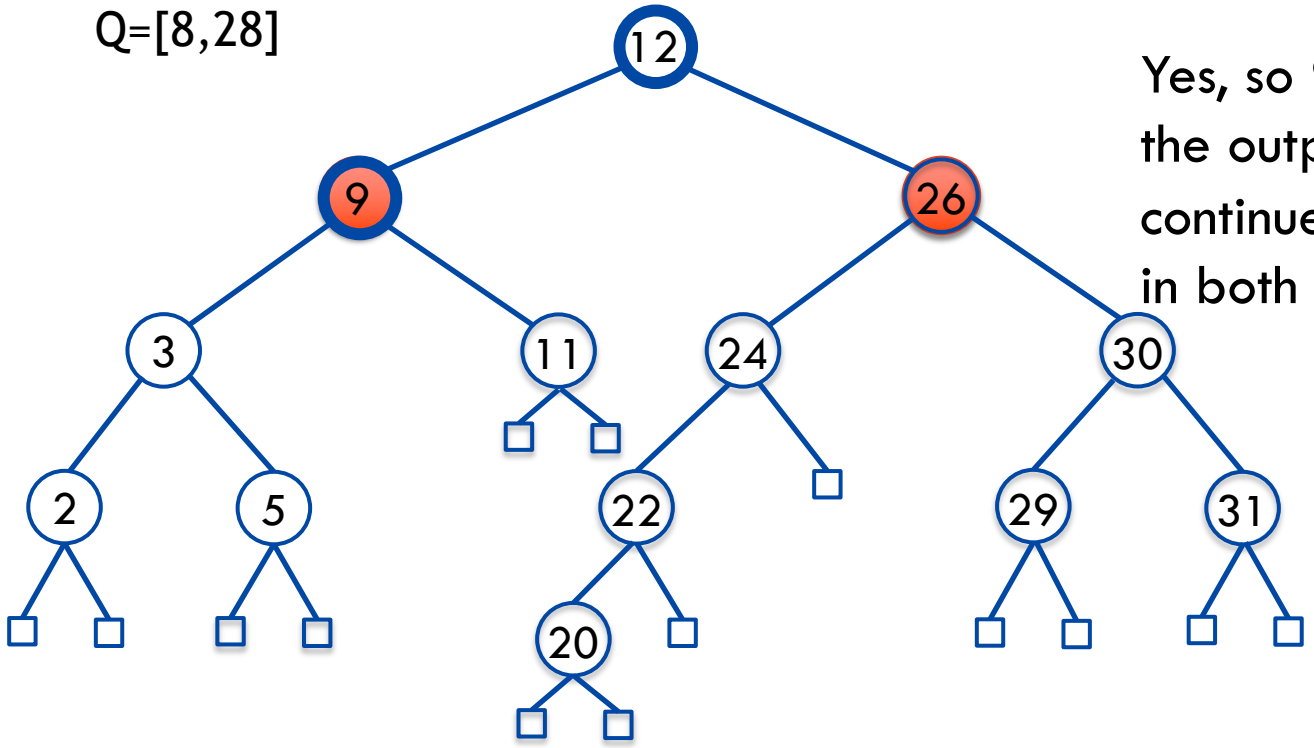
Is 12 in Q?

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

Range queries

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

Q=[8,28]



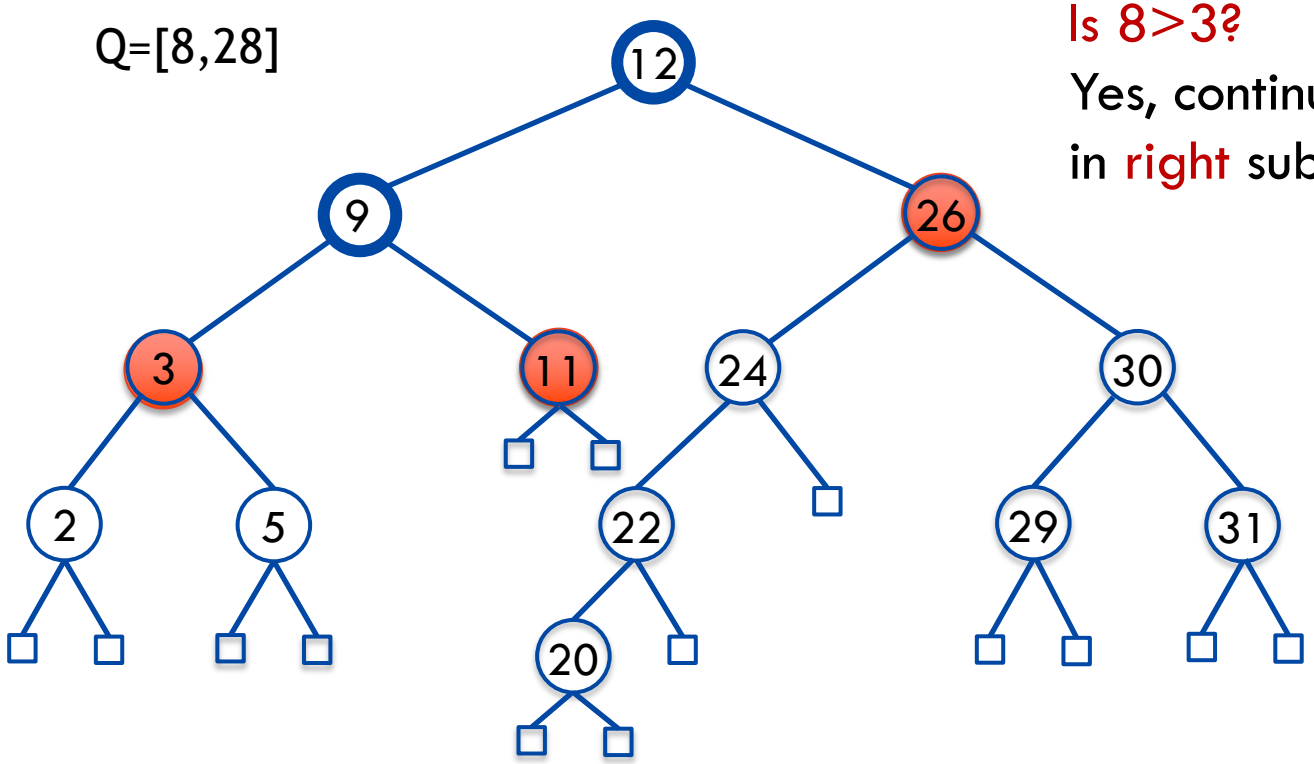
Is 9 in Q?

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

Range queries

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

Q=[8,28]



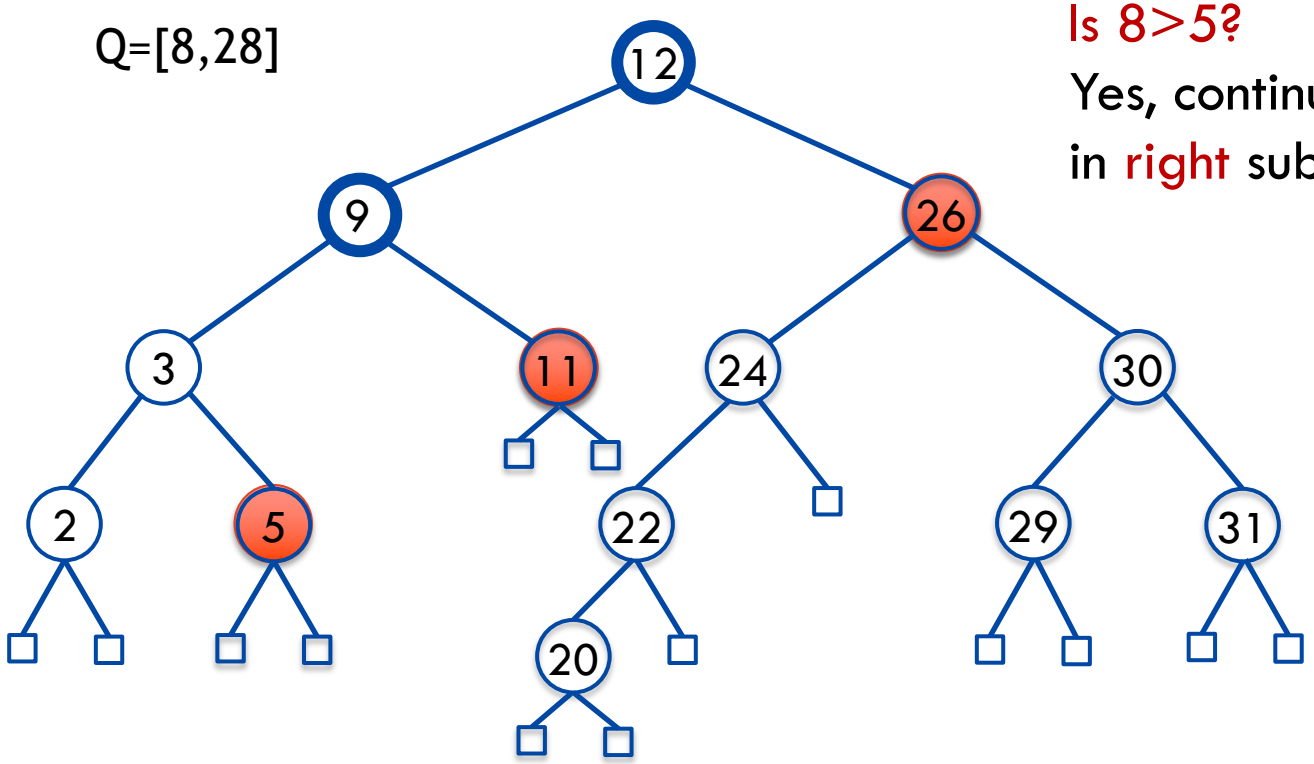
Is 8>3?

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]



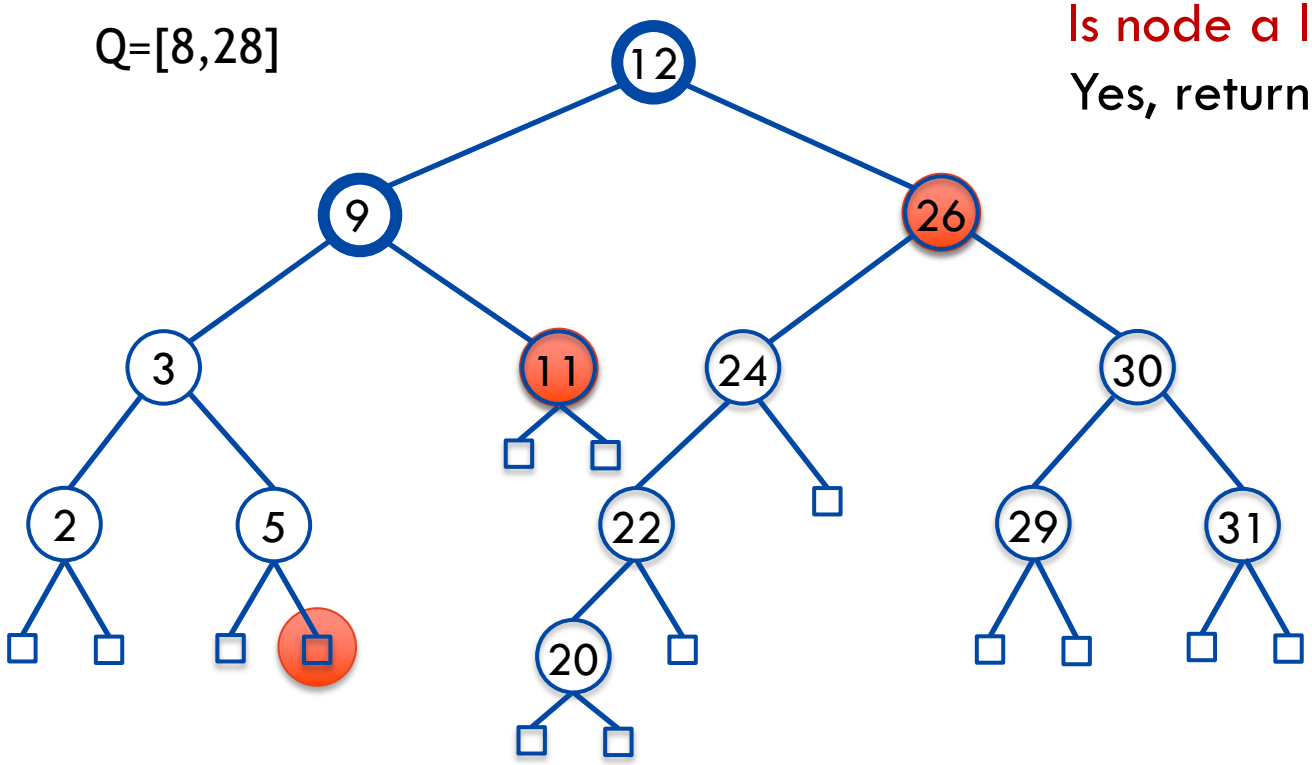
Is 8>5?

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]



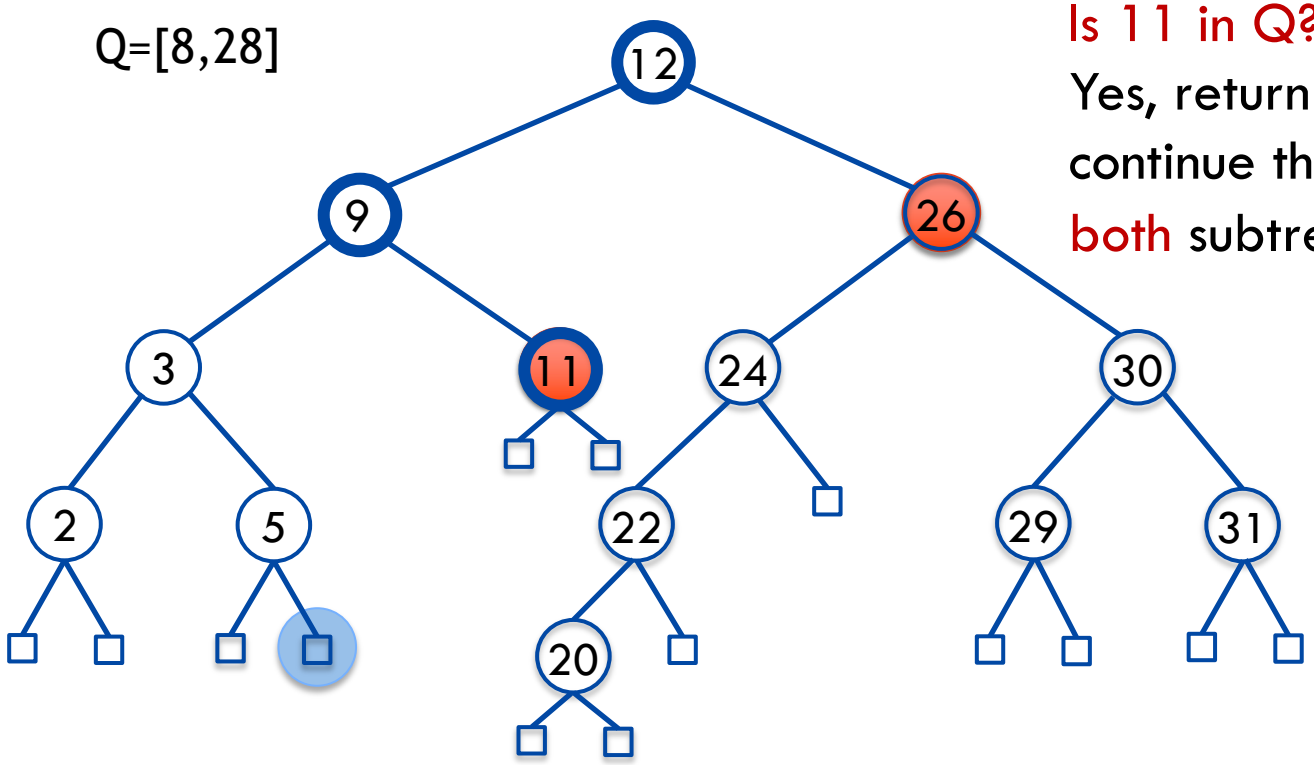
Is node a leaf?

Yes, return ∅.

Range queries

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

Q=[8,28]



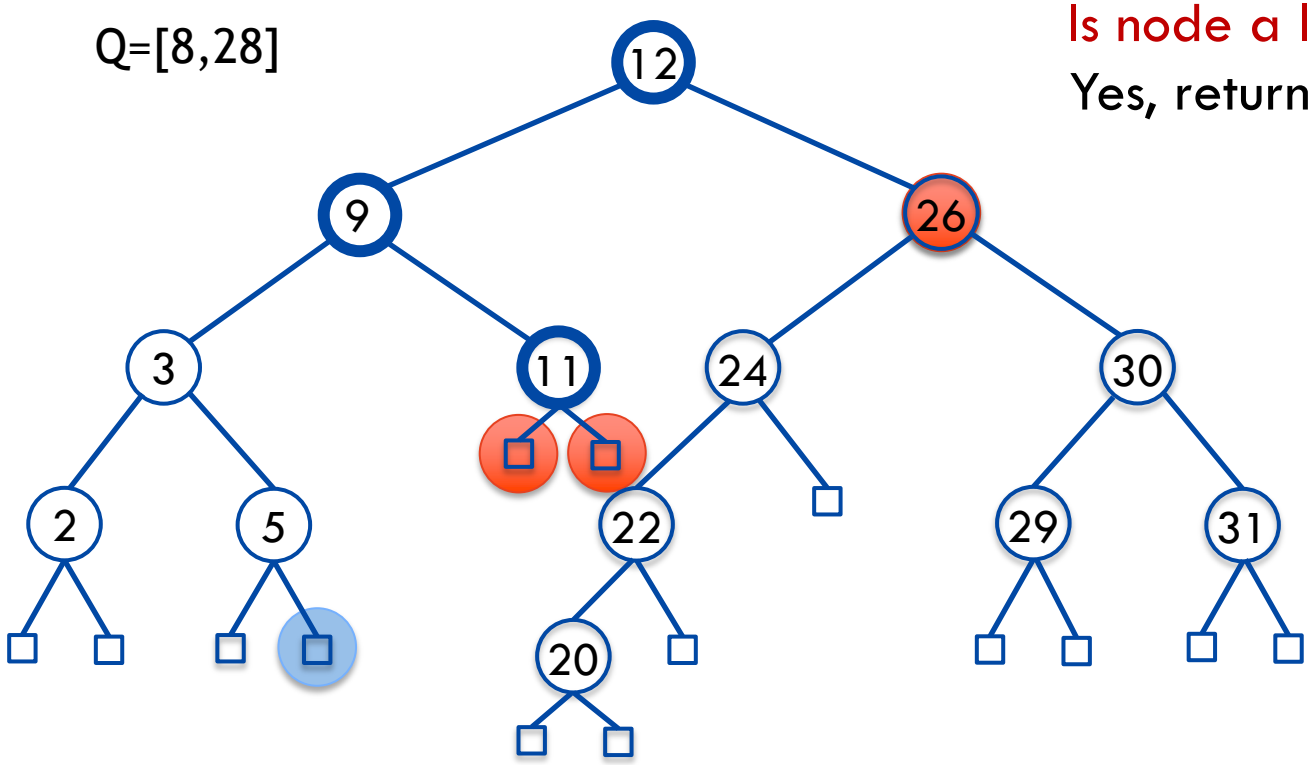
Is 11 in Q?

Yes, return 11 and continue the search in both subtrees.

Range queries

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

Q=[8,28]



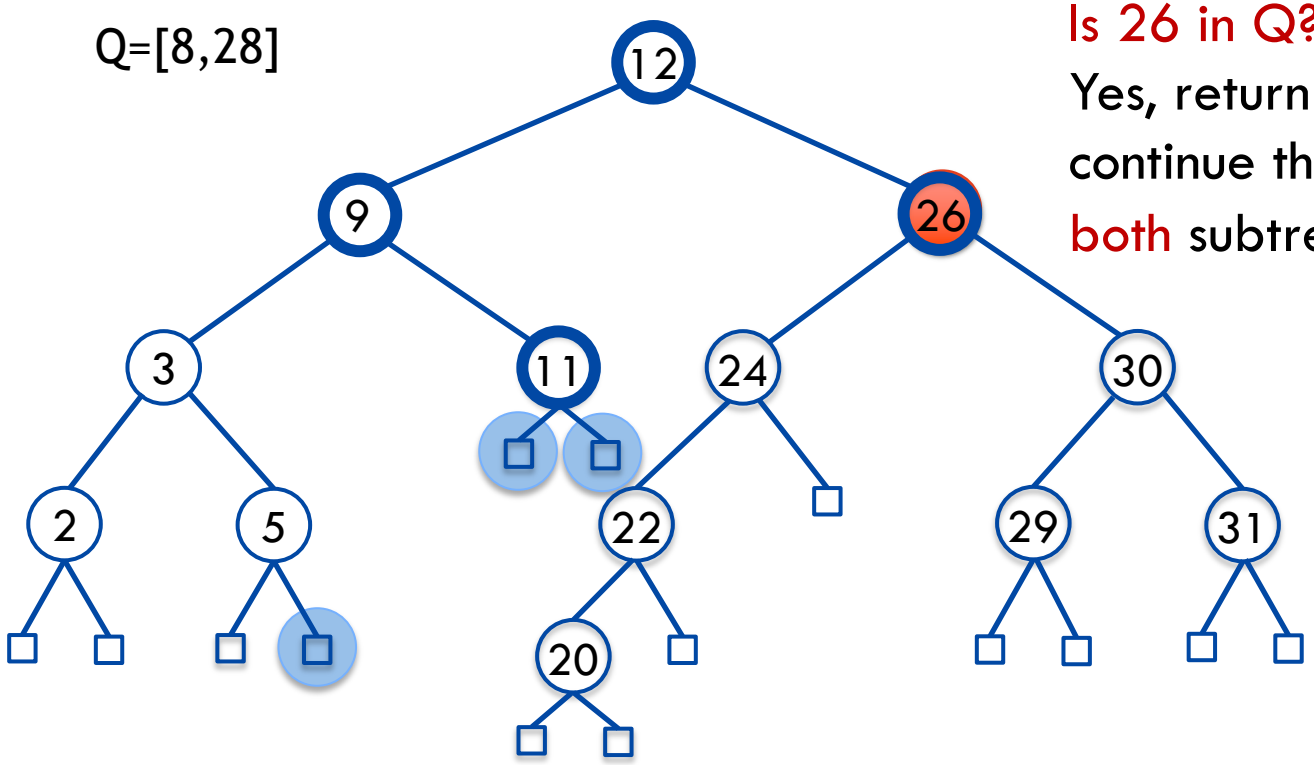
Is node a leaf (×2)?

Yes, return ∅.

Range queries

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

Q=[8,28]



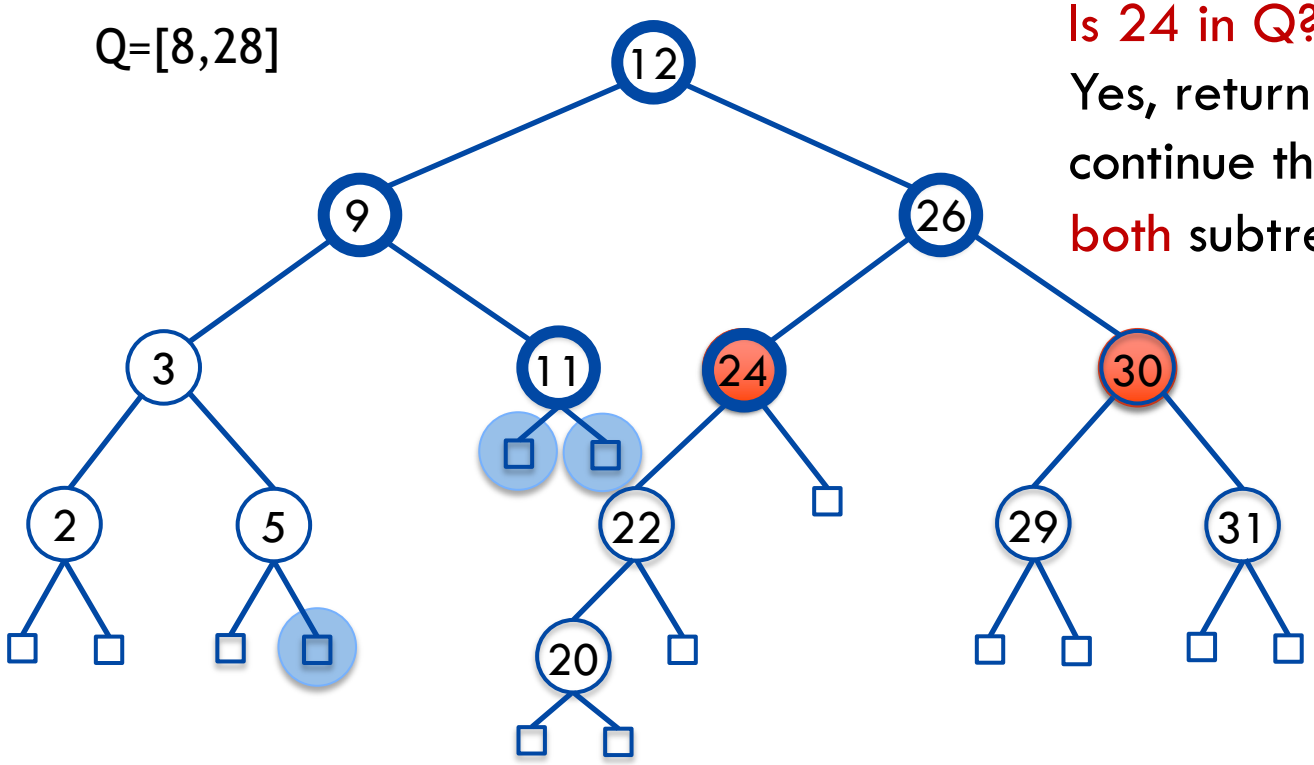
Is 26 in Q?

Yes, return 26 and continue the search in both subtrees.

Range queries

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

Q=[8,28]



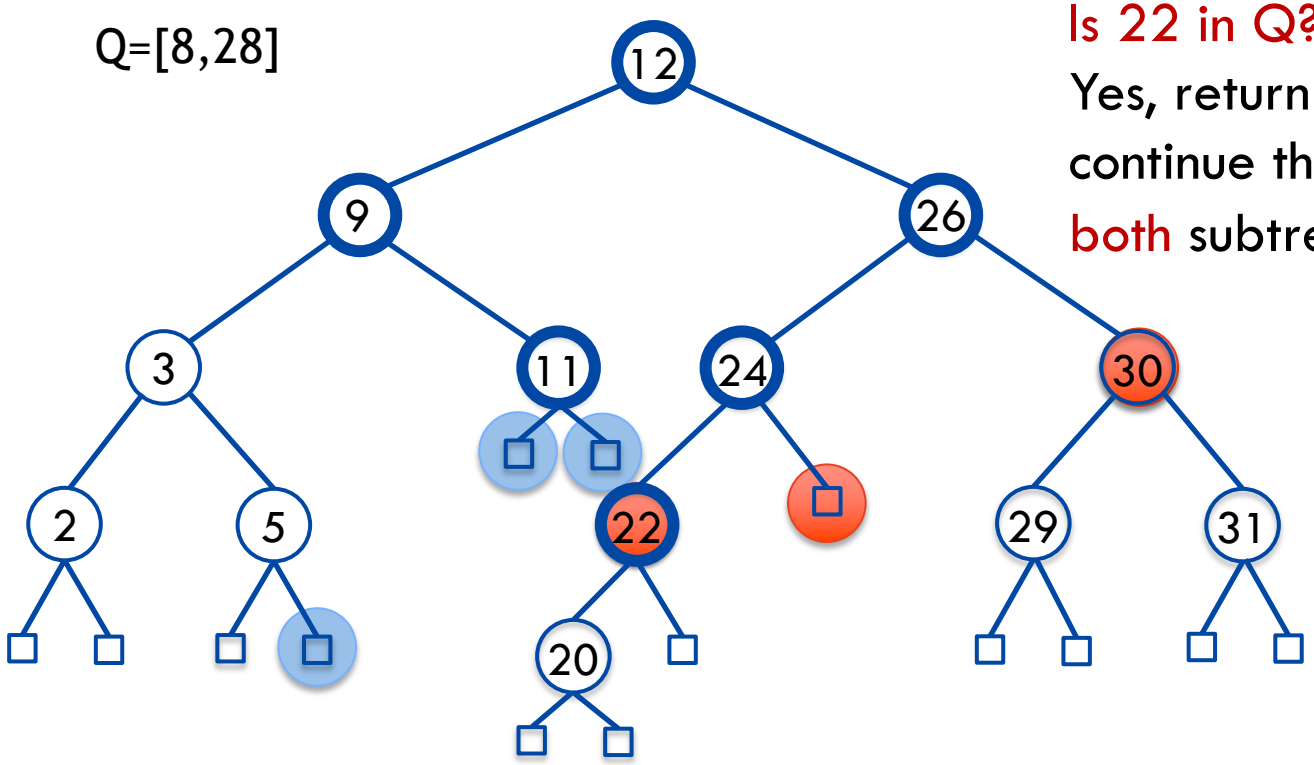
Is 24 in Q?

Yes, return 24 and continue the search in both subtrees.

Range queries

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

Q=[8,28]



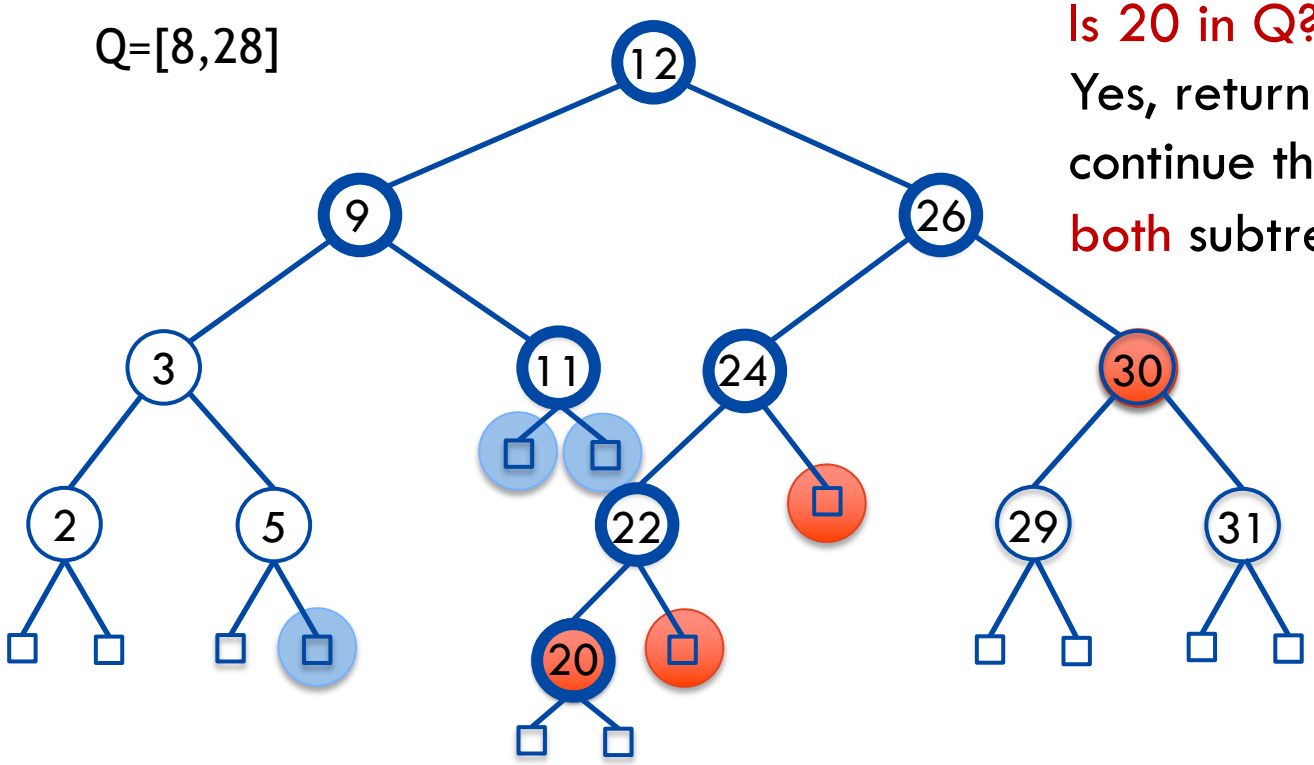
Is 22 in Q?

Yes, return 22 and continue the search in both subtrees.

Range queries

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

Q=[8,28]



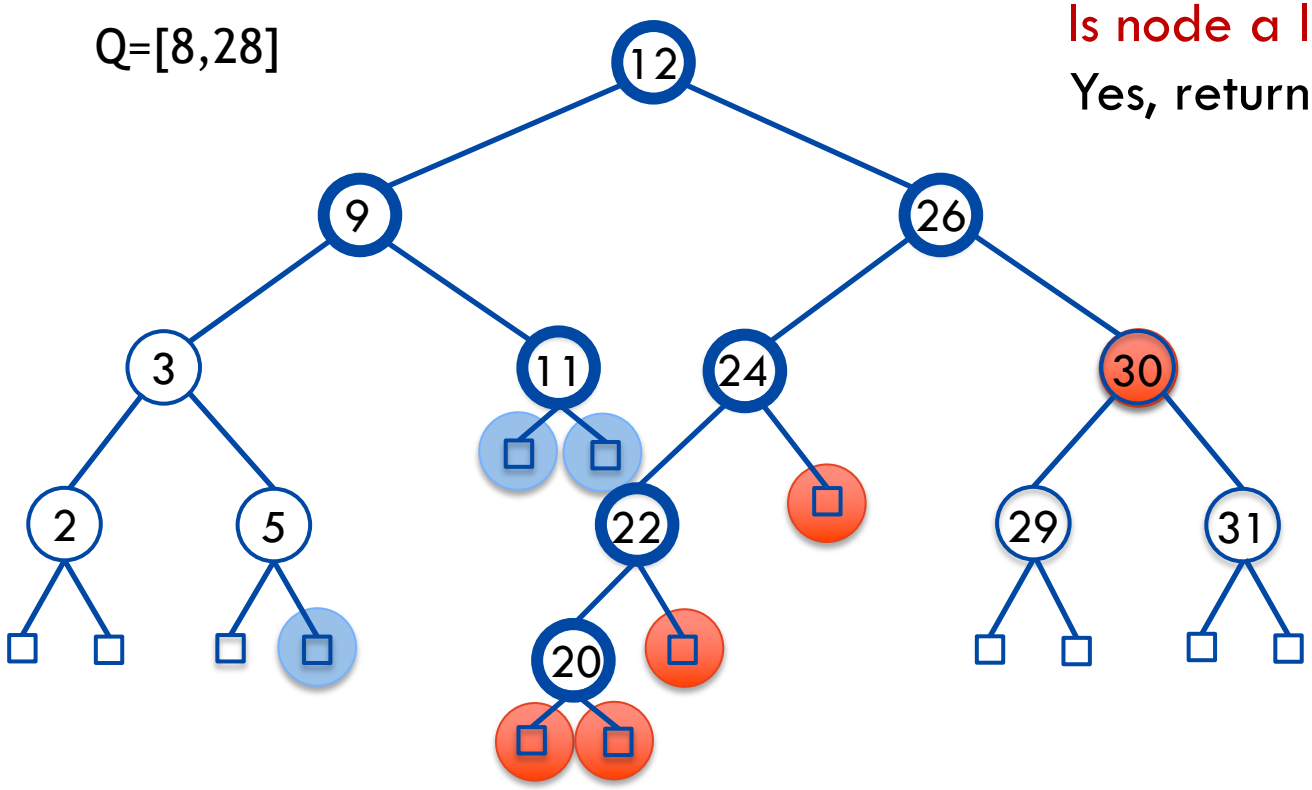
Is 20 in Q?

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

Range queries

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

Q=[8,28]



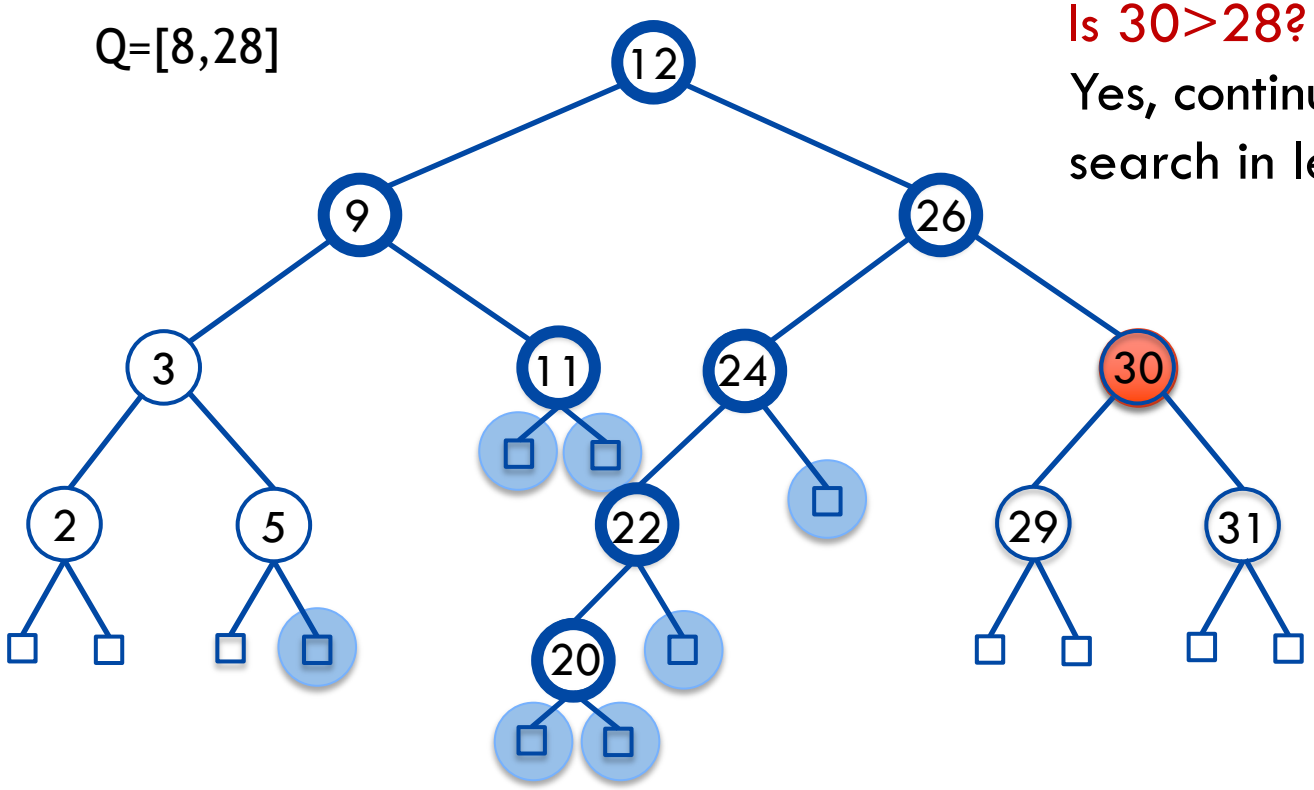
Is node a leaf (×4)?

Yes, return ∅.

Range queries

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

Q=[8,28]



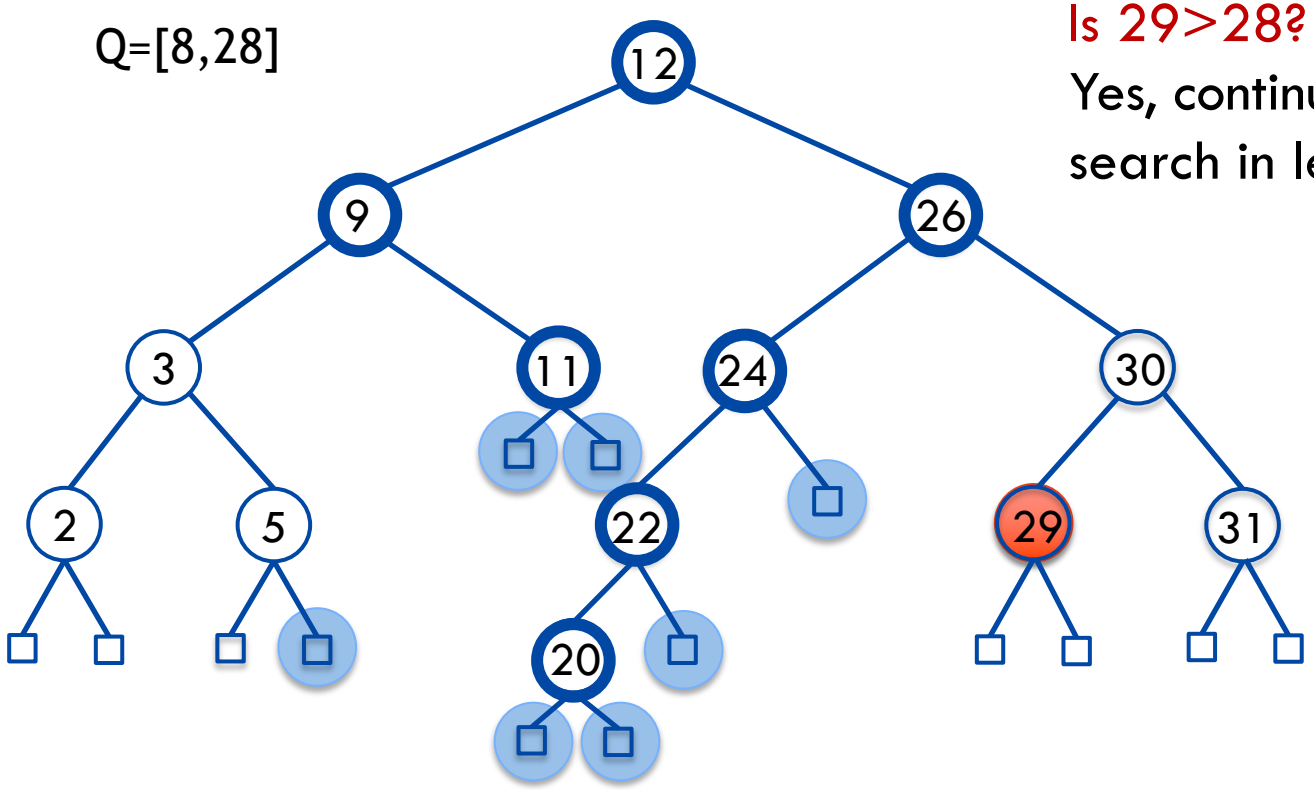
Is 30>28?

Yes, continue the search in left subtree.

Range queries

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

Q=[8,28]



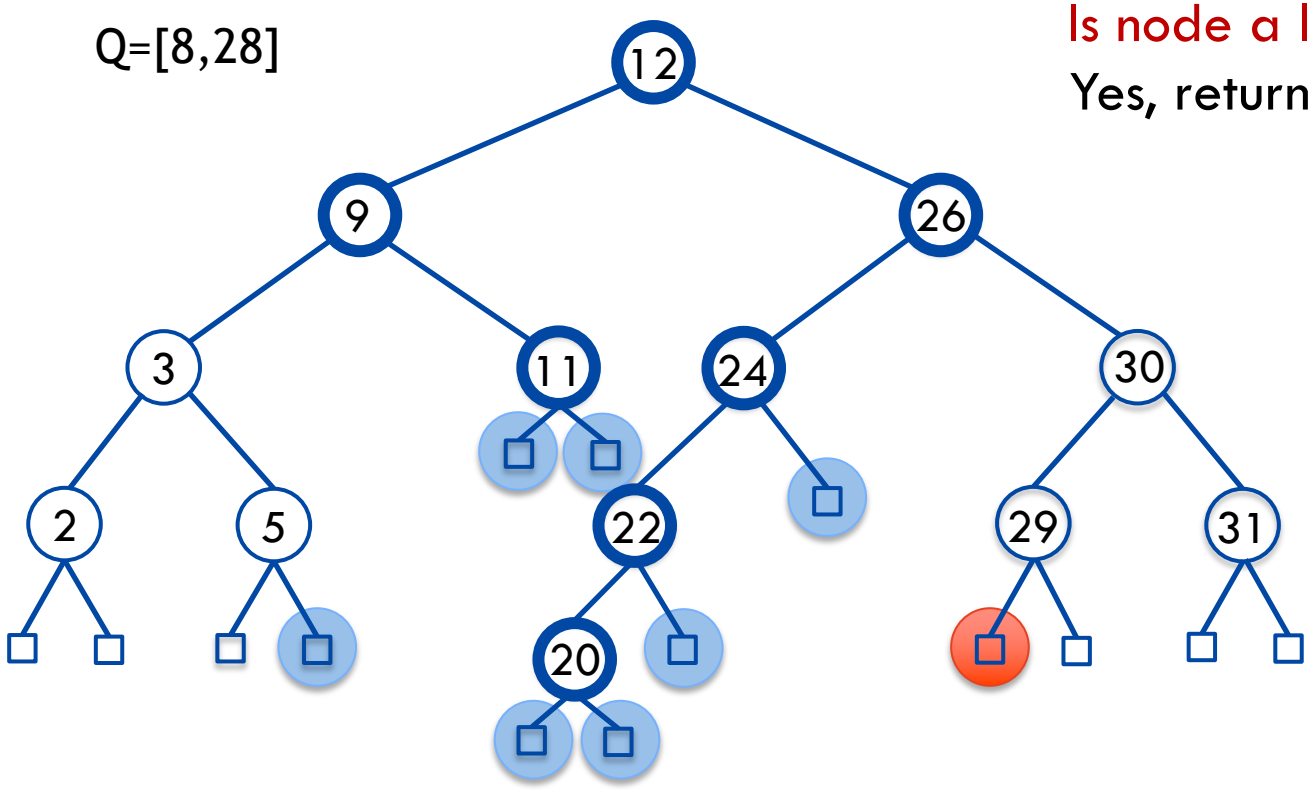
Is 29>28?

Yes, continue the search in left subtree.

Range queries

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

Q=[8,28]



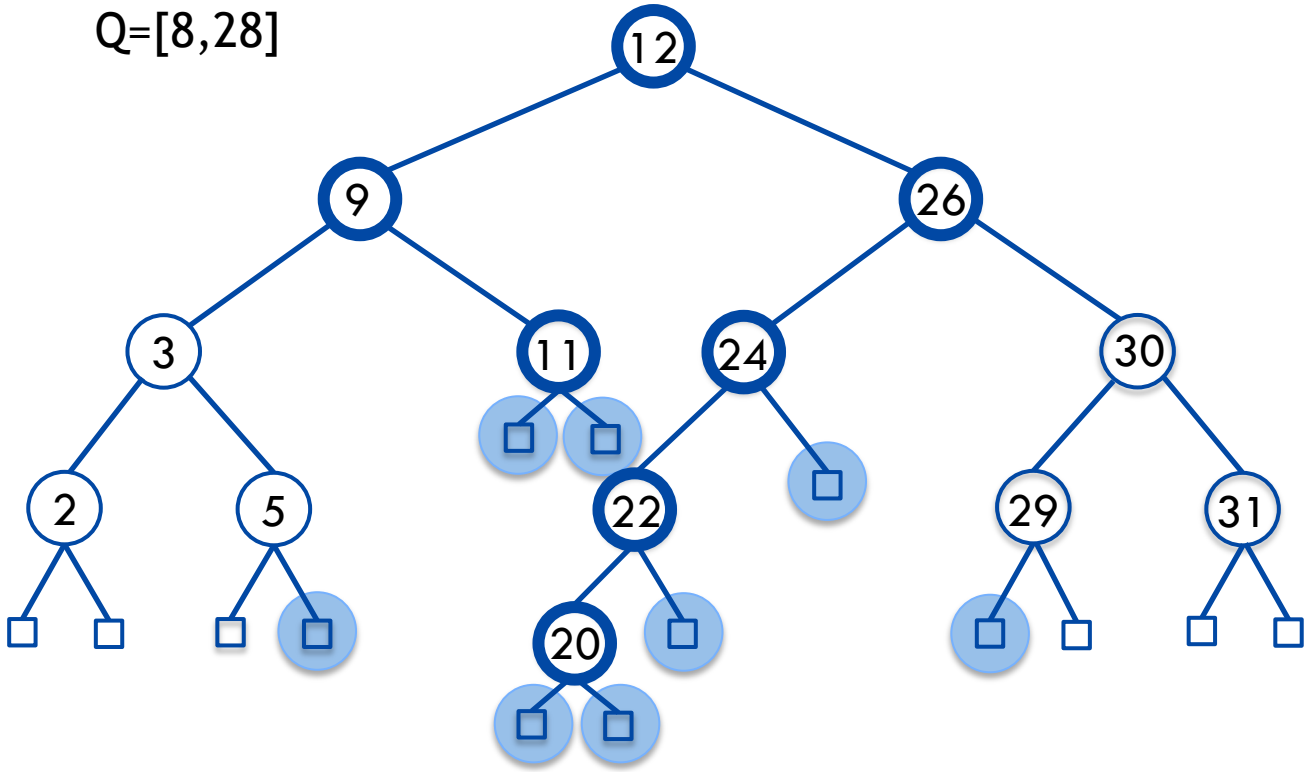
Is node a leaf?

Yes, return ∅.

Range queries

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

Q=[8,28]



Performance

To summarize:

We've learned:
- prop of BST
- insert()
- delete() and time complexity
- range query and time complexity

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

- $|\text{inside nodes}| \leq |\text{output}|$
- $|\text{boundary node}| \leq 2 * \text{tree height}$

This should be quite intuitive.
Just visualize the whole process in your head

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

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)

Next we will cover much more sophisticated algorithms that maintain a BST with height O(log n) at all times by rebalancing the tree with simple local transformations

This directly translates into O(log n) performance for searching

This comes naturally after we've known that a balanced BST has a lot of benefits.

We want to harness these properties.

Naturally, we're interested in seeing how to maintain this property.

Rank-balanced Trees

A family of balanced 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:

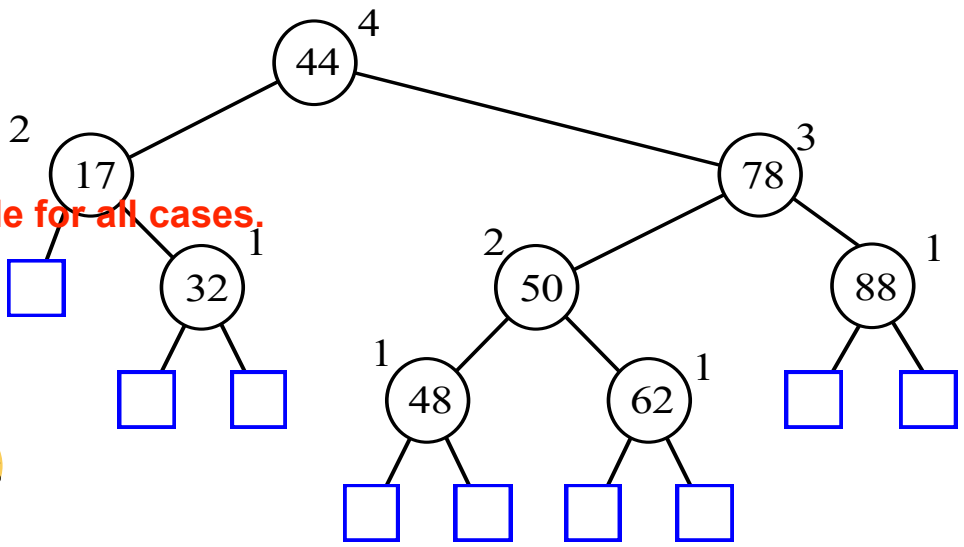
- AVL Trees (now)
- Red-Black Trees (book)

AVL Tree Definition

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

We can see that a difference by 0 is not possible for all cases.

We resort to 1 as the limit of the distance



Balance constraint: The ranks of 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 at most $O(\log n)$.

I couldn't come up with these proofs by myself

Proof (by induction): need to review this

- 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 smallest AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and another of height at least $h-2$:

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

- By induction we can show that for h even

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

- Taking logarithms: $h < 2 \log_2 N(h)$
- Thus the height of an AVL tree on n nodes is $O(\log n)$

Insertion in AVL trees

We of course just insert as we used to.

But what's really fascinating is how we rotate things.

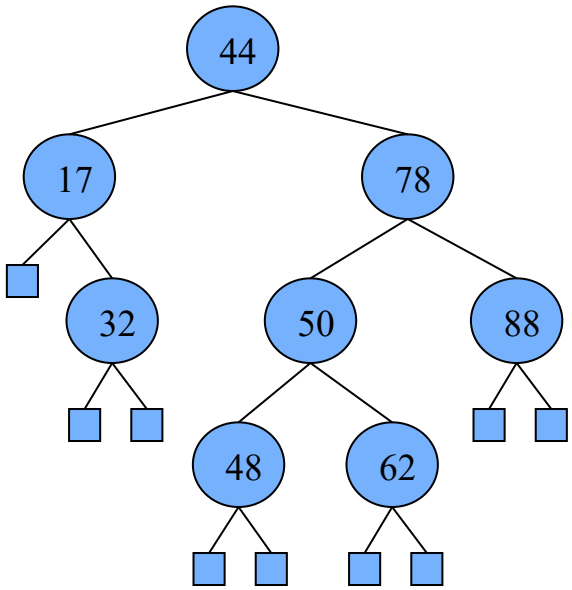
In this case, we conduct some operations after insertion.

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

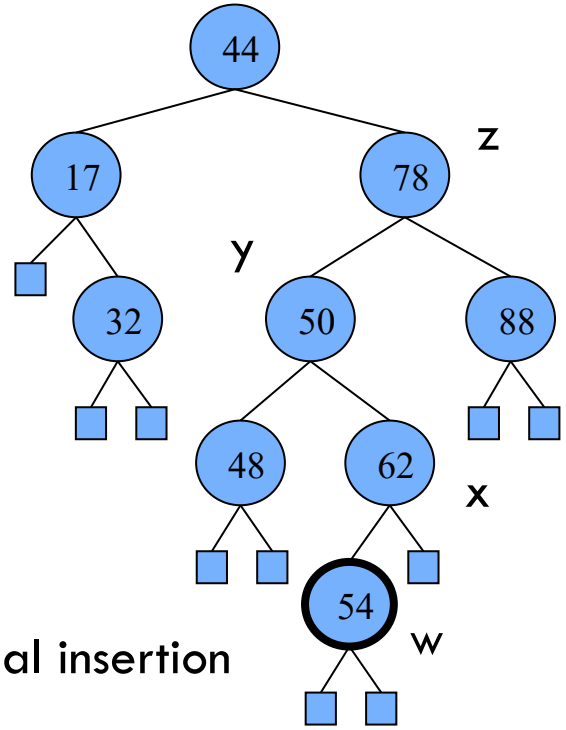
1. If k is in the tree, search for k ends at node holding k
   There is nothing to do so tree structure does not change
2. If k is not in the tree, search for k ends at external node w.
   Make this be a new internal node containing key k
3. The new tree has BST property, but it may not have AVL balance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

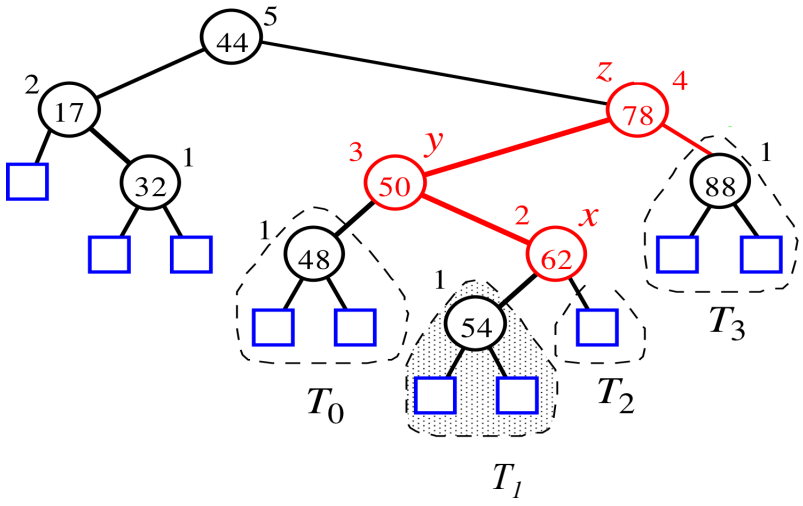


before inserting 54



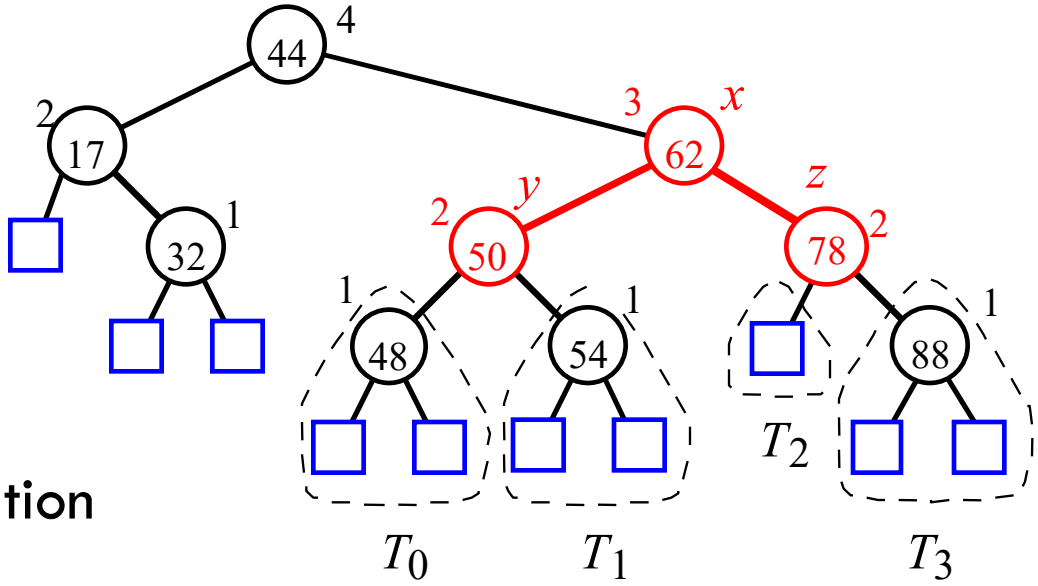
after initial insertion

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



Augmenting BST with a height attribute

But how do we know the height of each node? If we had to compute this from scratch it would take O(n) time

Therefore, we need to have this pre-computed and update the height value after each insertion and rebalancing operation:

- After we create a node w, we should set its height to be 1, and then update the height of its ancestors.
- After we rotate (z, y, x) we should update their height and that of their ancestors.

Thus, we can maintain the height only using O(h) work per insert

Improving Balance: Trinode Restructuring

This and the next exmaple should be very intuitive.

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

We can see that a very key thing is to always remember the BST property:

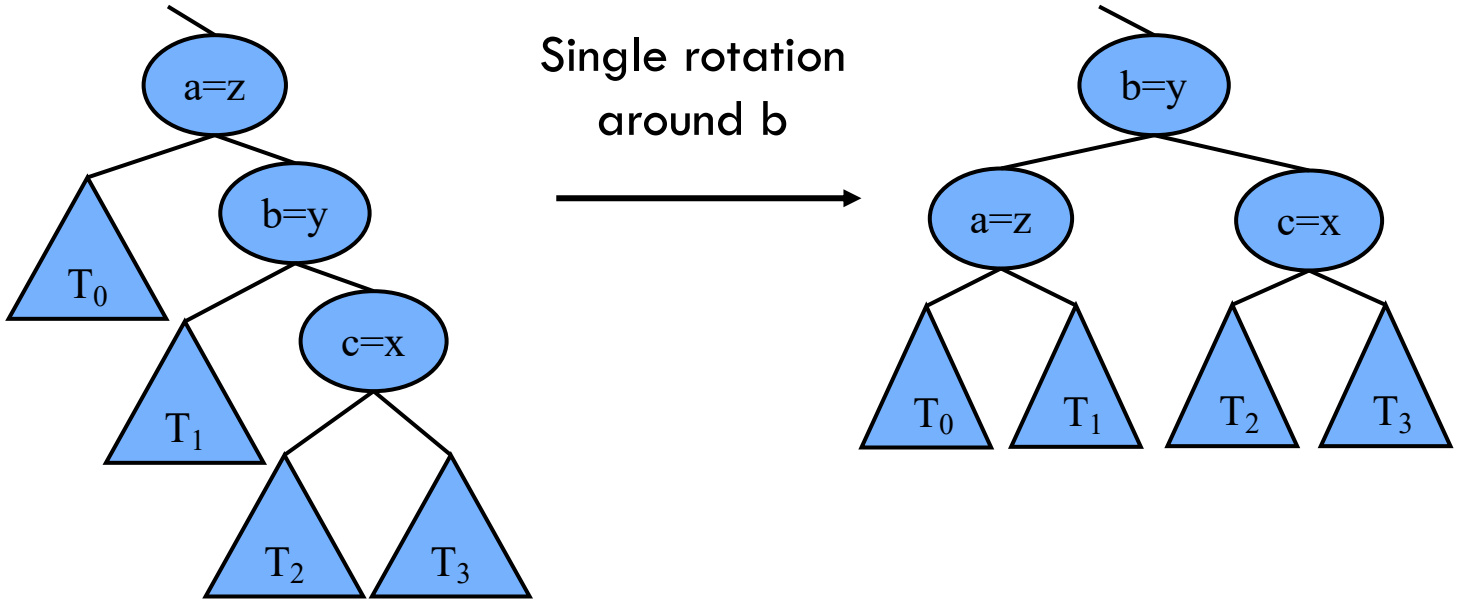
Let a, b, c be the inorder listing of *x*, *y*, *z*

Left < middle < right

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

We're uing this property

Note how the no. of Ts and the parent nodes are exactly matched (This sentence is not well written)



Improving Balance: Trinode Restructuring

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

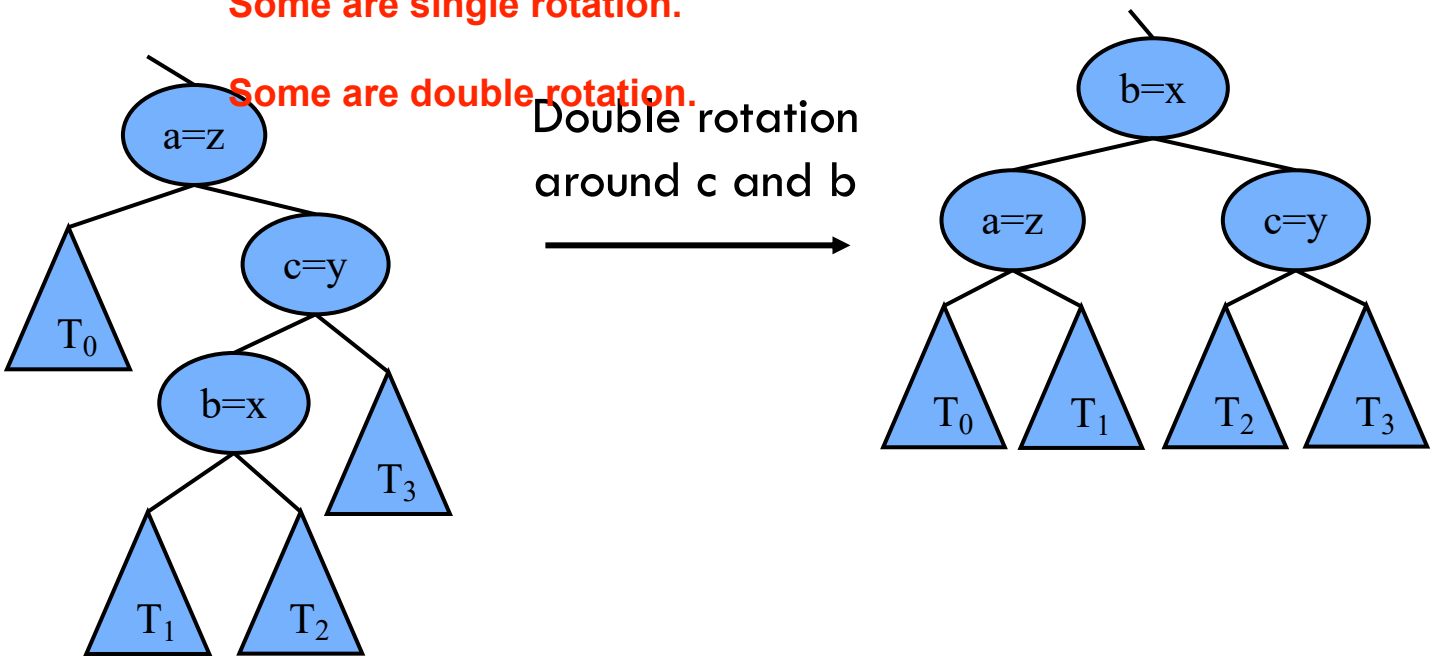
Let a, b, c be the inorder listing of *x, y, z*

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

Also notice the number of rotation taken at each time.

Some are single rotation.

Some are double rotation.



Pseudo-code

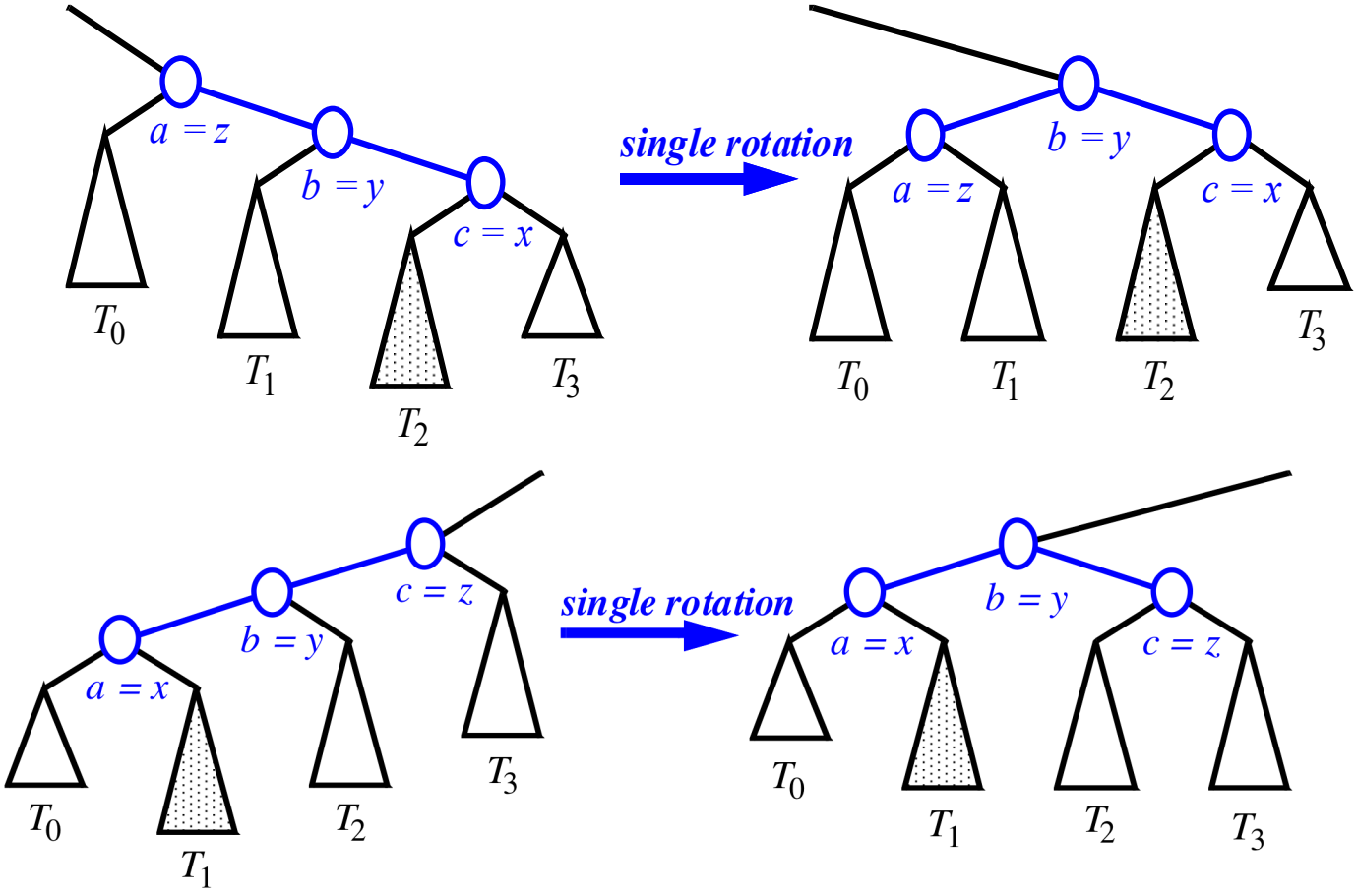
The algorithm for doing a trinode restructuring, which is used, possibly repeatedly, to restore balance after an insertion or deletion.

```
def 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 the left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0,T_1,T_2,T_3) be the left-to-right (inorder) listing of the four subtrees of x, y, and z that are not rooted at x, y, and z.
2. Replace the subtree 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
4. Let c be the right child of b and let T_2 and T_3 be the left and right subtrees of c
5. Recalculate the heights of a, b, and c from the corresponding values stored at their children
6. return b

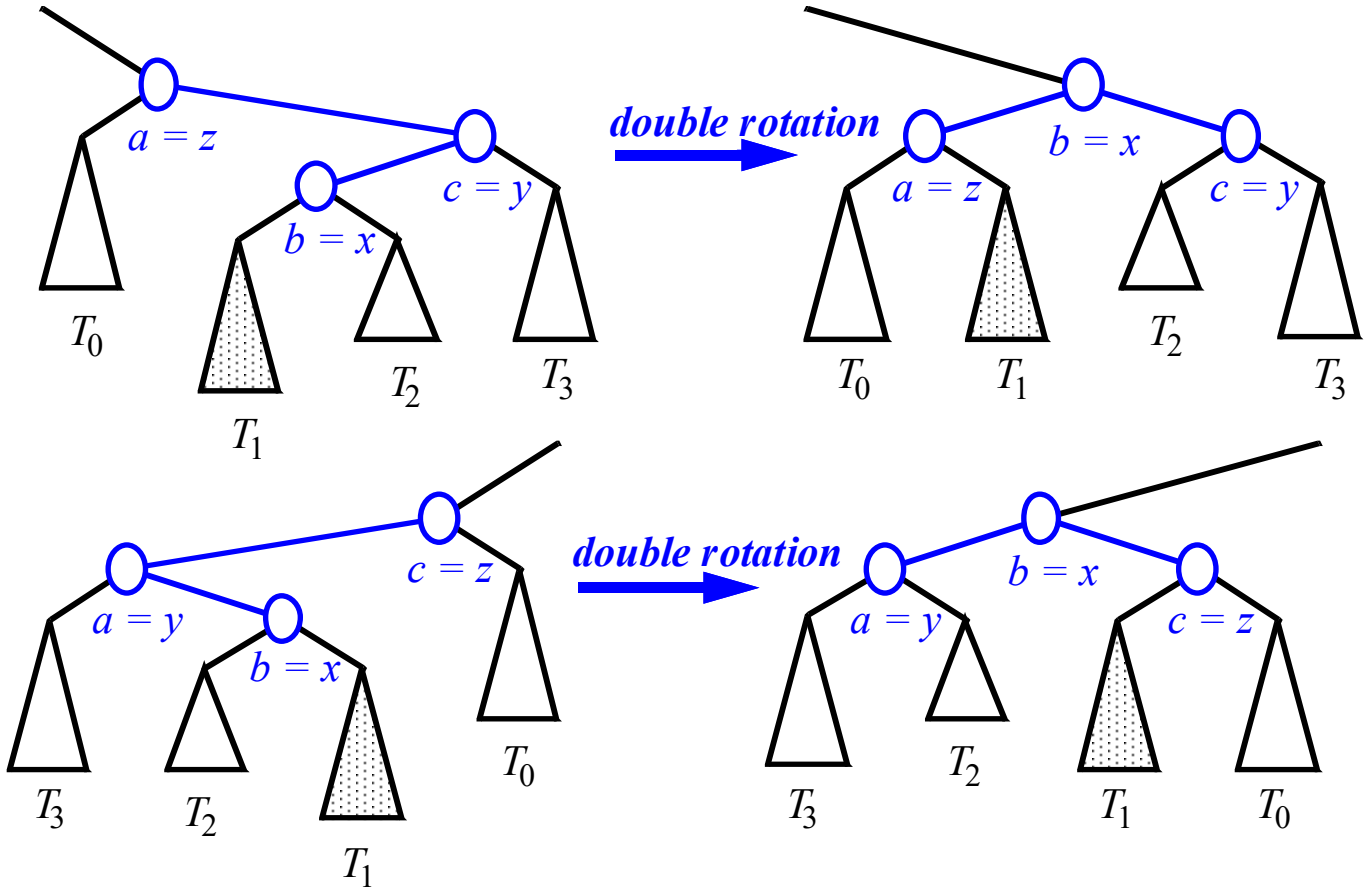
Trinode Restructuring (when done by Single Rotation)

Single Rotations:



Trinode Restructuring (when done by Double Rotation)

Double rotations:



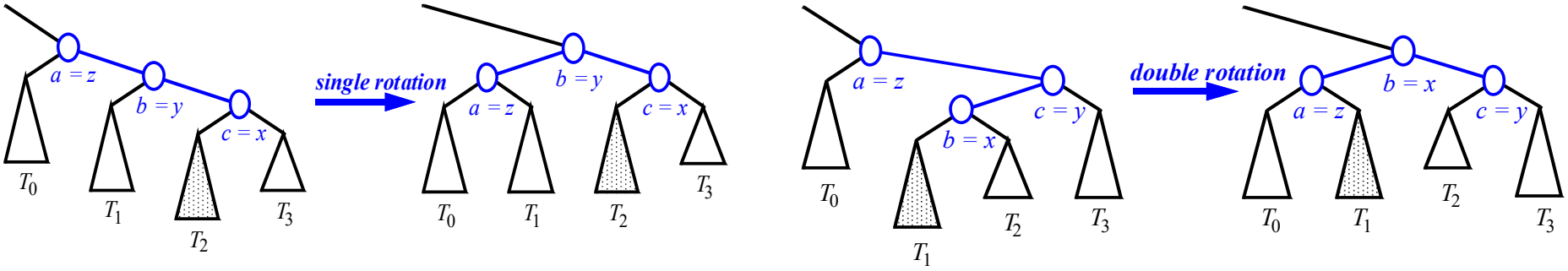
Performance

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.

TBH,

This is an amazing property



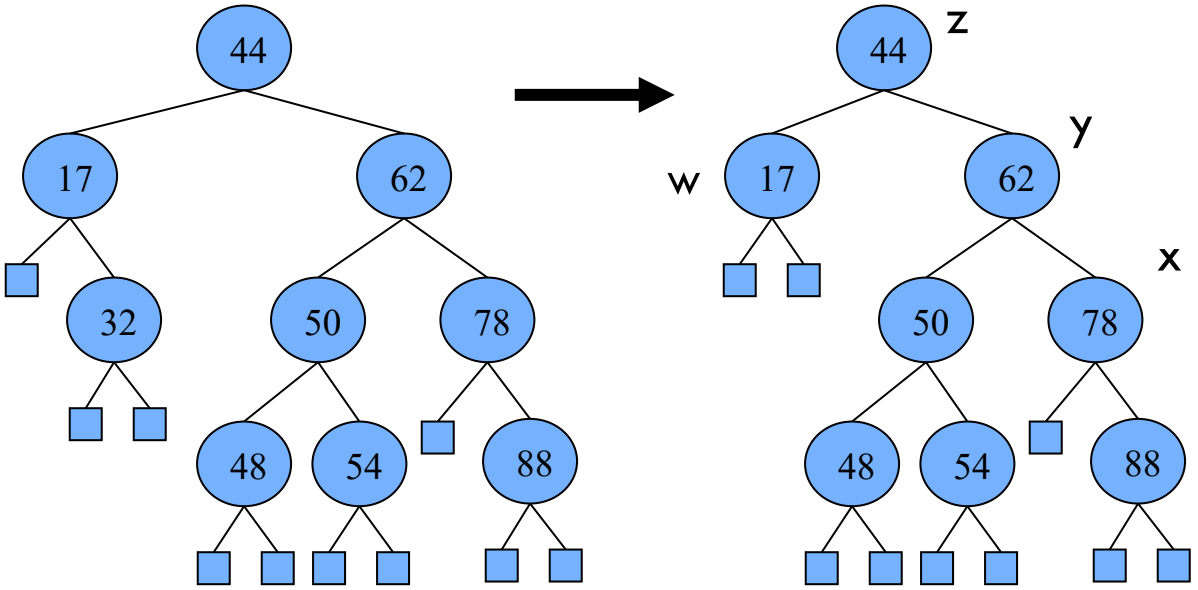
Removal in AVL trees

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 performs 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 decreased 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

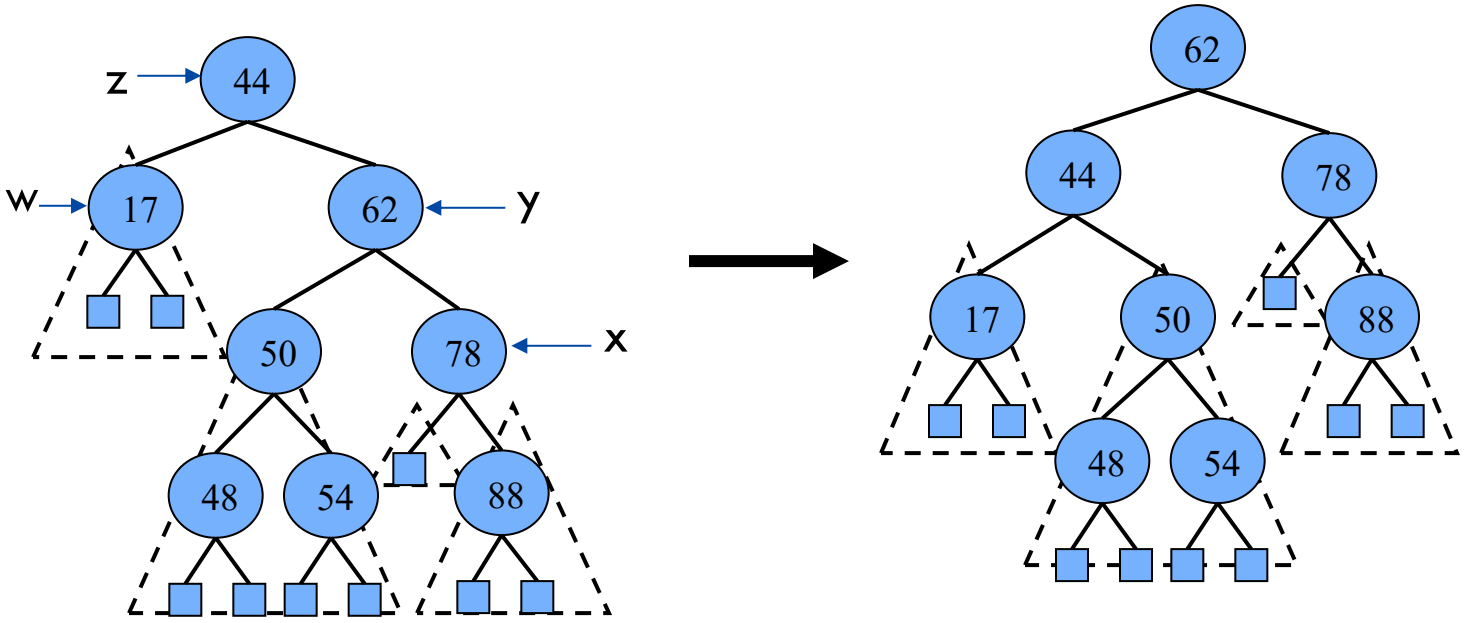


before deletion of 32

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 up 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

The AVL property is like the cherry on top but not the gravy on top.

You don't increase the number of cherries you put as you do with the amount of gravy

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

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**	∅
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)

These four concepts are very important since they're new.

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)

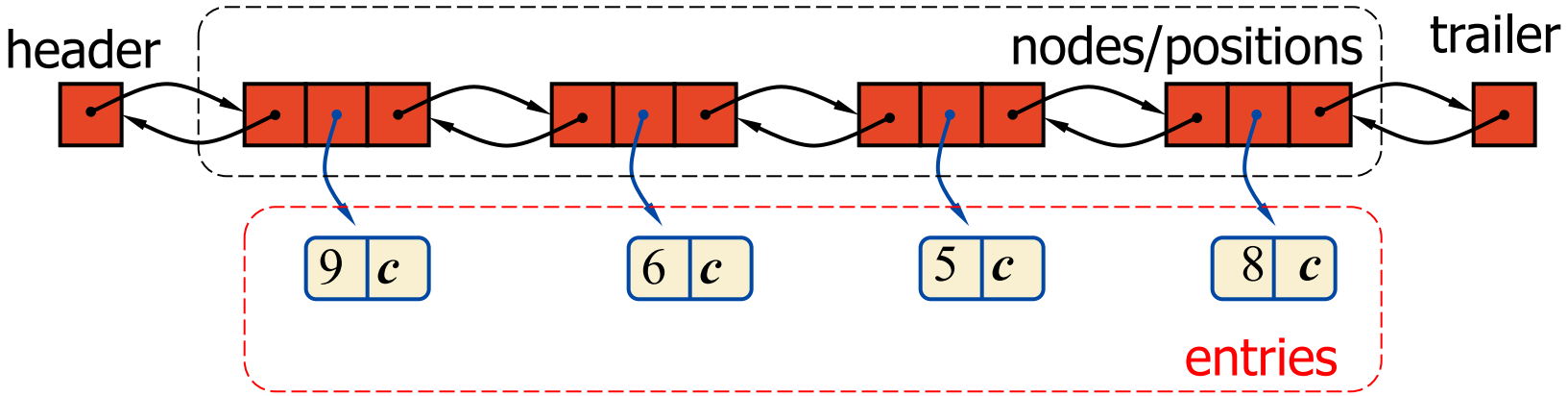
subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2

List-Based (unsorted) Map

We can implement a map using an unsorted list of key-item pairs

To do a get we may have to traverse the whole list, so those operations take O(n) time.

Only feasible if map is very small or if we put things at the end and do not need to perform many gets (i.e., system log)



Tree-Based (sorted) Map

We can implement a sorted map using an AVL tree, where each node stores a key-item pair

To do a get or a put we search for the key in the tree, so these operations take O(h) time, which can be O(log n) if the tree is balanced.

Only feasible if there is a total ordering on the keys.

Index-based searching

Think of it as an order list for the MAP ADT

Being able to do key lookups is very useful, but sometimes we need to access the ith key in the tree:

- select(i) returns the ith smallest key (assume no duplicate keys)

We could store the keys in a list. This solution needs to spend O(n) time for selecting and for searching.

We could store (i, k_i) in an AVL tree where k_i is the ith key we want to store. This solution can do O(log n) lookups and selection but takes O(n) time to do insertions and deletion.

There is a solution that can do all operations in O(log n) but requires us to augment the AVL tree data structure.

Index-based searching

An augmentation like this is necessary to calculate the index for select()

For each node u, let us define size(u) to be the size of the subtree rooted at u. The left subtree of u holds all the keys that are smaller than key(u), so the rank of u in its subtree is size(u.left) + 1

```
def select(x, i):
   input A node x of an AVL tree
         integer i : 1 ≤ i ≤ size(x) position in T_x to select
   output ith element in T_x (i=1 is first key)

   if i ≤ size(x.left) then
      return select(x.left, i)
   else if i = size(x.left) + 1 then
      return x.key
   else
      return select(x.right, i – size(x.left) – 1)
```

Skip lists

Another way to implement a Map.

So, why are we looking at another different way of doing this?

- Relatively simple data structure that's built in a randomized way
- No need for rebalancing like in AVL trees
- Still has O(log n) expected worst-case time

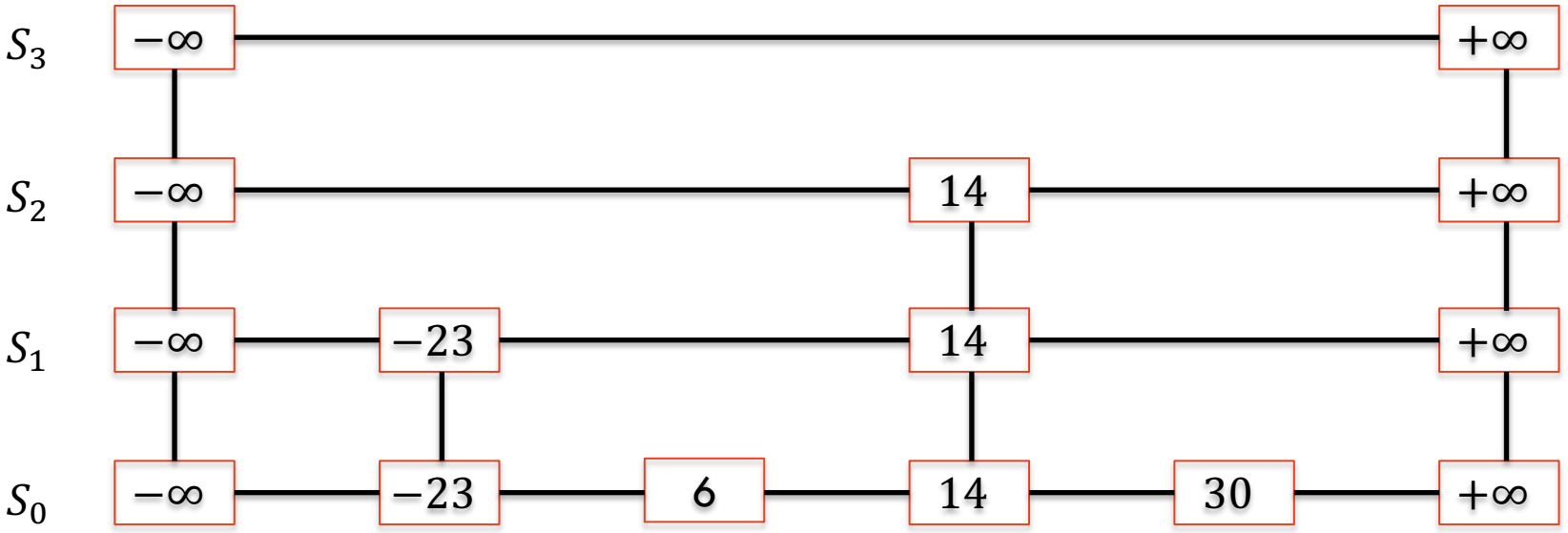
Applications:

- Various database systems use it
- Concurrent/parallel computing environments

Skip lists

Leveled structure, where every level is a subset of the one below it.

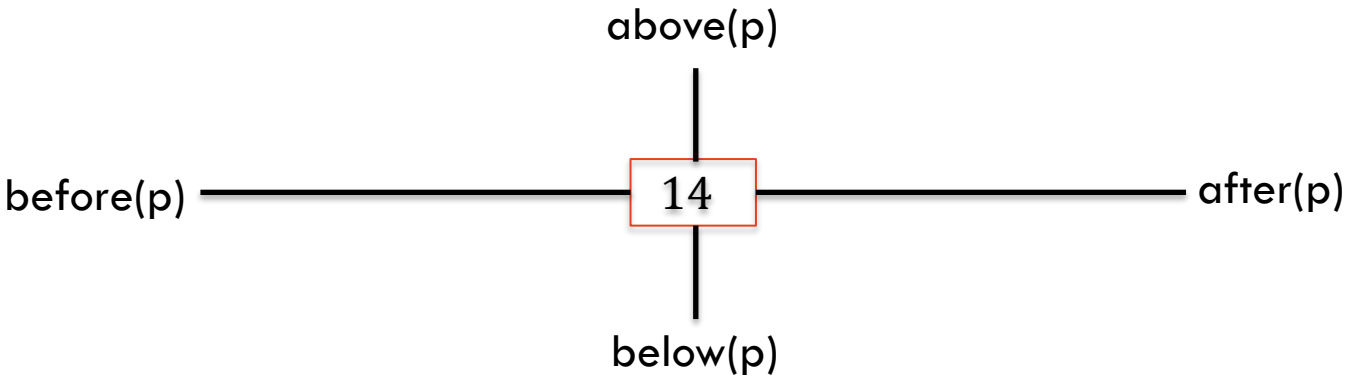
Next level's elements determined by coin flips.



Skip lists

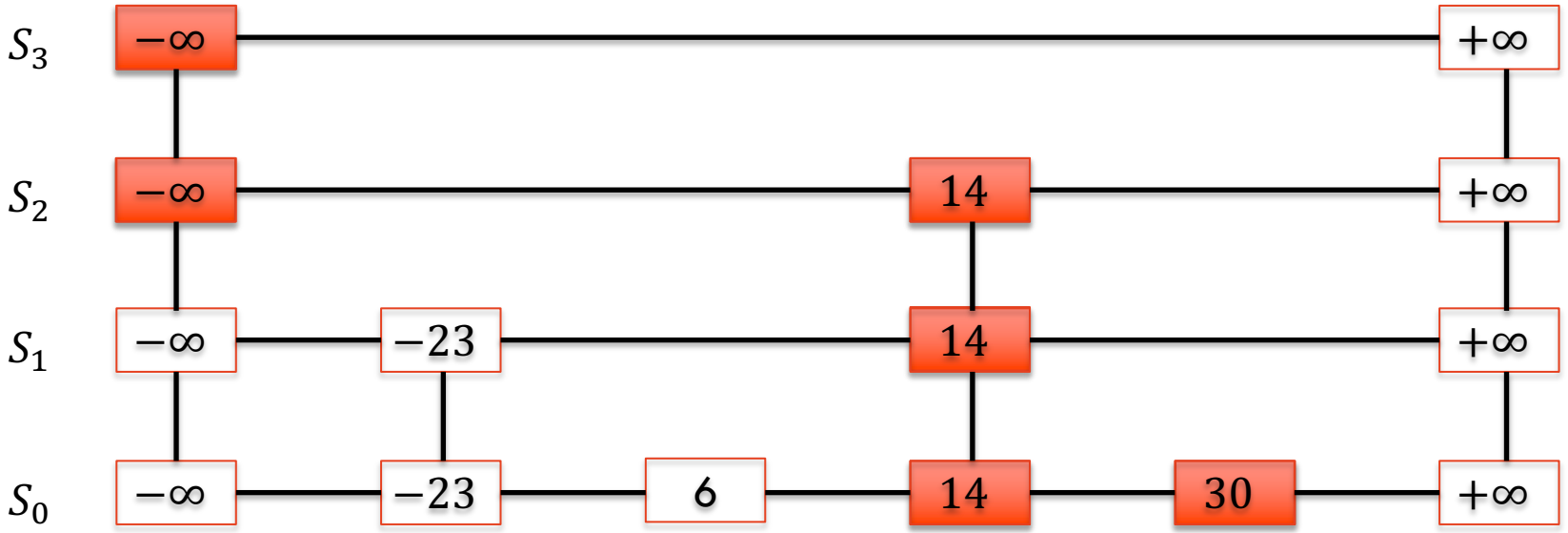
A node p has pointer to:

- after(p): Node following p on same level.
- before(p): Node preceding p in the same level.
- above(p): Node above p in the same tower.
- below(p): Node below p in the same tower.



Search

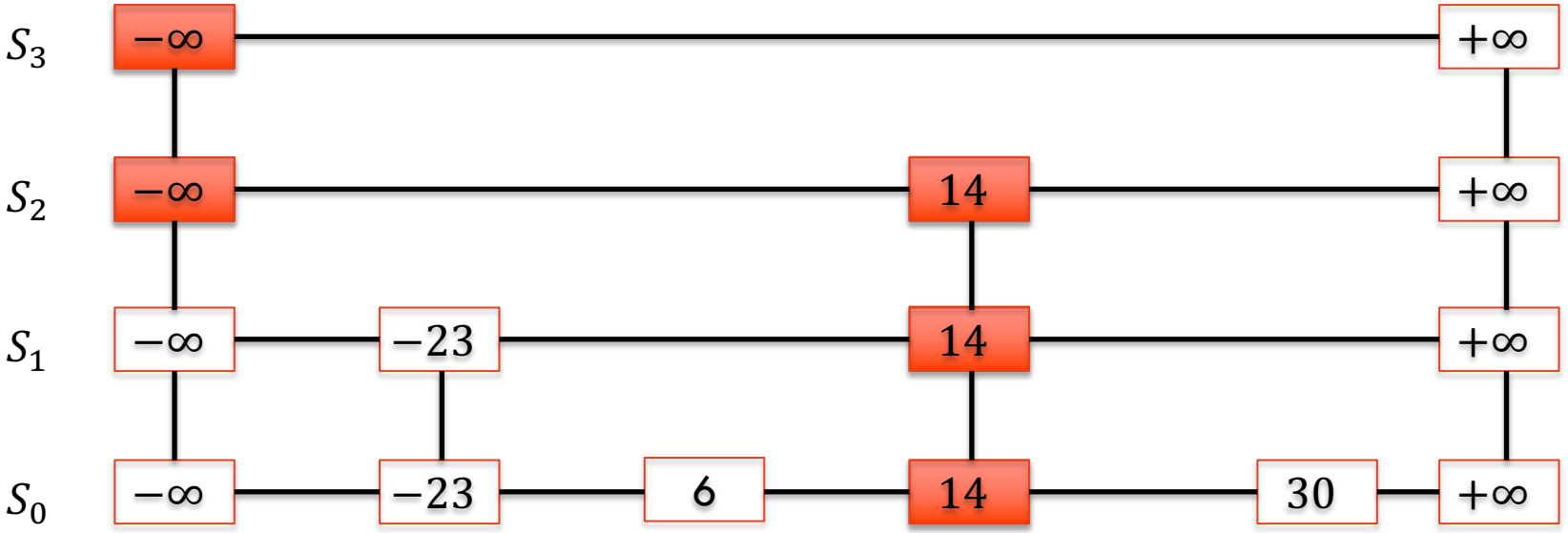
```
def search(p,k):
  while below(p) ≠ null do
    p ← below(p)
    while key(after(p)) ≤ k do
      p ← after(p)
  return p
```



Example: search(topleft node, 30)

Insertion

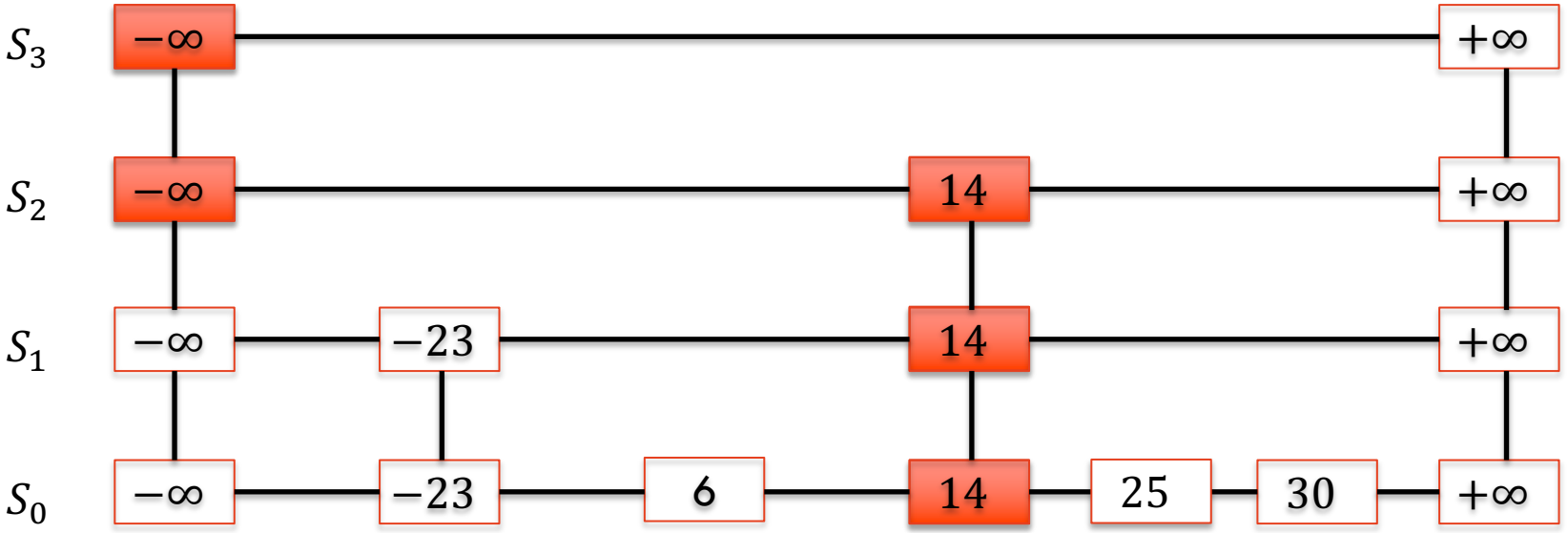
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Insertion

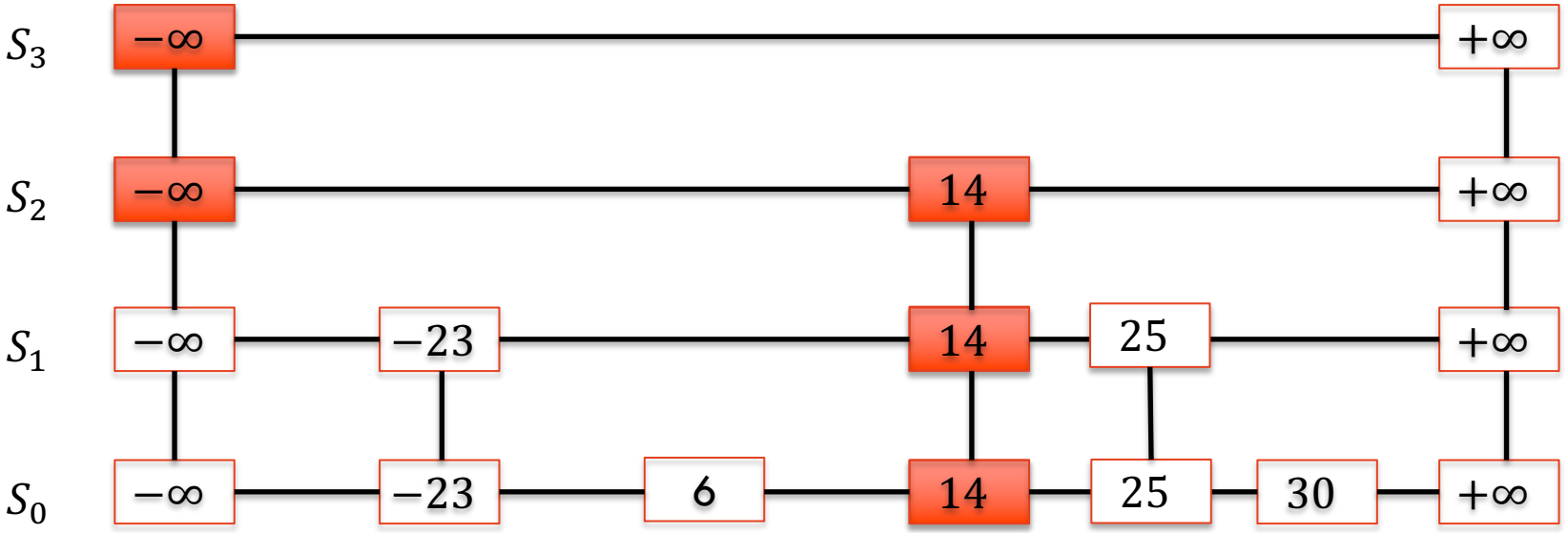
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Insertion

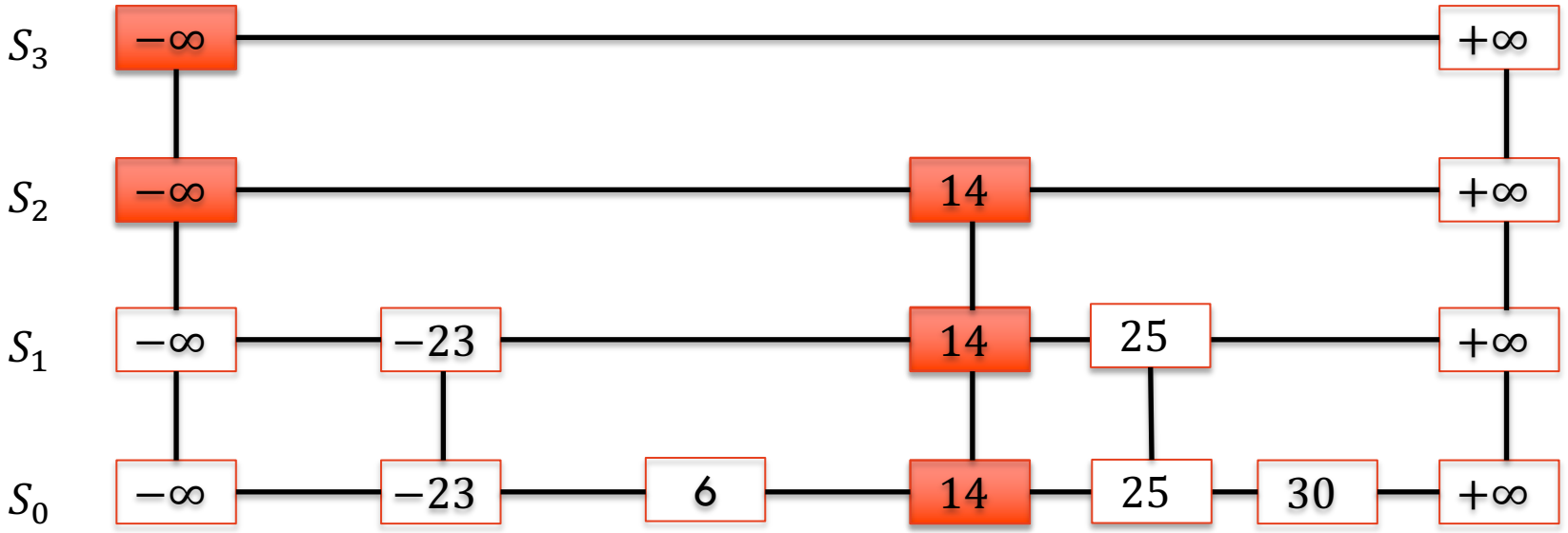
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Removal

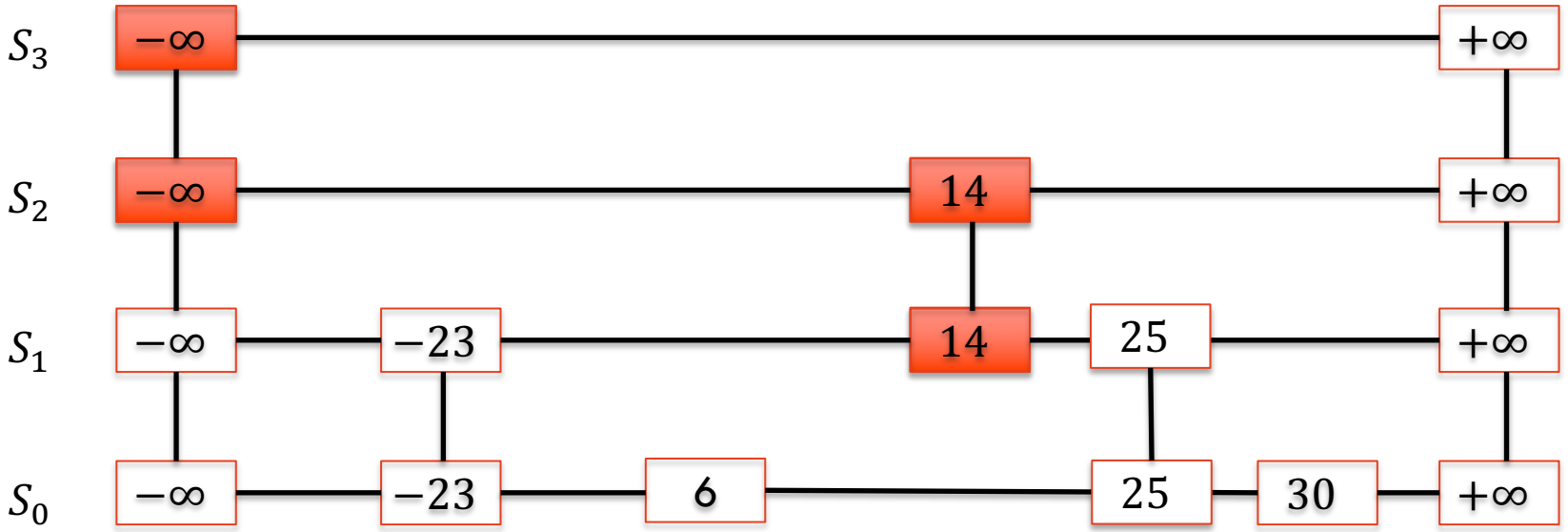
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

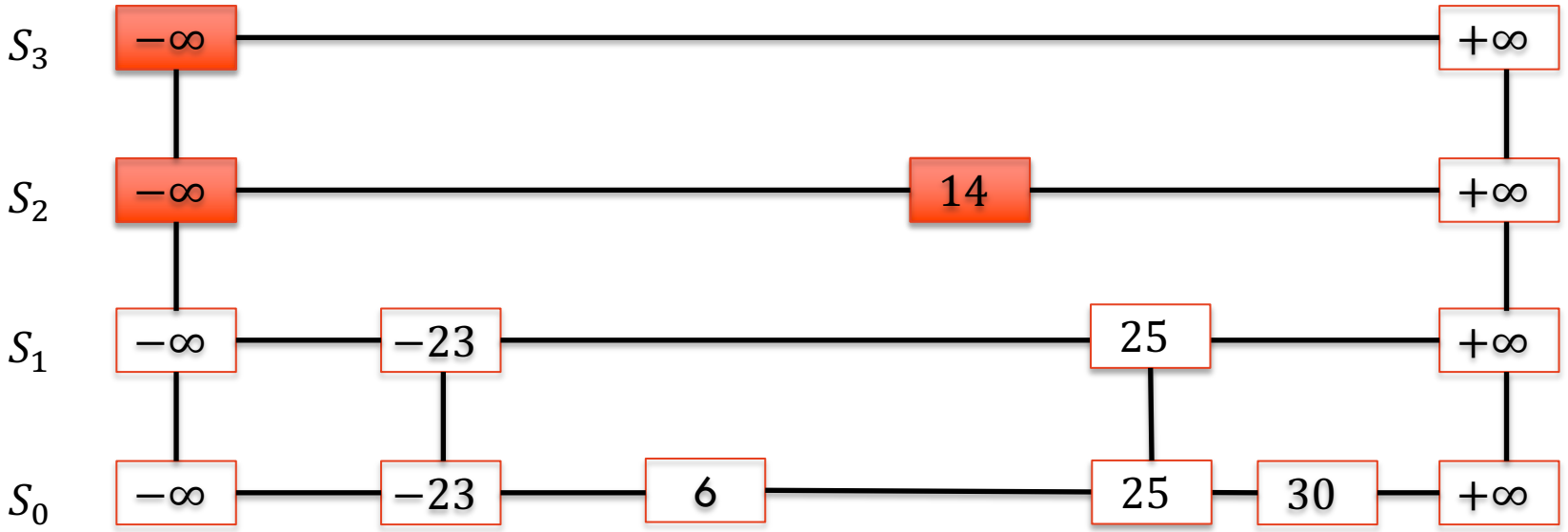
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

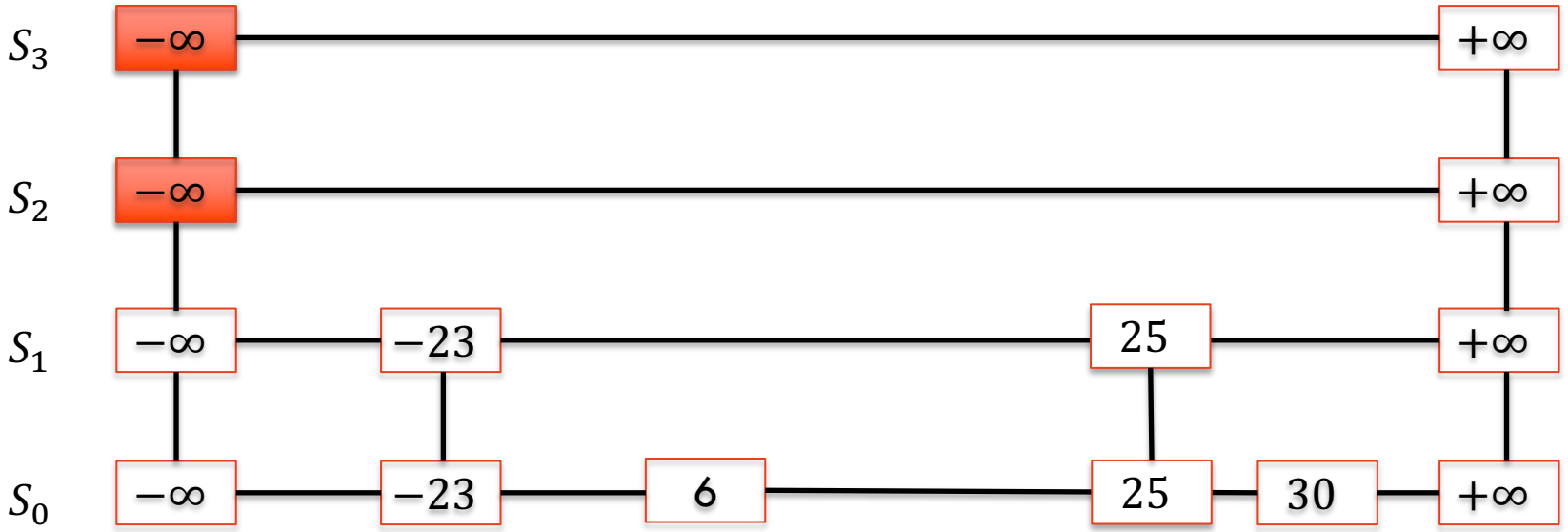
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Skip lists: Top layer

Keep a pointer to the topleft node.

Choices for the top layer:

- Keep at a fixed level, say $\max\{10, 3\lceil\log(n)\rceil\}$
  - Insertion needs to take this into account

- Variable level
  - Continue insertion until coin comes up tails
  - No modification required
  - Probability that this gives more than O(log n) levels is very low

Skip lists: Analysis

Theorem:

With high probability, the height of a skip list is O(log n).

Proof:

- The probability that an element is present at height i is $1/2^i$.
  - I.e., the probability that the coin comes up heads i times.
- The probability that level i has at least one item is at most $n/2^i$.
- The probability that skip list has height h is probability that level h has at least one element.
- So, probability that skip list has height larger than c log n is at most

$$\frac{n}{2^{c \log n}} = \frac{n}{(2^{\log n})^c} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

- So, probability that skip list has height O(log n) is at least

$$1 - \frac{1}{n^{c-1}}$$

Skip lists: Search Analysis

Theorem:

The expected search time of a skip list is O(log n).

Proof:

- Searching consists of horizontal and vertical steps.
- There are h vertical steps, so O(log n) with high probability.
- To have a horizontal step on level i, the next node can't be on level i+1.
- The probability of this is 1/2.
- This means that the expected number of horizontal steps per level is 2.
- So we expect to spend O(1) time per level.
- Expected search time: O(log n) time with high probability.

Insertion and deletion take expected O(log n) time using similar analysis.

Skip lists: Space Analysis

Theorem:

The expected space used by a skip list is O(n).

Proof:

- Space per node: O(1)
- Expected number of nodes at level i is $n/2^i$.
- Thus expected number of nodes is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

Announcements

Assignment 1: Available on Ed, submit via Gradescope

- read guidelines (last page)
- read guide to written assignments (in Ed)
- ask clarifying questions in lecture, during tutorials, in Ed