NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 7, Lecture 4

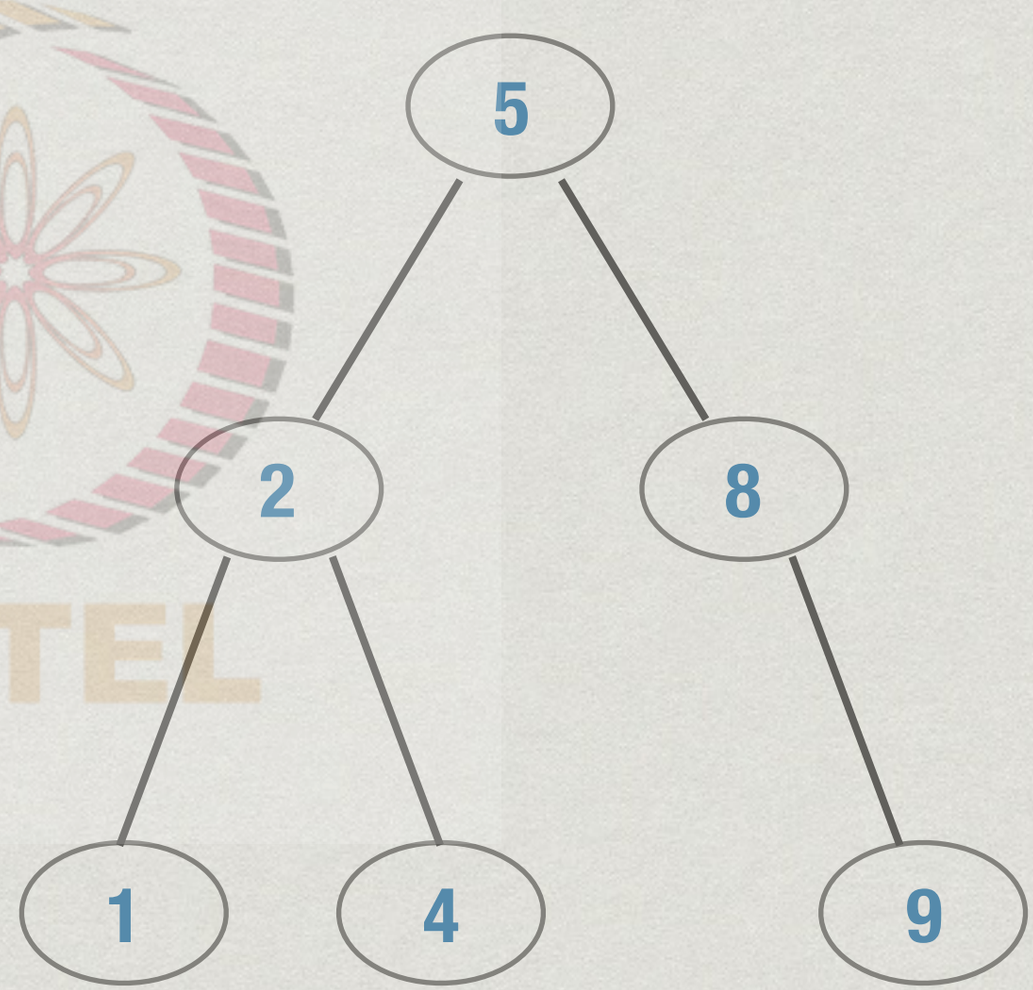
Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
  - Insert/delete in sorted list take time O(n)
- Like priority queues, move to a tree structure

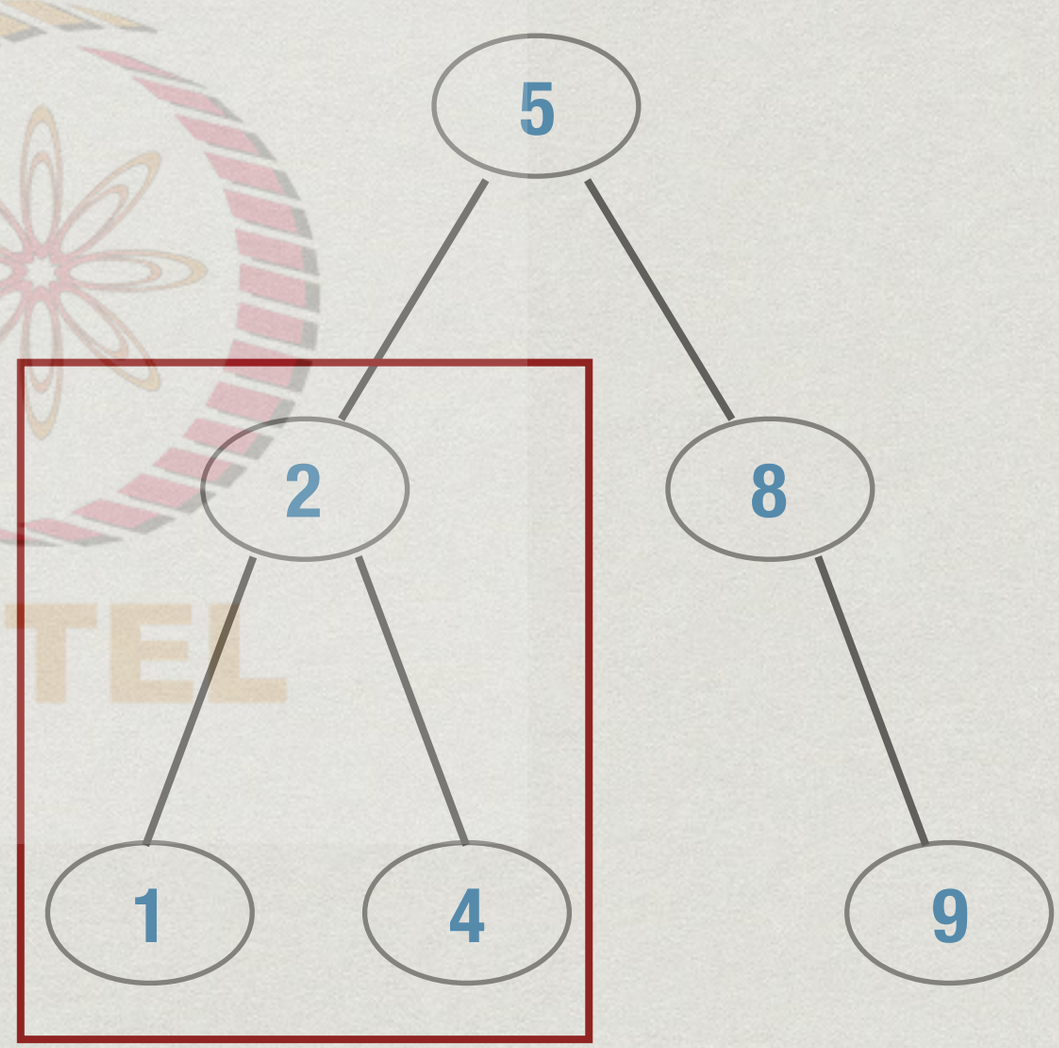
Binary search tree

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



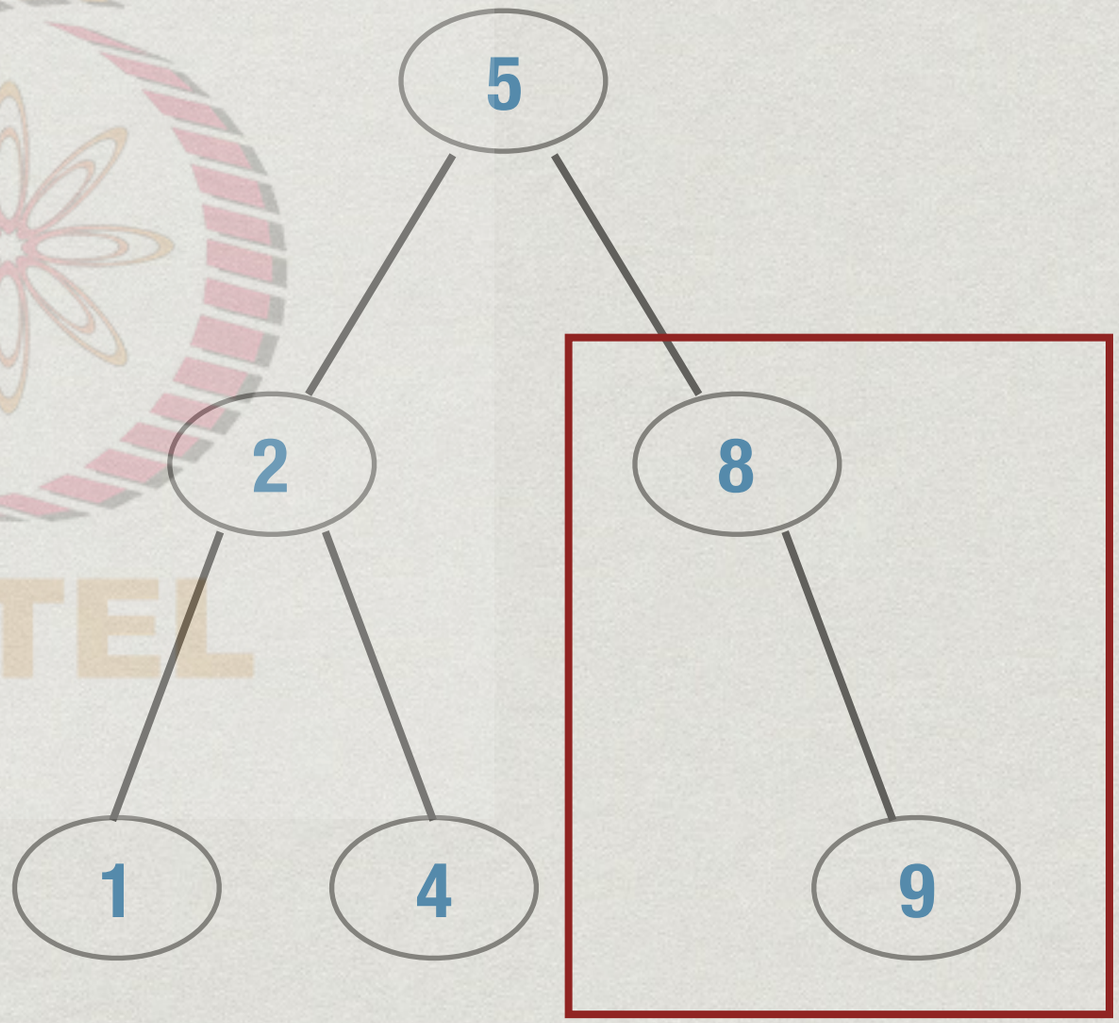
Binary search tree

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



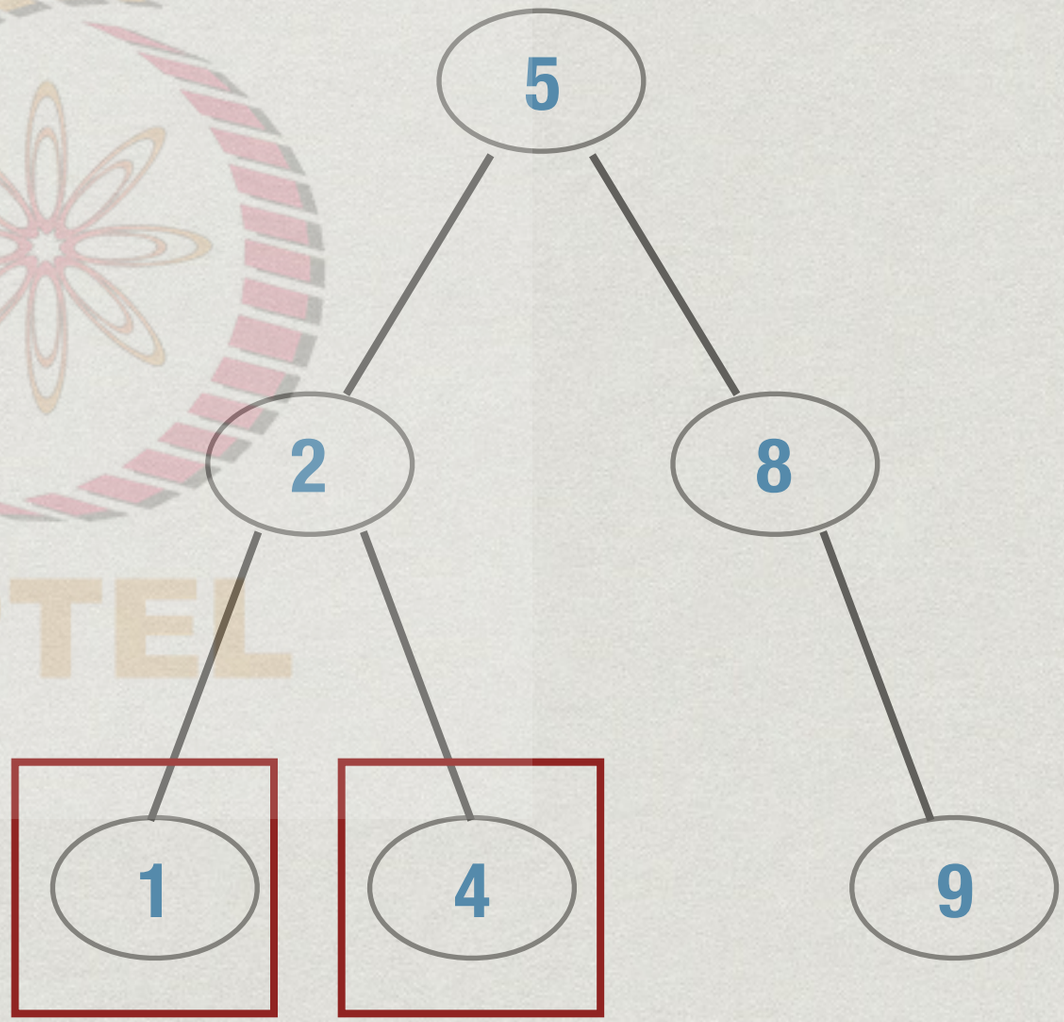
Binary search tree

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



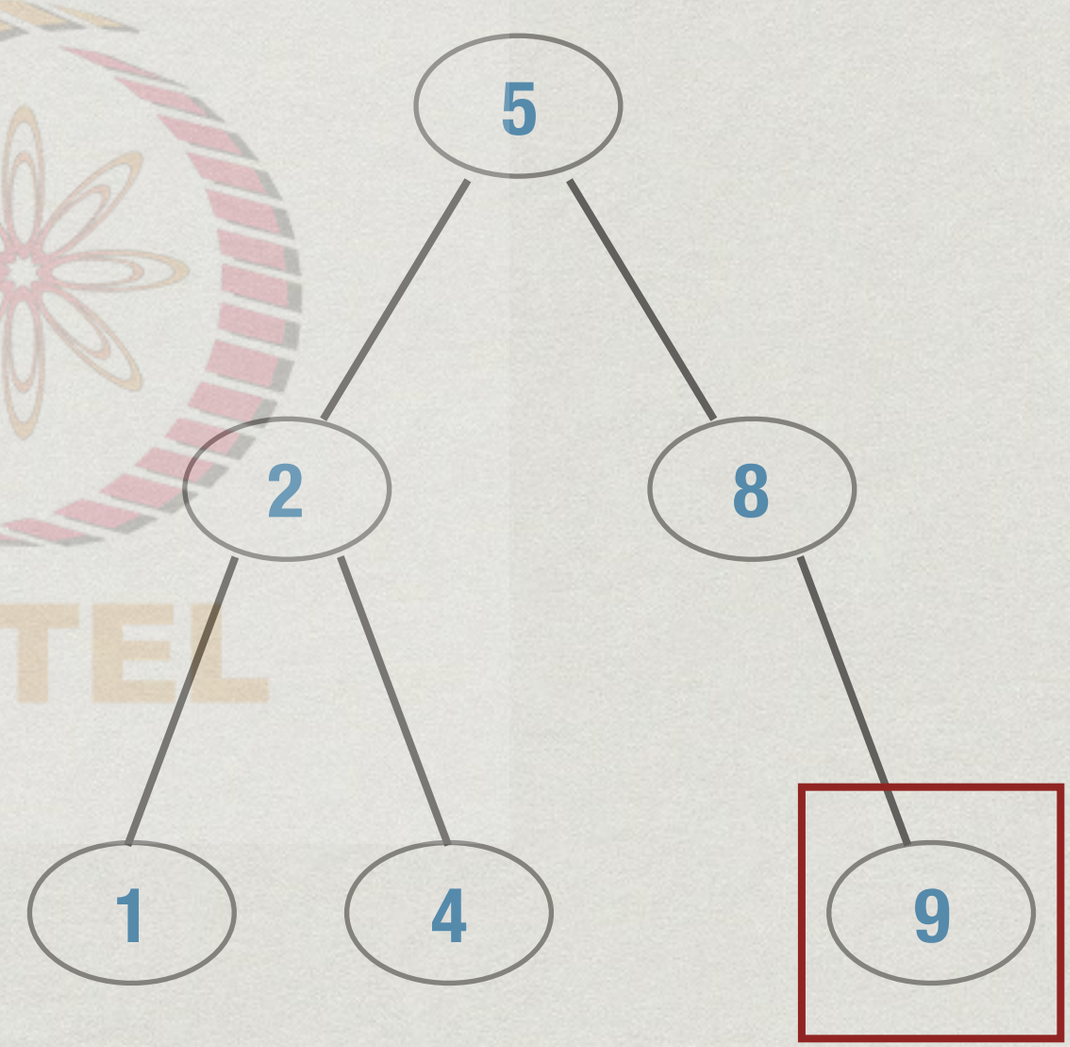
Binary search tree

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



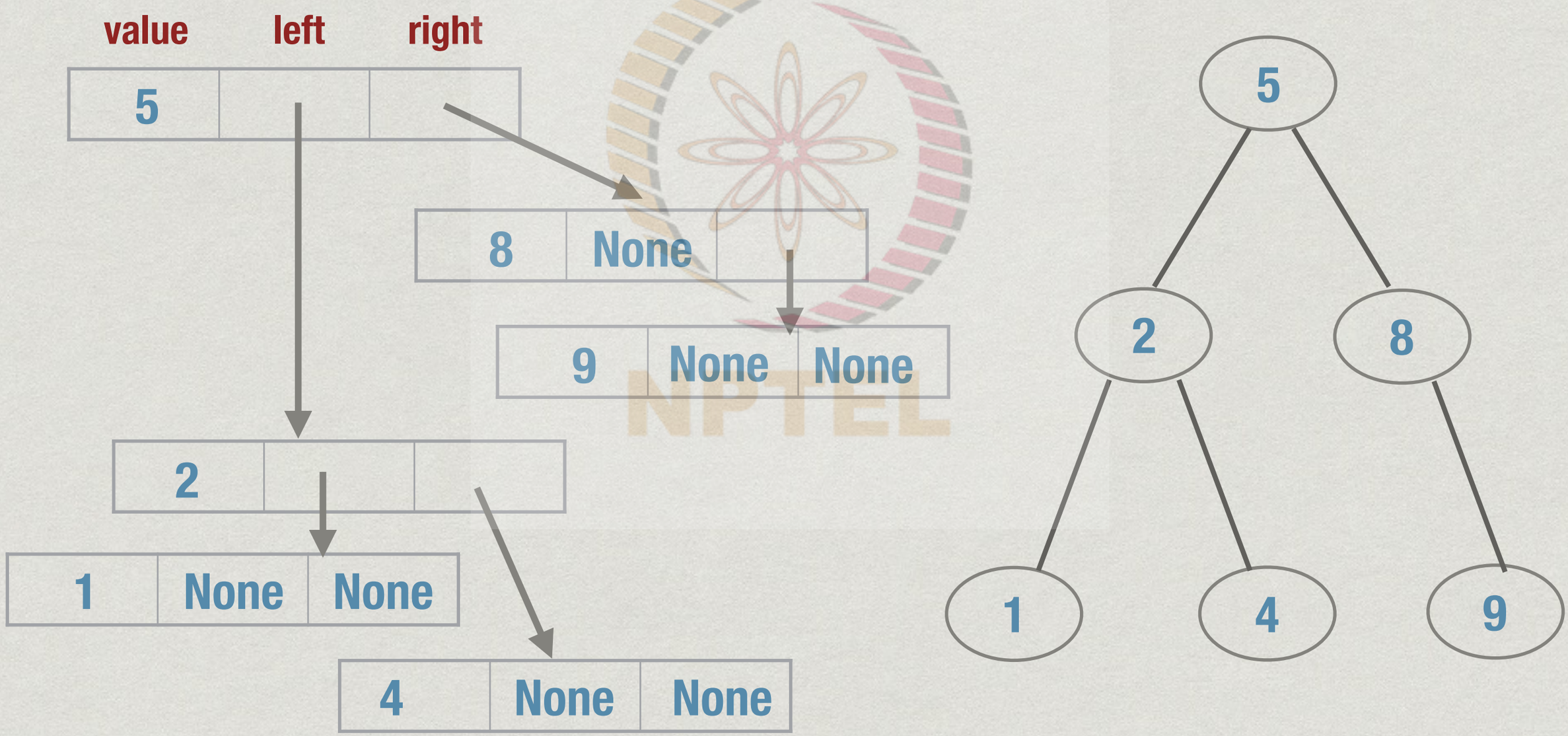
Binary search tree

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



Binary search tree

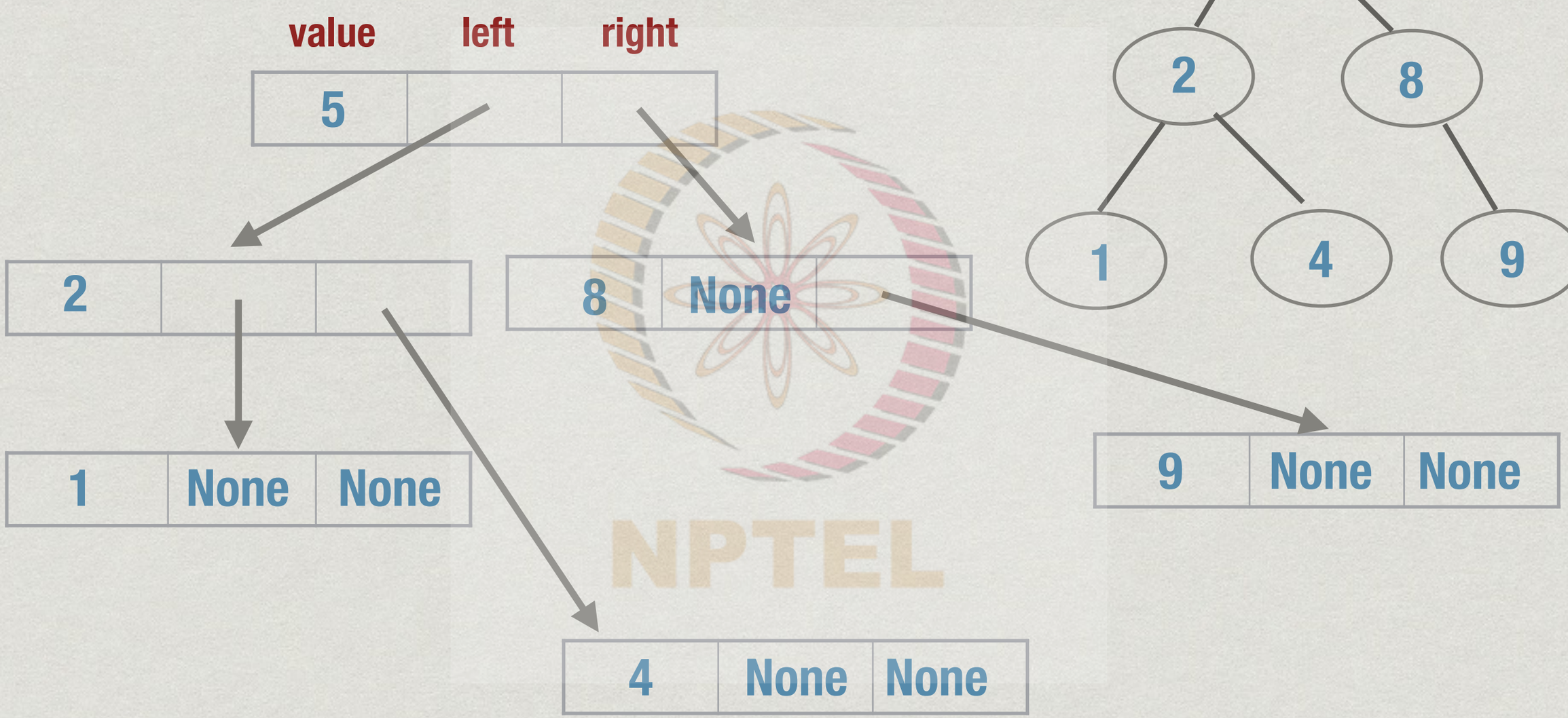
- Each node has a value and points to its children

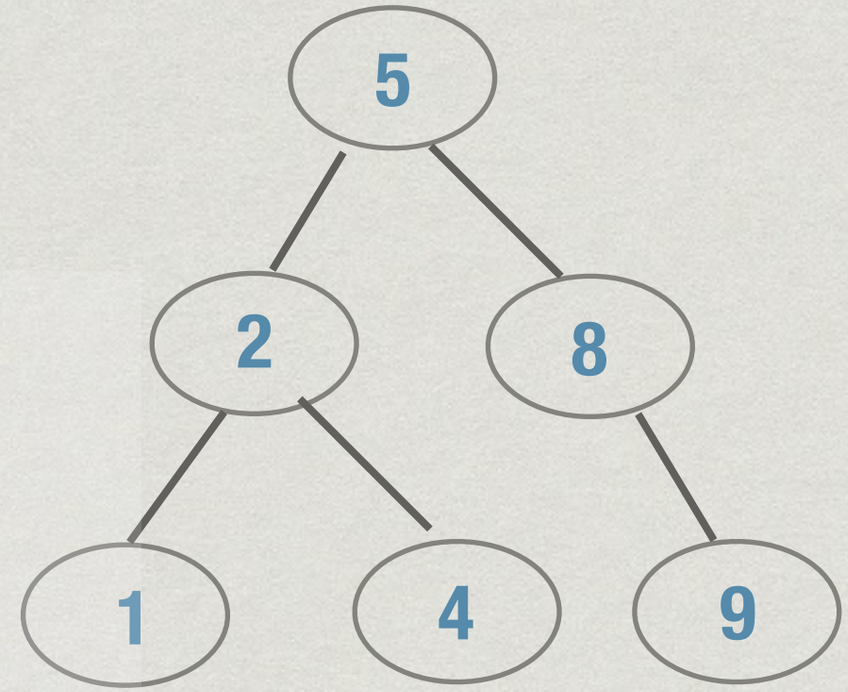


A better representation

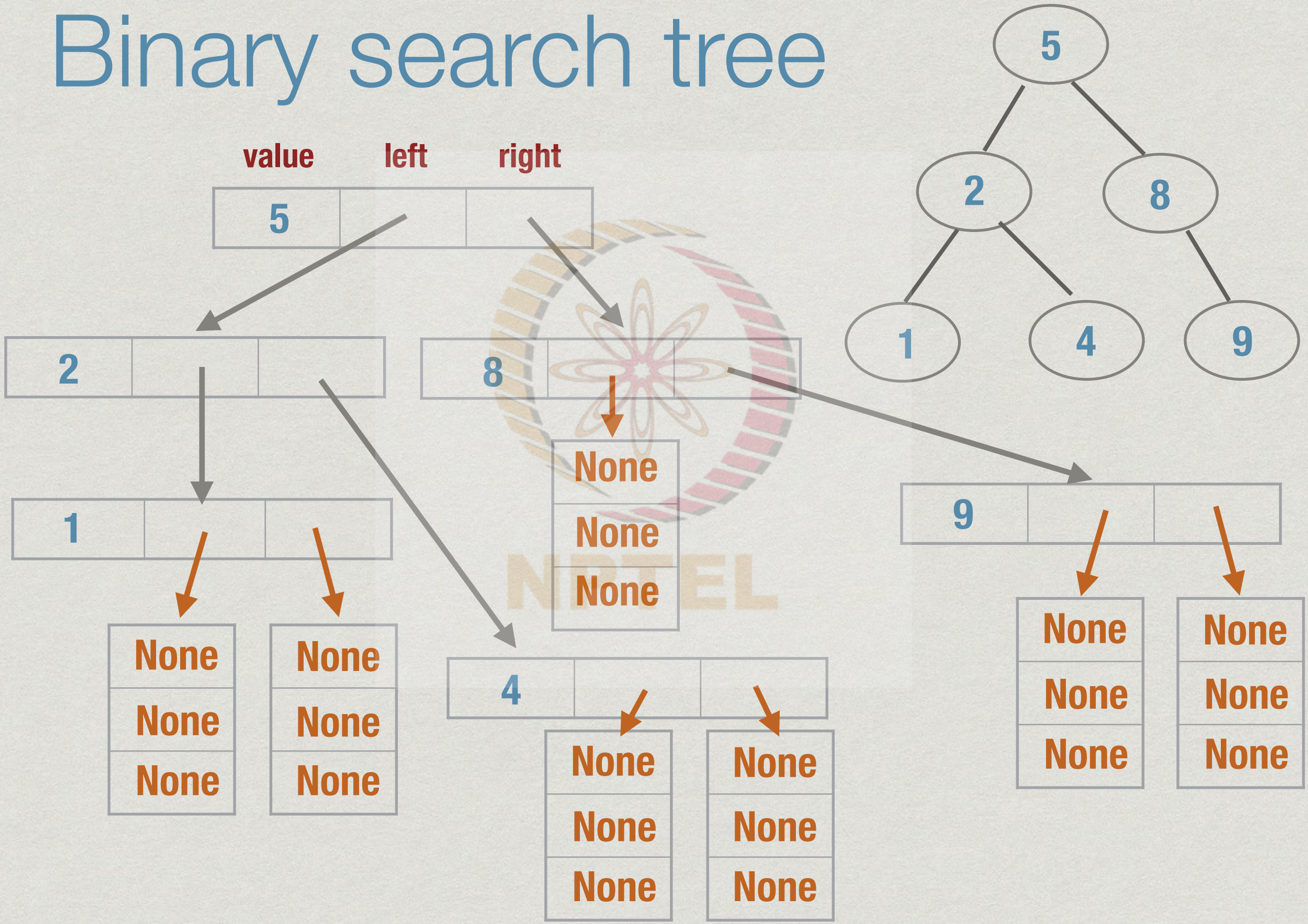
- Add a frontier with empty node: all fields None
- Empty tree is a single empty node
- Leaf node has value that is not None, left and right children point to empty nodes
- Makes it easier to write recursive functions to traverse the tree

Binary search tree





Binary search tree

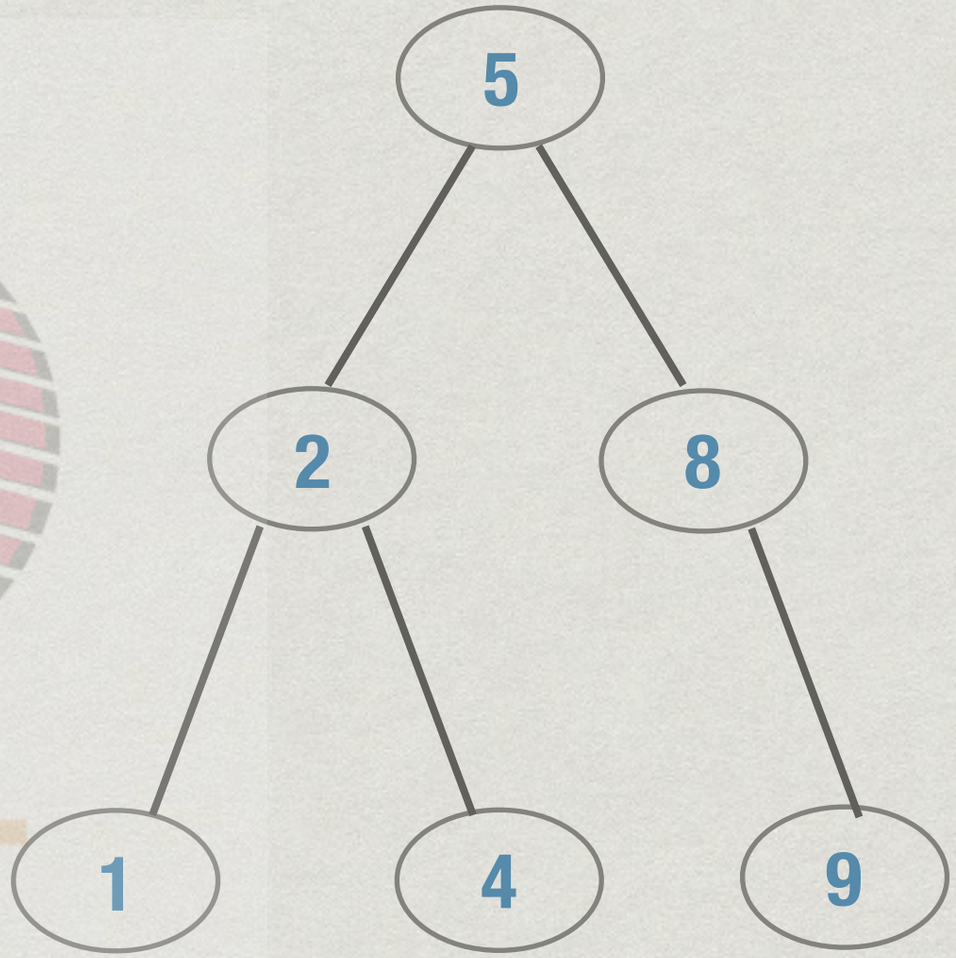


The class Tree

```
class Tree:
  def __init__(self,initval=None):
    self.value = initval
    if self.value:
      self.left = Tree()
      self.right = Tree()
    else:
      self.left = None
      self.right = None
    return()

  def isempty(self):
    return(self.value == None)
```

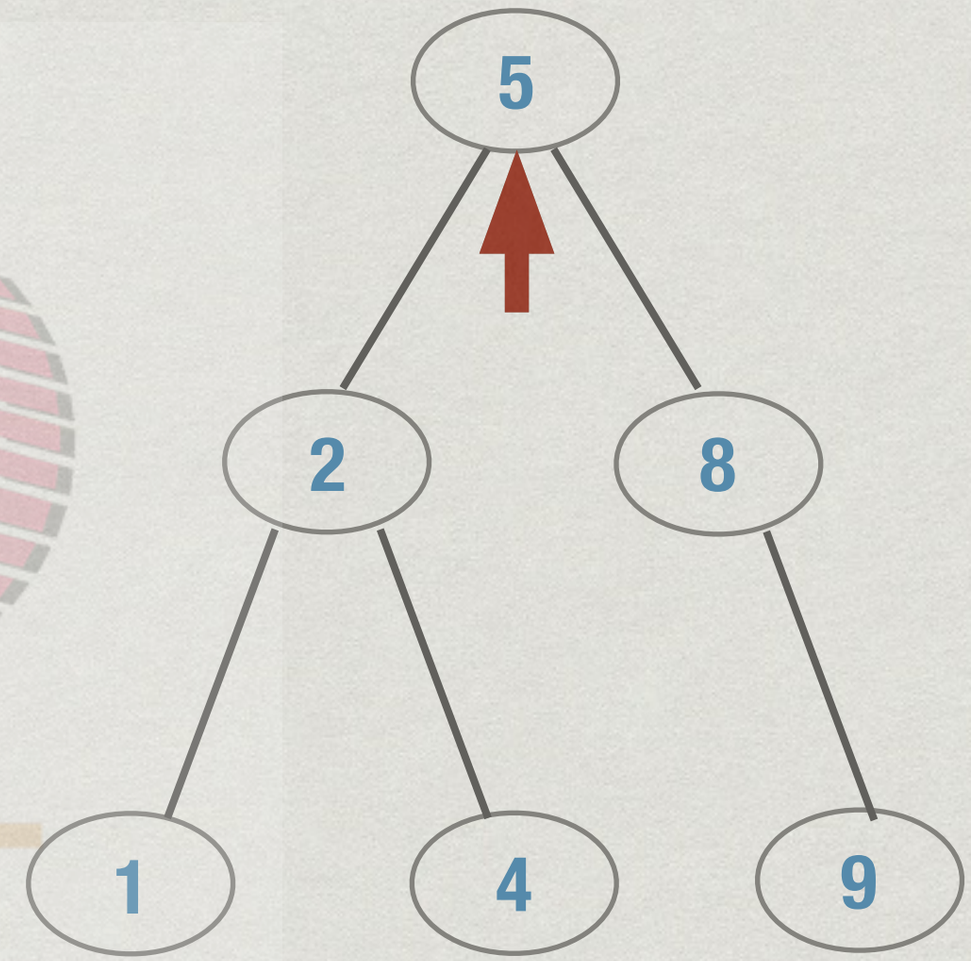
Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



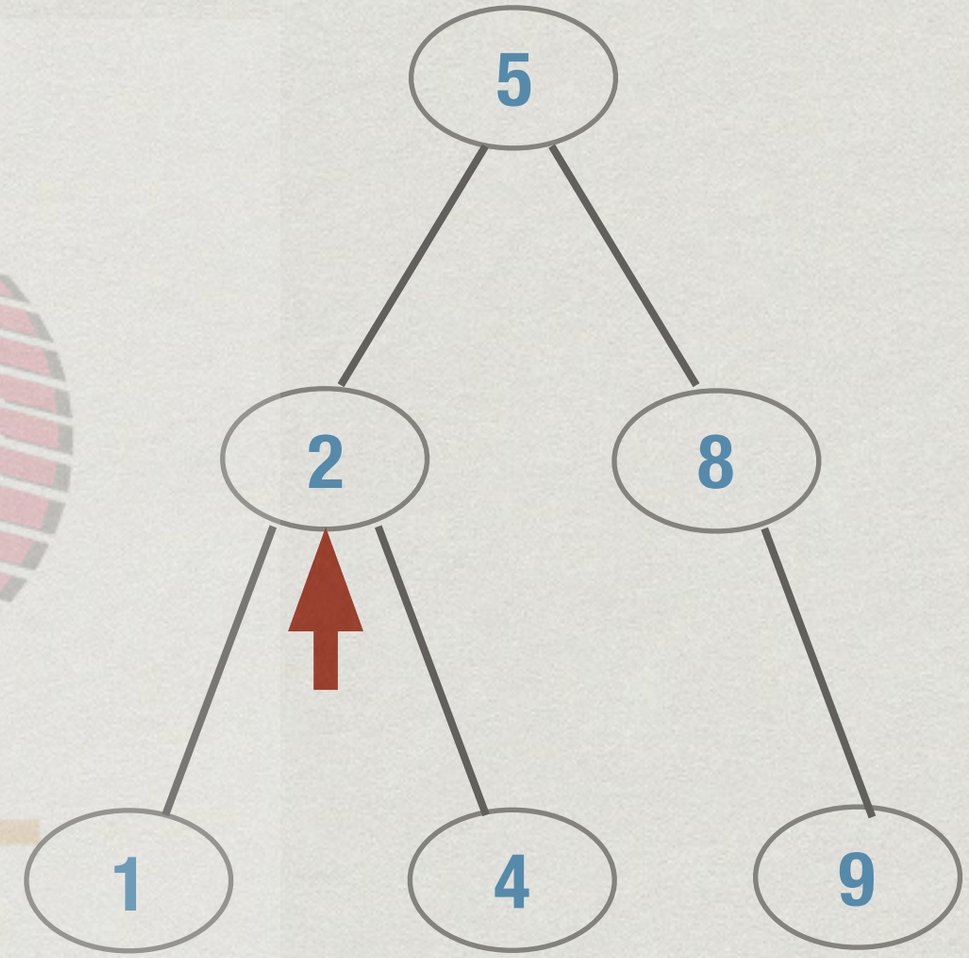
Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



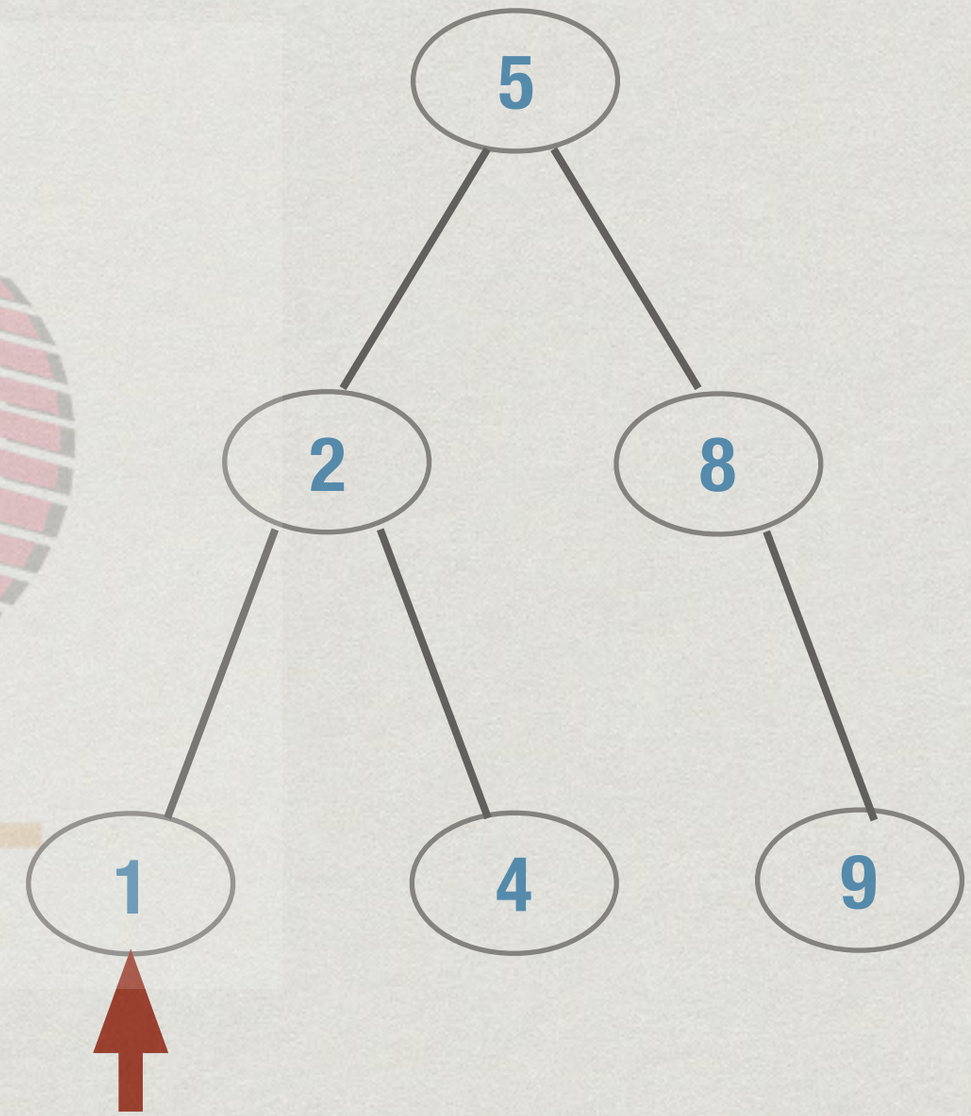
Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



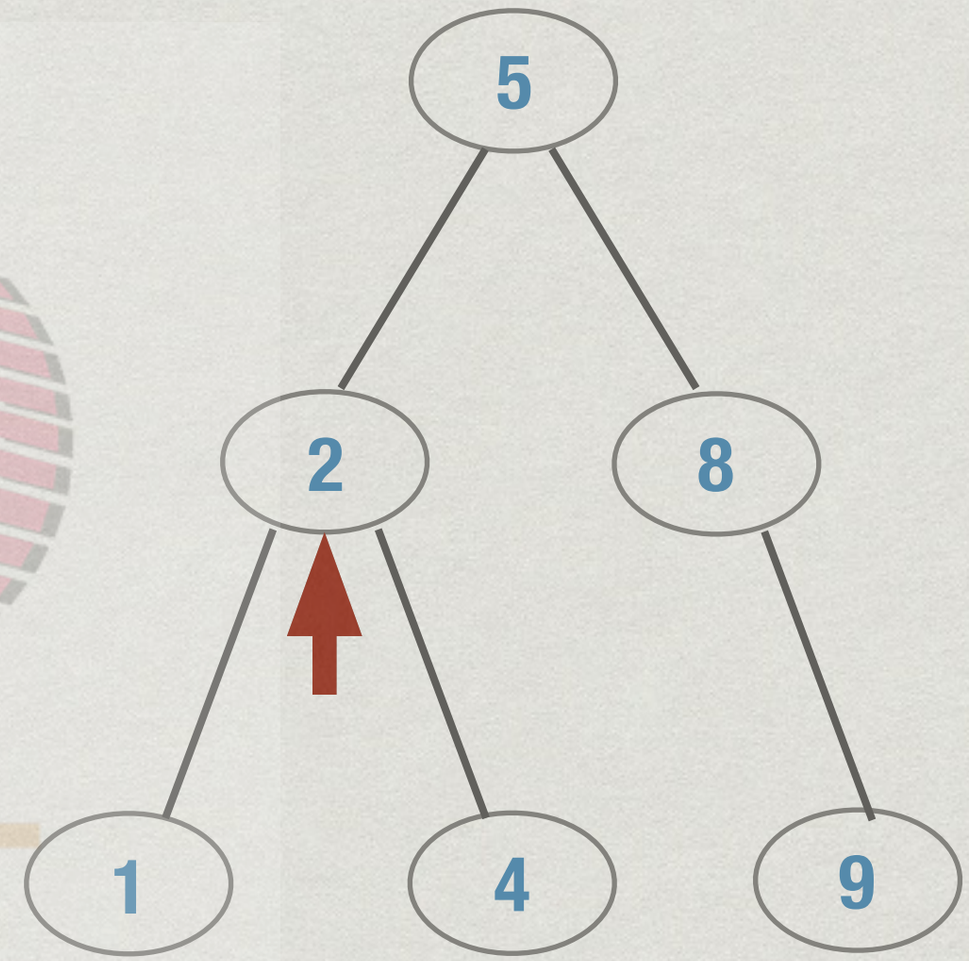
Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



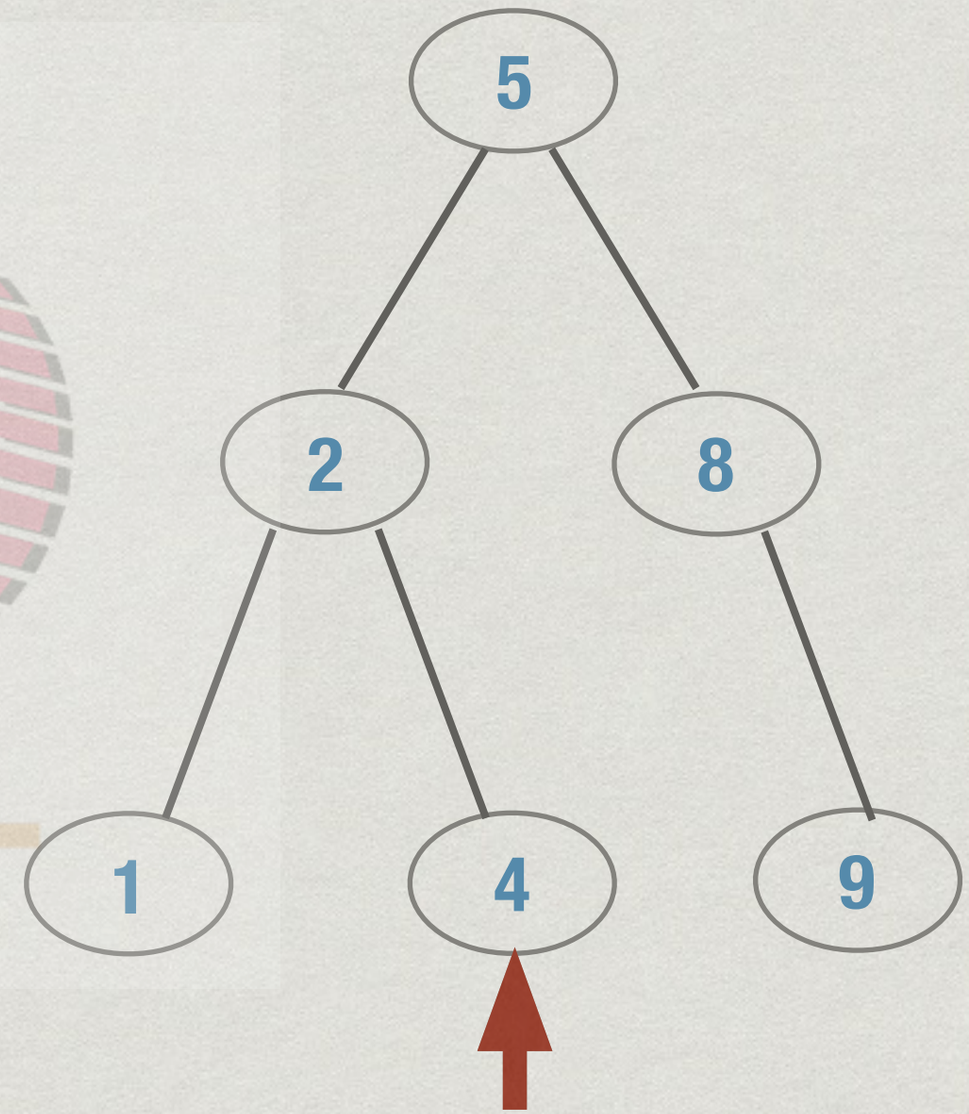
Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



Inorder traversal

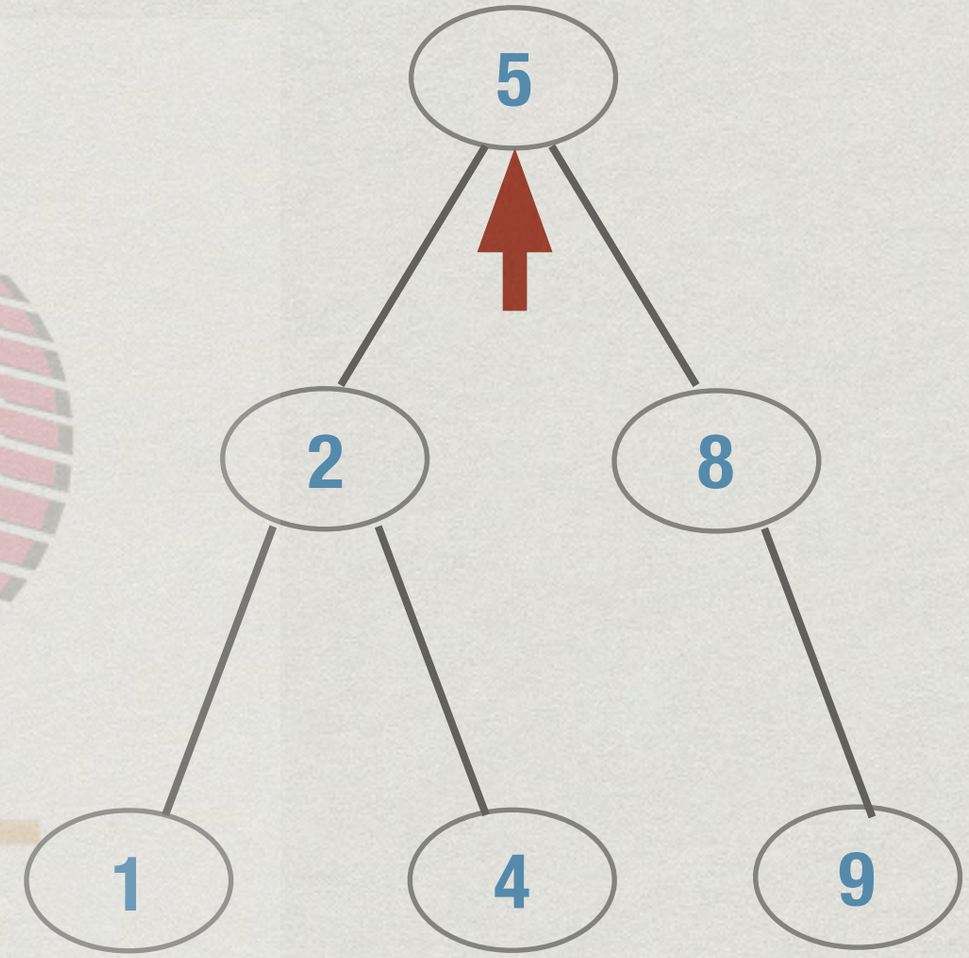
```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



1 2

Inorder traversal

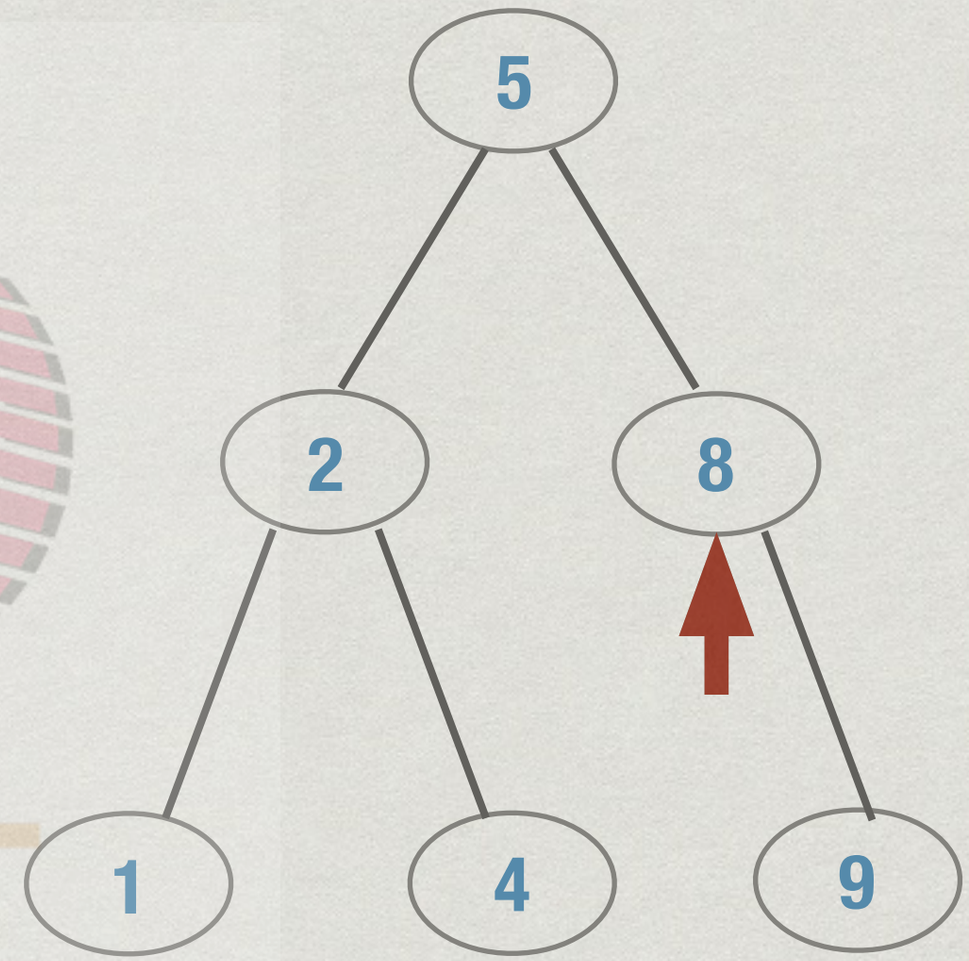
```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



1 2 4

Inorder traversal

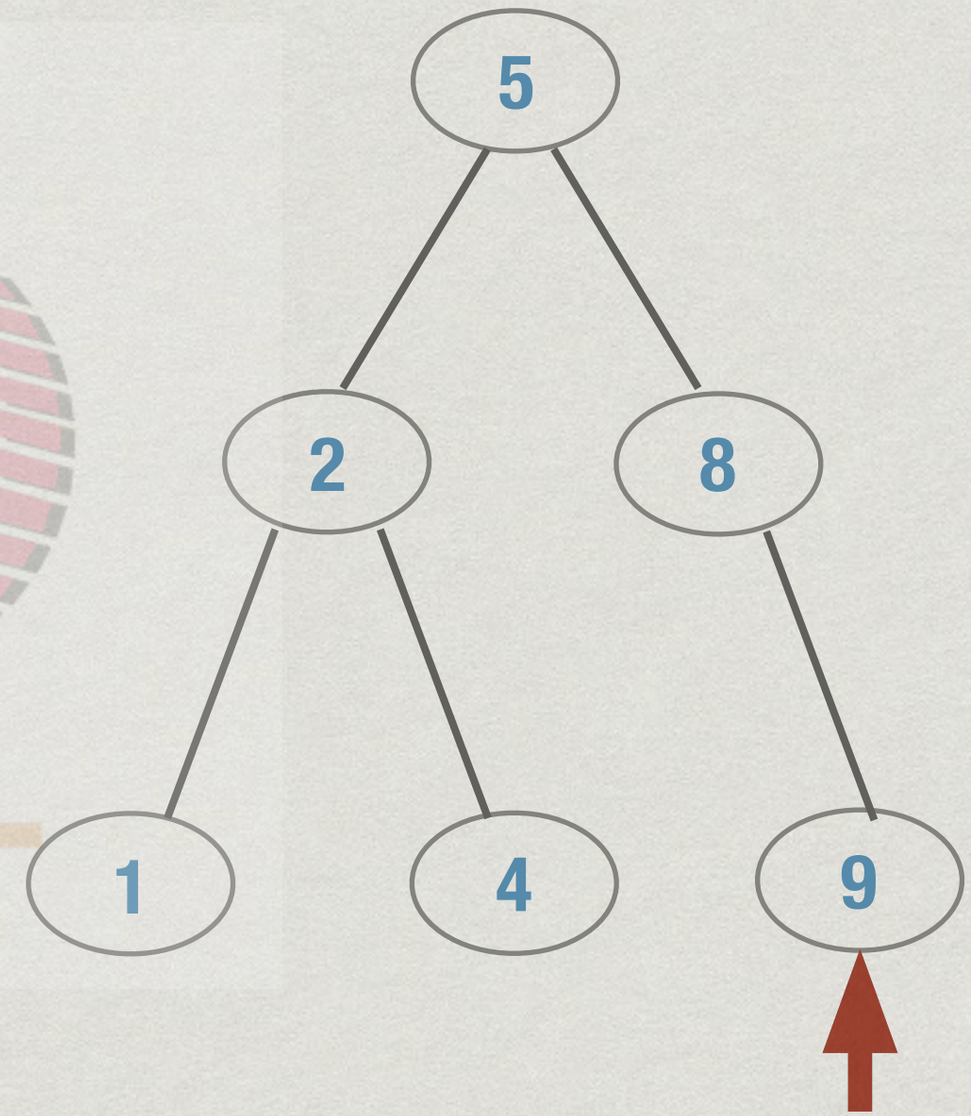
```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



1 2 4 5

Inorder traversal

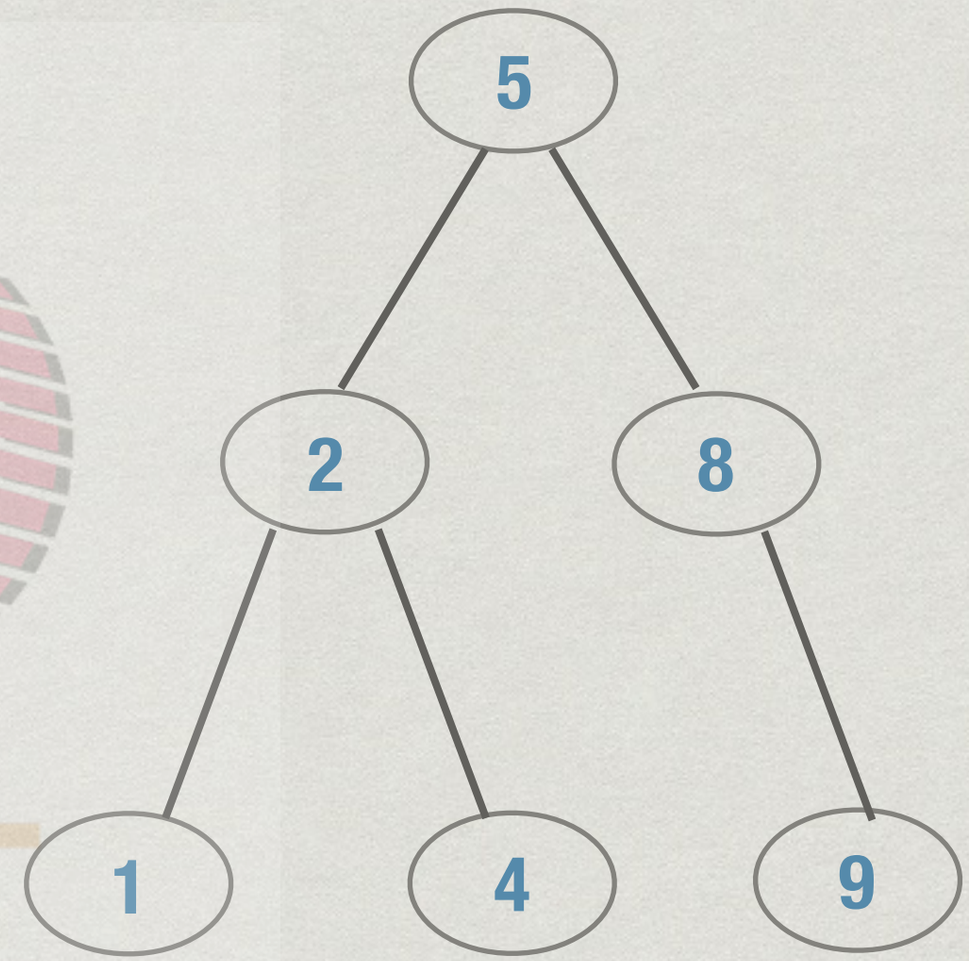
```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



1 2 4 5 8

Inorder traversal

```
def inorder(self):
 if self.isempty():
   return([])
 else:
   return(
     self.left.inorder() +
     [self.value] +
     self.right.inorder()
     )

def __str__(self):
  return(str(self.inorder()))
```



- Lists values in sorted order

1 2 4 5 8 9

Find a value v

- Scan the current node
- Go left if v is smaller than this node
- Go right if v is larger than this node
- Natural generalization of binary search

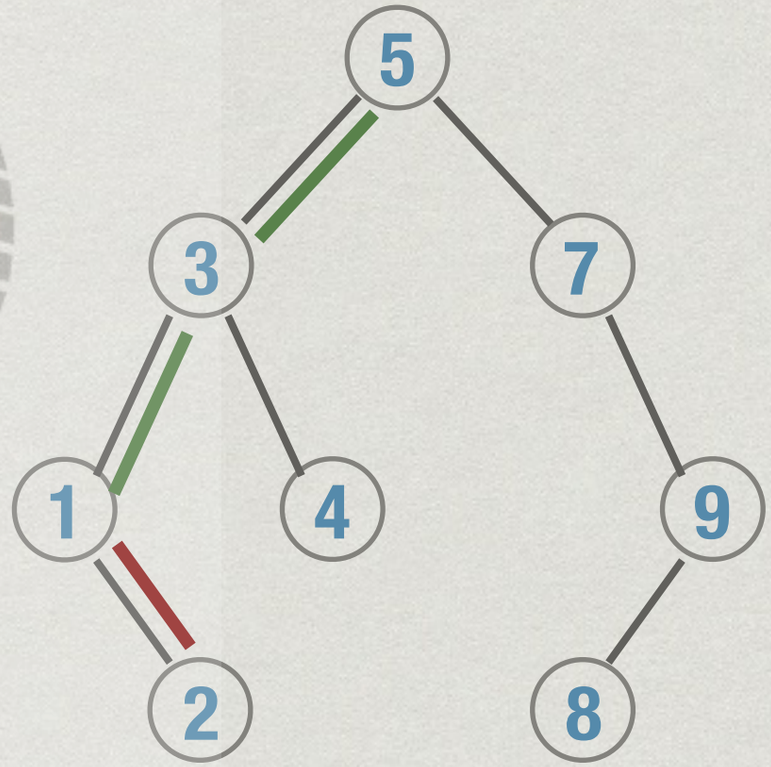
Find a value v

```
def find(self,v):

  if self.isempty():
    return(False)

  if self.value == v:
    return(True)

  if v < self.value:
    return(self.left.find(v))
  else
    return(self.right.find(v))
```

Minimum

- Left most node in the tree

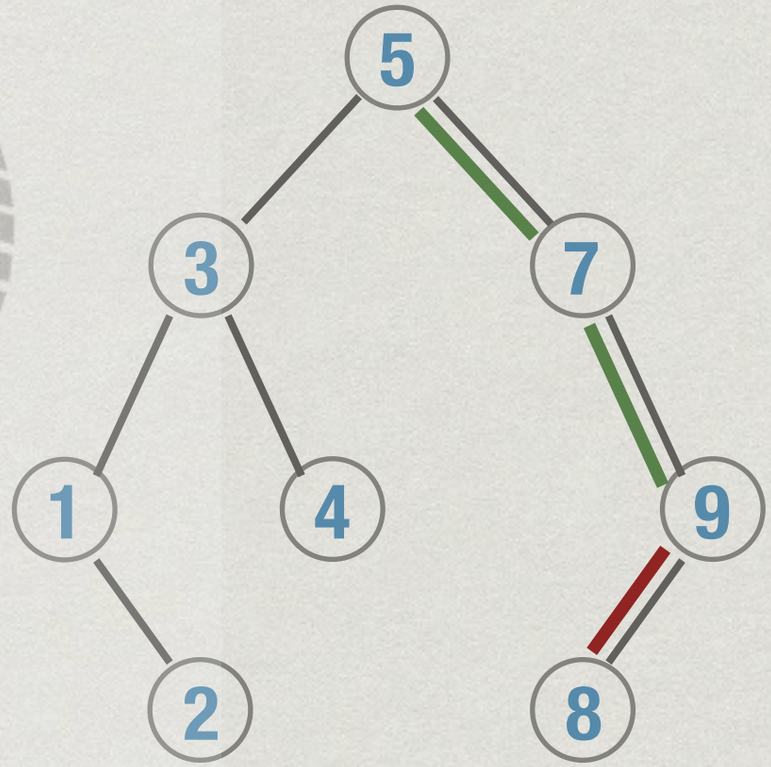
```
def minval(self):
# Assume t is not empty
    if self.left == None:
      return(self.value)
    else:
      return(self.left.minval())
```



Maximum

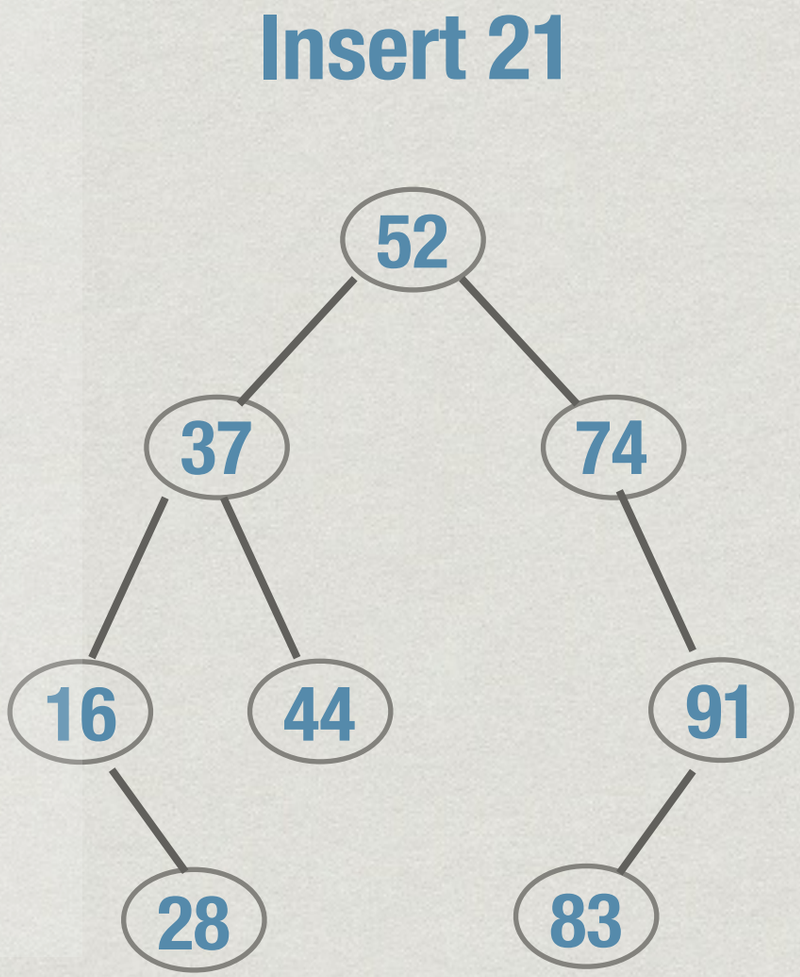
- Right most node in the tree

```
def maxval(self):
# Assume t is not empty
    if self.right == None:
      return(self.value)
    else:
      return(self.right.maxval())
```



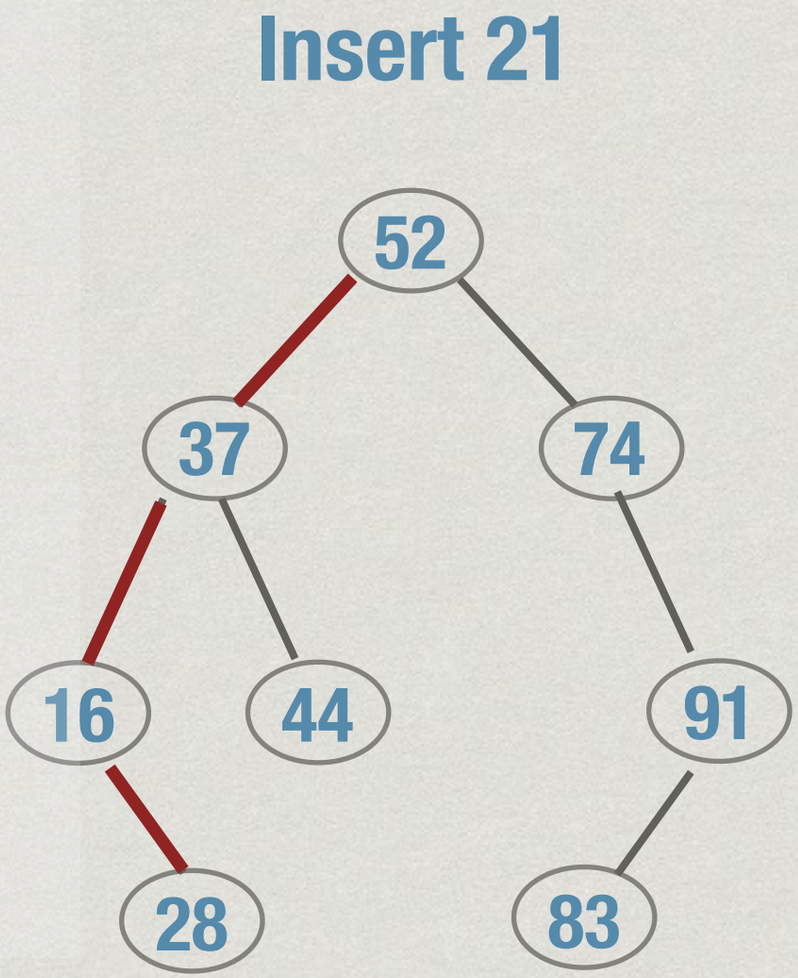
Insert v

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



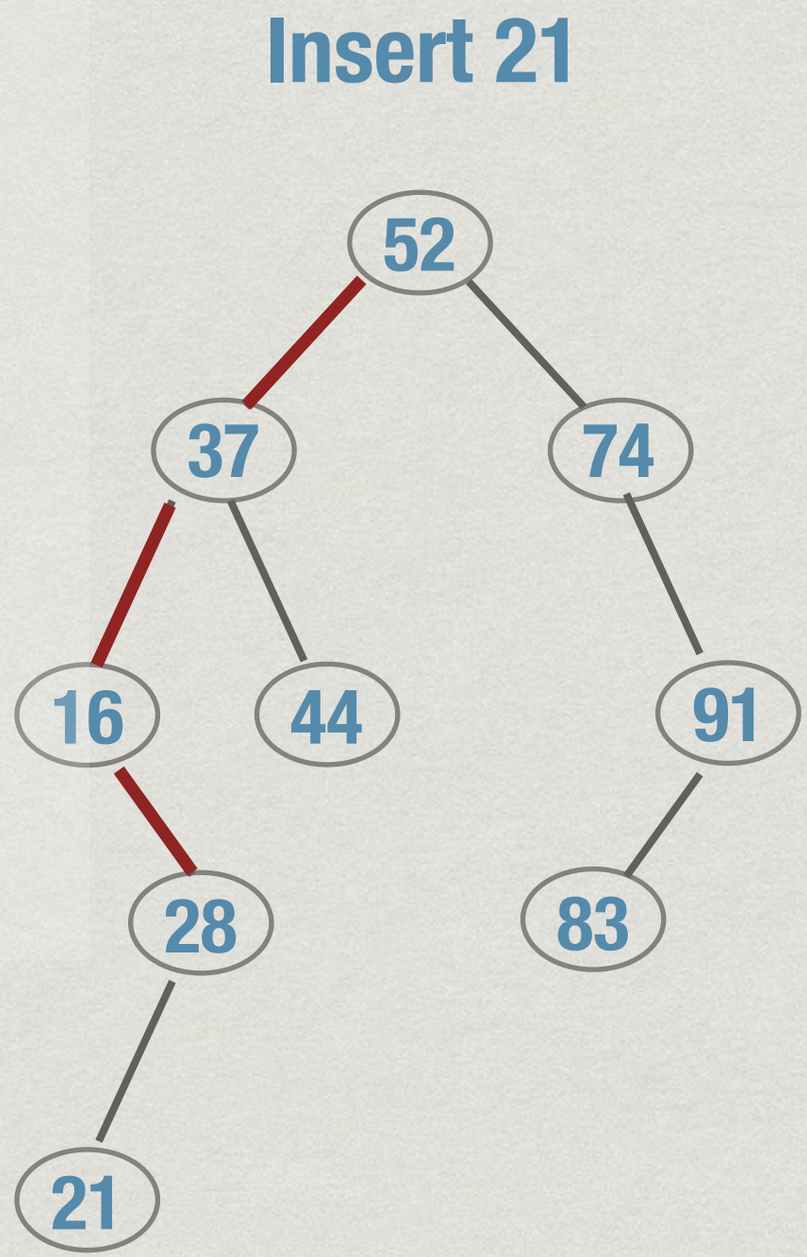
Insert v

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



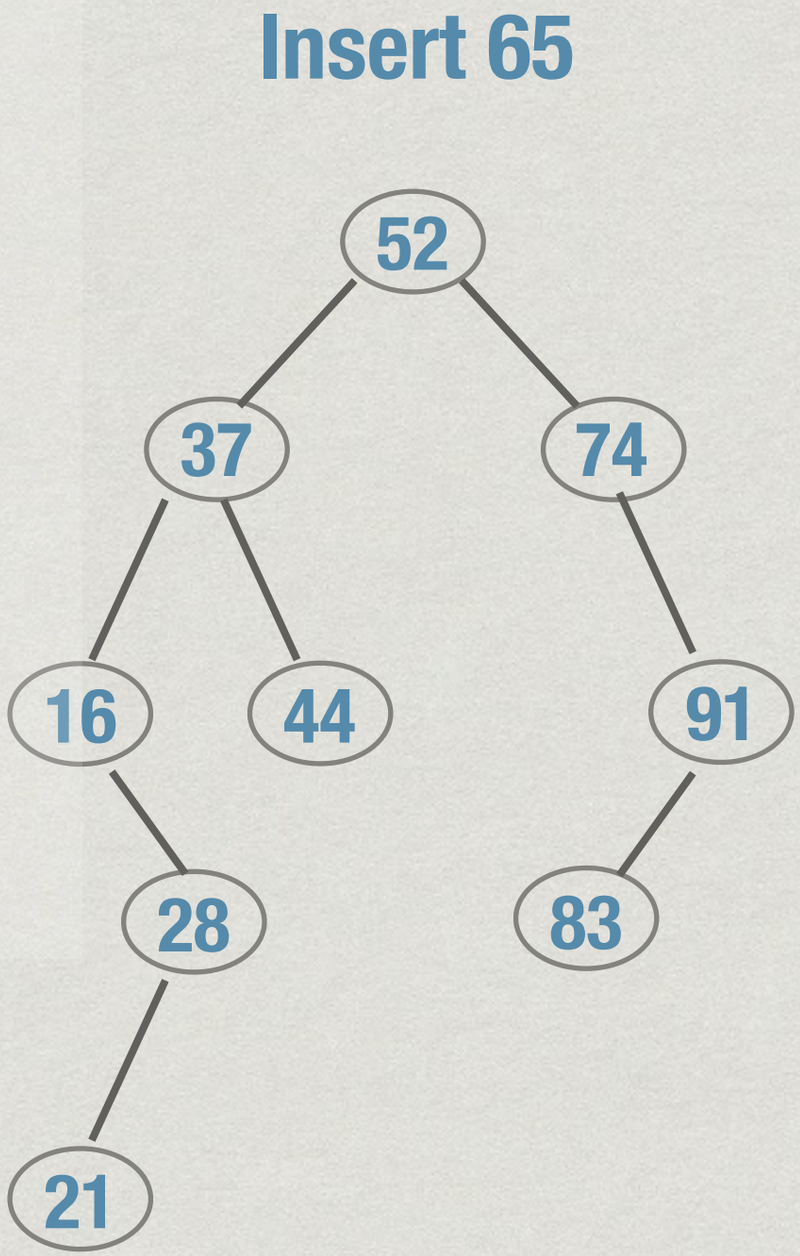
Insert v

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



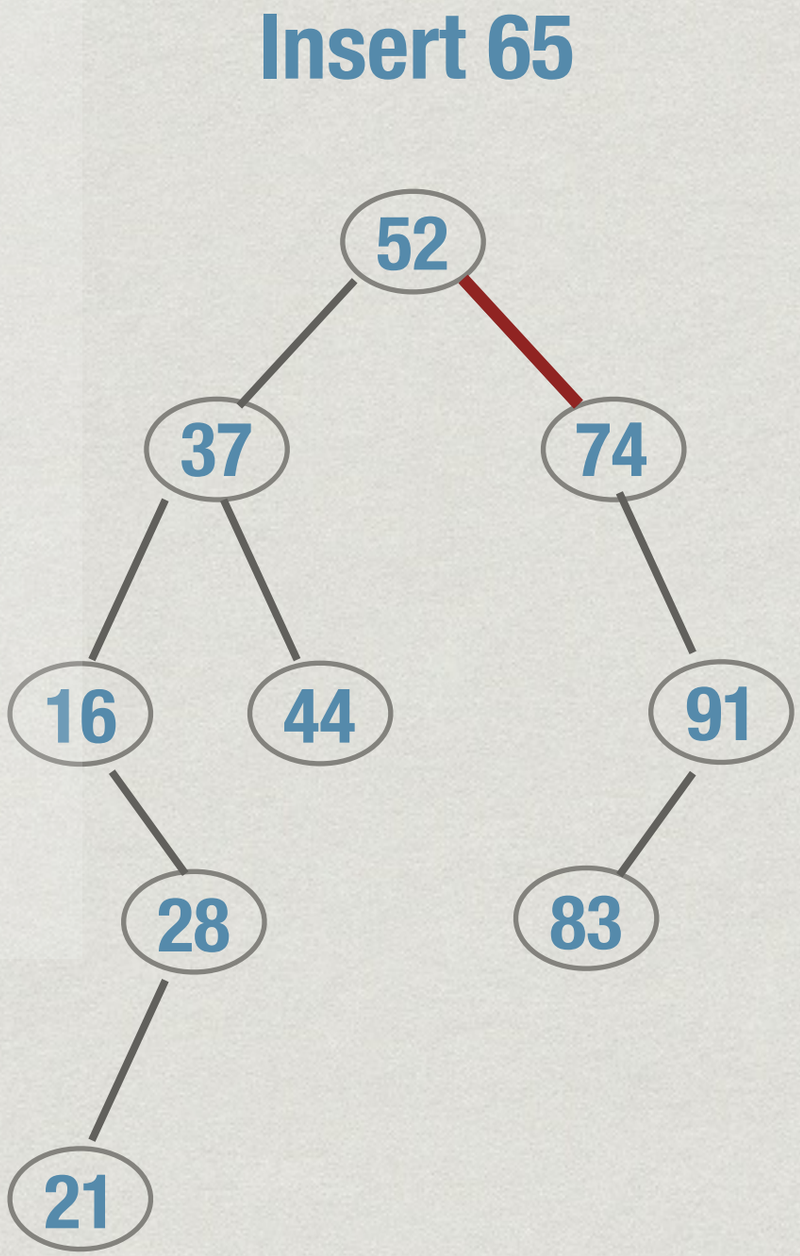
Insert v

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



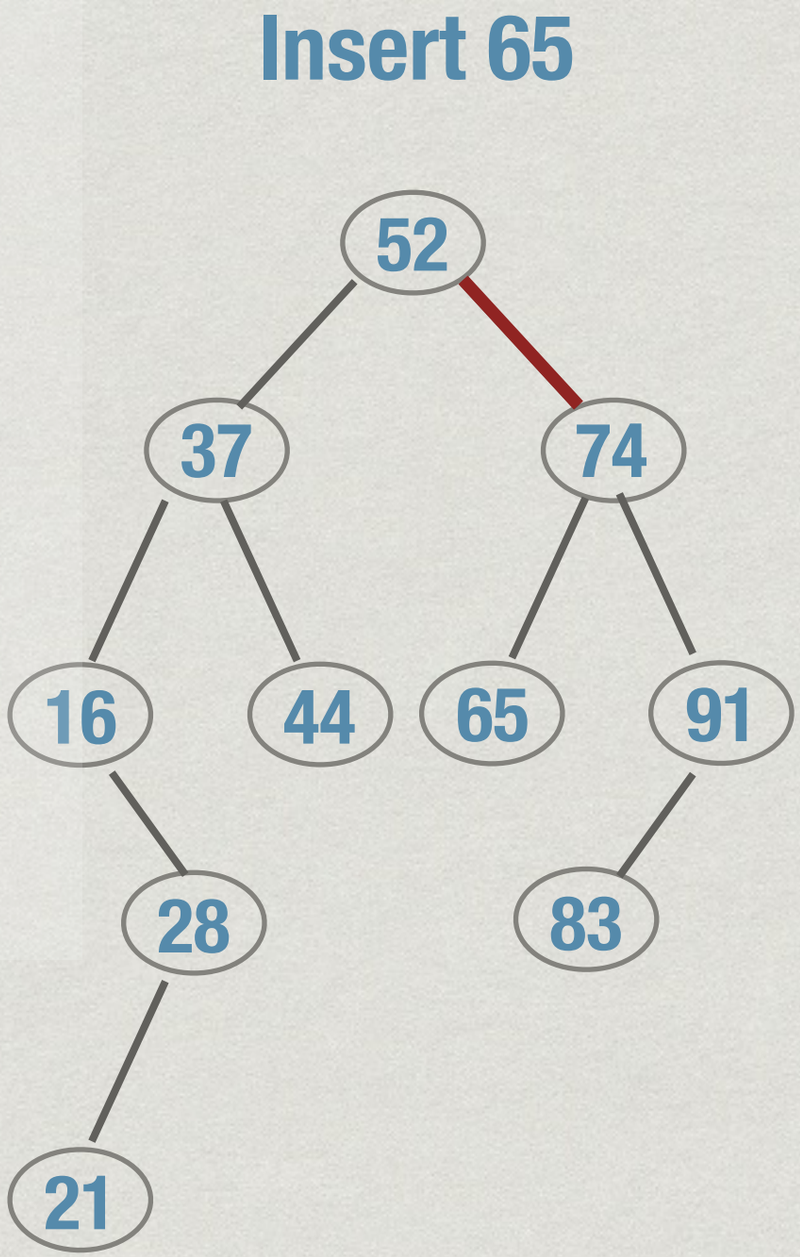
Insert v

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



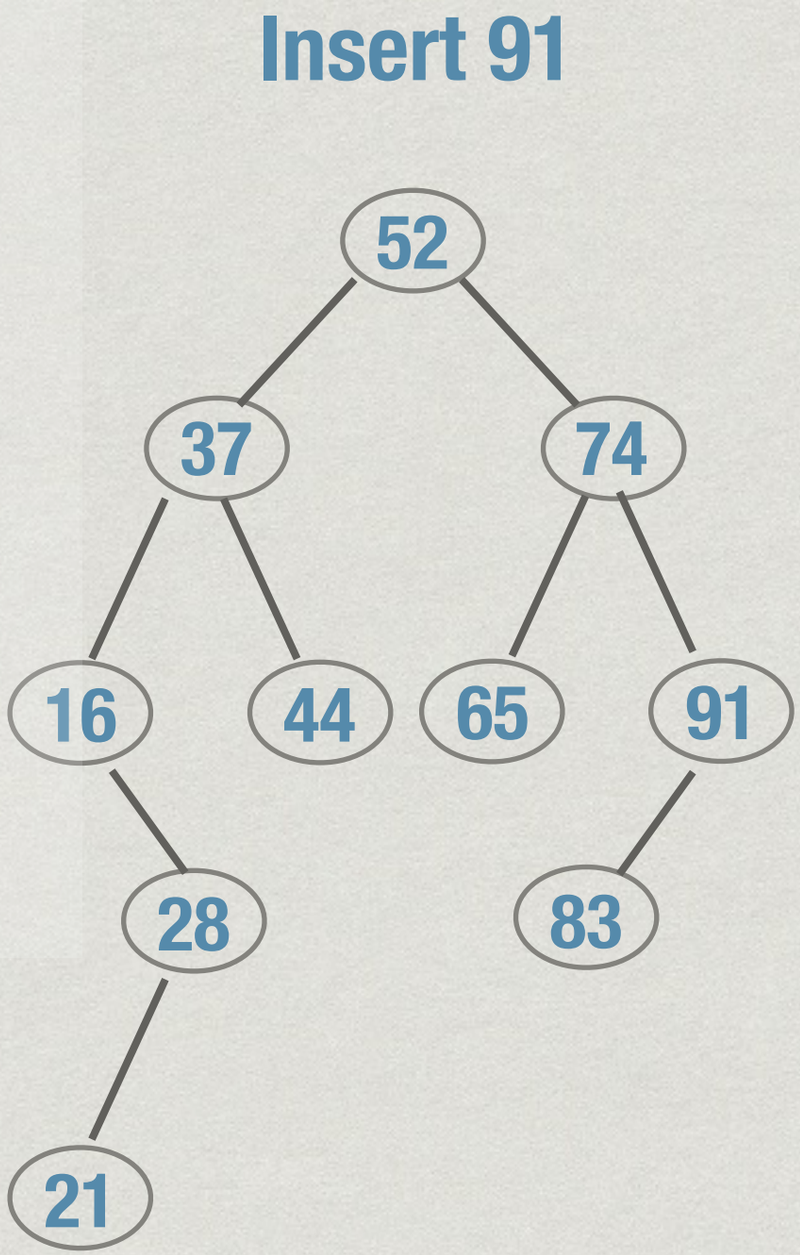
Insert v

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



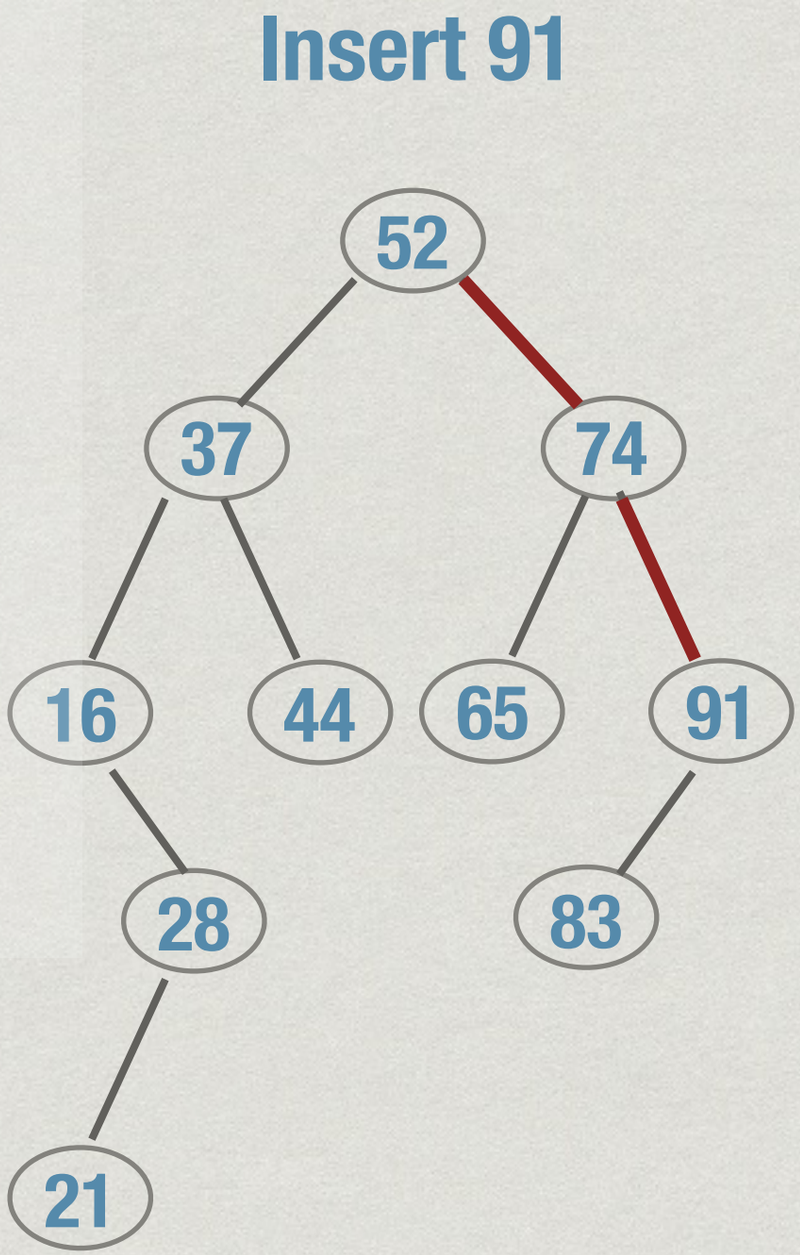
Insert v

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



Insert v

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



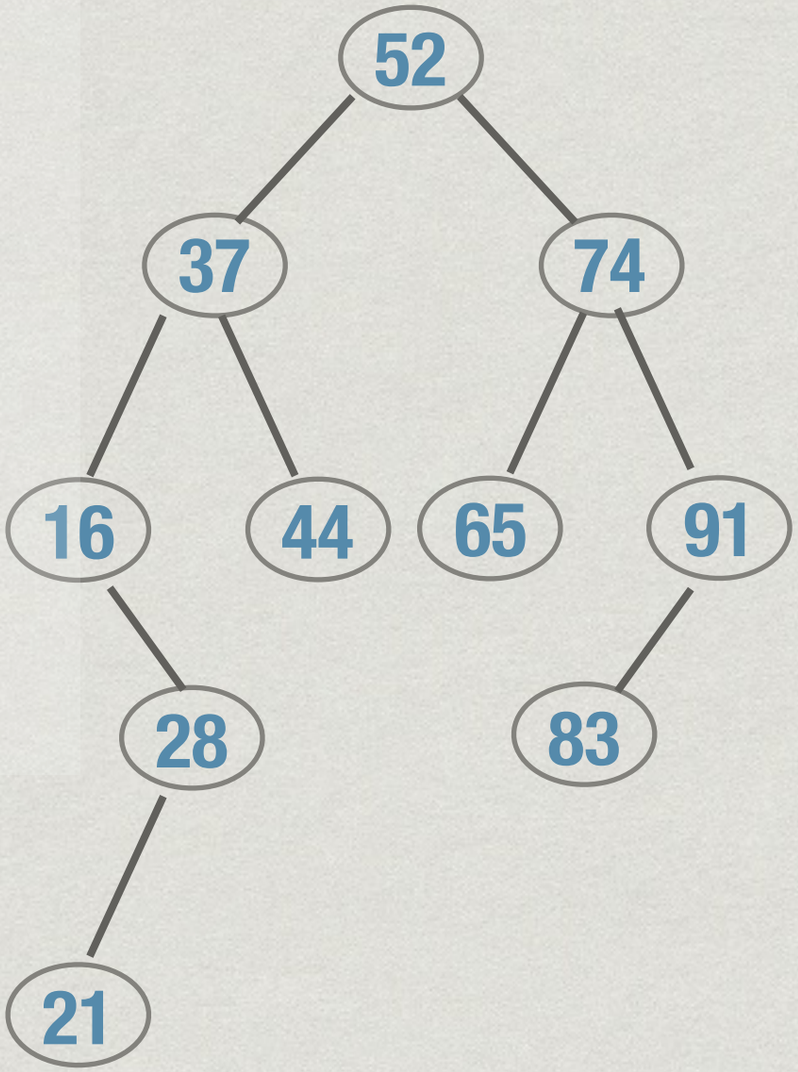
Insert v

```
def insert(self,v):

  if self.isempty():    # Add v as a new leaf
    self.value = v
    self.left = Tree()
    self.right = Tree()

  if self.value == v:   # Value found, do nothing
    return

  if v < self.value:
    self.left.insert(v)
    return

  if v > self.value:
    self.right.insert(v)
    return
```

Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

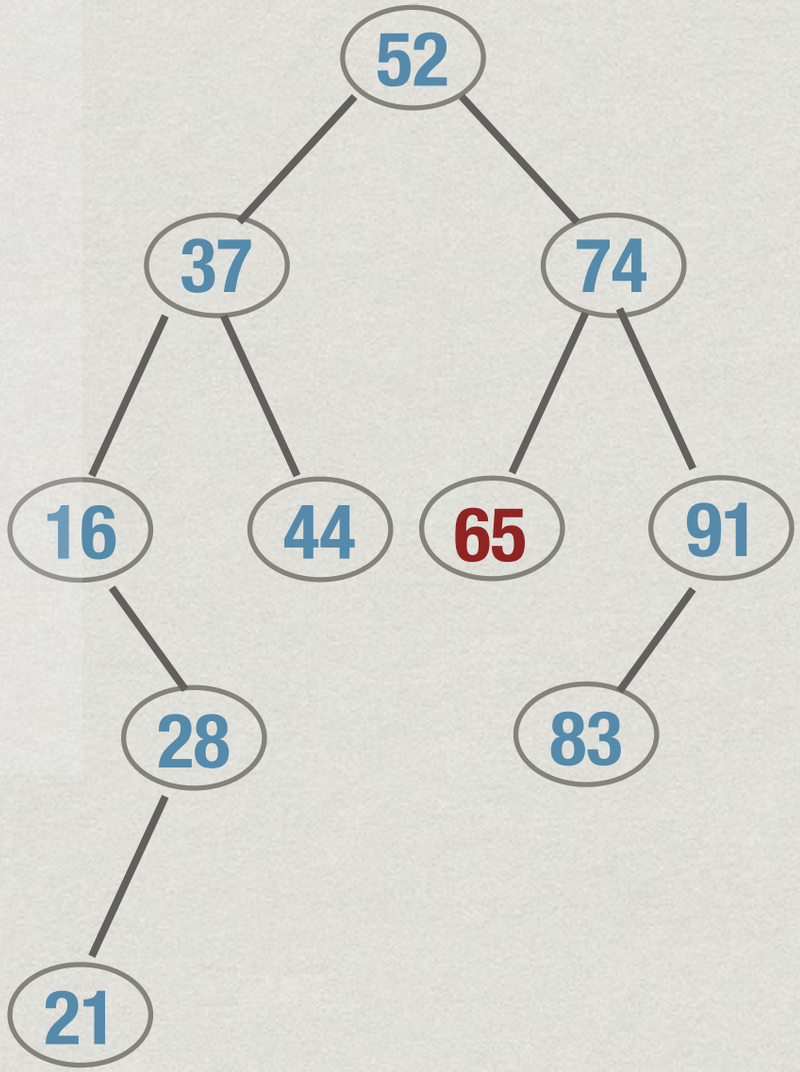
Delete 65



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

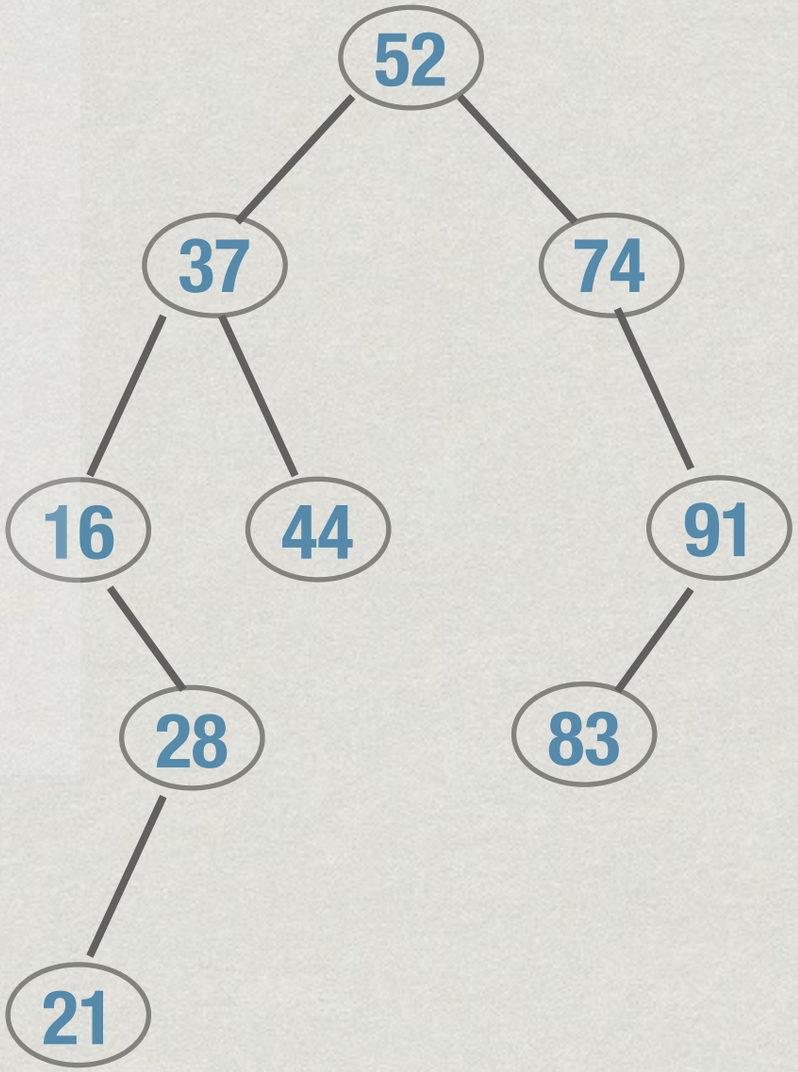
Delete 65



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

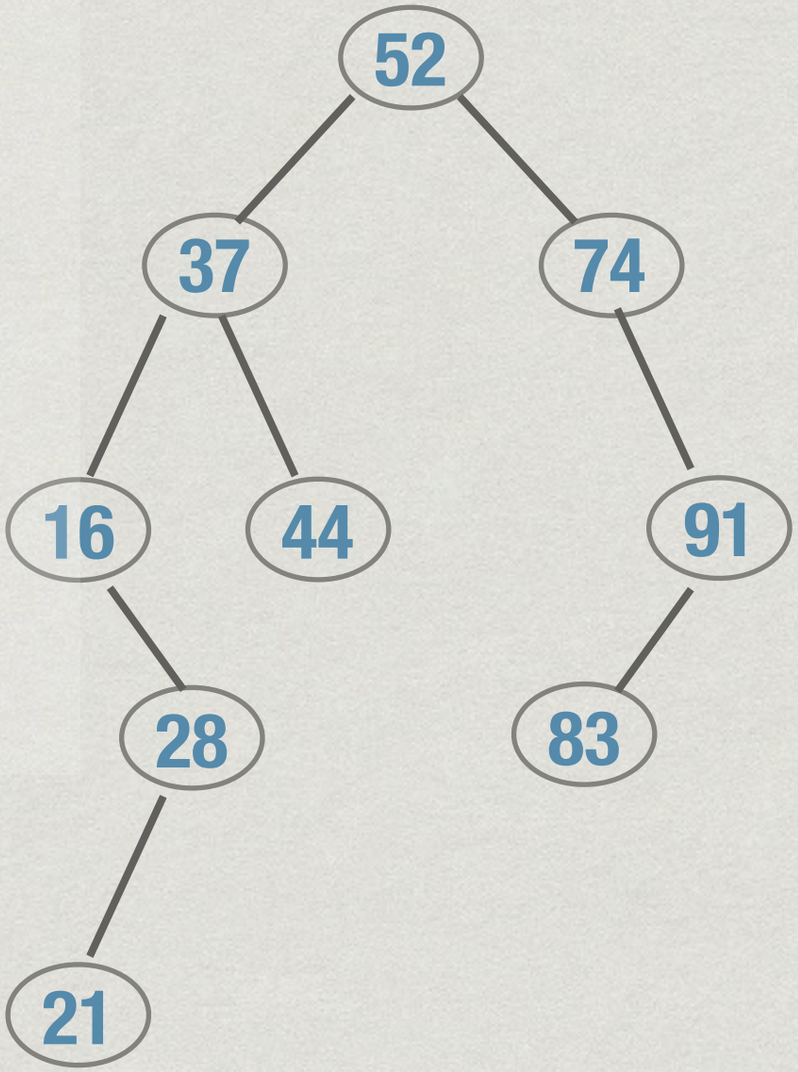
Delete 65



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

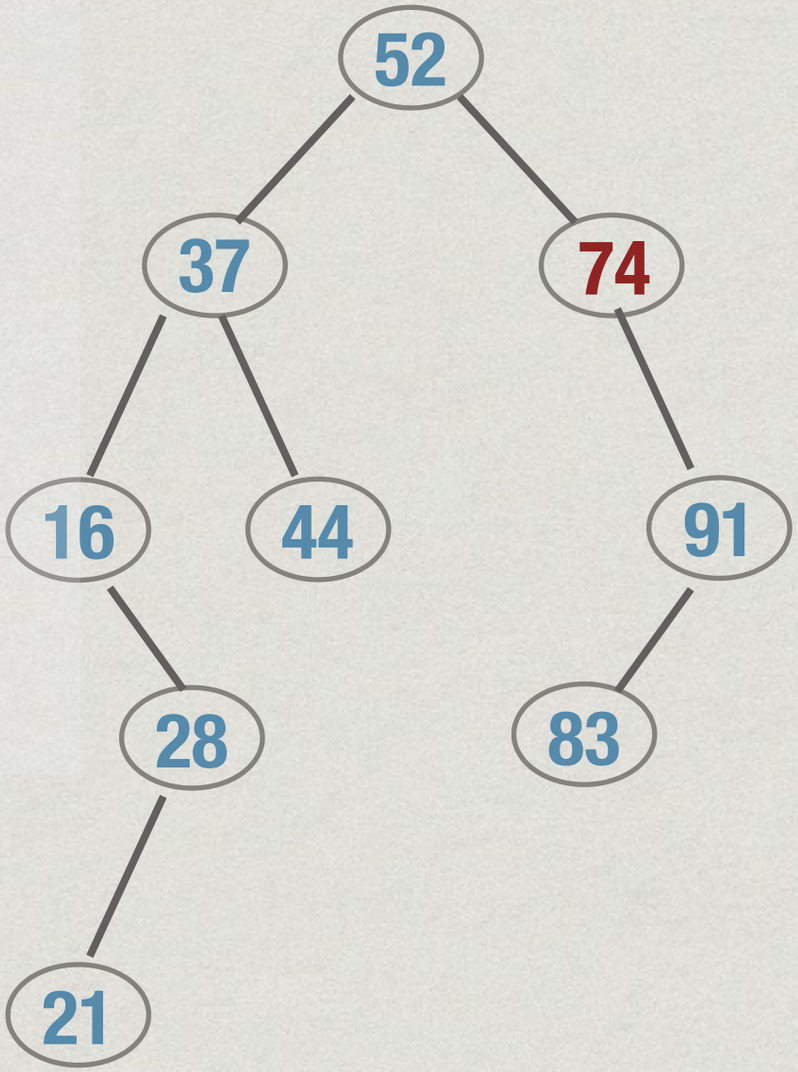
Delete 74



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

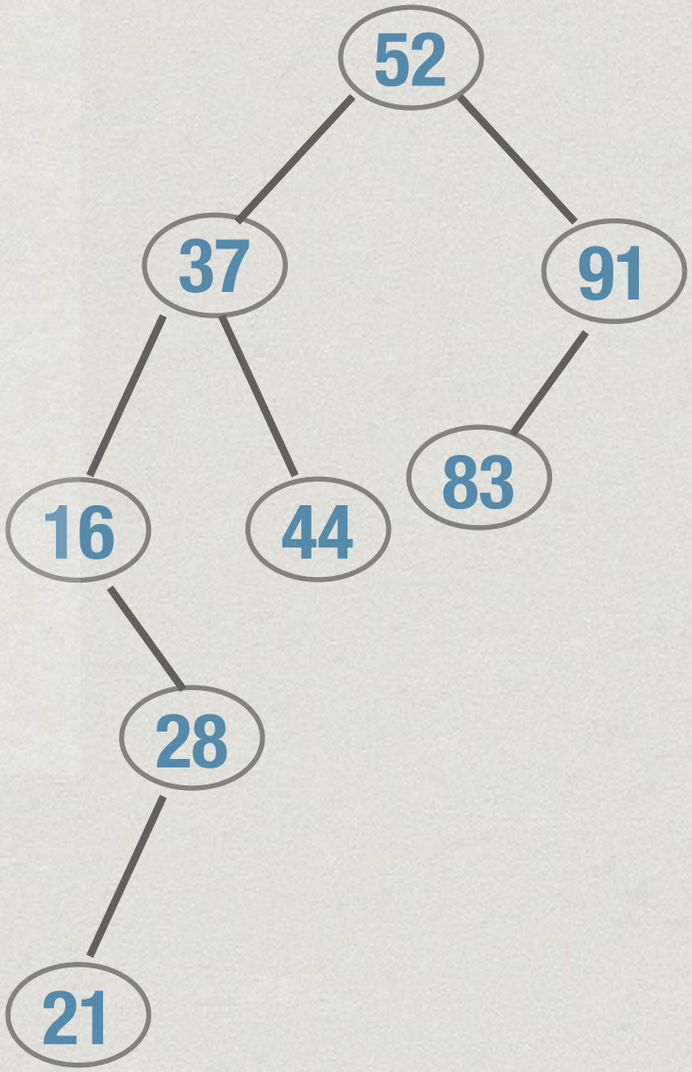
Delete 74



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

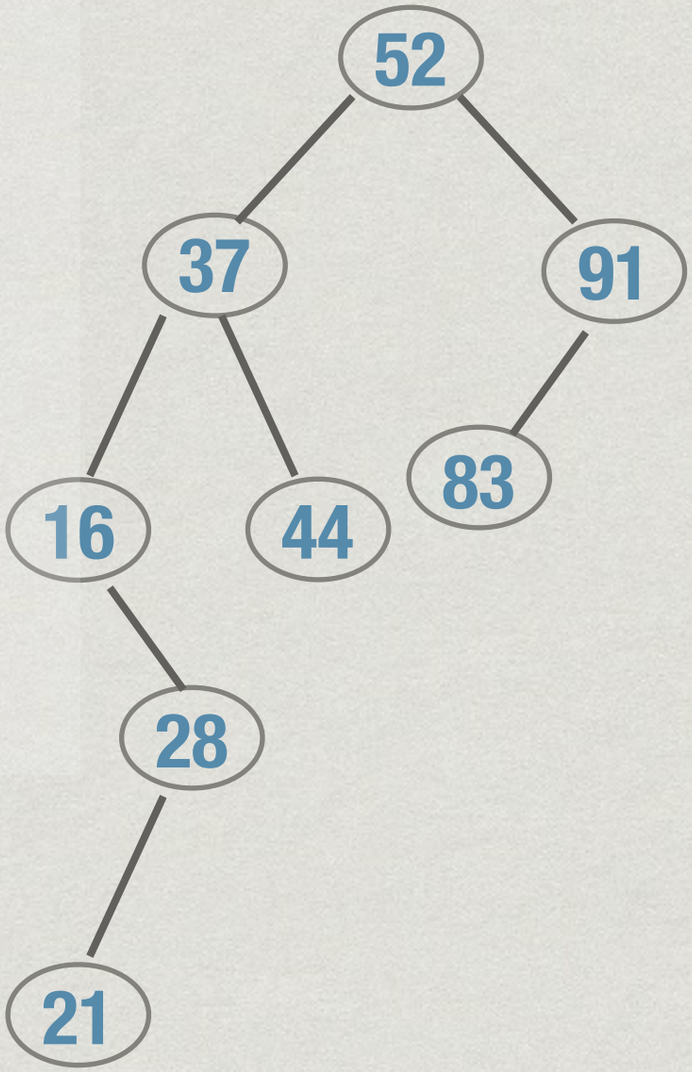
Delete 74



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

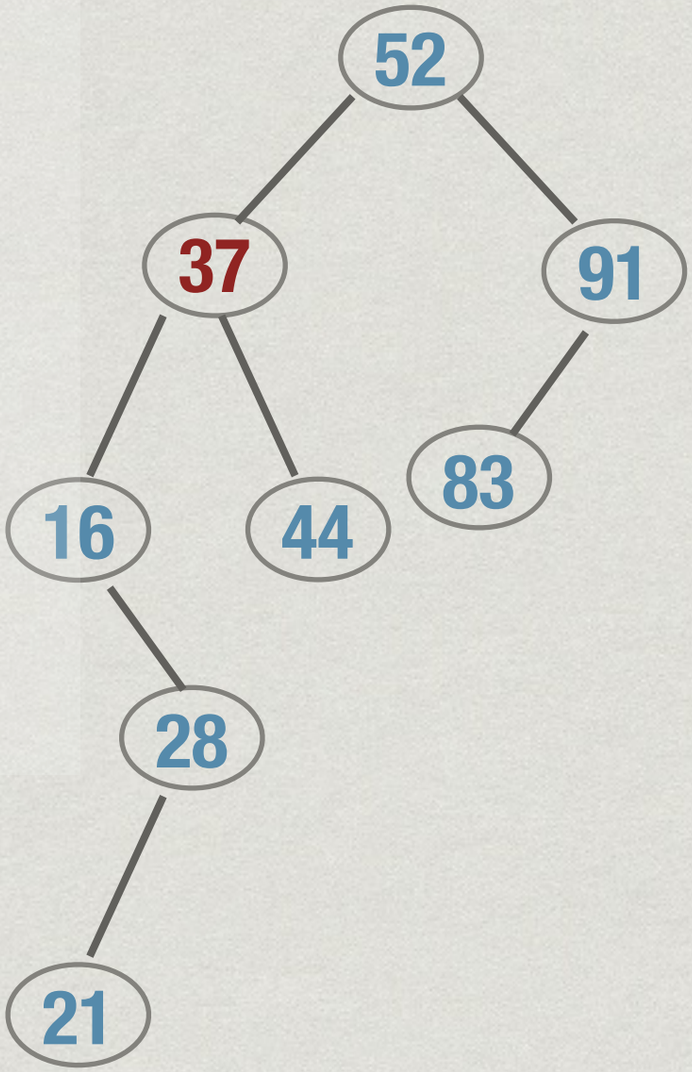
Delete 37



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

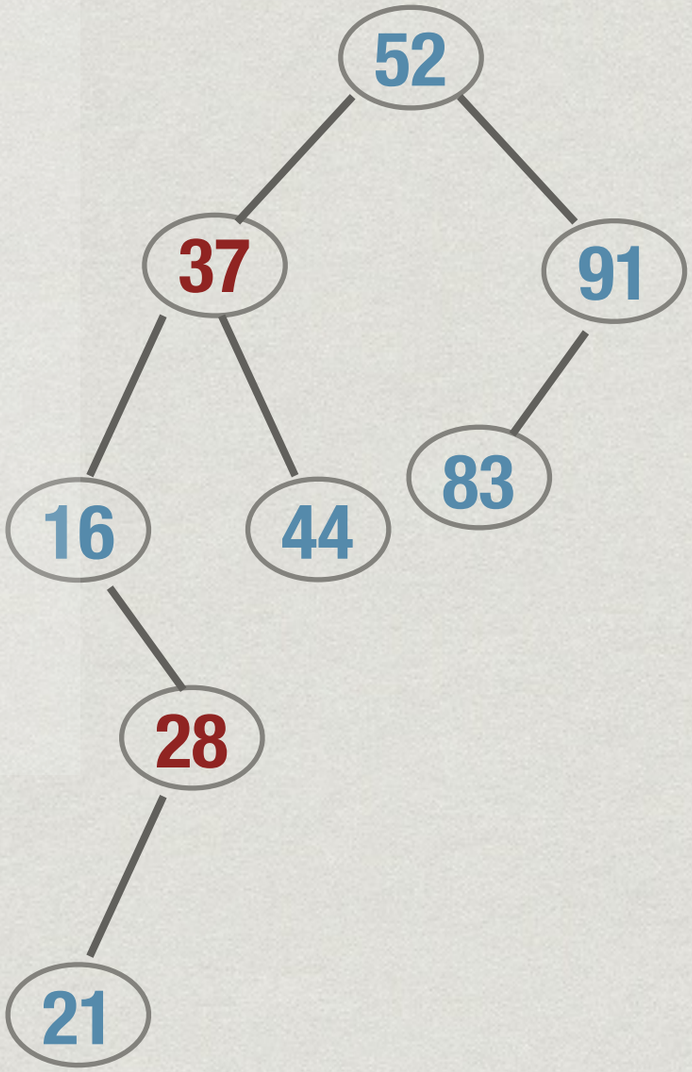
Delete 37



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

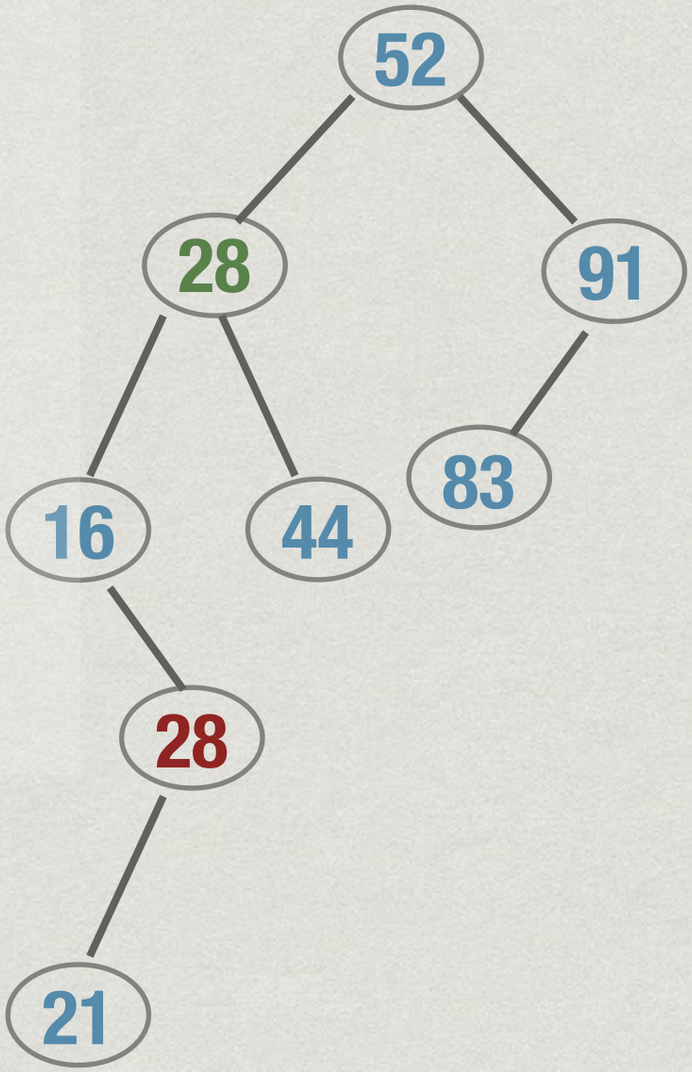
Delete 37



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

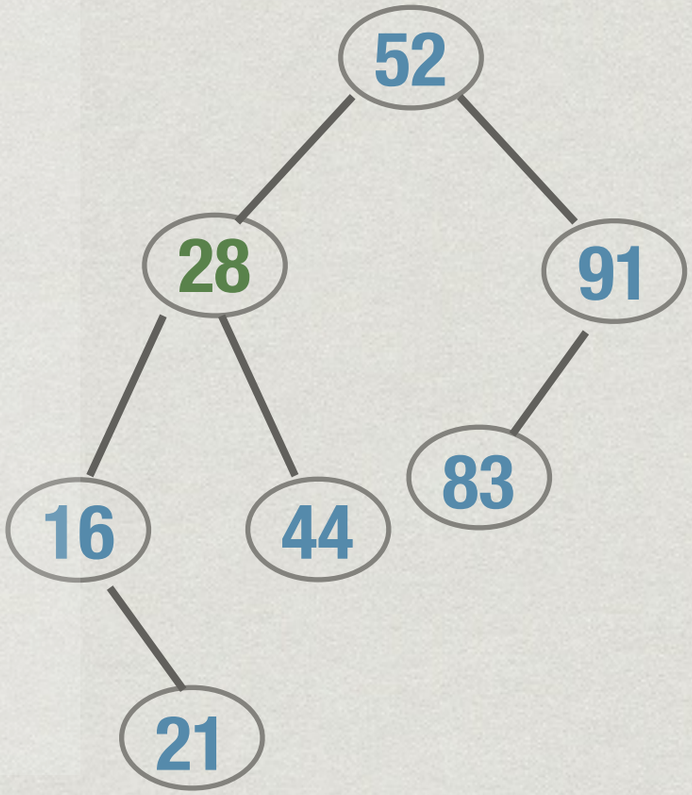
Delete 37



Delete v

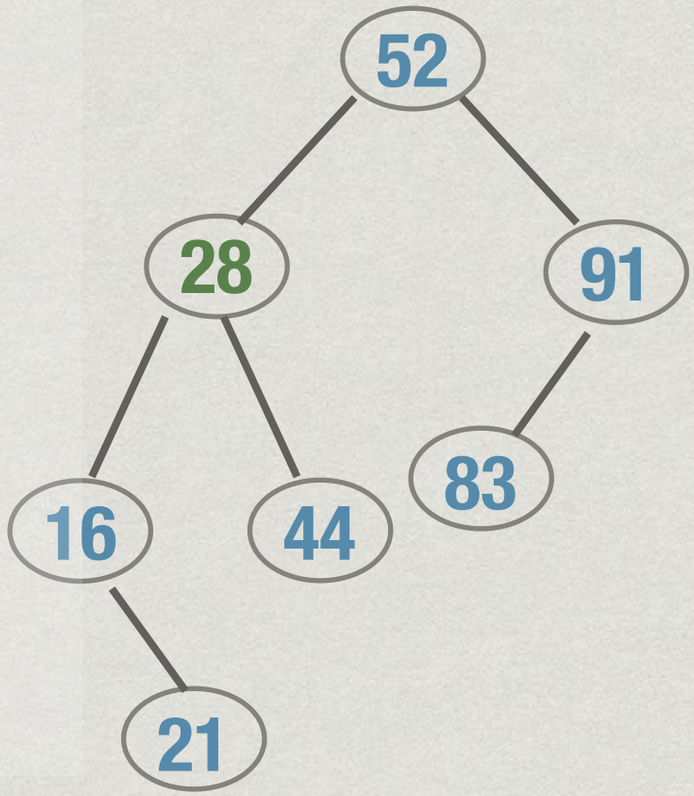
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()` — must be leaf or have only one child

Delete 37



Delete v

- 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 `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child



Delete v

```
def delete(self,v):
  if self.isempty():
    return

  if v < self.value:
    self.left.delete(v)
    return

  if v > self.value:
    self.right.delete(v)
    return

  if v == self.value:
    if self.isleaf():
       self.makeempty()
    elif self.left.isempty():
       self.copyright()
    else:
       self.value = self.left.maxval()
       self.left.delete(self.left.maxval())
    return
```

Delete v

```
def delete(self,v):
  if self.isempty():
    return

  if v < self.value:
    self.left.delete(v)
    return

  if v > self.value:
    self.right.delete(v)
    return

  if v == self.value:
    if self.isleaf():
       self.makeempty()
    elif self.left.isempty():
       self.copyright()
    else:
       self.value = self.left.maxval()
       self.left.delete(self.left.maxval())
    return
```

```
# Convert leaf to
# empty node
def makeempty(self):
  self.value = None
  self.left = None
  self.right = None
  return
```

Delete v

```
def delete(self,v):
  if self.isempty():
    return

  if v < self.value:
    self.left.delete(v)
    return

  if v > self.value:
    self.right.delete(v)
    return

  if v == self.value:
    if self.isleaf():
       self.makeempty()
    elif self.left.isempty():
       self.copyright()
    else:
       self.value = self.left.maxval()
       self.left.delete(self.left.maxval())
    return
```

```
# Convert leaf to
# empty node
def makeempty(self):
  self.value = None
  self.left = None
  self.right = None
  return


# Copy right child values
# to current node
def copyright(self):
  self.value = self.right.value
  self.left = self.right.left
  self.right = self.right.right
  return
```

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
- Tree can be balanced using rotations — look up AVL trees