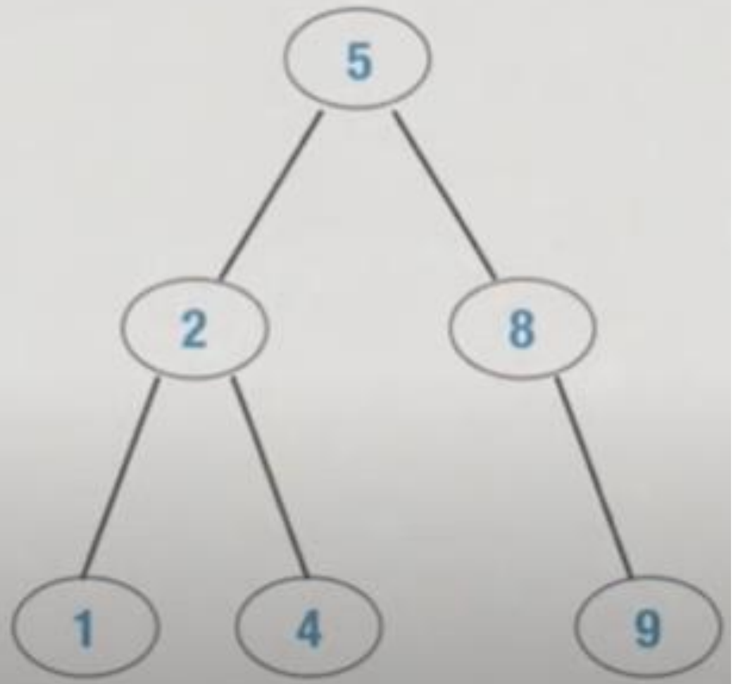
# Binary search tree

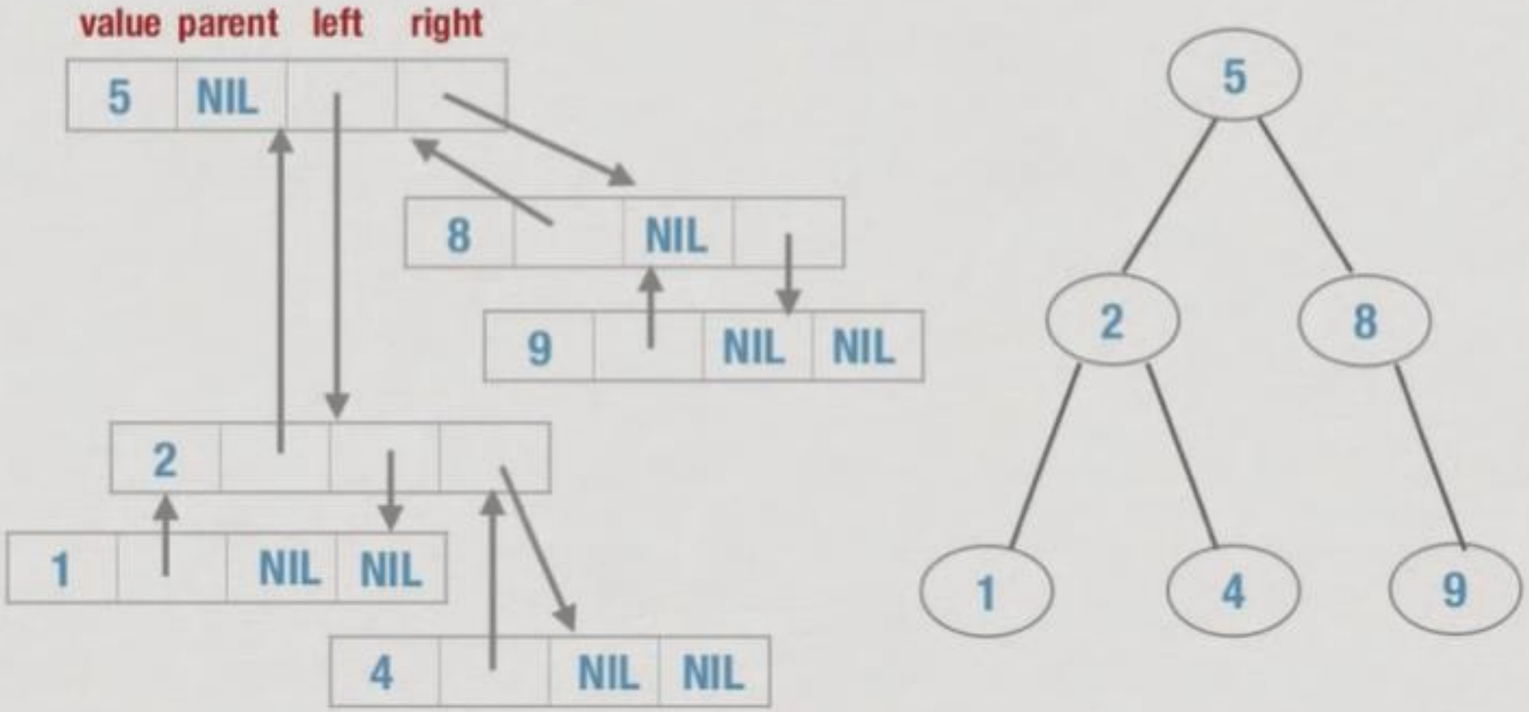
# Binary search tree

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



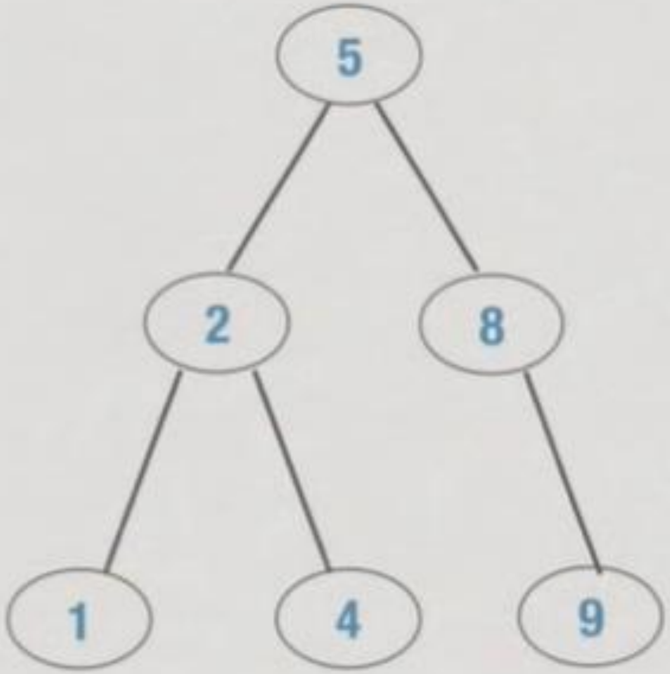
# Binary search tree

- Implement using pointers



# Inorder traversal

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




- Lists values in sorted order

1 2 4 5 8 9

# find(v)

**Recursive**

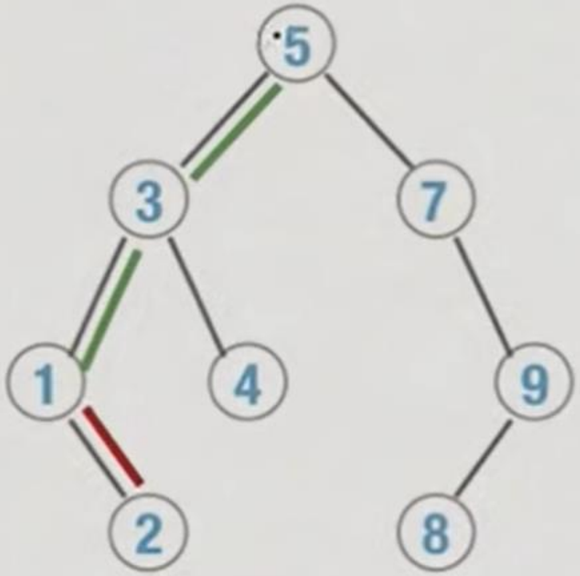
```
function find(t,v)

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))
```

**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

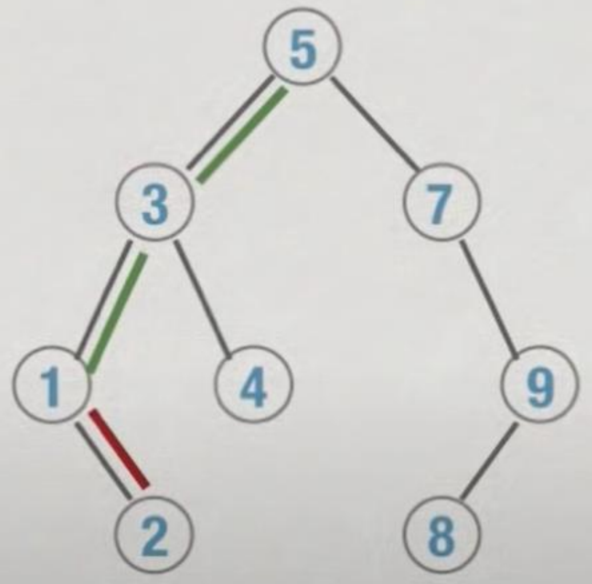


# Minimum

- Left most node in the tree

**Recursive**

```
function minval(t)

# Assume t is not empty

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

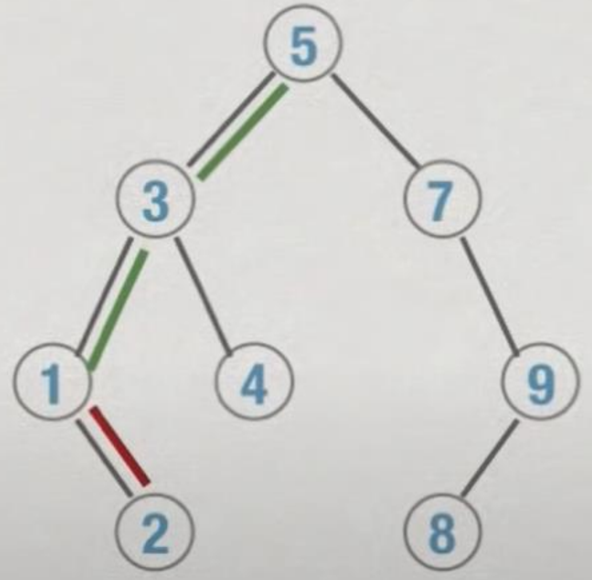


# Minimum

- 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)
```

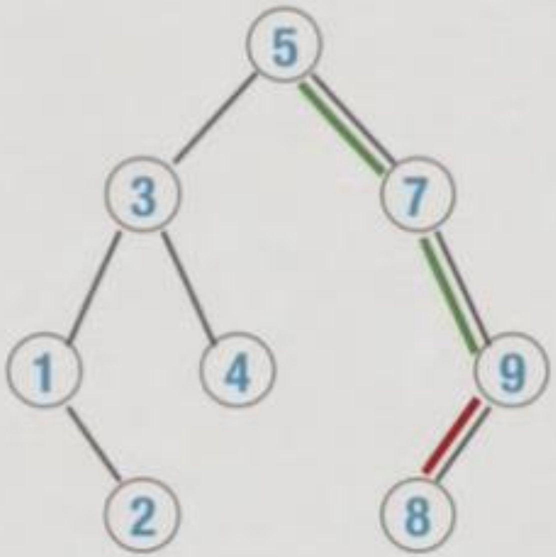


# Maximum

- Right most node in the tree

**Recursive**

```
function maxval(t)

# Assume t is not empty

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

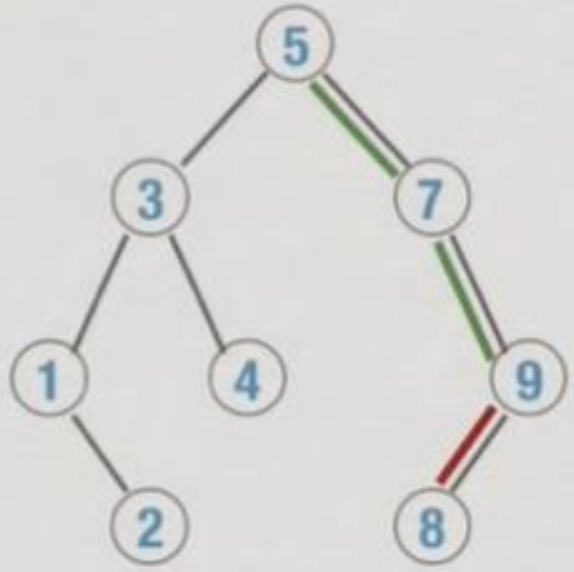


# Maximum

- Right most node in the tree

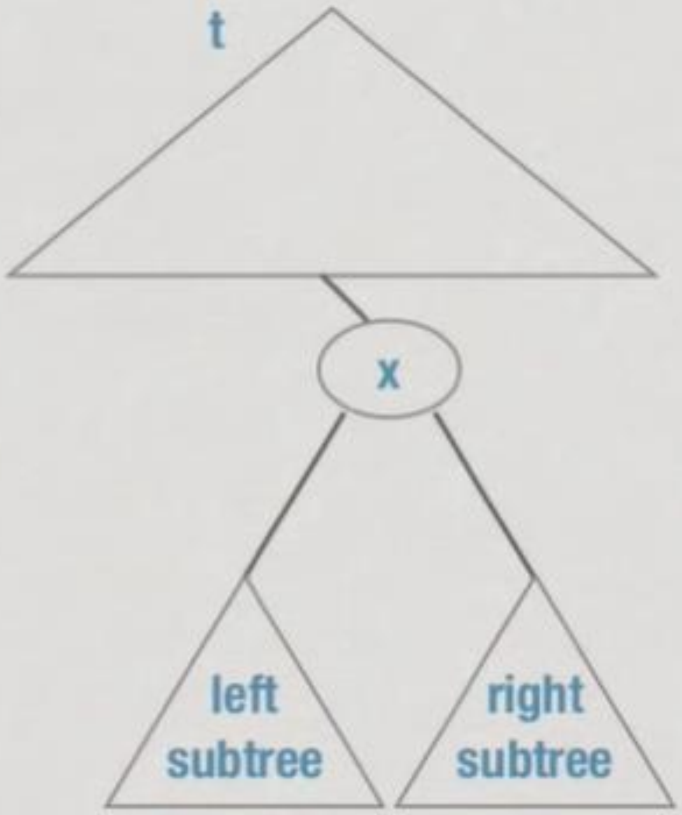
**Iterative**

```
function maxval(t)

# Assume t is not empty

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

return(t.value)
```



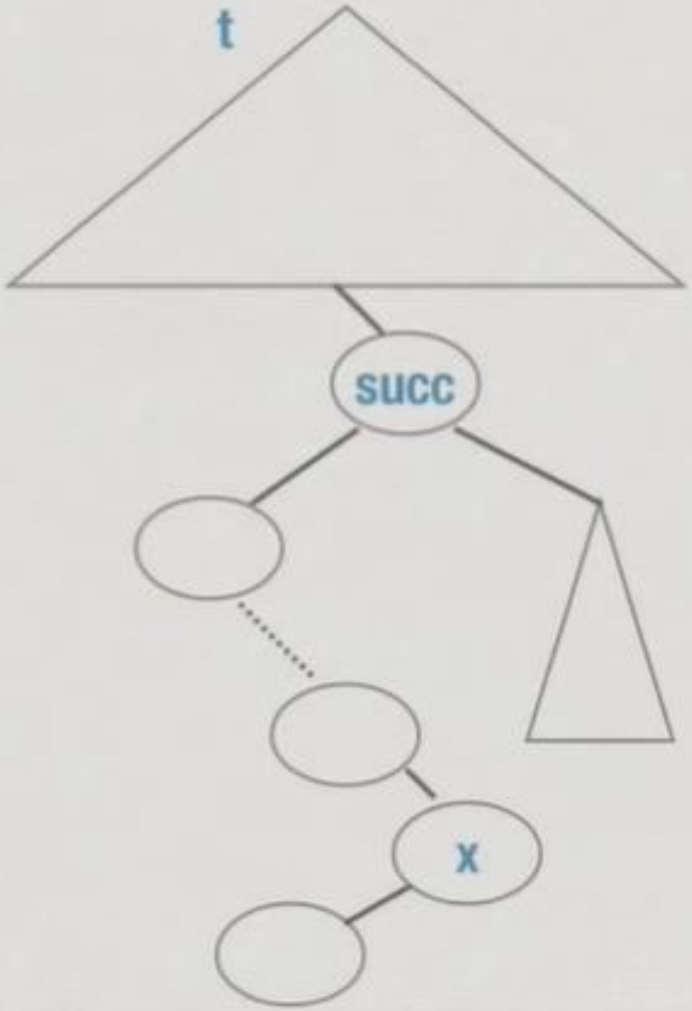
# Successor

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



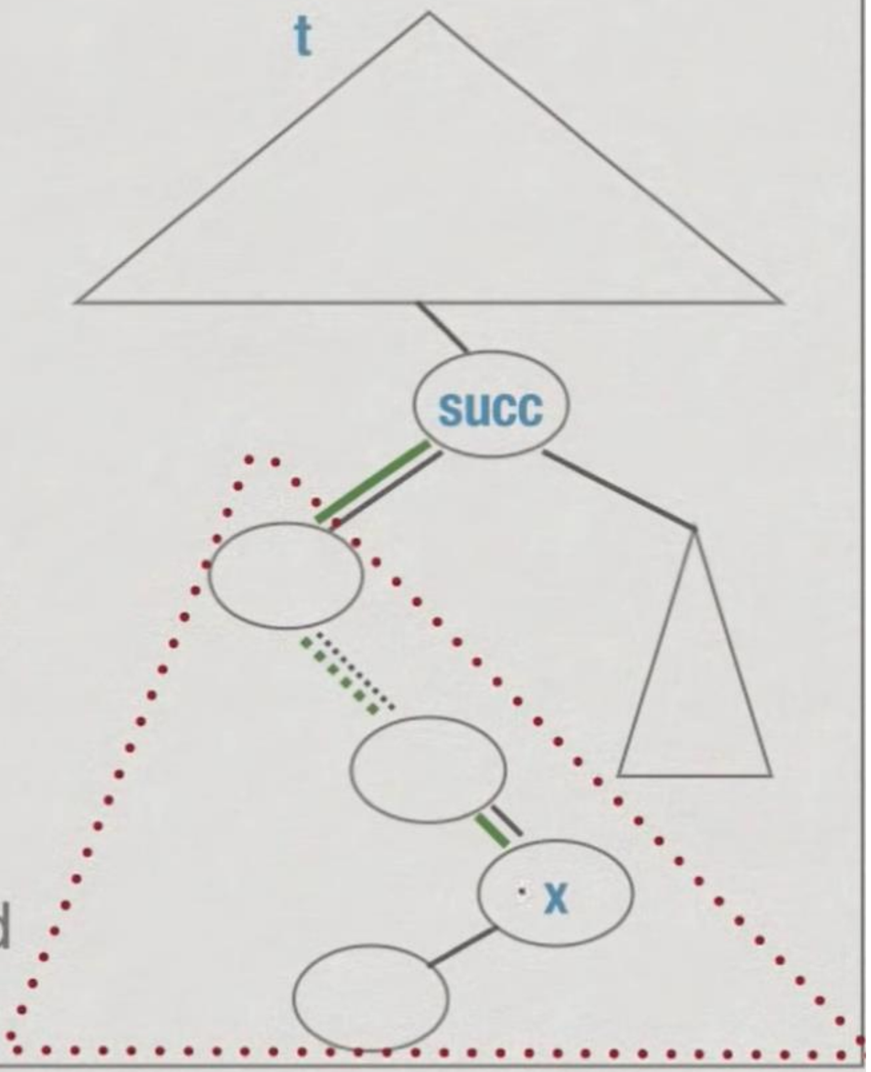
# Successor

- 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



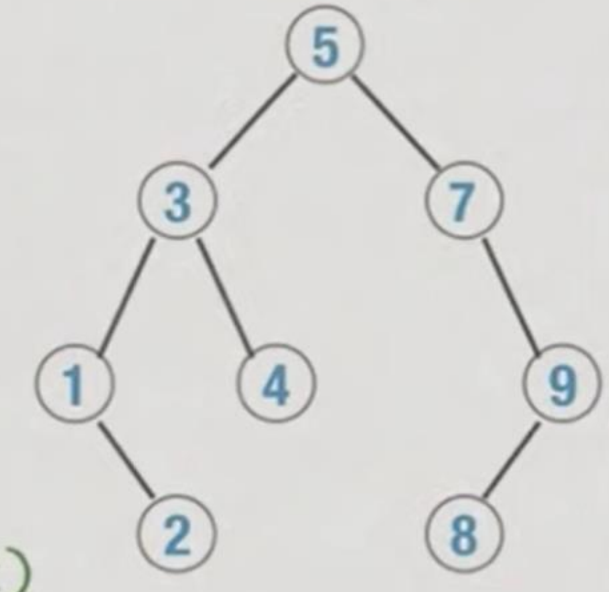
# Successor

* 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

```
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)
```




if no right sub tree then

2- 3

4-5

8-9

9- no ?

3 -4

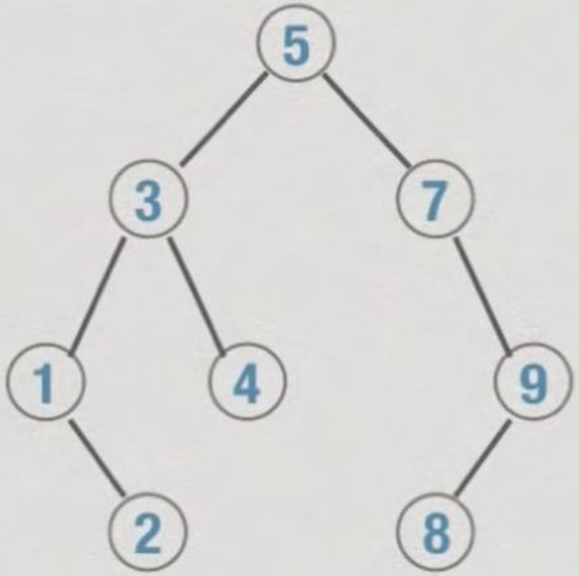
1-2

7-8

# 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)
```

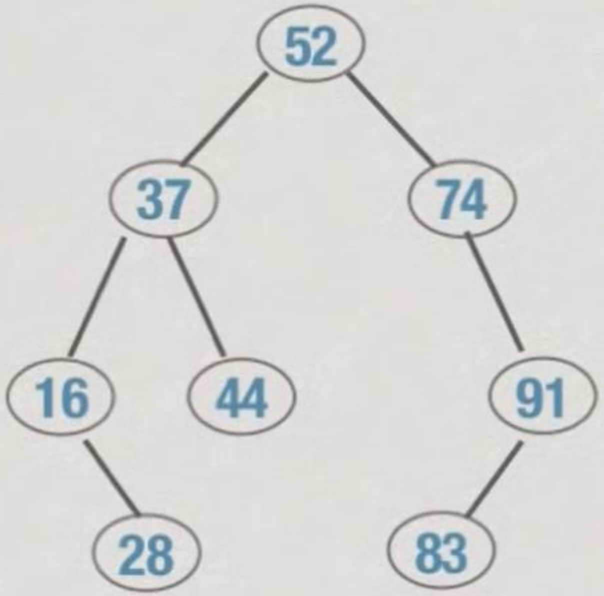



Left side
5-4
3-2
9-8
1-no?

No left side
2-1
4-3
7-5
8-7

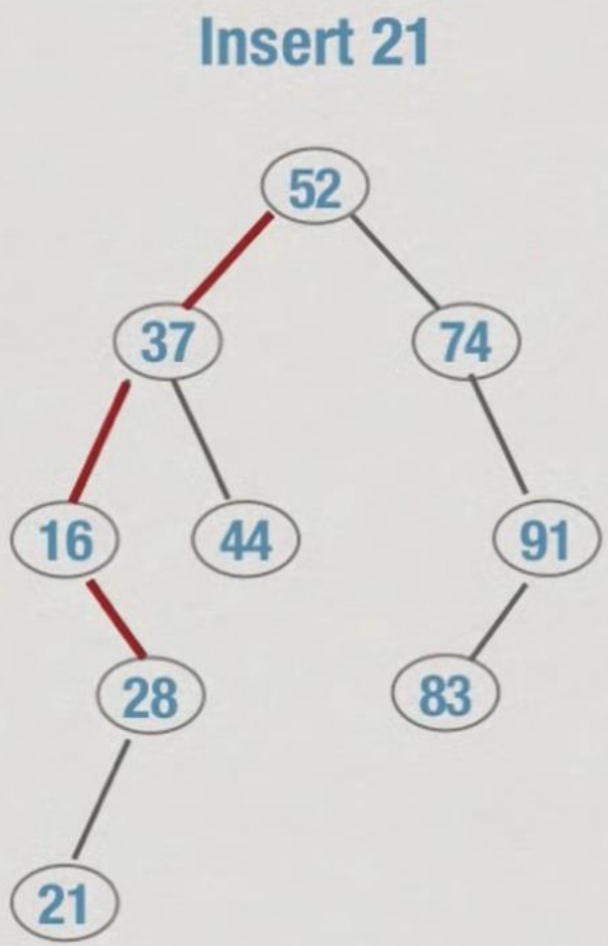
# Insert

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



# Insert

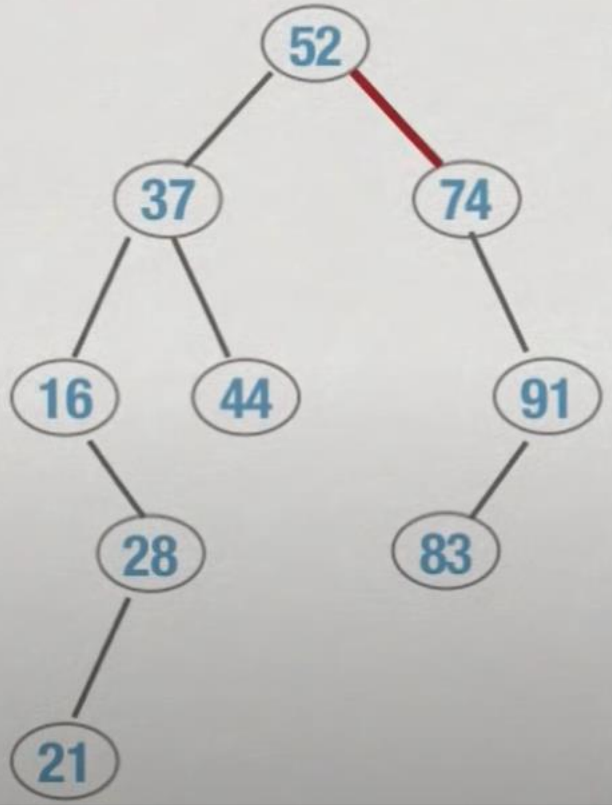
- Try to find v
- If it is not present, add it where the search fails



# Insert

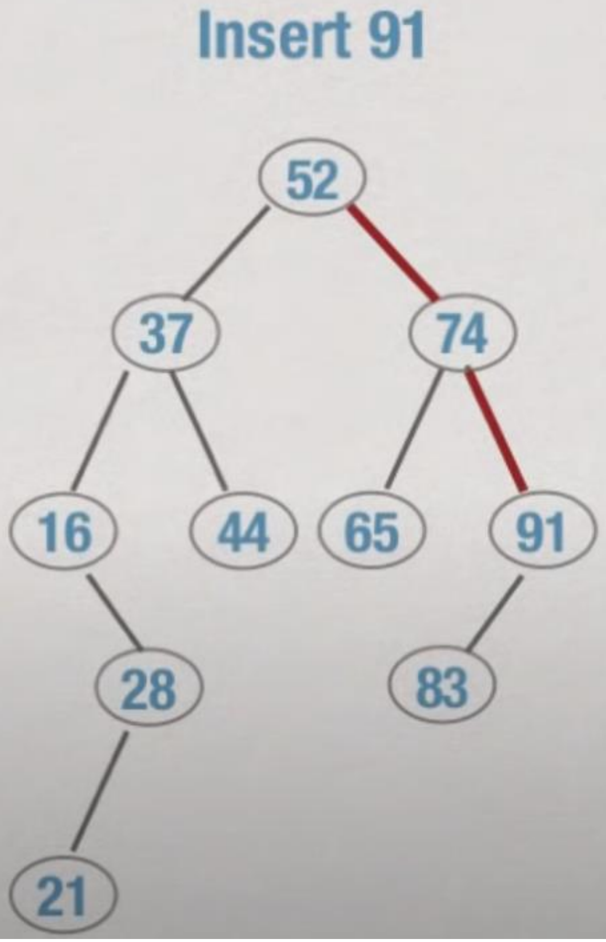
* Try to find v
* If it is not present, add it where the search fails

Insert 65



# Insert

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

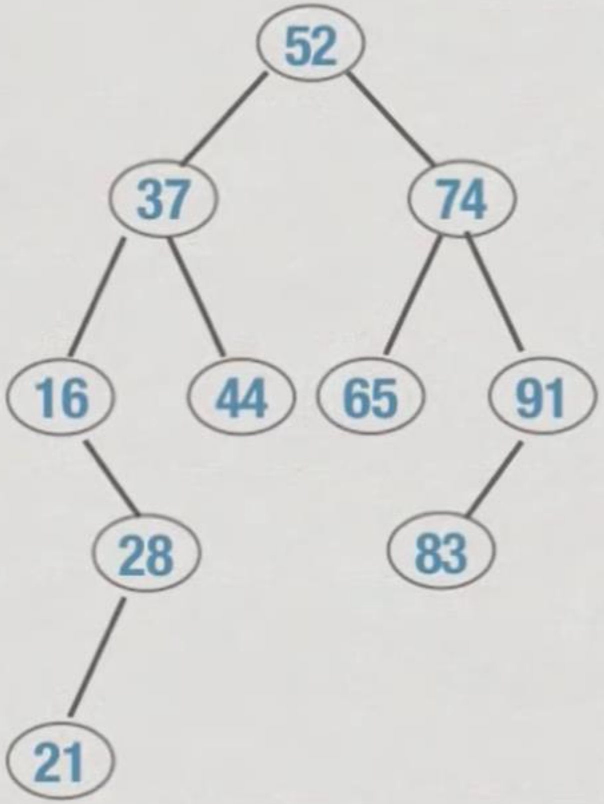


# insert(v)

```
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)
```

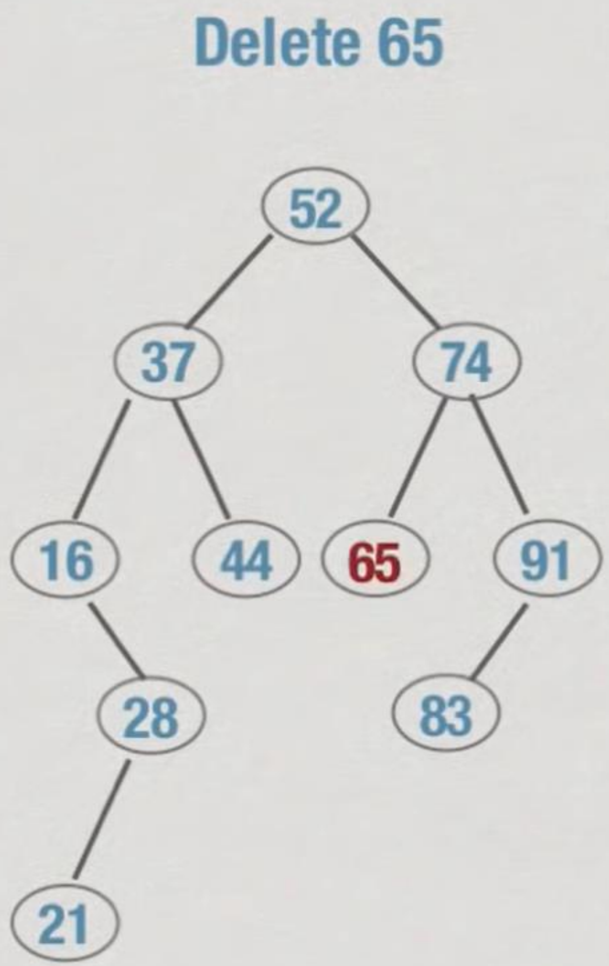
# Delete

- 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

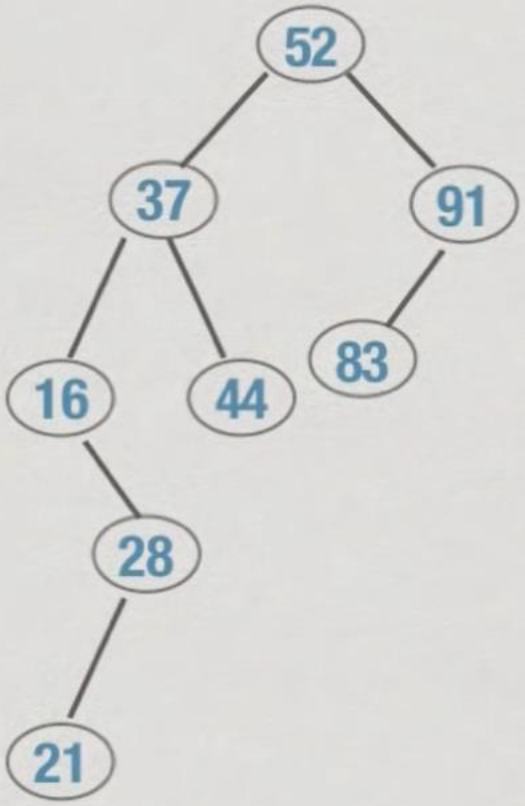
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
    - Delete pred(v) / succ(v)
    - Either leaf or only one child



# Delete

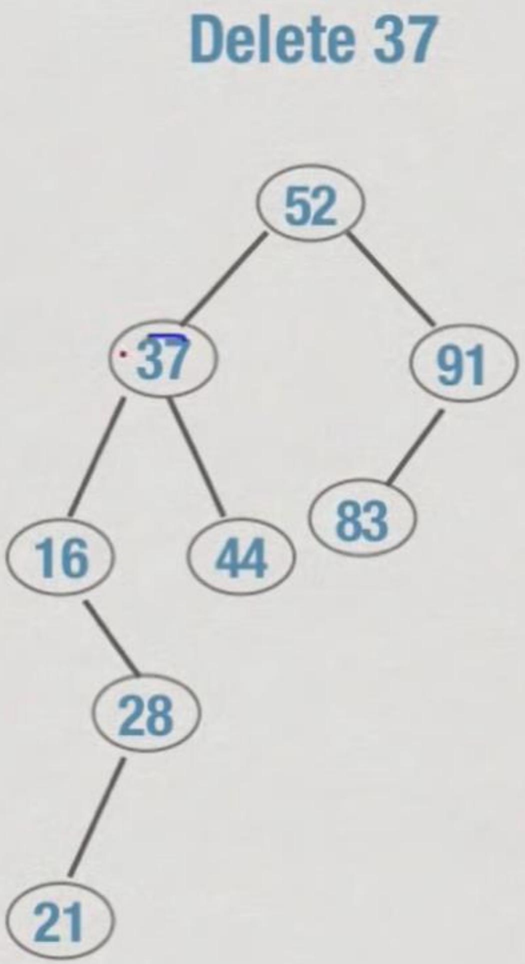
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
    - Delete pred(v) / succ(v)
    - Either leaf or only one child

**Delete 74**



# Delete

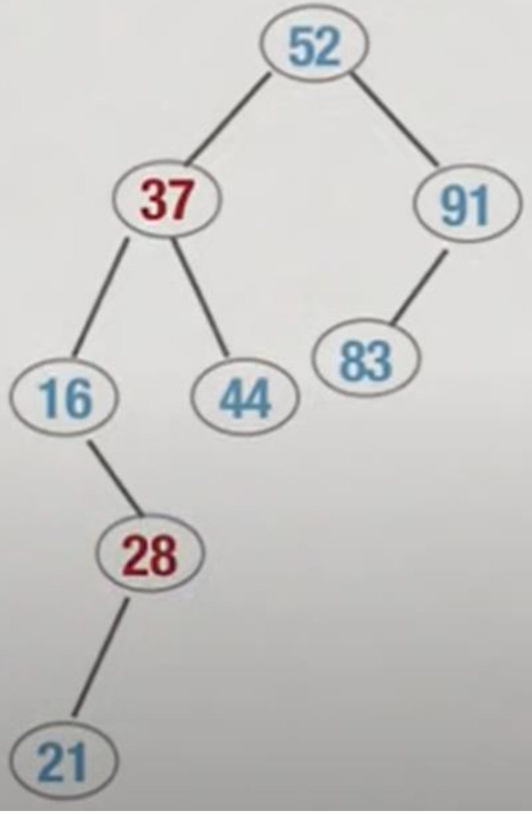
- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, “promote” that child
- If deleted node has two children, fill in the hole with pred(v) or succ(v)
    - Delete pred(v) / succ(v)
    - Either leaf or only one child



# 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 37



# 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
```