ECS32B

Introduction to Data Structures

Trees

Lecture 22

Announcements

- SLAC tonight 6:30-9:00 in 73 Hutchison
- Homework 5 Wednesday at 11:59pm

Textbook Tree Vocabulary

- **Node** A node is a fundamental part of a tree. It can have a name, which we call the **key**. A node may also have additional information. We call this additional information the **payload**. While the payload information is not central to many tree algorithms, it is often critical in applications that make use of trees.

- **Edge** An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by exactly one incoming edge from another node. Each node may have several outgoing edges.

- **Root** The root of the tree is the only node in the tree that has **no incoming edges**.

- **Leaf** A leaf node is a node that has **no outgoing edges.**

- **Path** A path is an ordered list of nodes that are connected by edges.

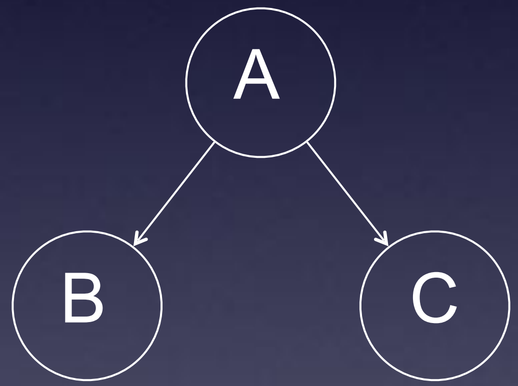
Node Relationships

- **Children** The set of nodes that have incoming edges from the same node to are said to be the children of that node. B and C are children of A.

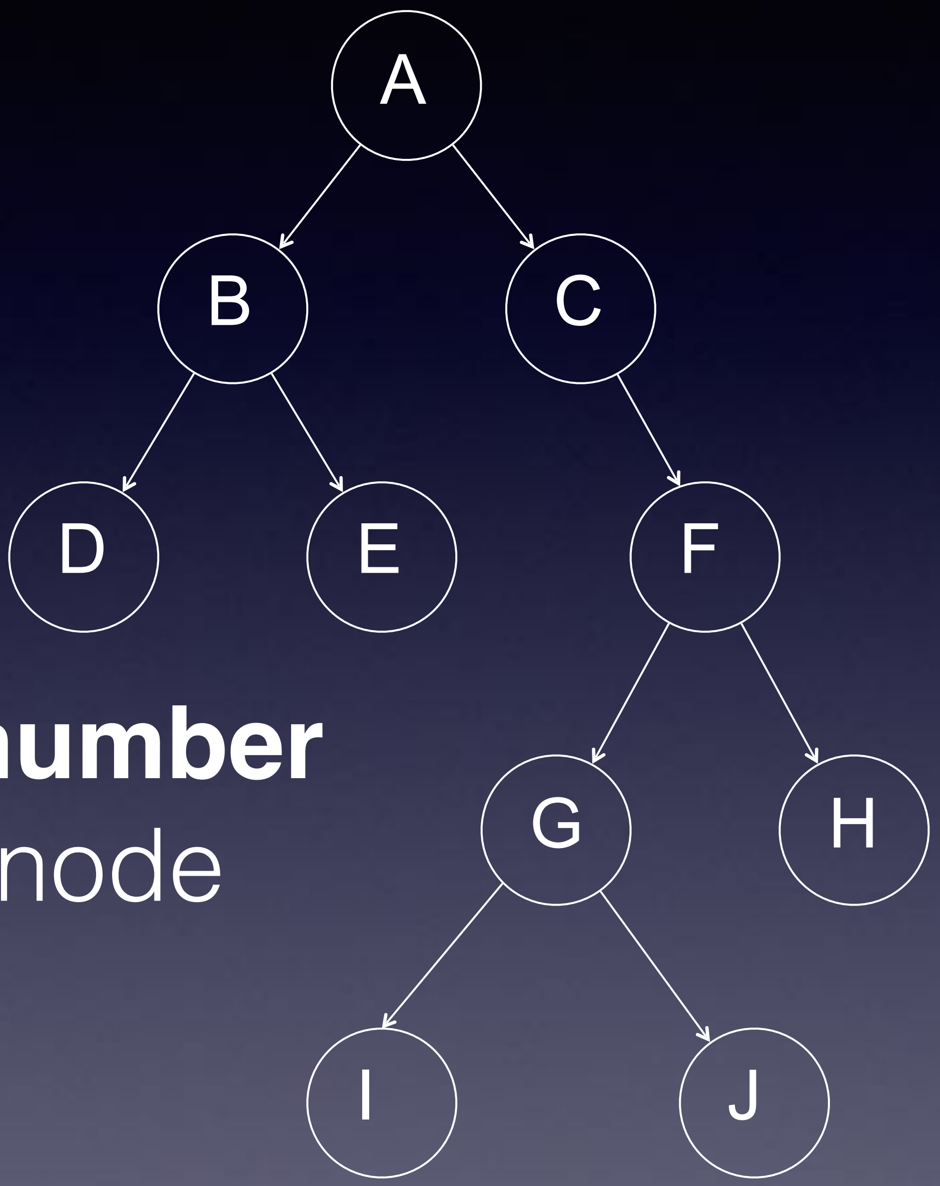
- **Parent** A node is the parent of all the nodes it connects to with outgoing edges. A is the parent of B and C

- **Sibling** Nodes in the tree that are children of the same parent are said to be siblings. B and C are siblings.

- **Subtree** A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent. A, B, and C is a subtree.



Tree Height

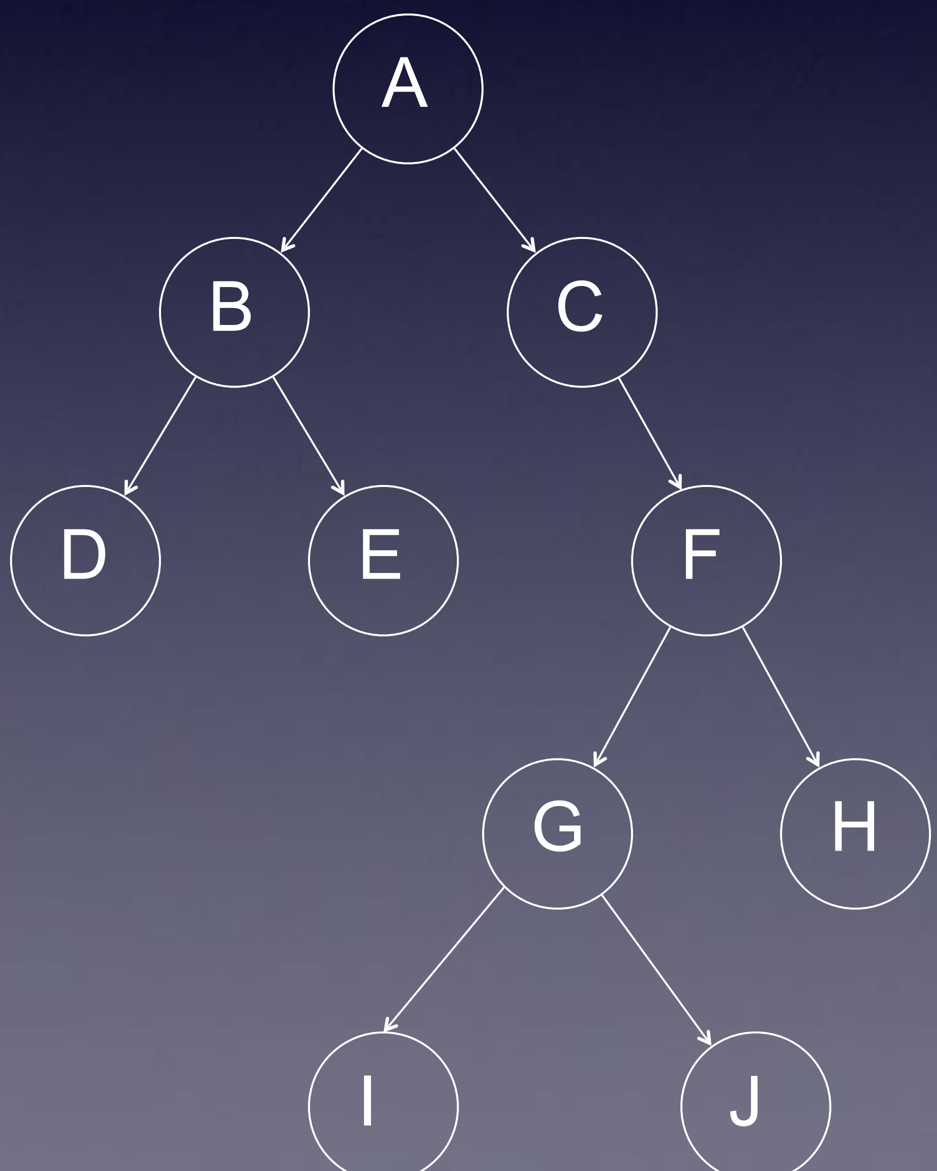


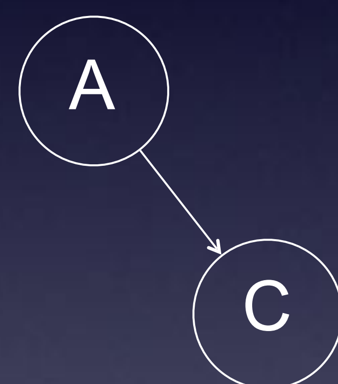
- **Level** The level of a node n is **the number of edges** on the path from the root node to node n.

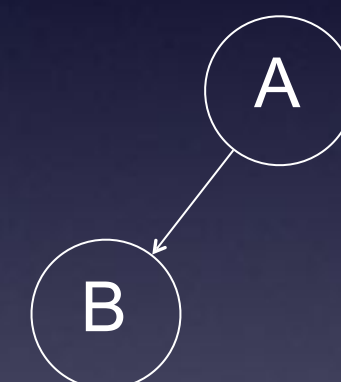
- **Height** The height of a tree is the the maximum level of any node in the tree.

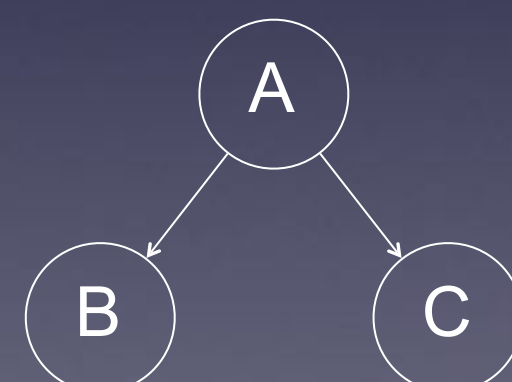
A **binary tree** is a tree that is either

- **empty** or
- a node called the root node and two possibly empty binary trees called the left subtree and a right subtree



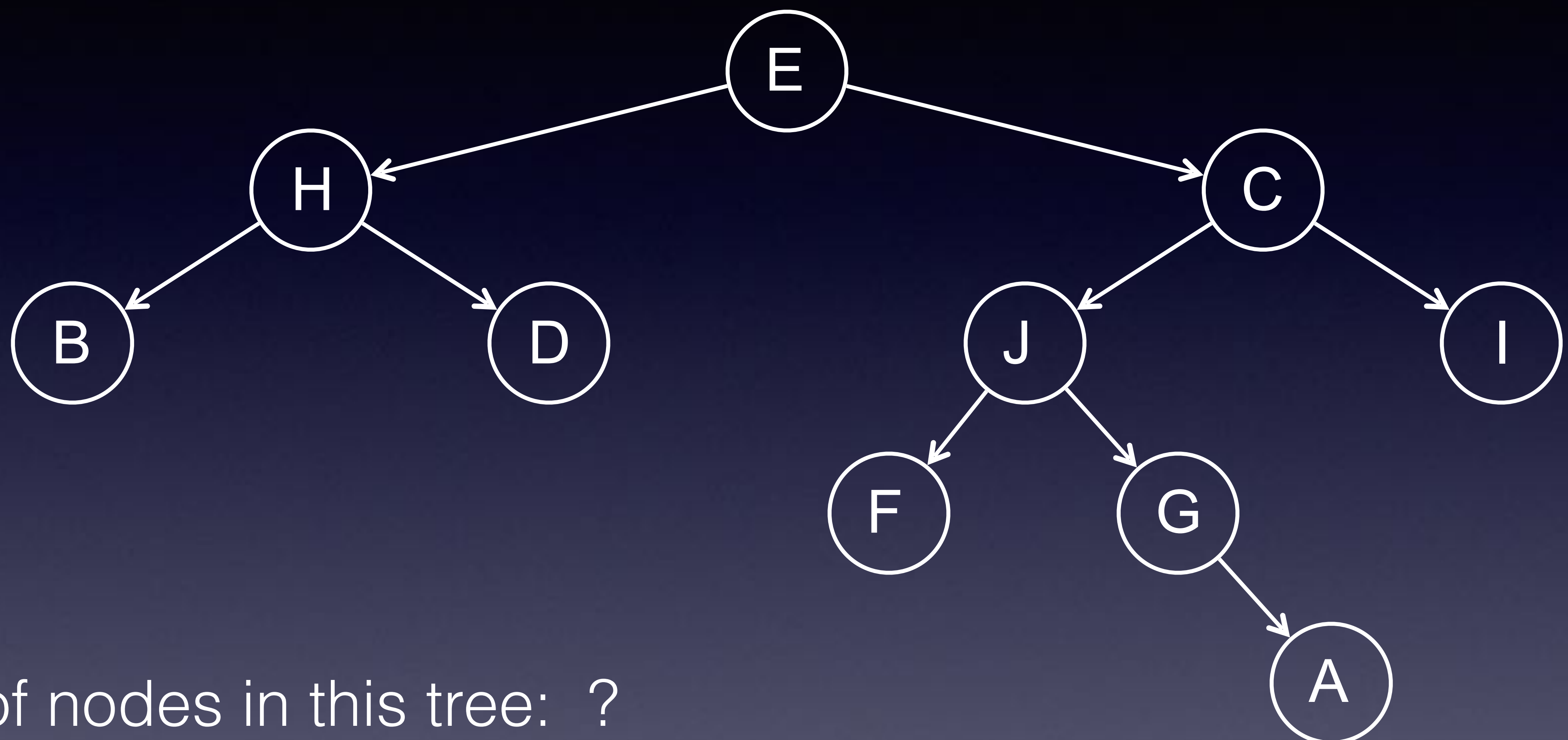






These are all binary trees.

Tree terminology



# of nodes in this tree: ?

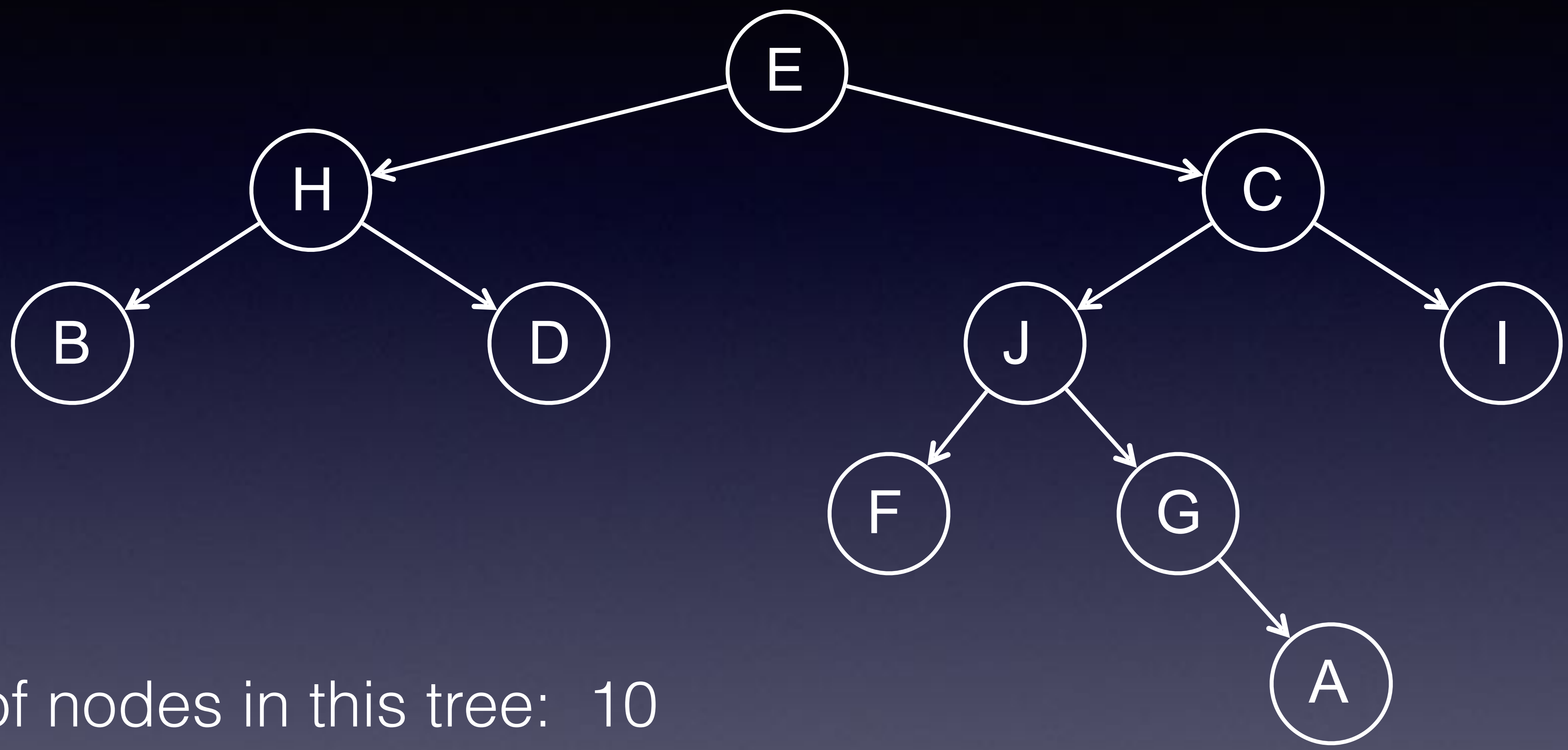
# of leaf nodes: ?

# of non-leaf nodes: ?

Level of node containing B: ? E: ? A: ?

Height of tree: ?

Tree terminology



# of nodes in this tree: 10

# of leaf nodes: 5

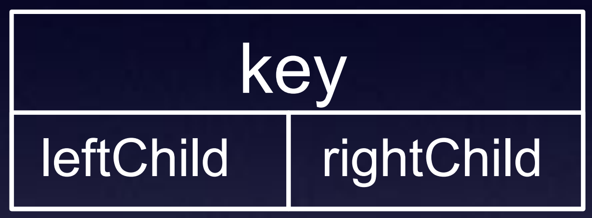
# of non-leaf nodes: 5

Level of node containing B: 2   E: 0   A: 4

Height of tree: 4

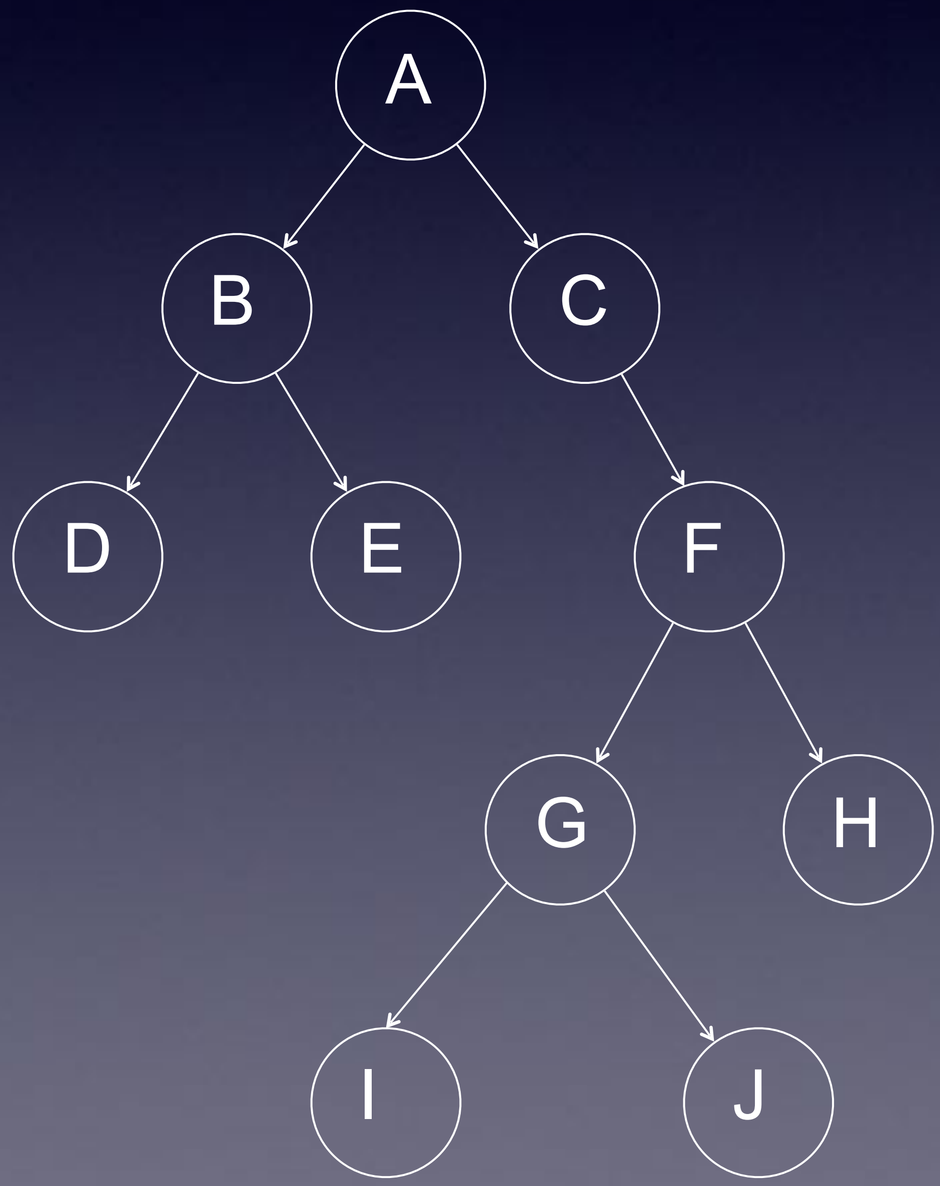
Binary Tree Node

Each binary tree node holds a key, a reference to its left subtree and a pointer to its right subtree.

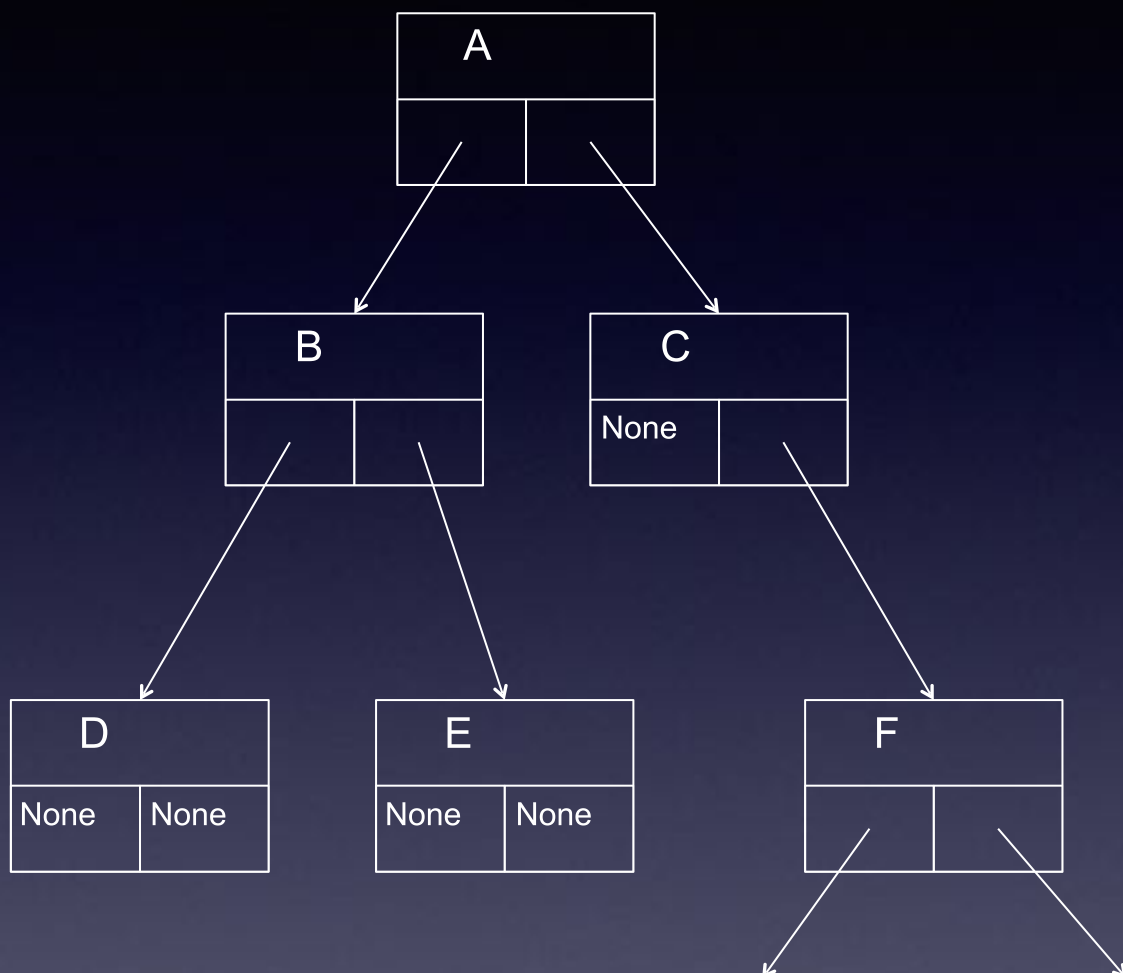


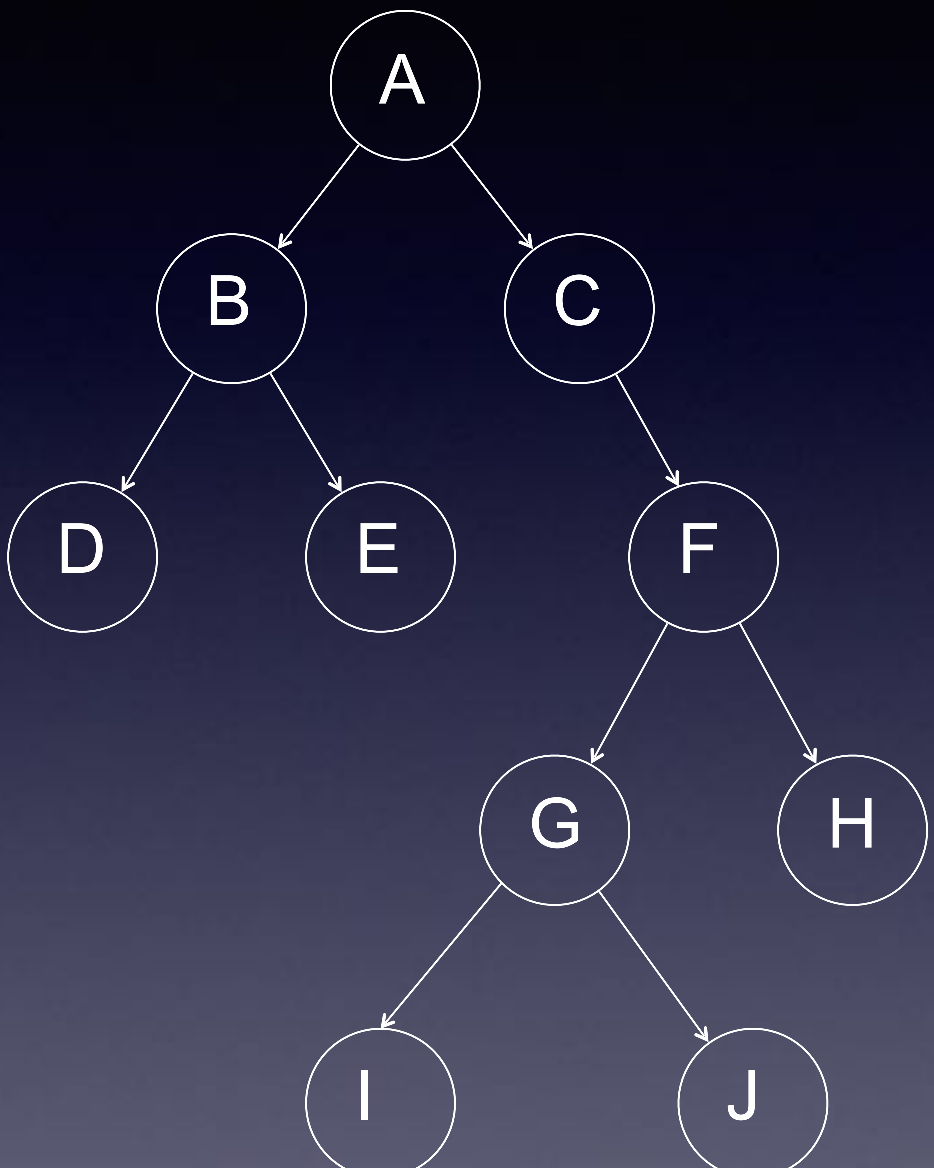
key is just some data, here the stings 'A', 'B', 'C', etc.

```
class BinaryTree:
  def __init__(self, key):
    self.key = key
    self.leftChild = None
    self.rightChild = None
```



Binary Trees





Logical representation of the BinaryTree objects in memory. Note the similarity to a linked list except there are two references from each node: leftChild and rightChild

Binary search trees*

Binary trees are cool, but for searching, we are really interested in is a class of binary trees called binary search trees.

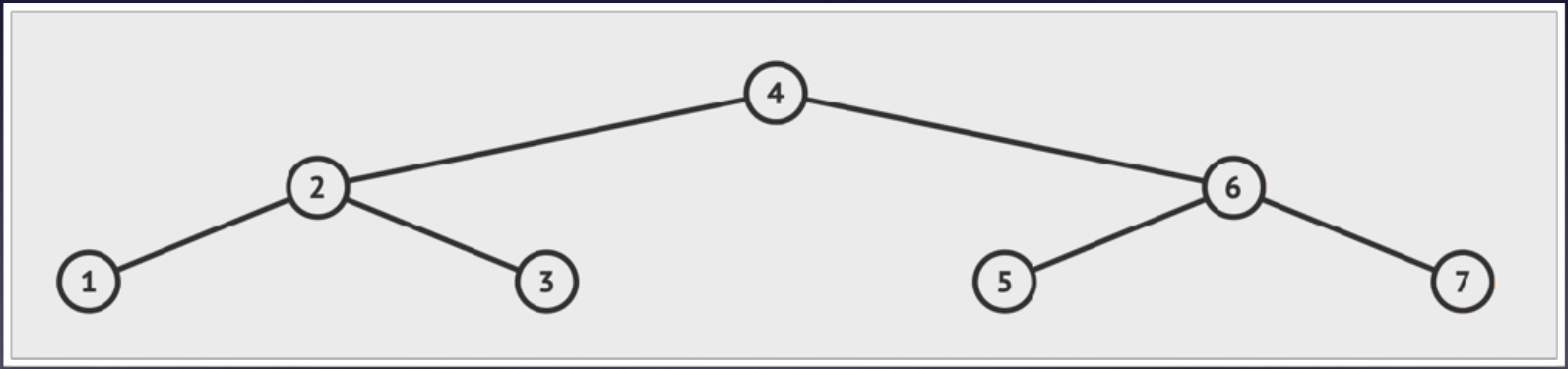
A **binary search tree** is a binary tree in which every node is

- **empty** or
- the root of a binary tree in which all the **values in the left subtree are less than the value at the root**, and all the **values in the right subtree are greater than the value at the root**.

* We are deviating from the order in which things are presented in chapter 6.

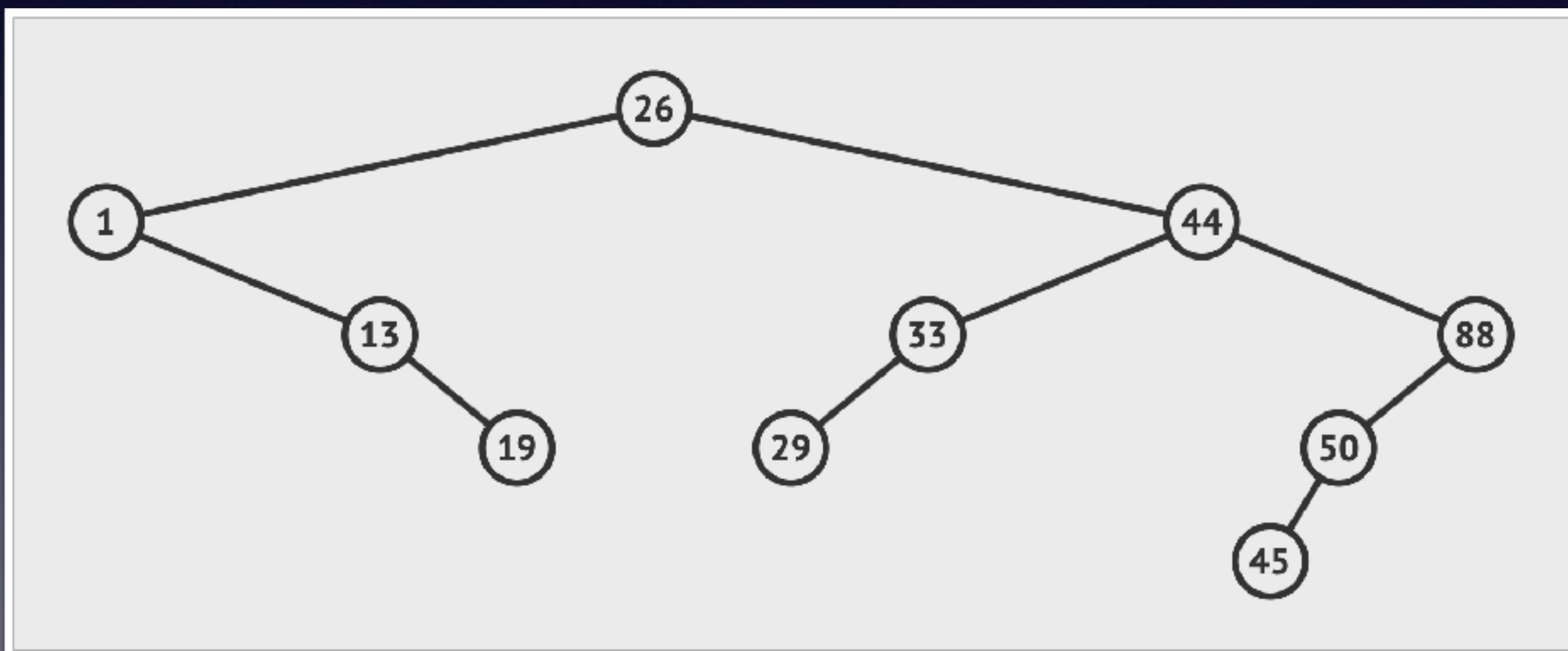
Binary search trees

This is a binary search tree:



Binary search trees

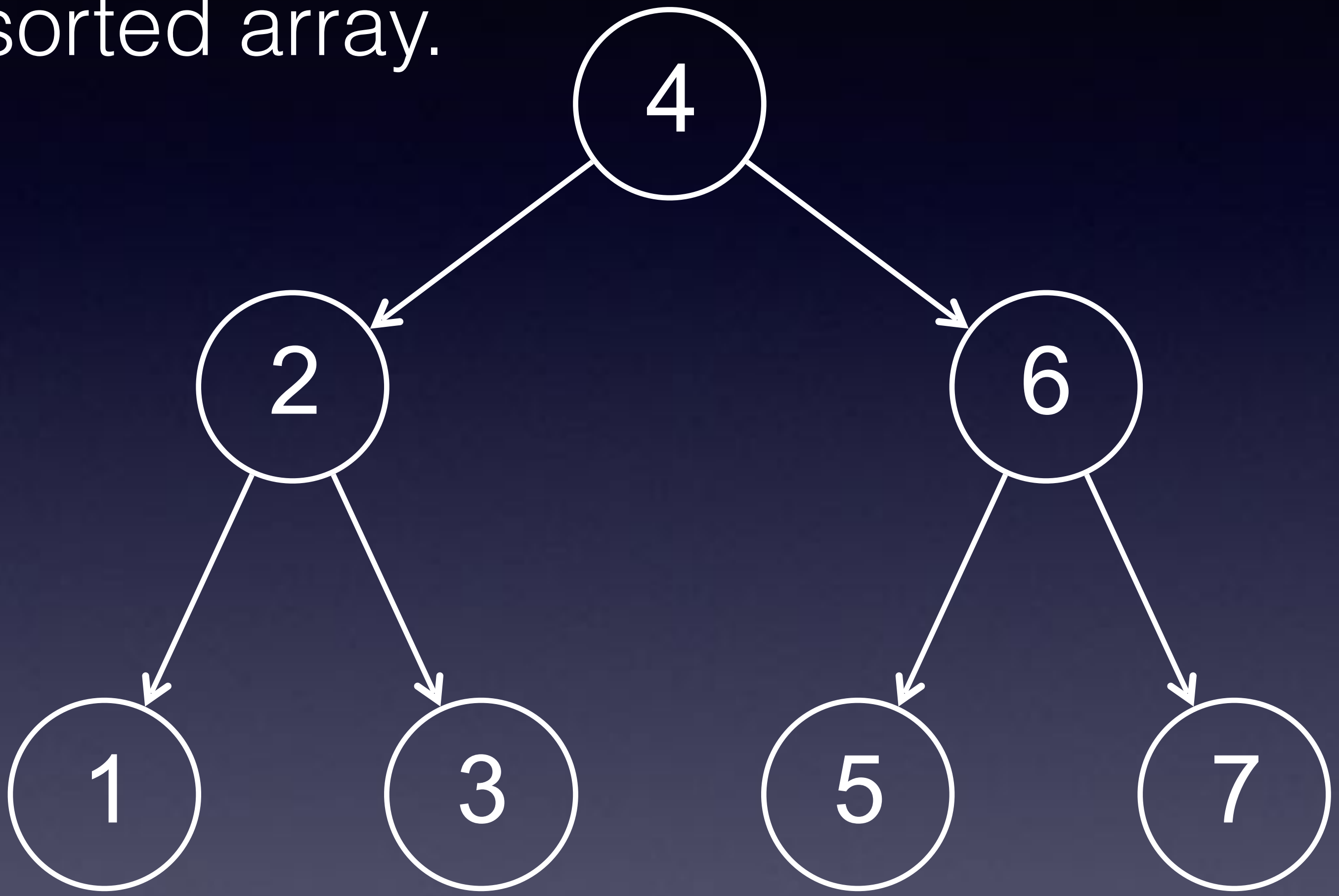
This is also a binary search tree



Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```

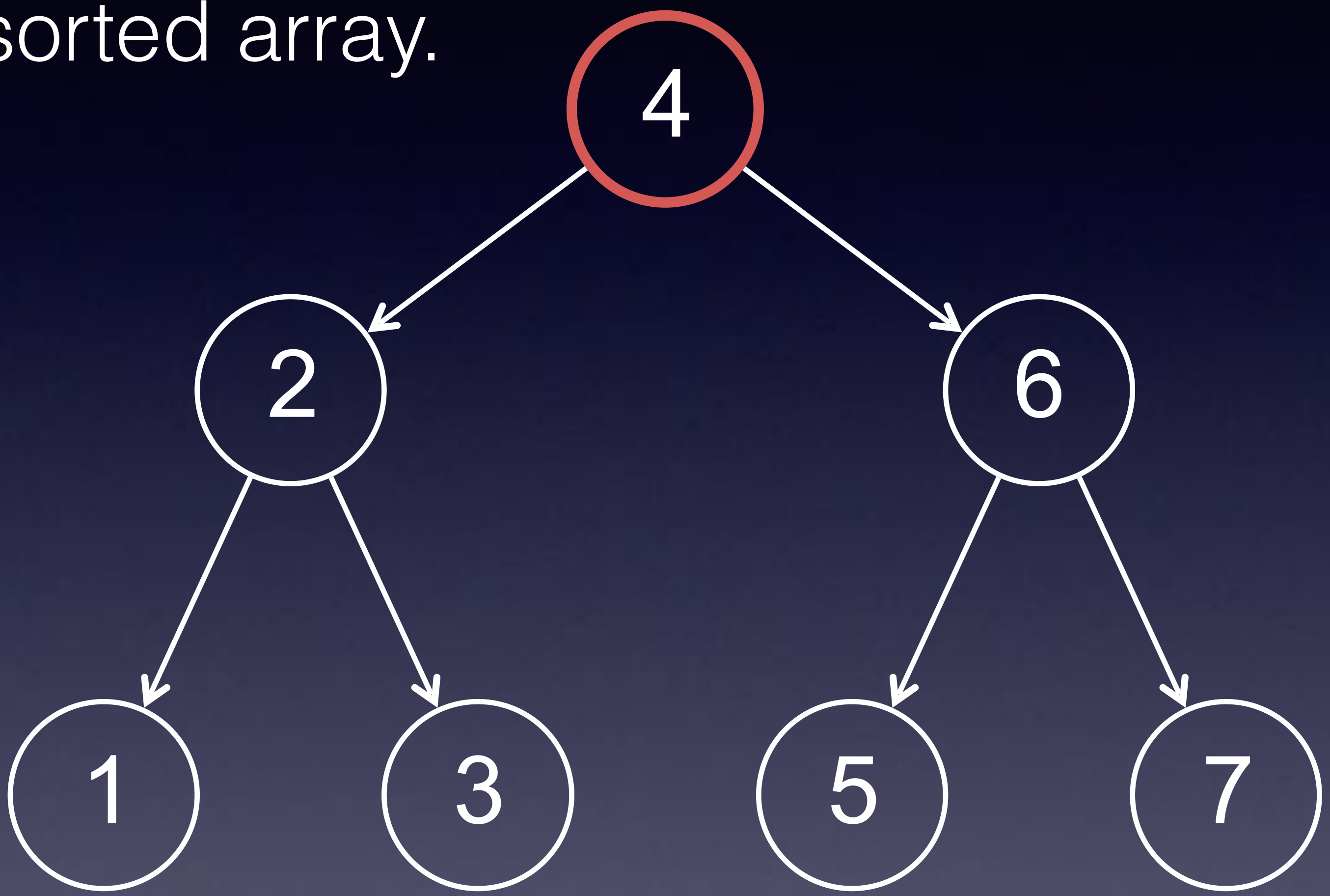


Search for 3.

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```

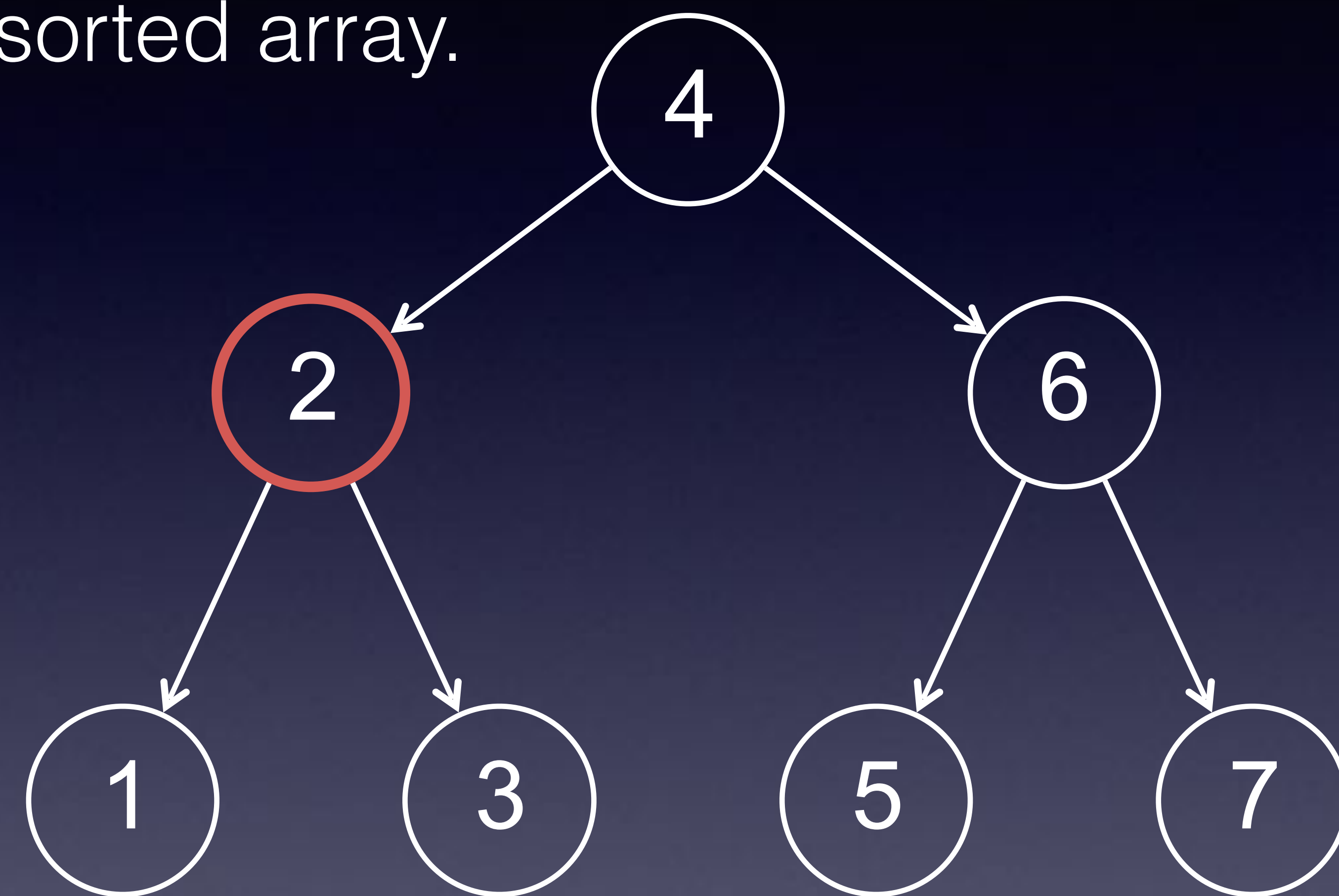


Search for 3.
Is it here? No. 3 is less than 4

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```



Search for 3.

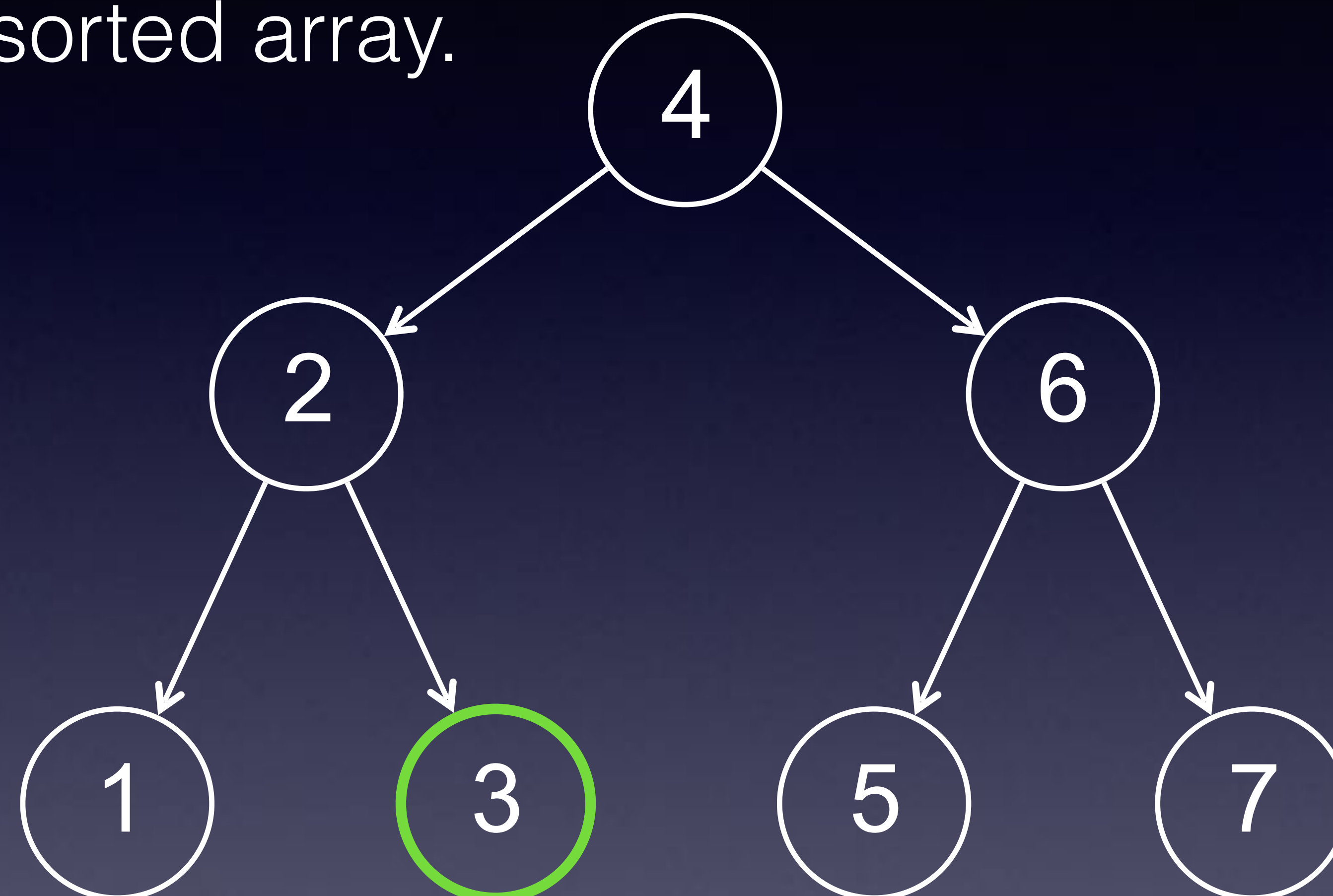
Is it here? No. 3 is less than 4

Is it here? No. 3 is greater than 2

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```



Search for 3.

Is it here? No. 3 is less than 4

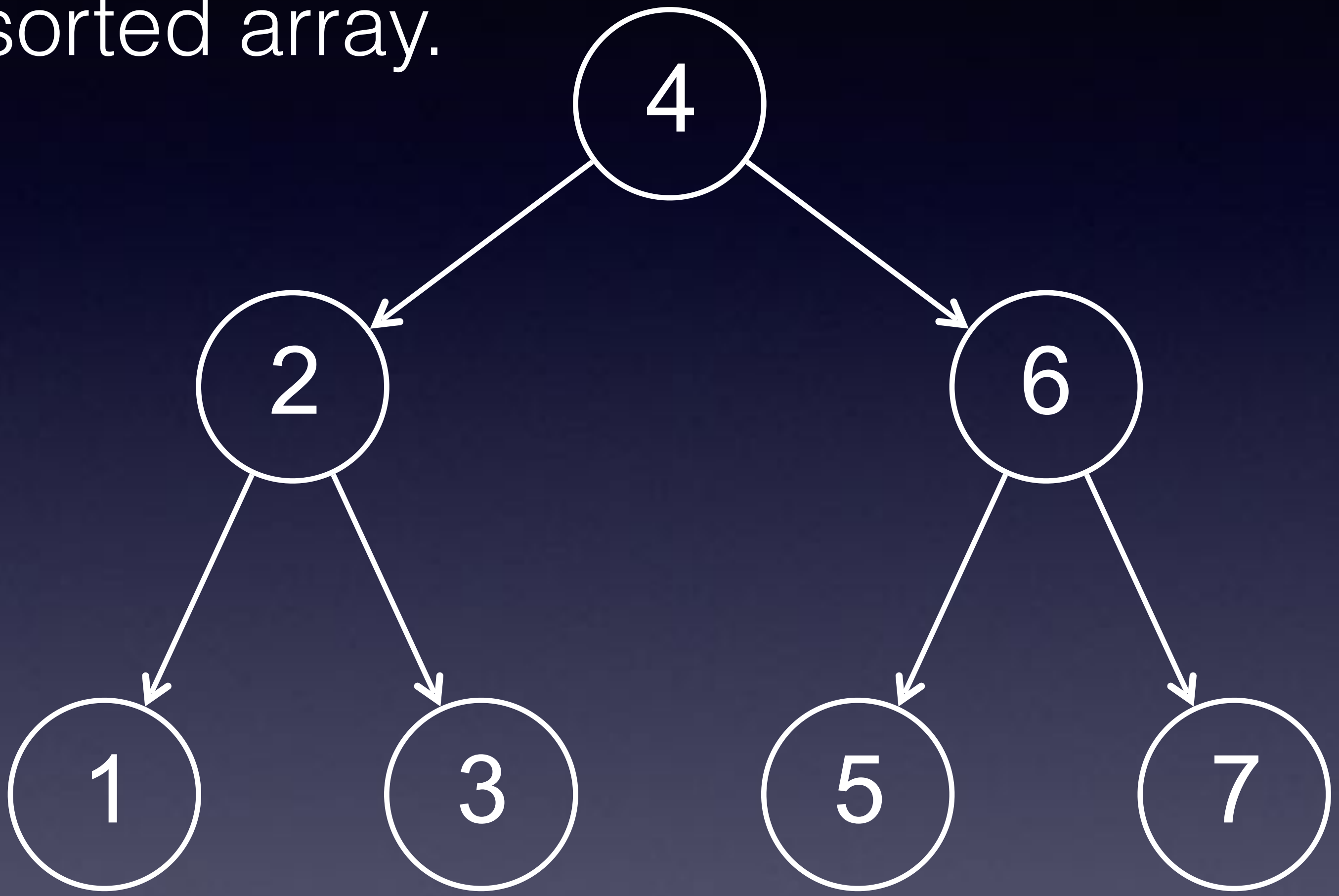
Is it here? No. 3 is greater than 2

Is it here? Yes. 3 is equal to 3

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```

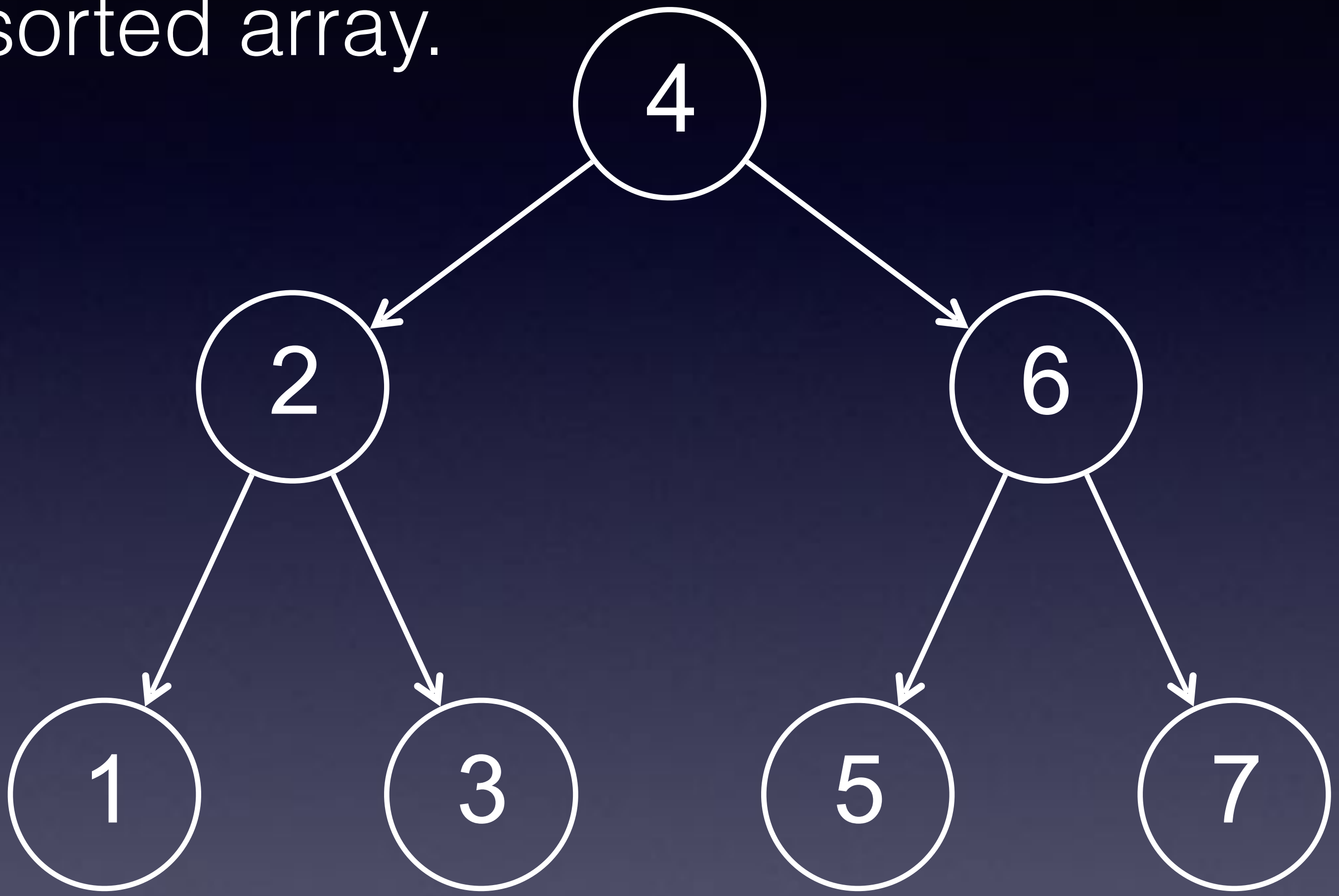


What's your guess as to time complexity of finding a target in this BST?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure
else if
  the target value == the
  value at the root node
  then return success
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree
else
  return the result of
  search the right subtree
```

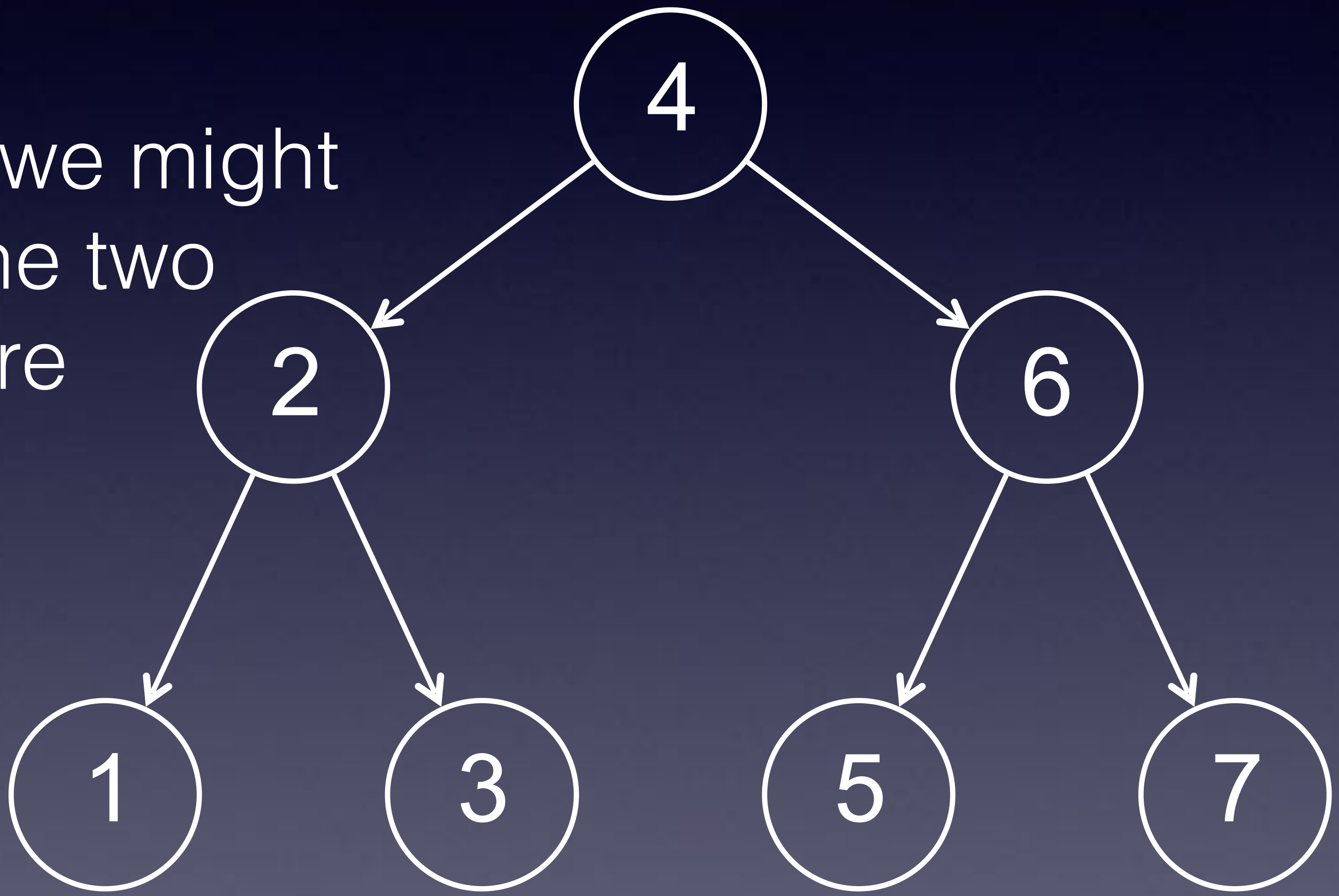


What's your guess as to time complexity of finding a target in this BST?

O(log n) is a pretty good guess. Why?

Binary search trees

When we add a ‘**find**’ operation to our BST data structure, we have a new abstract data type. Simpler than the one in the book.



There are other operations that we might want to use with this ADT, but the two we really want to know about are

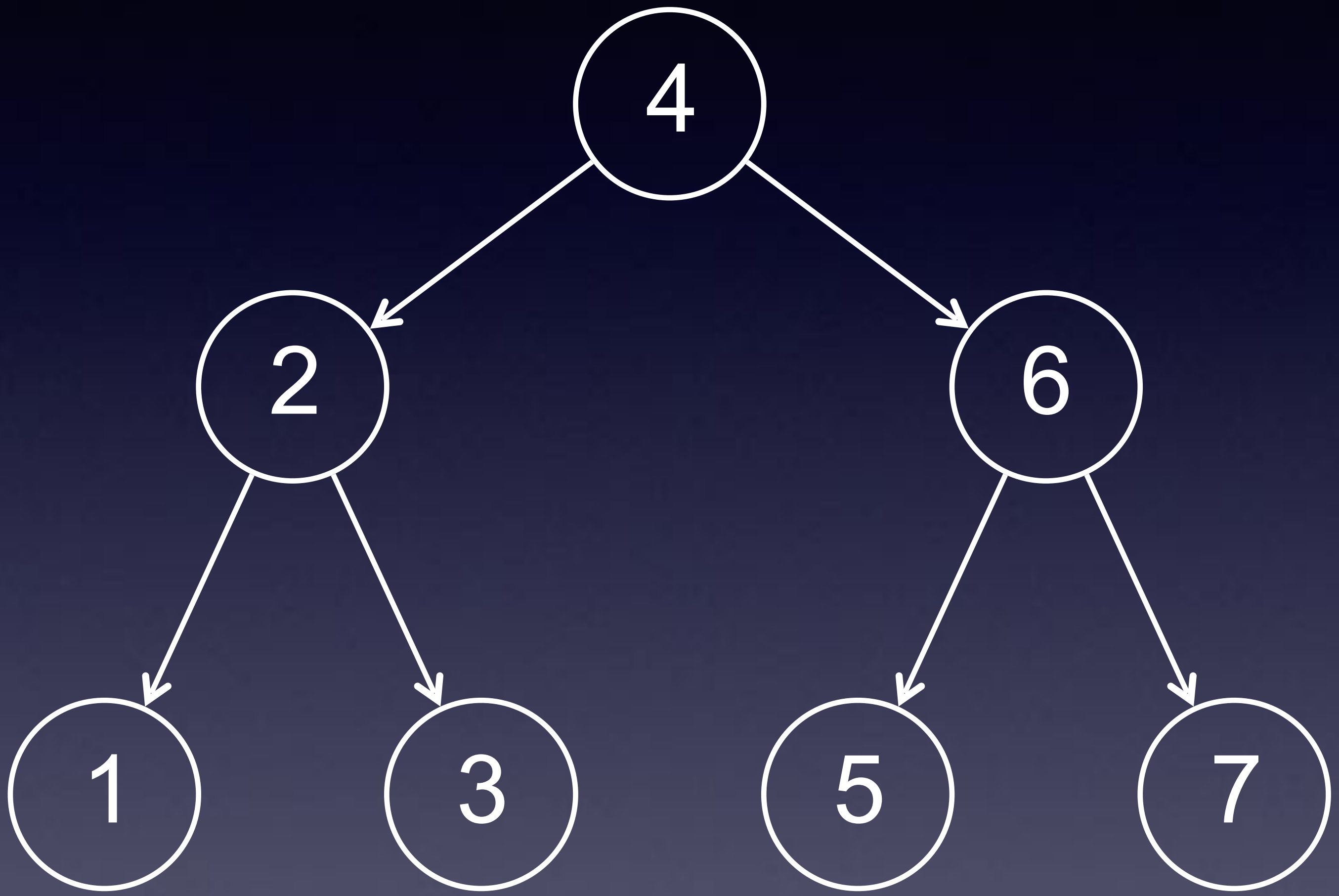
‘**insert**’, because that’s what started this conversation.

‘**delete**’, because if we’re going to insert things, we also want to delete things.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

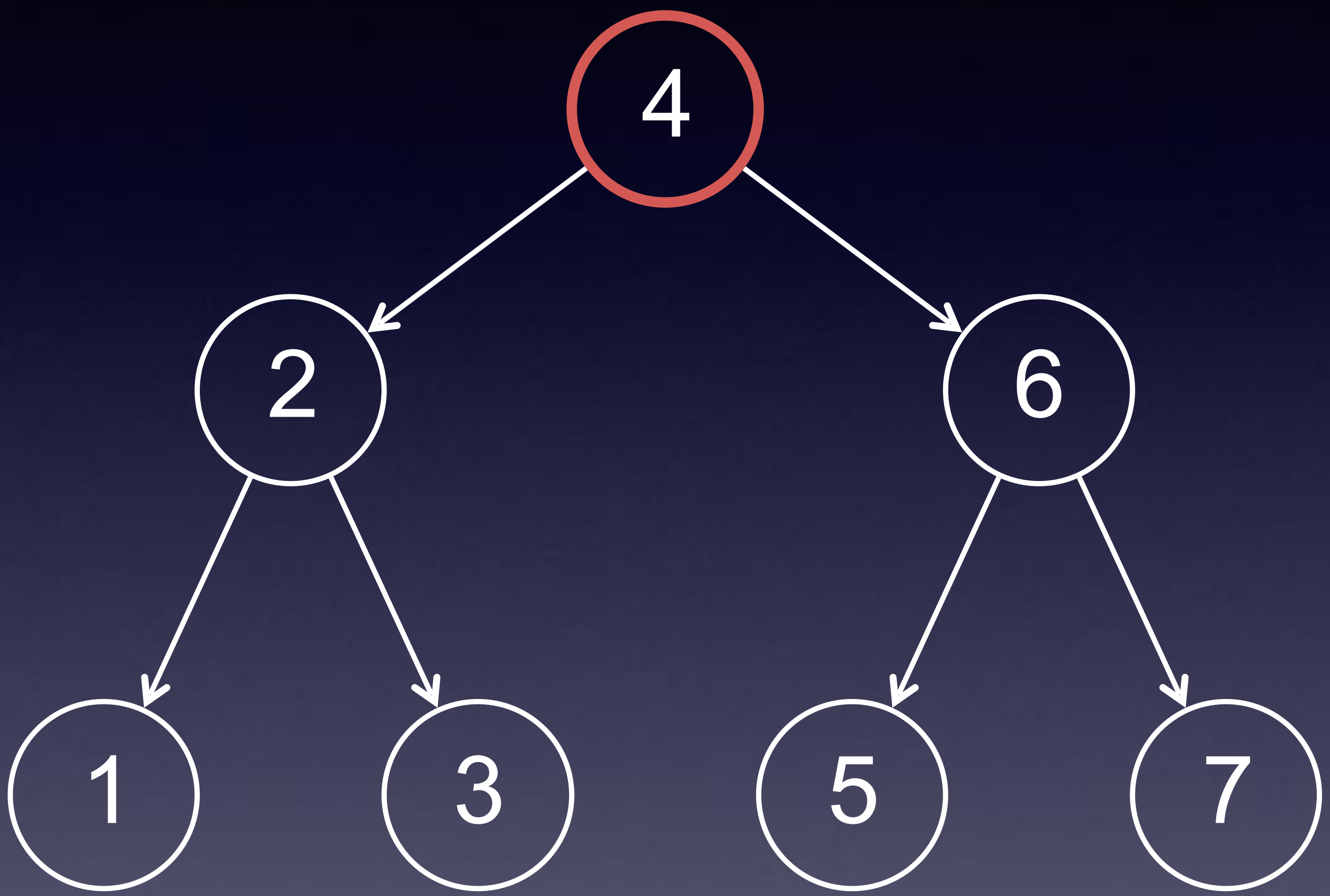


Notice our insert is **recursive** and it does not allow duplicates

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

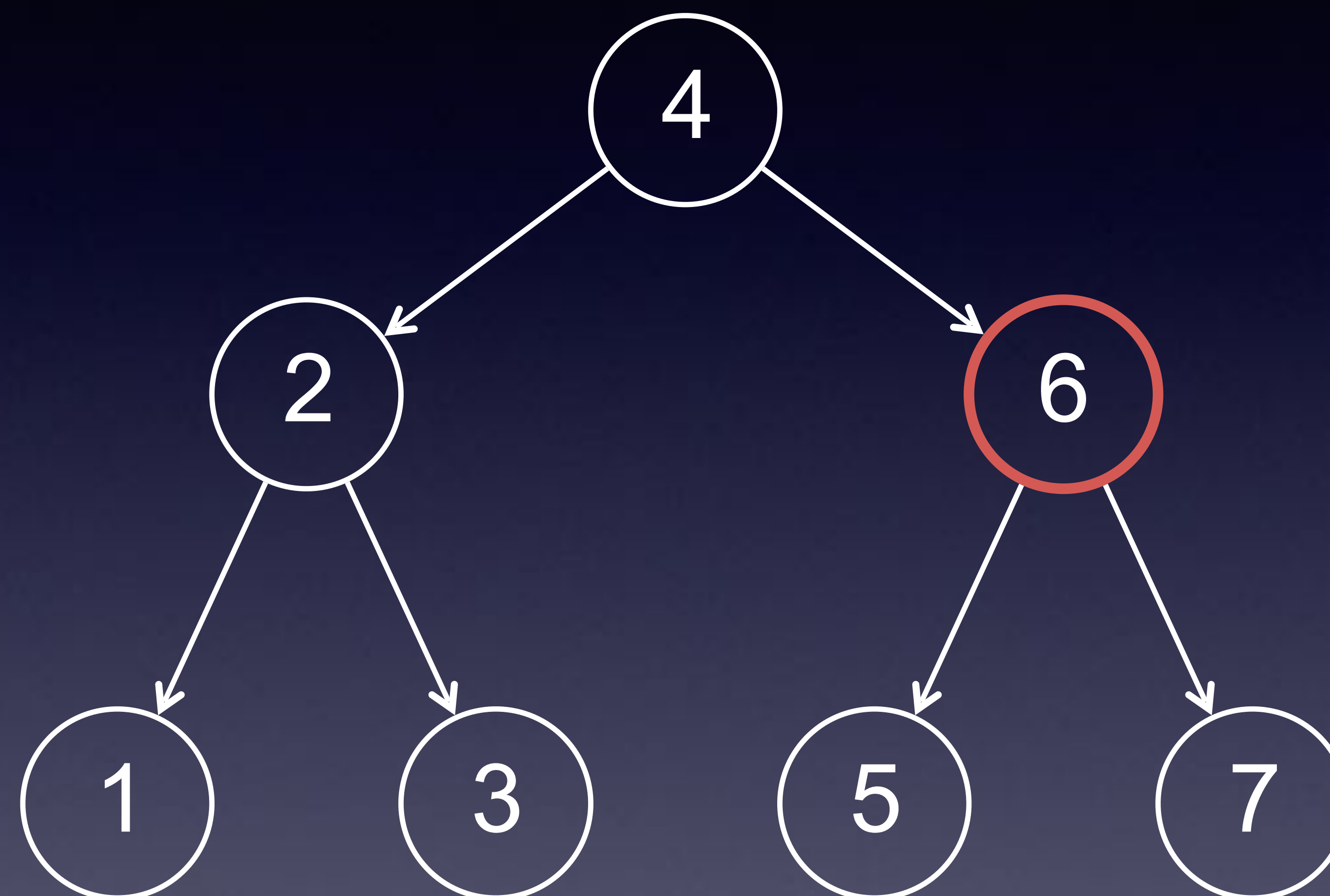


Insert 8.
Is this the place? No. 8 is greater than 4

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



Insert 8.

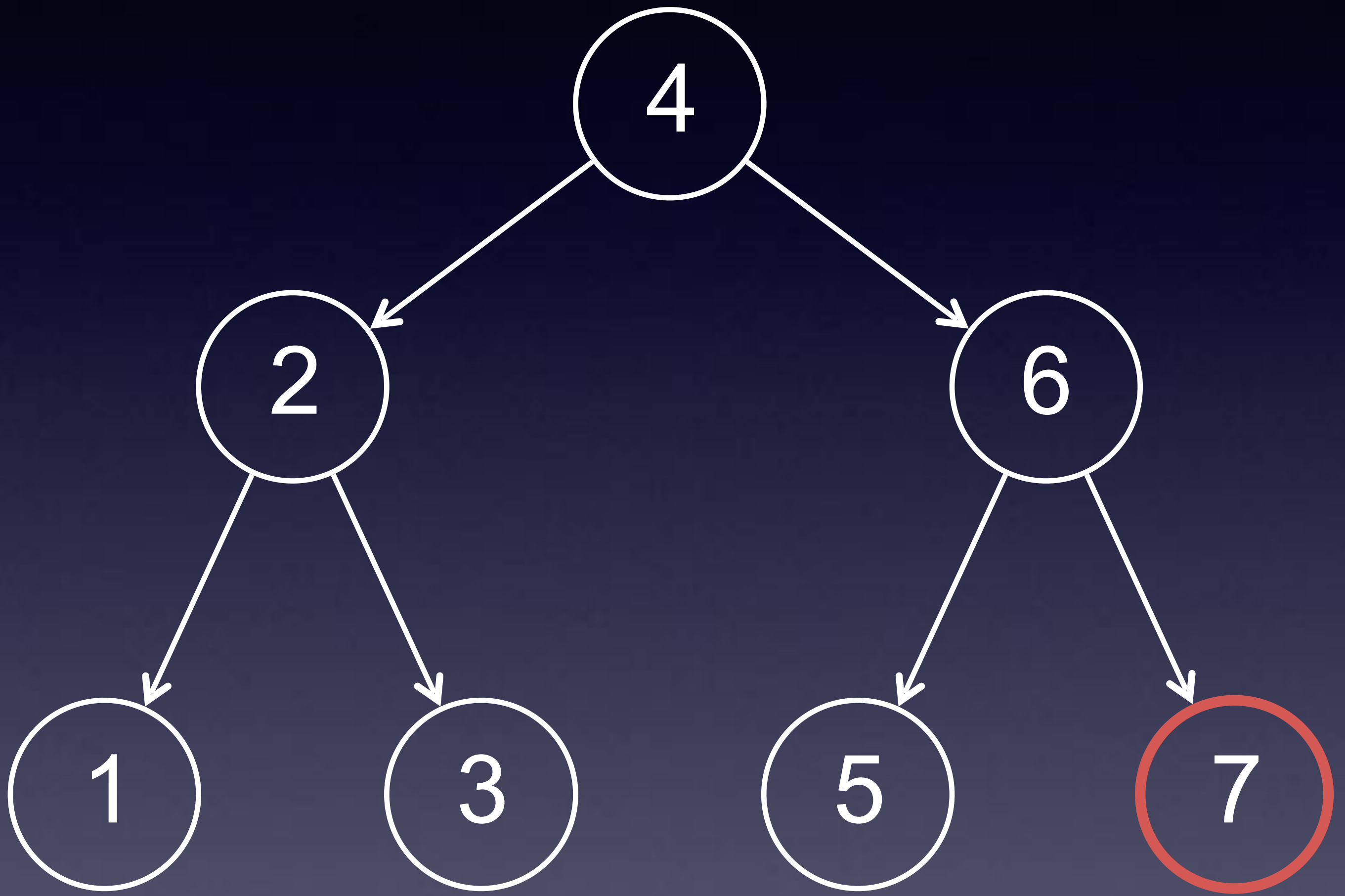
Is this the place? No. 8 is greater than 4

Is this the place? No. 8 is greater than 6

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



Insert 8.

Is this the place? No. 8 is greater than 4

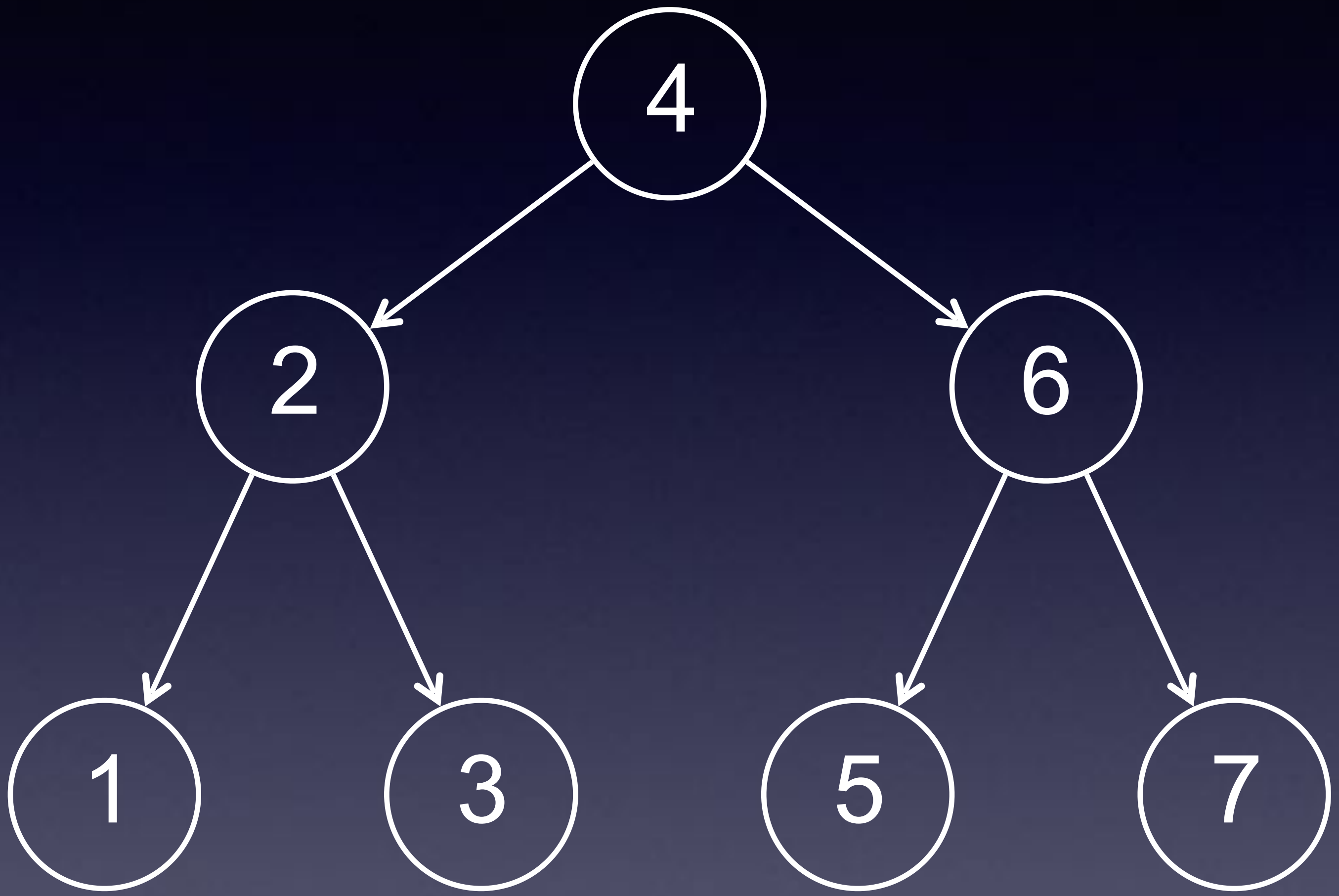
Is this the place? No. 8 is greater than 6

Is this the place? No. 8 is greater than 7

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



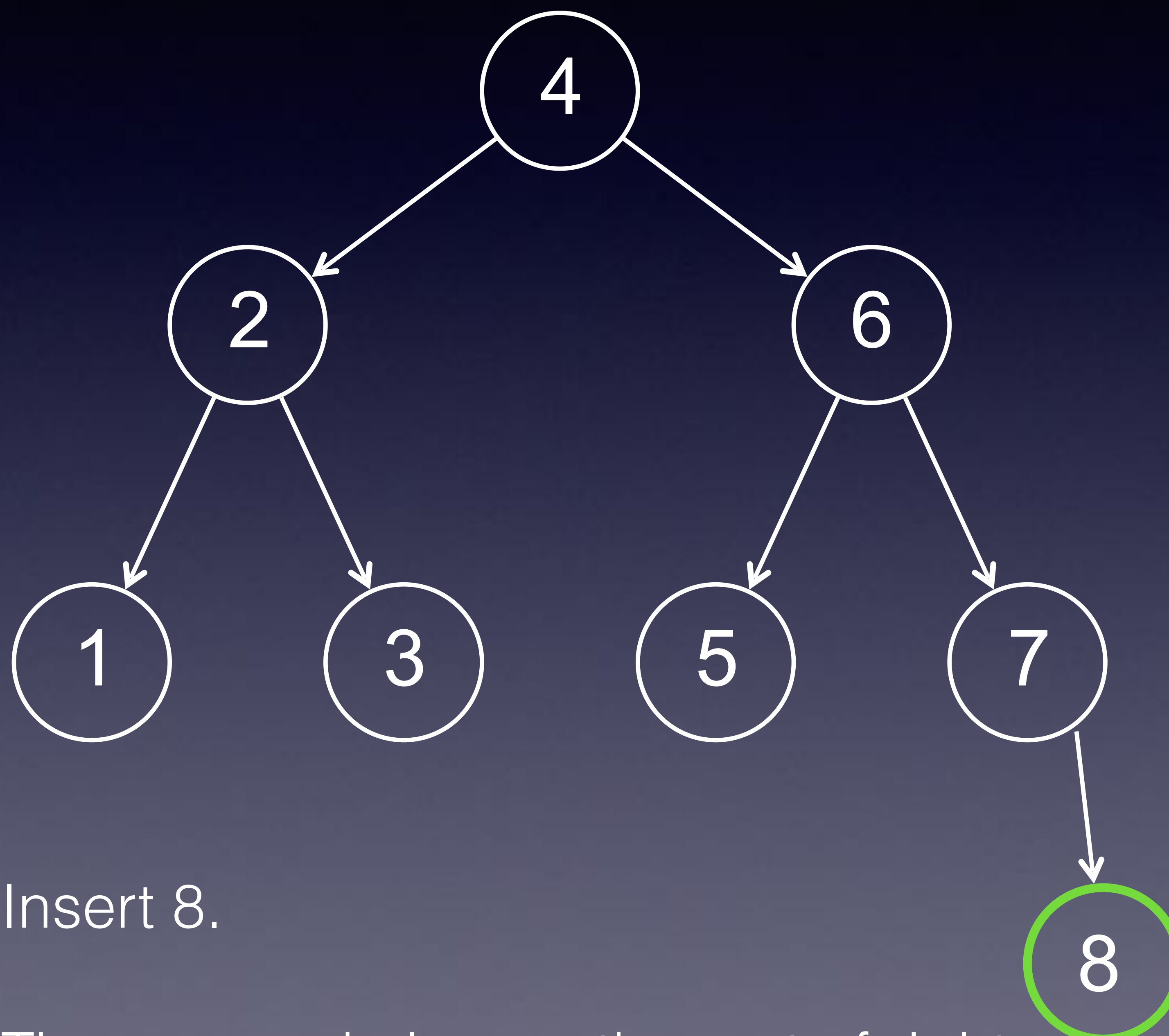
Insert 8.

But the right subtree of 7 is empty, so we create a new node which will hold 8

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



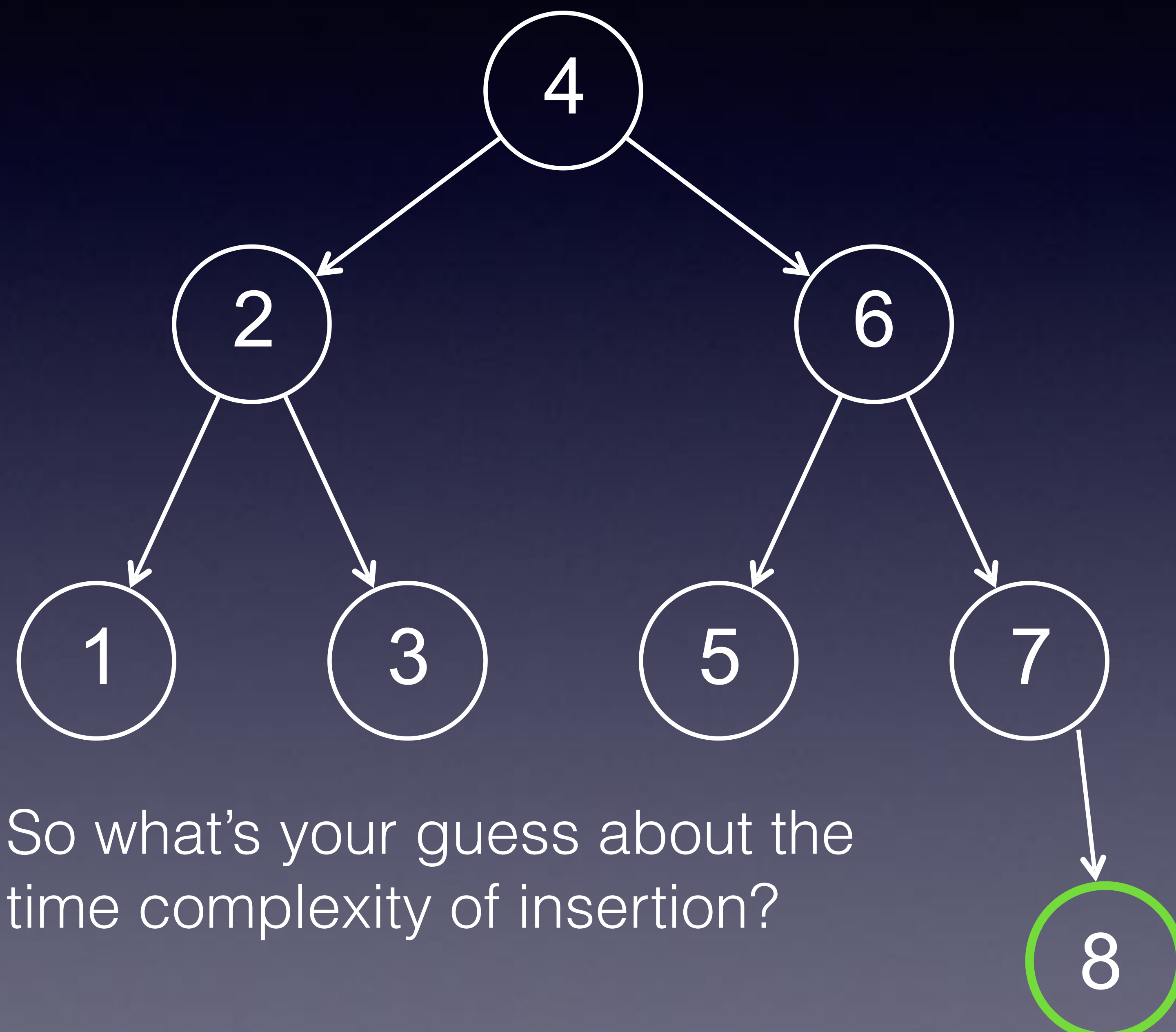
Insert 8.

The new node is now the root of right subtree of 7.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

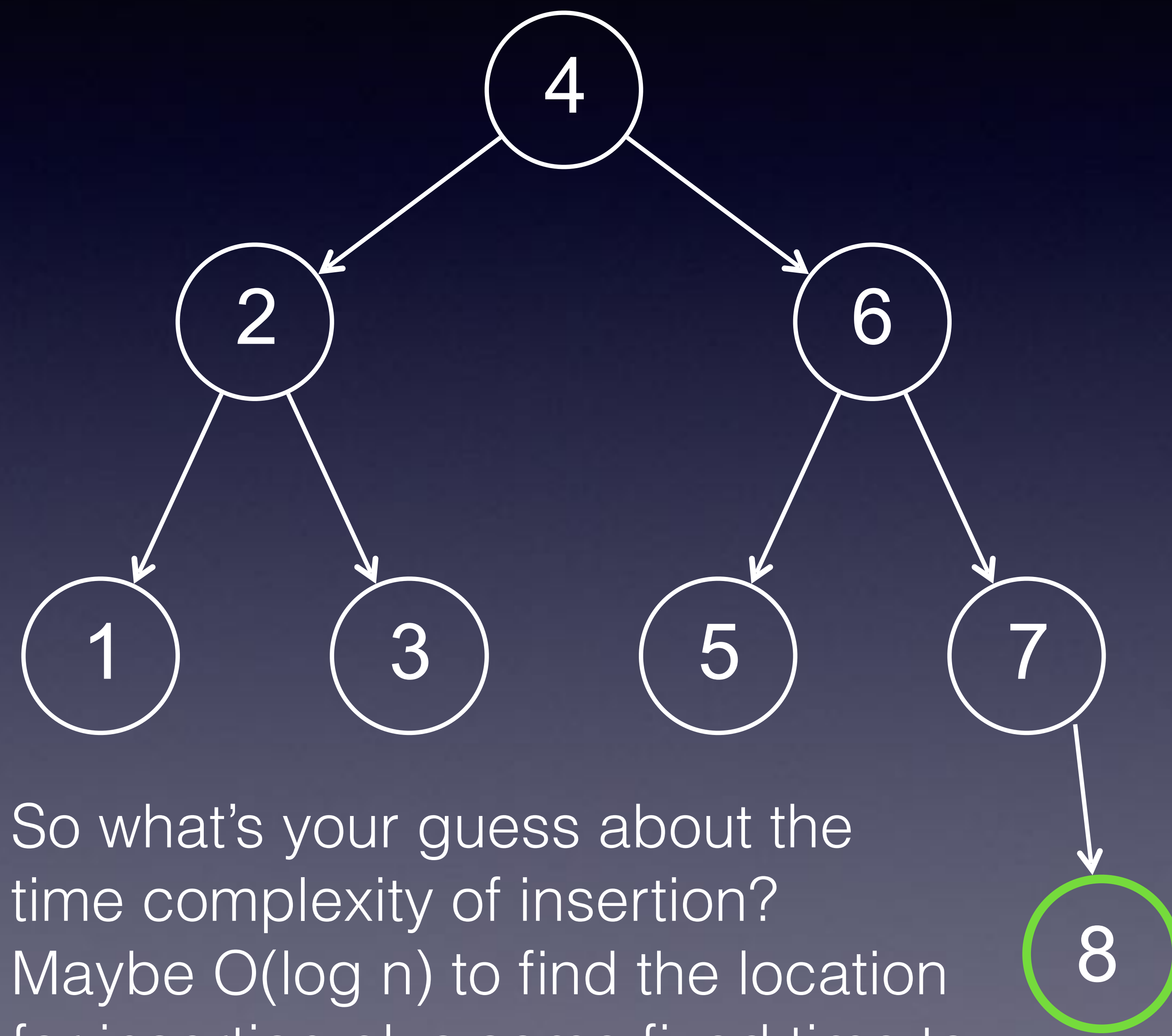


So what's your guess about the time complexity of insertion?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



So what's your guess about the time complexity of insertion? Maybe O(log n) to find the location for insertion plus some fixed time to create the new node and add the link.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

How do you build a binary search tree from scratch?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

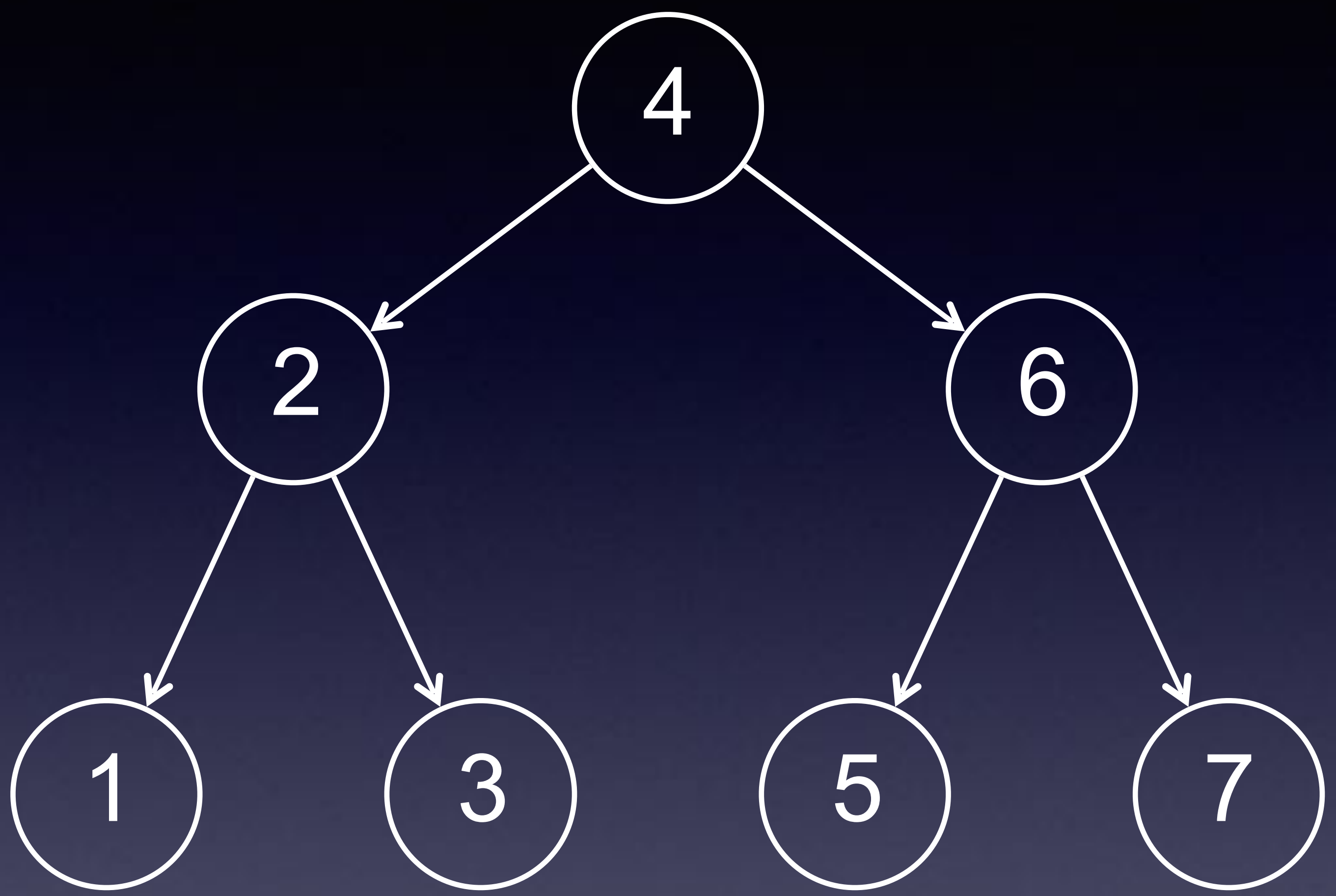
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
4 6 2 5 3 1 7 in that order? Try here

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```



Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
4 6 2 5 3 1 7 in that order?

Binary search trees

insert(target) works like this:

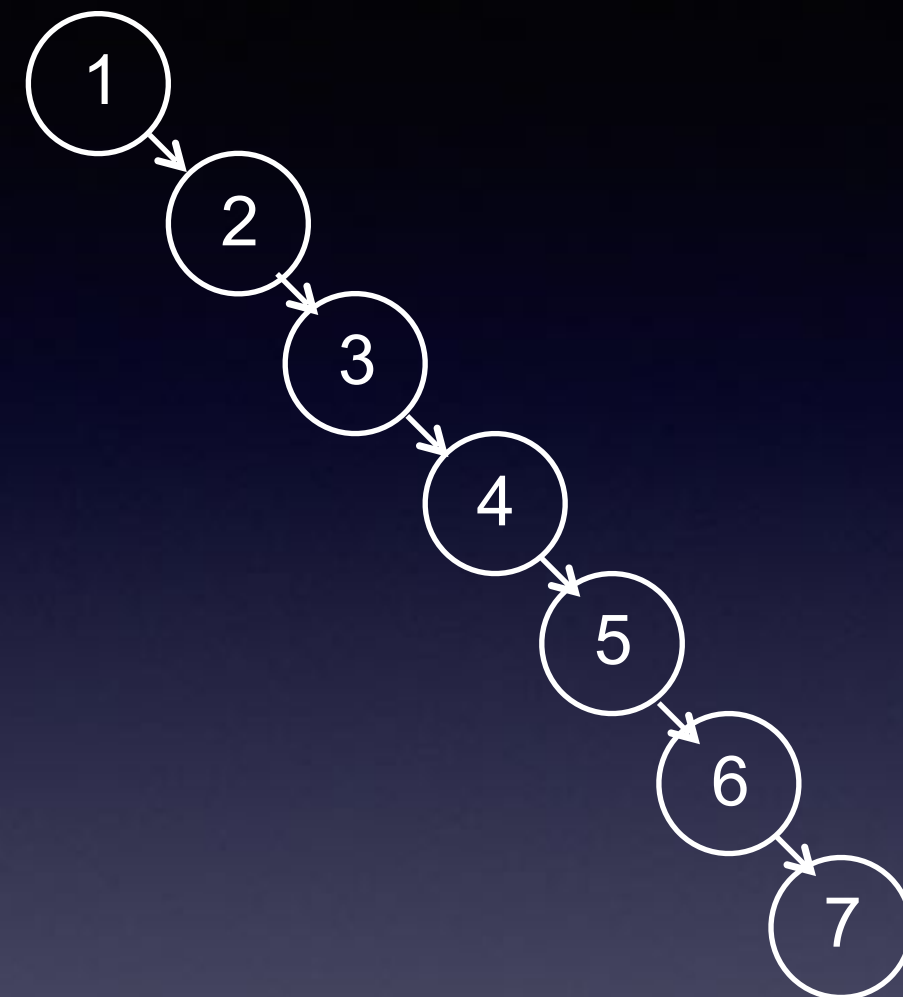
```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert 1 2 3 4 5 6 7 in that order? Try here

Binary search trees

insert(target) works like this:



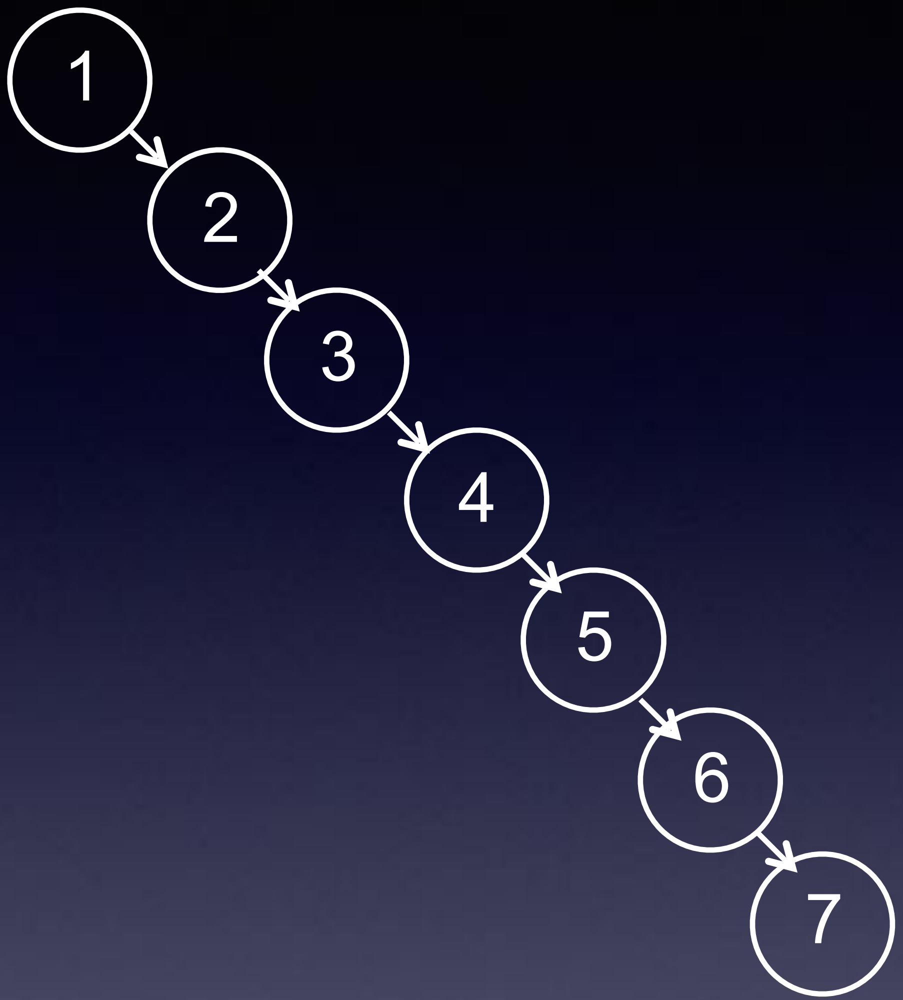

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert 1 2 3 4 5 6 7 in that order?

Binary search trees

insert(target) works like this:




```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success
else if
  the target value == the
  value at the root then
  the target is already
  in the BST, return failure
else if
  the target < the value at
  the root then call insert
  on the left subtree
else
  call insert on the right
  subtree
```

Do you see the problem here?

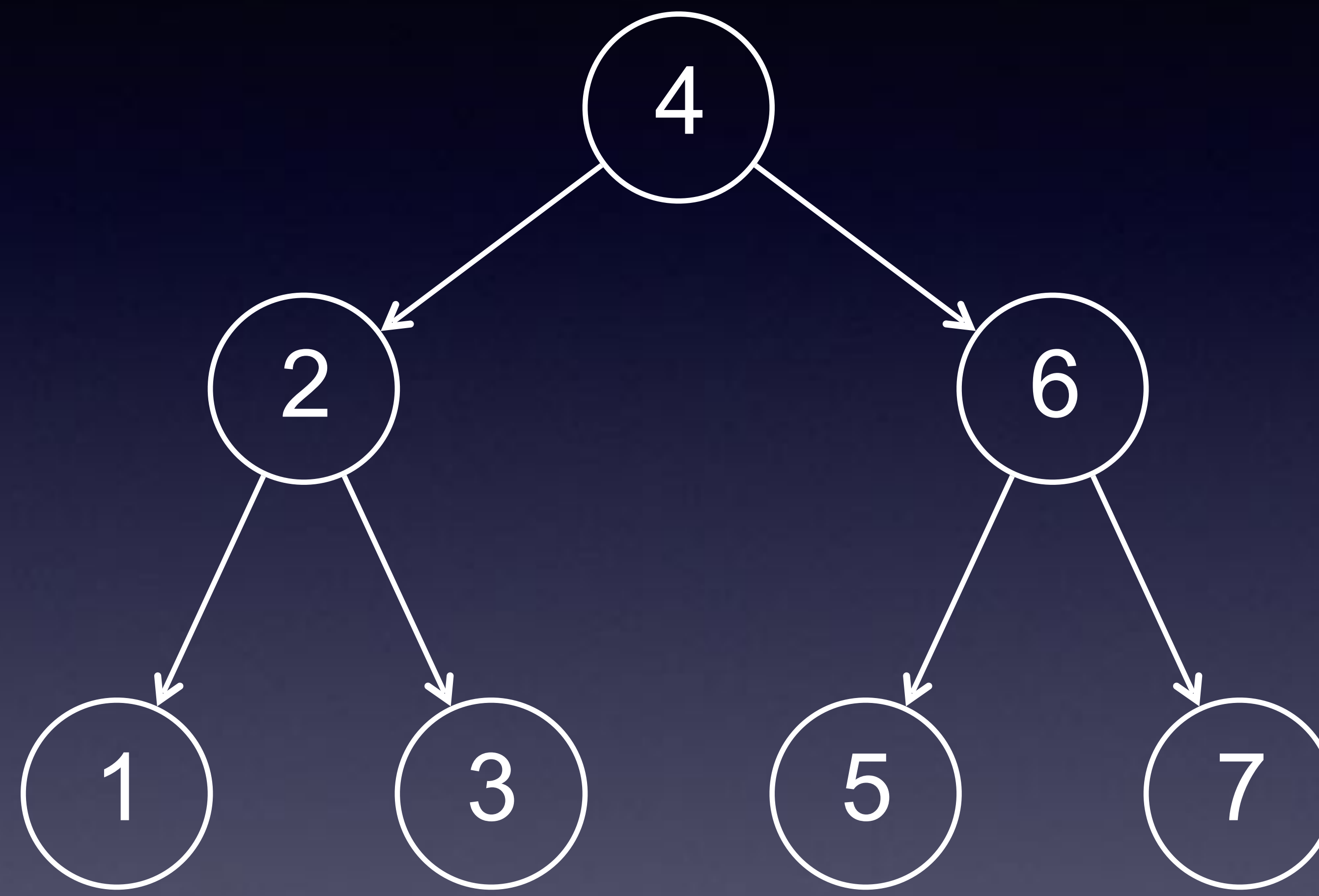
Is this even a binary search tree?

What's the complexity of find and insert?

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null
```

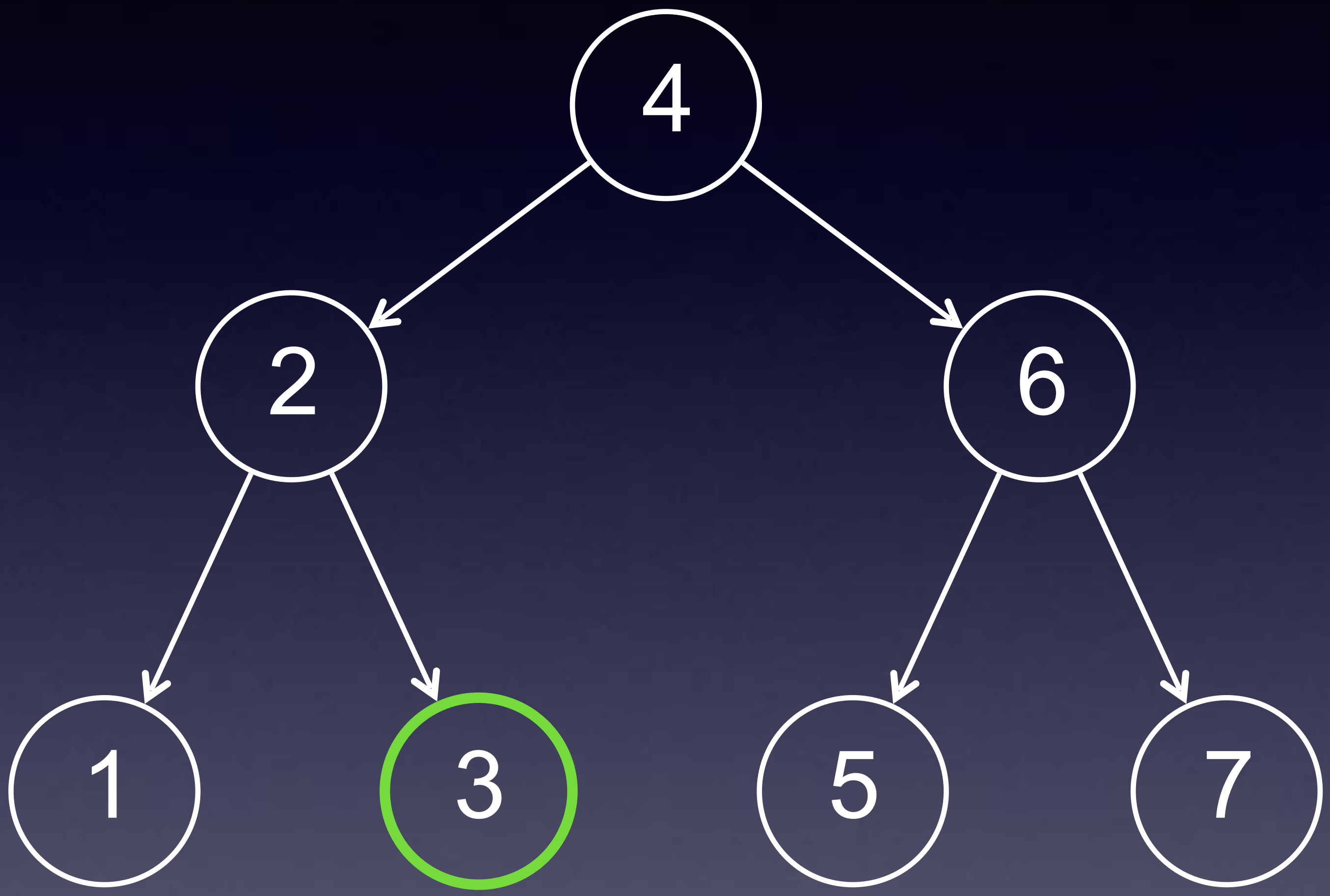


Case 1: The node to be deleted is a leaf node

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null
```

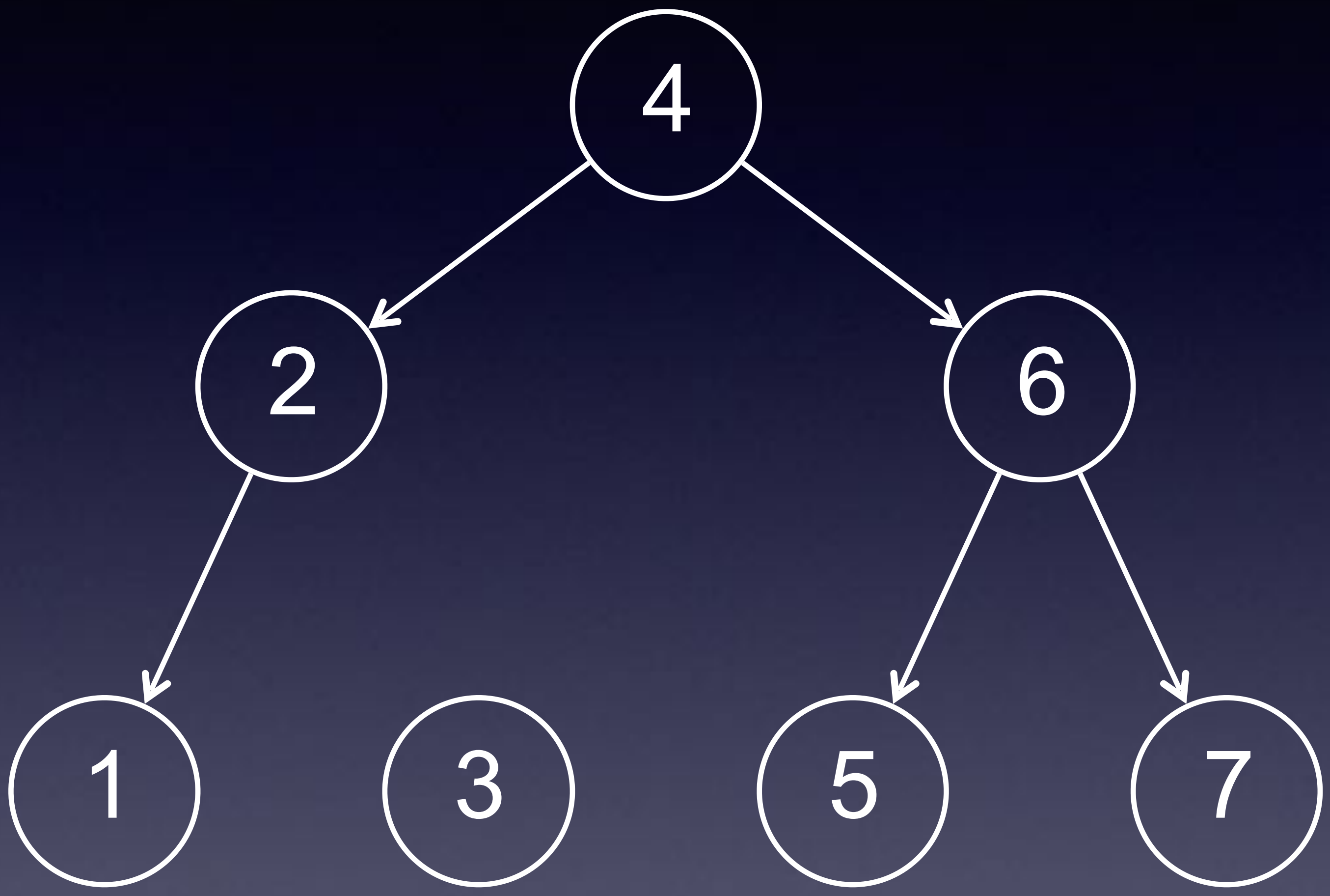


Delete 3

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null
```

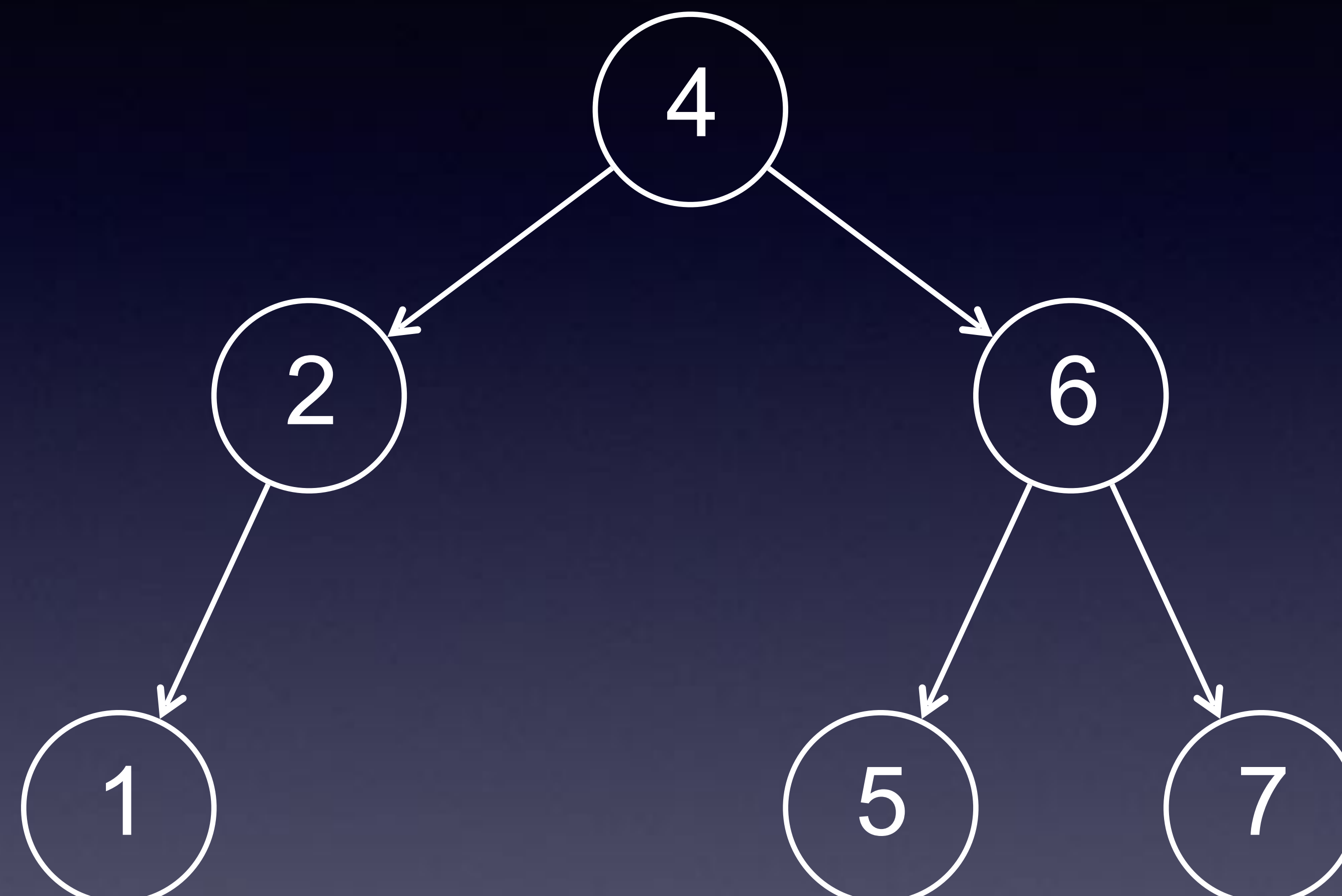


Delete 3

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null
```

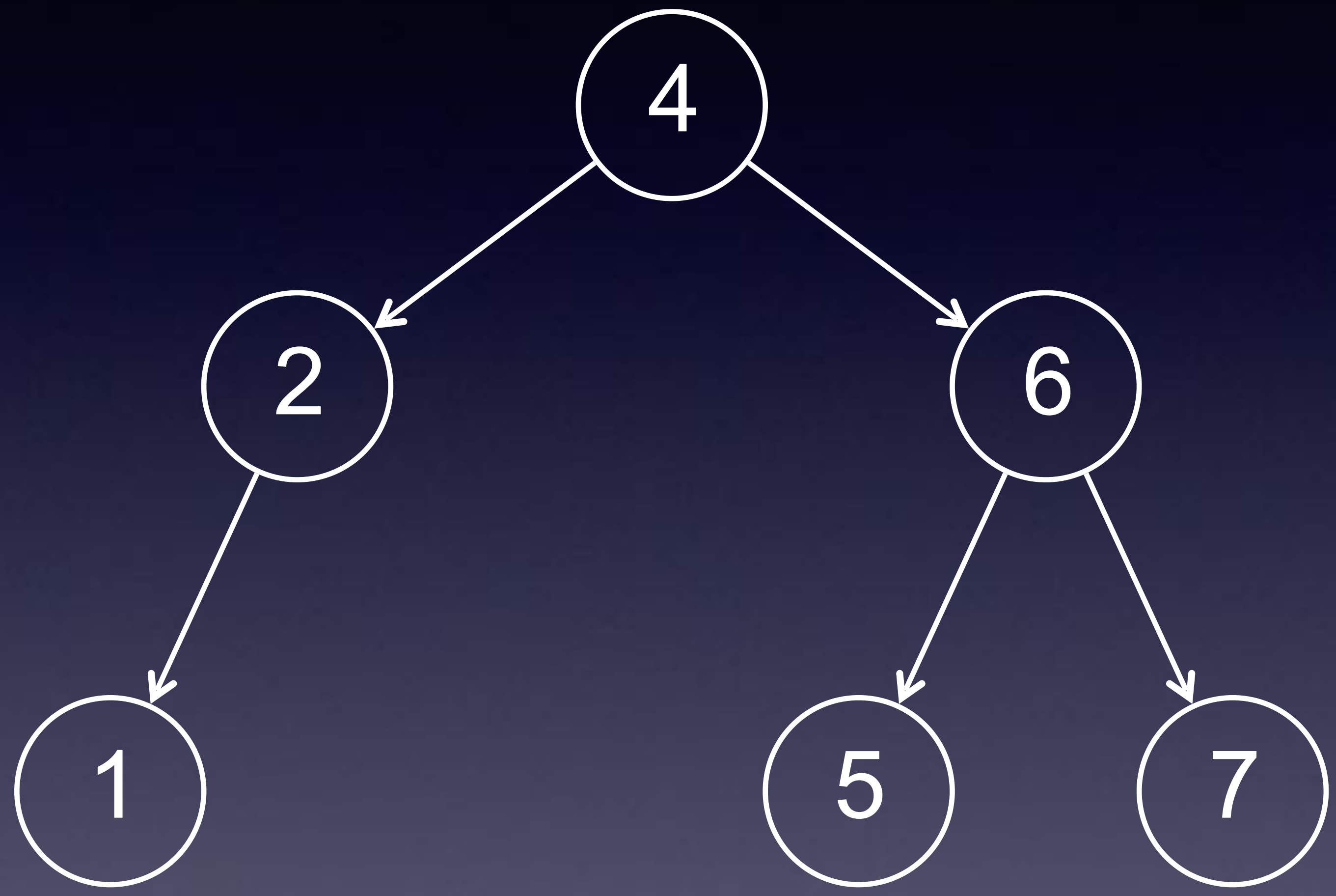


Delete 3

Binary search trees

delete(target) is more complicated. Let’s break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child
```

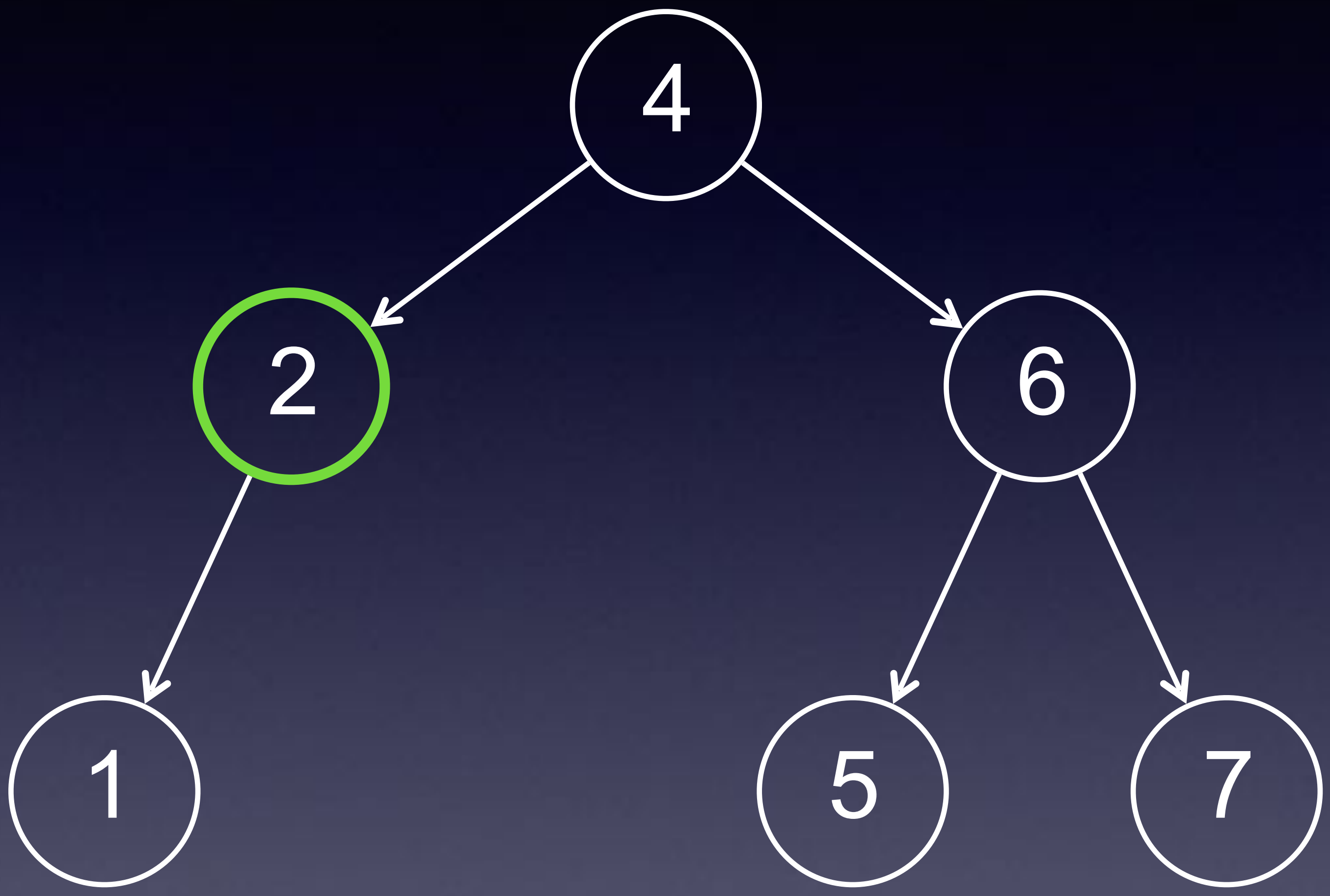


Case 2: The node to be deleted has one child

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child
```

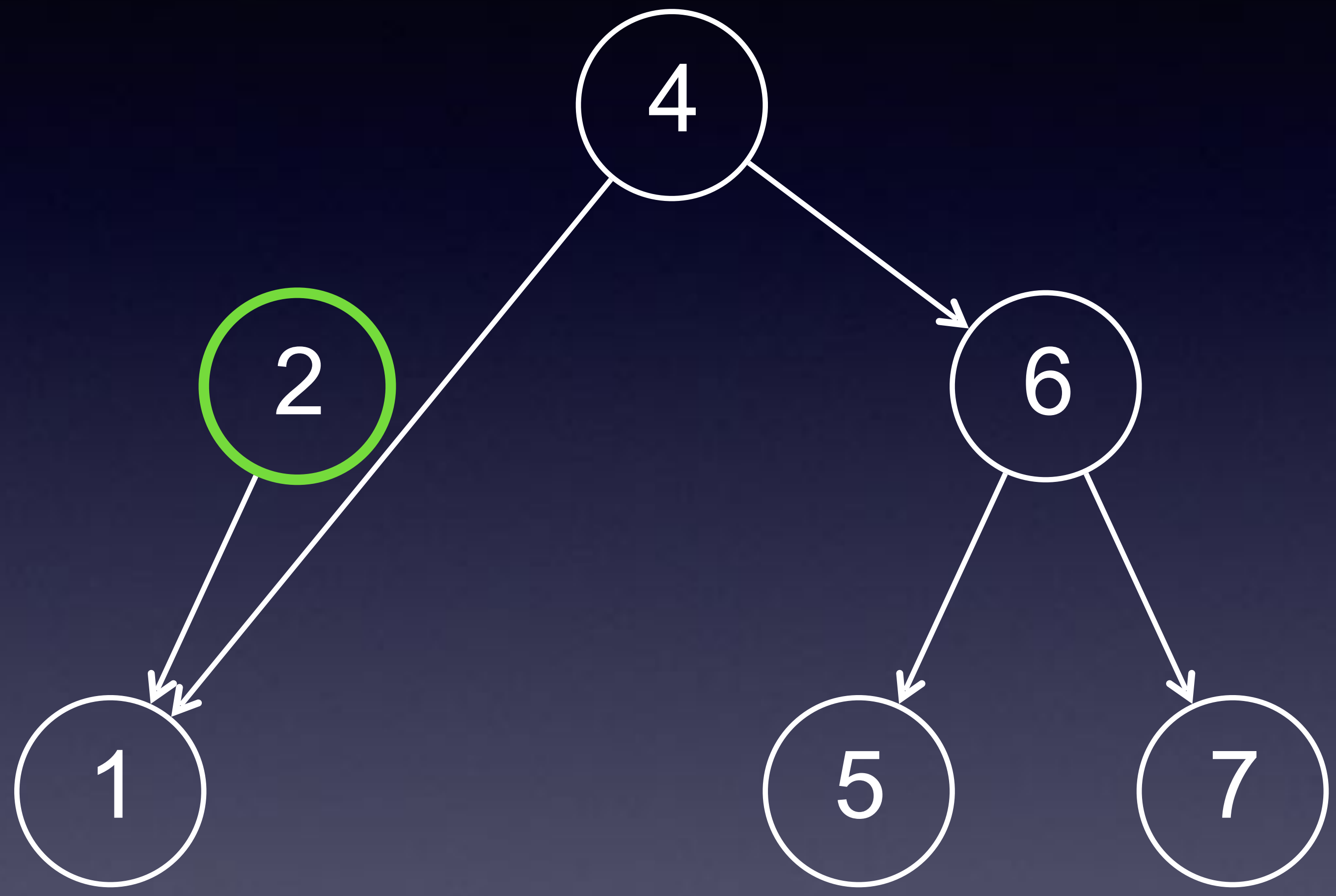


Delete 2

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child
```

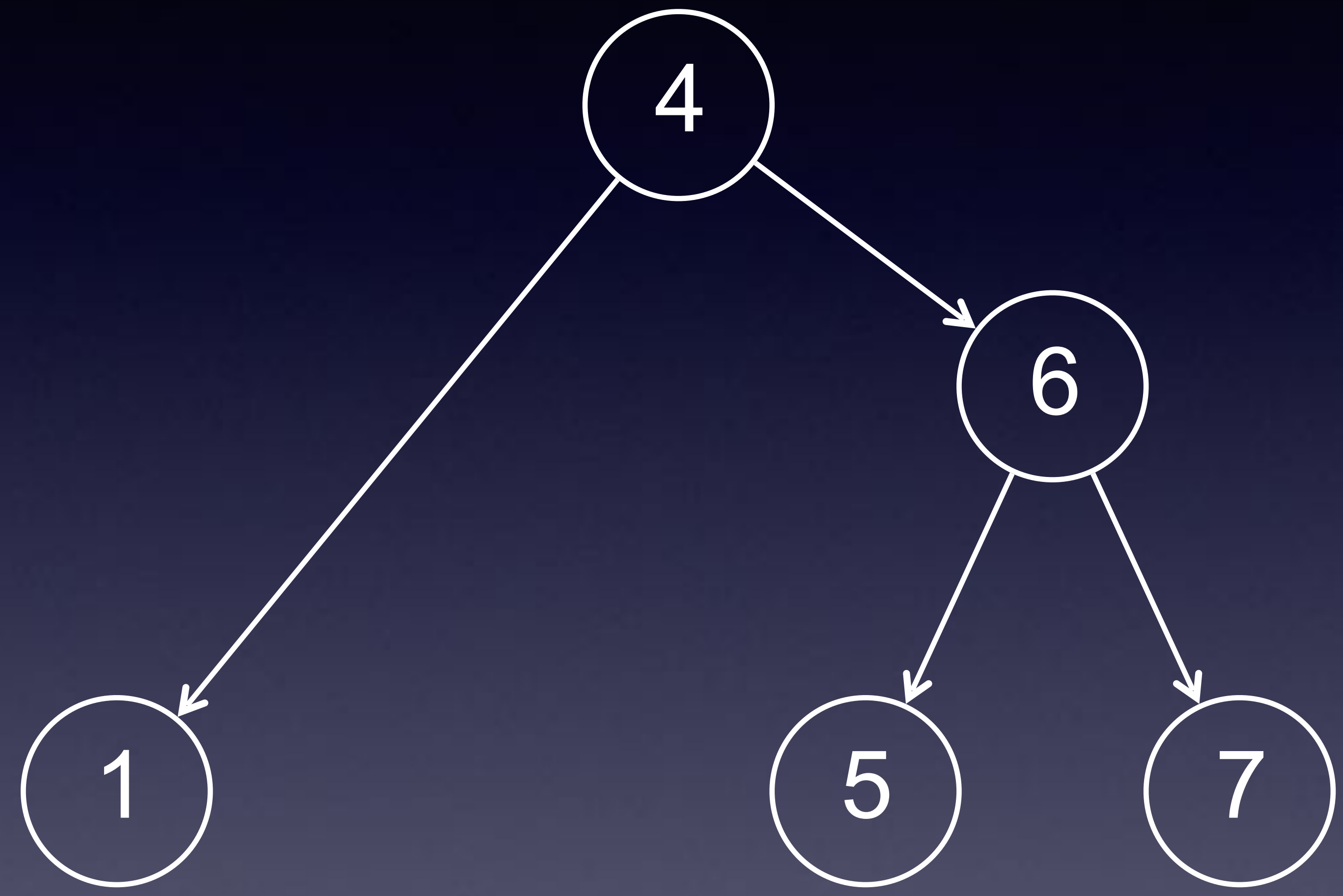


Delete 2

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child
```

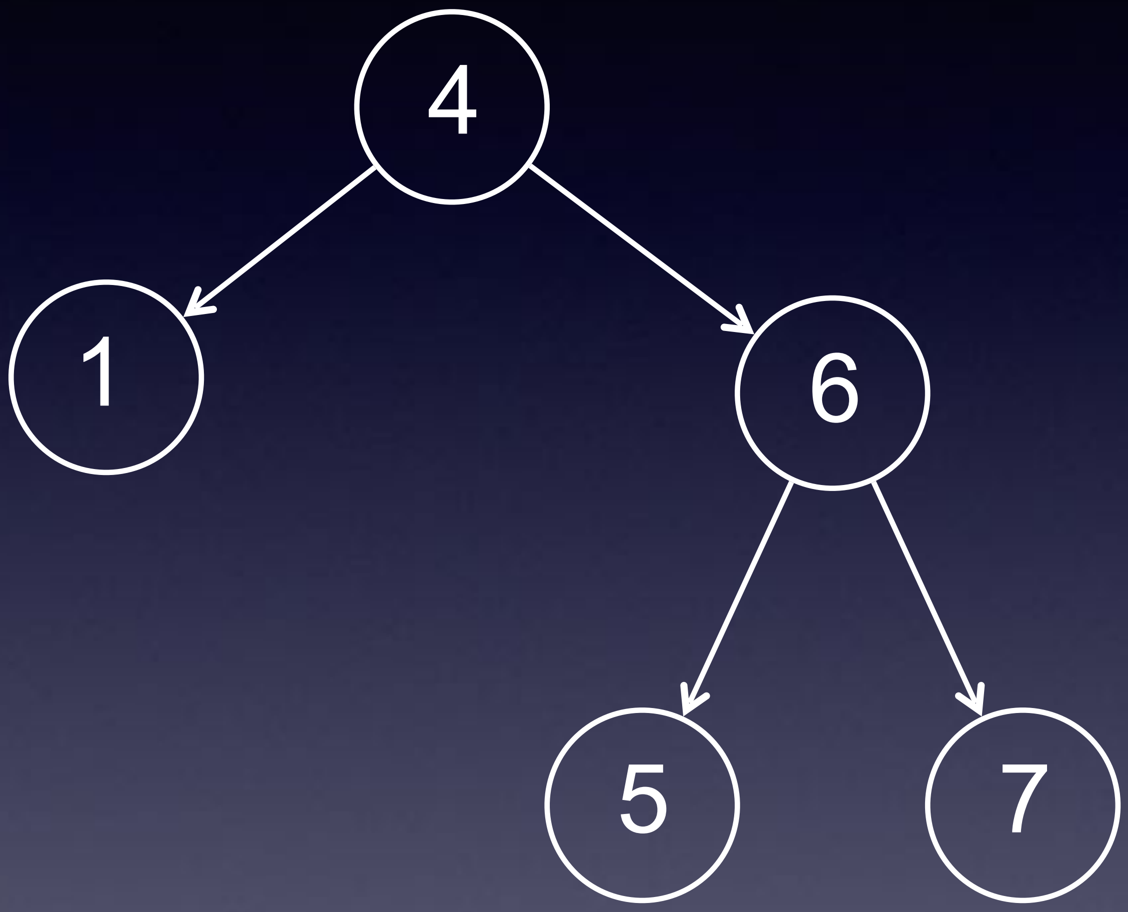


Delete 2

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child
```

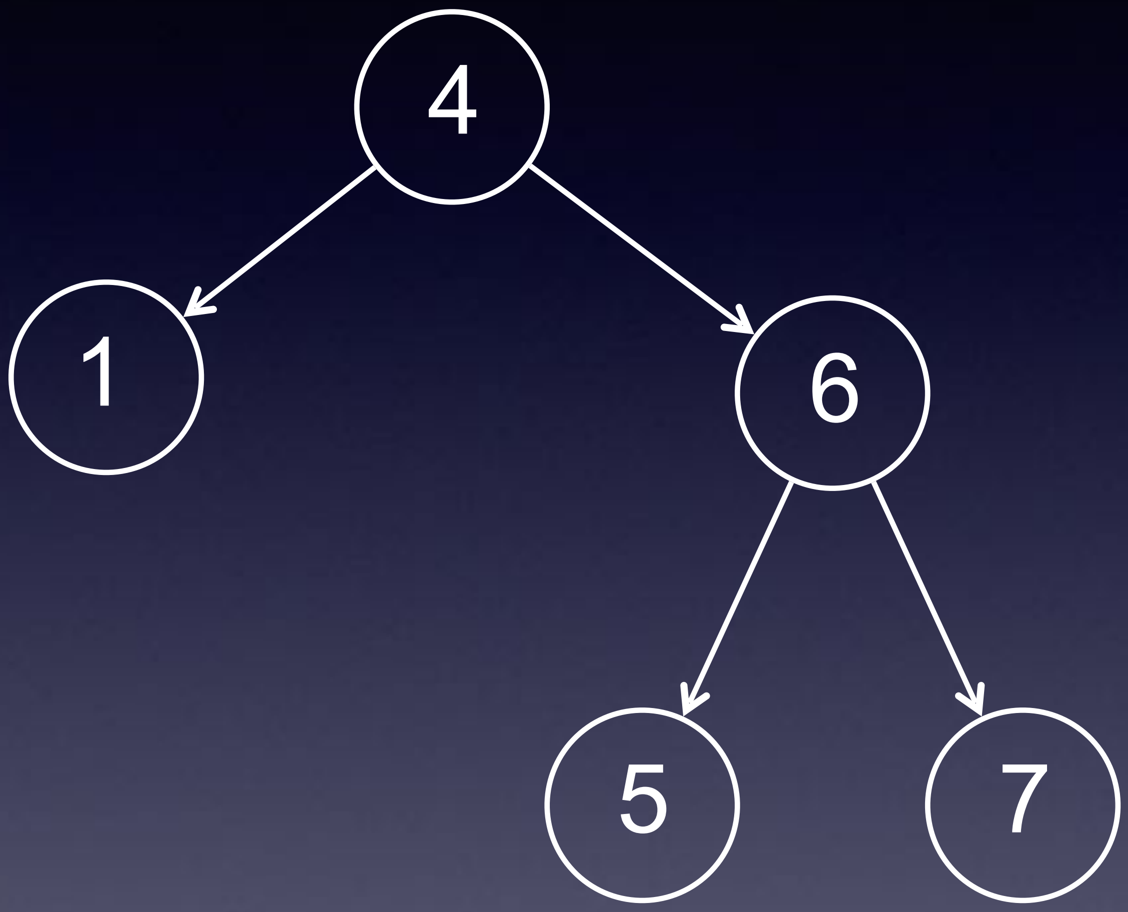


Delete 2

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

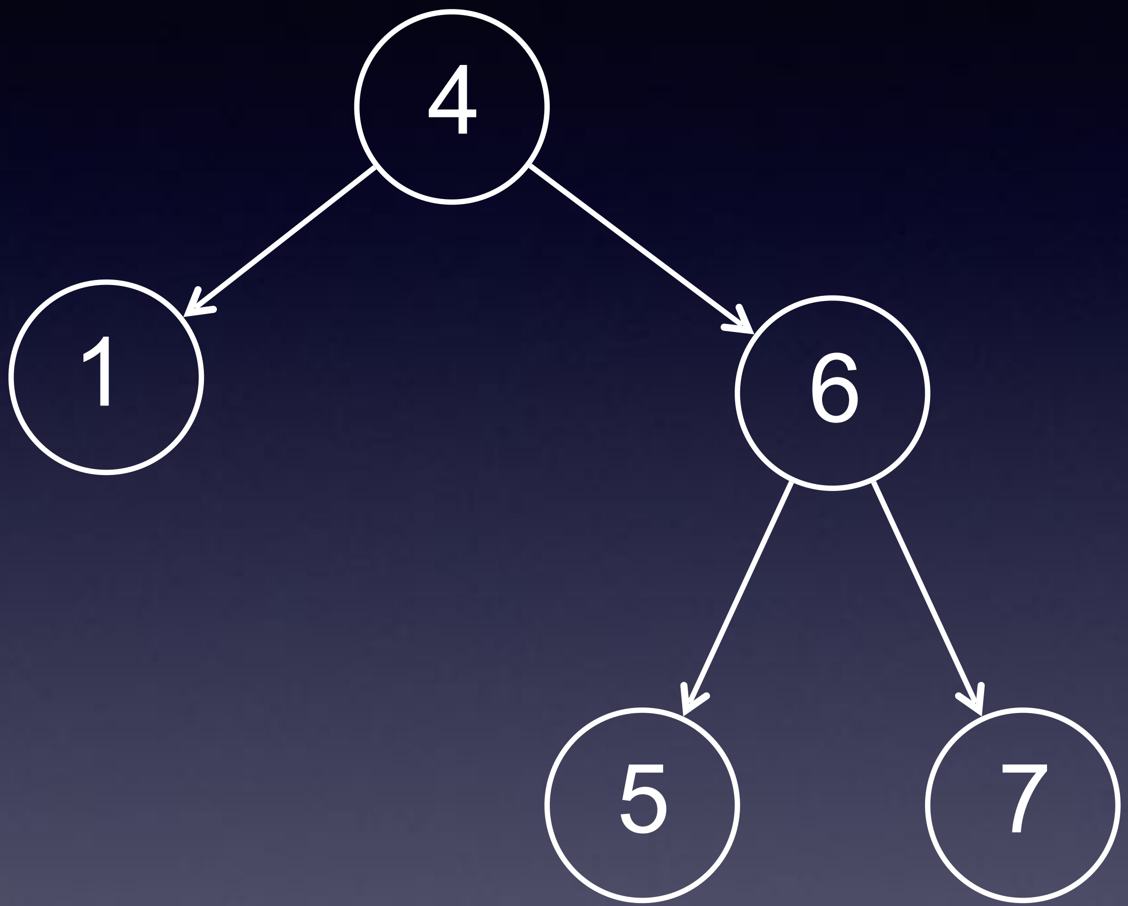


Case 3: The node to be deleted has two children

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

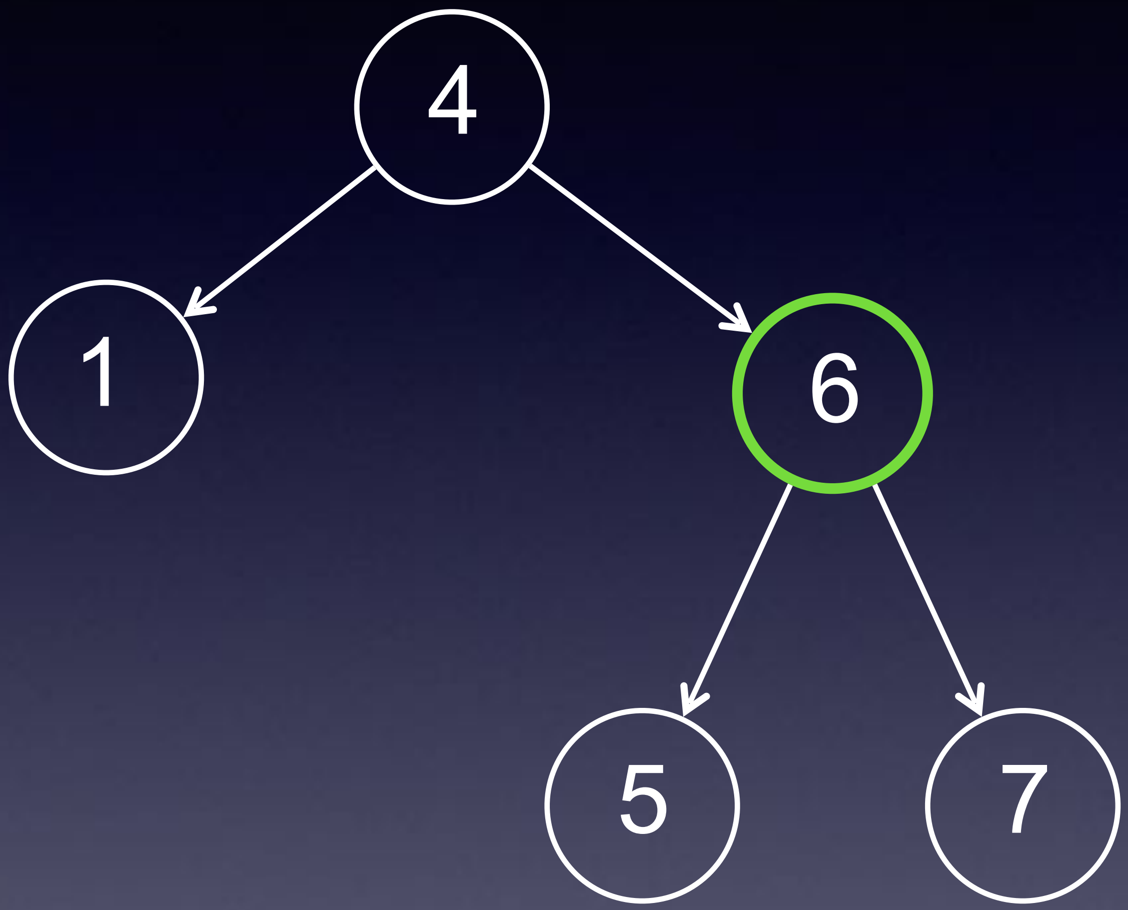


Delete 6

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

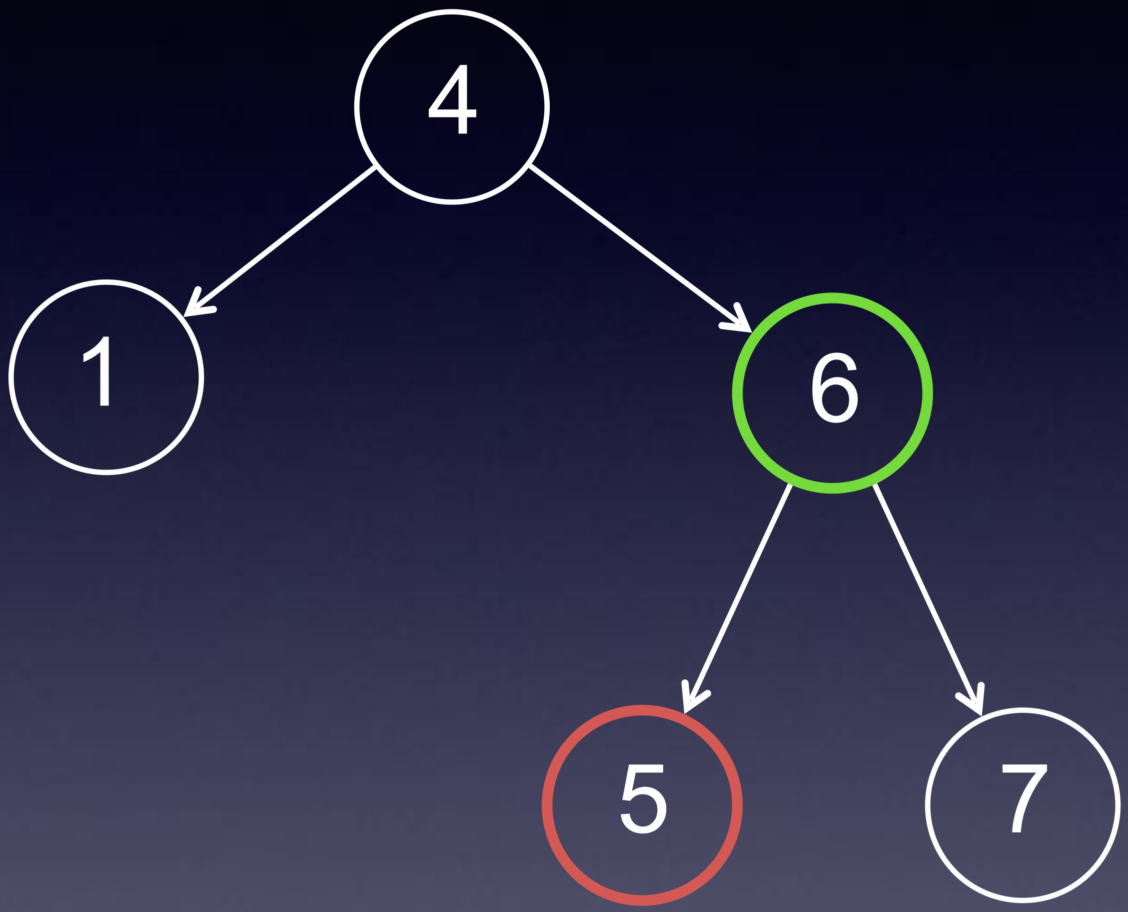


Delete 6

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

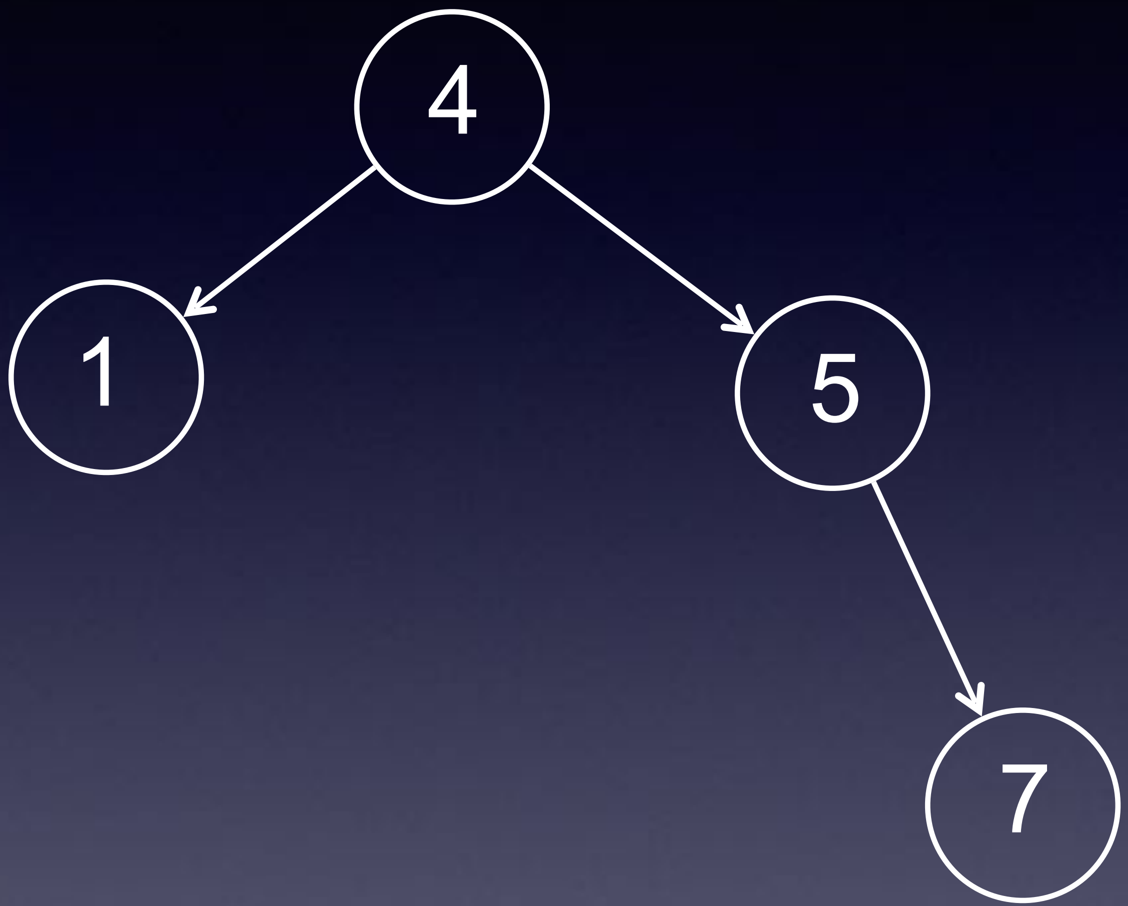


Delete 6

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

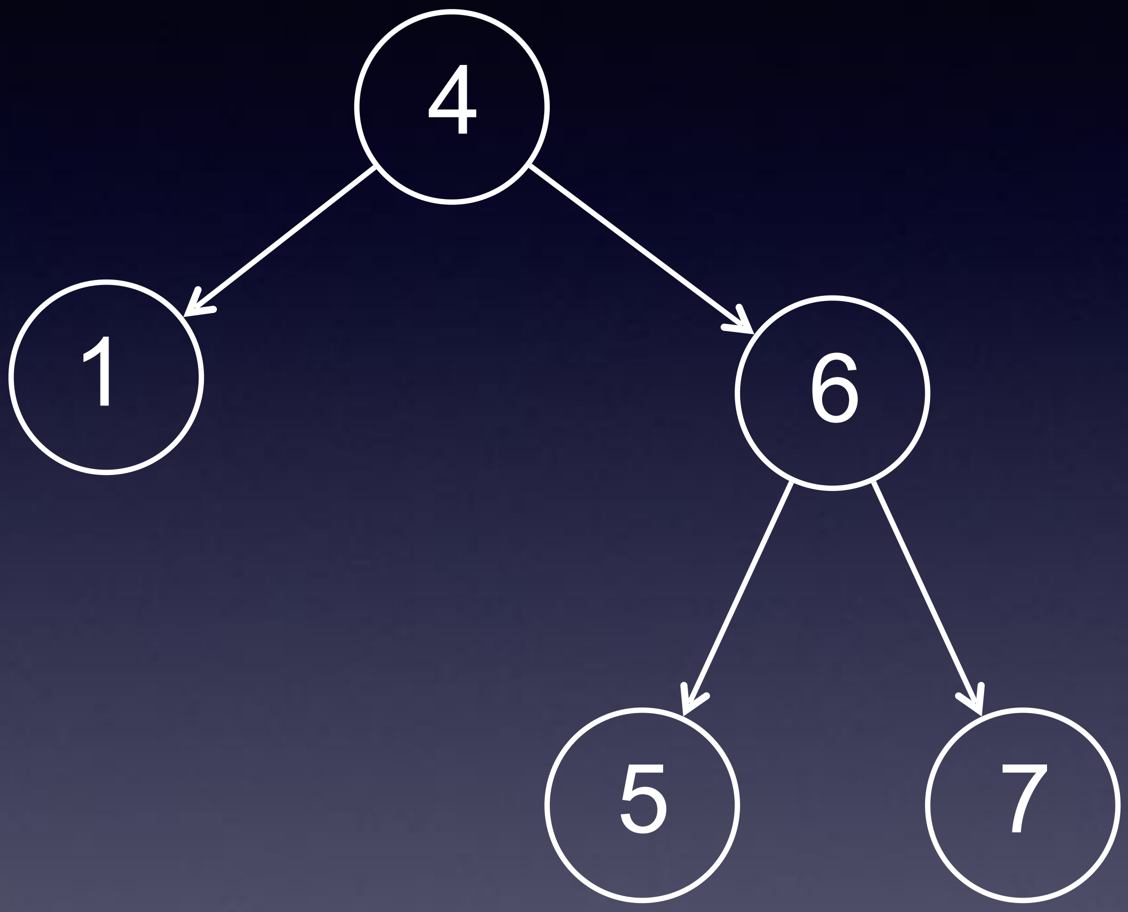


Delete 6

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

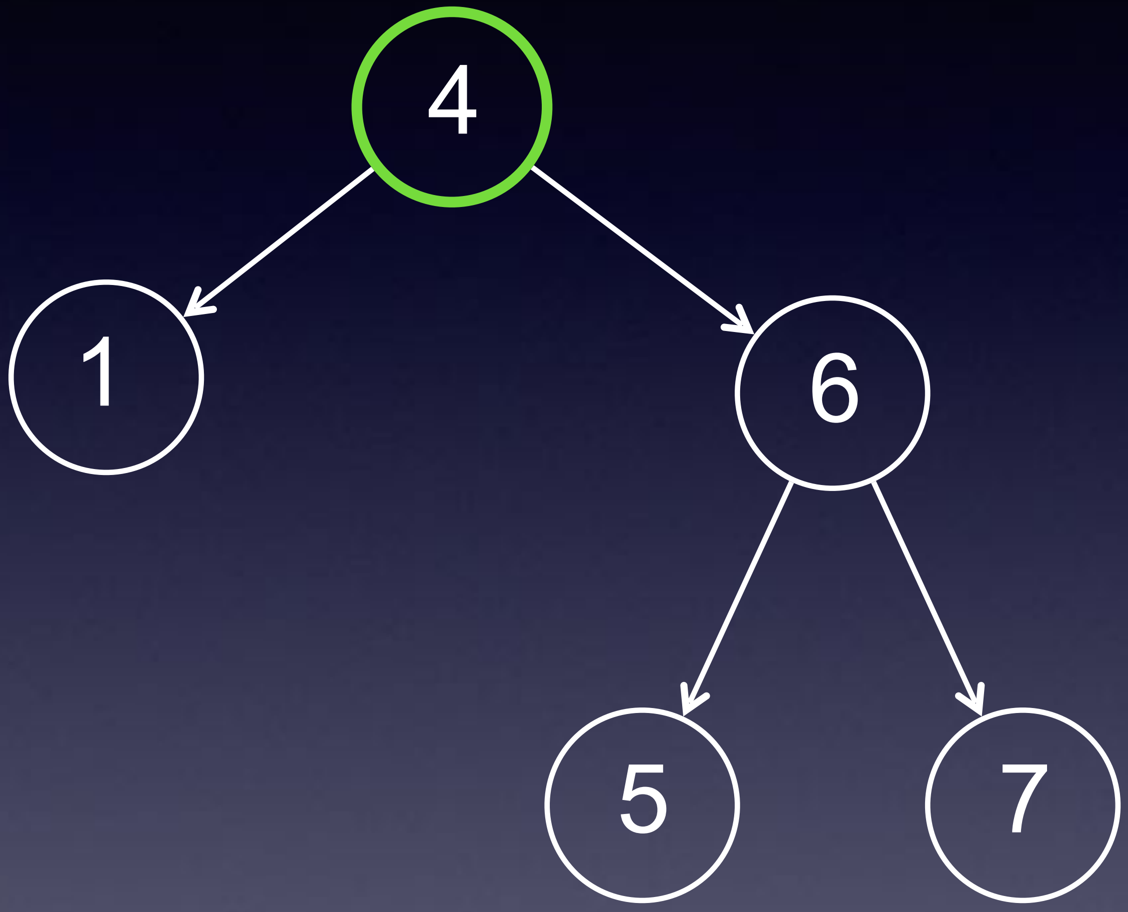


Can we also use the **smallest value in the right subtree**? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

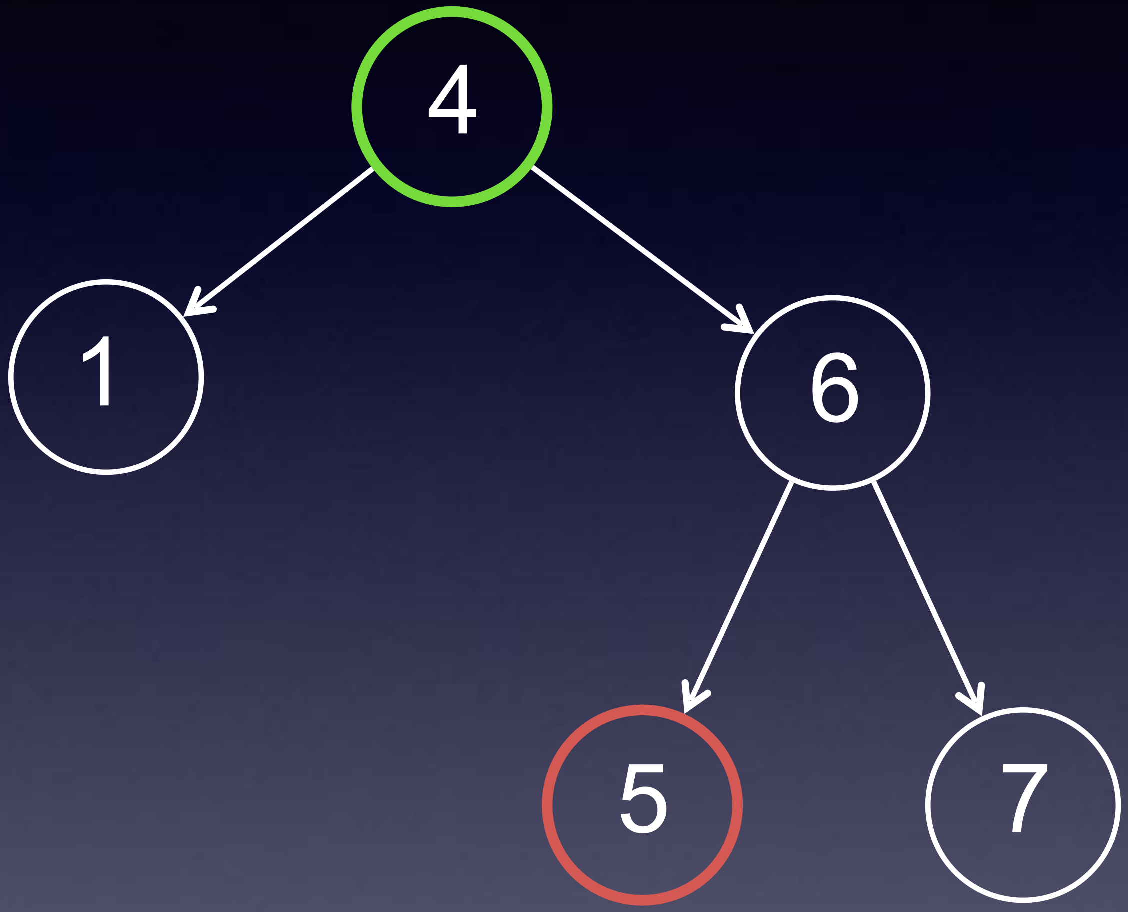


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```

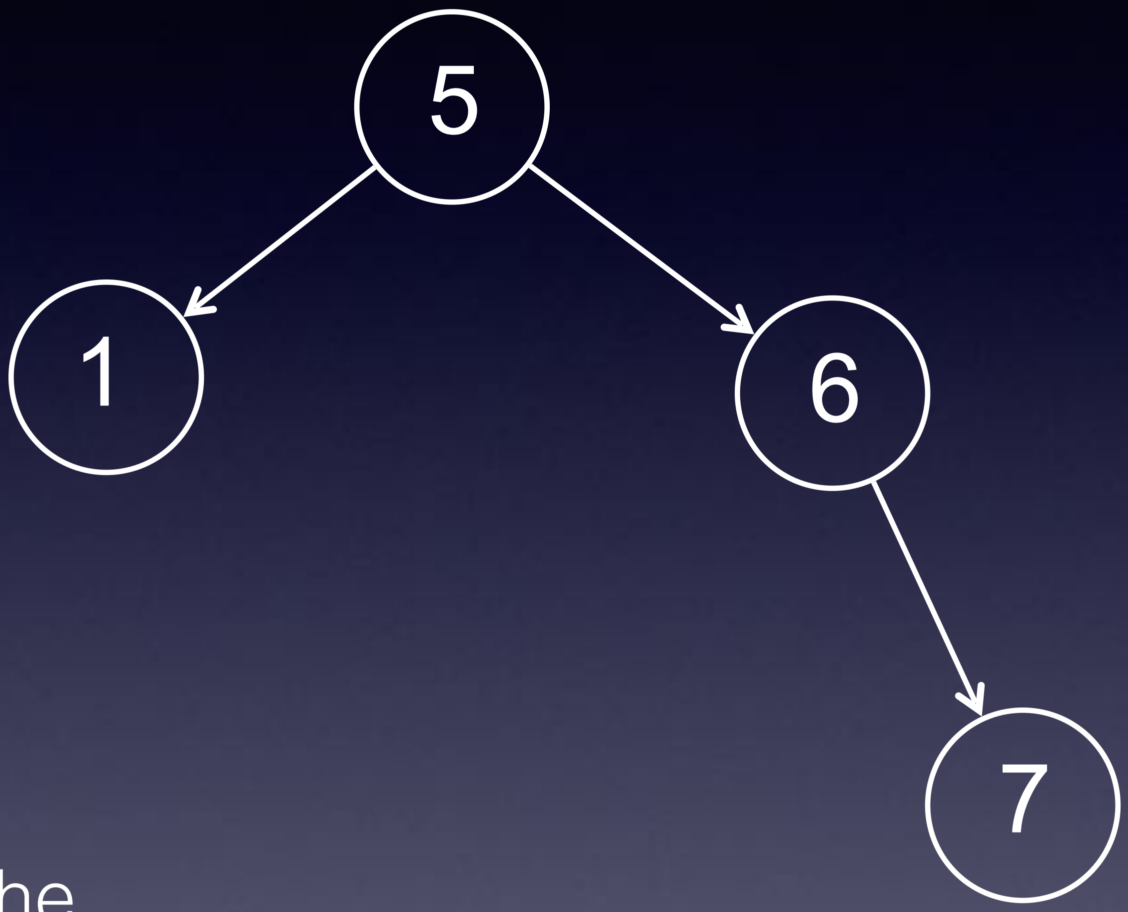


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```



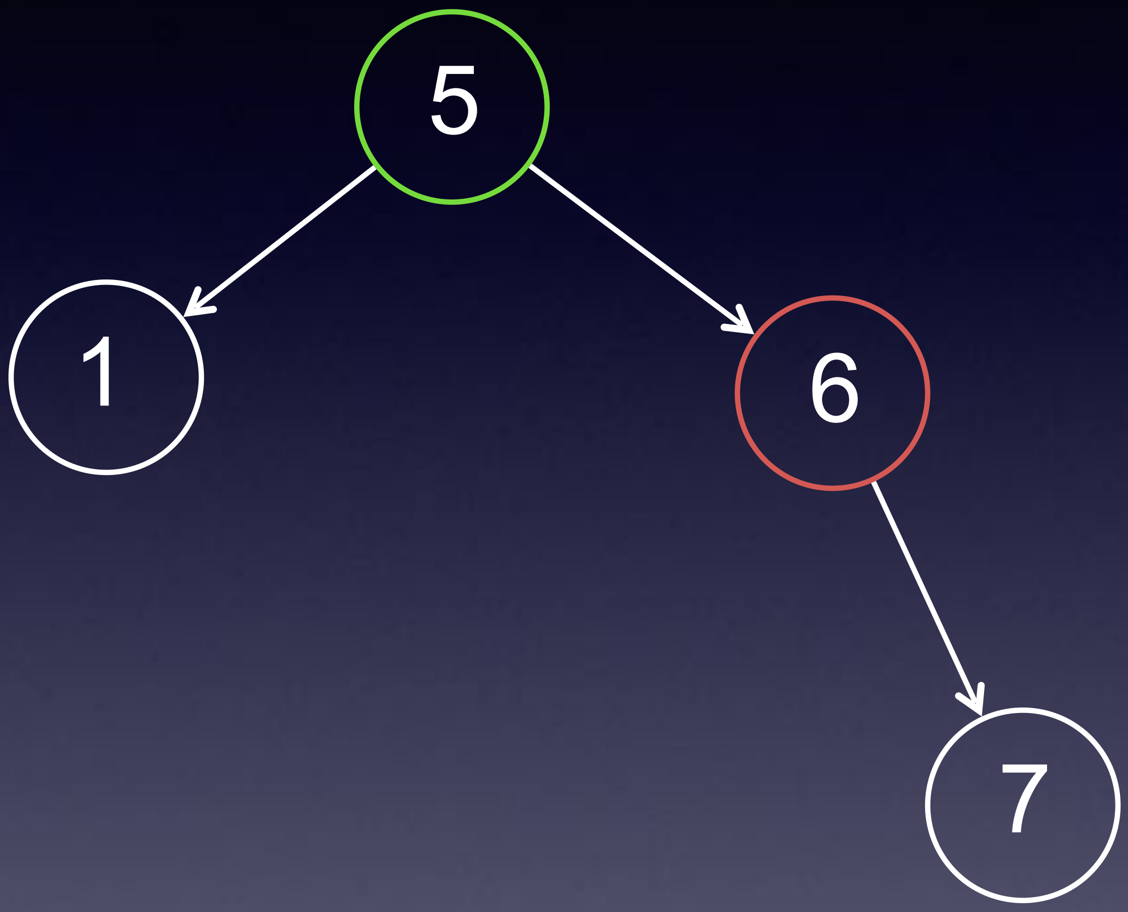
Can we also use the smallest value in the right subtree? Sure. The book calls this value the **successor**.

What about delete 5?

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```



What about delete 5?

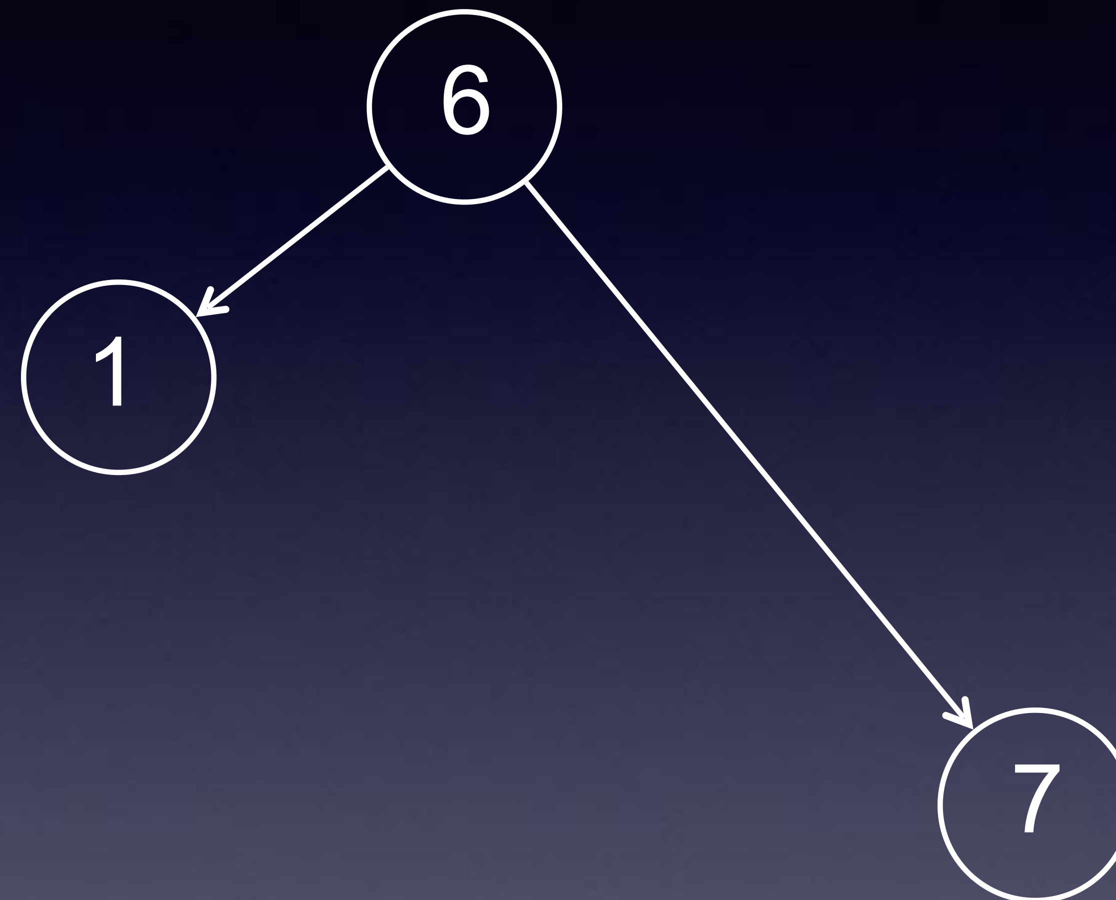
A slightly more complicated case, I didn't show in lecture, is when the smallest value on the right has a child. It will never have more than one child. We need to delete 6 in its old location, but we know how to handle deletion with one child.

So delete 5, replaces 5 with 6 and then we delete the old 6

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one
```



What about delete 5?

A slightly more complicated case, I didn't show in lecture, is when the smallest value on the right has a child. It will never have more than one child. We need to delete 6 in its old location, but we know how to handle deletion with one child.

So delete 5, replaces 5 with 6 and then we delete the old 6

Tree Height and Nodes

Recall, the **height of a tree** is the the maximum number of edges in a path from the root to any node in the tree.

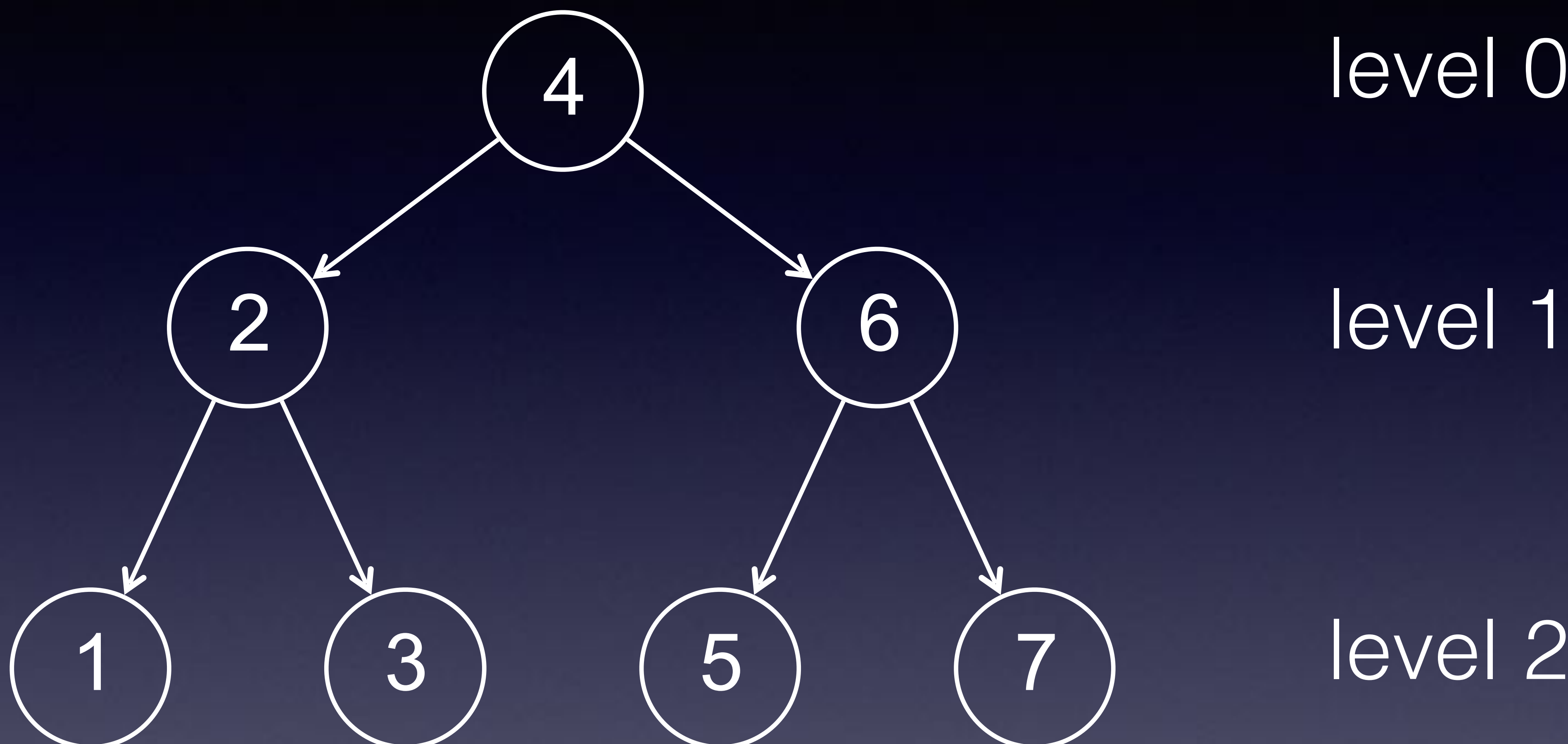
The height of a tree with only a single node is 0.

The height of the empty tree is -1.

The number of nodes in a binary tree of height h is:
at least $h + 1$ and no more than $2^{h+1} - 1$.

Why to we care? The height of a binary search tree is also its worst case find complexity. Good binary search trees have the smallest height for a given number of nodes.

Tree Height and Nodes

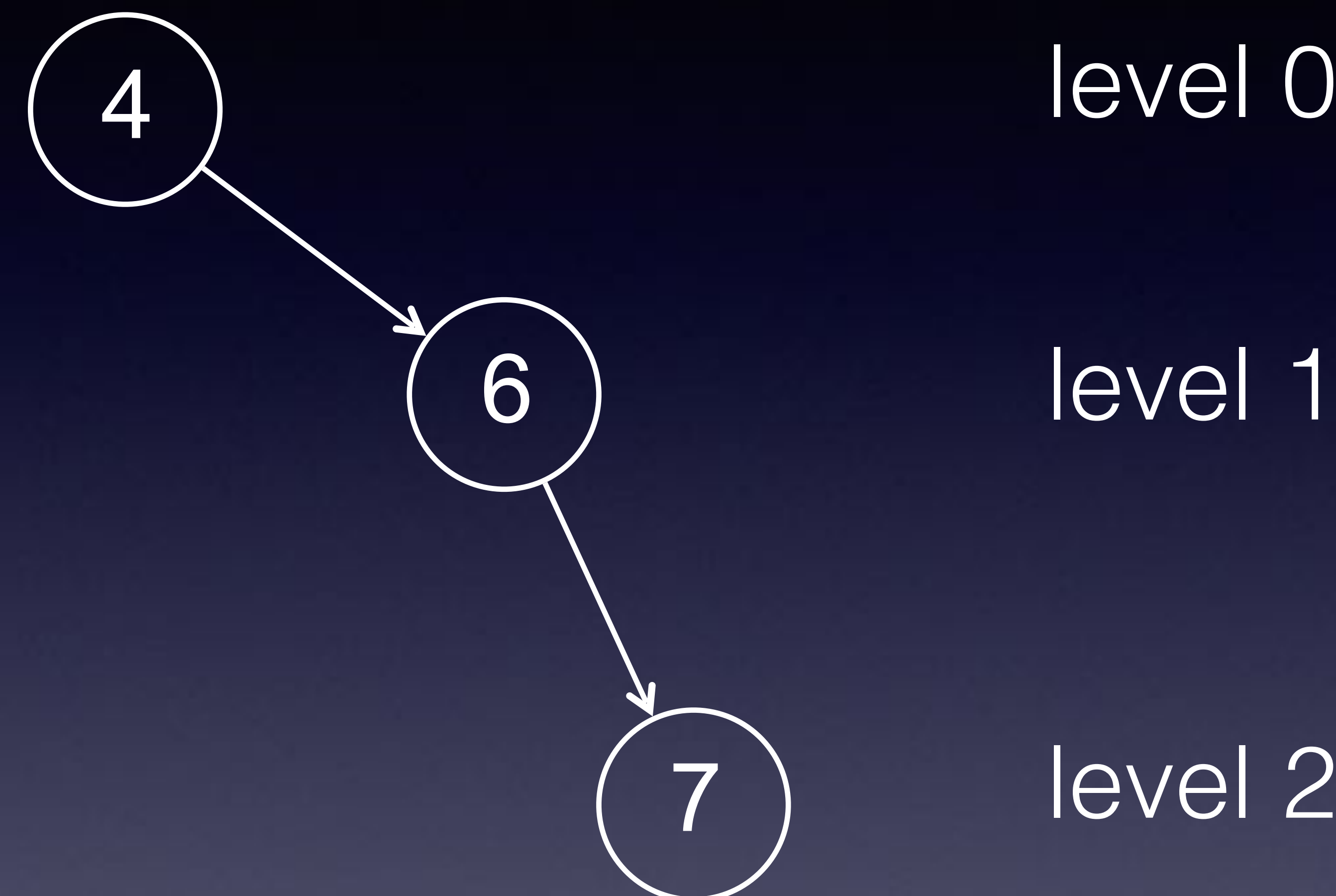


Height = 2

Max nodes at level i = 2^i

Max nodes in tree = $2^{h+1} - 1$ = One less than the max nodes at level h+1

Tree Height and Nodes



Height = 2
Min nodes at level i = 1
Min nodes in tree = h + 1

Binary search trees

To keep things simple, we have been talking about binary search trees as if a node contains only a value and two references to its subtrees.

In reality, a node will have a **key** (usually unique) to identify the associated value(s) or **payload** contained in the node (and, of course, the references to the subtrees). Like a Python dictionary, it is the key that is used in searching.

For example:

Student number:	987654321	← key
Student name:	Dennis Moore	payload
Year:	3	
Program:	ECS	

Binary search trees

Our book uses TreeNode objects to implement a binary search tree. A TreeNode contains two attributes (variables) in addition to what is used for basic binary trees: payload and parent.

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                       parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
```

The BinarySearchTree class implements a binary search tree.

The book uses a BinarySearchTree object to implement the map abstract data type. Recall, map behaves like a Python dictionary. Several data structures that can be used to implement the map abstract data type, for example a hash table.

Map Abstract Data Type

- **Map()** Create a new, empty map.
- **put(key,val)** Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- **get(key)** Given a key, return the value stored in the map or None otherwise.
- **del** Delete the key-value pair from the map using a statement of the form del map[key].
- **len()** Return the number of key-value pairs stored in the map.
- **in** Return True for a statement of the form key in map, if the given key is in the map.