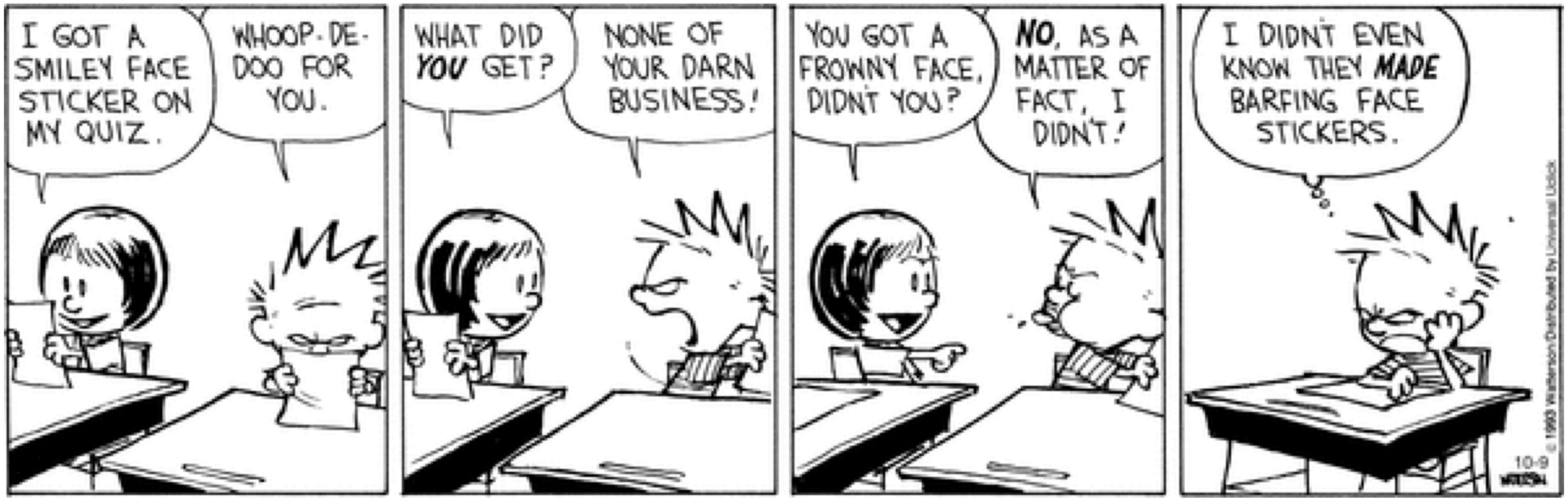
BALANCED SEARCH TREES

David Kauchak
CS 62 – Spring 2020

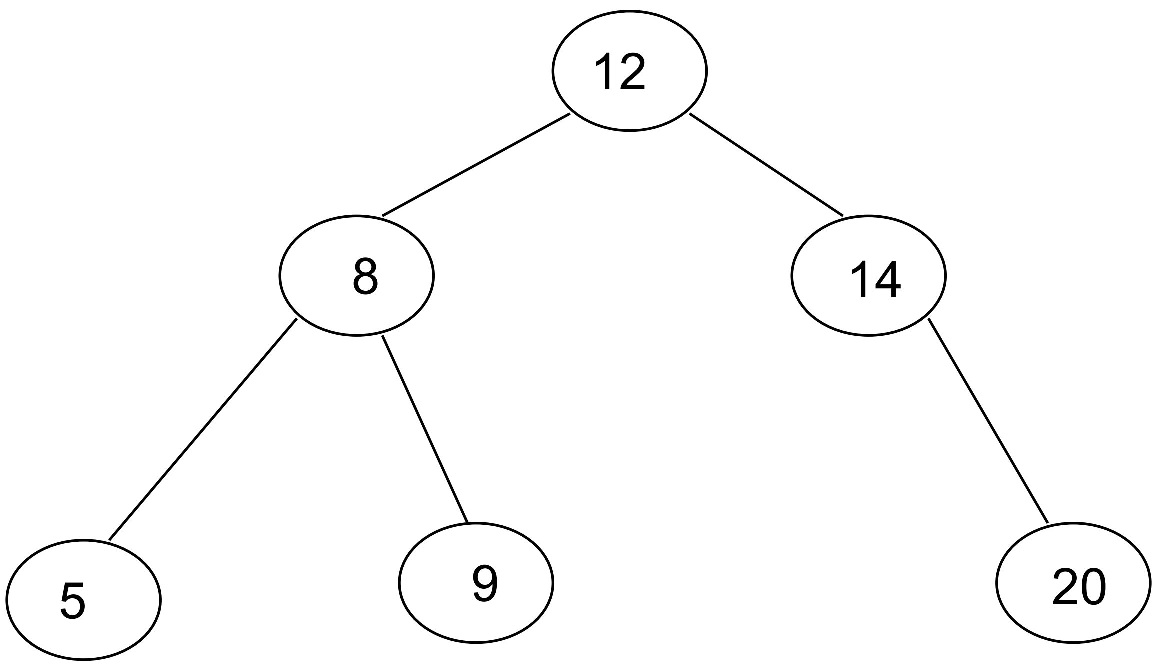

I GOT A SMILEY FACE STICKER ON MY QUIZ.
WHOOP-DE-DOO FOR YOU.
WHAT DID YOU GET?
NONE OF YOUR DARN BUSINESS!
YOU GOT A FROWNY FACE, DIDN'T YOU?
NO, AS A MATTER OF FACT, I DIDN'T!
I DIDN'T EVEN KNOW THEY MADE BARFING FACE STICKERS.
10-9

Quiz

Binary Search Trees

BST – A binary tree where each each node has a key, and every node's key is:

- Larger than all keys in its left subtree. (everything left is smaller)
- Smaller than all keys in its right subtree. (everything right is larger)



Operations

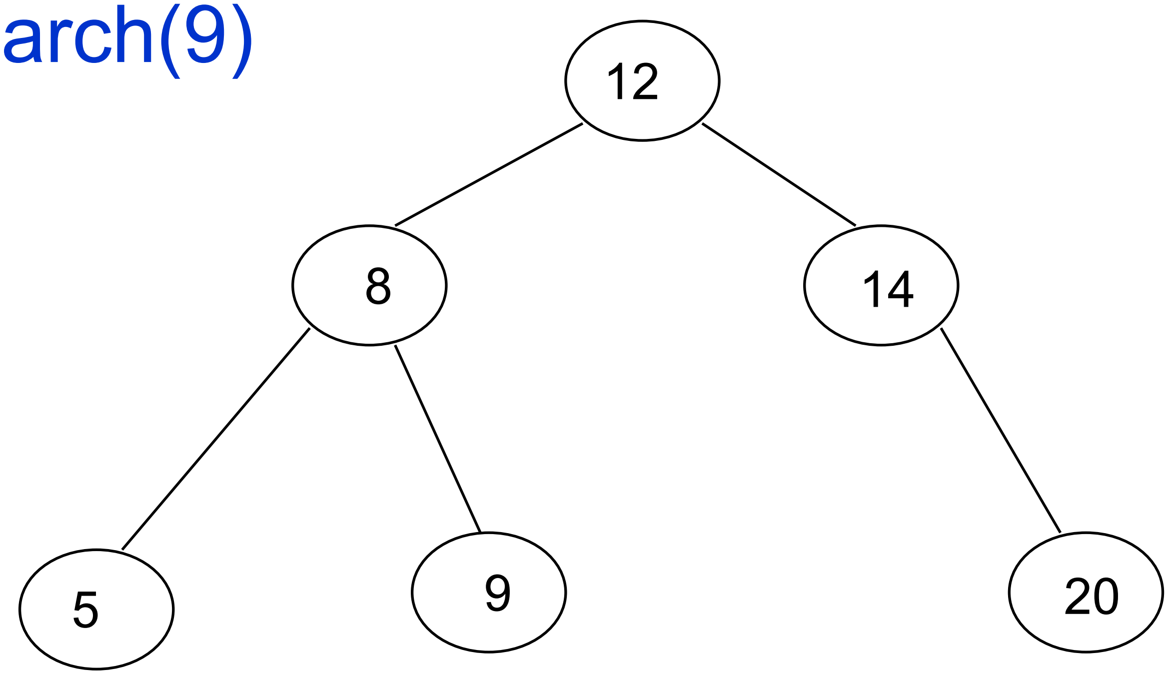
Search – Does the key exist in the tree

Insert – Insert the key into tree

Delete – Delete the key from the tree

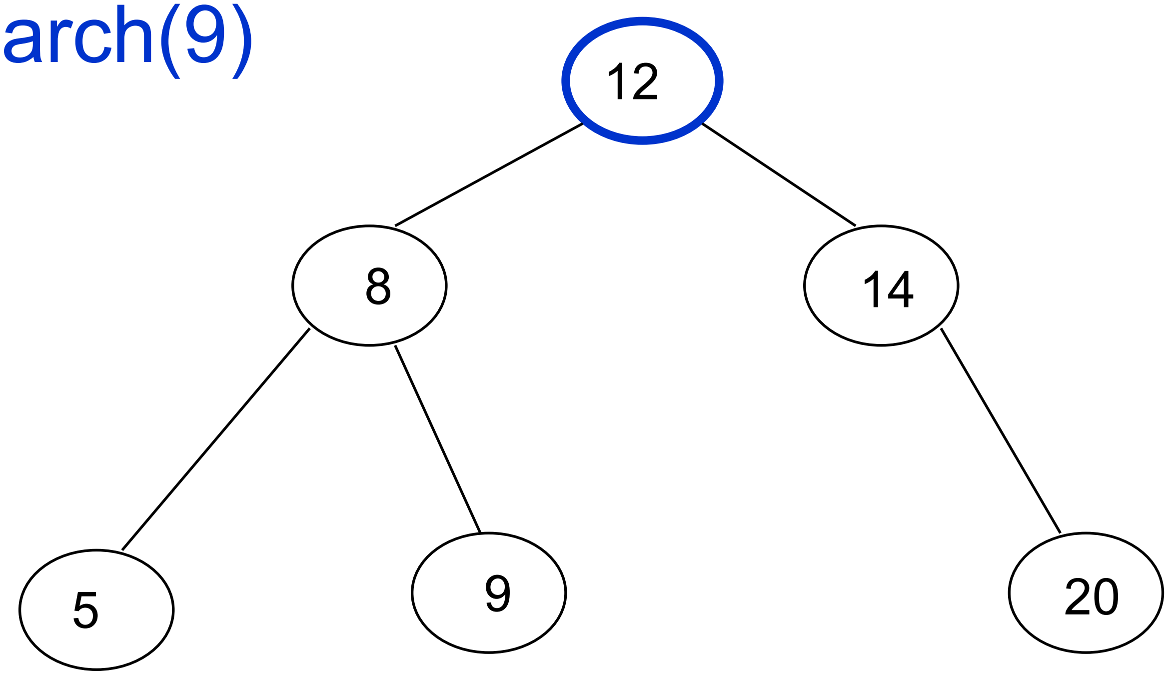
Finding an element

Search(9)



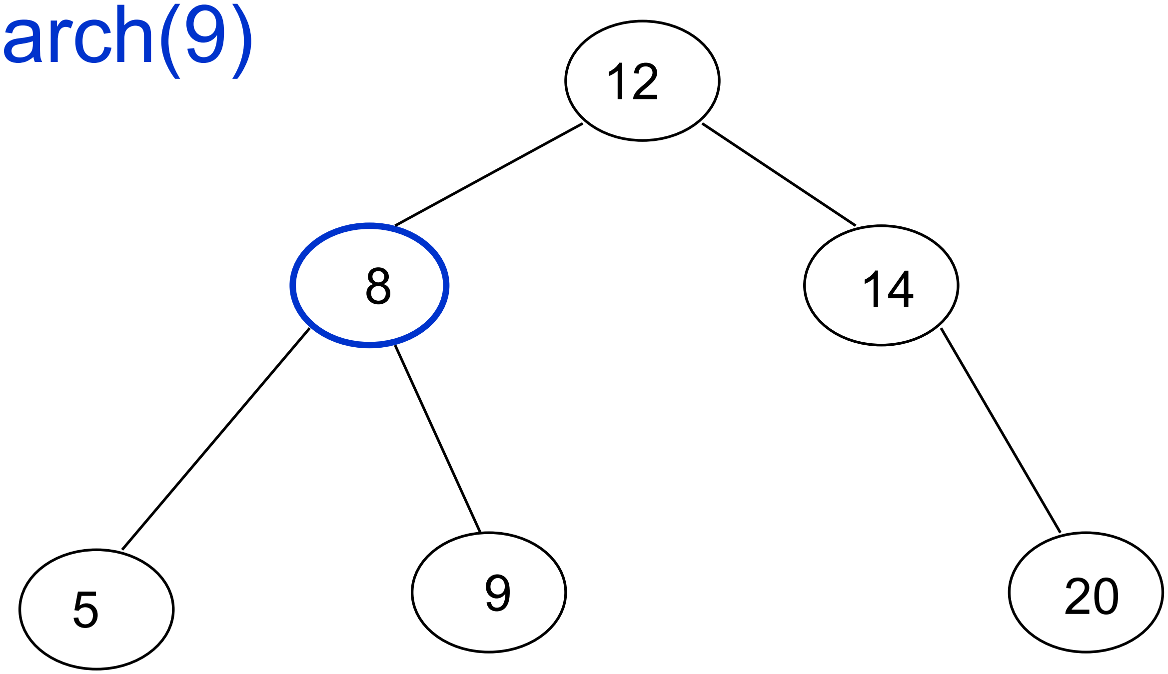
Finding an element

Search(9)



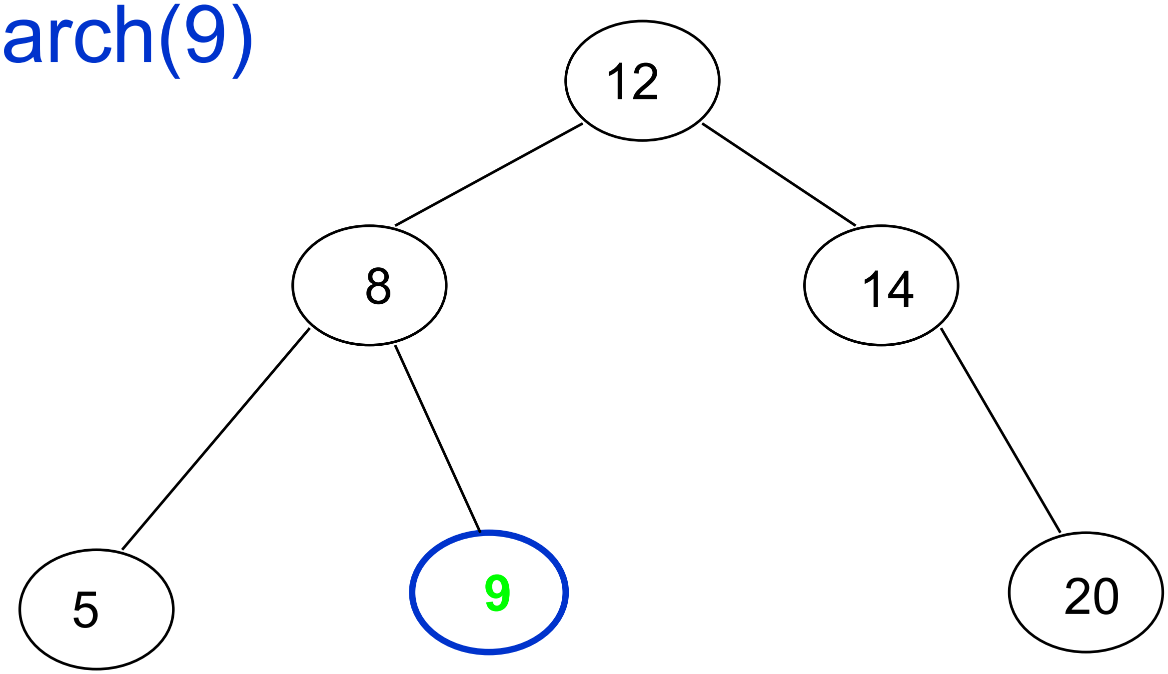
Finding an element

Search(9)



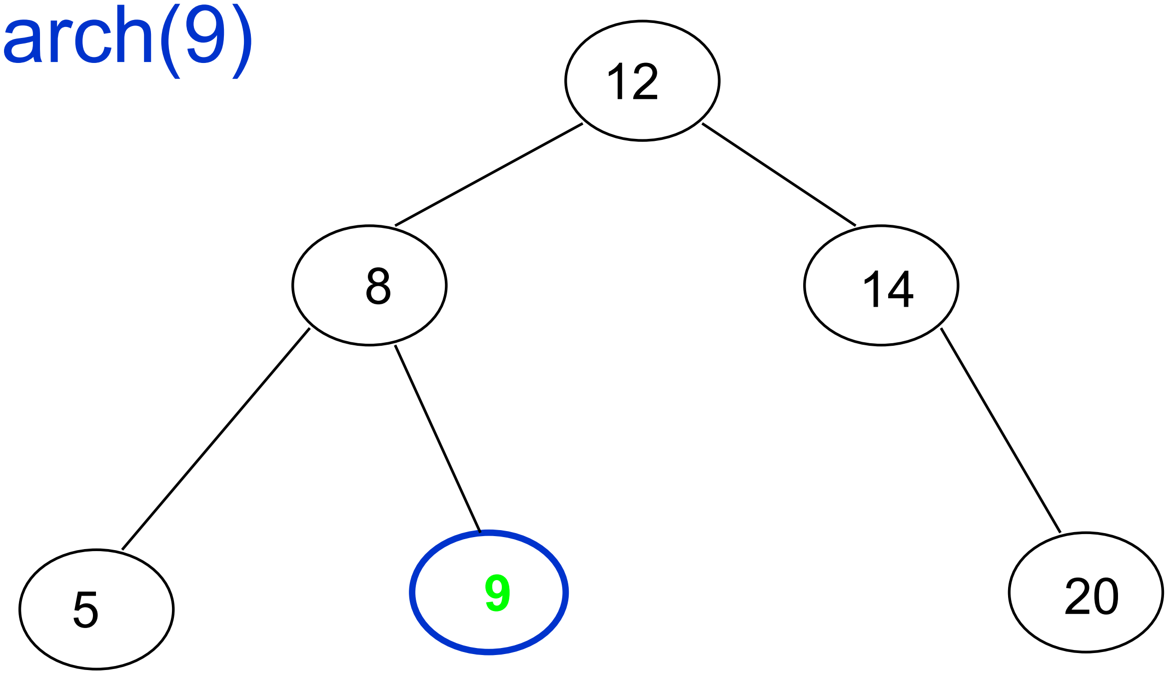
Finding an element

Search(9)



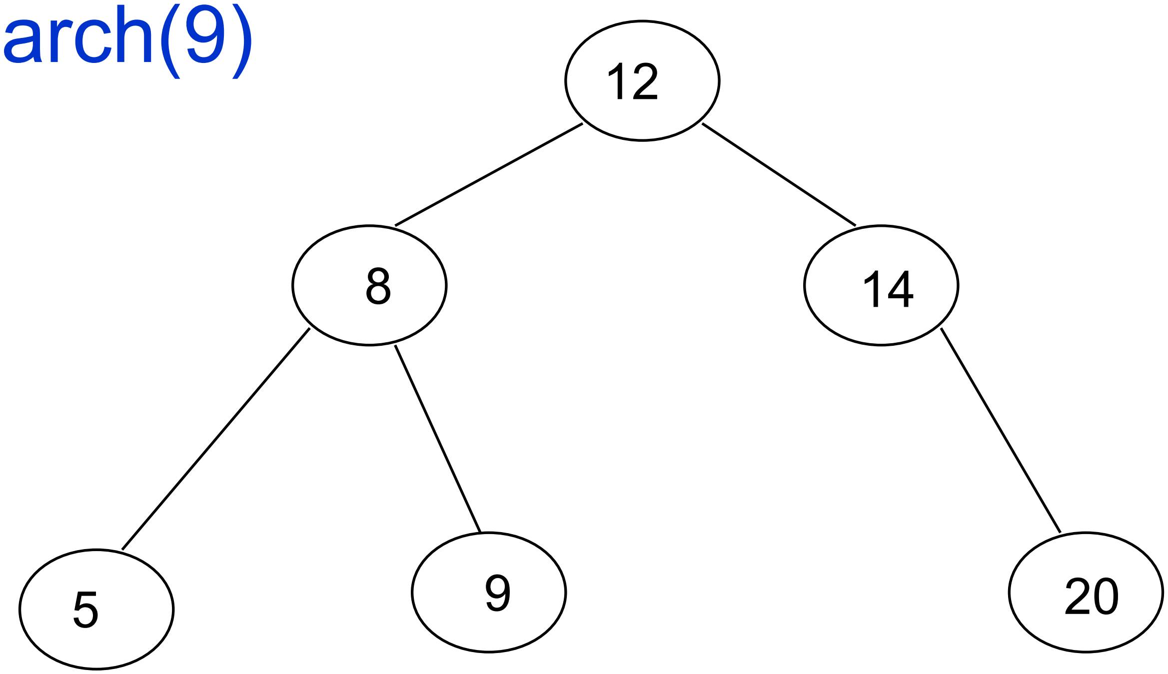
Finding an element

Search(9)



Finding an element

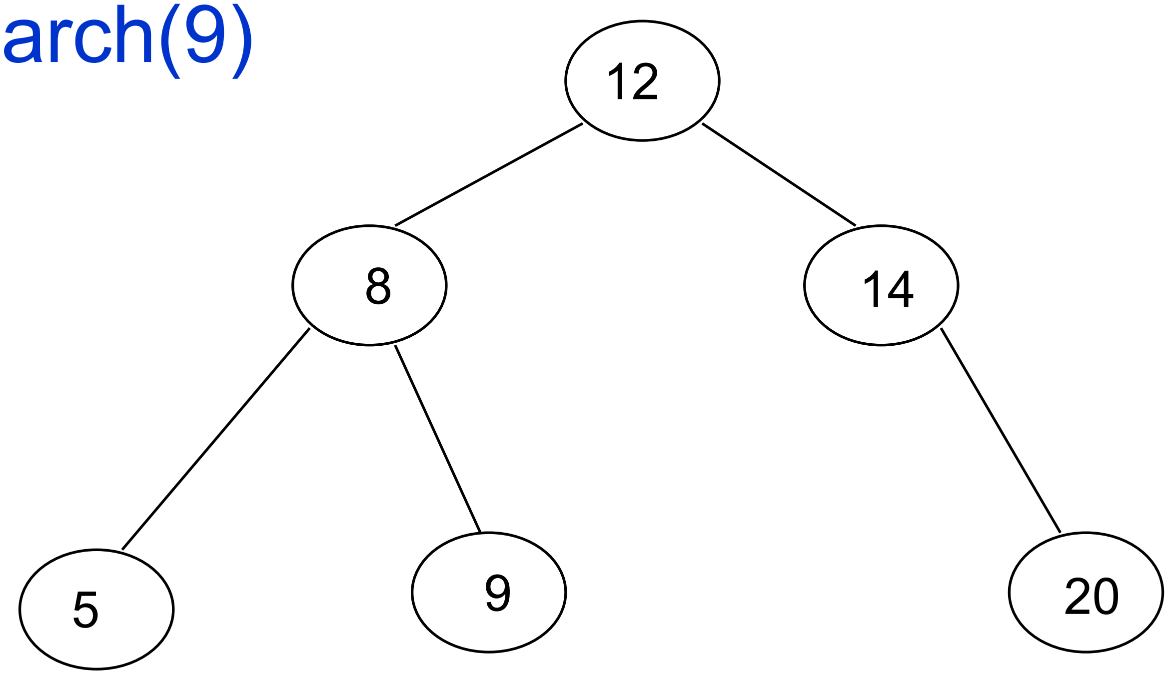
Search(9)



What is the worst case running time of search?

Finding an element

Search(9)

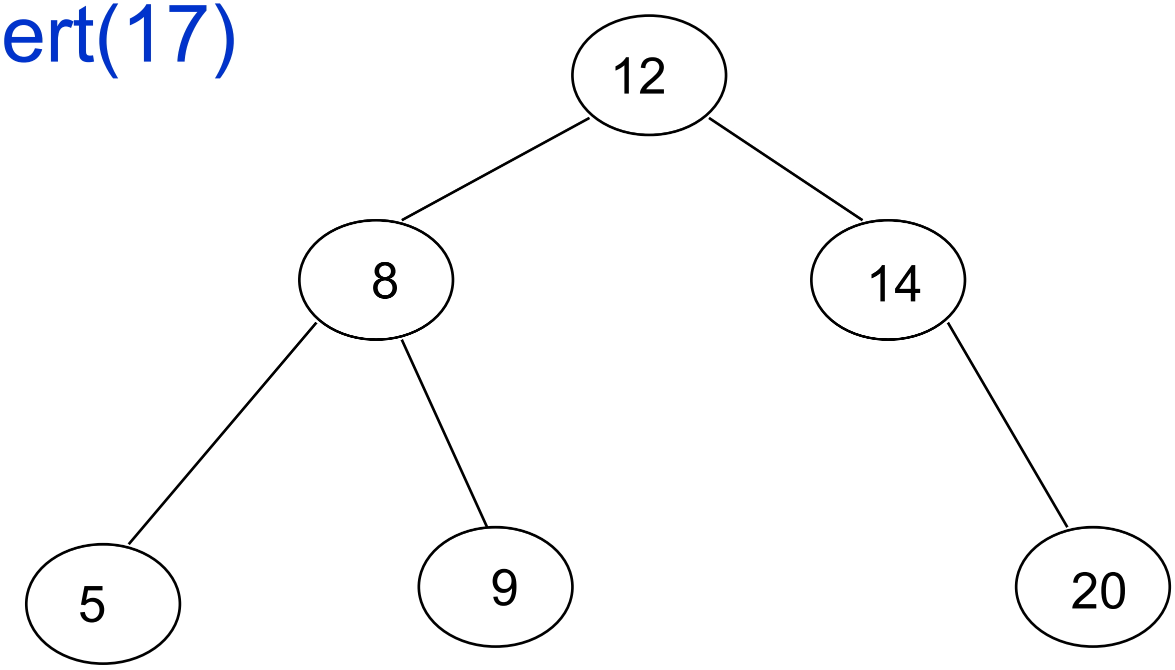


Worst case, have to search to the lowest leaf

O(height)

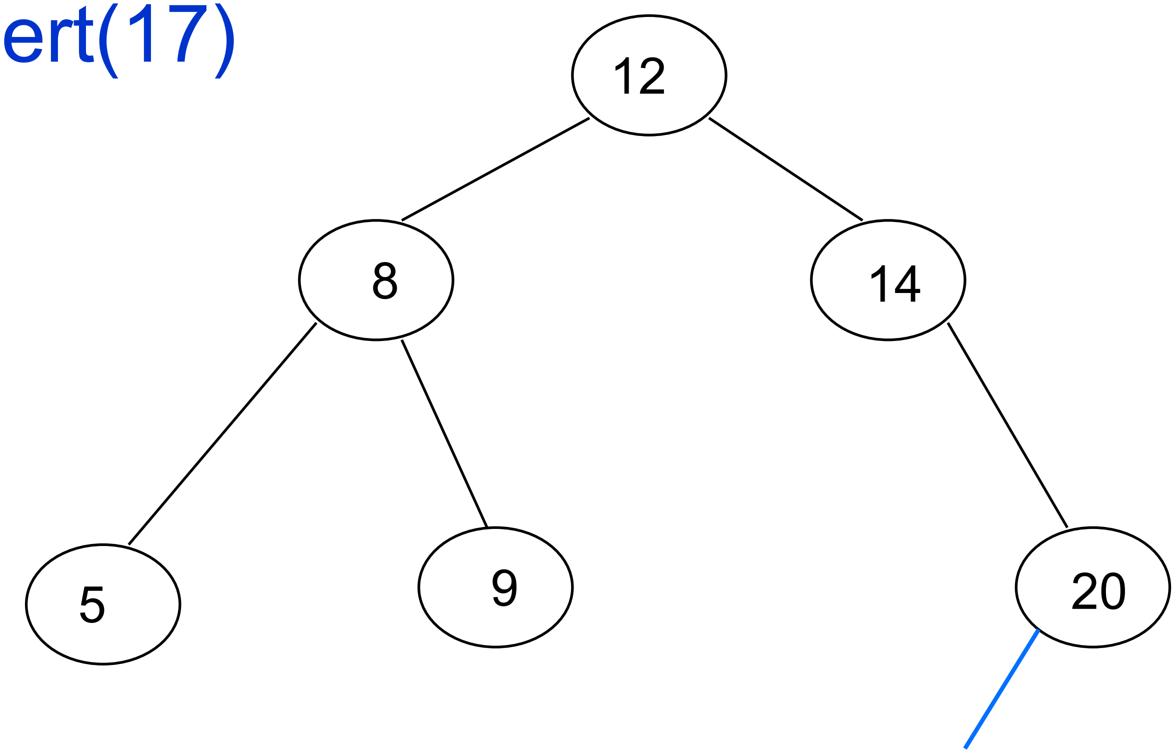
Inserting

Insert(17)



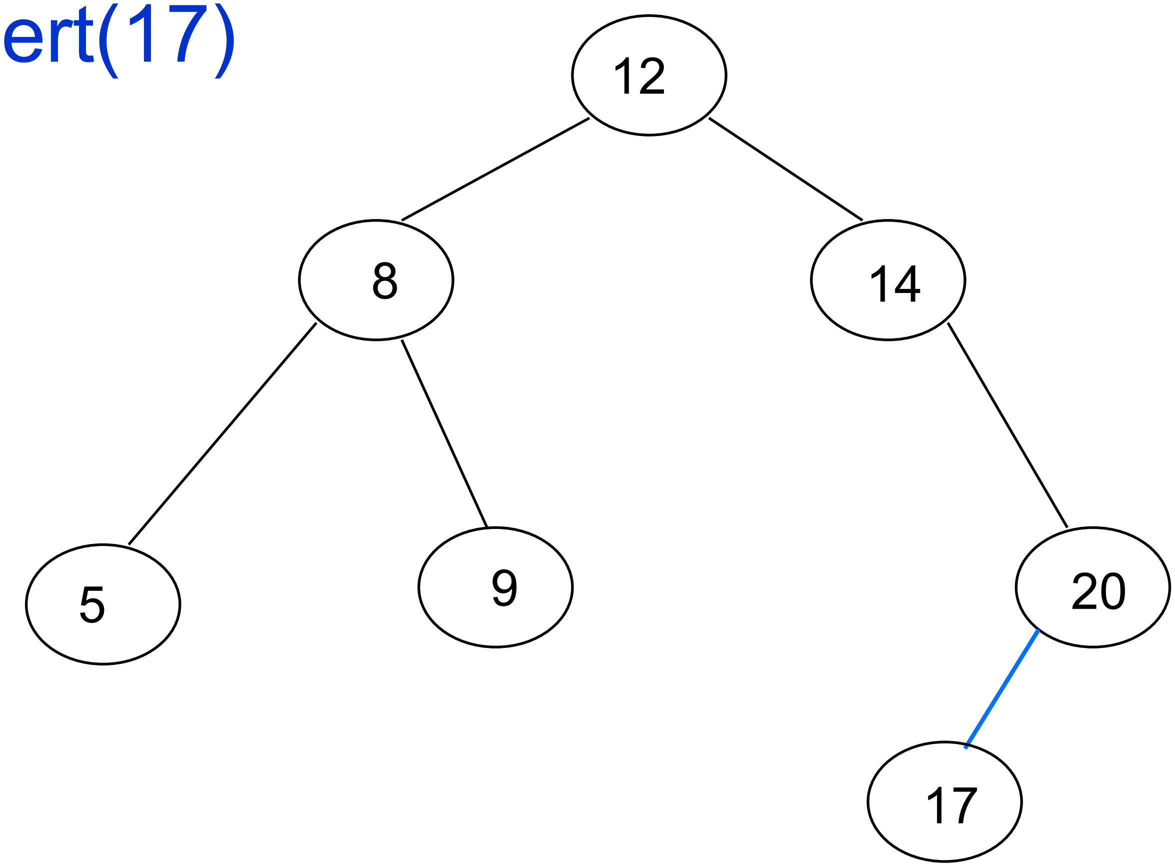
Inserting

Insert(17)



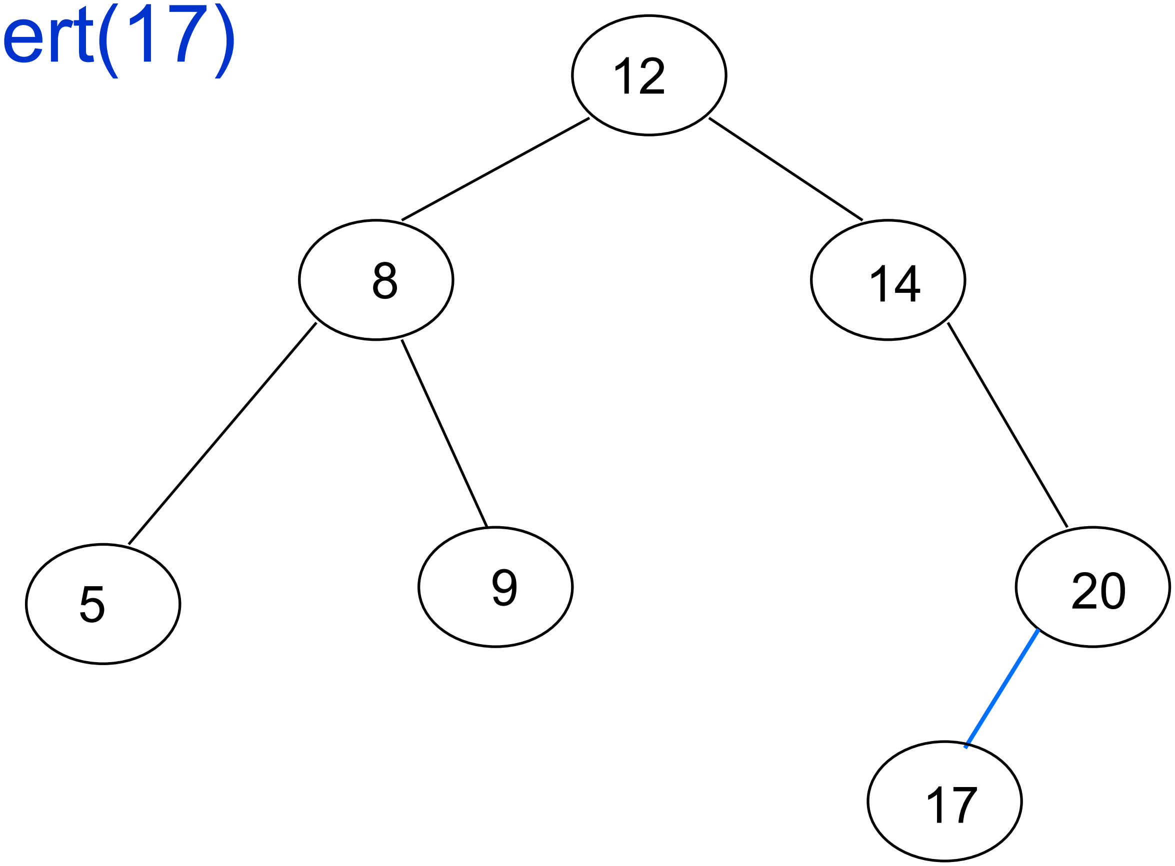
Inserting

Insert(17)



Inserting

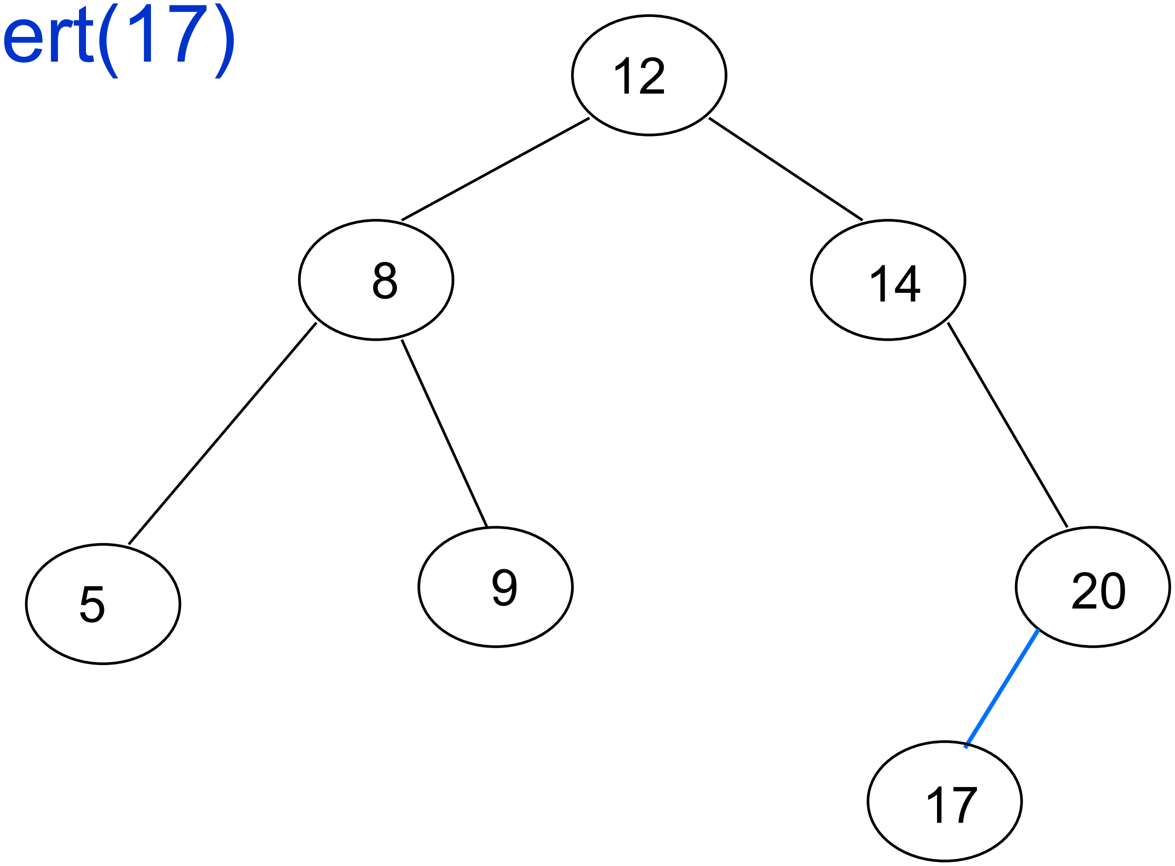
Insert(17)



What is the worst case running time of search?

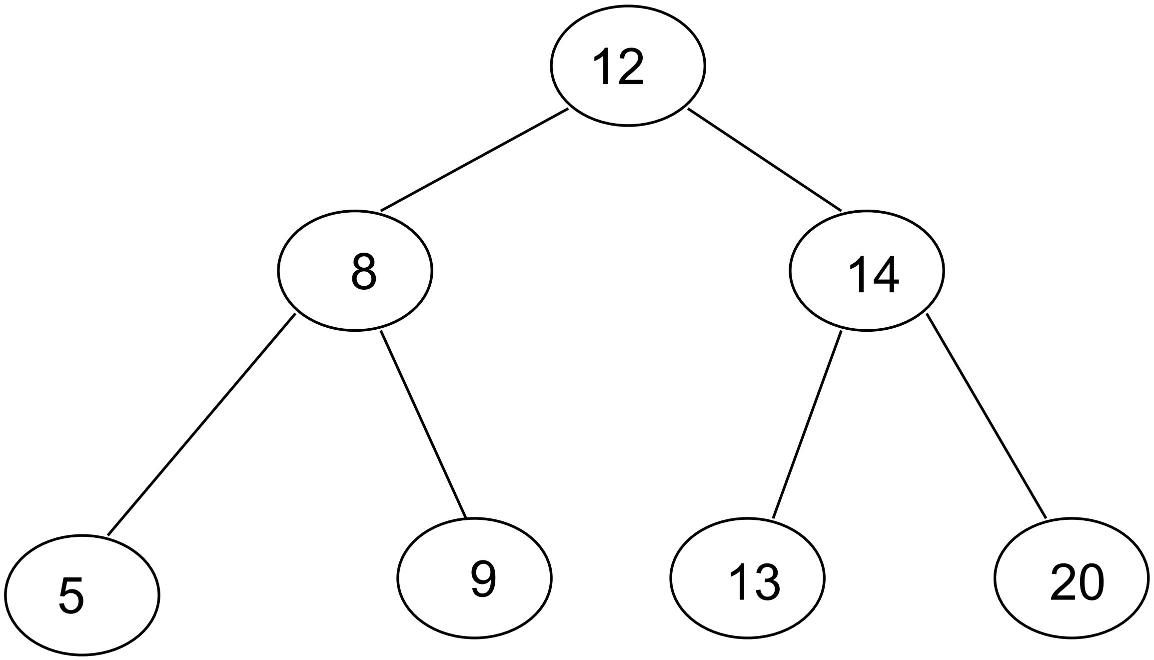
Inserting

Insert(17)



Worst case, have to search to the lowest leaf
O(height)

Deletion

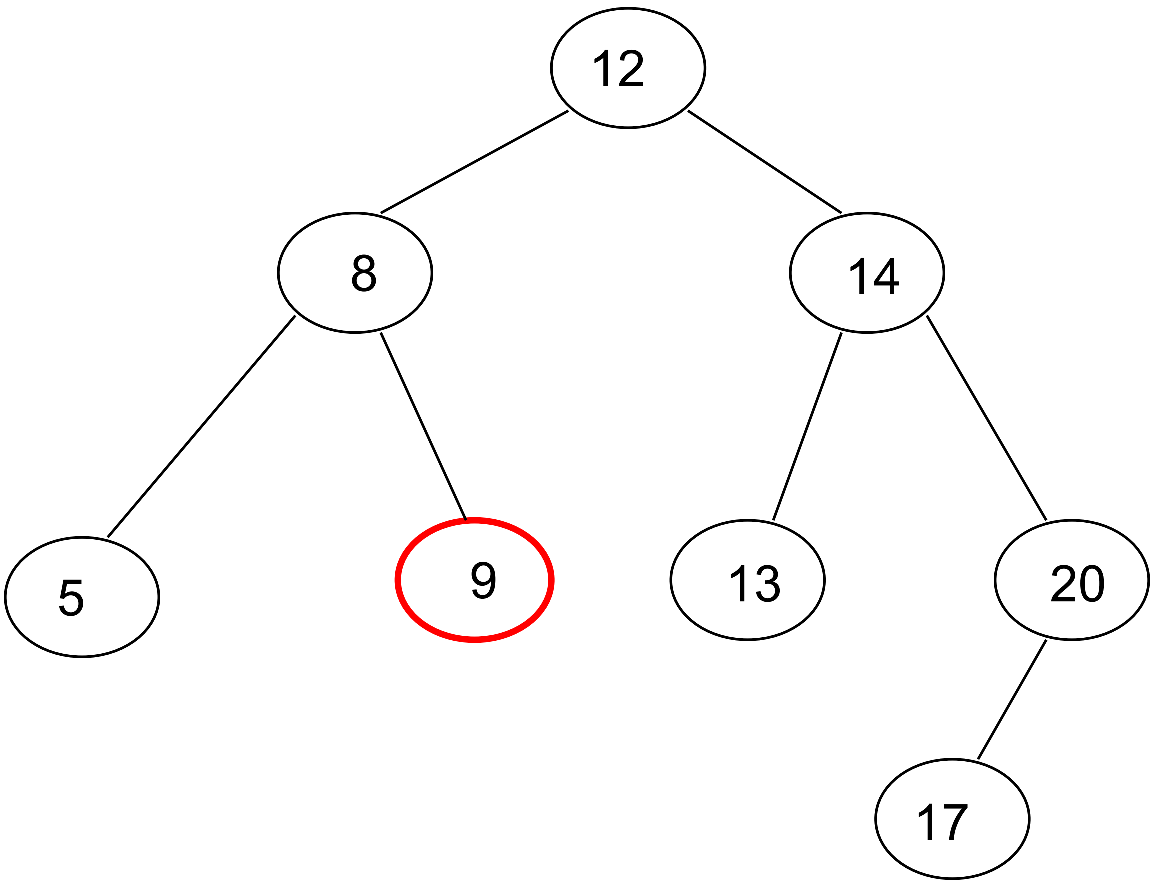


Three cases!

Deletion: case 1

No children

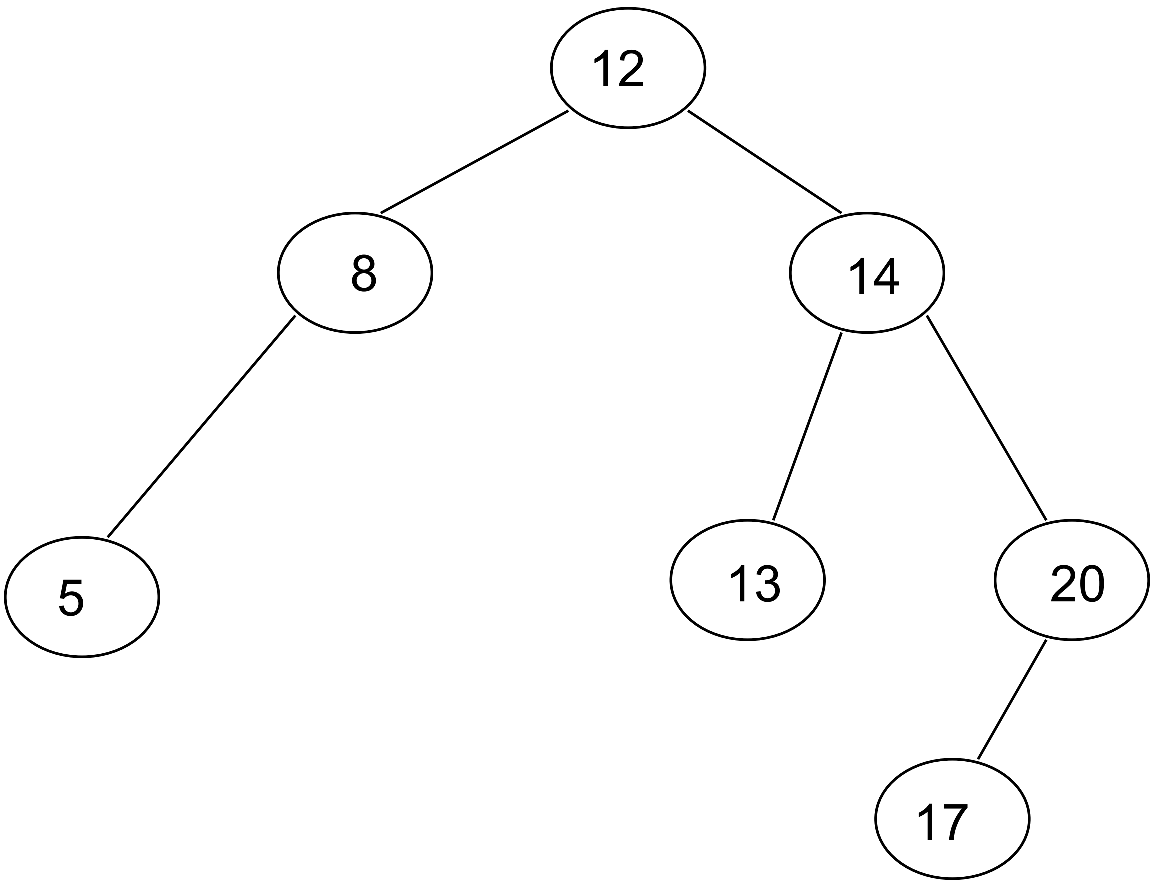
Just delete the node



Deletion: case 1

No children

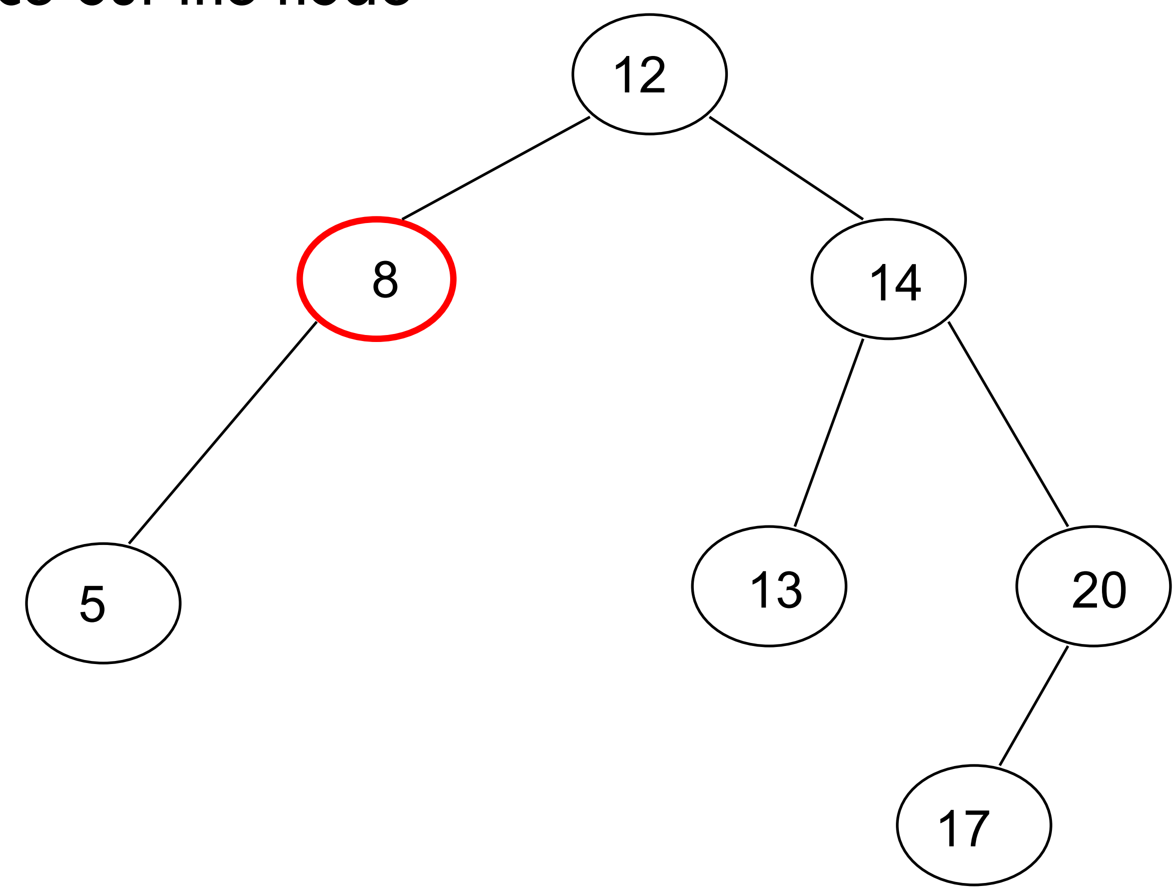
Just delete the node



Deletion: case 2

One child

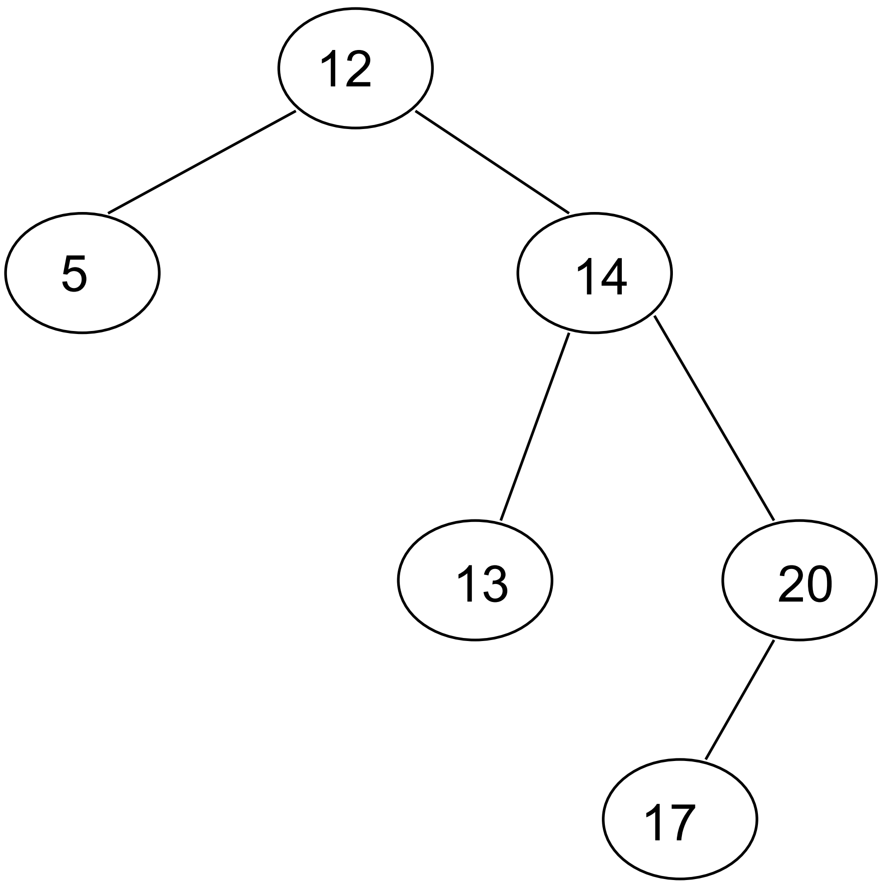
Splice out the node



Deletion: case 2

One child

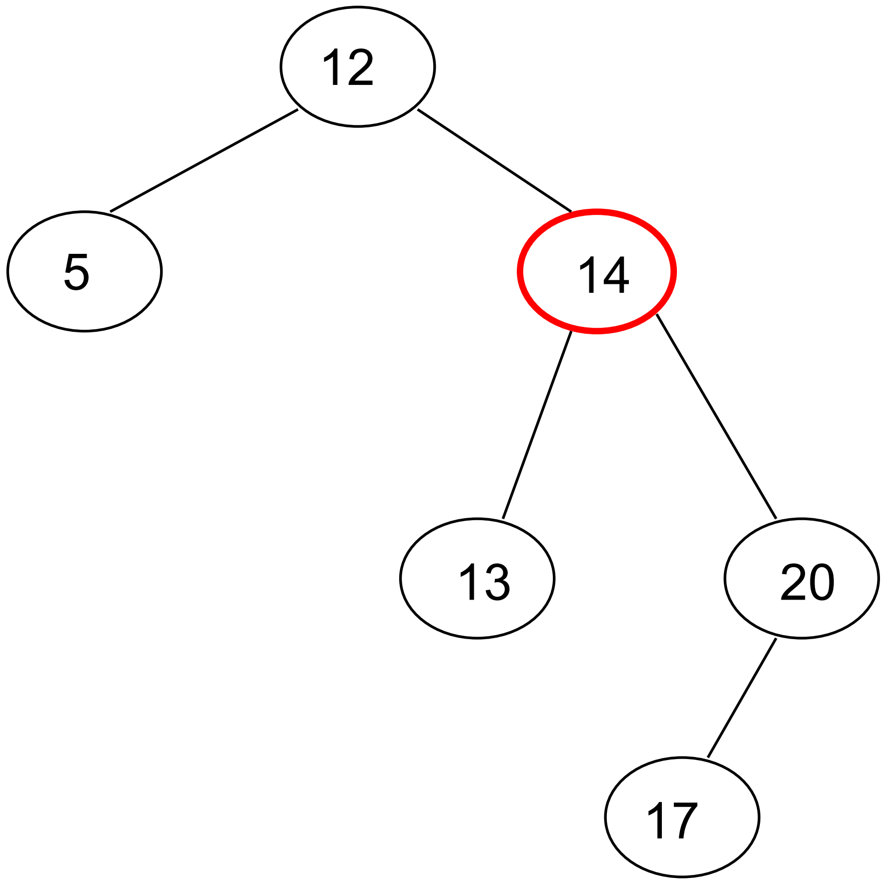
Splice out the node



Deletion: case 3

Two children

Replace x with the smallest value of the right subtree

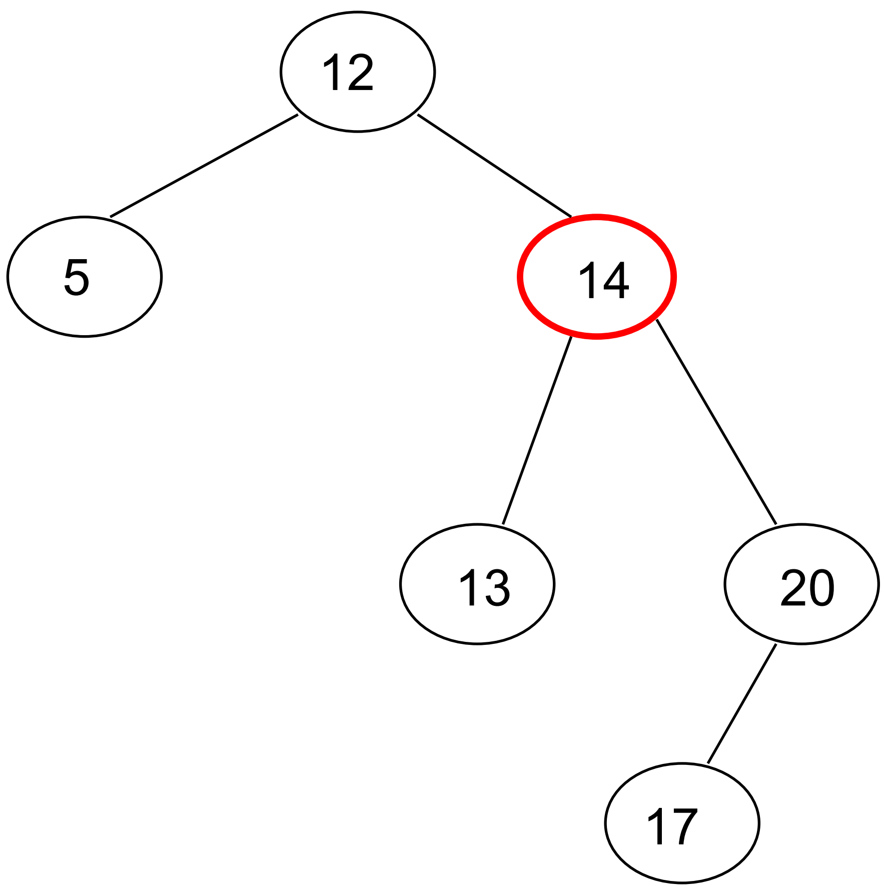


How does this maintain the search tree property?

Deletion: case 3

Two children

Replace x with the smallest value of the right subtree

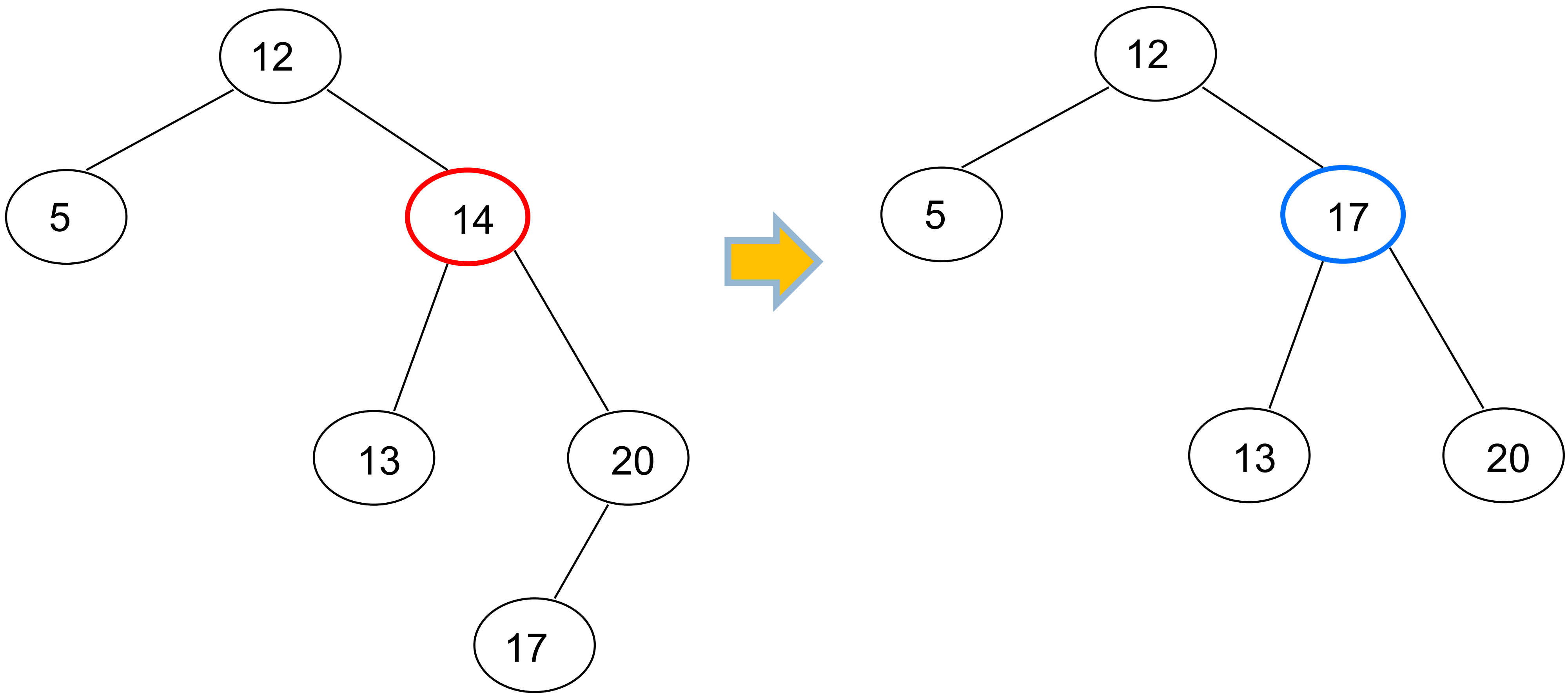


- Larger than everything to the left
- Smaller than everything to the right

Deletion: case 3

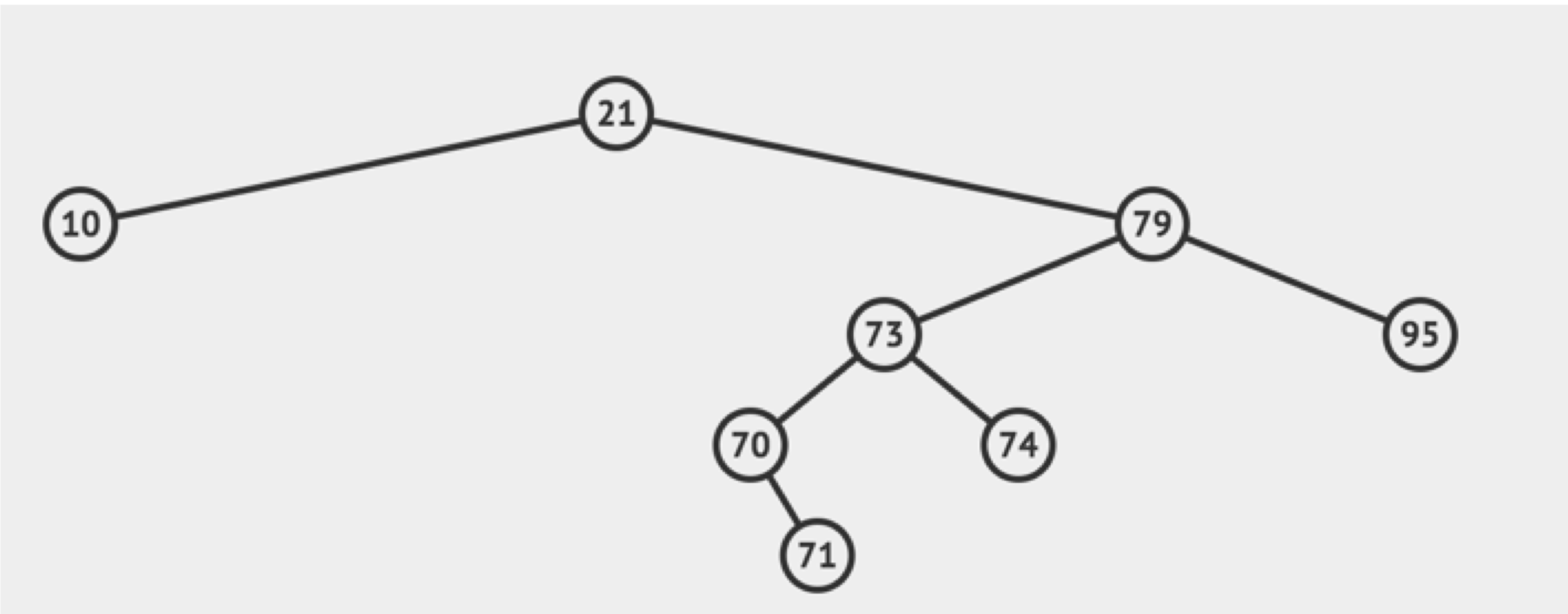
Two children

Replace x with the smallest value of the right subtree



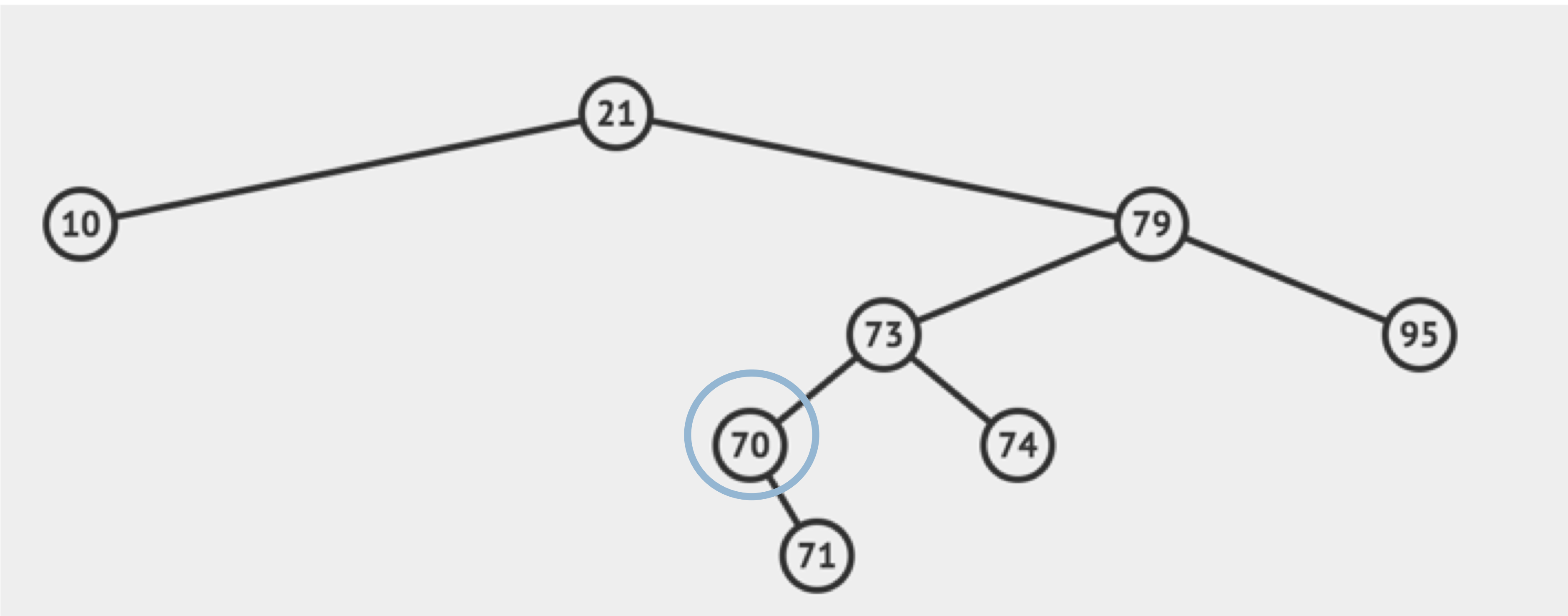
Deletion

Delete 21



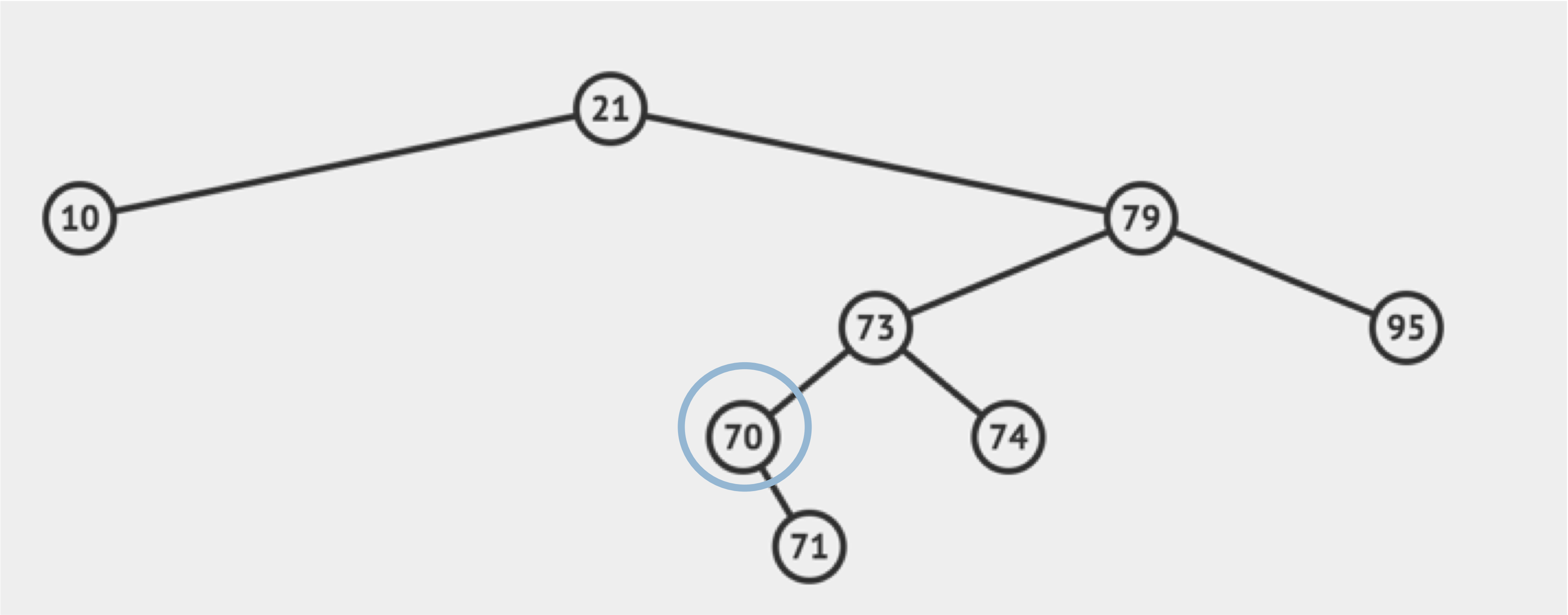
Deletion

Min of the right subtree



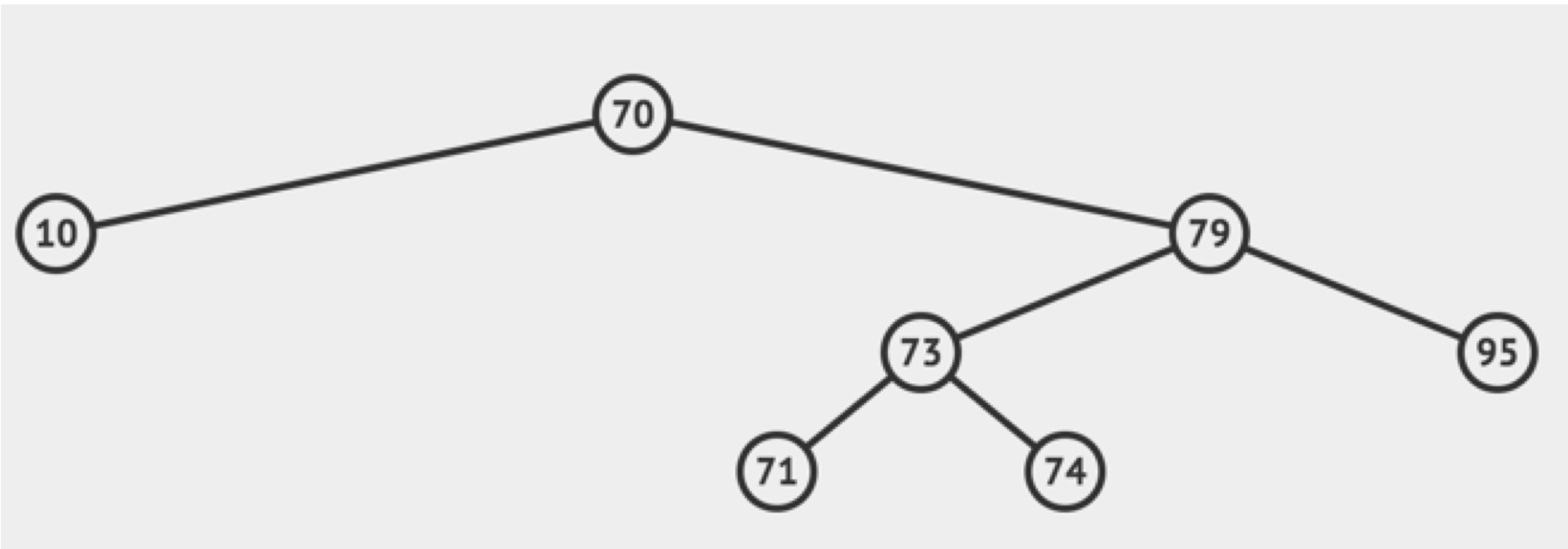
Deletion

Replace the value: involves a case 2 deletion



Deletion

Replace the value: involves a case 2 deletion



Deletion: case 3

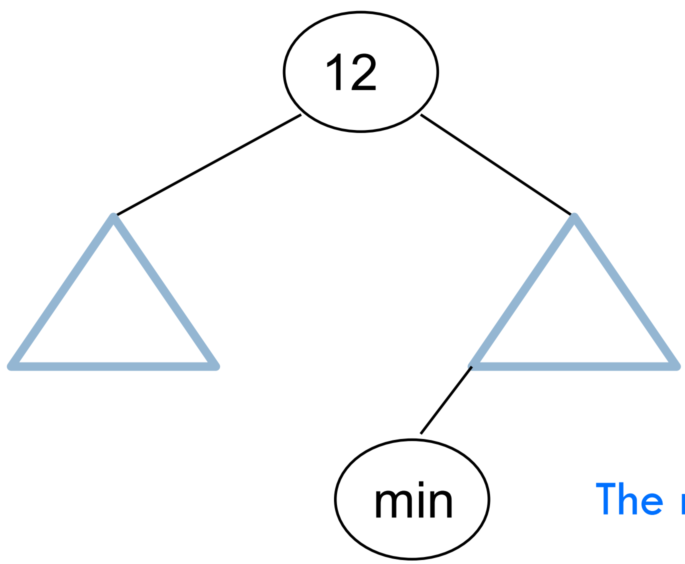
The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

Why?

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

Why?

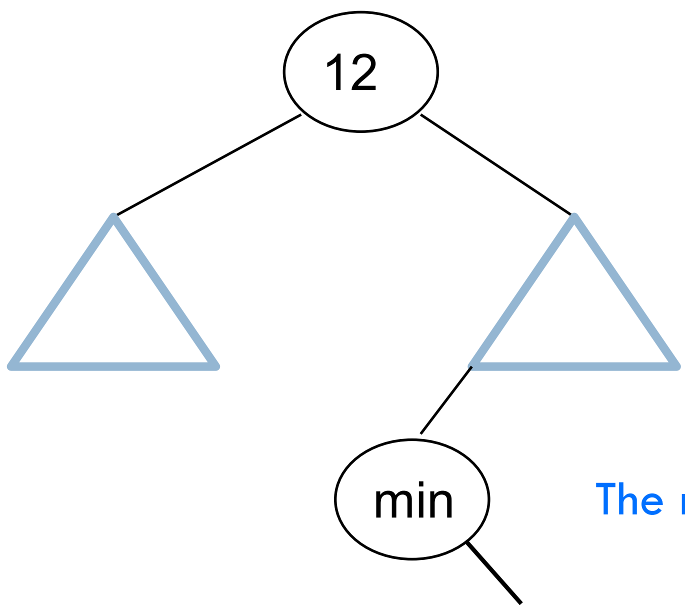


The minimum cannot have a left child

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion

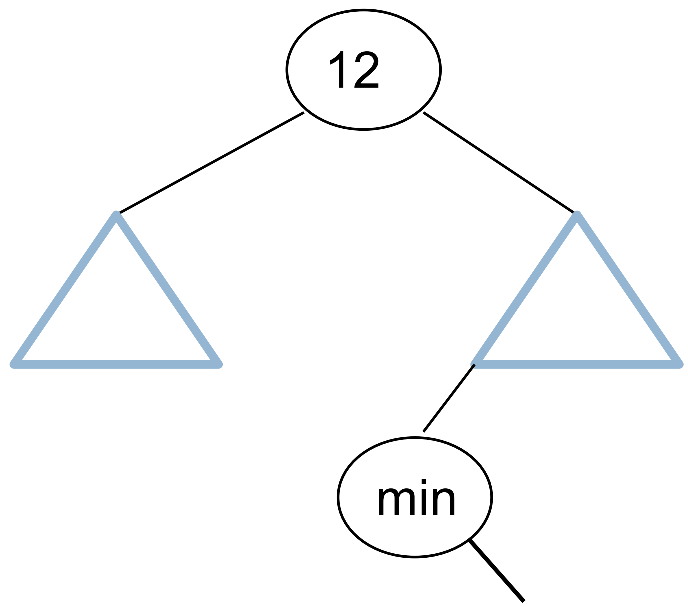
Why?



The minimum cannot have a left child

Deletion: case 3

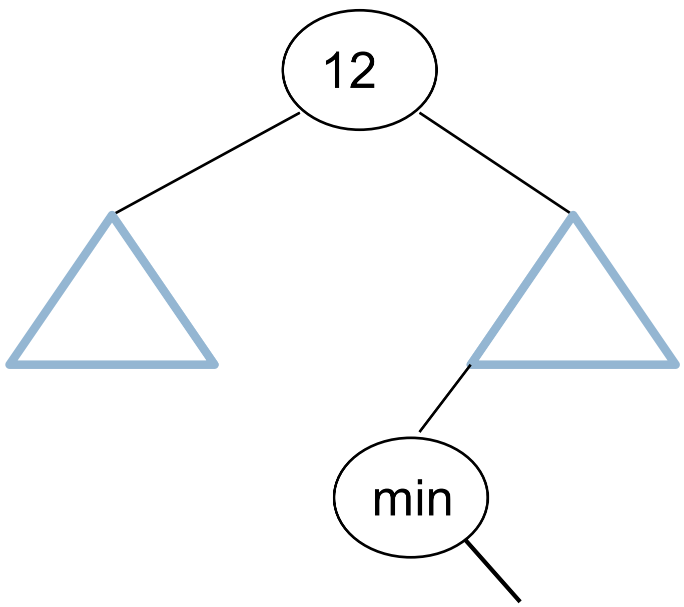
The min of the right subtree will always be either a case 1 deletion or a case 2 deletion



What is the worst case running time of delete?

Deletion: case 3

The min of the right subtree will always be either a case 1 deletion or a case 2 deletion



Case 1 and Case 2: O(1)
Case 3: Find min and do a case 1 or case 2 delete
O(height)

Delete implemented

```
public void delete(Key key) {
    root = delete(root, key);
}

 private Node delete(Node x, Key key) {
     if (x == null) return null;

     int cmp = key.compareTo(x.key);
     if (cmp < 0)
         x.left  = delete(x.left,  key);
     else if (cmp > 0)
         x.right = delete(x.right, key);
     else {
         if (x.right == null)
             return x.left;
         if (x.left  == null)
             return x.right;
         Node t = x; //replace with successor
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
     }
     x.size = size(x.left) + size(x.right) + 1;
     return x;
 }
```

Delete implemented

```
public void delete(Key key) {
    root = delete(root, key);
}

 private Node delete(Node x, Key key) {
     if (x == null) return null;

     int cmp = key.compareTo(x.key);
     if (cmp < 0)
         x.left  = delete(x.left,  key);
     else if (cmp > 0)
         x.right = delete(x.right, key);
     else {
         if (x.right == null)
             return x.left;
         if (x.left  == null)
             return x.right;
         Node t = x; //replace with successor
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
     }
     x.size = size(x.left) + size(x.right) + 1;
     return x;
 }
```

Search: find the key

Delete implemented

```
public void delete(Key key) {
    root = delete(root, key);
}

 private Node delete(Node x, Key key) {
     if (x == null) return null;

     int cmp = key.compareTo(x.key);
     if (cmp < 0)
         x.left  = delete(x.left,  key);
     else if (cmp > 0)
         x.right = delete(x.right, key);
     else {
         if (x.right == null)
             return x.left;
         if (x.left  == null)
             return x.right;
         Node t = x; //replace with successor
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
     }
     x.size = size(x.left) + size(x.right) + 1;
     return x;
 }
```

Case 1 and Case 2

Delete implemented

```
public void delete(Key key) {
    root = delete(root, key);
}

 private Node delete(Node x, Key key) {
     if (x == null) return null;

     int cmp = key.compareTo(x.key);
     if (cmp < 0)
         x.left  = delete(x.left,  key);
     else if (cmp > 0)
         x.right = delete(x.right, key);
     else {
         if (x.right == null)
             return x.left;
         if (x.left  == null)
             return x.right;
         Node t = x; //replace with successor
         x = min(t.right);
         x.right = deleteMin(t.right);
         x.left = t.left;
     }
     x.size = size(x.left) + size(x.right) + 1;
     return x;
 }
```

Case 3

Height of the tree

Most of the operations take time
O(height)

We said trees built from random data have height
O(log n), which is asymptotically tight

Two problems:

- We can't always insure random data
- What happens when we delete nodes and insert others after building a tree?

Worst case height for binary search trees is O(n) ☹

Operations

Search – Does the key exist in the tree

Insert – Insert the key into tree

Delete – Delete the key from the tree

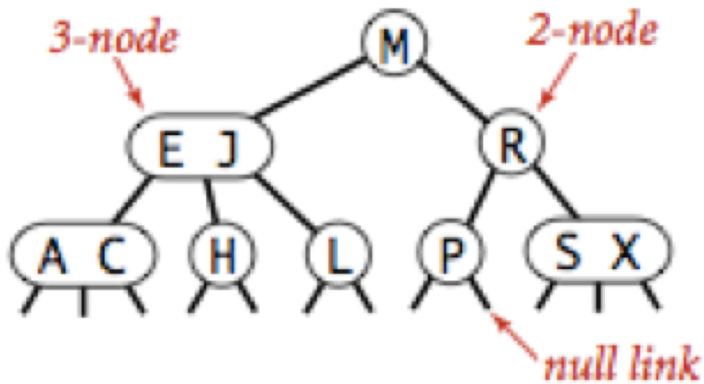
Balanced trees

Make sure that the trees remain balanced!

- Red-black trees
- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- …

Height is guaranteed to be O(log n)

2-3 trees



Anatomy of a 2-3 search tree

2-node: one key and two children (left and right)

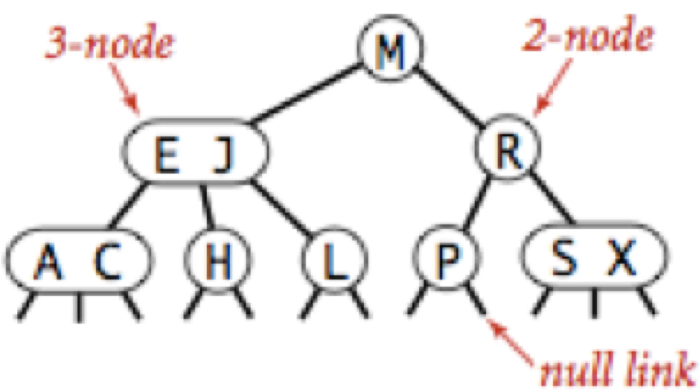
- everything in left is smaller than key
- everything right is larger than key

3-node: two keys (k_1, k_2) and three children, left, middle and right

- $k_1 < k_2$
- everything in left is less than k_1
- everything in middle is between k_1 and k_2
- everything in right is larger than k_2

Search

How do we search for a key?

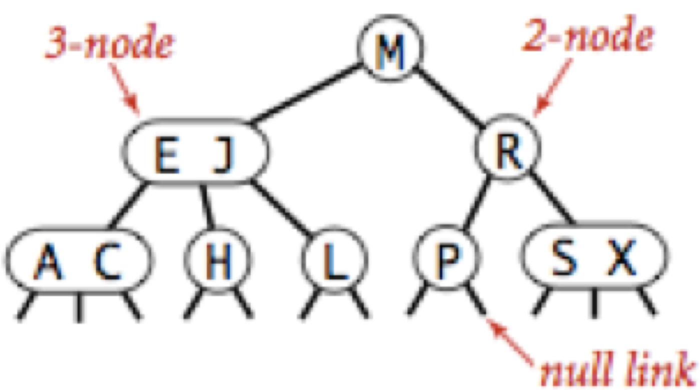


Anatomy of a 2-3 search tree

Search

Almost identical to BST search

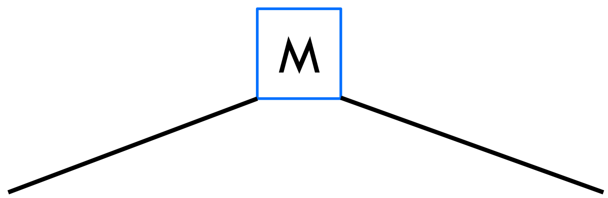
Only difference: sometimes we have two keys



Anatomy of a 2-3 search tree

Search

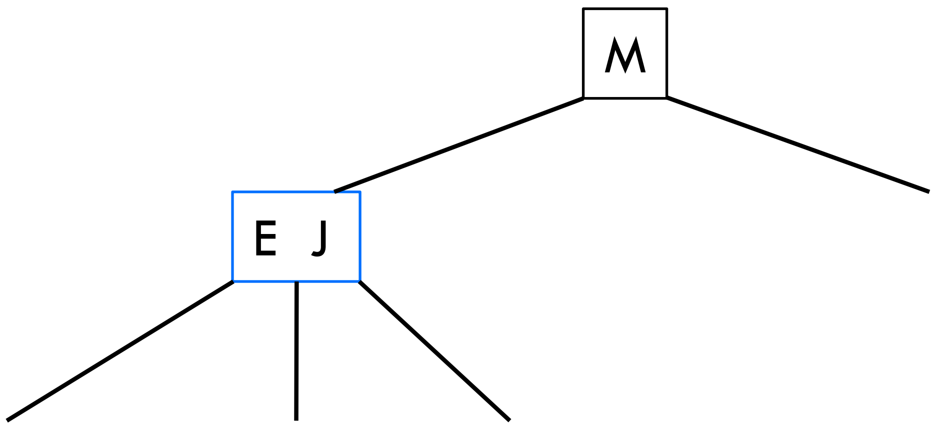
Search(H)



Which child?

Search

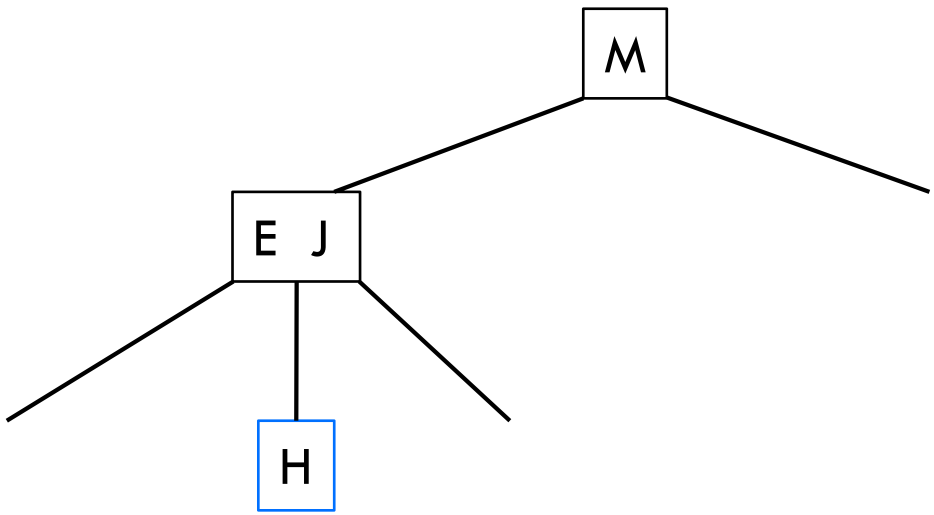
Search(H)



Which child?

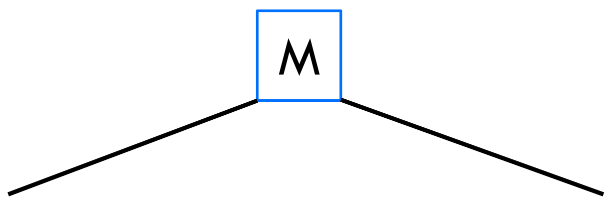
Search

Search(H)



Search

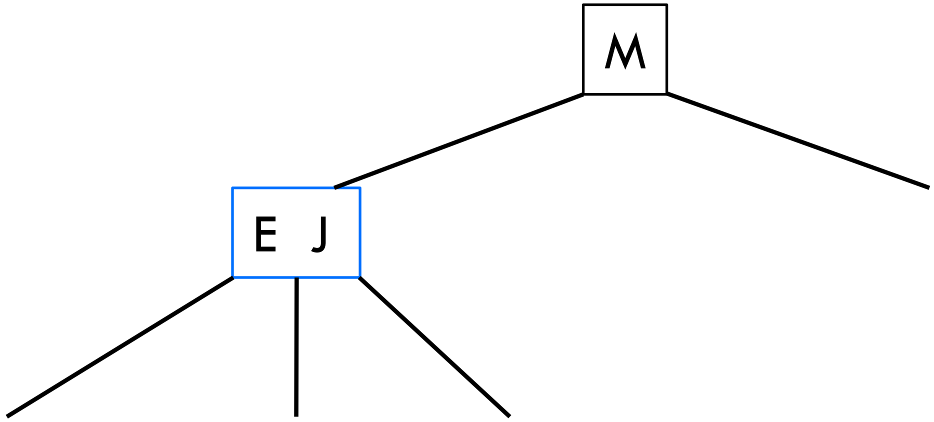
Search(B)



Which child?

Search

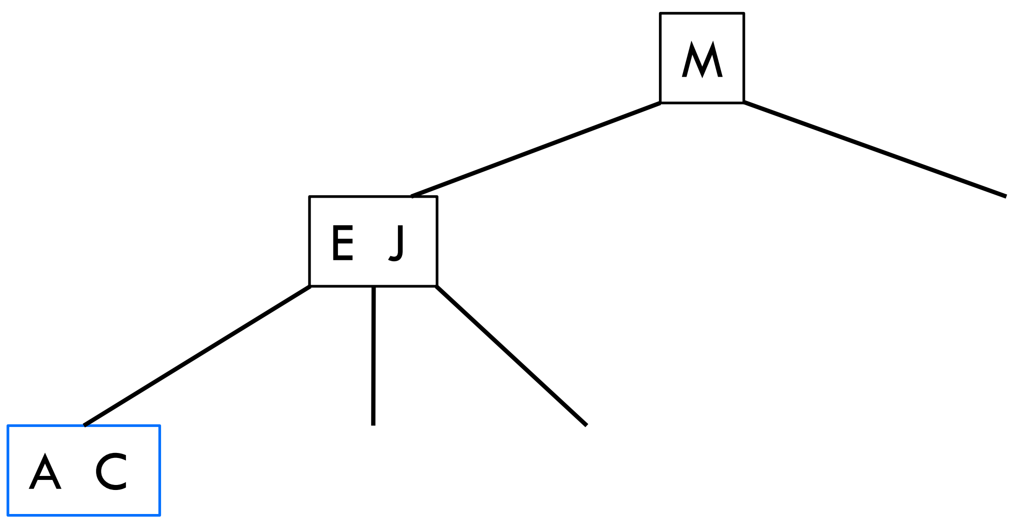
Search(B)



Which child?

Search

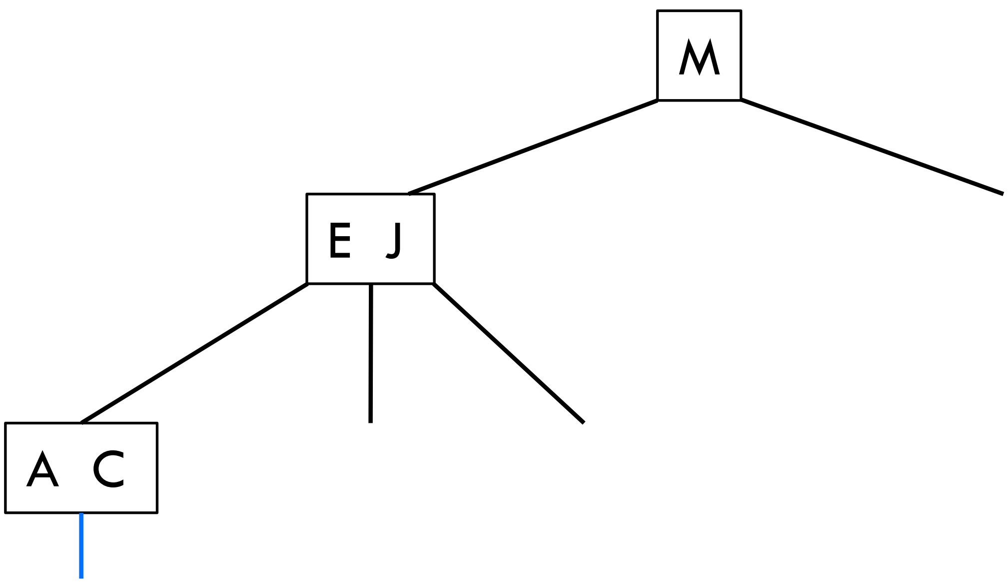
Search(B)



Which child?

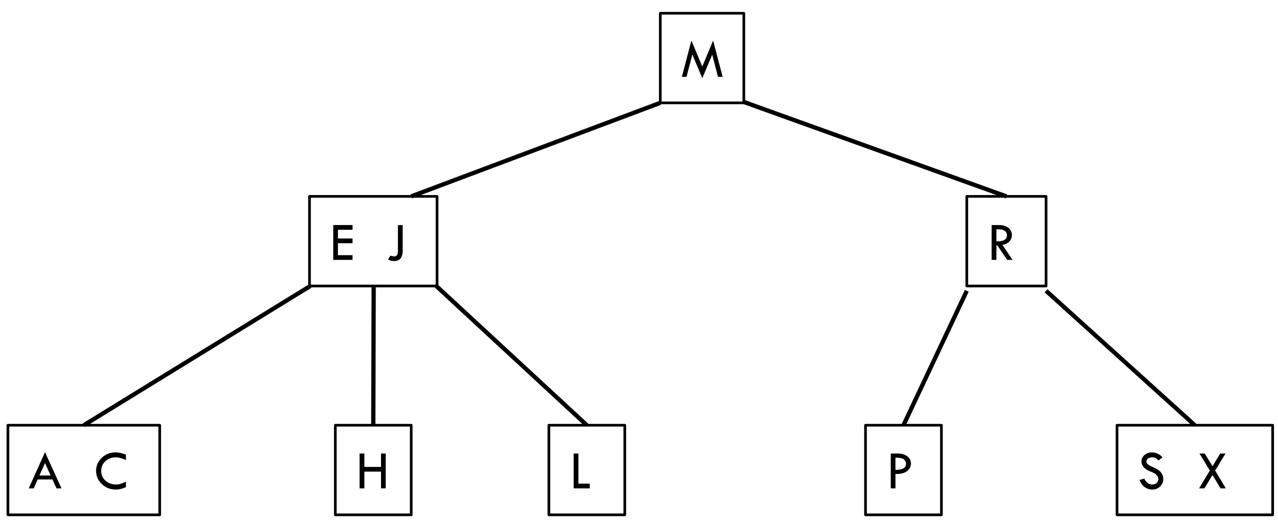
Search

Search(B)



Not found!

Search



Insertion

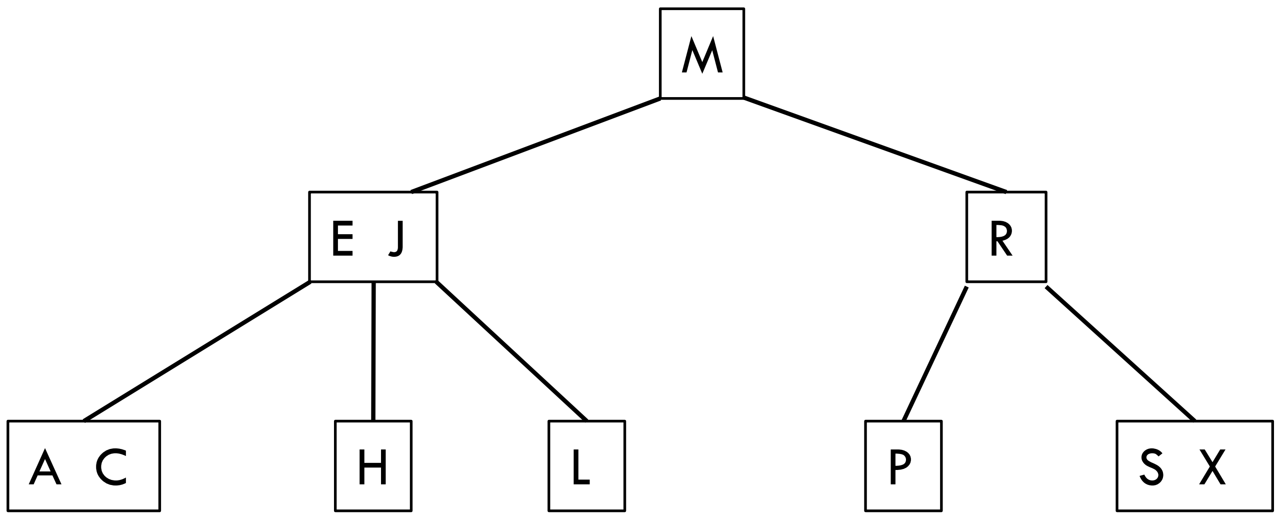
Like BST, insert always happens at a leaf

If the leaf is a 2-node, just insert it directly

Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)

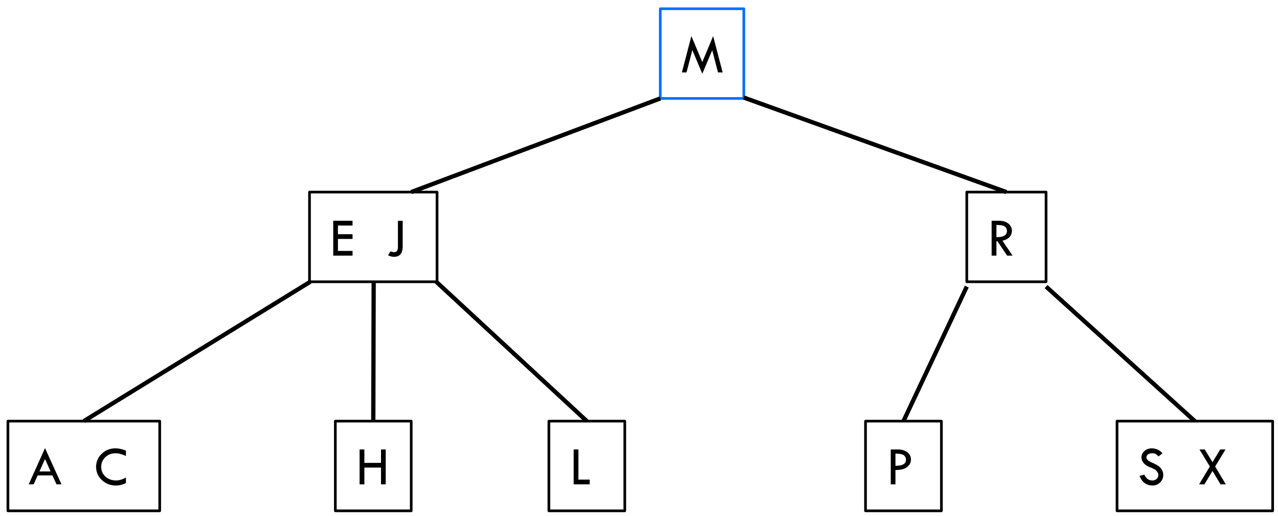


Where should it go?

Insertion

If the leaf is a 2-node, just insert it directly

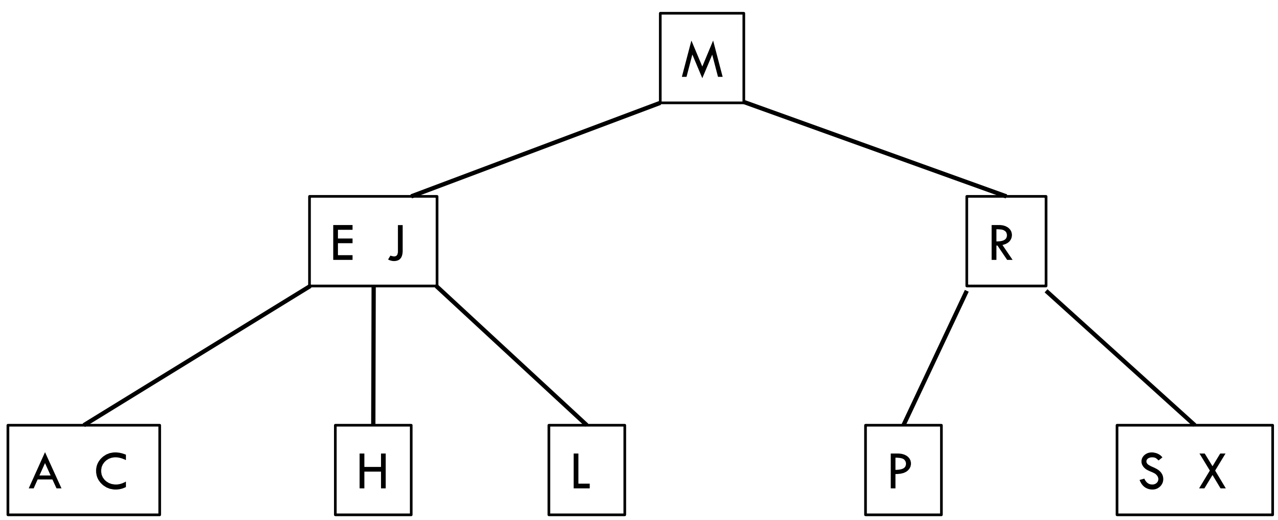
Insert(F)



Insertion

If the leaf is a 2-node, just insert it directly

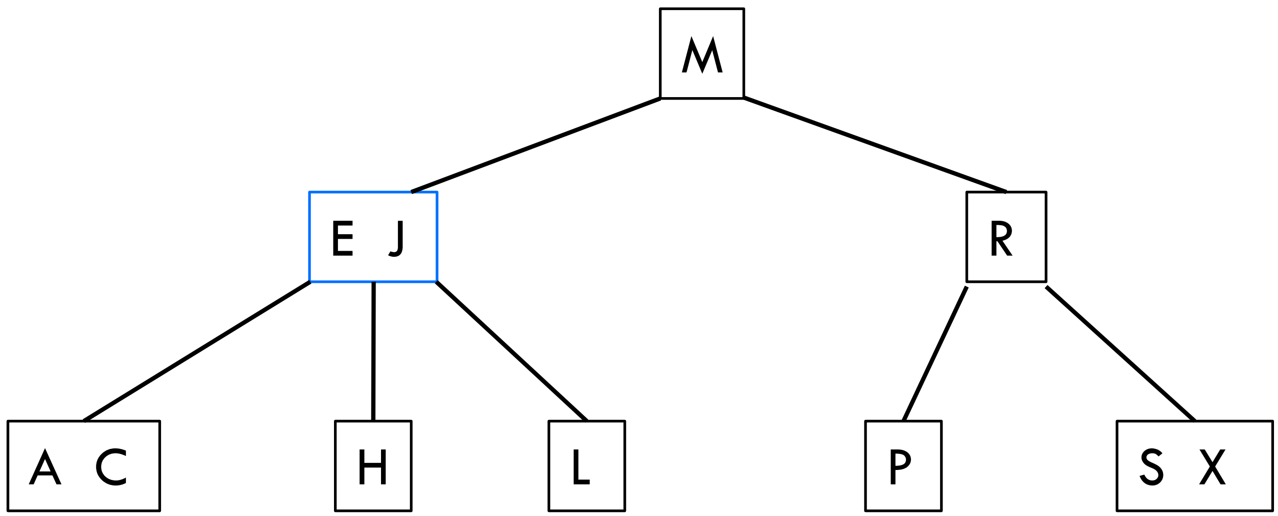
Insert(F)



Insertion

If the leaf is a 2-node, just insert it directly

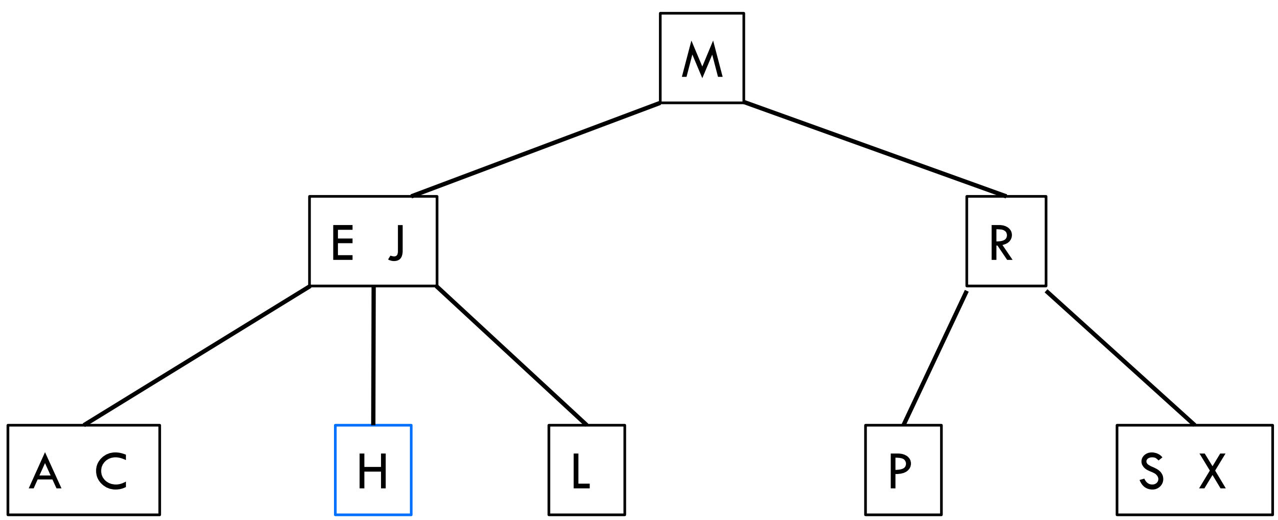
Insert(F)



Insertion

If the leaf is a 2-node, just insert it directly

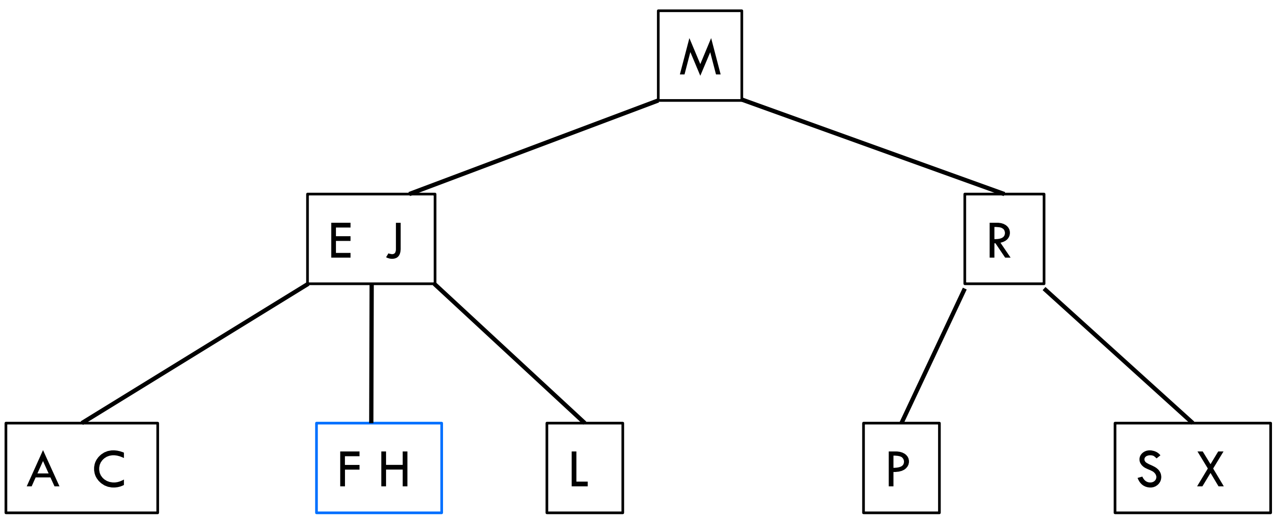
Insert(F)



Insertion

If the leaf is a 2-node, just insert it directly

Insert(F)



Insertion

Like BST, insert always happens at a leaf

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

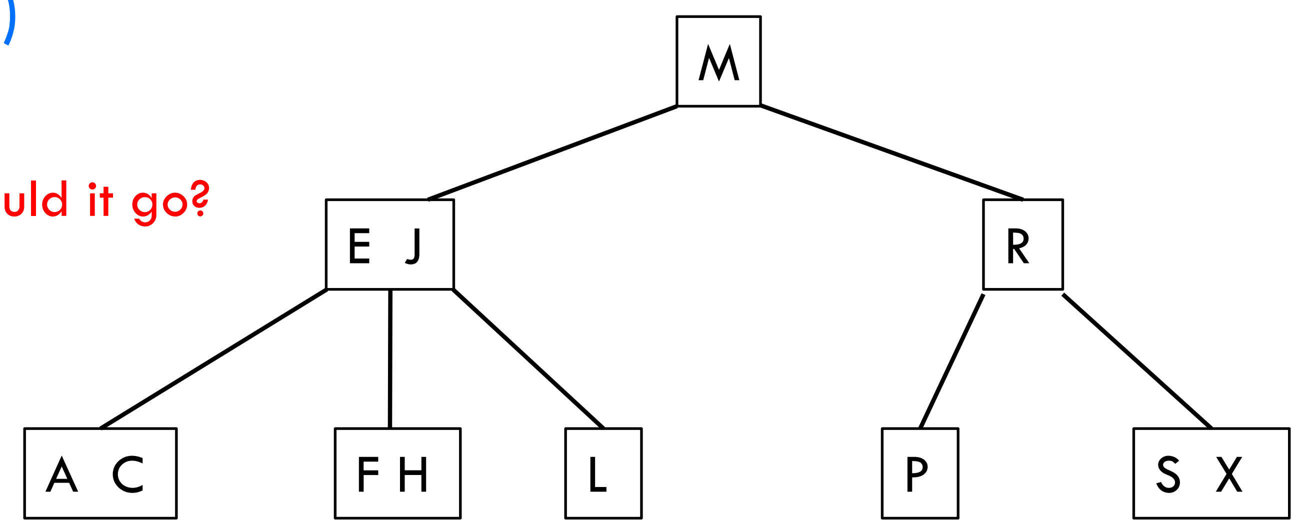
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)

Where should it go?

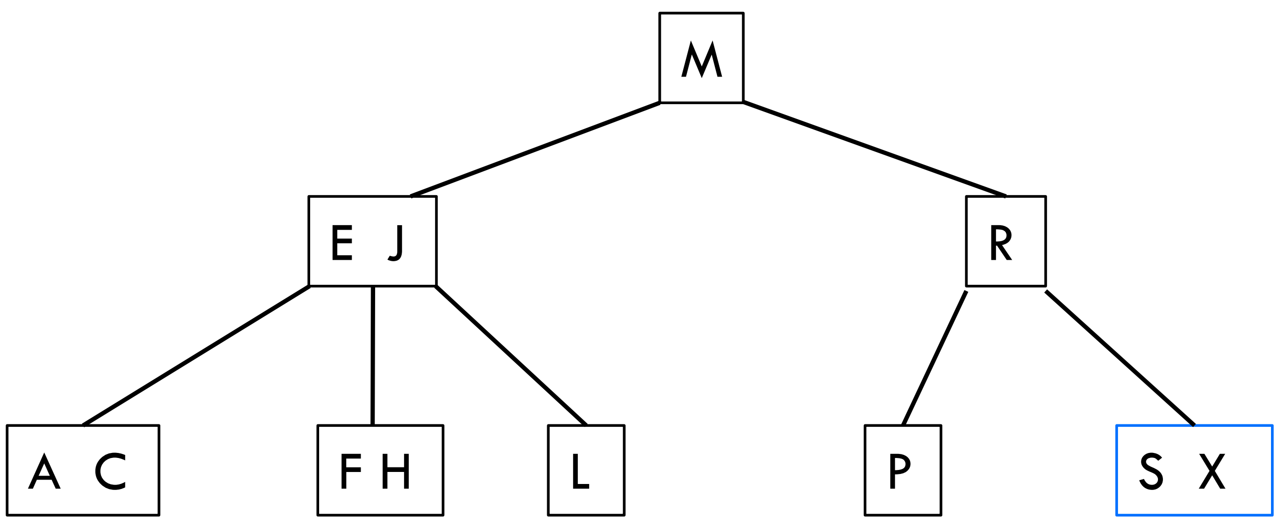


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)

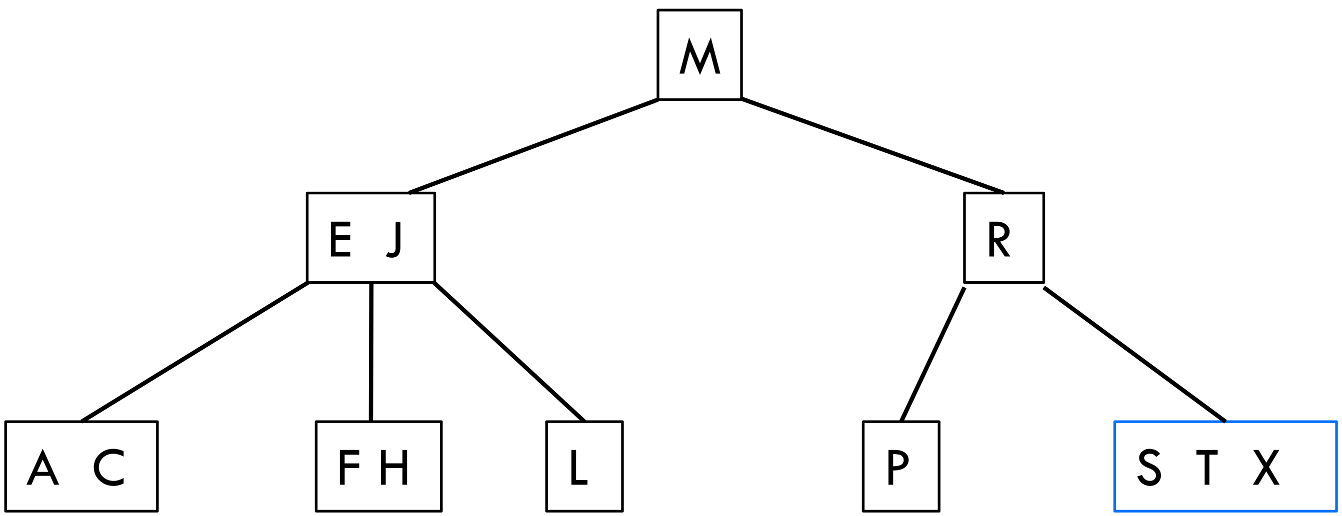


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)

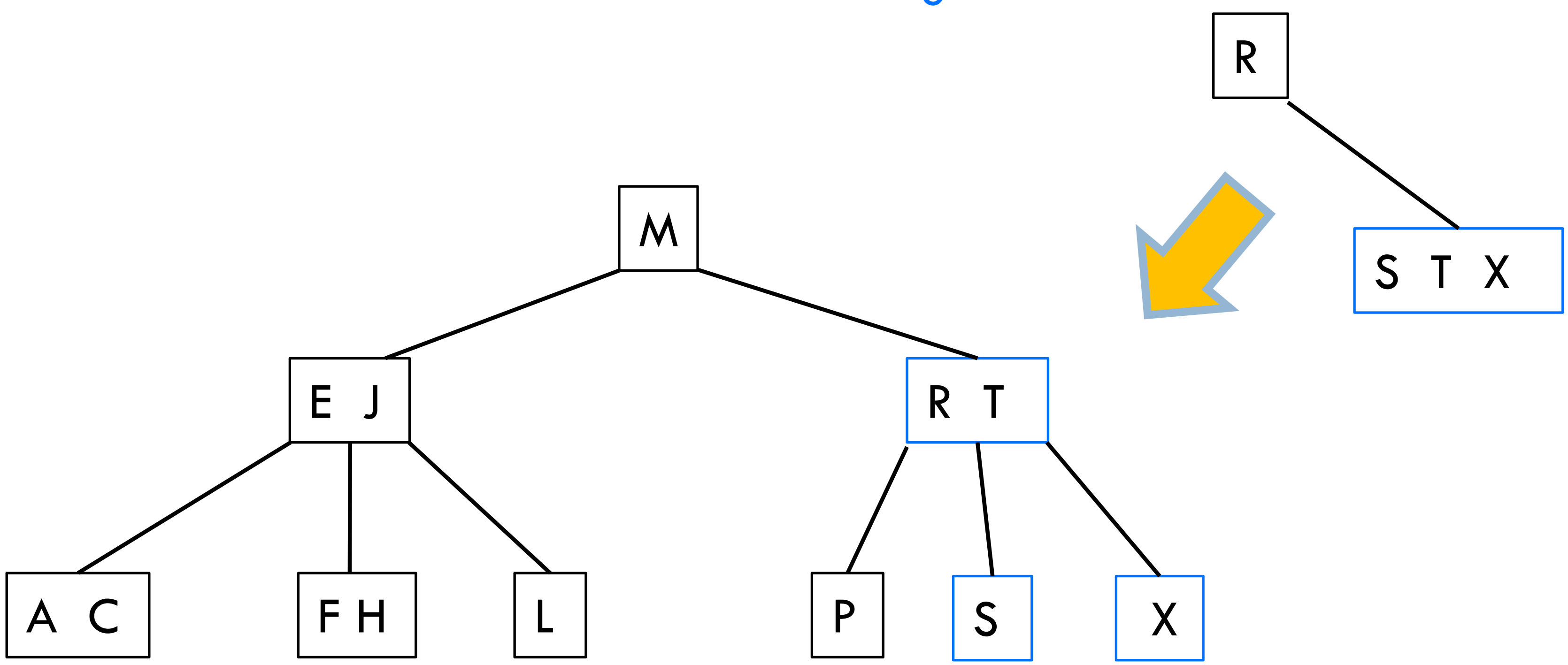


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(T)



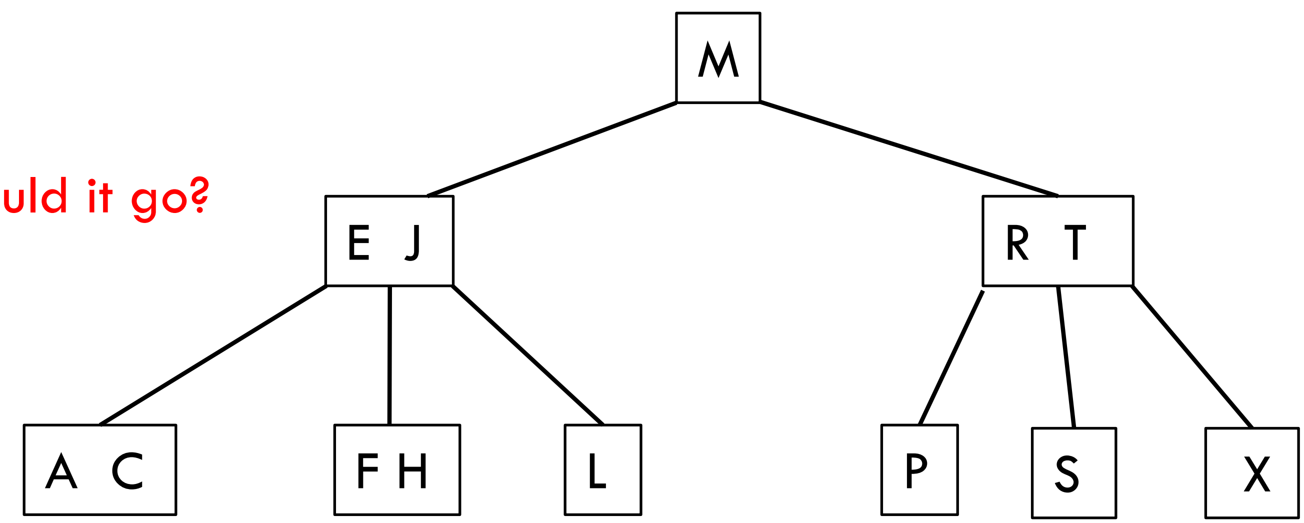
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

Where should it go?

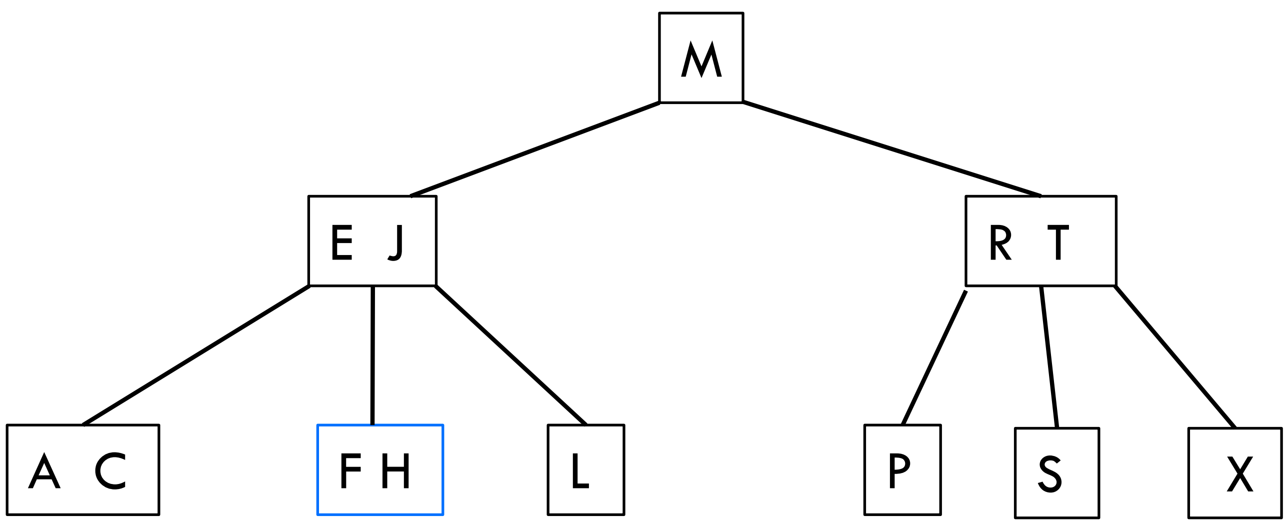


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

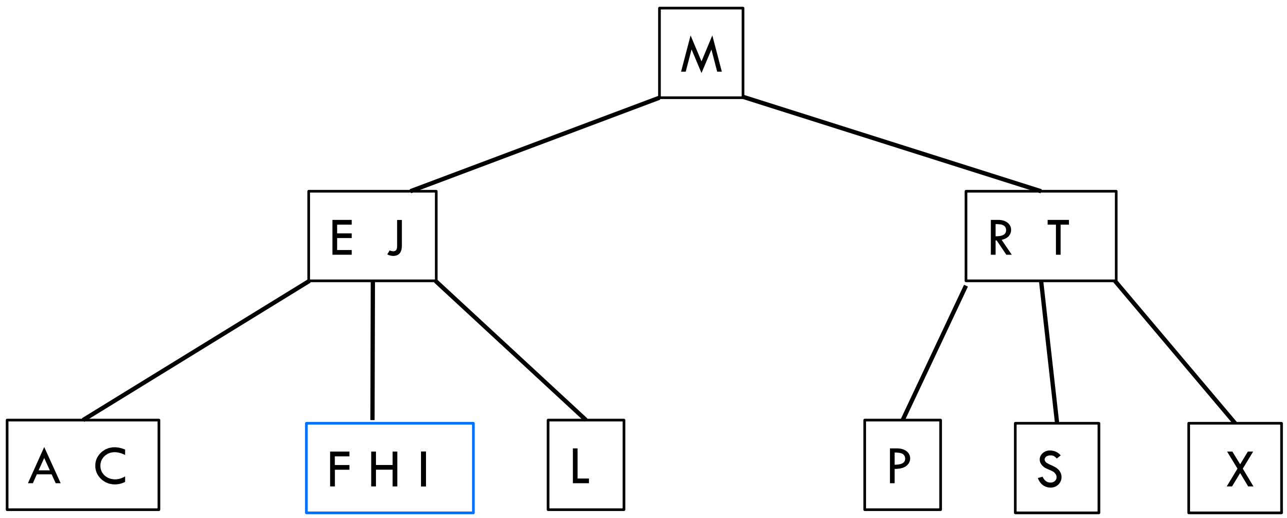


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

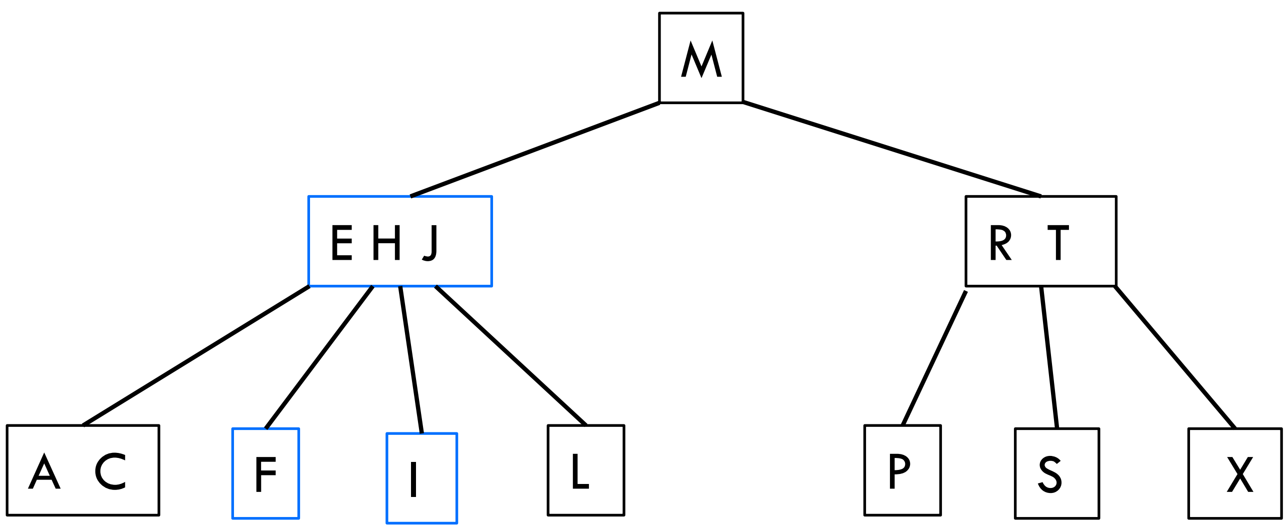


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



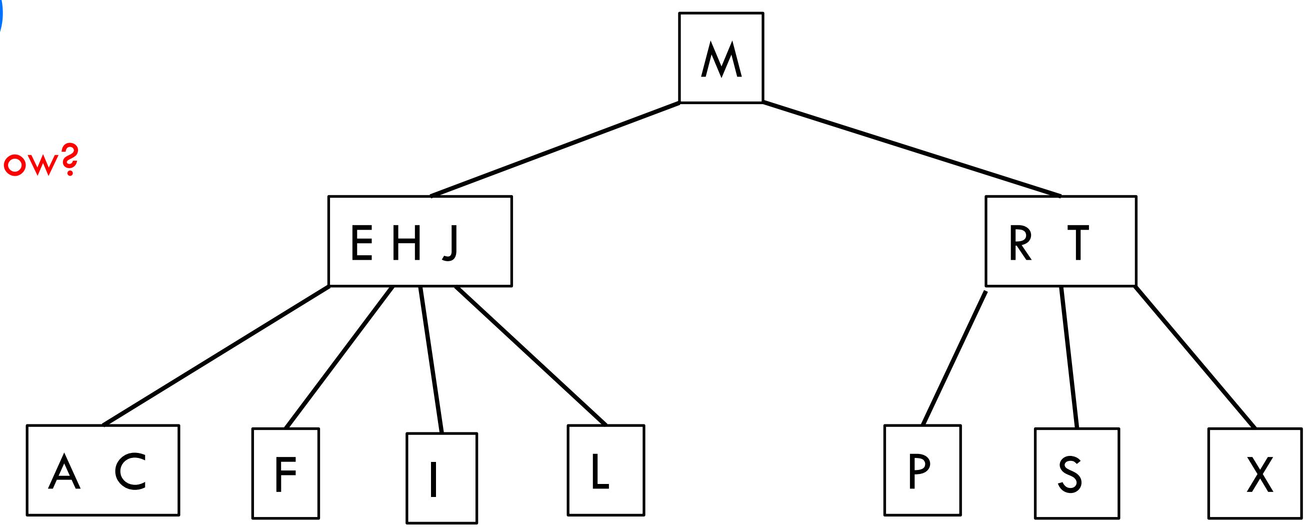
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

What now?



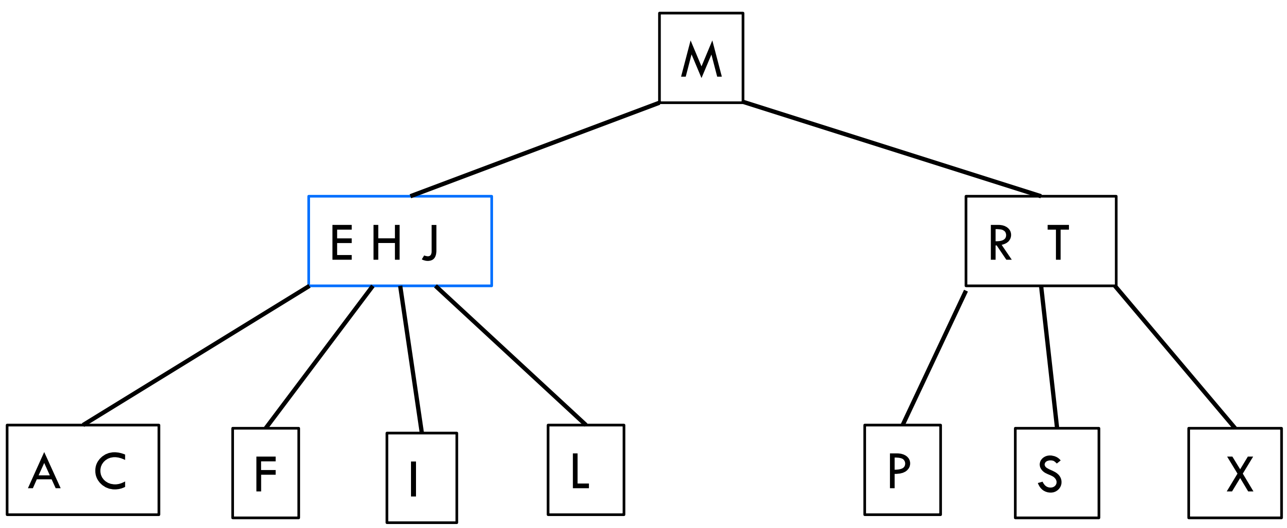
Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

Repeat!

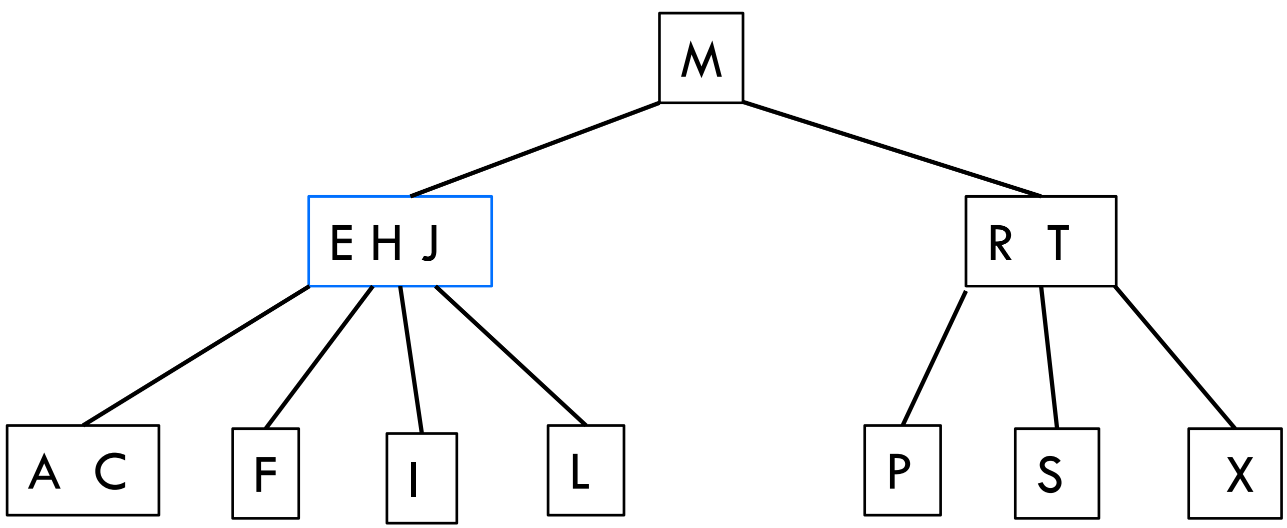


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

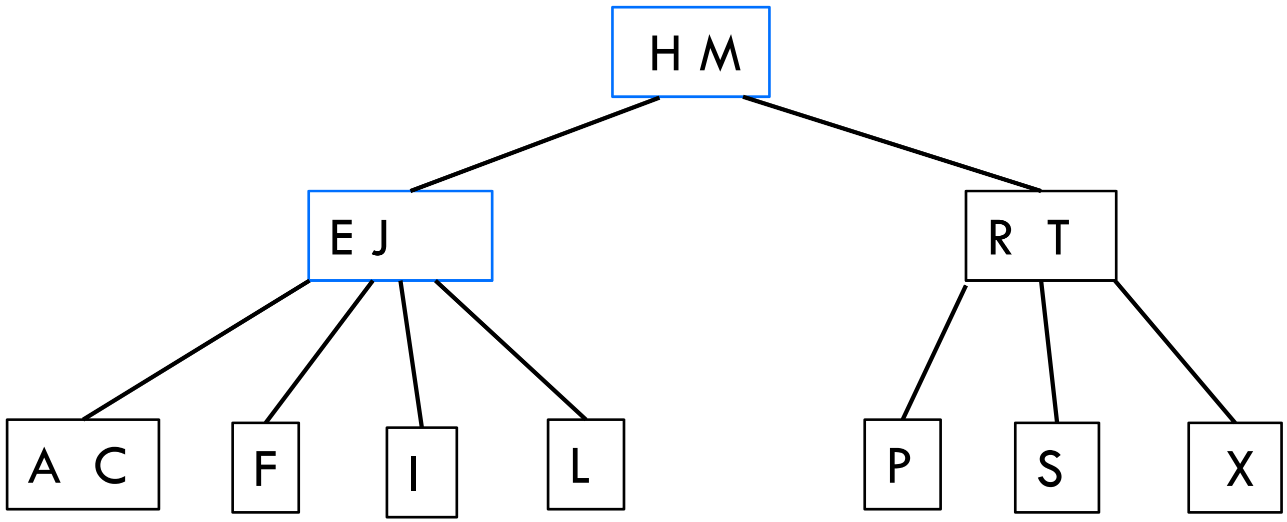


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)

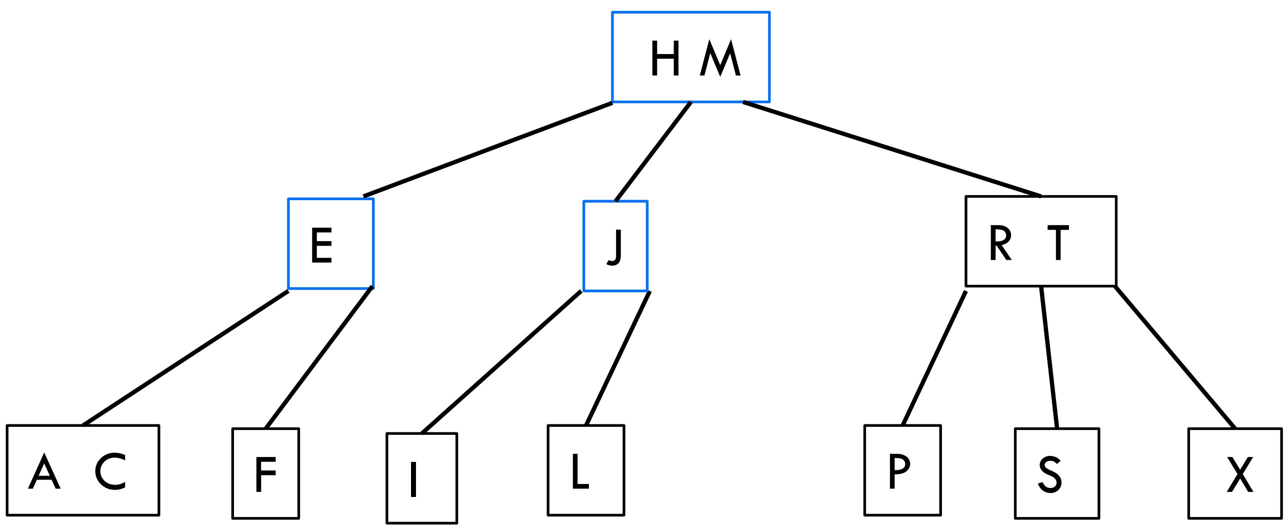


Insertion

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

Insert(I)



Insertion

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

When will the height of the tree change?

Insertion

If the leaf is a 2-node, just insert it directly

If the leaf is a 3-node:

- We now have three values at this leaf
- Send the middle value up a node
- Make new 2-nodes out of the smallest and largest

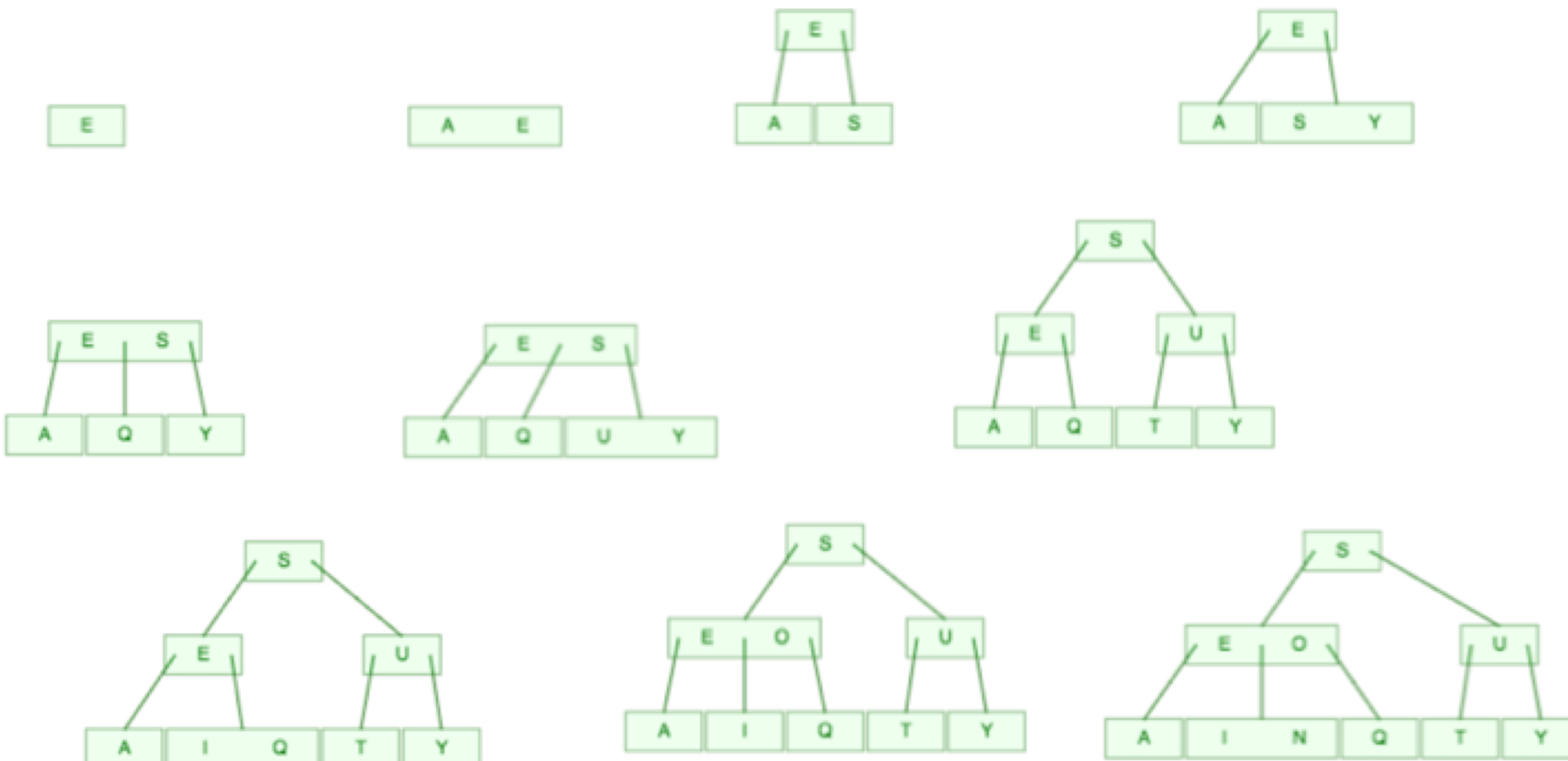
Only when the root is a 3-node and we insert into a path that is all 3-nodes!

Effect: The tree can hold quite a few values before having to increase the height

Practice

Draw the 2-3 tree that results when you insert the keys:

E A S Y Q U T I O N in that order in an initially empty tree.



Running time

Worst case height: O(log n)

What does that mean?

Running time

Worst case height: O(log n)

Insert, search and delete are all O(log n)

2-3 search trees in practice

A pain to implement

Overhead can often make slower than standard BST

Other balanced trees exist that provide the same worst case guarantee, but are faster (e.g, red-black trees)

Readings and practice problems

Textbook: Chapter 3.3 (Pages 424-431)

Website: https://algs4.cs.princeton.edu/33balanced/

Practice problems: 3.3.2– 3.3.5