NPTEL MOOC,JAN-FEB 2015
Week 6, Module 1

DESIGN AND ANALYSIS OF ALGORITHMS

Search trees

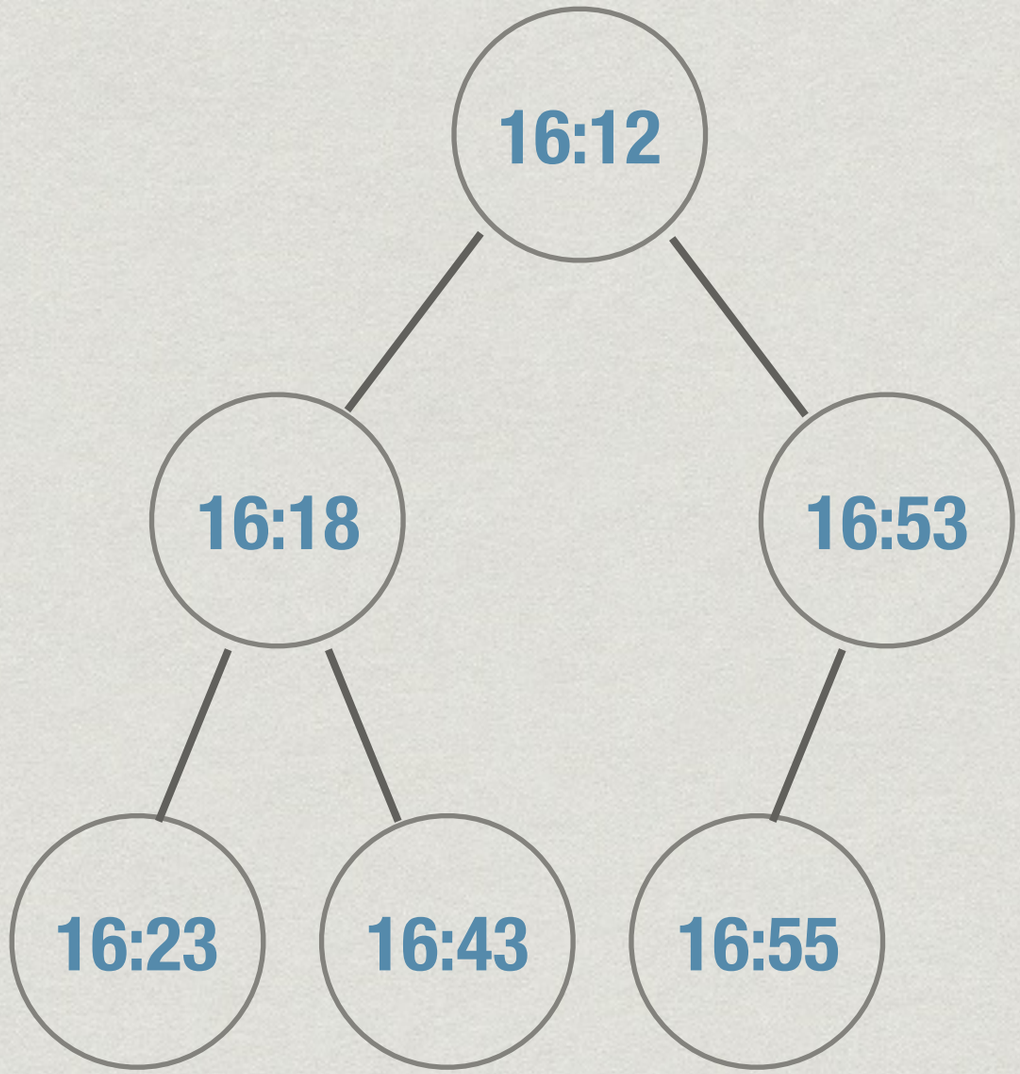
MADHAVAN MUKUND, CHENNAI MATHEMATICAL INSTITUTE
http://www.cmi.ac.in/~madhavan

Example: Air traffic control

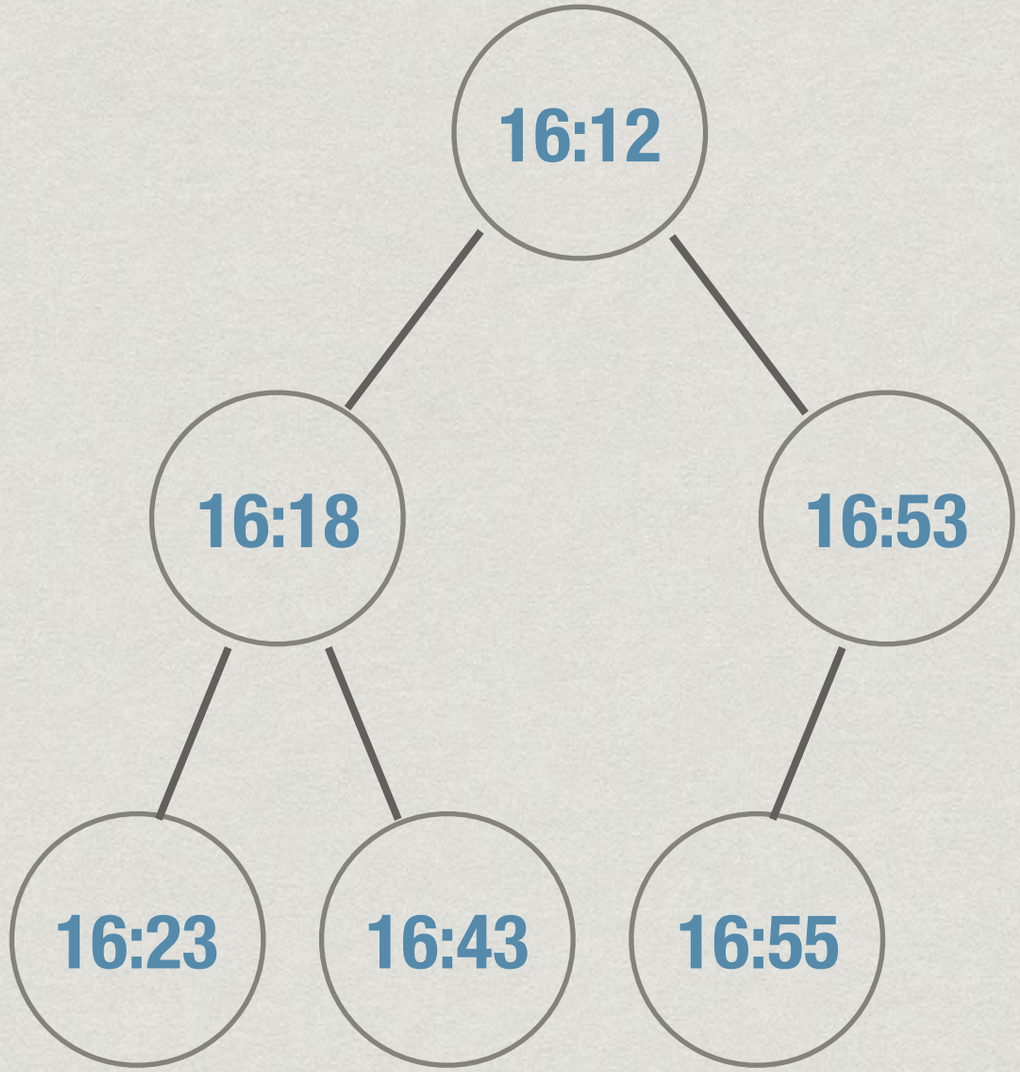
- Flights arrive and depart at an airport
- Single runway — reserve slots for landing / takeoff
- Requests for slots come in arbitrary order
  - Trivandrum-Chennai, Air India, 16:23
  - Chennai-Madurai, SpiceJet, 16:12
  - Delhi-Chennai, Indigo, 16:55
  - Port Blair Chennai, Jet Airways, 16:18
- Give priority to flights with earliest usage time

Air traffic control … heaps?

- Pending landing and take off requests
  - 16:23, 16:12, 16:55, 16:18, 16:43, 16:53
- At 16:10, Chennai-Madurai flight of 16:12 given clearance to take off

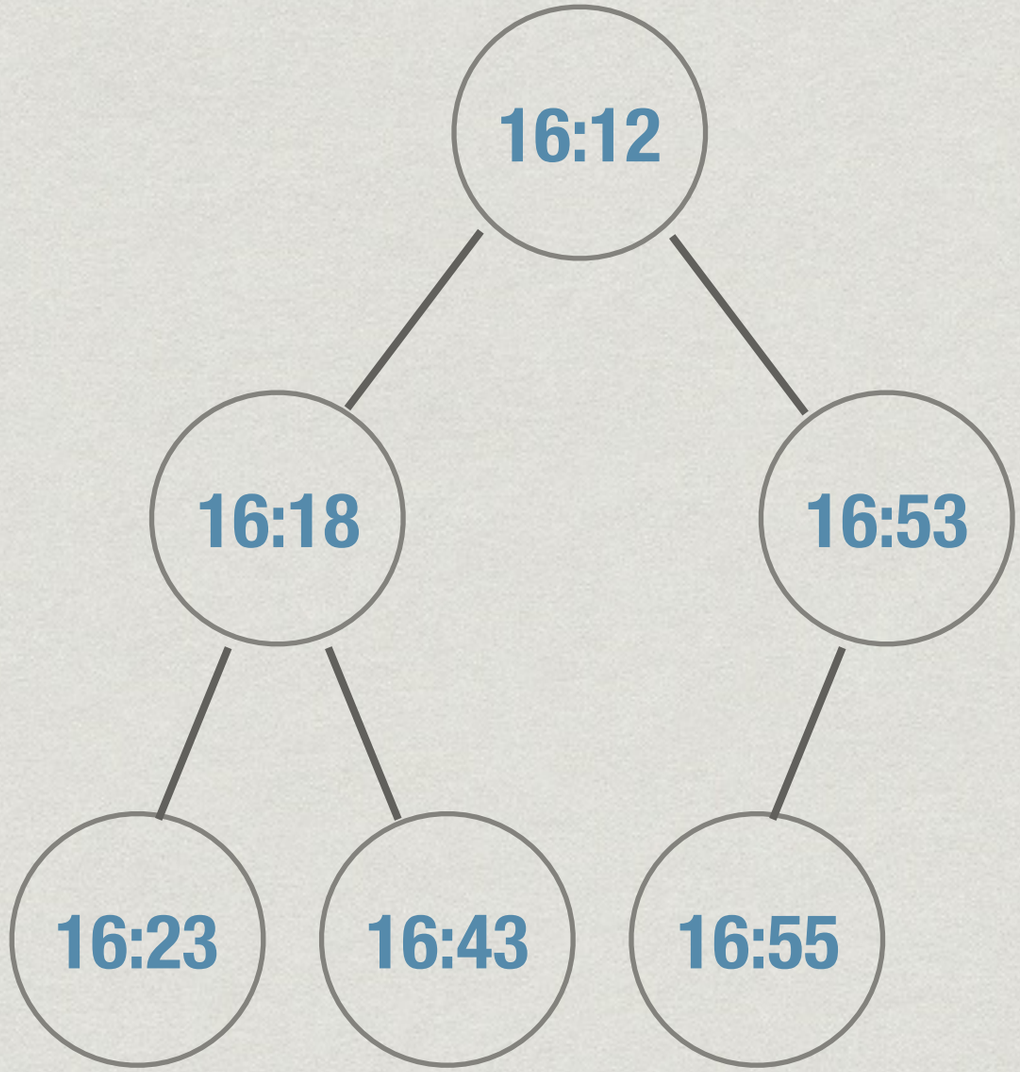


Air traffic control … heaps?



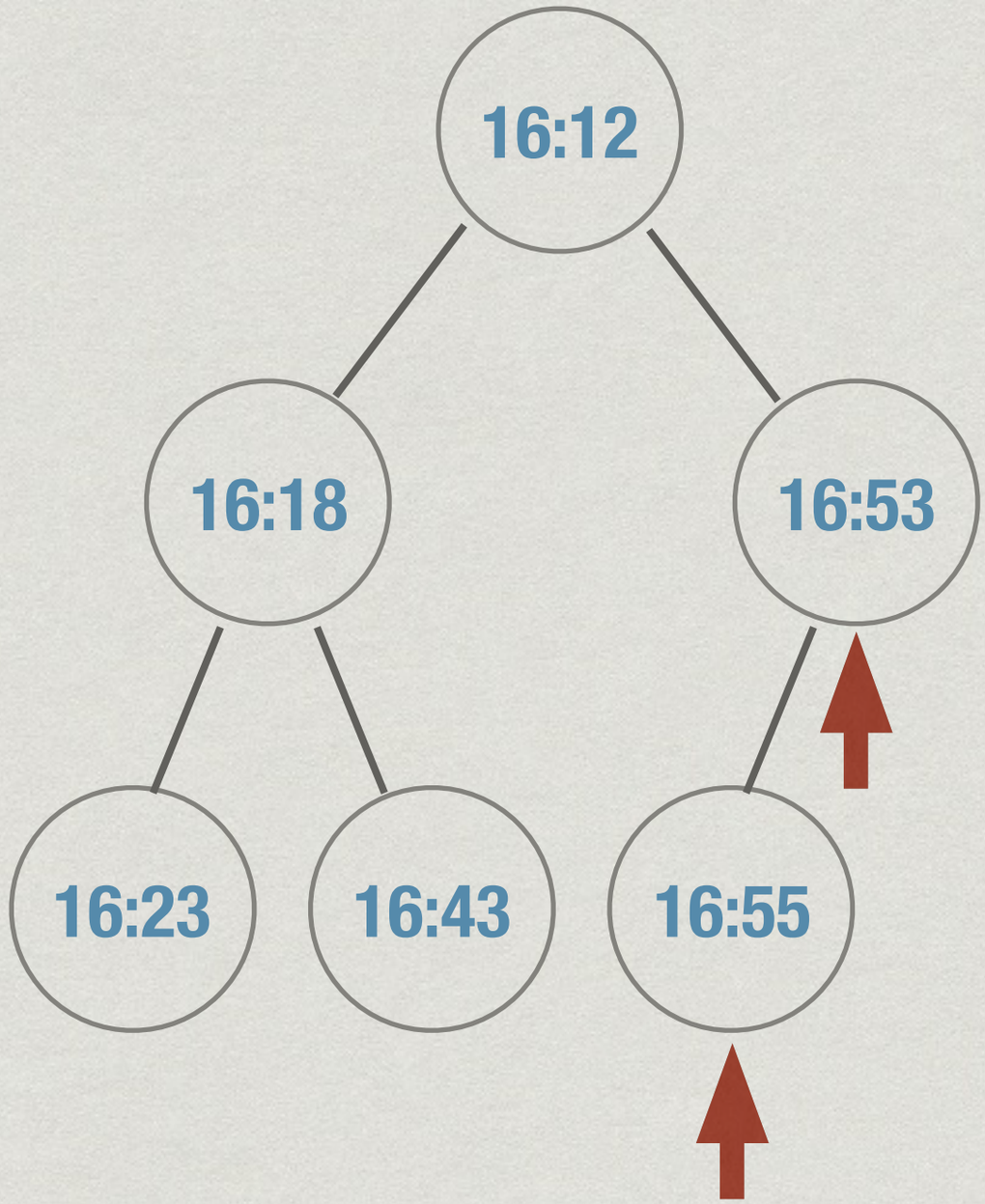
Air traffic control … heaps?

- Suppose we want flights to be spaced apart
- Minimum of 3 minutes between planes on runway



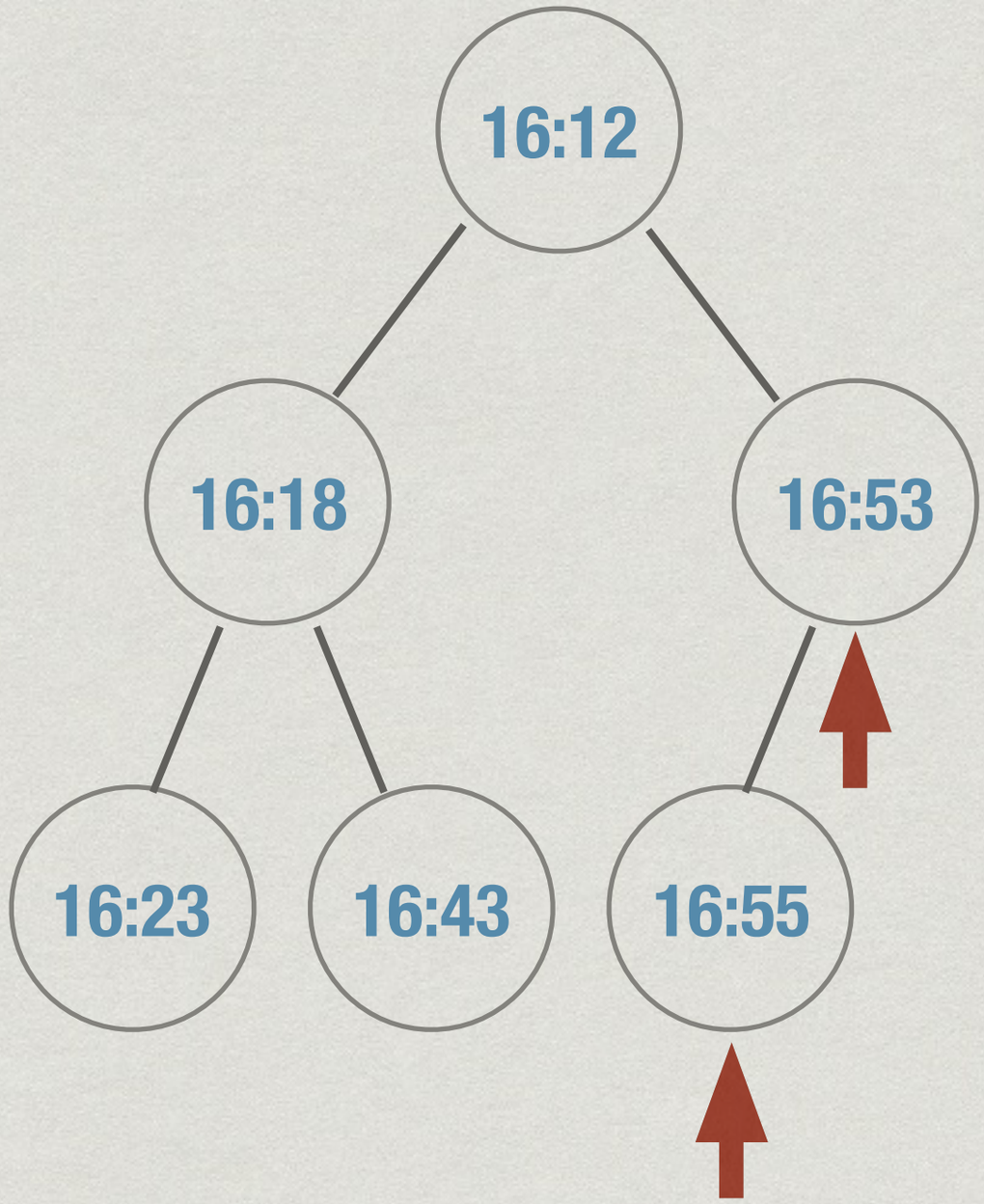
Air traffic control … heaps?

- Suppose we want flights to be spaced apart
  - Minimum of 3 minutes between planes on runway
- Rule is violated



Air traffic control … heaps?

- Suppose we want flights to be spaced apart
  - Minimum of 3 minutes between planes on runway
- Rule is violated
- Requires O(n) scan when inserting

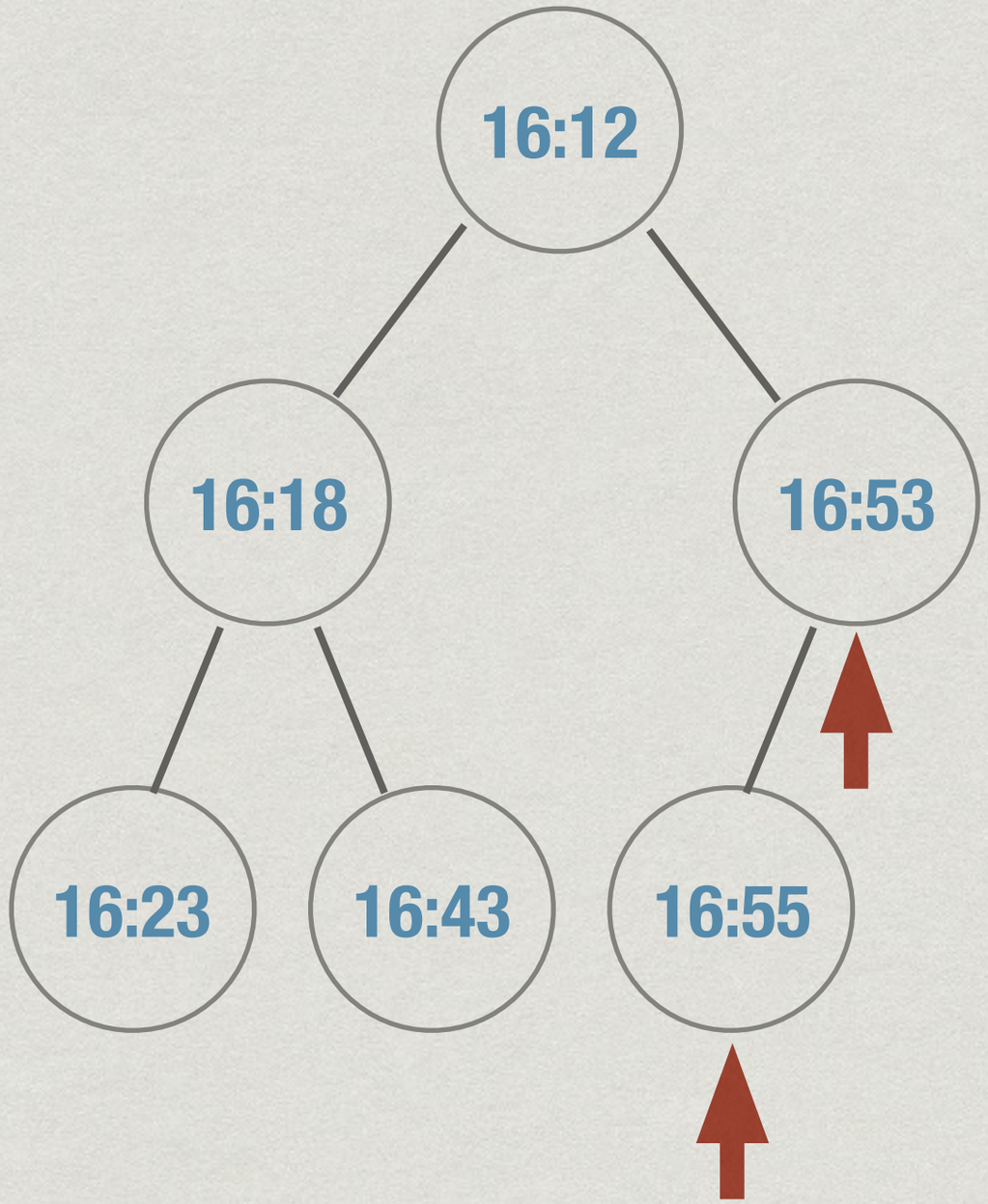


Air traffic control … heaps?

- If we could compute predecessor and successor, we could check this easily

- pred(16:18) = 16:12
- succ(16:18) = 16:23

- pred(16:53) = 16:43
- succ(16:53) = 16:55

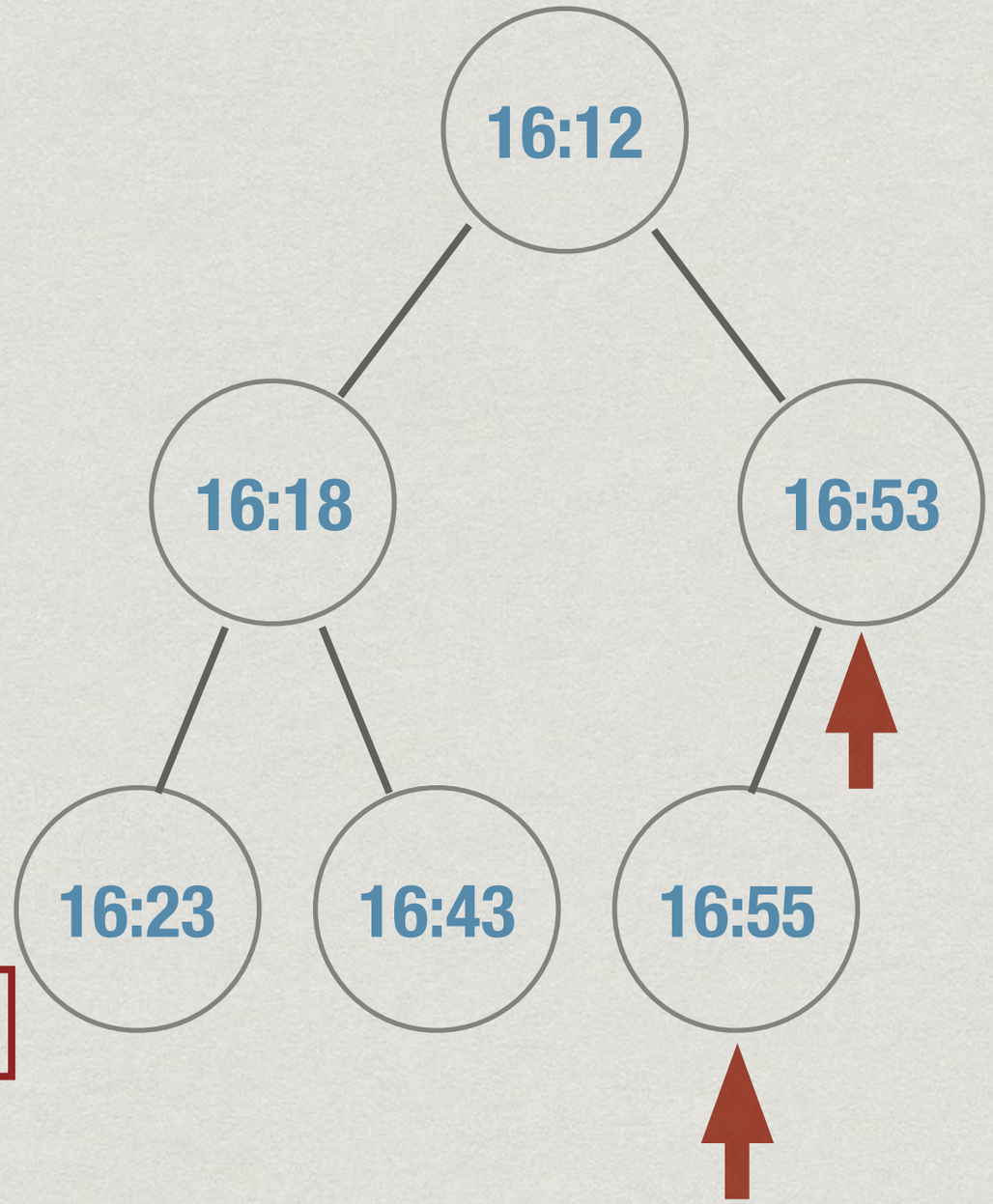


Air traffic control … heaps?

- If we could compute predecessor and successor, we could check this easily

- pred(16:18) = 16:12
- succ(16:18) = 16:23

- pred(16:53) = 16:43
- succ(16:53) = 16:55



Comparing data structures

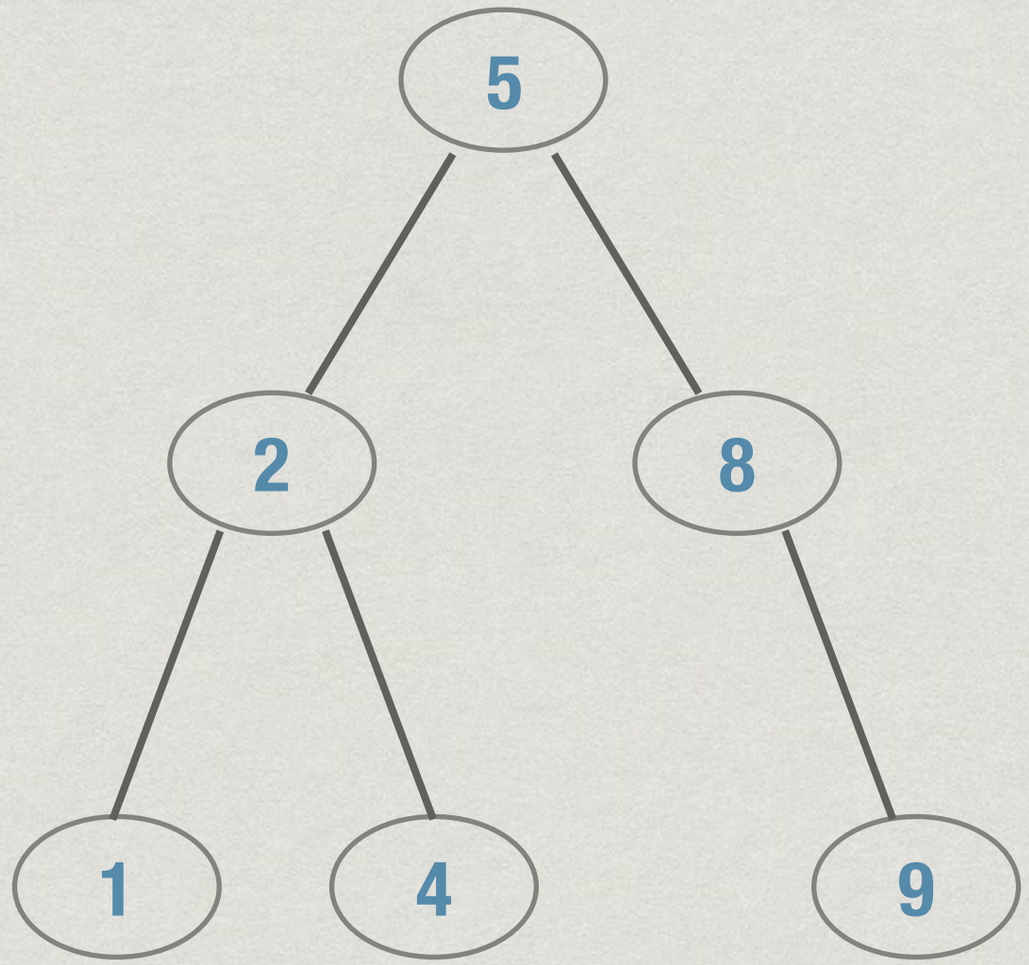
	Unsorted array	Sorted array	Min Heap
Min	O(n)	O(1)	O(1)
Max	O(n)	O(1)	O(n)
Insert	O(1)	O(n)	O(log n)
Delete	O(n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(n)
Succ	O(n)	O(1)	O(n)

Binary search trees

	Heap	Sorted array	Search tree
Find	O(n)	O(log n)	O(log n)
Min	O(1)	O(1)	O(log n)
Max	O(n)	O(1)	O(log n)
Insert	O(log n)	O(n)	O(log n)
Delete	O(log n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(log n)
Succ	O(n)	O(1)	O(log n)

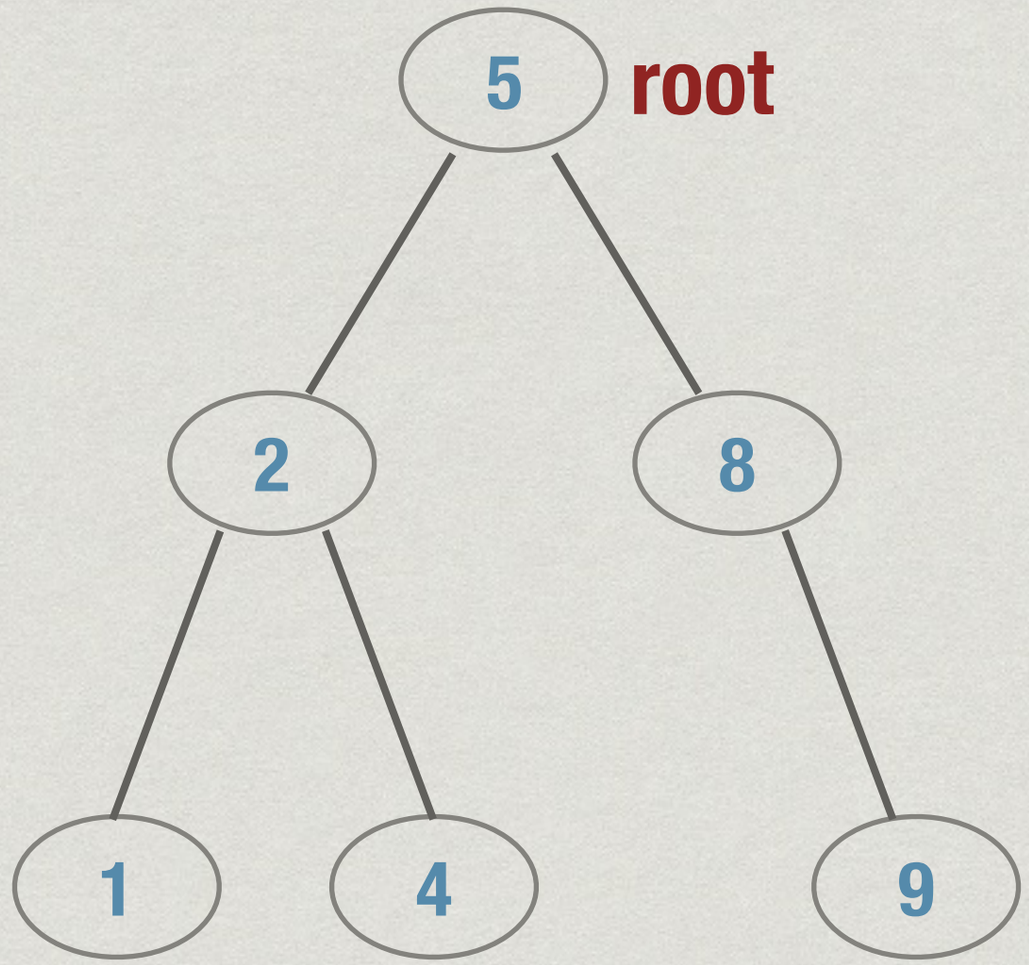
Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child



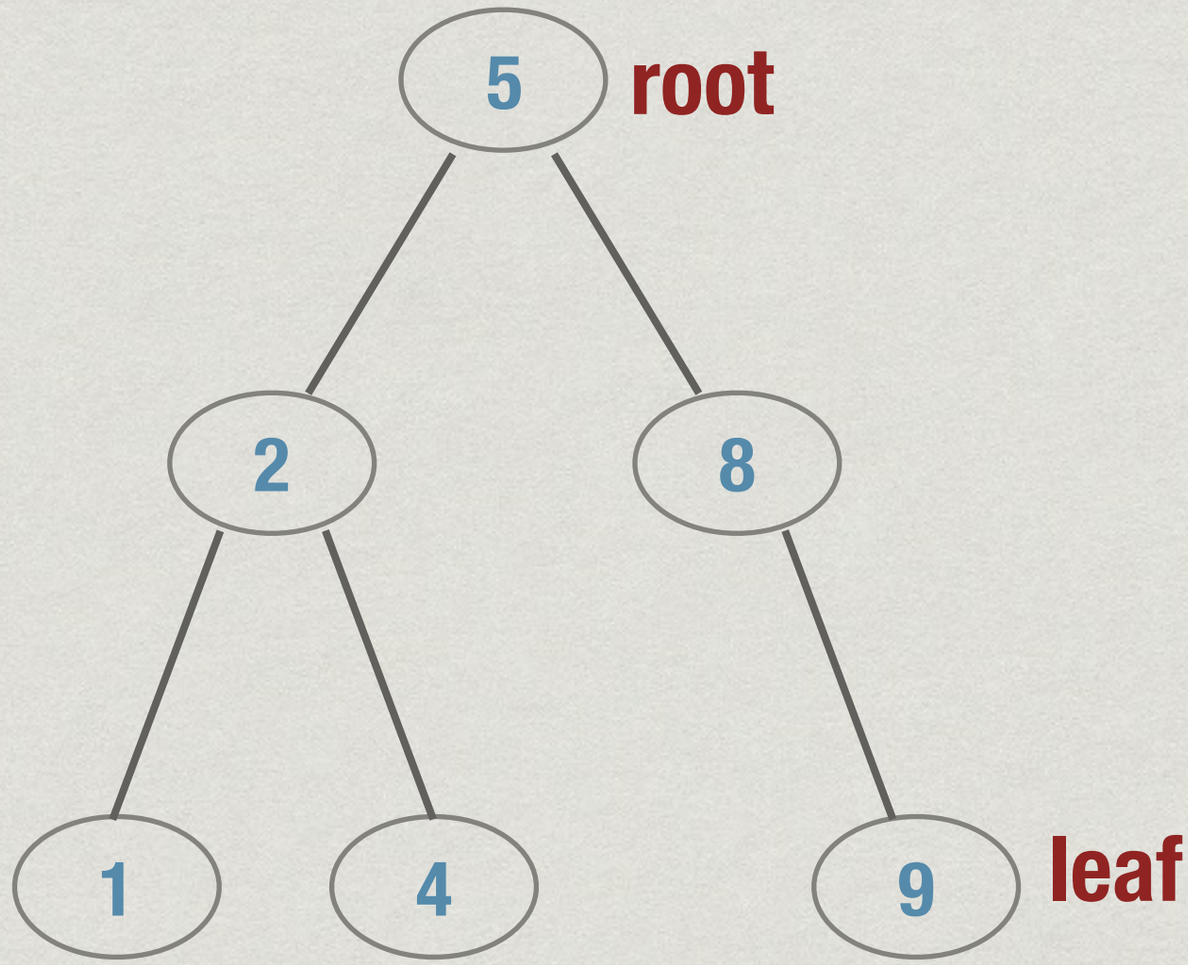
Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child



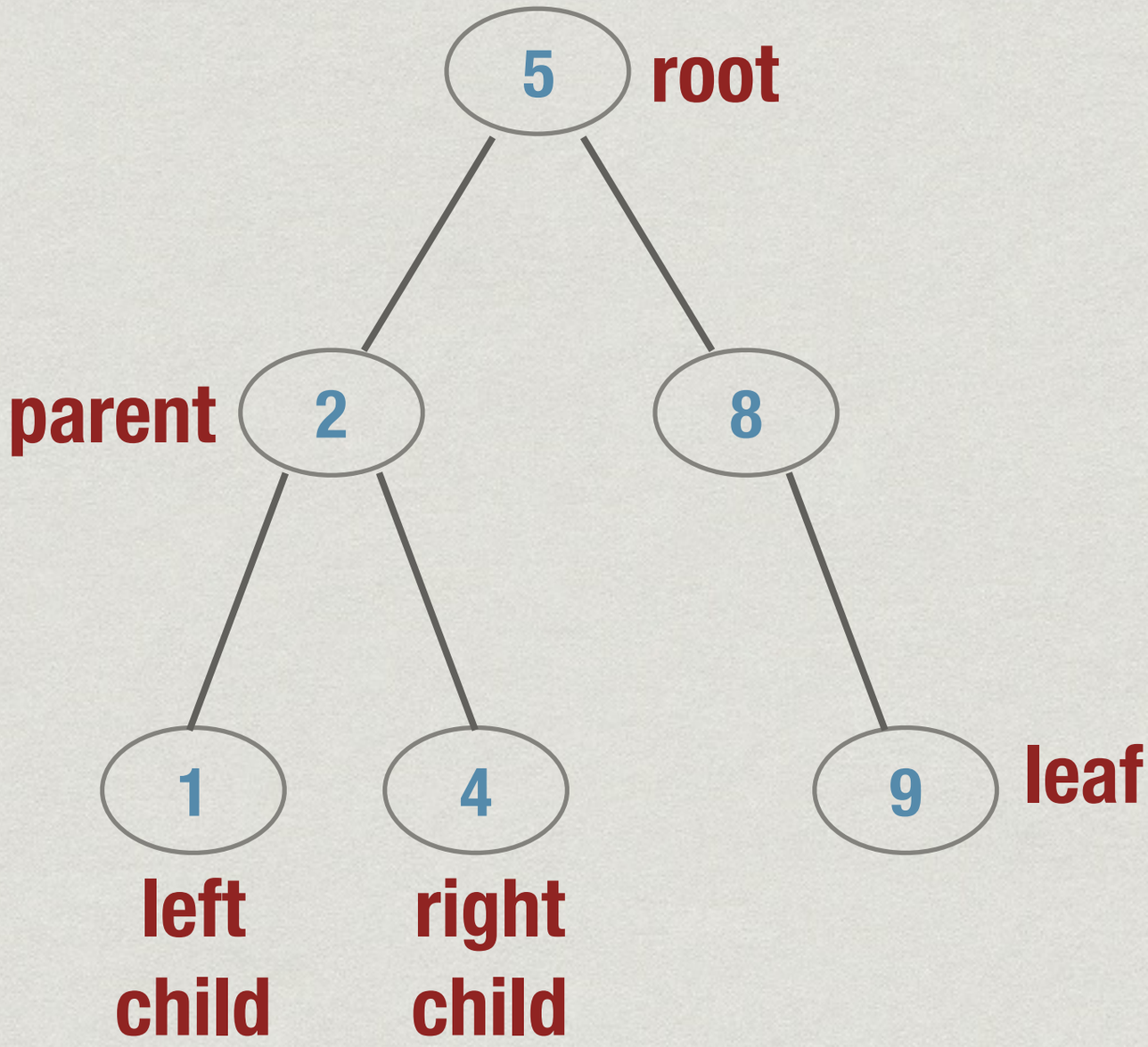
Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child



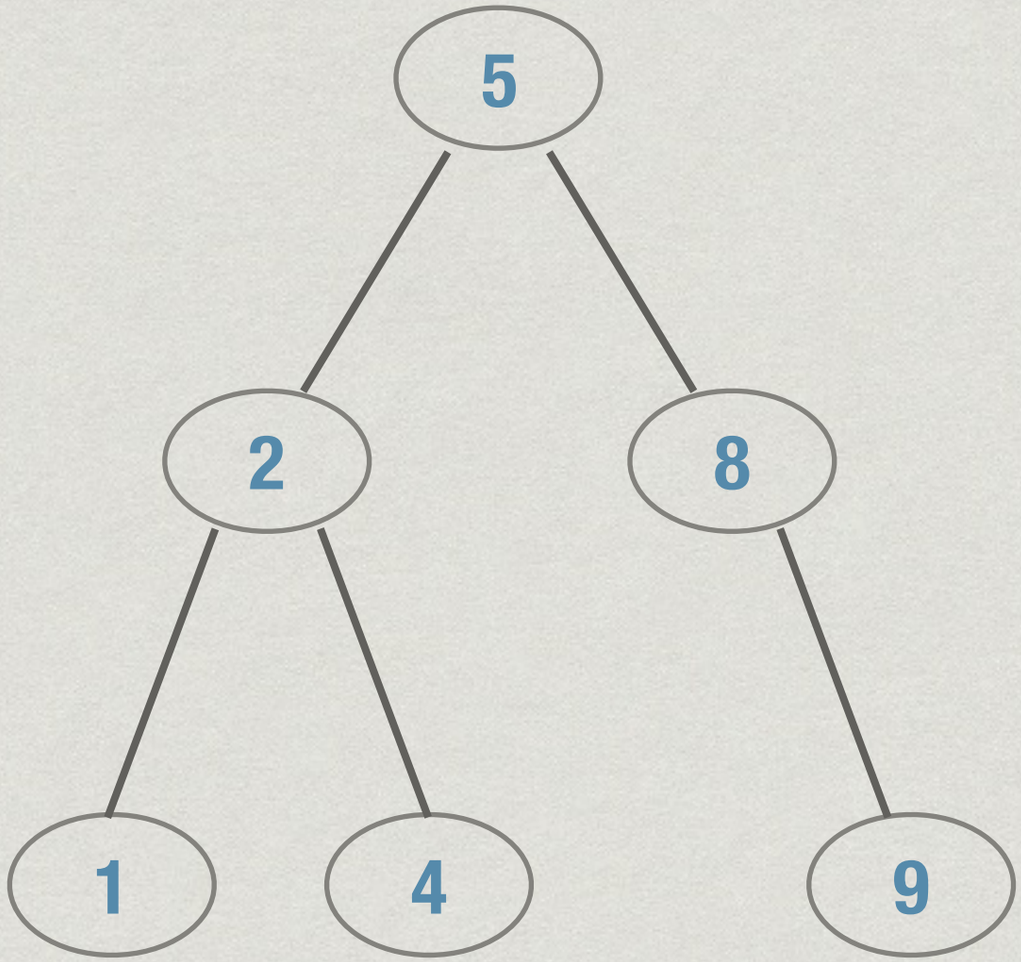
Binary tree

- Structure is not constrained, unlike heap
- At each node
  - Value
  - Link to parent, left child, right child



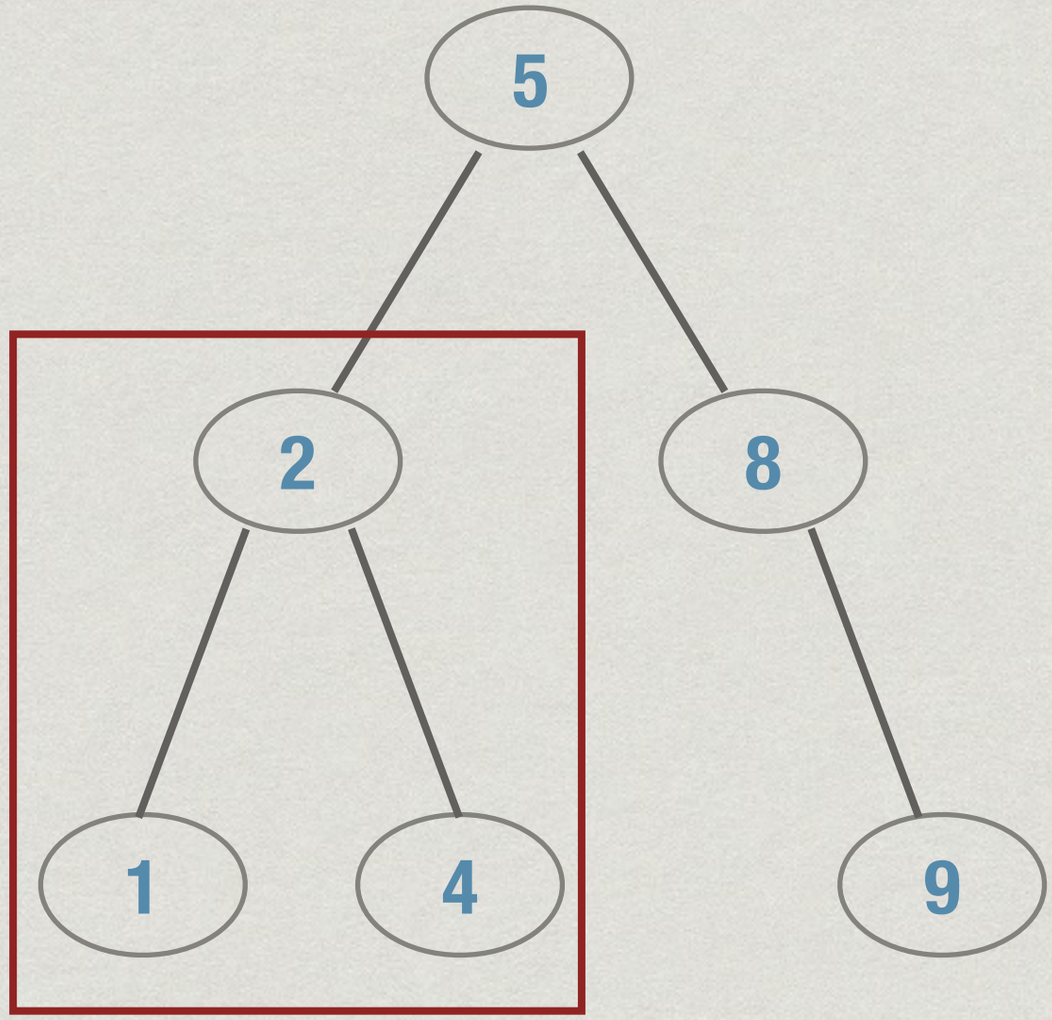
Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



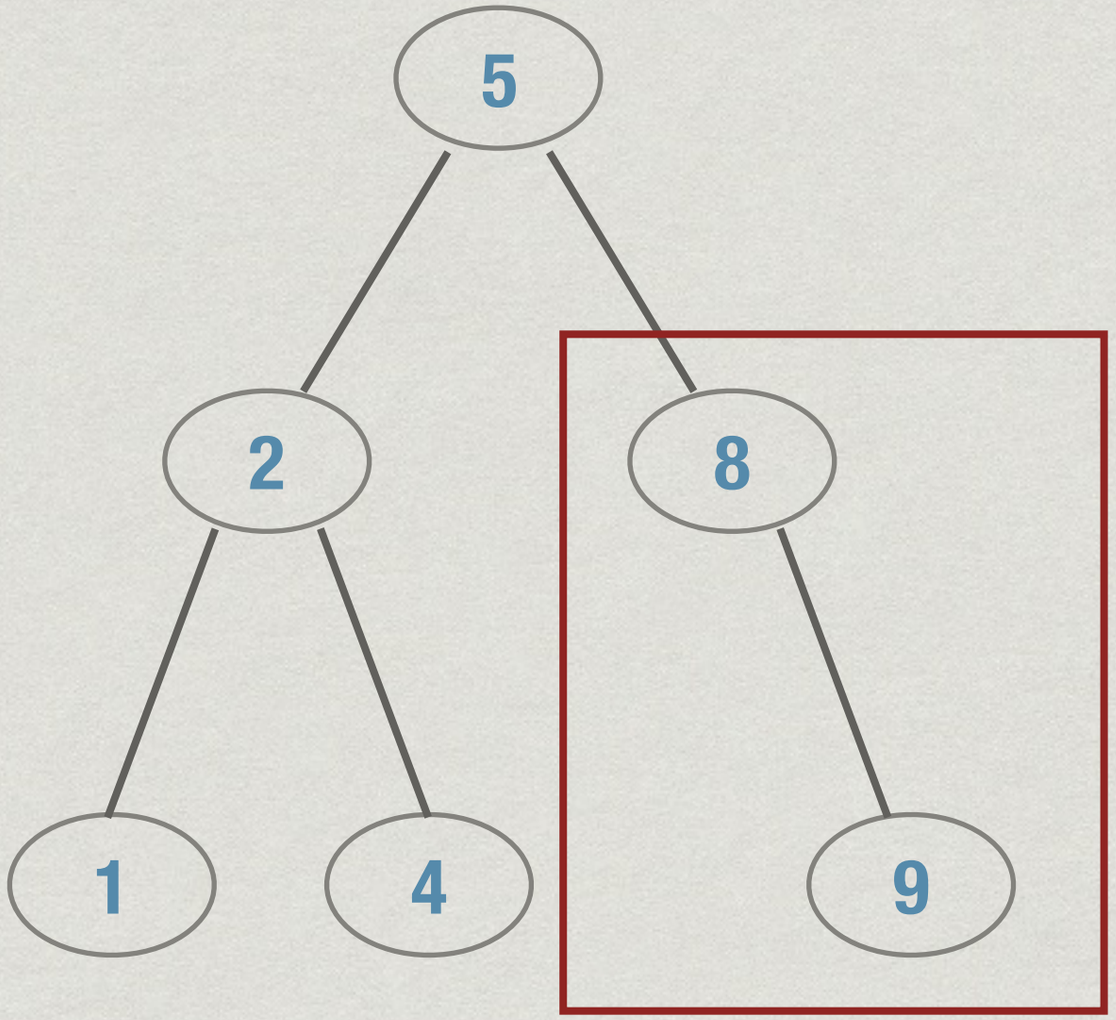
Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



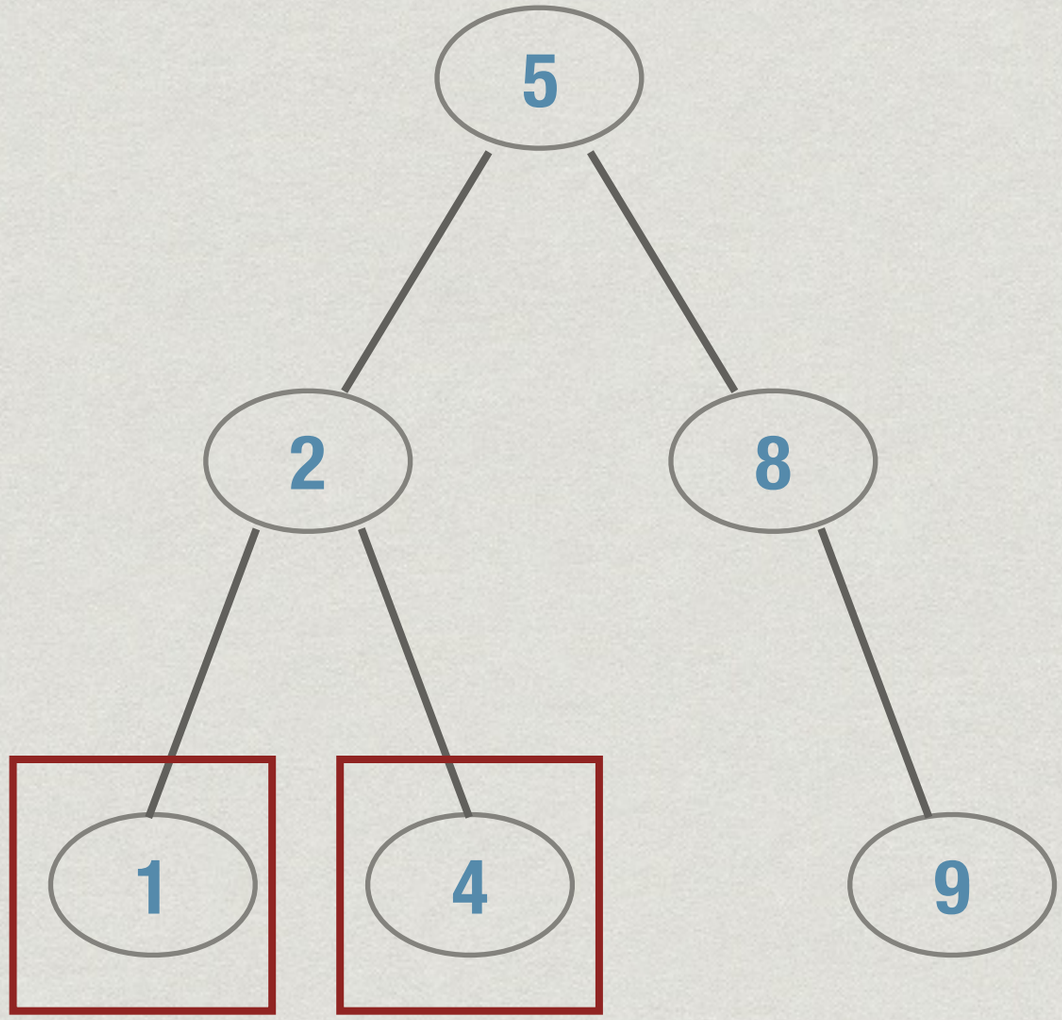
Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



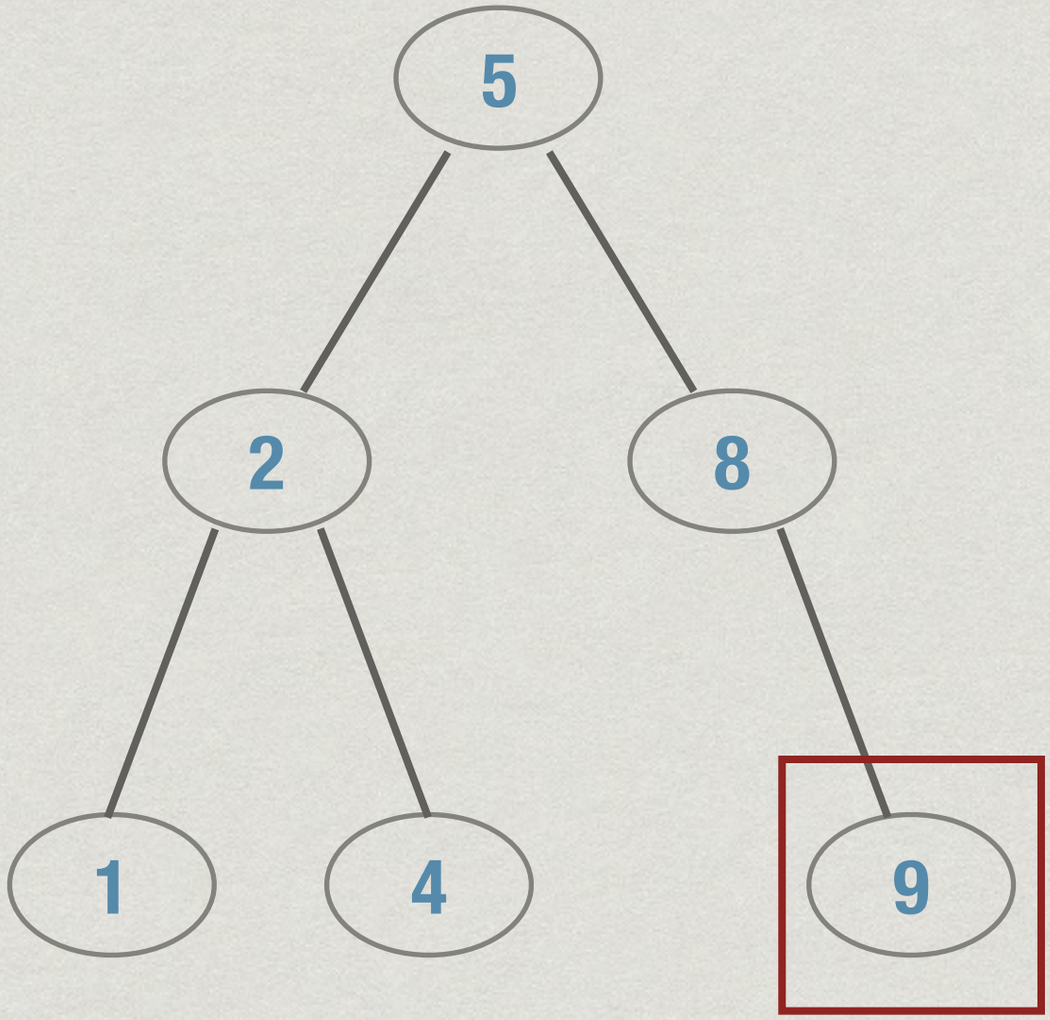
Binary search tree

- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



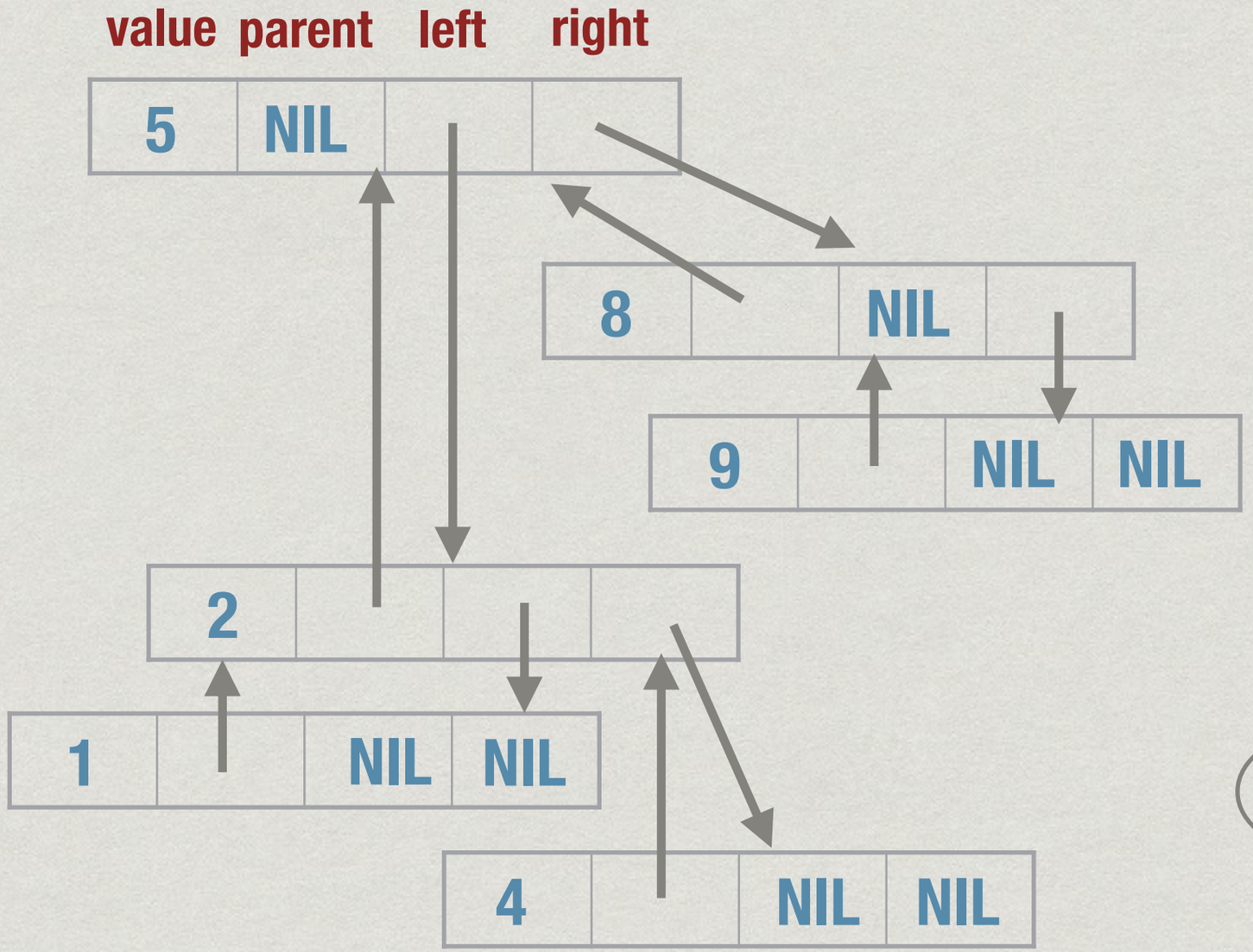
Binary search tree

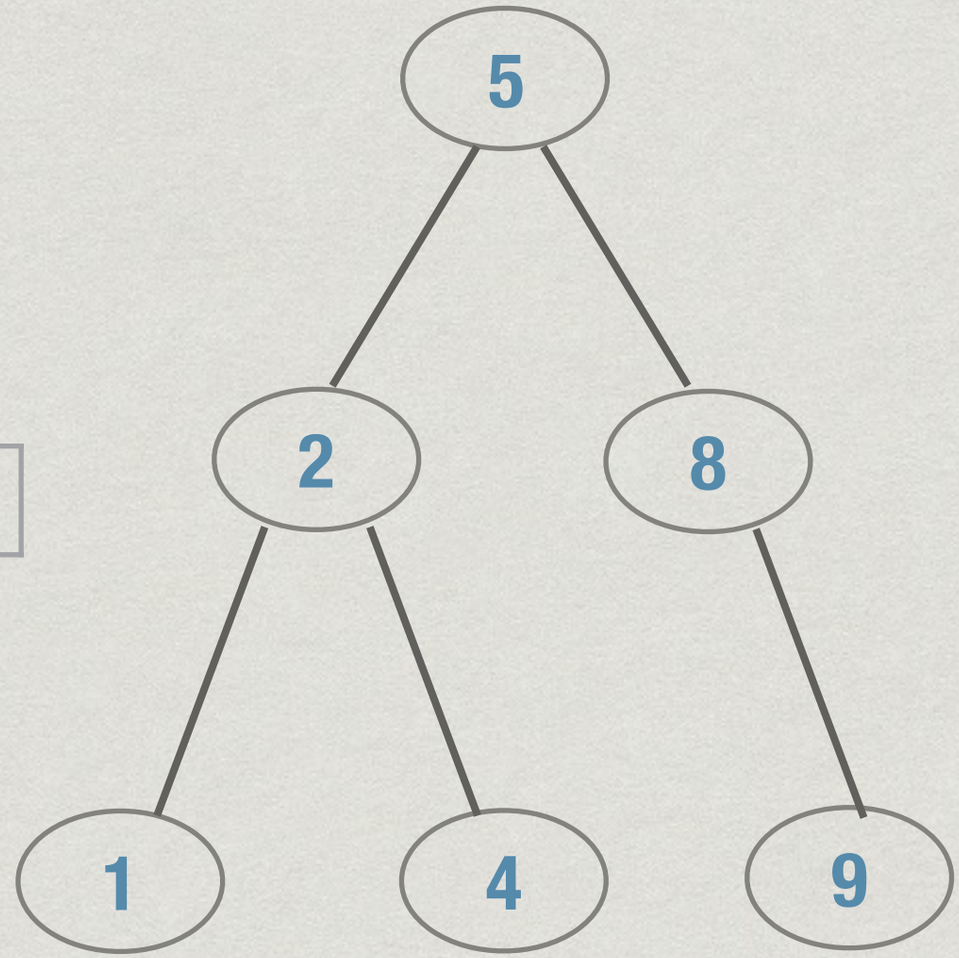
- For each node with value v
  - Values in left subtree < v
  - Values in right subtree > v
- No duplicate values



Binary search tree

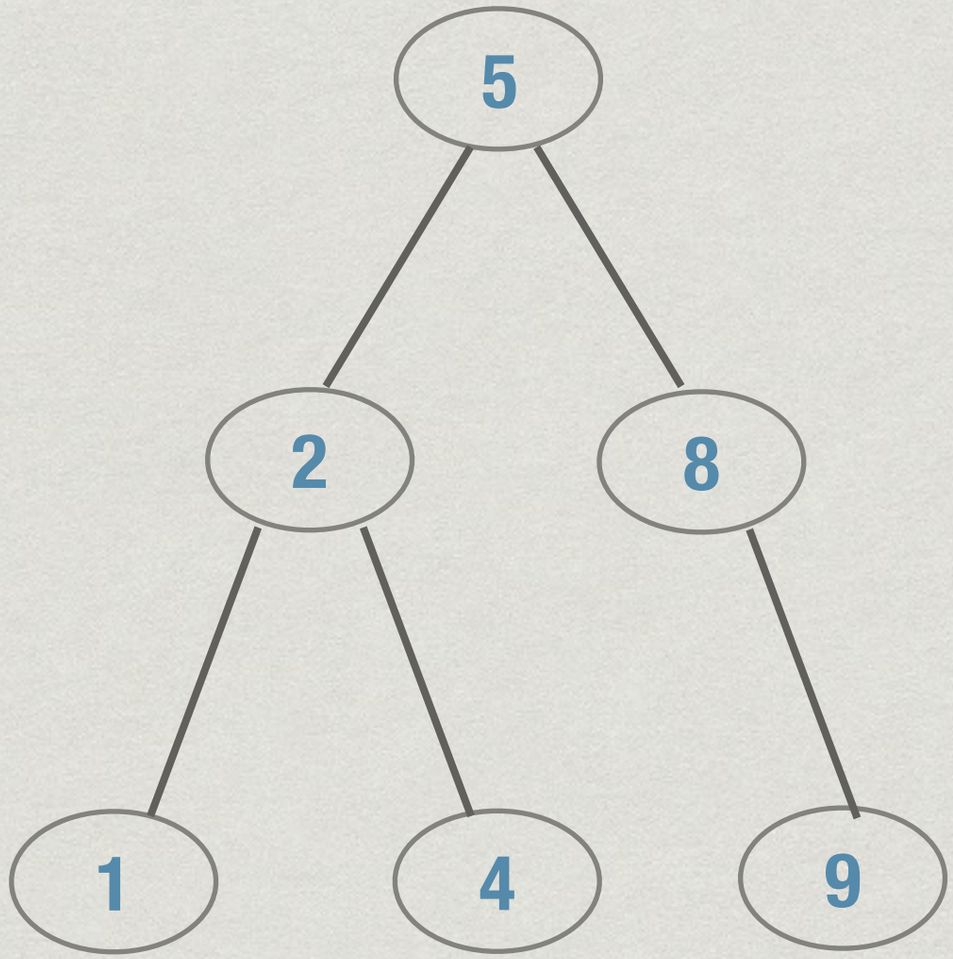
- Implement using pointers





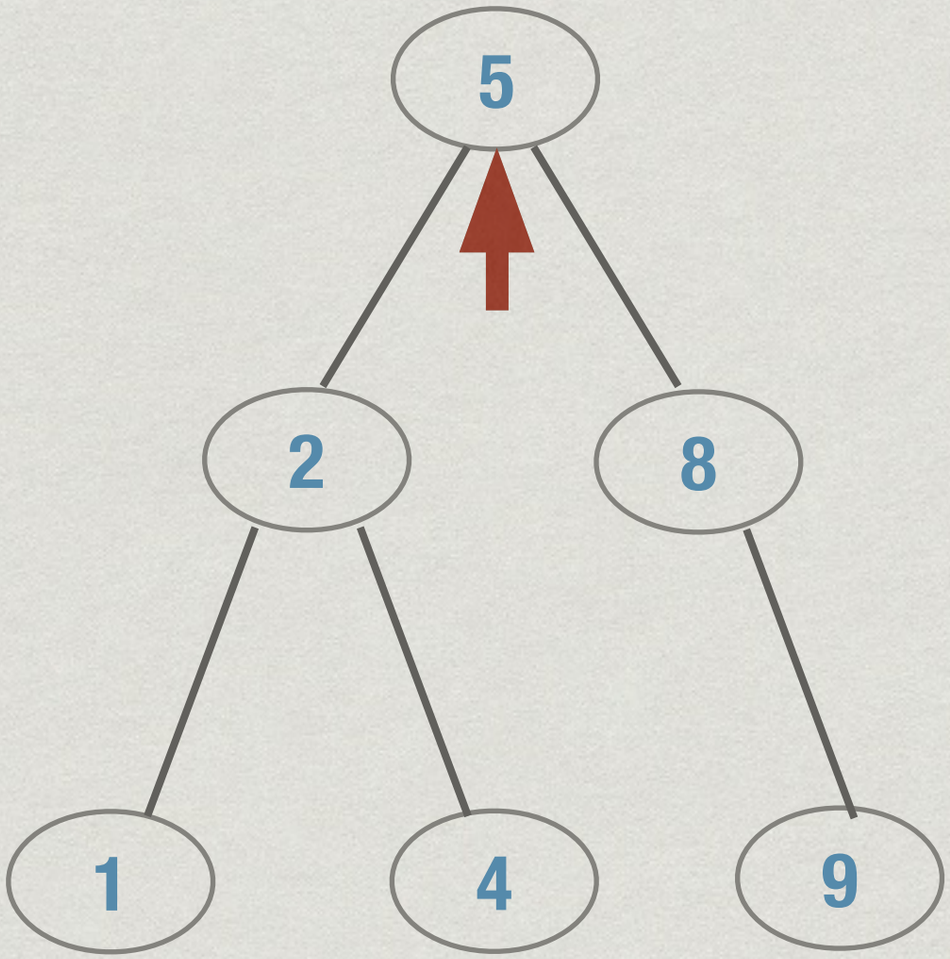
Inorder traversal

```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



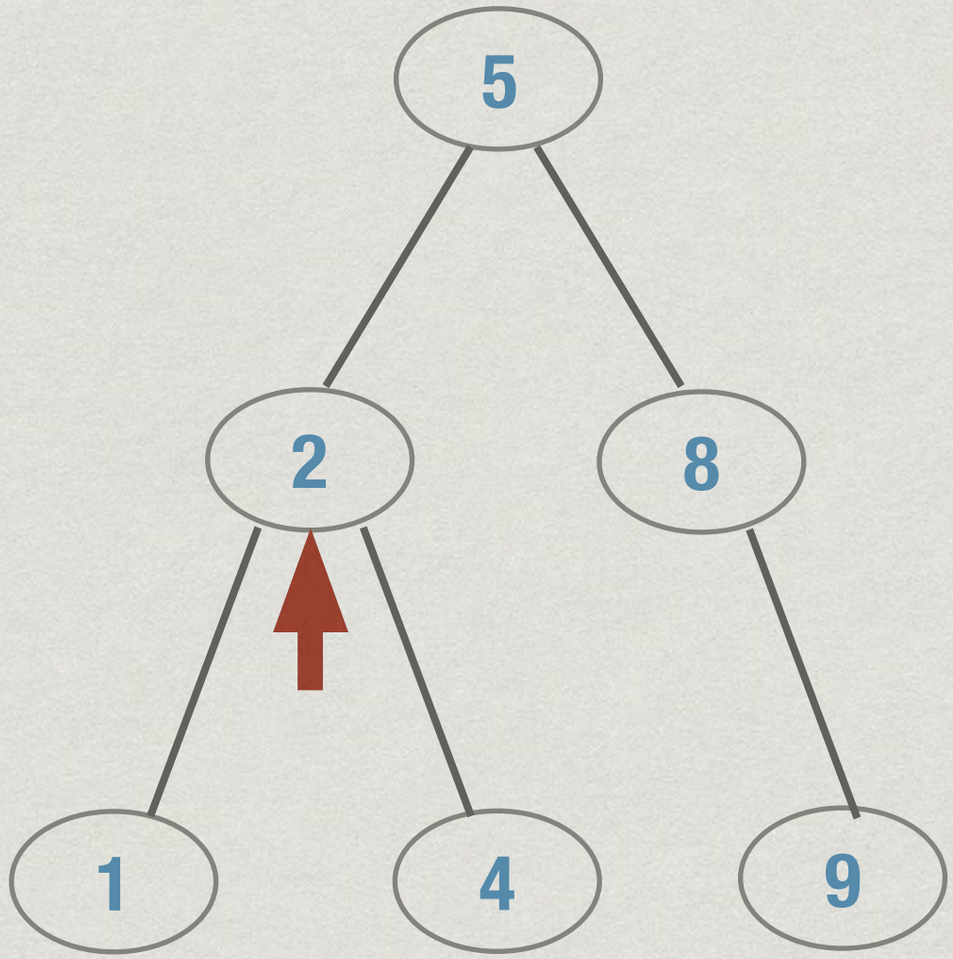
Inorder traversal

```
function inOrder(t)
if (t != NIL)
   inOrder(t.left)
   print(t.value)
   inOrder(t.right)
```



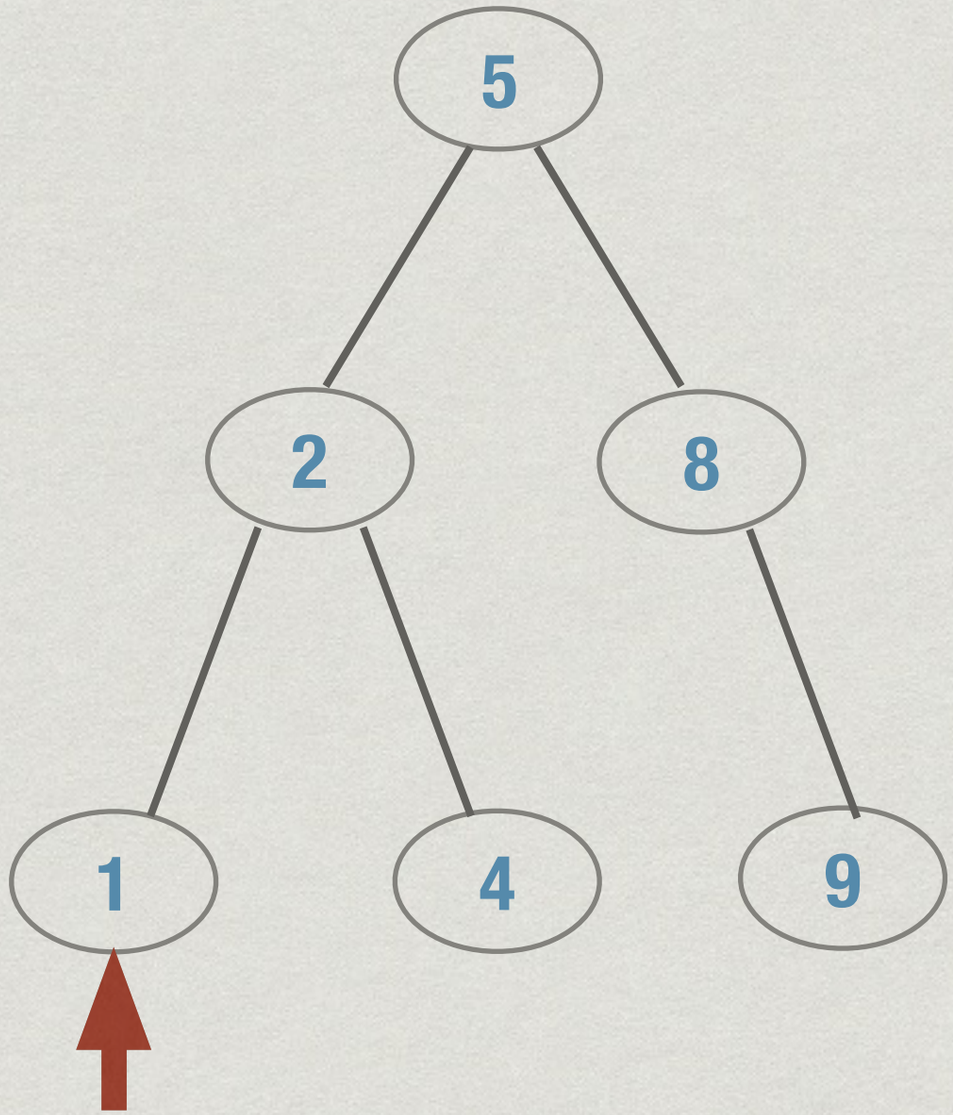
Inorder traversal

```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



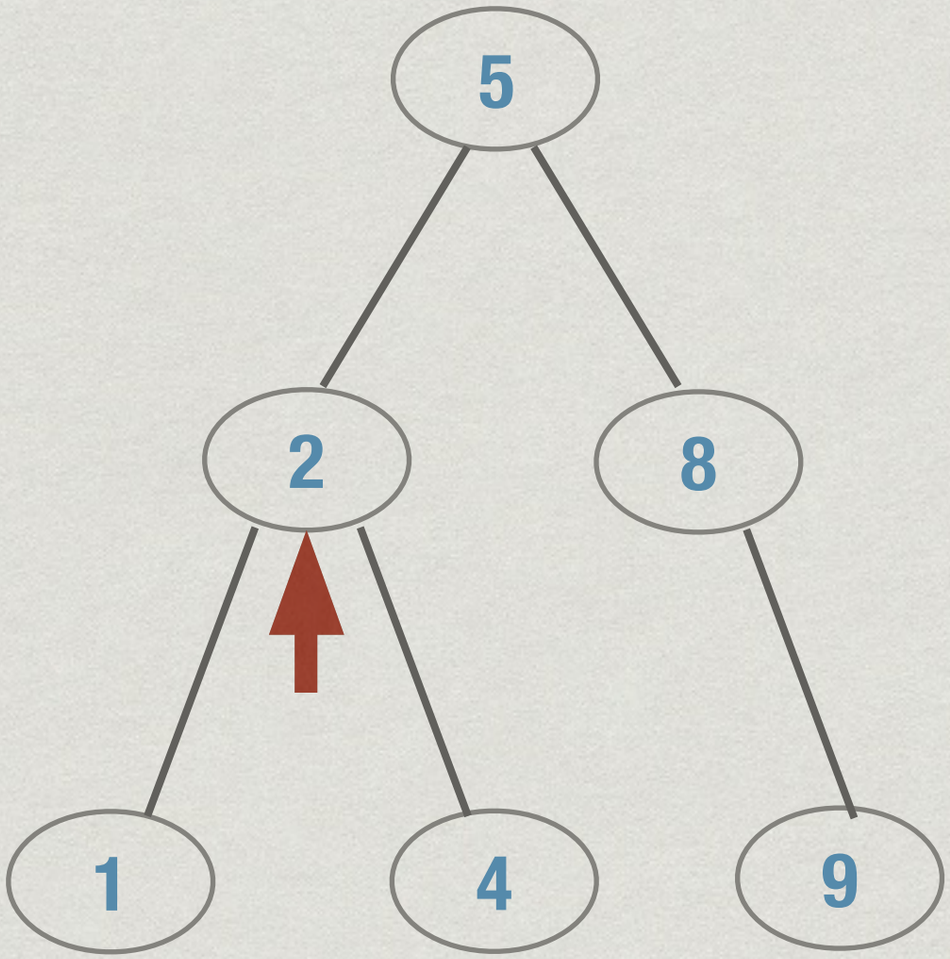
Inorder traversal

```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



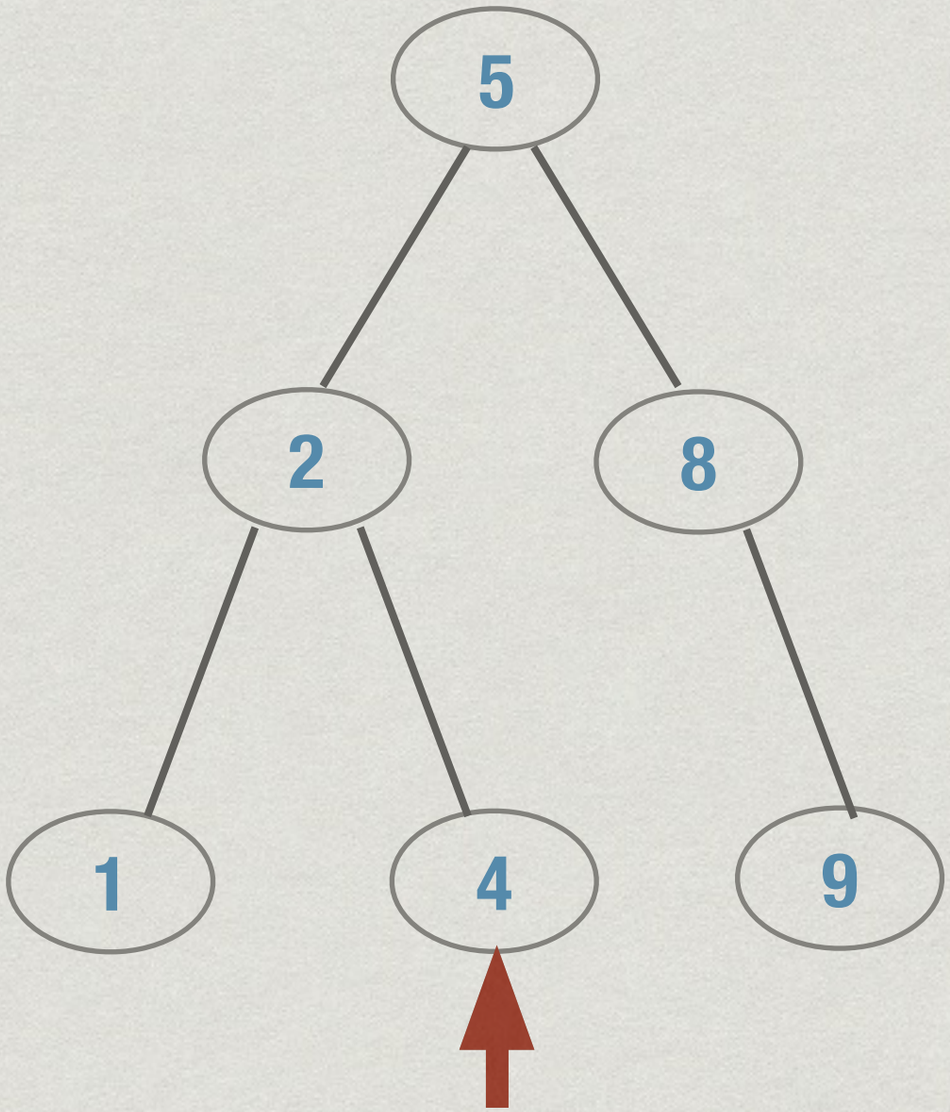
Inorder traversal

```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



Inorder traversal

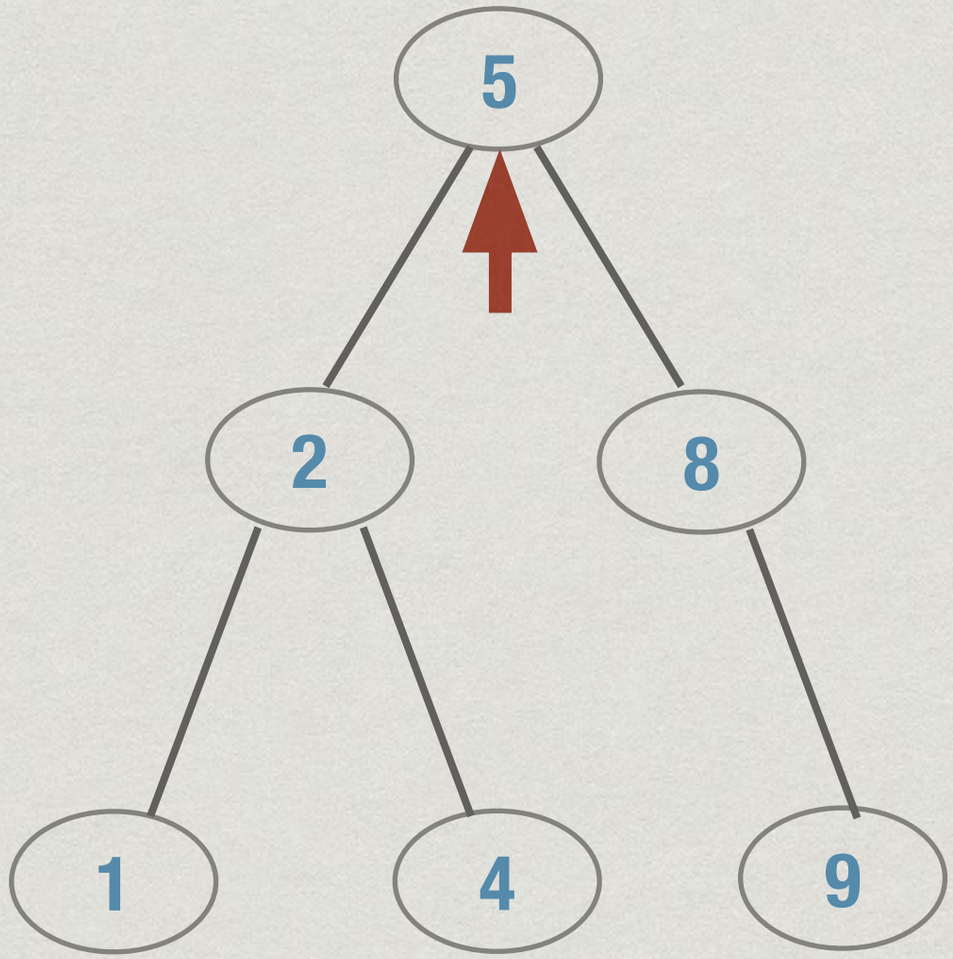
```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



1 2

Inorder traversal

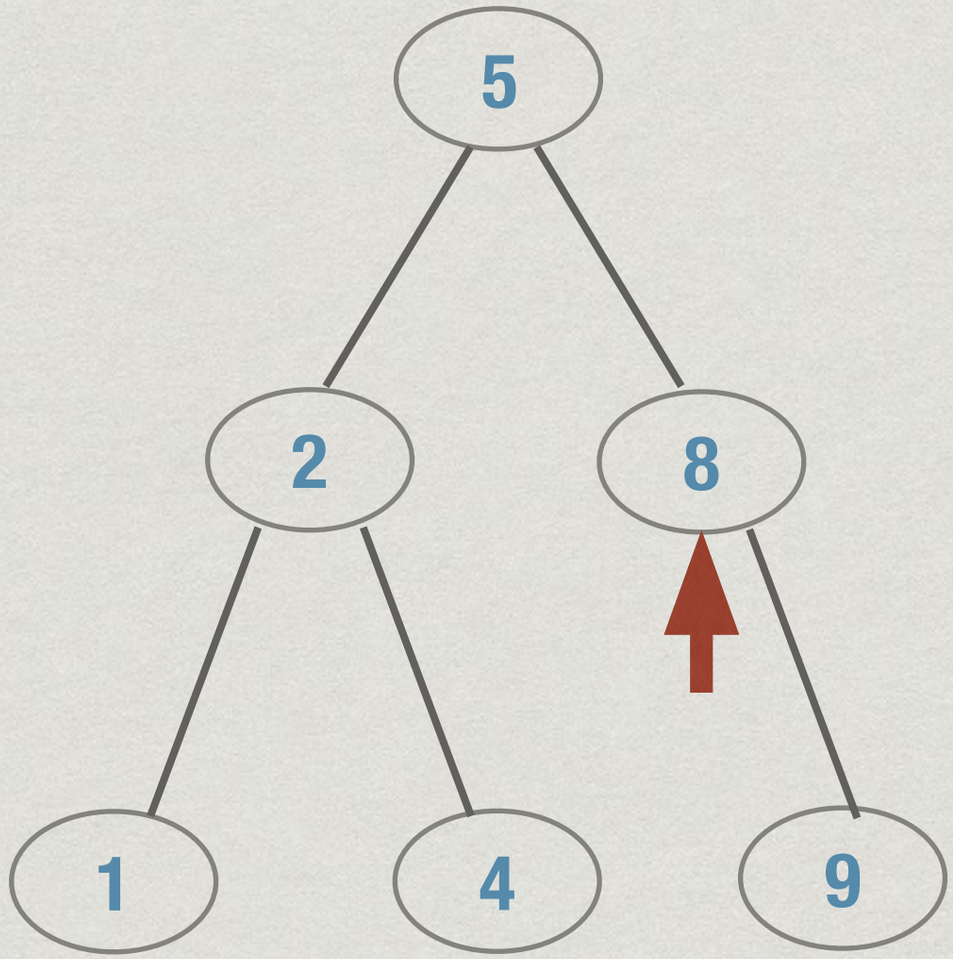
```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



1 2 4

Inorder traversal

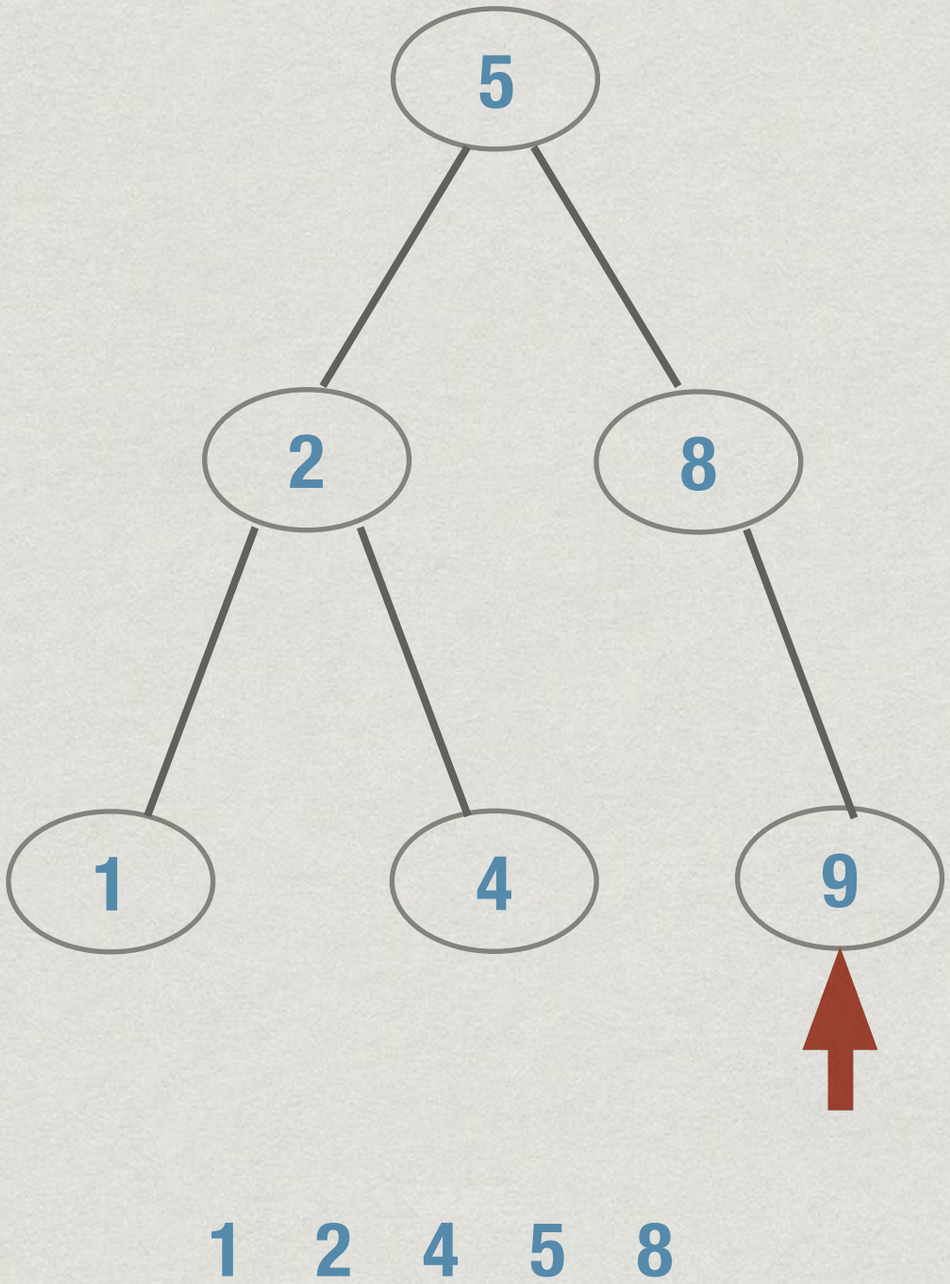
```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



1 2 4 5

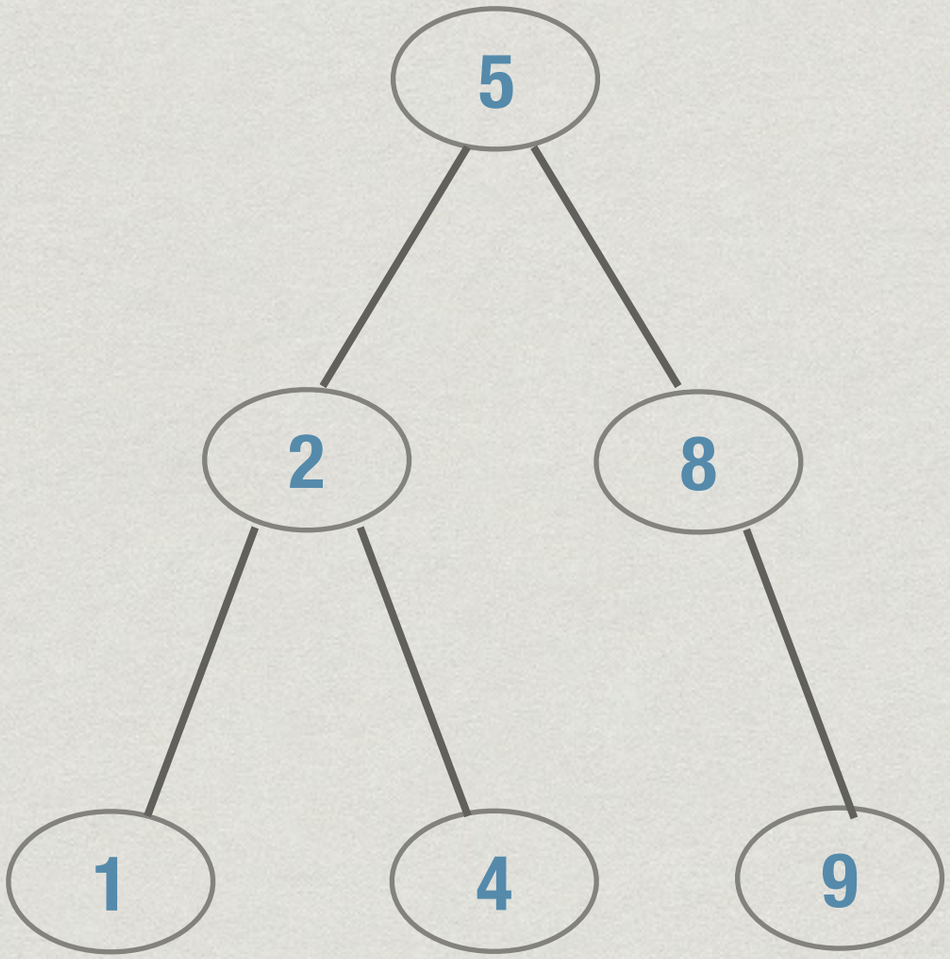
Inorder traversal

```
function inOrder(t)

if (t != NIL)

   inOrder(t.left)

   print(t.value)

   inOrder(t.right)
```



Inorder traversal

```
function inOrder(t)
if (t != NIL)
   inOrder(t.left)
   print(t.value)
   inOrder(t.right)
```



- Lists values in sorted order

1 2 4 5 8 9

find(v)

Recursive

```
function find(t,v)

if (t == NIL)
   return(False)

if (t.value == v)
   return(True)

if (v < t.value)
   return(find(t.left,v))
else
   return(find(t.right,v))
```

find(v)

Recursive

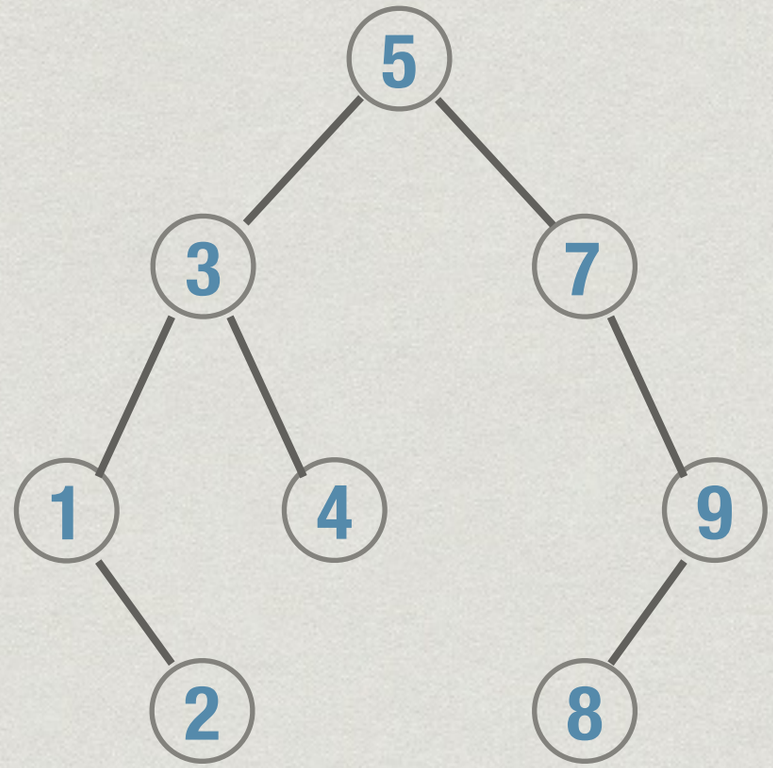
```
function find(t,v)

if (t == NIL)
   return(False)

if (t.value == v)
   return(True)

if (v < t.value)
   return(find(t.left,v))
else
   return(find(t.right,v))
```

Iterative

```
function find(t,v)

while (t != NIL) {

 if (t.value == v)
    return(True)

 if (v < t.value)
    t = t.left
 else
    t = t.right
}

return(False)
```

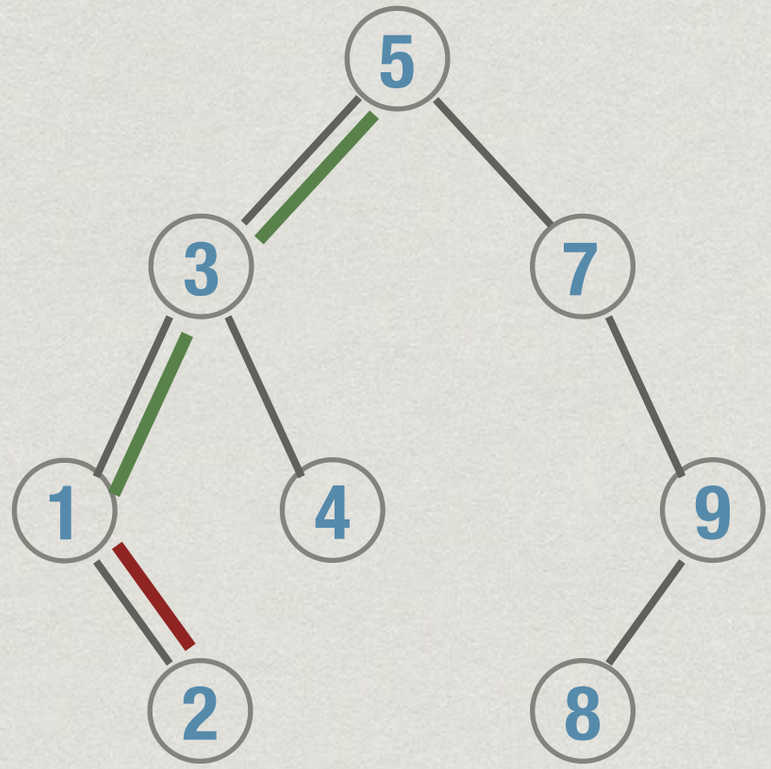
Minimum

- Left most node in the tree



Minimum

- Left most node in the tree

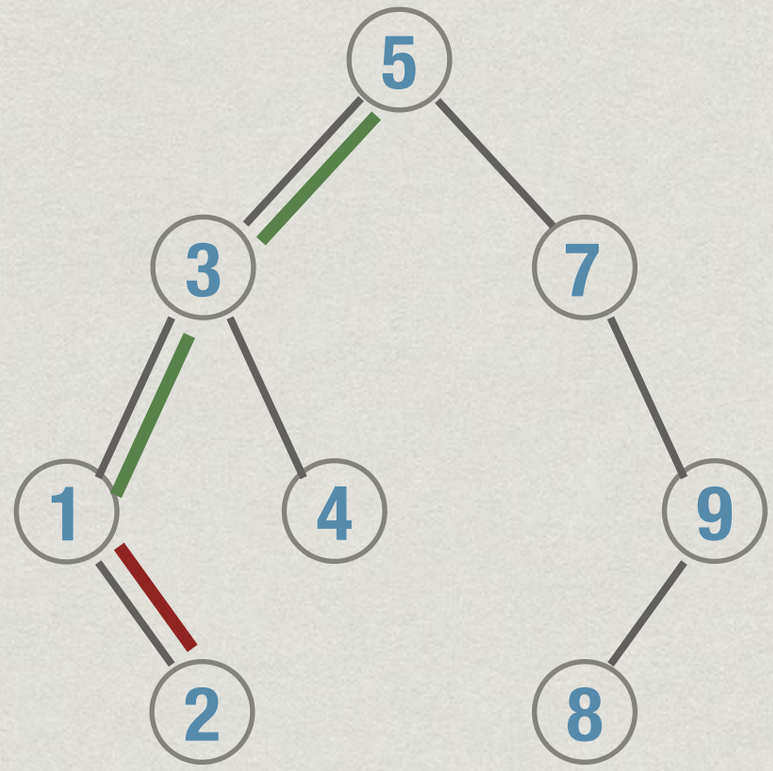


Minimum

- Left most node in the tree

Recursive

```
function minval(t)

# Assume t is not empty

if (t.left == NIL)
 return(t.value)
else
 return(minval(t.left))
```

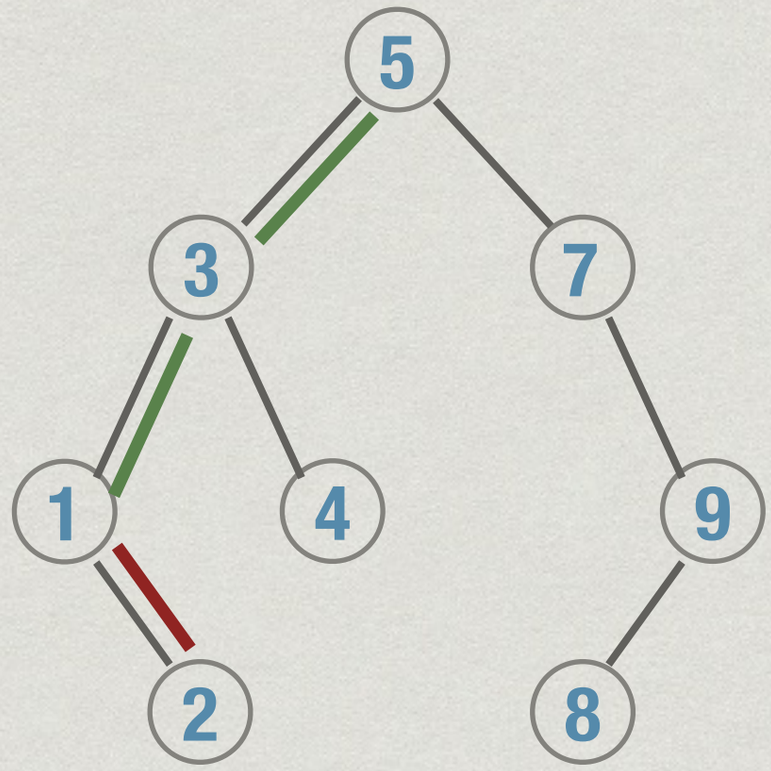


Minimum

- Left most node in the tree

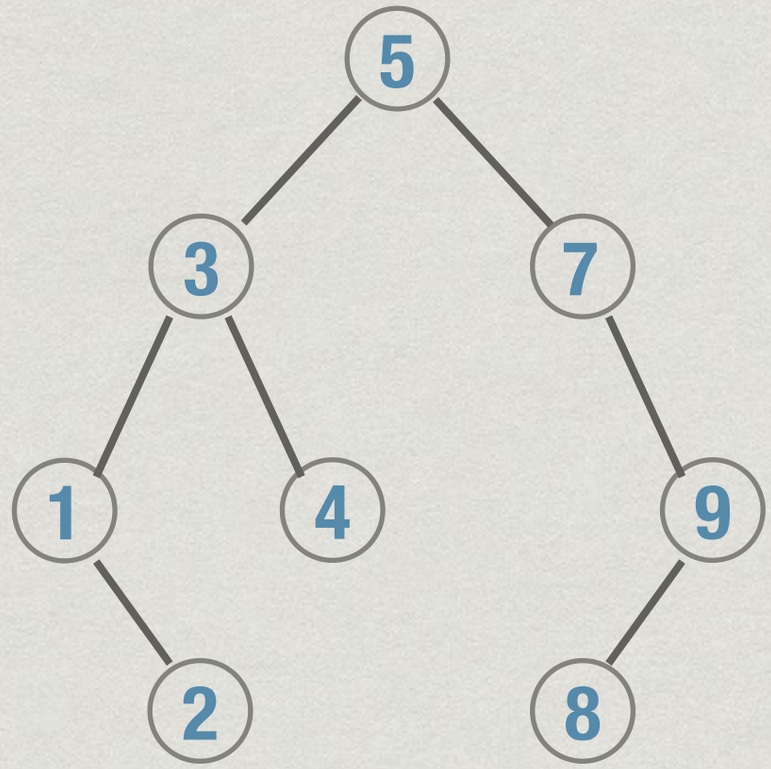
Iterative

```
function minval(t)

# Assume t is not empty

while (t.left != NIL)
 t = t.left

return(t.value)
```



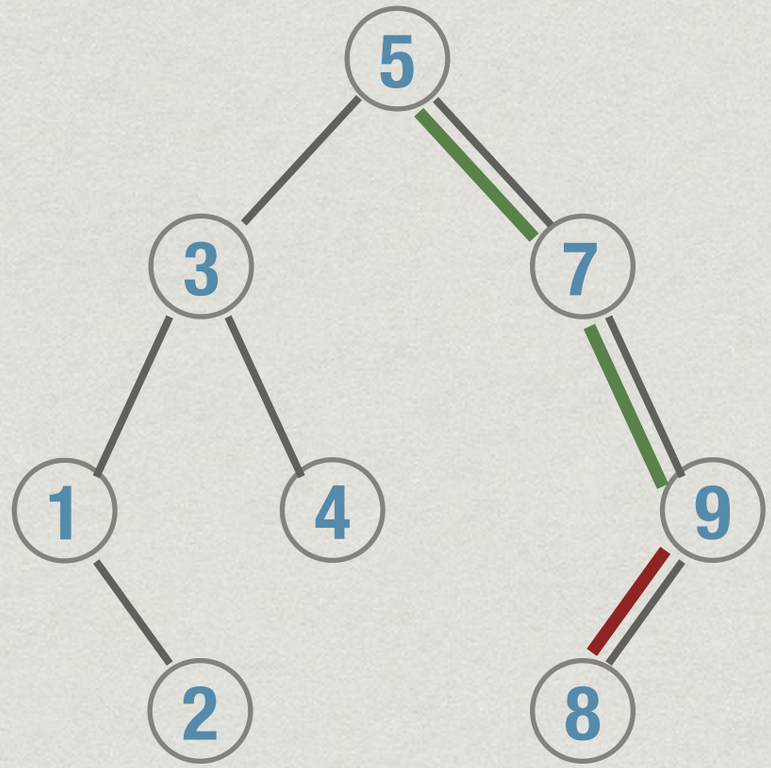
Maximum

- Right most node in the tree



Maximum

- Right most node in the tree

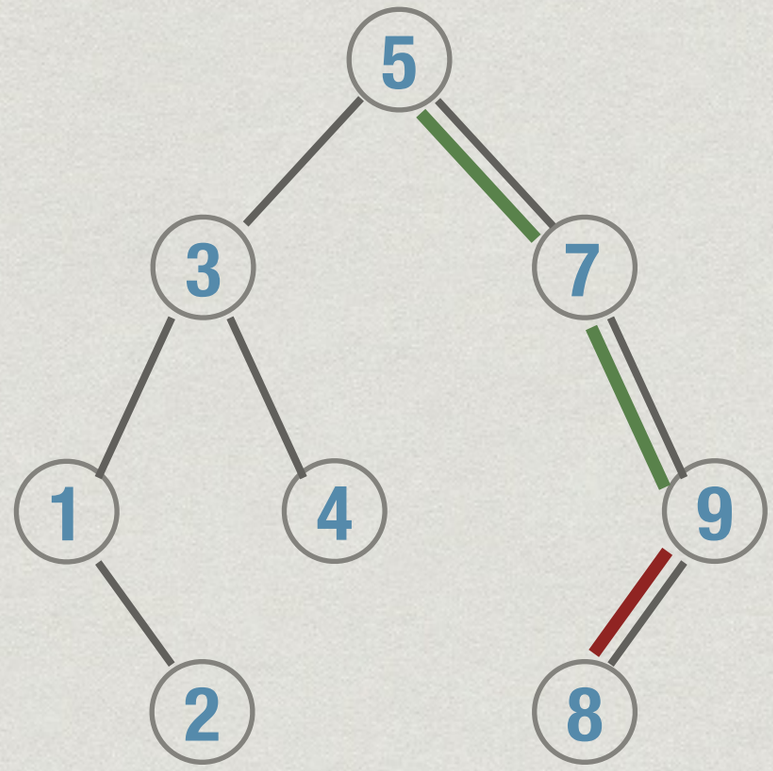


Maximum

- Right most node in the tree

Recursive

```
function maxval(t)

# Assume t is not empty

if (t.right == NIL)
 return(t.value)
else
 return(maxval(t.right))
```

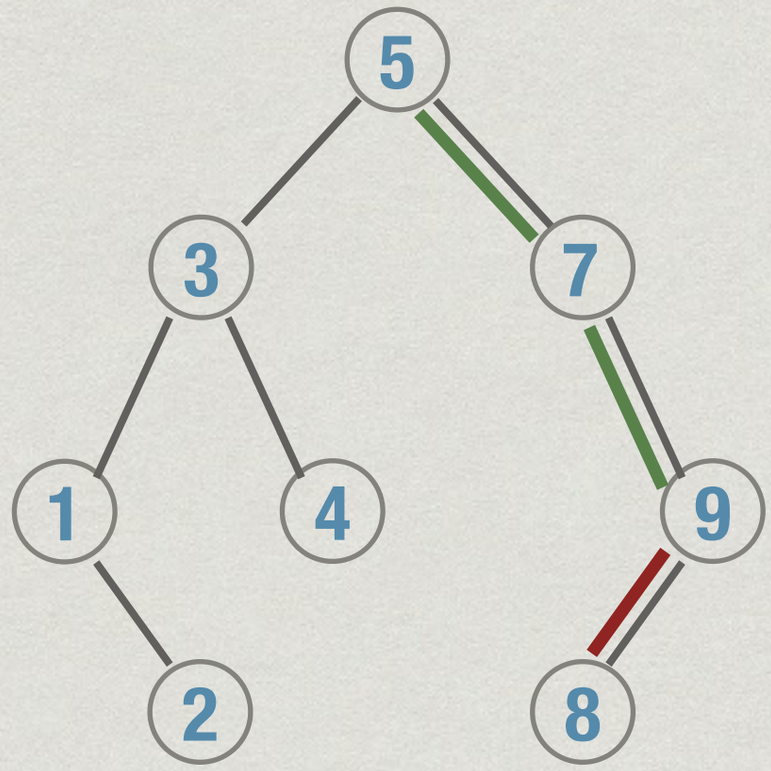


Maximum

- Right most node in the tree

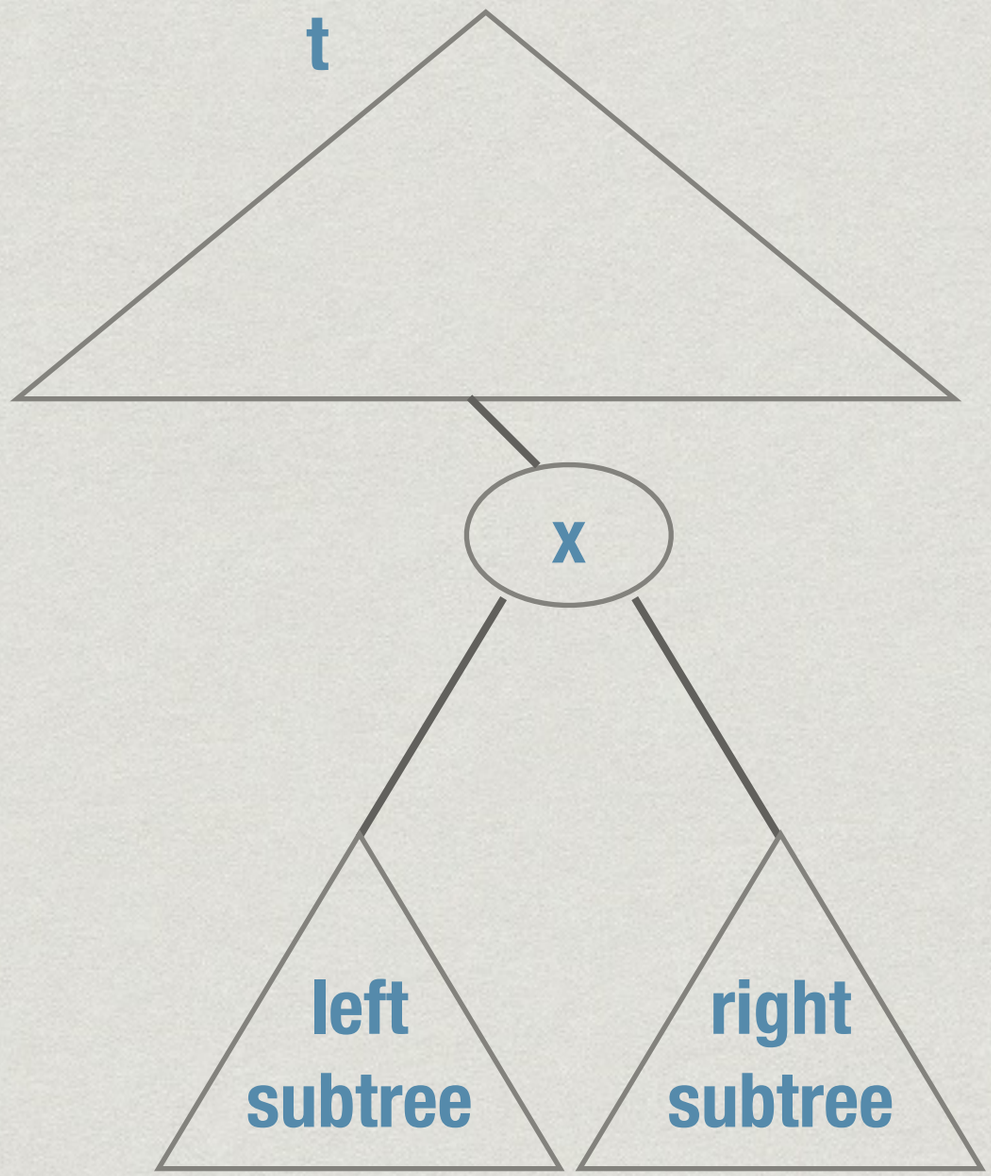
Iterative

```
function maxval(t)

# Assume t is not empty

while (t.right != NIL)
  t = t.right

return(t.value)
```



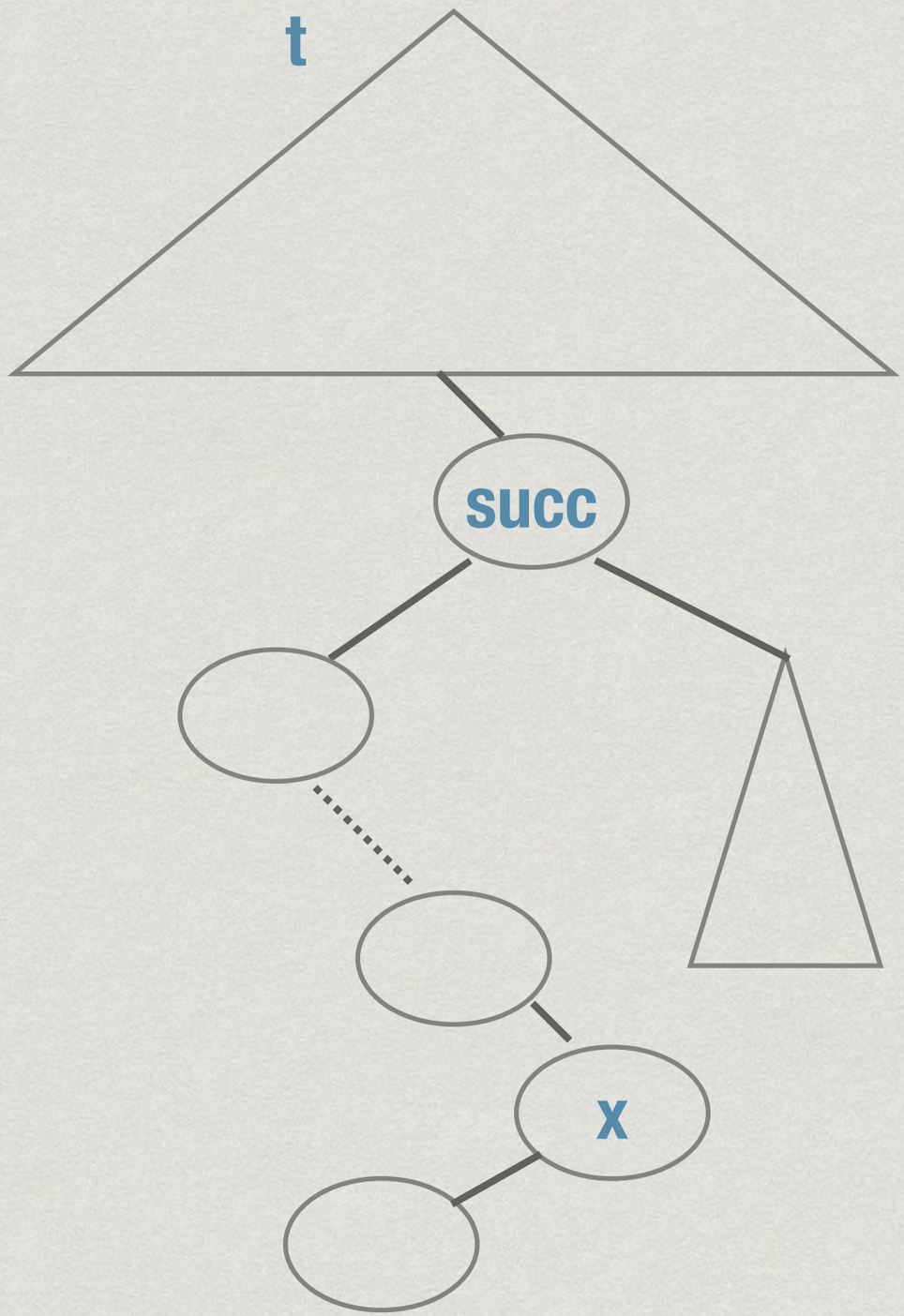
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)



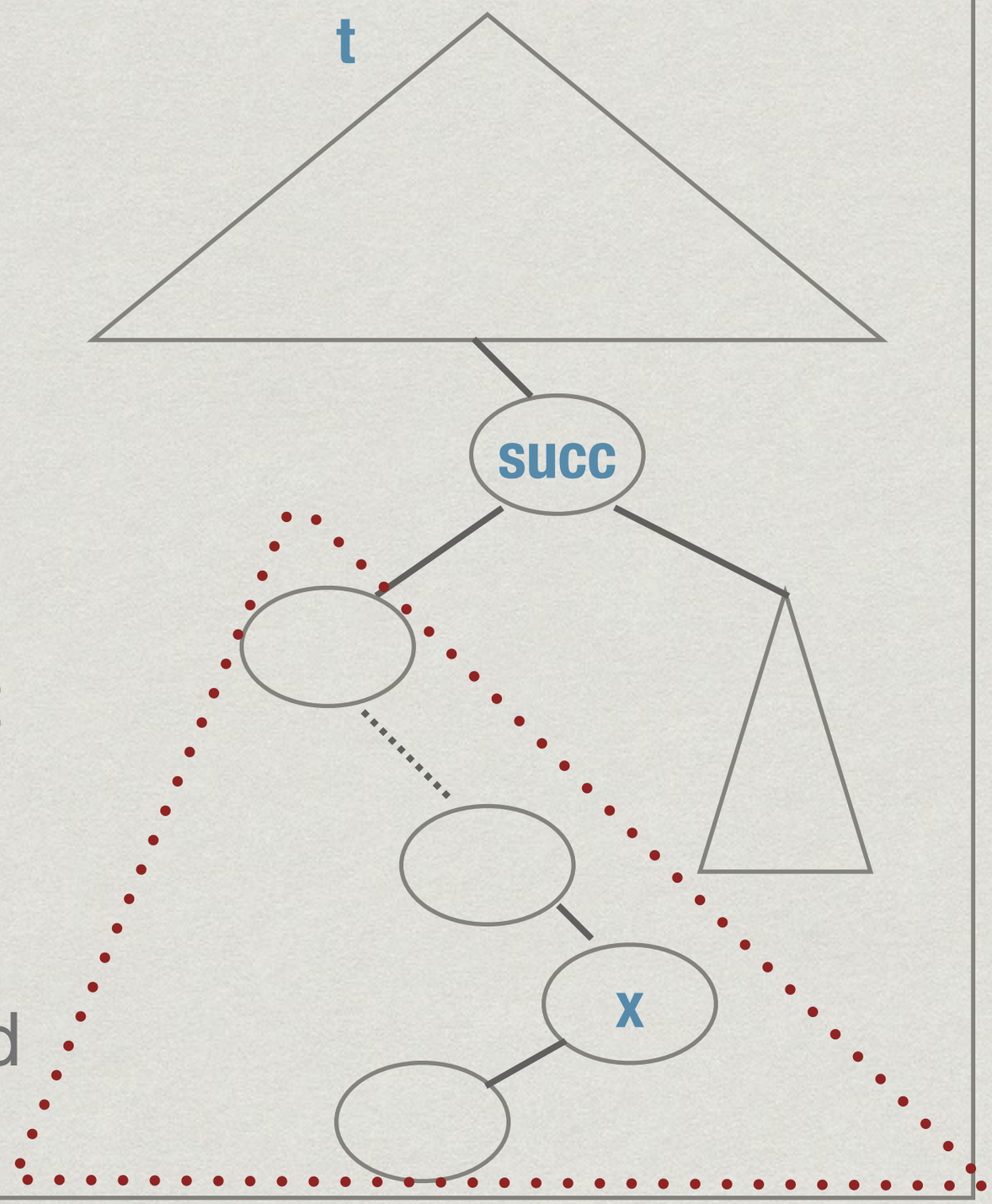
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)
- if x has no right subtree
  - x is max of the subtree it belongs to
  - walk up to find where this subtree is connected



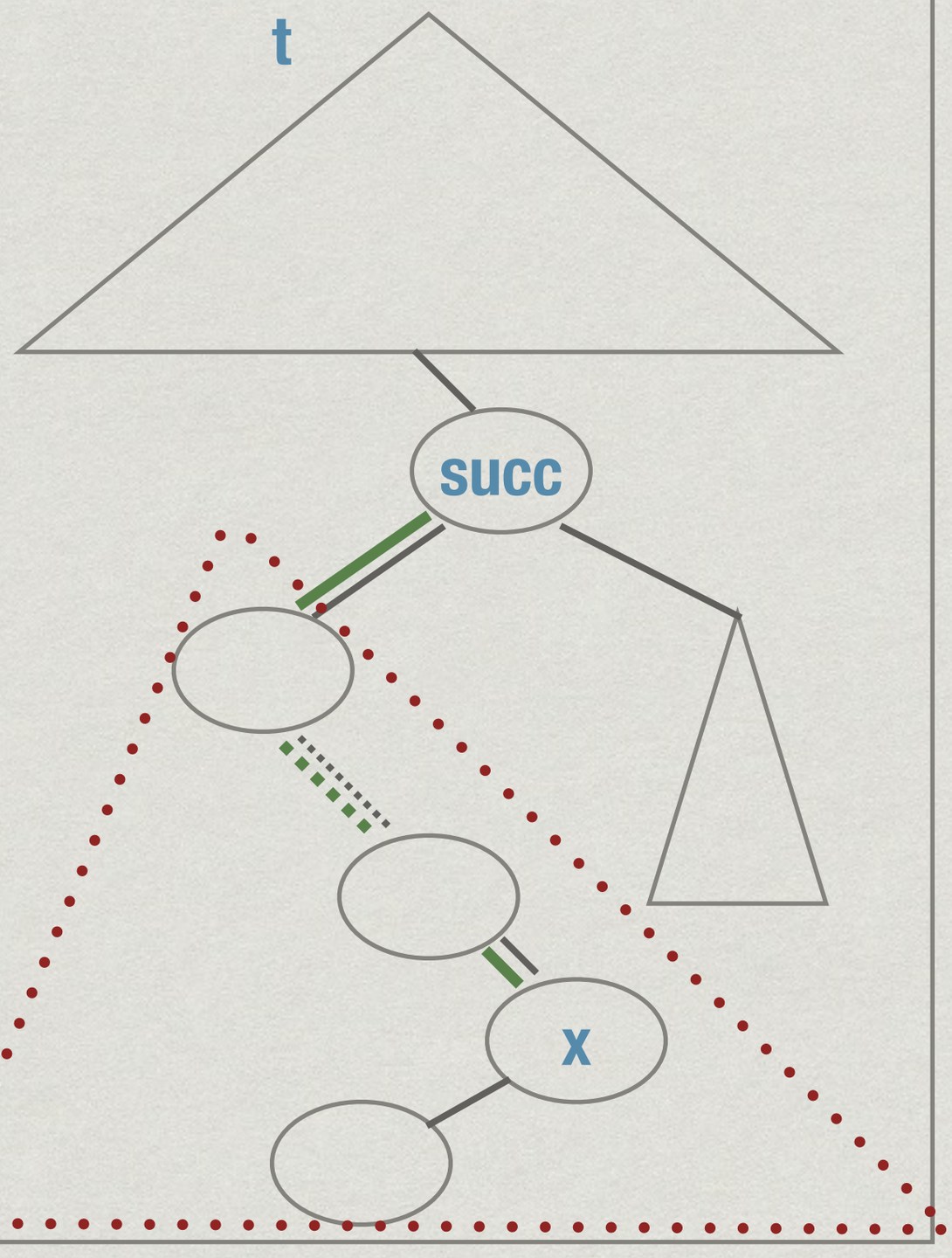
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)
- if x has no right subtree
  - x is max of the subtree it belongs to
  - walk up to find where this subtree is connected



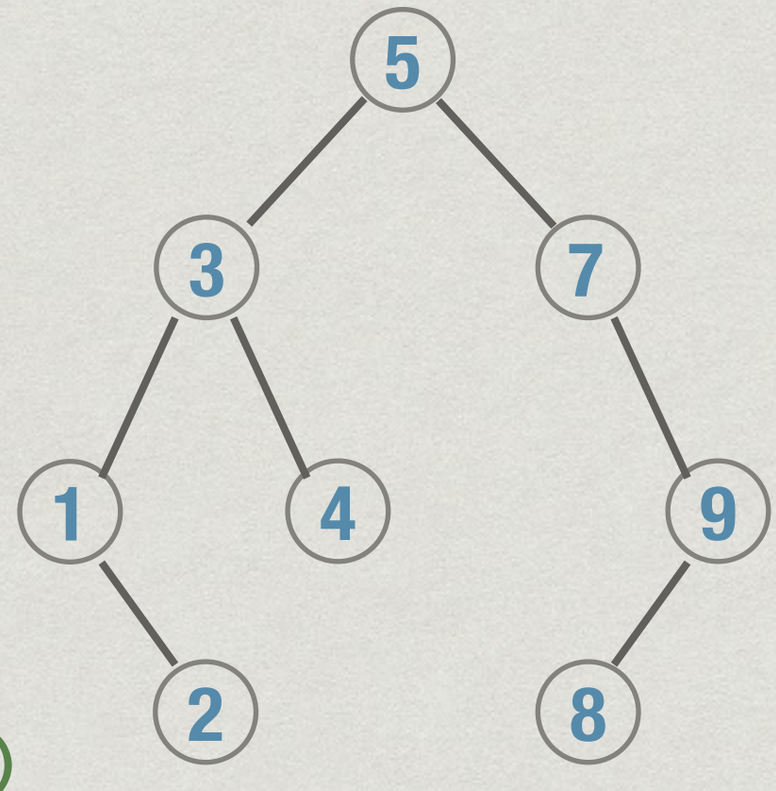
Successor

- succ(x) is what inorder(t) prints after x
- If x has a right subtree, min(right subtree)
- if x has no right subtree
  - x is max of the subtree it belongs to
  - walk up to find where this subtree is connected



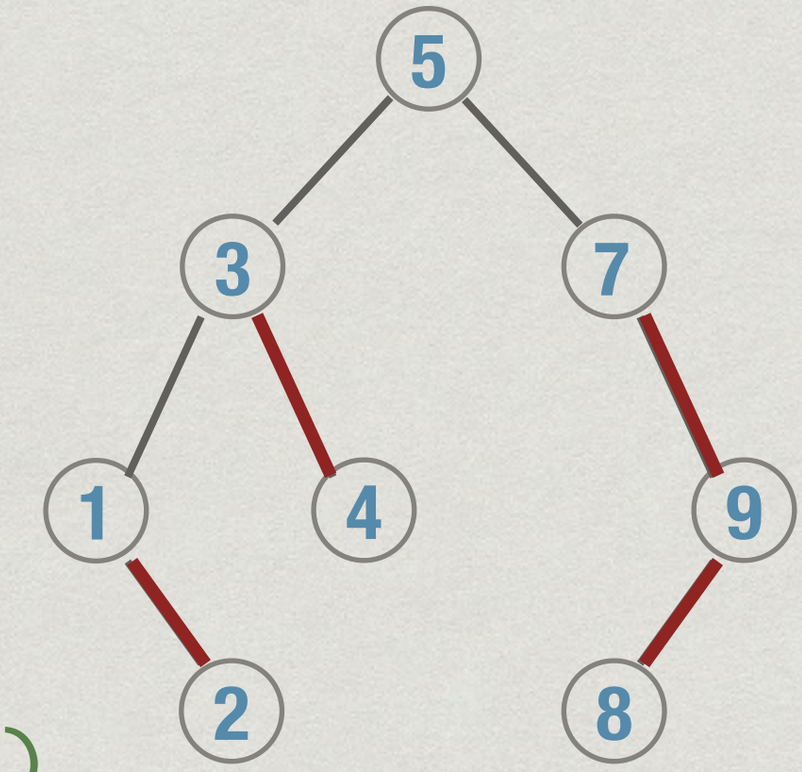
Successor

```
function succ(t)

if (t.right != NIL)
   return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
   t = y
   y = y.parent

return(y)
```



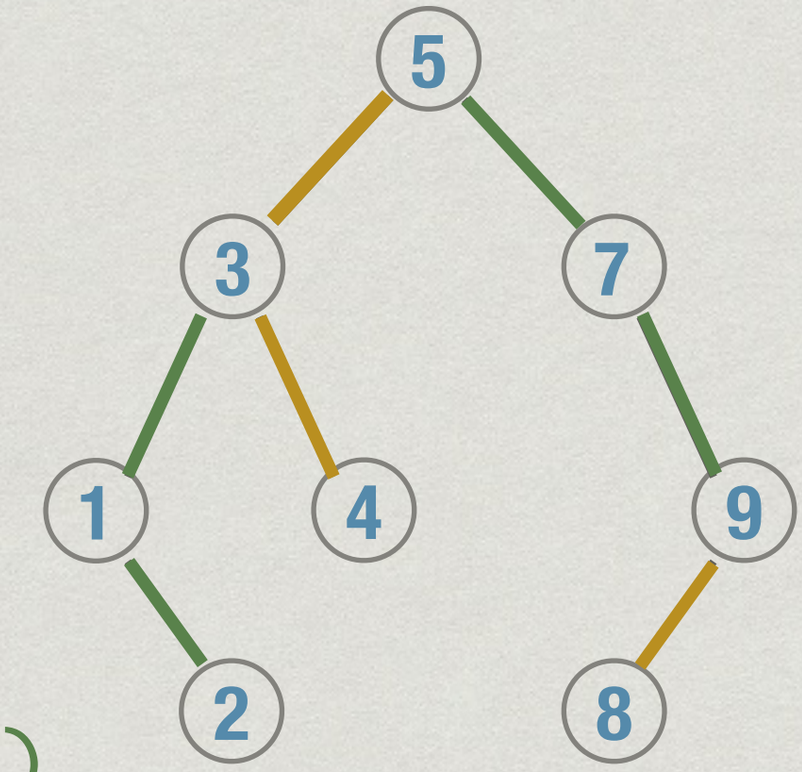
Successor

```
function succ(t)

if (t.right != NIL)
   return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
   t = y
   y = y.parent

return(y)
```



Successor

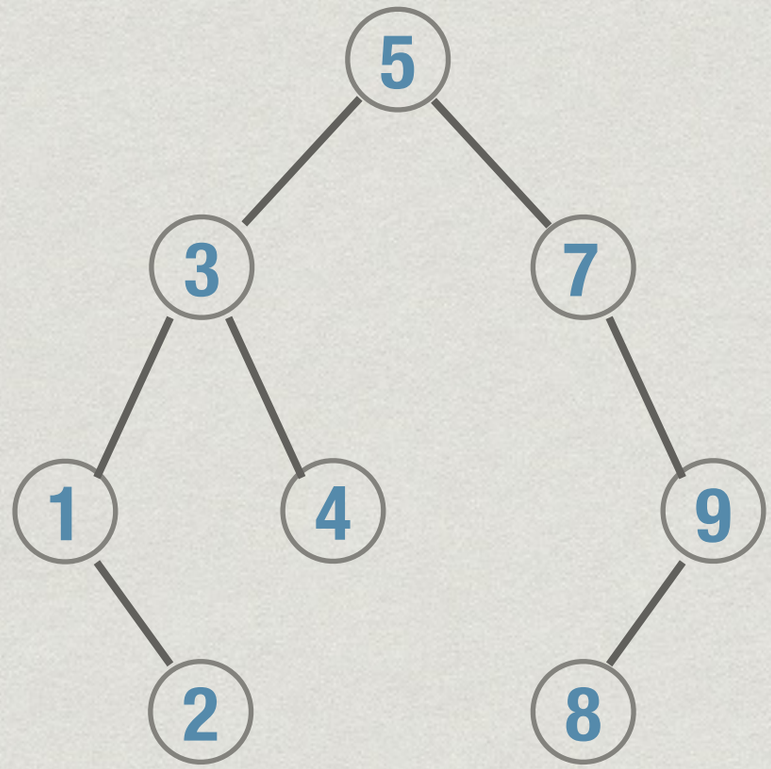
```
function succ(t)

if (t.right != NIL)
   return(minval(t.right))

y = t.parent

while (y != NIL and t == y.right)
   t = y
   y = y.parent

return(y)
```



Predecessor

- Symmetric

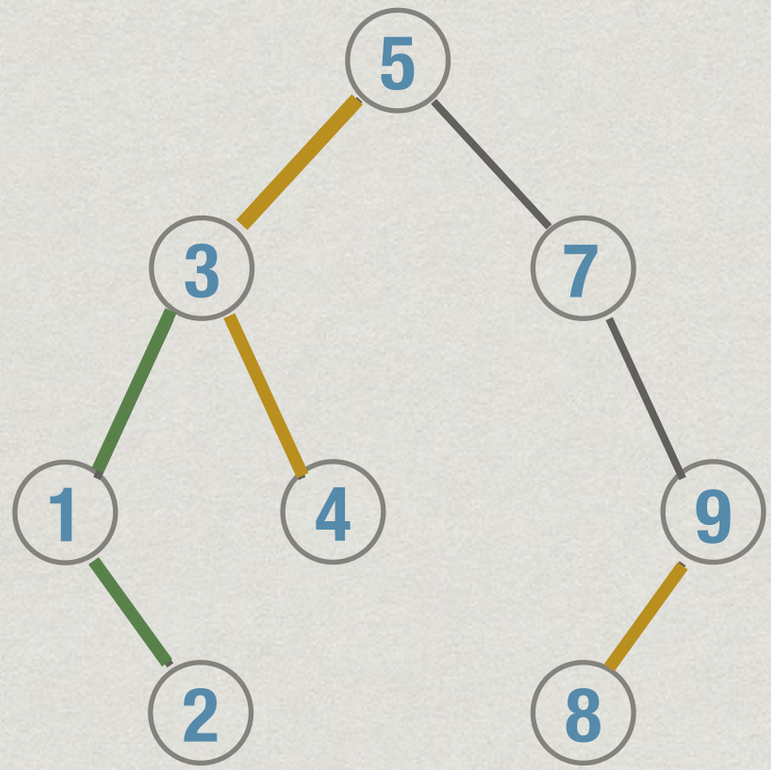
```
function pred(t)

if (t.left != NIL)
   return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
   t = y
   y = y.parent

return(y)
```



Predecessor

- Symmetric

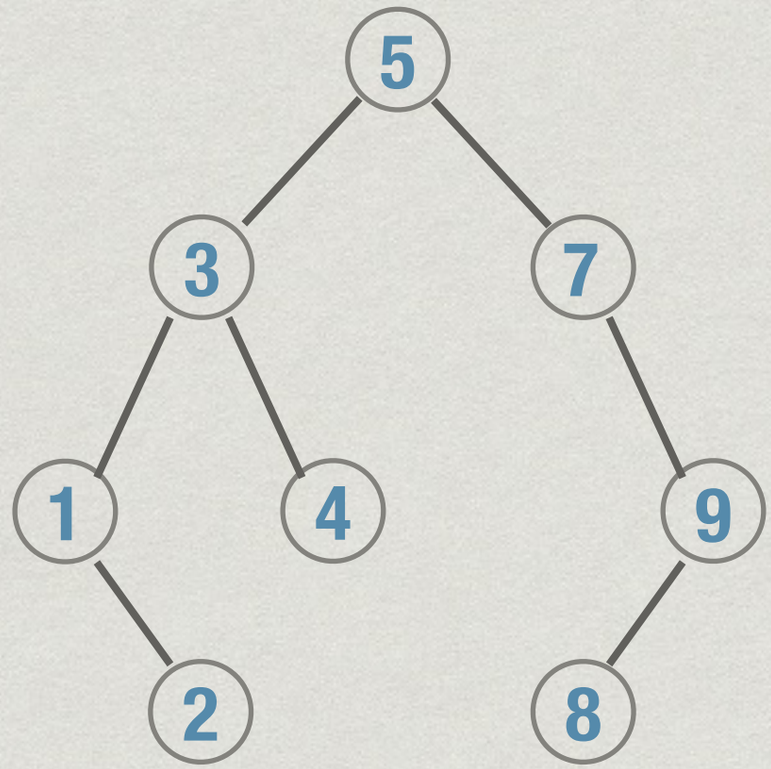
```
function pred(t)

if (t.left != NIL)
   return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
   t = y
   y = y.parent

return(y)
```



Predecessor

- Symmetric

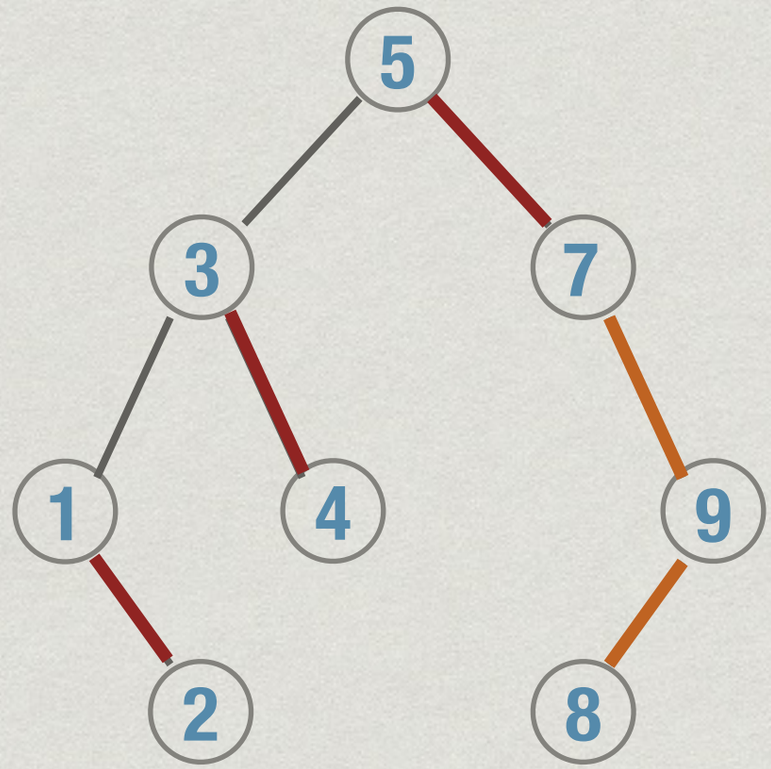
```
function pred(t)

if (t.left != NIL)
   return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
   t = y
   y = y.parent

return(y)
```



Predecessor

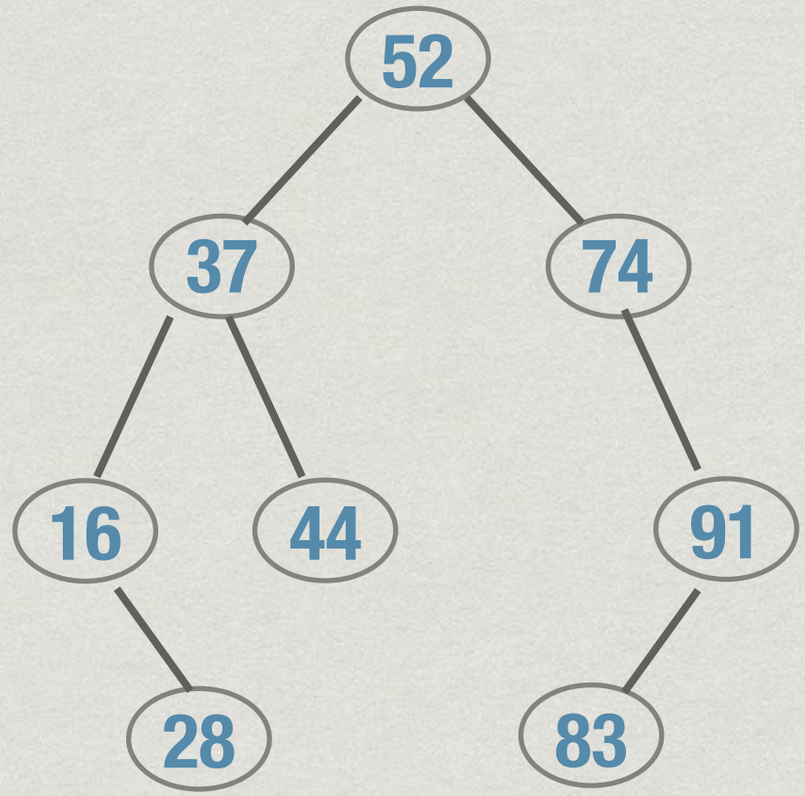
- Symmetric

```
function pred(t)

if (t.left != NIL)
   return(maxval(t.left))

y = t.parent

while (y != NIL and t == y.left)
   t = y
   y = y.parent

return(y)
```



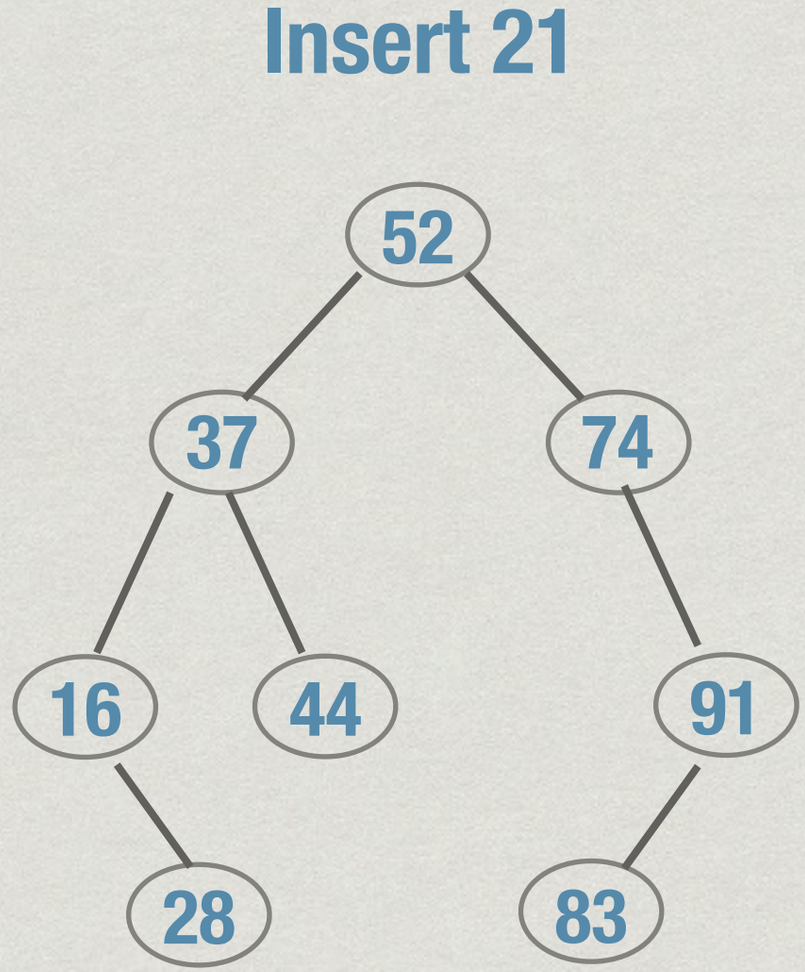
Insert

- Try to find v
- If it is not present, add it where the search fails



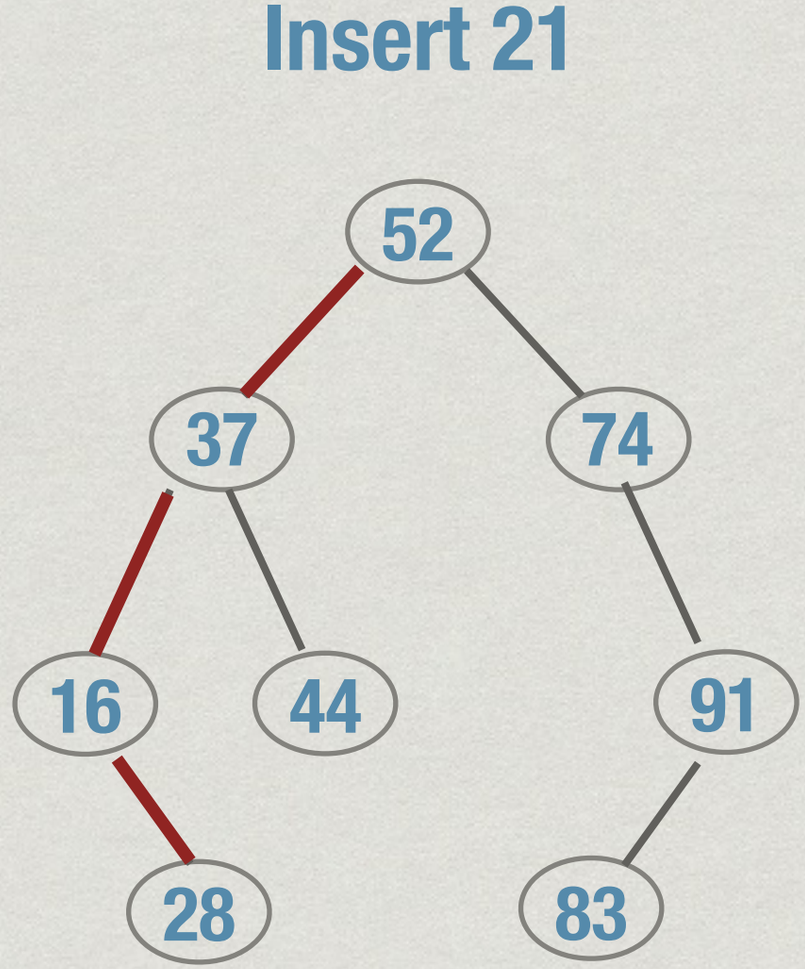
Insert

- Try to find v
- If it is not present, add it where the search fails



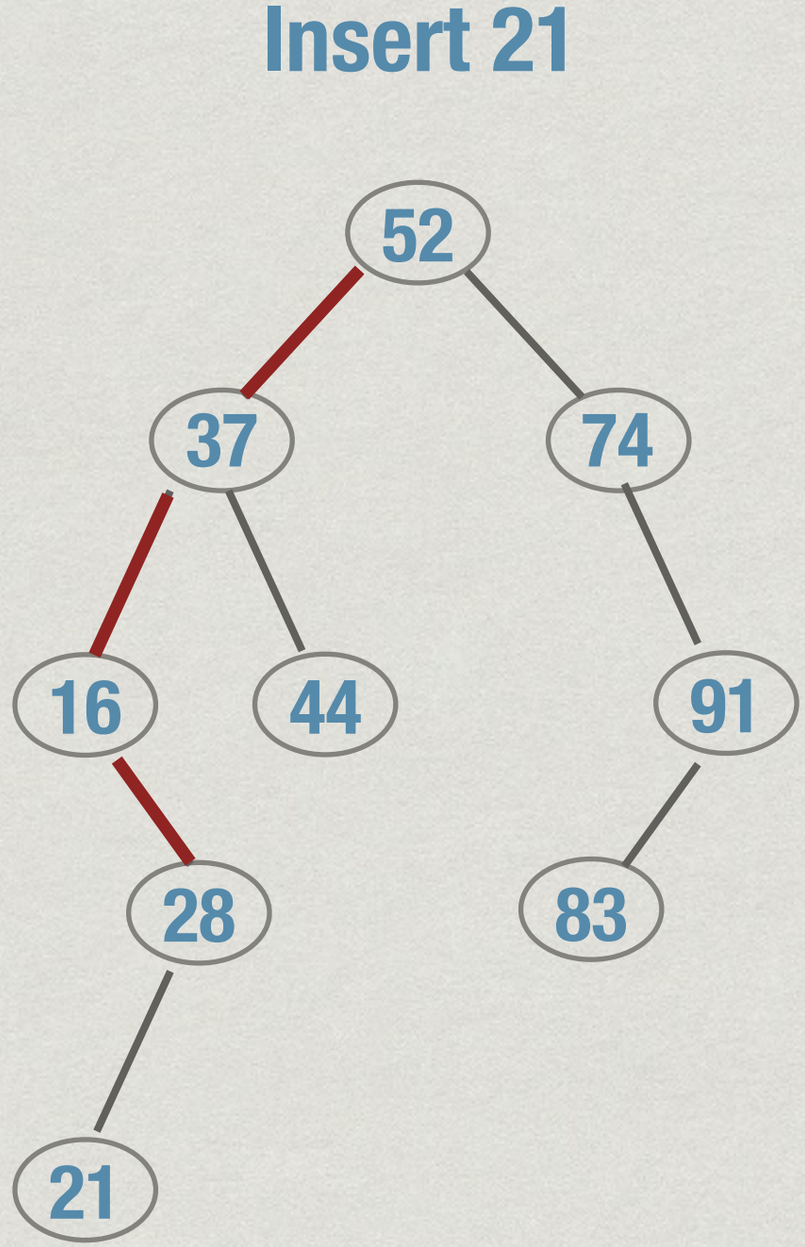
Insert

- Try to find v
- If it is not present, add it where the search fails



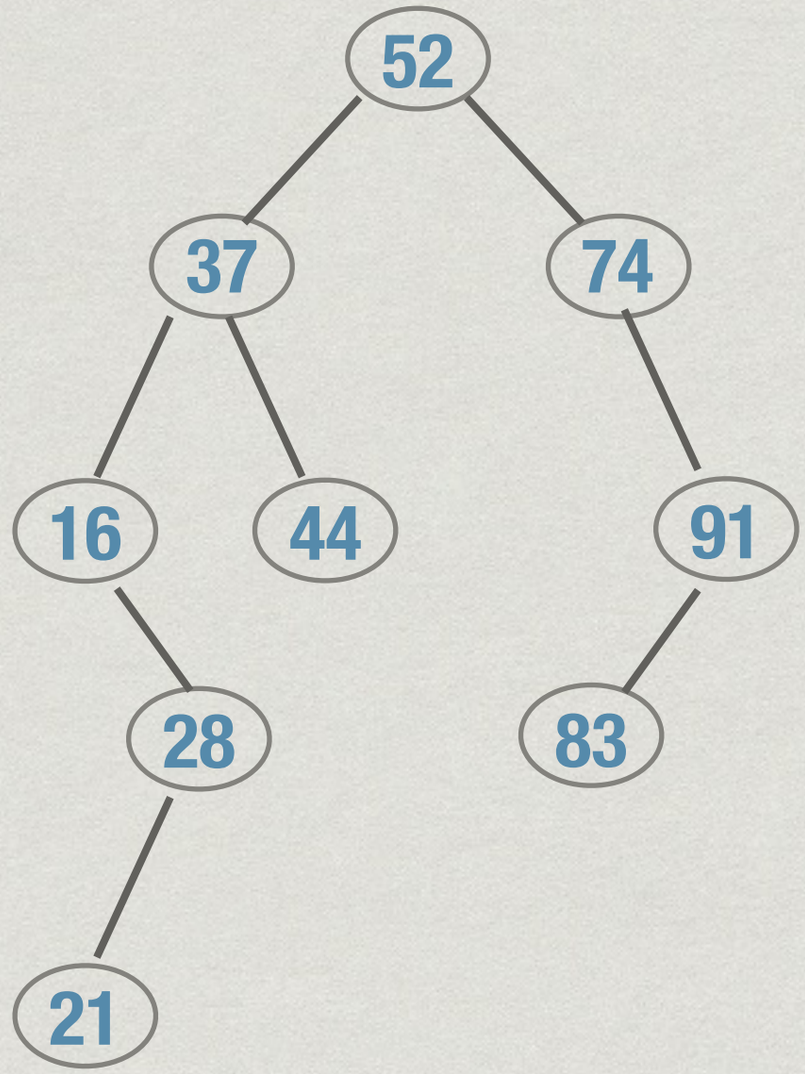
Insert

- Try to find v
- If it is not present, add it where the search fails



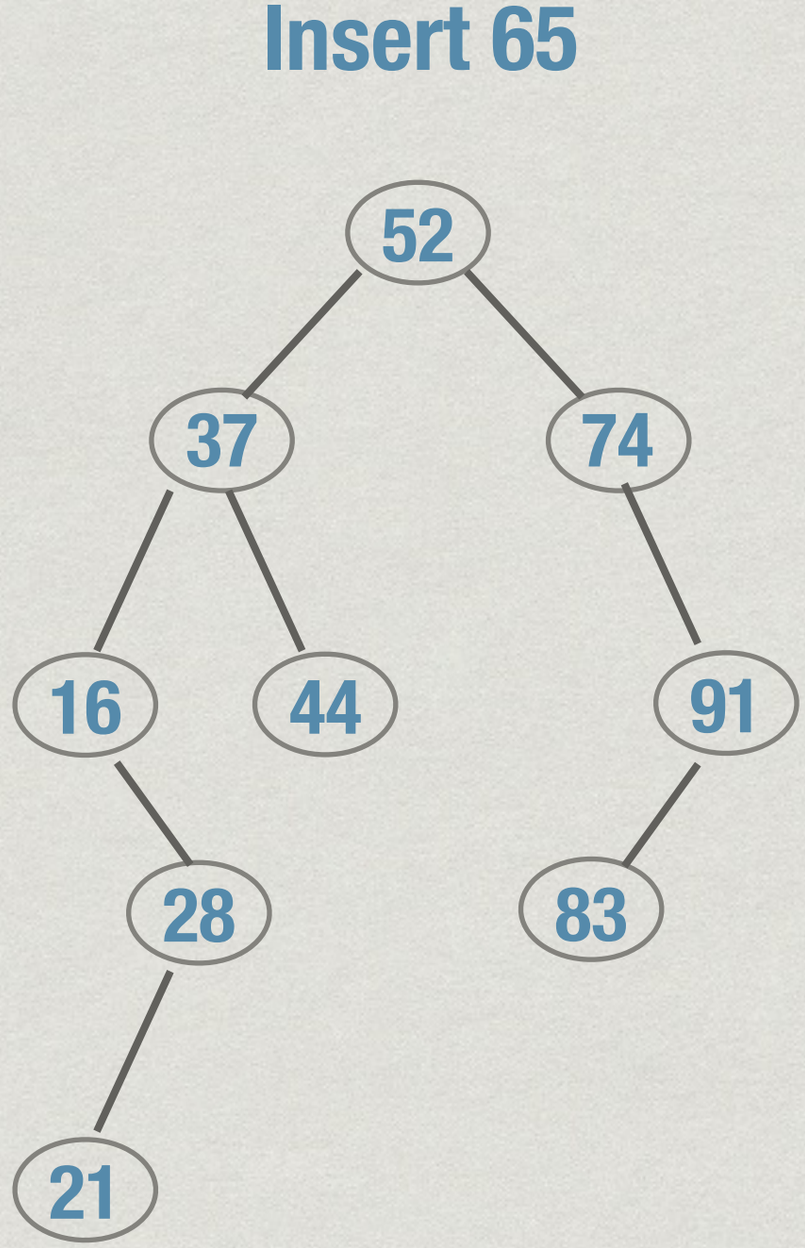
Insert

- Try to find v
- If it is not present, add it where the search fails



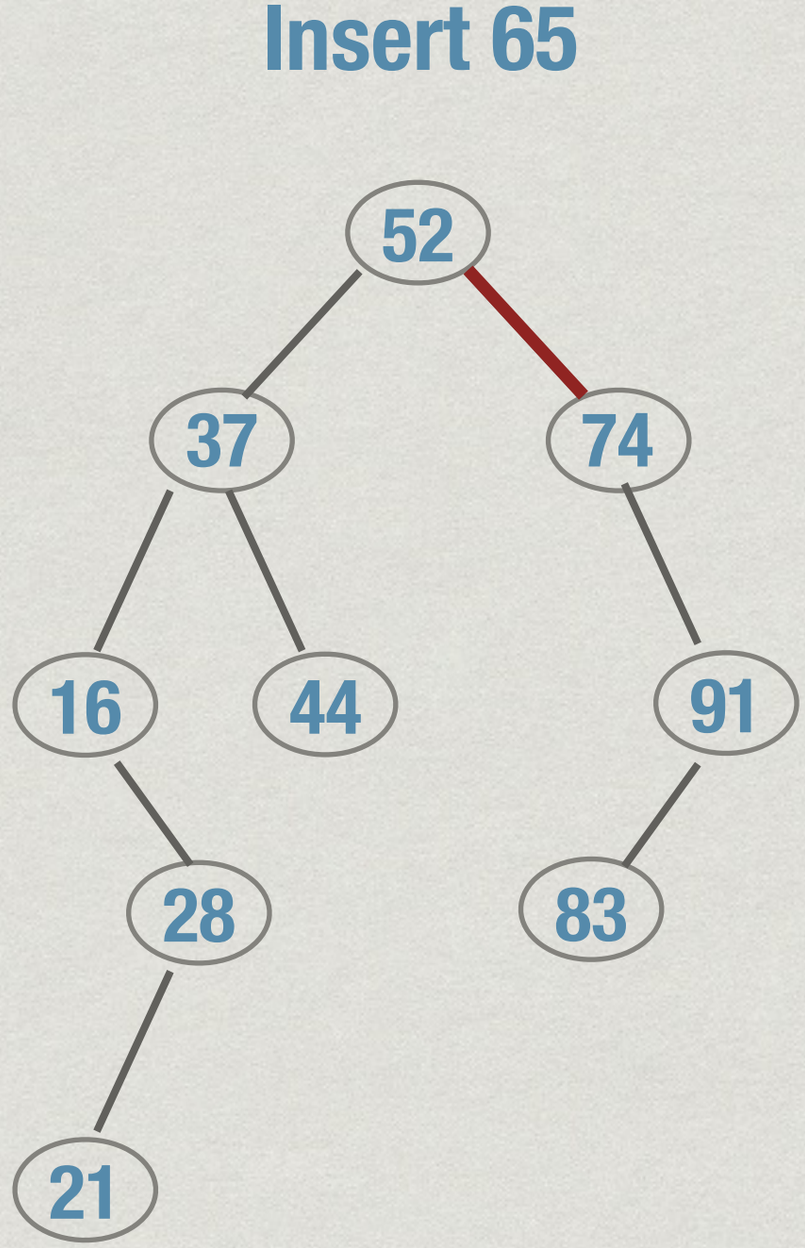
Insert

- Try to find v
- If it is not present, add it where the search fails



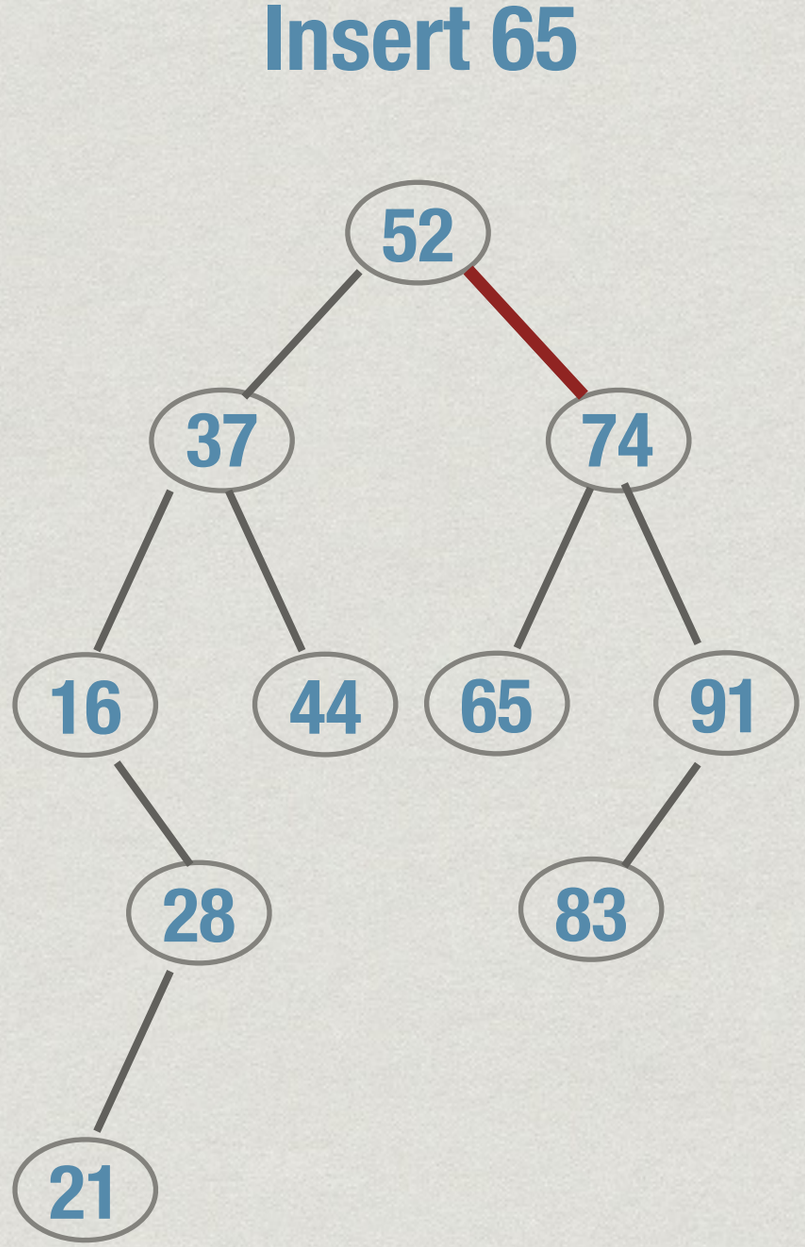
Insert

- Try to find v
- If it is not present, add it where the search fails



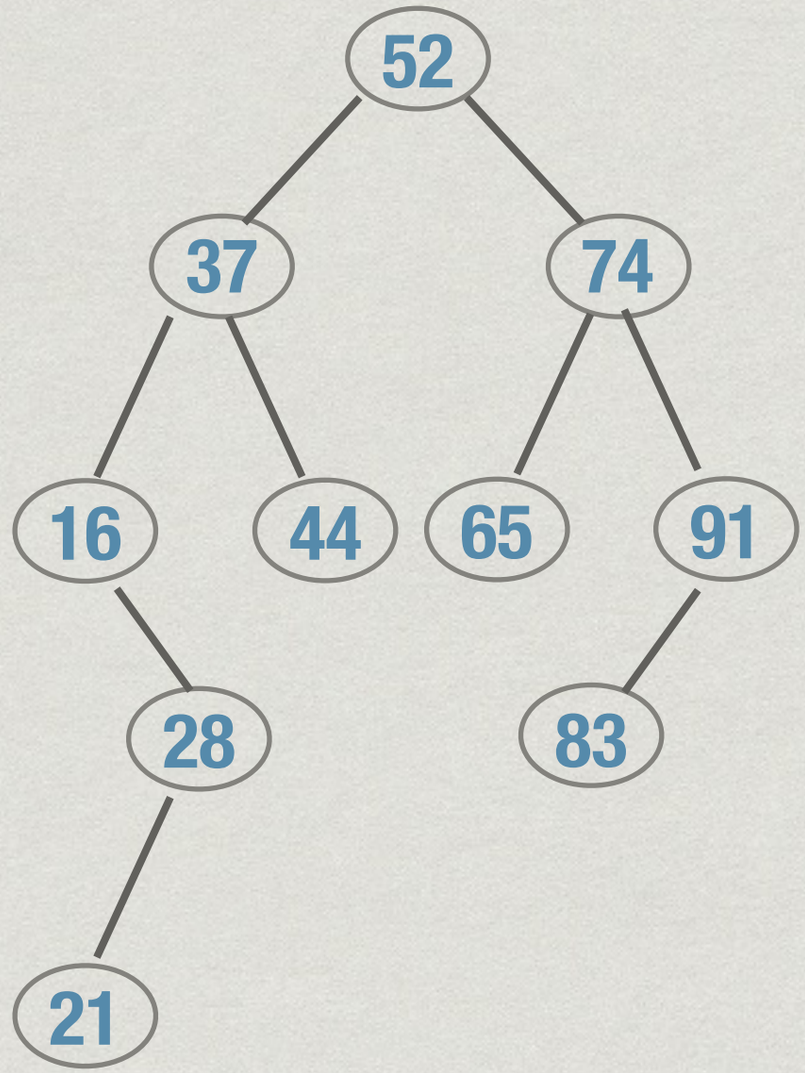
Insert

- Try to find v
- If it is not present, add it where the search fails



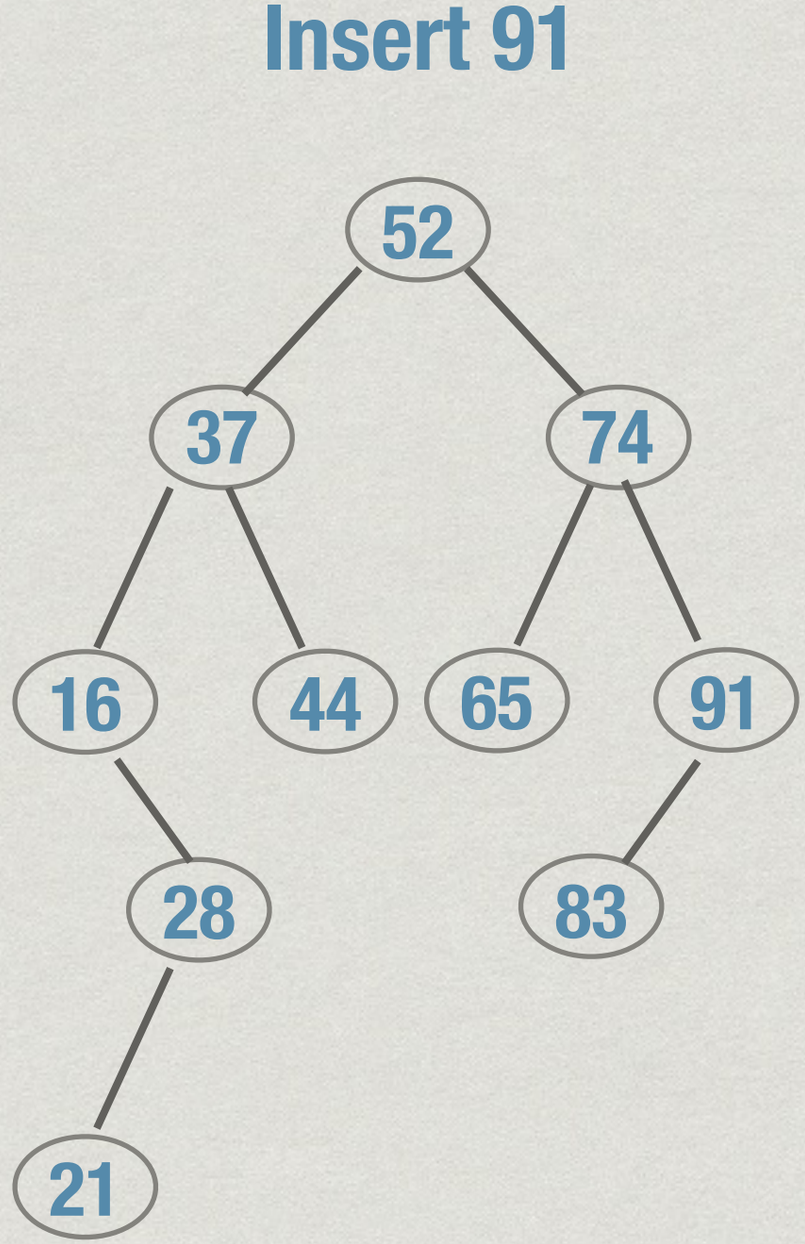
Insert

- Try to find v
- If it is not present, add it where the search fails



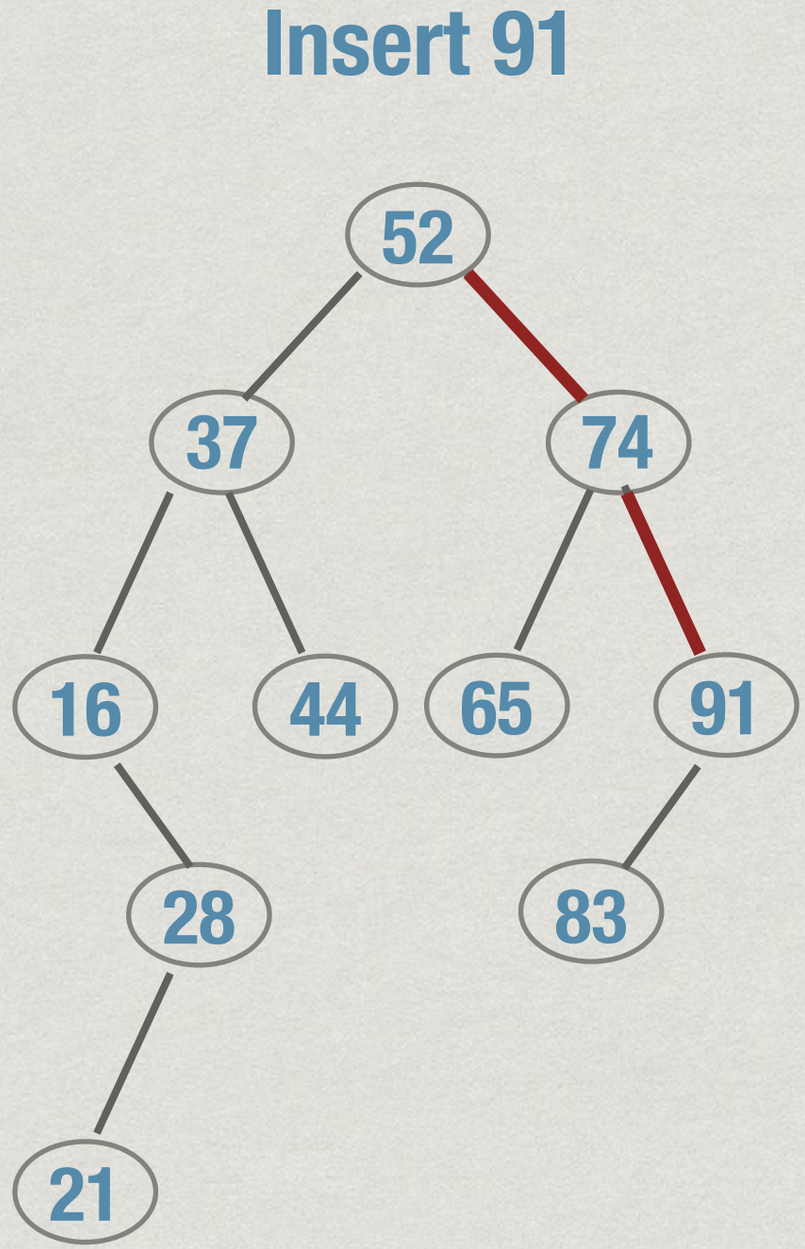
Insert

- Try to find v
- If it is not present, add it where the search fails



Insert

- Try to find v
- If it is not present, add it where the search fails

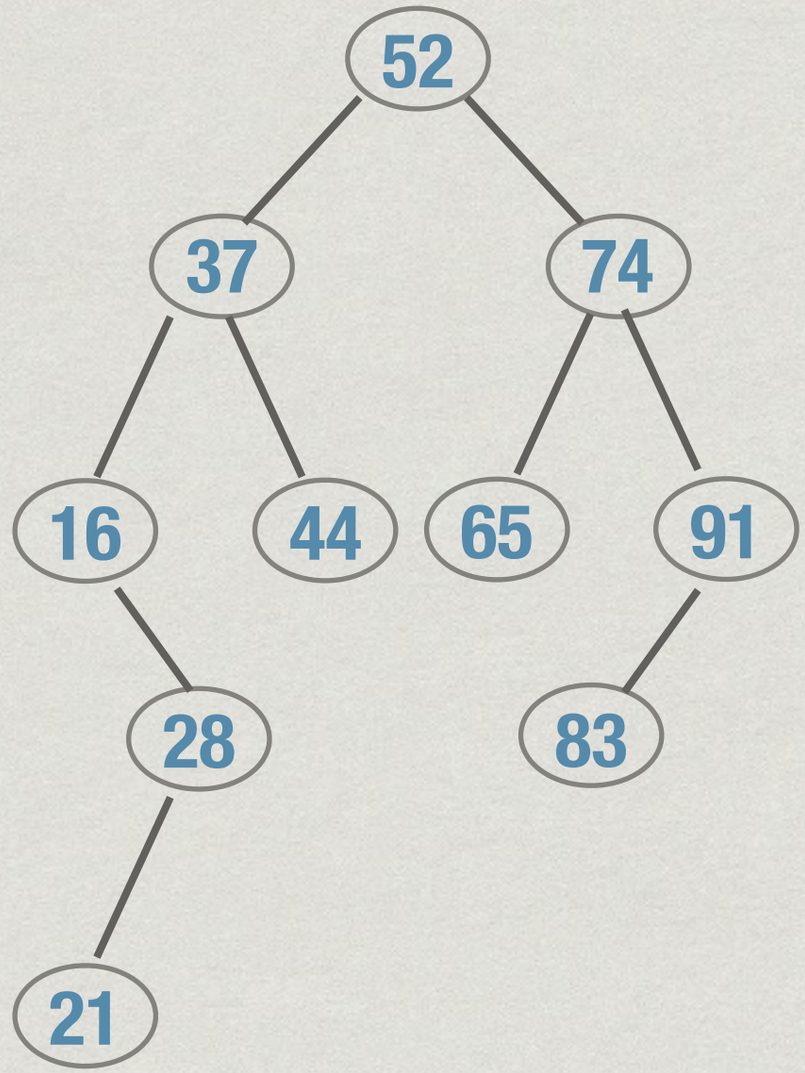


insert(v)

```
function insert(t,v)

if (t == NIL)
  t = Node(v); return  # Node(v) : isolated node, value v

if (t.value == v) return

if (v < t.value)
  if (t.left == NIL) # Add a left child with value v
    t.left = Node(v); t.left.parent = t; return
  else                 # Recursively insert in left subtree
    insert(t.left,v); return
else
  if (t.right == NIL) # Add a right child with value v
    t.right = Node(v); t.right.parent = t; return
  else                 # Recursively insert in right subtree
    insert(t.right,v)
```

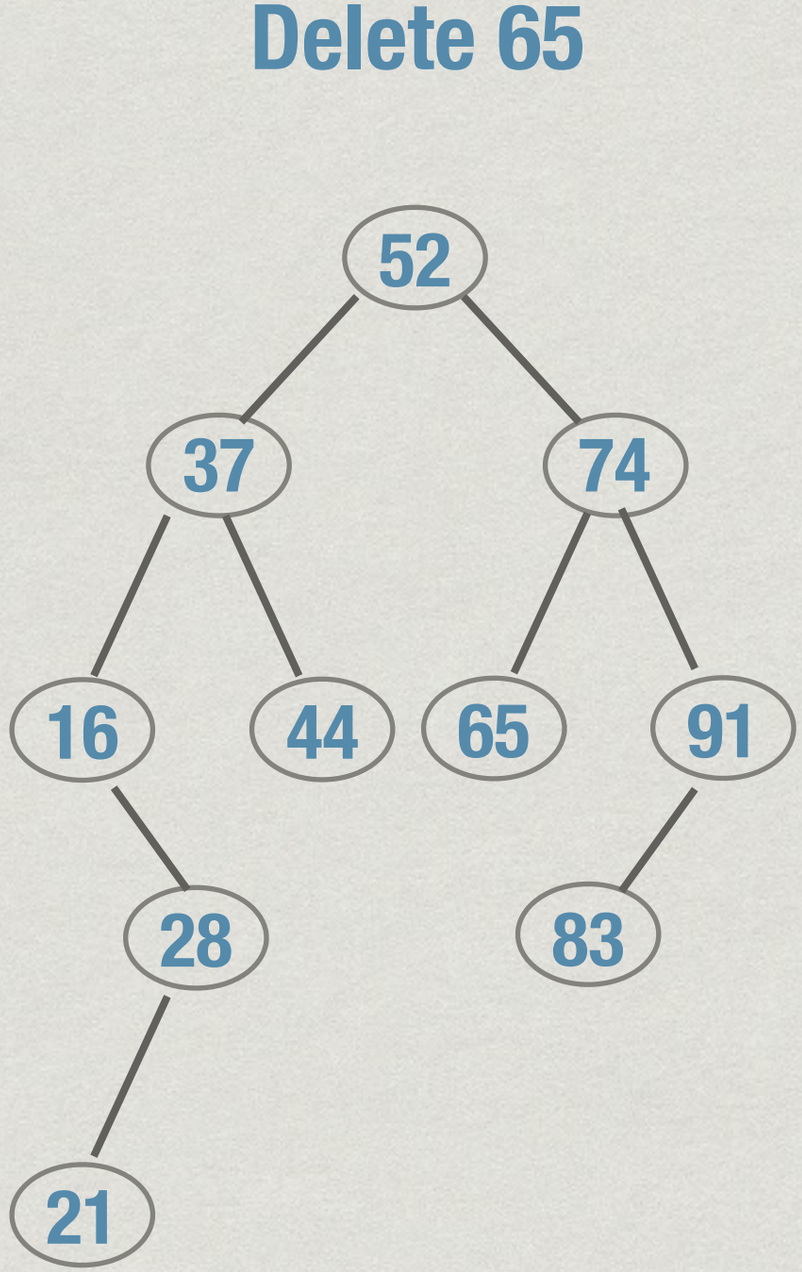
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



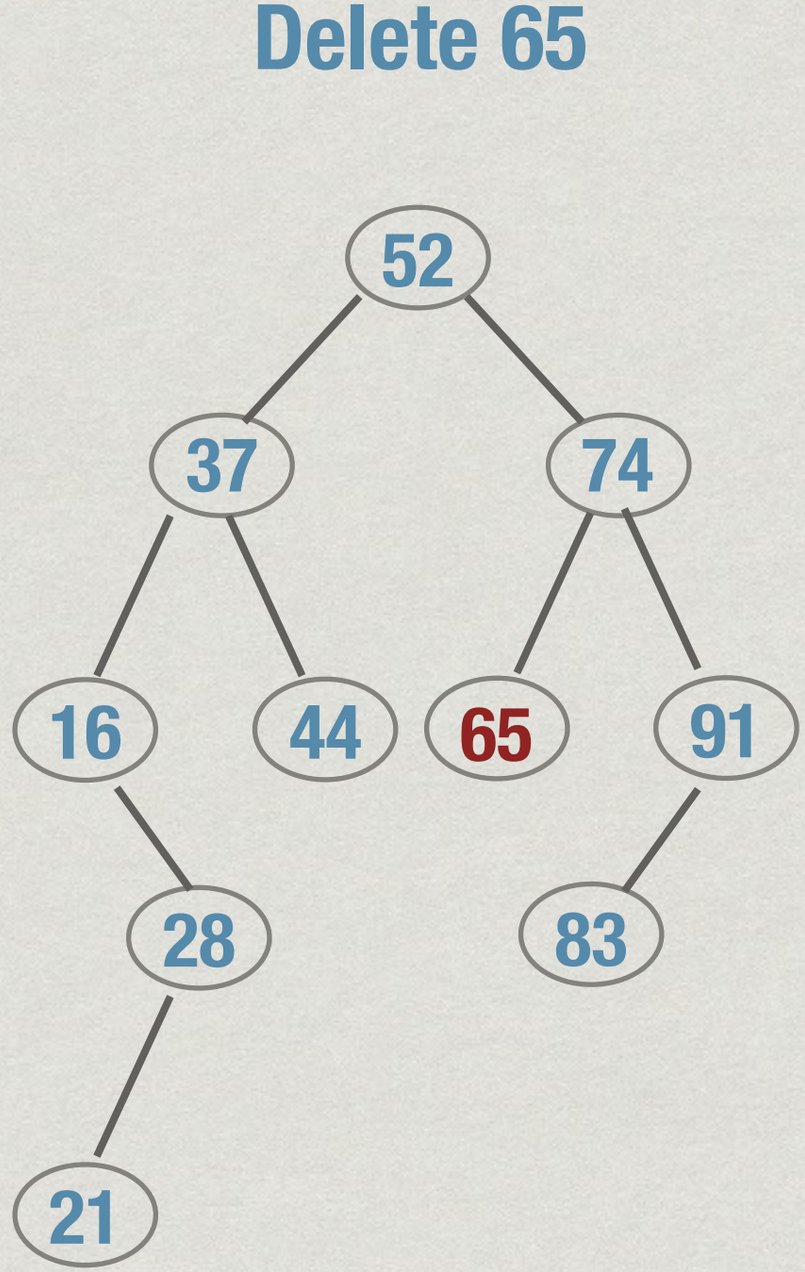
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



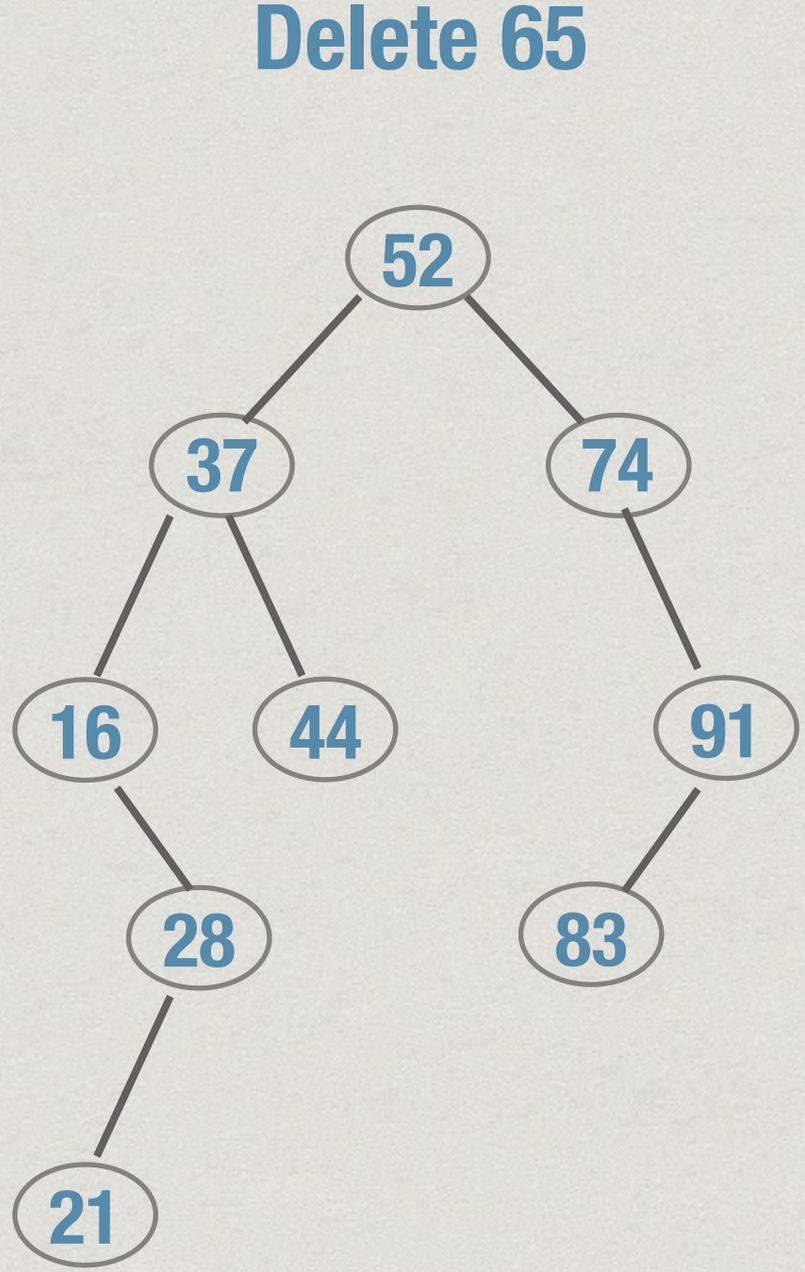
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



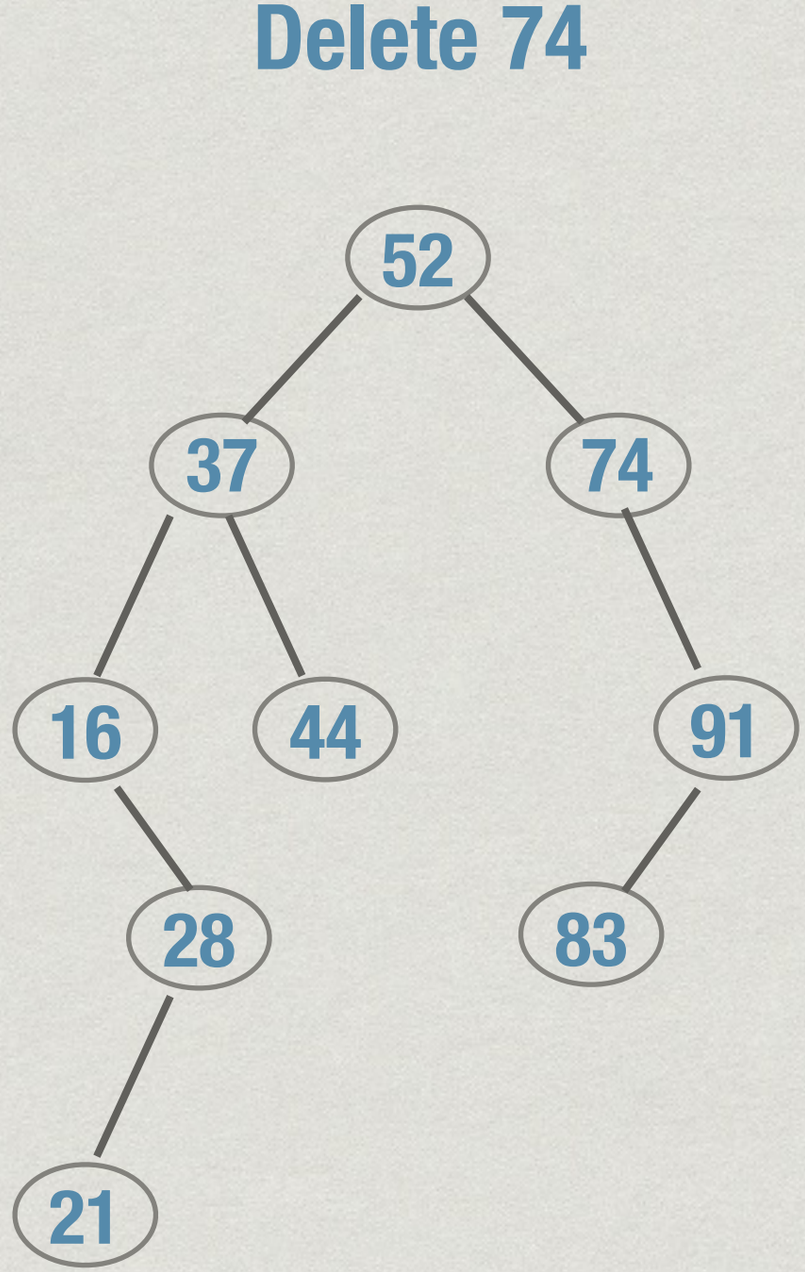
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



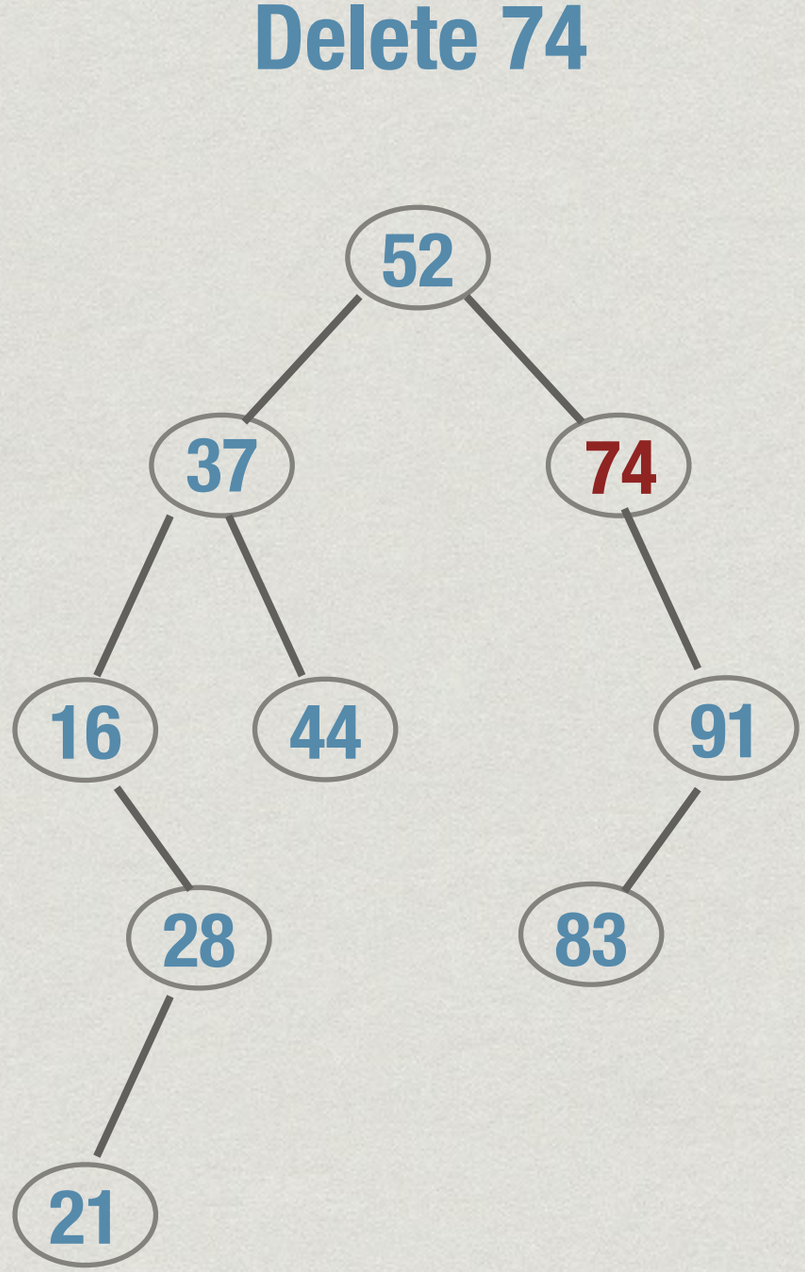
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



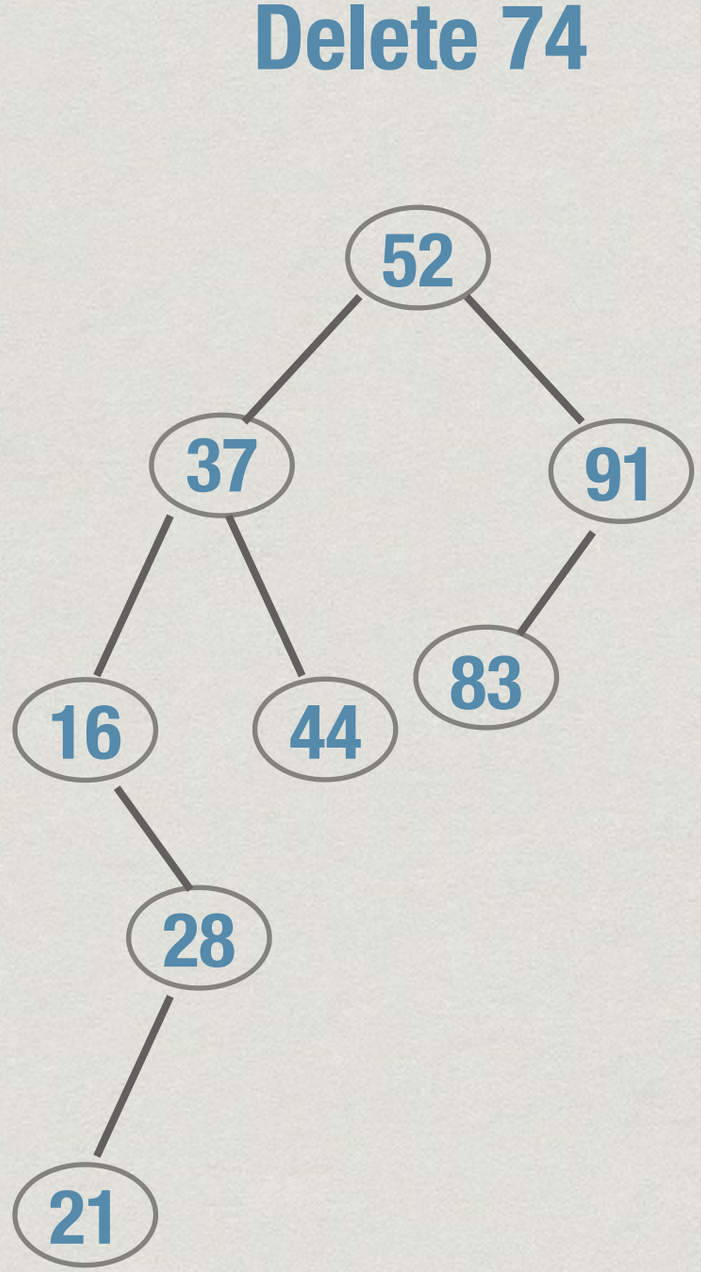
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



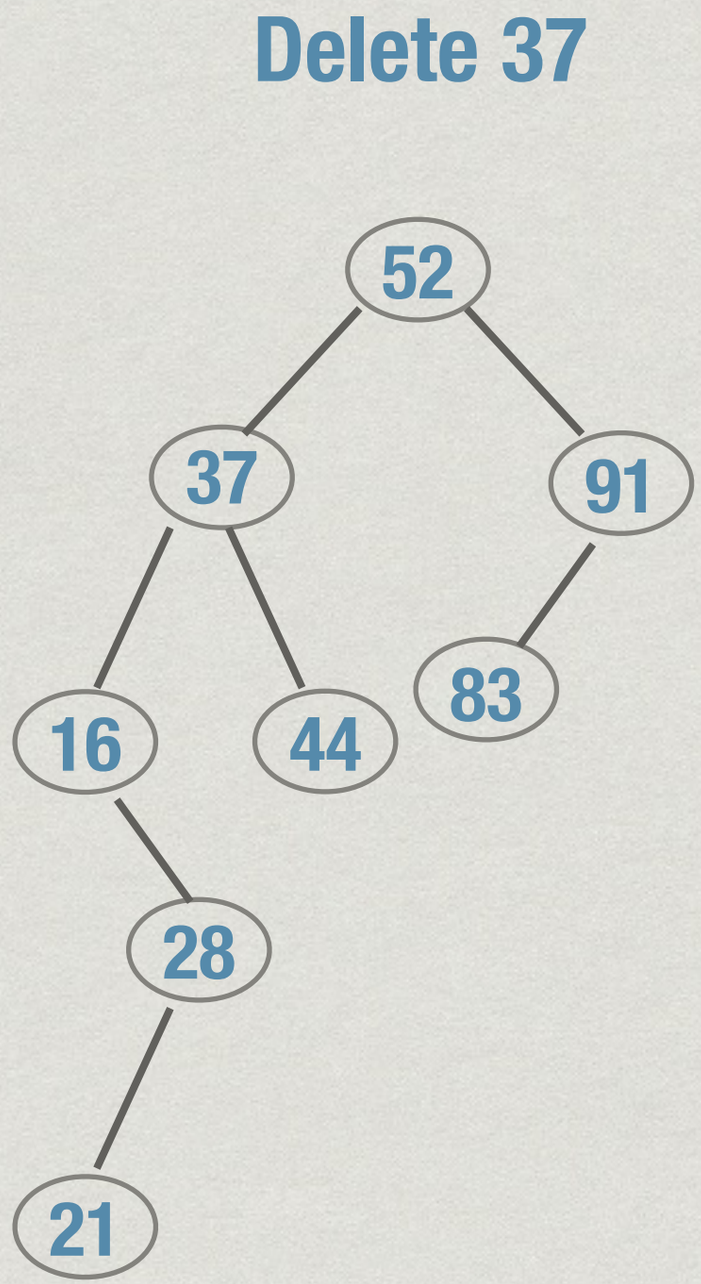
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



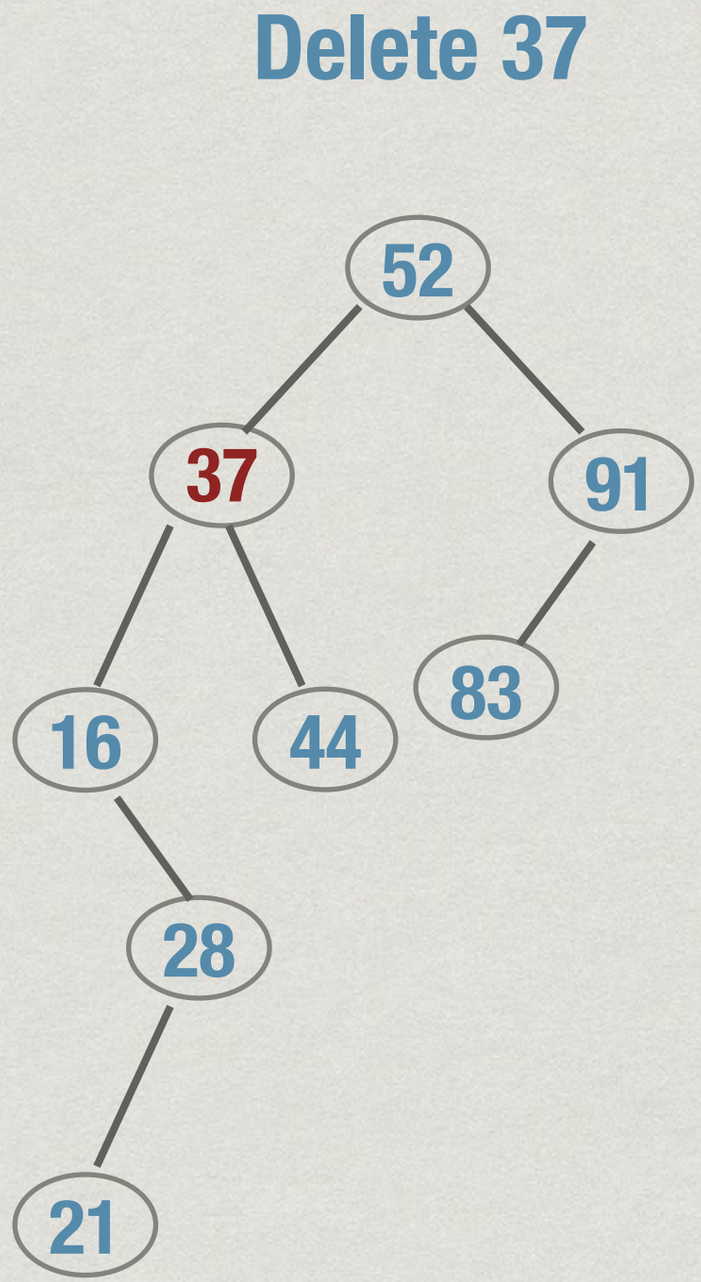
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



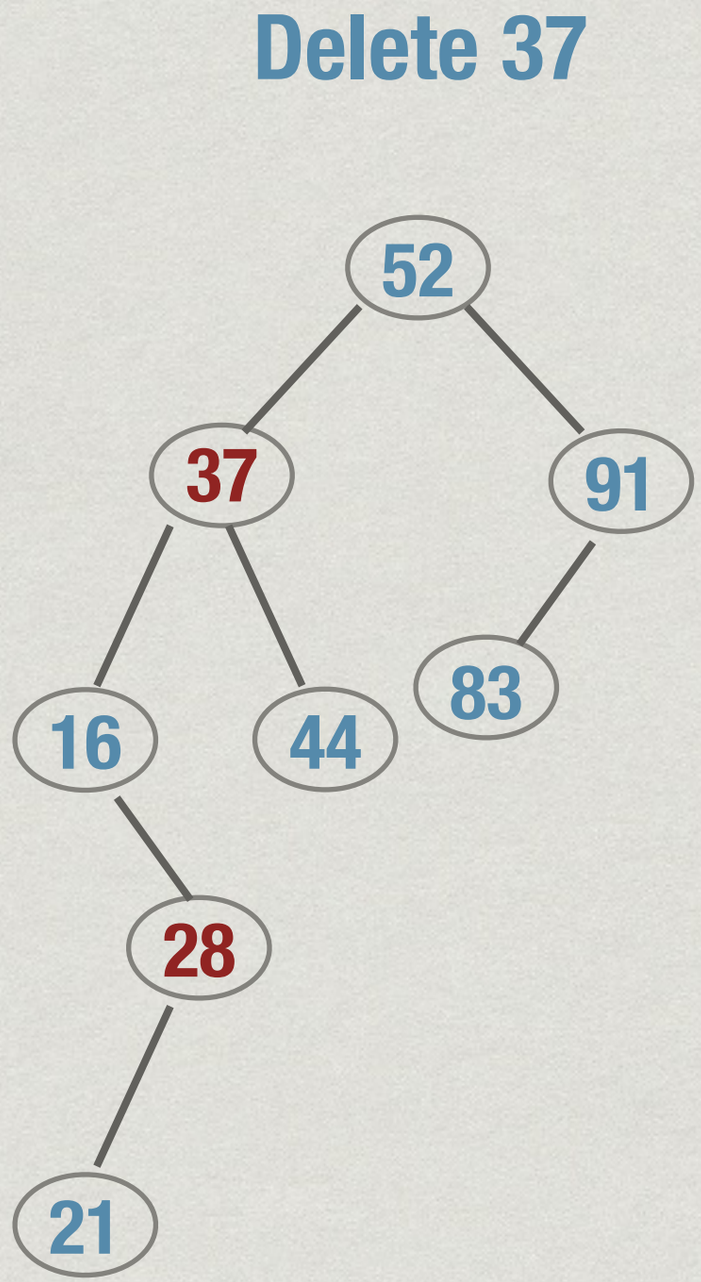
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



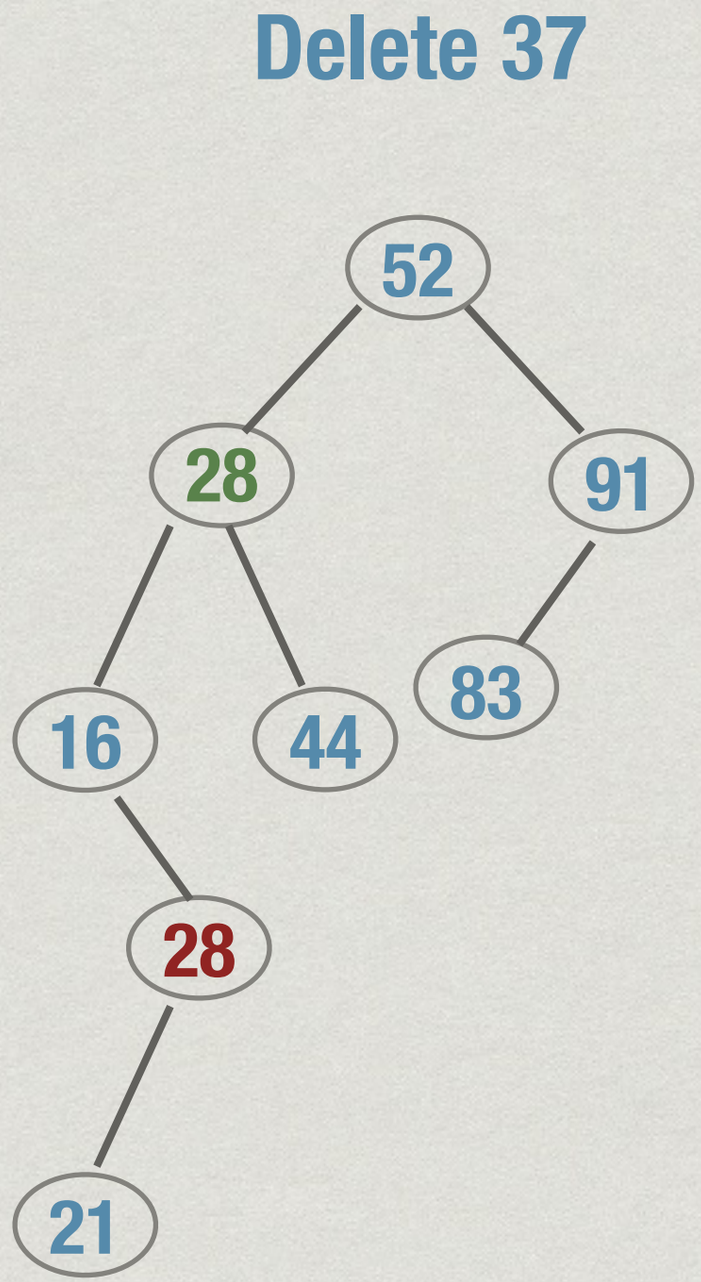
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



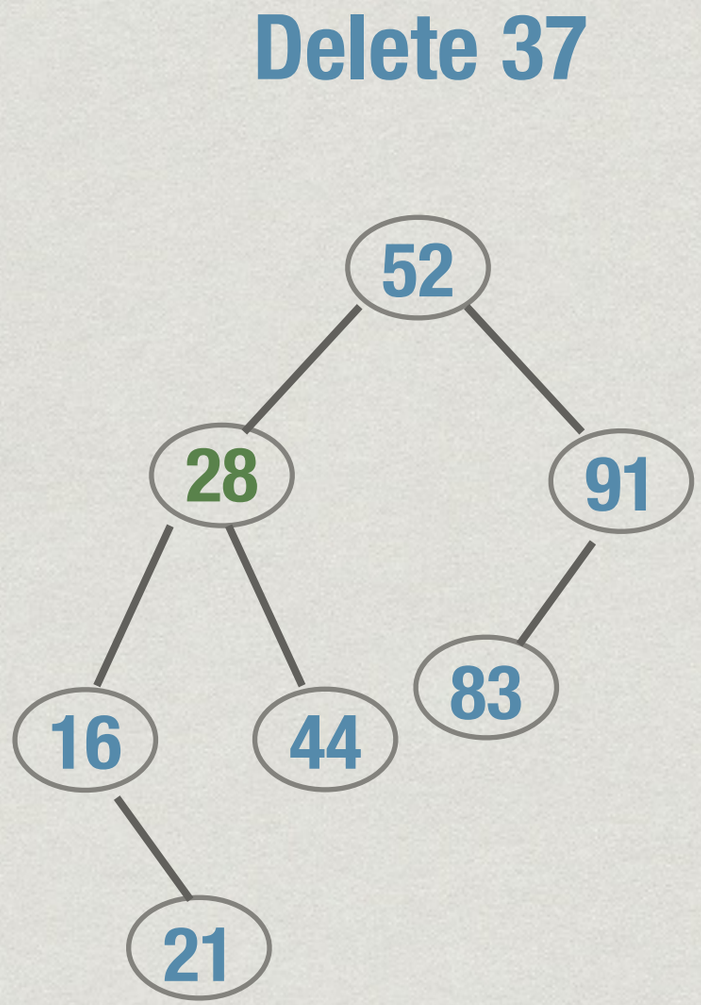
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



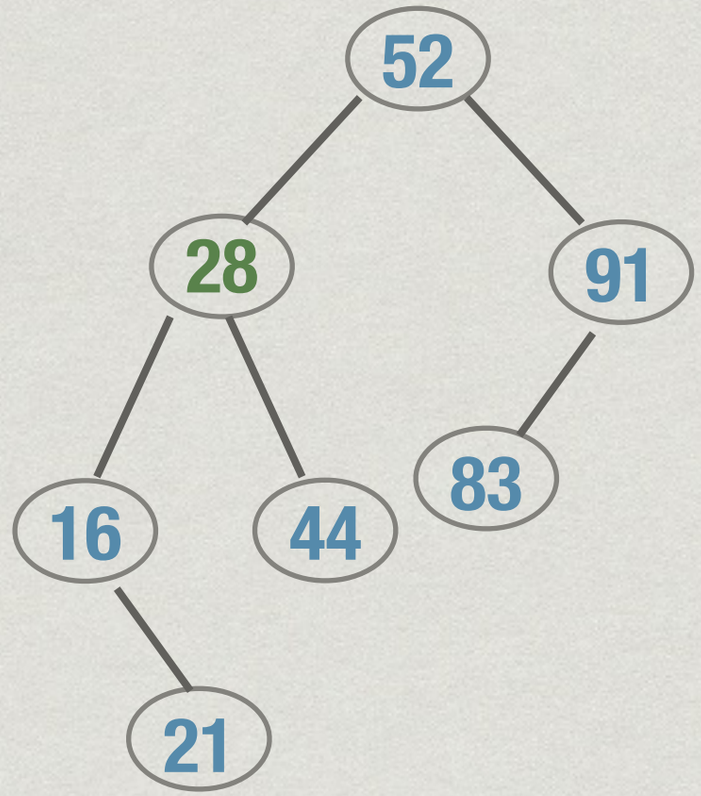
Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



Delete

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
  - Delete pred(v) / succ(v)
  - Either leaf or only one child



delete(v)

```
function delete(t,v)

if (t == NIL) return

# Recursive cases, t.value != v
if (v < t.value)
  if (t.left != NIL)
    delete(t.left,v)
  return

if (v > t.value)
  if (t.right != NIL)
    delete(t.right,v)
  return
```

delete(v)

```
# t.value == v, delete here

# Delete root
if (t.parent == NIL)
   t = NIL
   return

# Delete leaf
if (t.left == NIL and t.right == NIL)
   if (t = t.parent.left)
      t.parent.left = NIL
   else
      t.parent.right = NIL
   return
```

delete(v)

```
# Delete node with one child

# Only left child

if (t.left != NIL and t.right == NIL)
   t.left.parent = t.parent
   if (t == t.parent.left)
     t.parent.left = t.left
   else
     t.parent.right = t.left
   return
```

delete(v)

```
# Delete node with one child

# Only right child

if (t.left == NIL and t.right != NIL)
   t.right.parent = t.parent
   if (t == t.parent.left)
     t.parent.left = t.right
   else
     t.parent.right = t.right
   return
```

delete(v)

```
# Delete node with two children

# Copy pred(v) into current node

pv = pred(v)
t.value = pv

# Delete pv from left subtree
# - pv either leaf or has single child

delete(t.left,pv)
```

Complexity

- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is O(log n) for n nodes
- Will see later how to maintain balance

Summary

	Heap	Sorted array	Search tree
Find	O(n)	O(log n)	O(log n)
Min	O(1)	O(1)	O(log n)
Max	O(n)	O(1)	O(log n)
Insert	O(log n)	O(n)	O(log n)
Delete	O(log n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(log n)
Succ	O(n)	O(1)	O(log n)