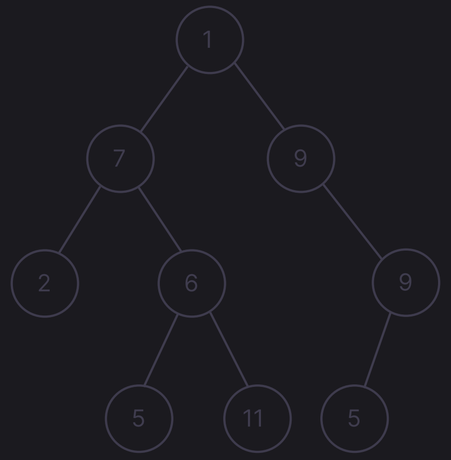
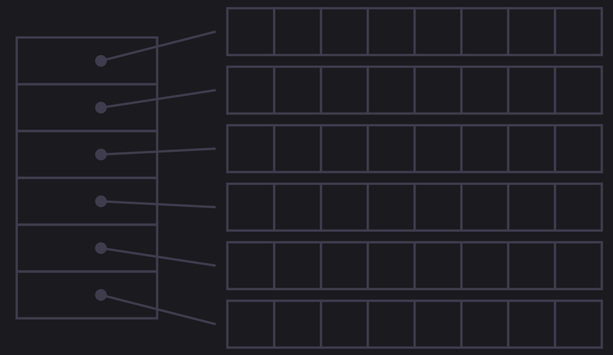
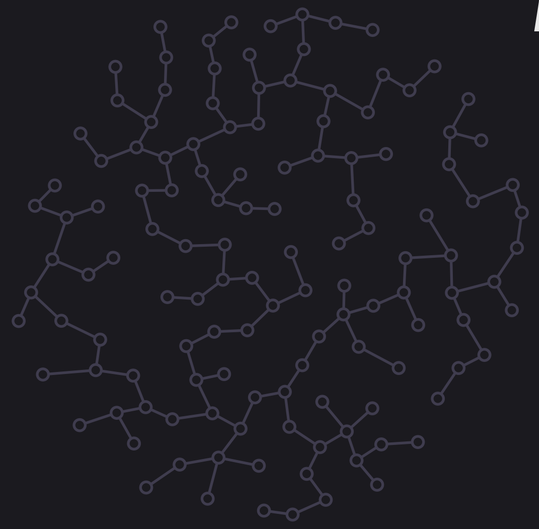


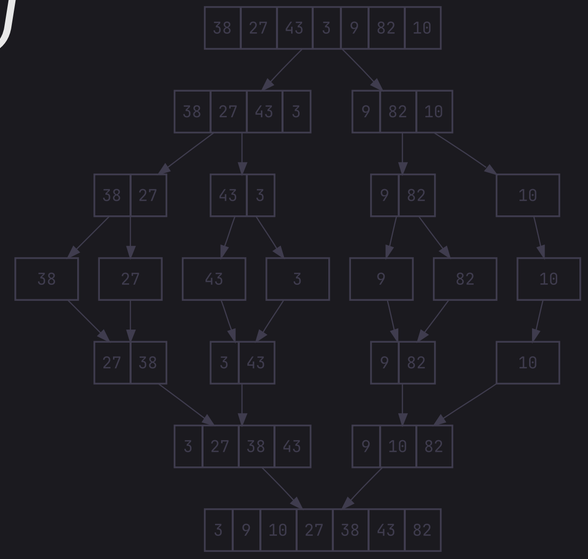


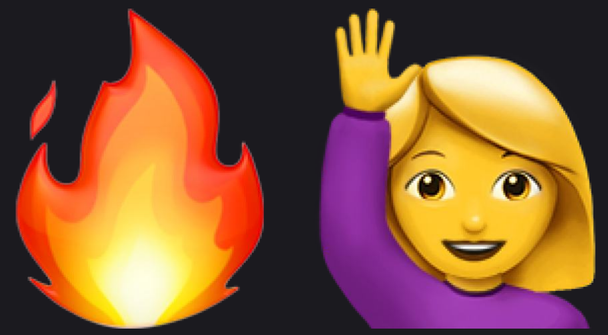
data structures & algorithms

Tutorial 10

(Week 10)







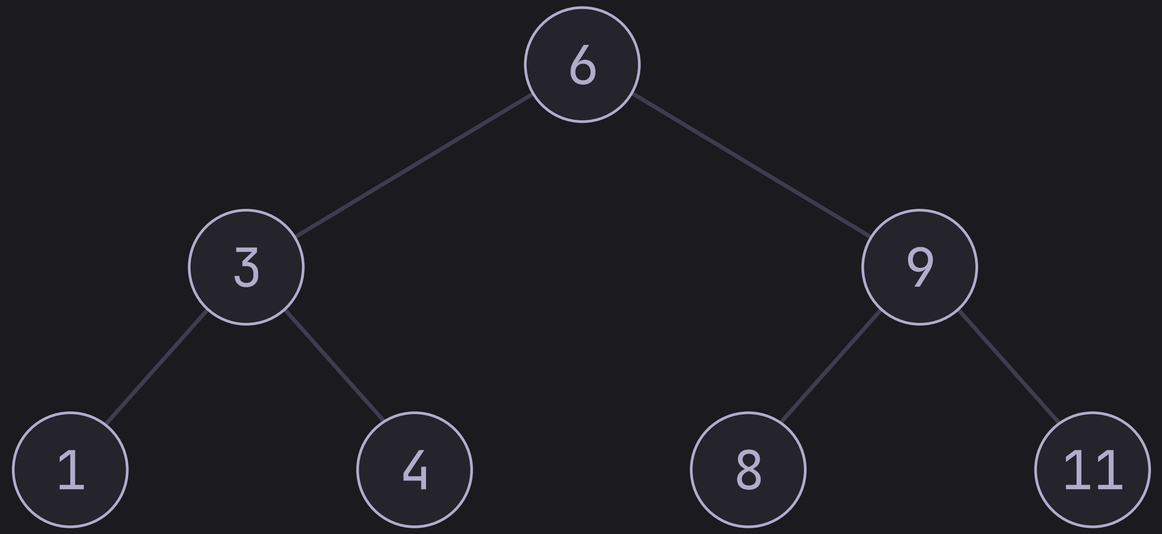
Burning questions from the previous tutorial?

This week’s lab

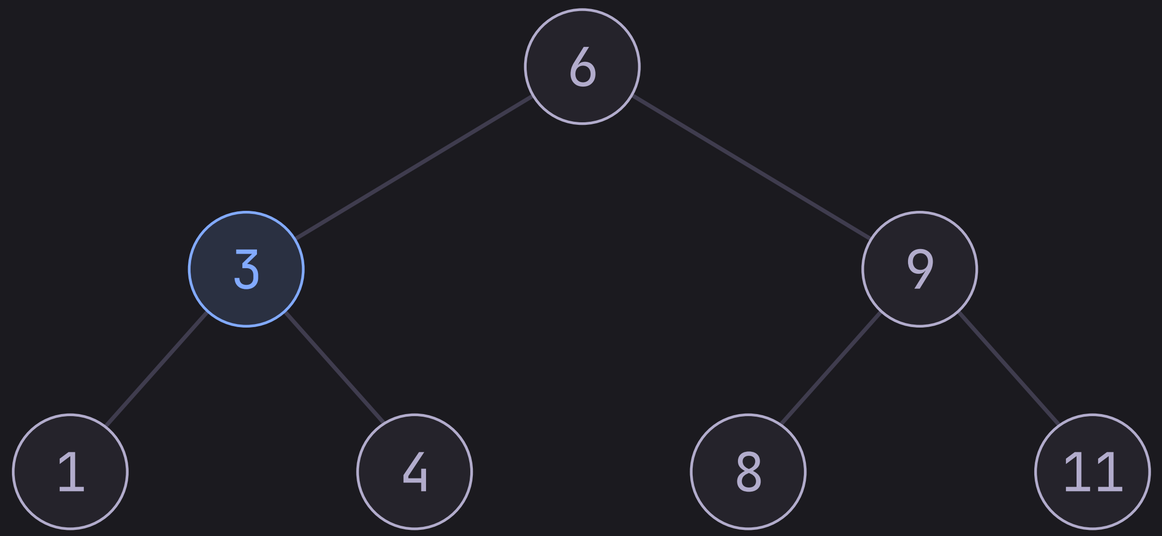
Trees

- Binary Search Tree (BST)
- Common BST methods
- BST Traversals

Binary Search Tree

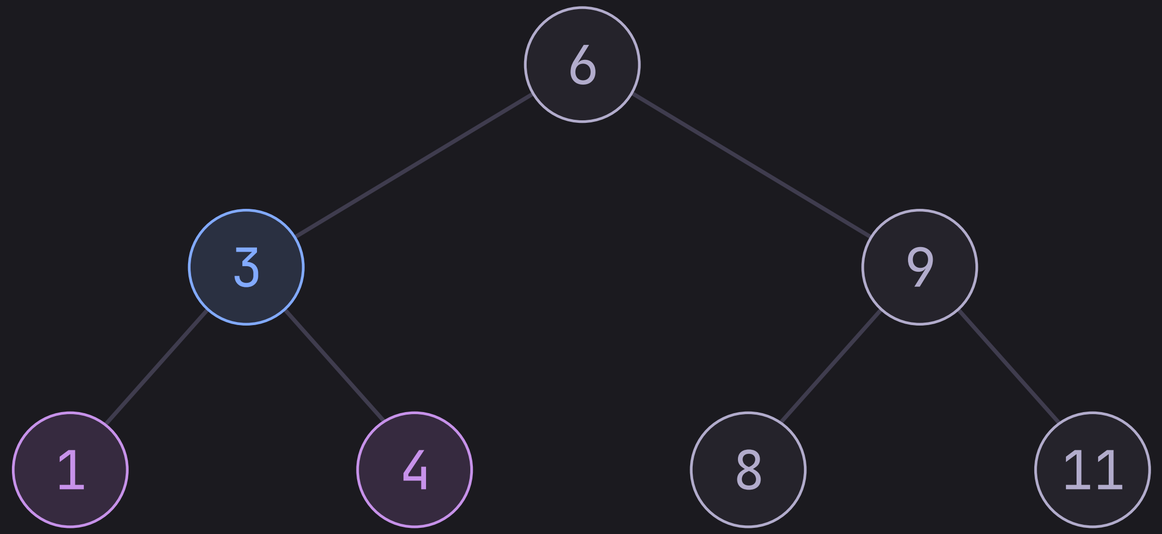


Binary Search Tree



Each circle is called a node

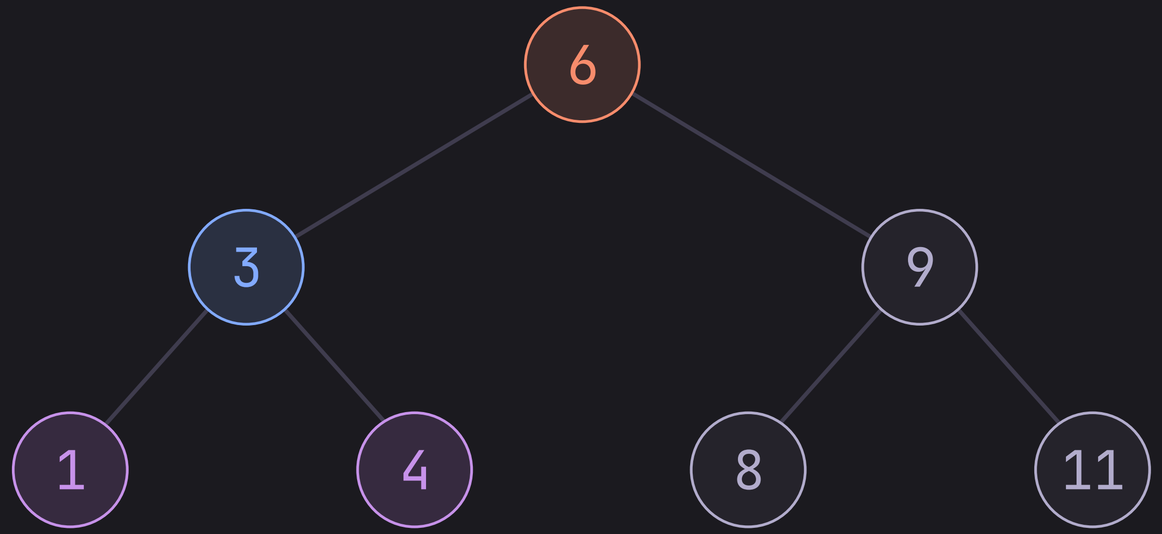
Binary Search Tree



Each circle is called a node

each node has two children

Binary Search Tree

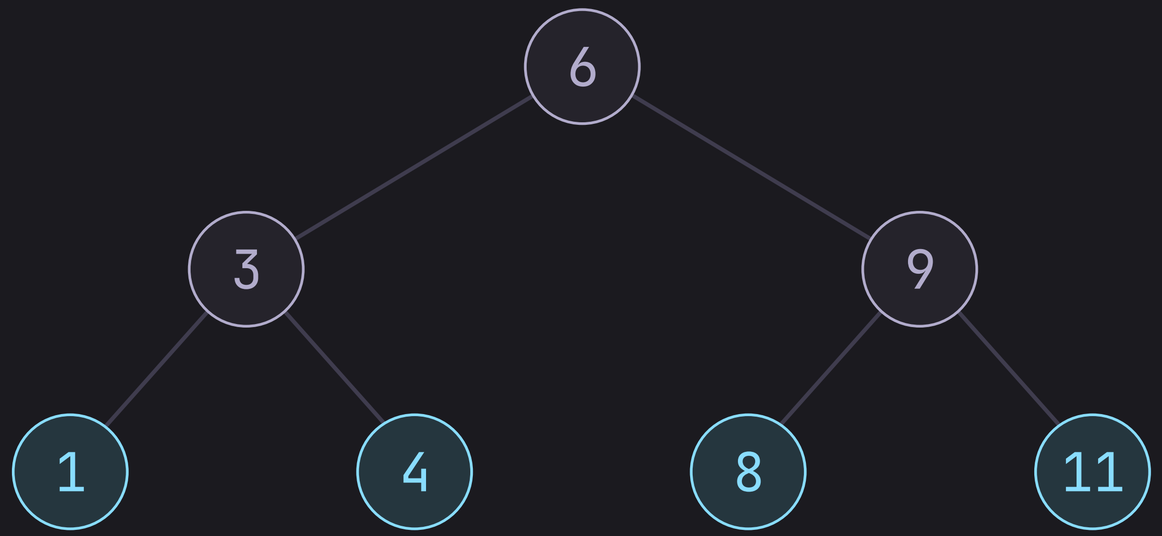


Each circle is called a node

each node has two children

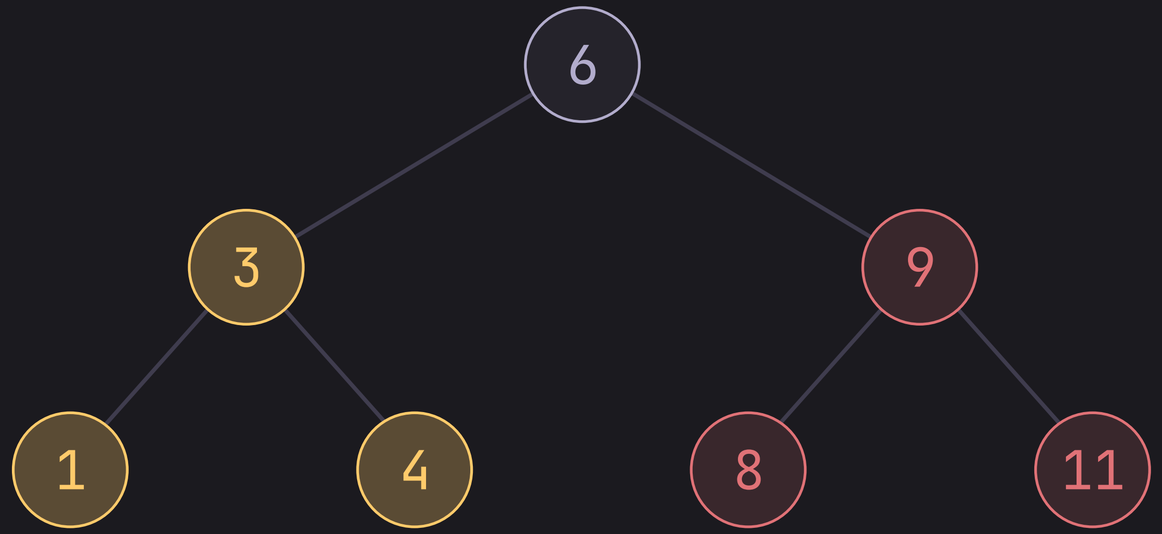
and one parent

Binary Search Tree



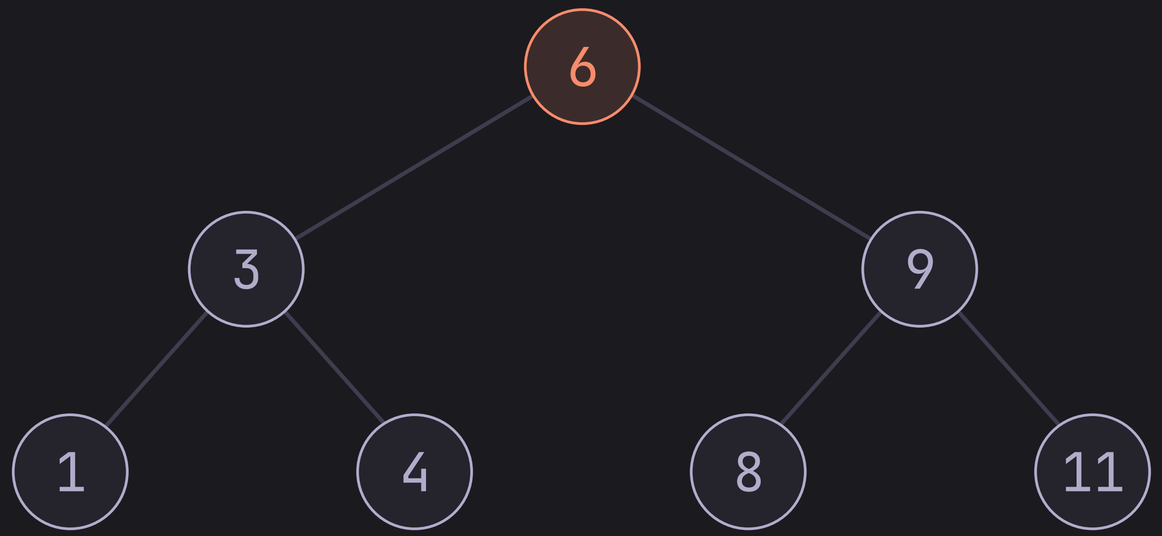
We call all the nodes with no children leaf nodes

Binary Search Tree



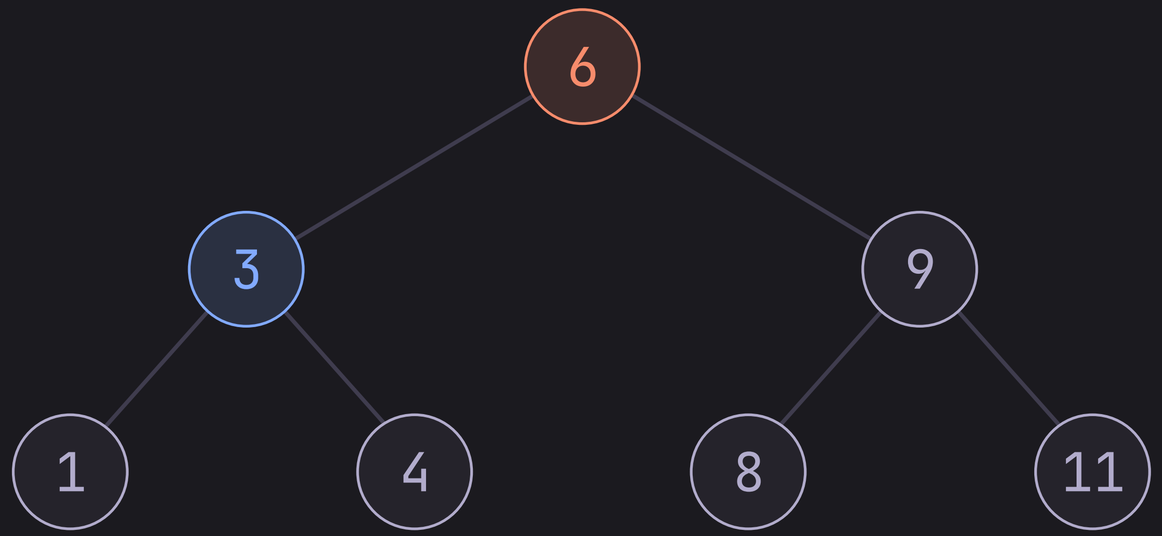
Each node’s children define a
left subtree and a right subtree

Binary Search Tree



The top node is called the root

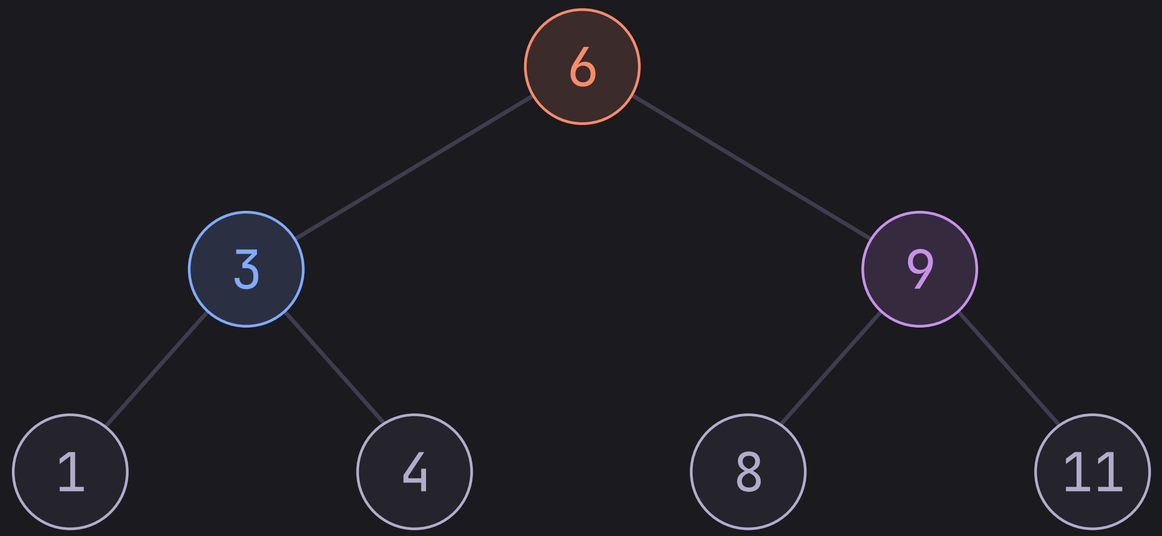
Binary Search Tree



The top node is called the root

This is the left child of the root

Binary Search Tree

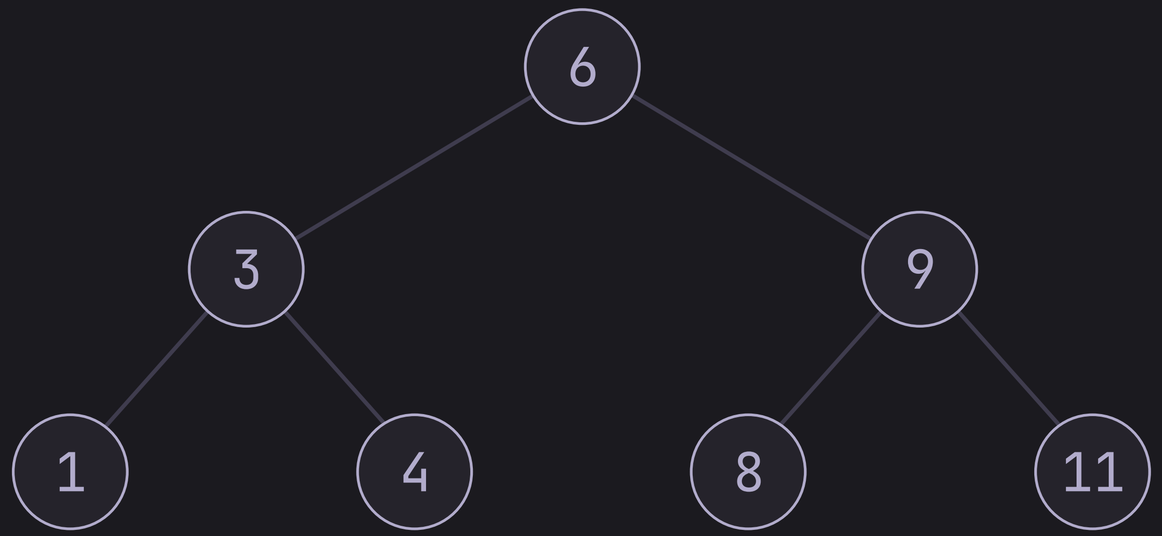


The top node is called the root

This is the left child of the root

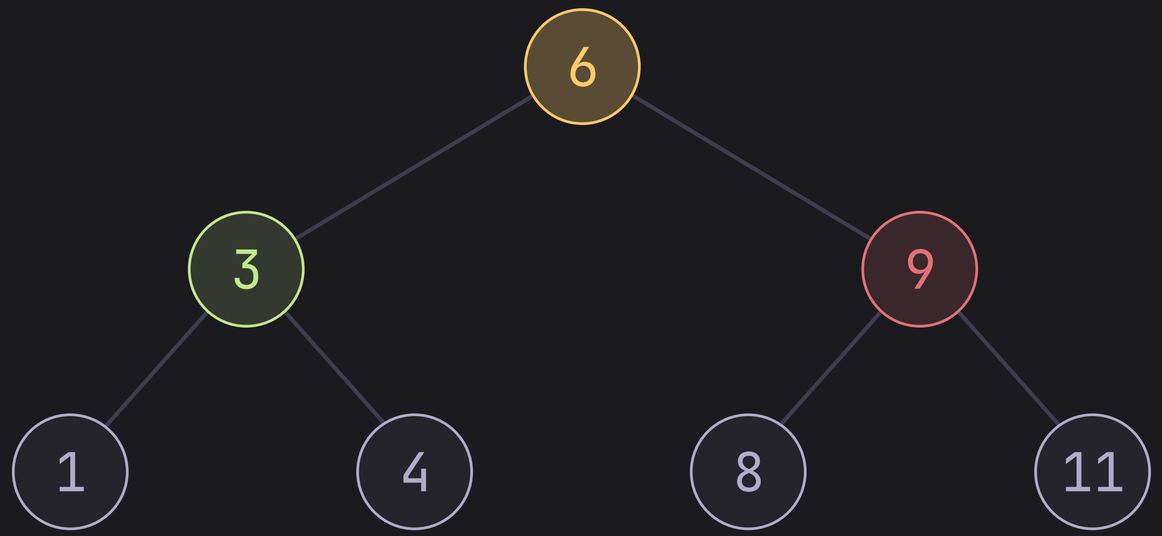
and this is the right child of the root

Binary Search Tree



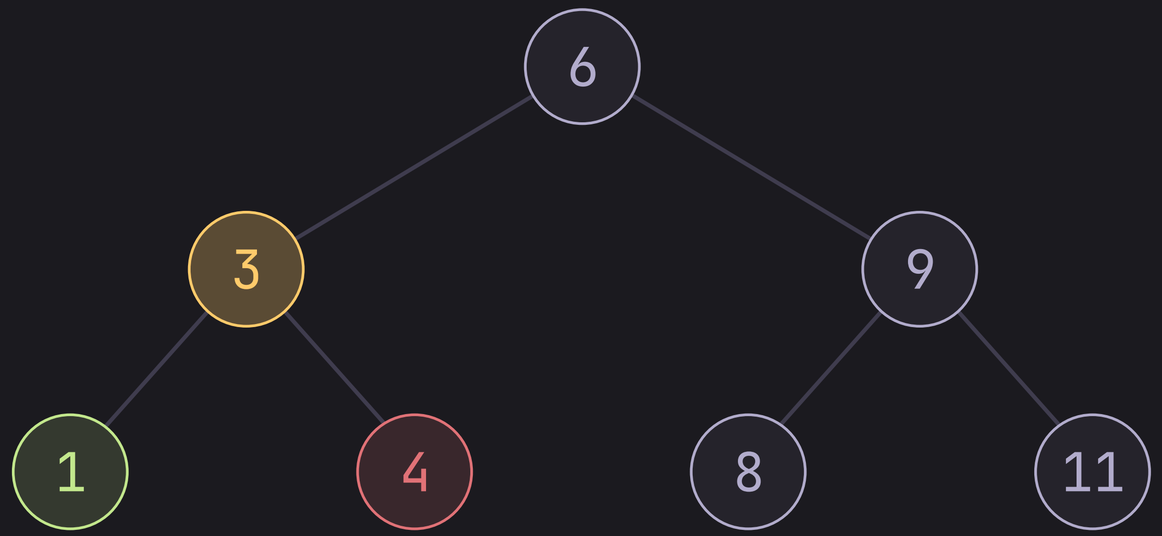
left child is less than its parent and the
right child is greater than its patent

Binary Search Tree



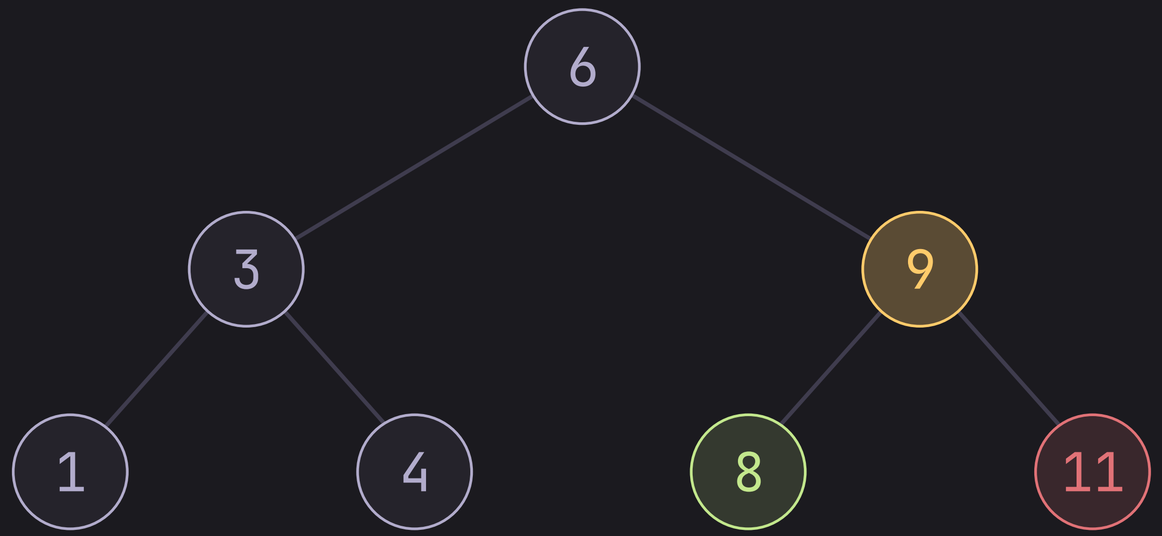
left child is less than its parent and the
right child is greater than its patent

Binary Search Tree



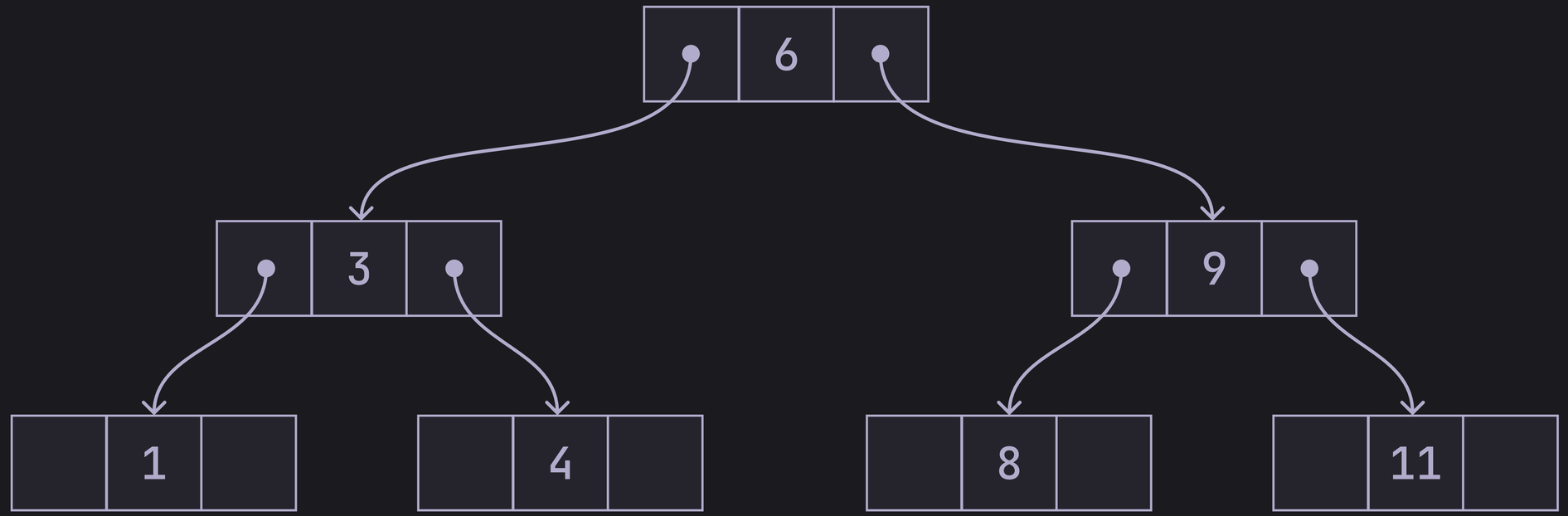
left child is less than its parent and the
right child is greater than its patent

Binary Search Tree



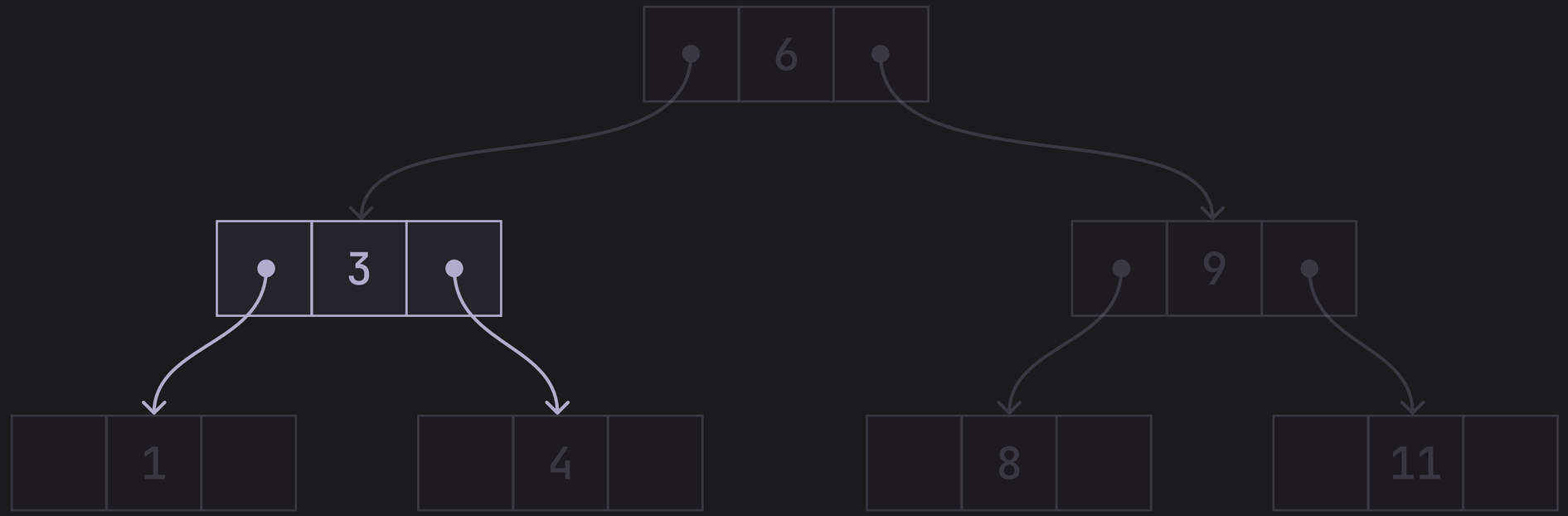
left child is less than its parent and the
right child is greater than its patent

Binary Search Tree



You can think of a binary tree as an extension of a linked list

Binary Search Tree



You can think of a binary tree as an extension of a linked list

Binary Search Tree

Binary Search Tree

```
struct Node {
}
```

Binary Search Tree

```
struct Node {
  T data {};
  Node* left {nullptr};
  Node* right {nullptr};
}
```

Binary Search Tree

```
struct Node {
  T data {};
  Node* left {nullptr};
  Node* right {nullptr};
  Node* parent {nullptr};
}
```

Binary Search Tree

```
struct Node {
  T data {};
  Node* left {nullptr};
  Node* right {nullptr};
  Node* parent {nullptr};
  Node() {}
  Node(T d, Node* node = nullptr) : data {d}, parent {node} {}
}
```

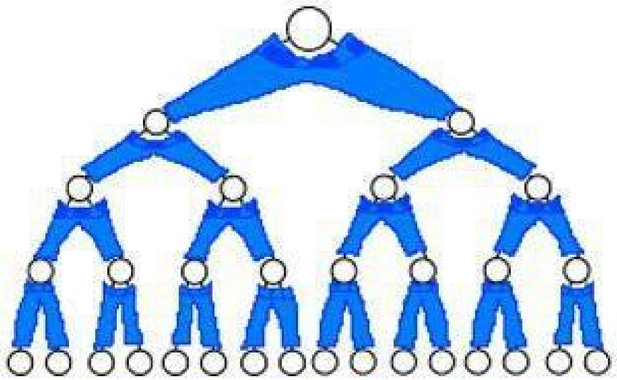
Now time for a very important question...

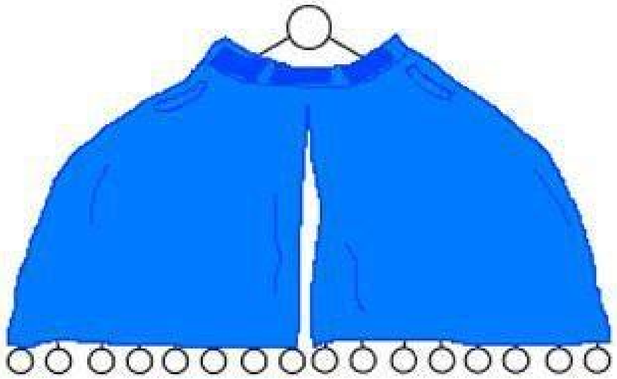
If a binary tree wore pants would he wear them

like this

or

like this?



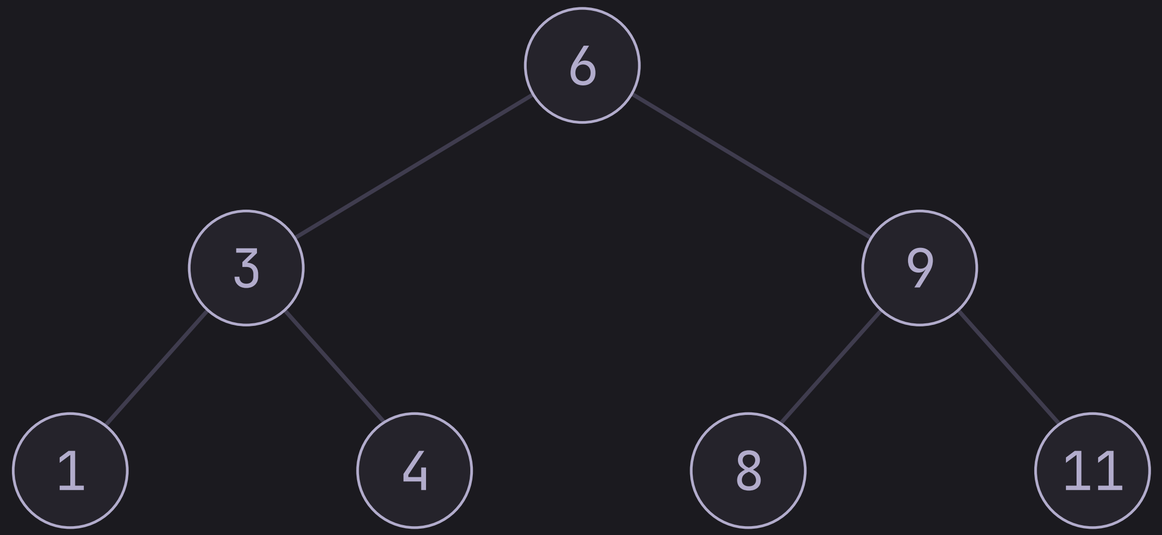


Ok now time to look at some common methods for binary search trees

Binary Search Tree

7

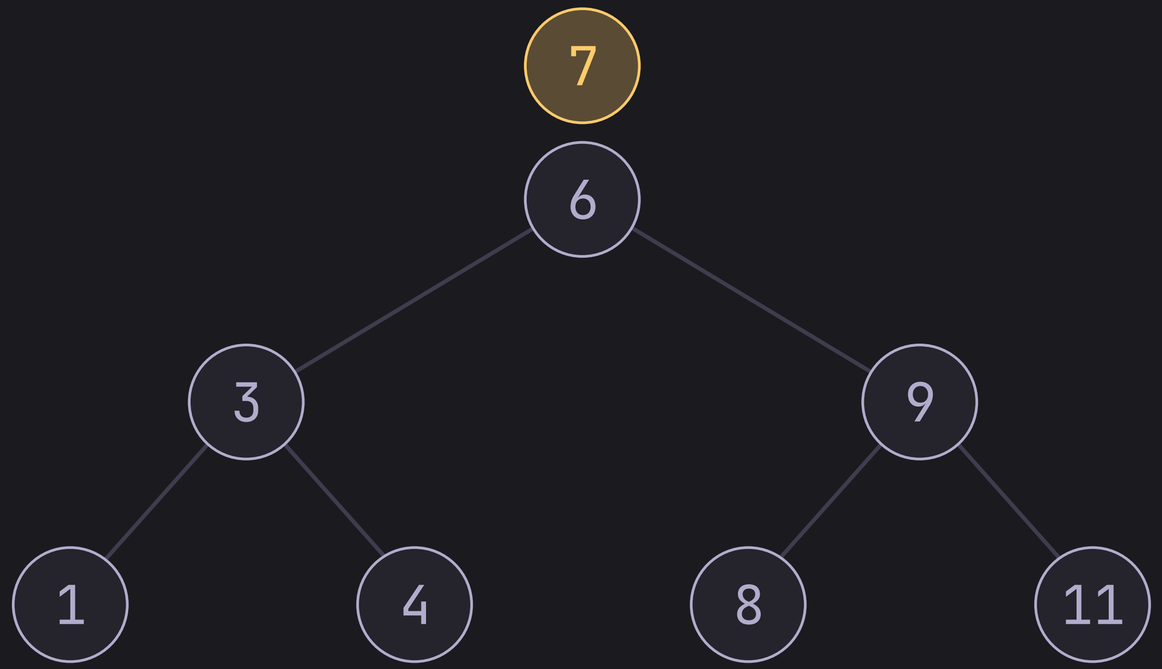
insert(7)



To insert an element we start at the root

Binary Search Tree

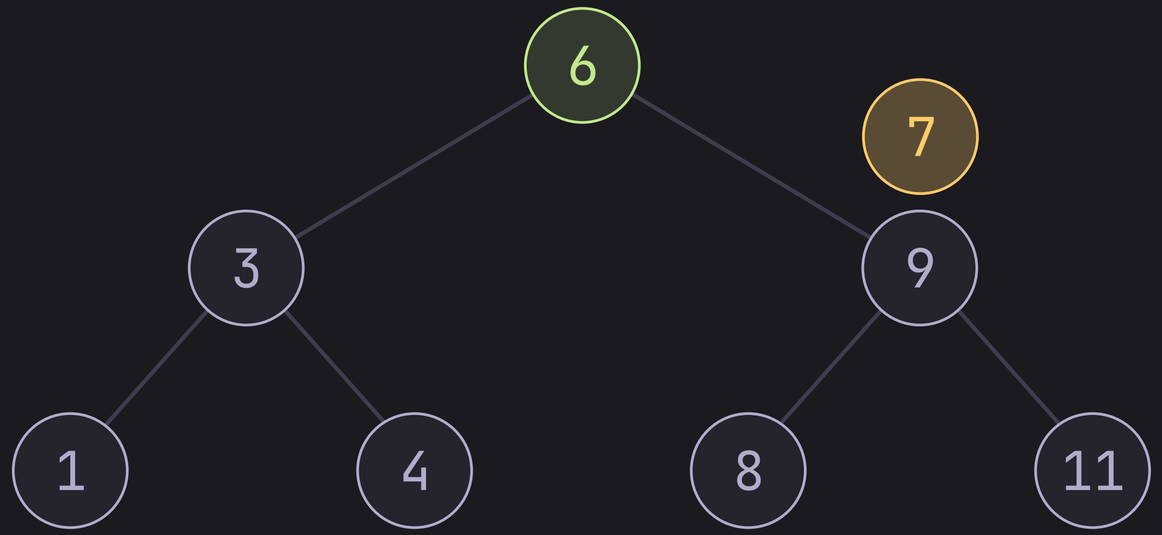
```
insert(7)
```



Now we compare if the new node is greater or less than the current node

Binary Search Tree

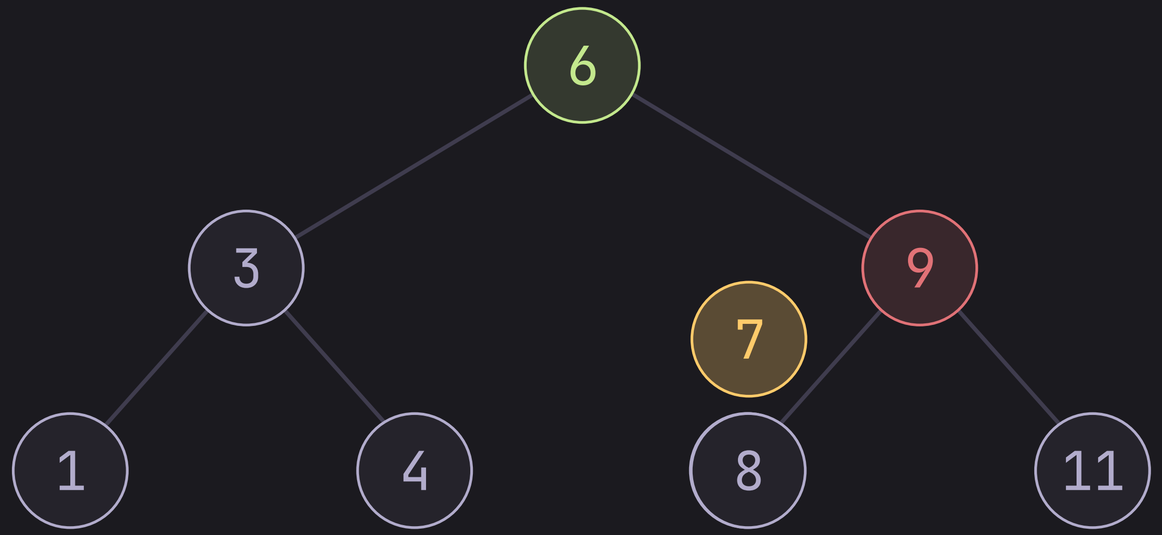
```
insert(7)
```



Now we compare if the new node is greater or less than the current node

Binary Search Tree

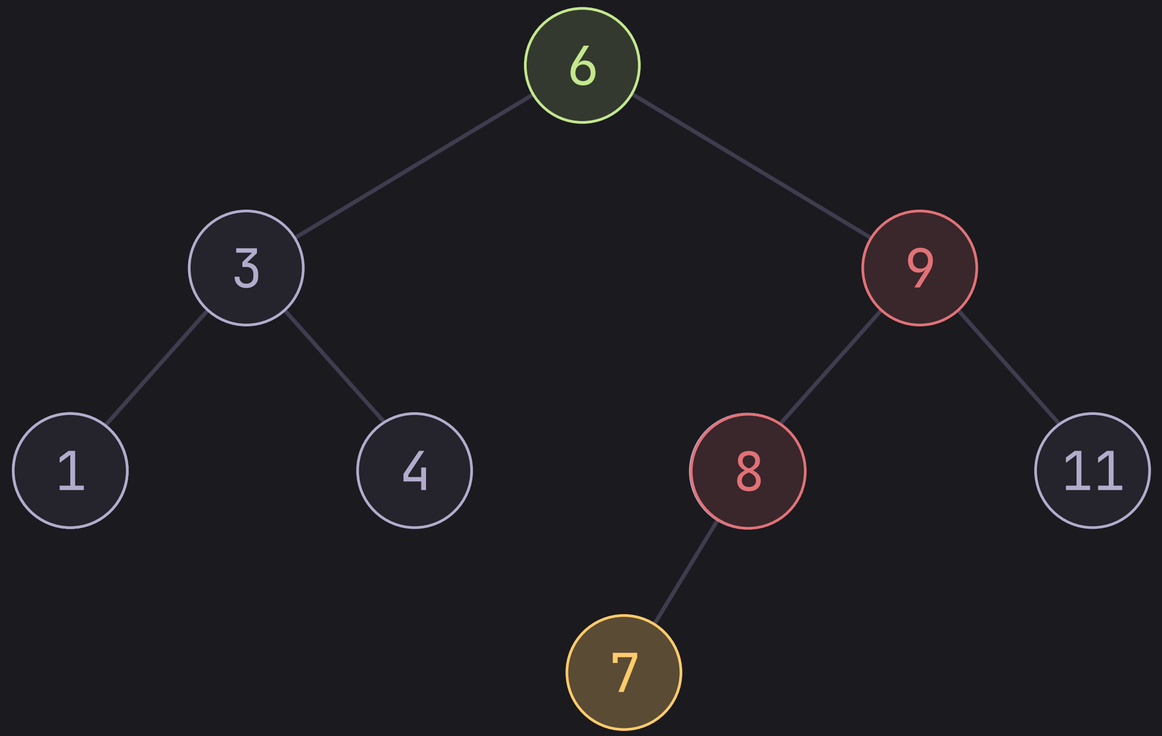
```
insert(7)
```



Our new node is less than the current node, but the current node does not have a left child

Binary Search Tree

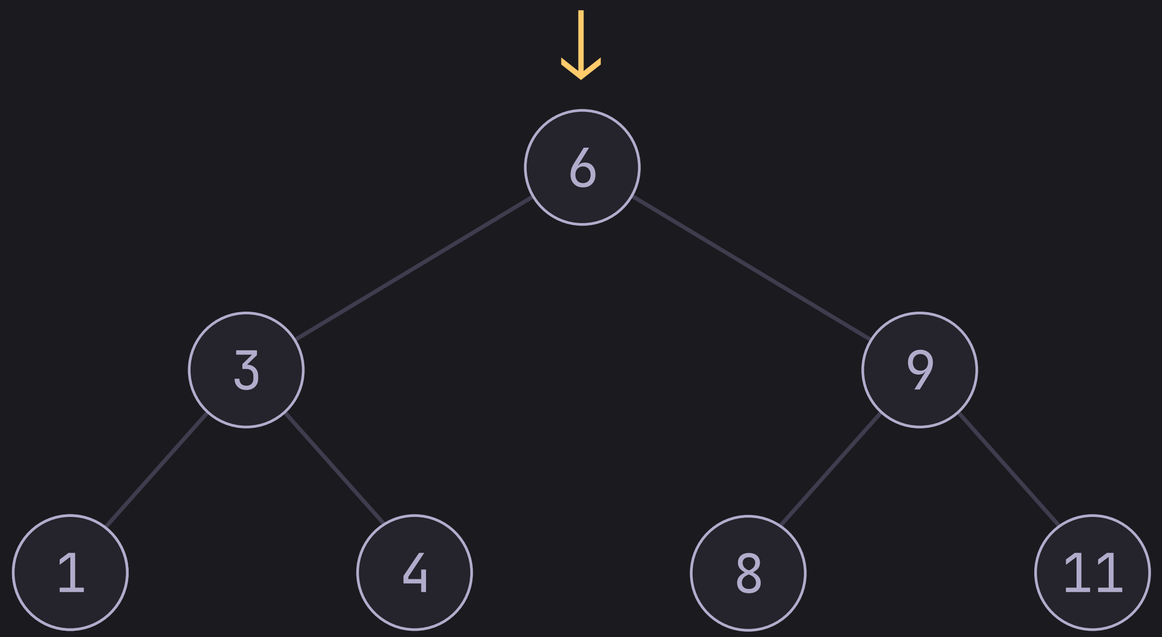
```
insert(7)
```



So we have found the place to insert the node

Binary Search Tree

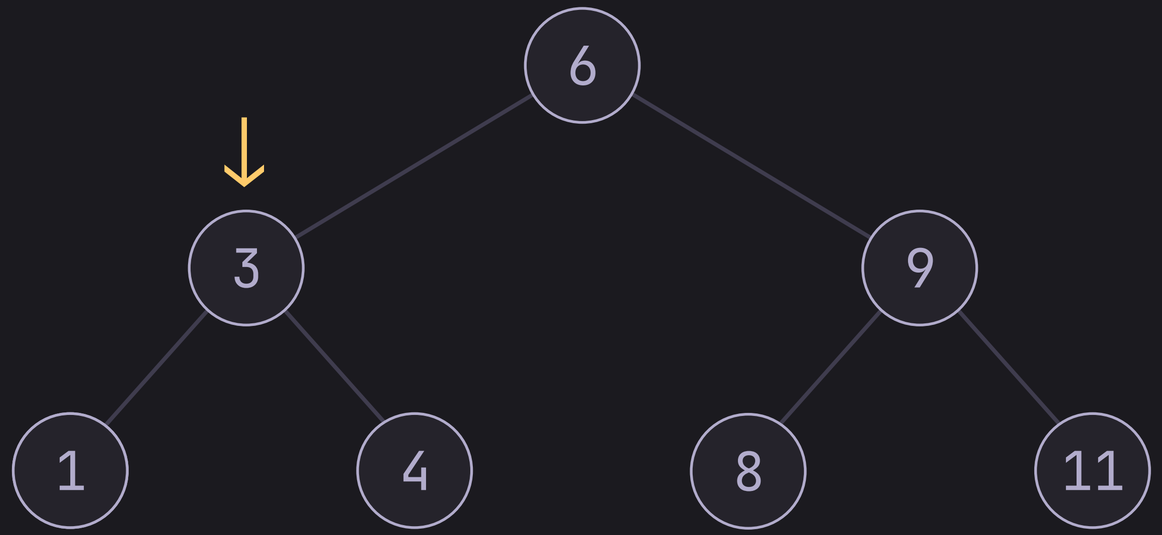
```
find(4)
```



To find a node by its key we again start at the top

Binary Search Tree

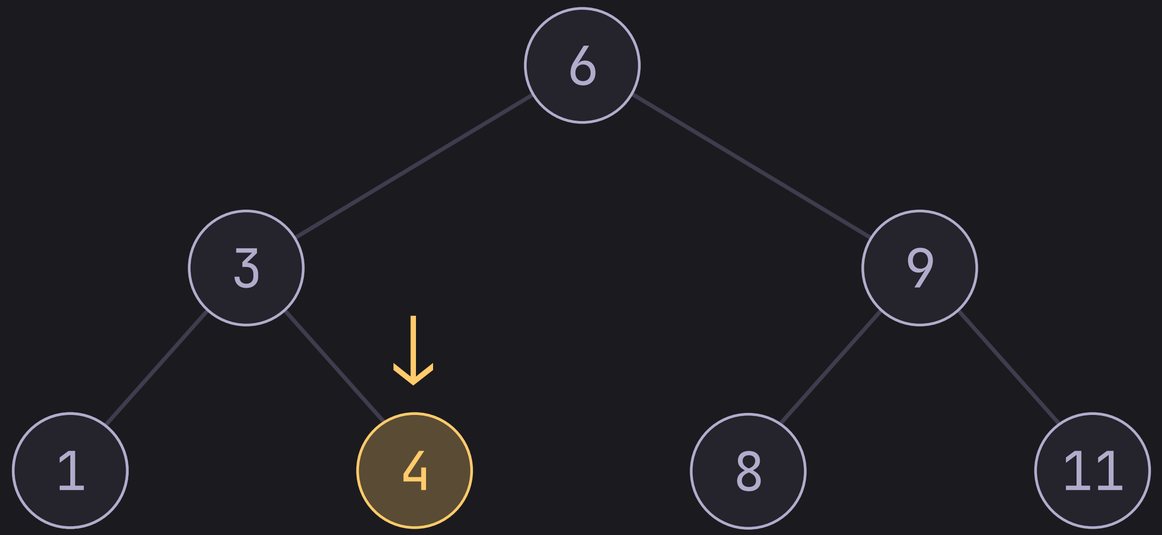
```
find(4)
```



If the key is greater than the current node go right, otherwise go left

Binary Search Tree

```
find(4)
```

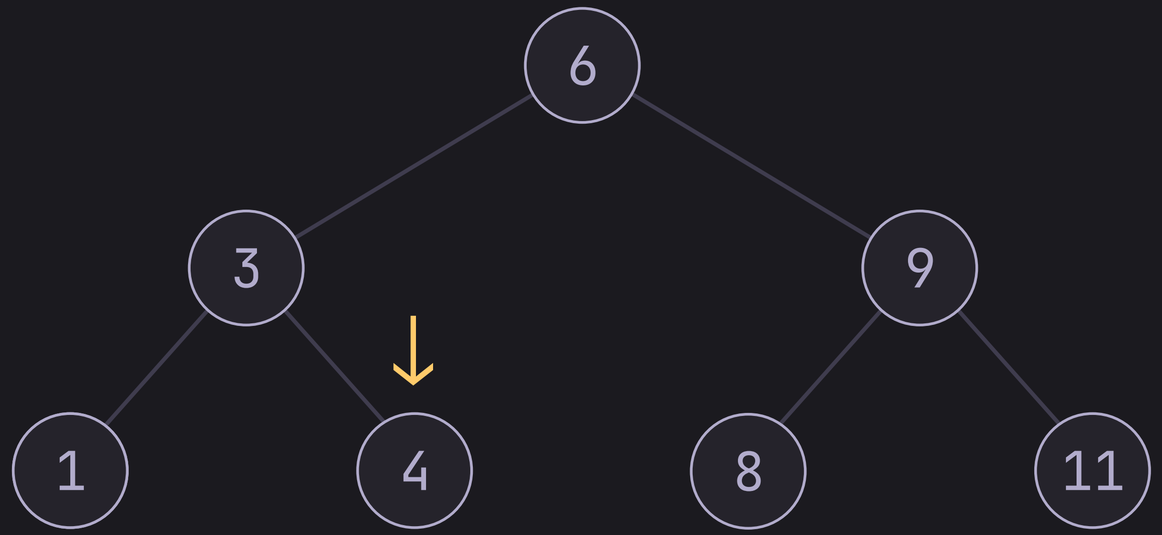


Bingo!

If the key is equal to current node then we have found it

Binary Search Tree

```
find(5)
```

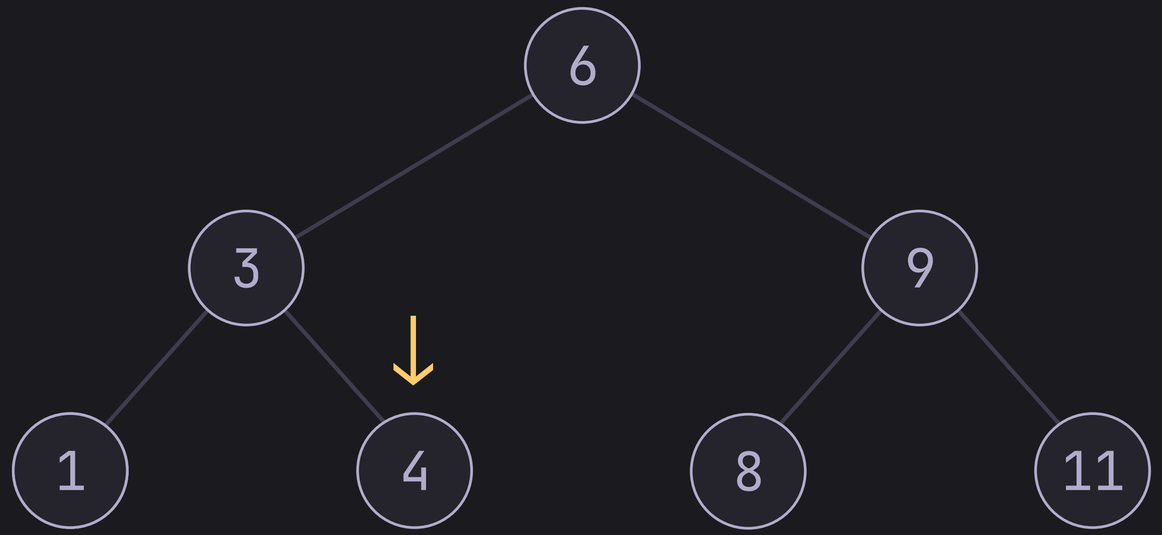


But say we were looking for 5 instead

Then we would need to go right

Binary Search Tree

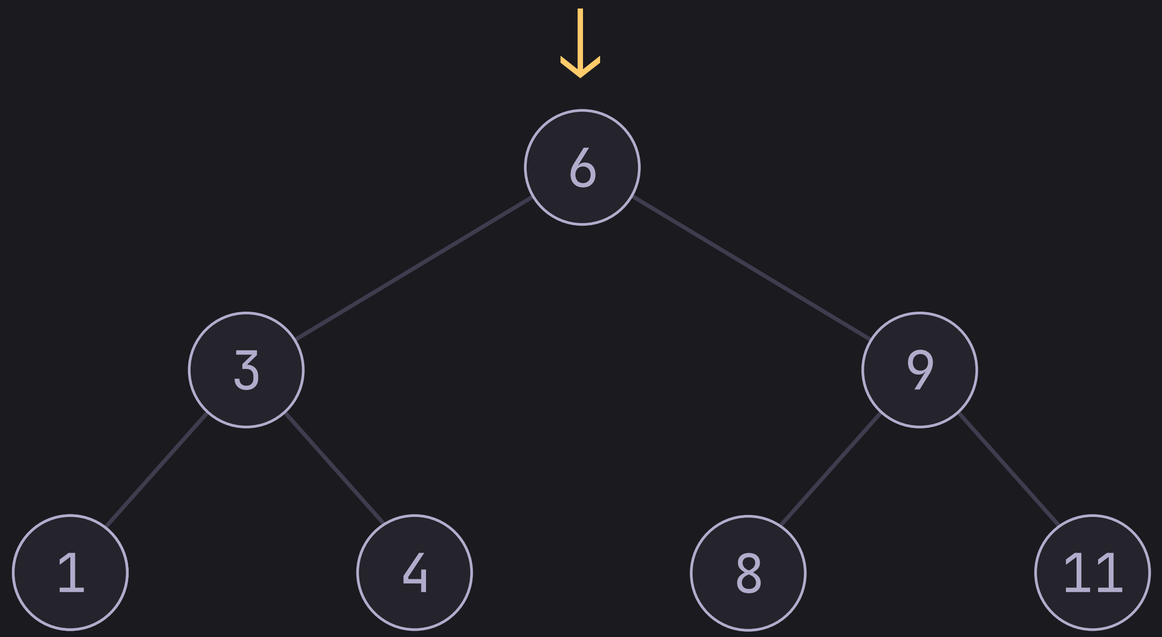
```
find(5)
```



But the right child is null!

So this tells us that 5 is not in the BST

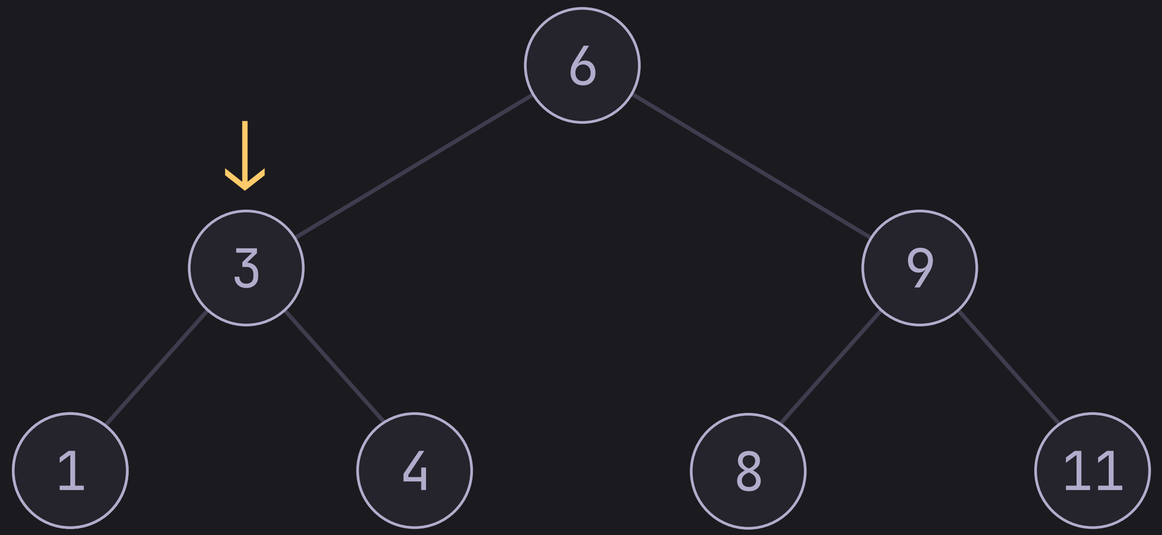
Binary Search Tree



`min()`

Again we start at the root

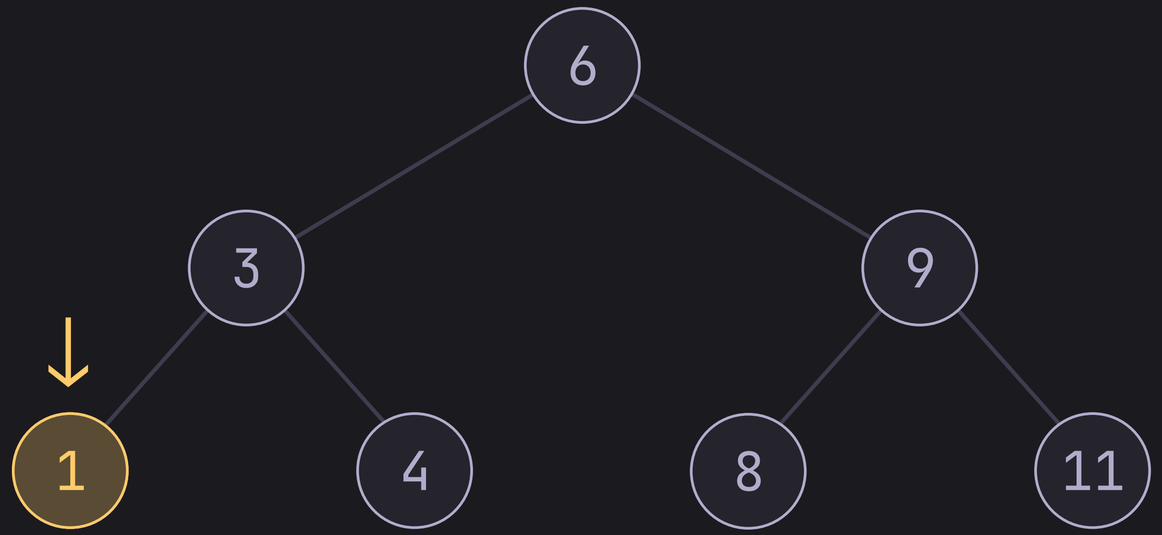
Binary Search Tree



`min()`

But now we just always go left

Binary Search Tree

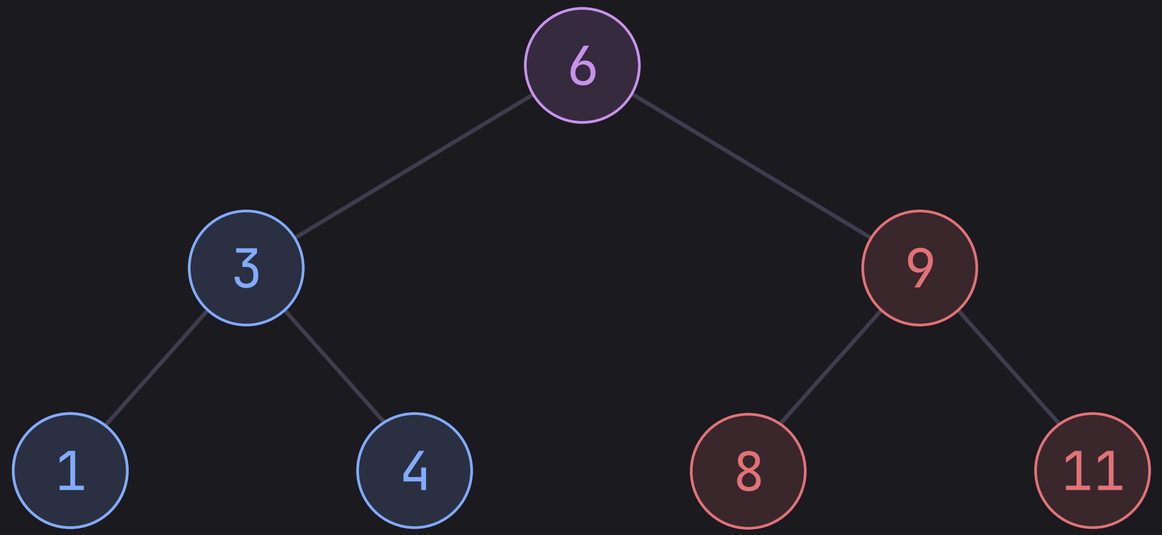


`min()`

Once we can’t go left anymore we have found the min

Binary Search Tree

Traversal

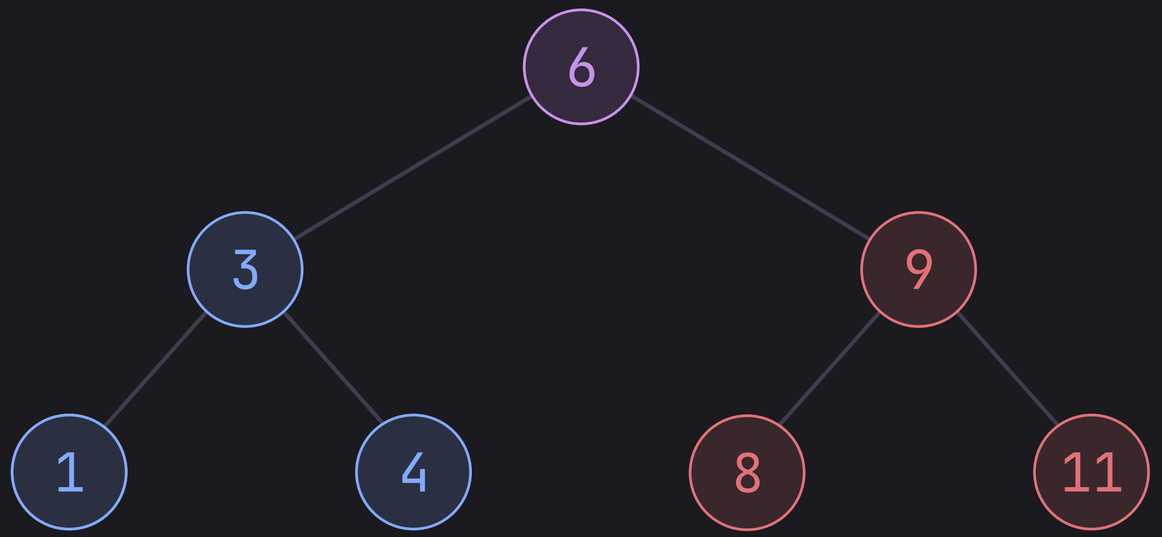


Starting at the root if we visit the:

- left child we traverse the left subtree
- right child we traverse the right subtree

Binary Search Tree

Traversal



Preorder

root → left → right

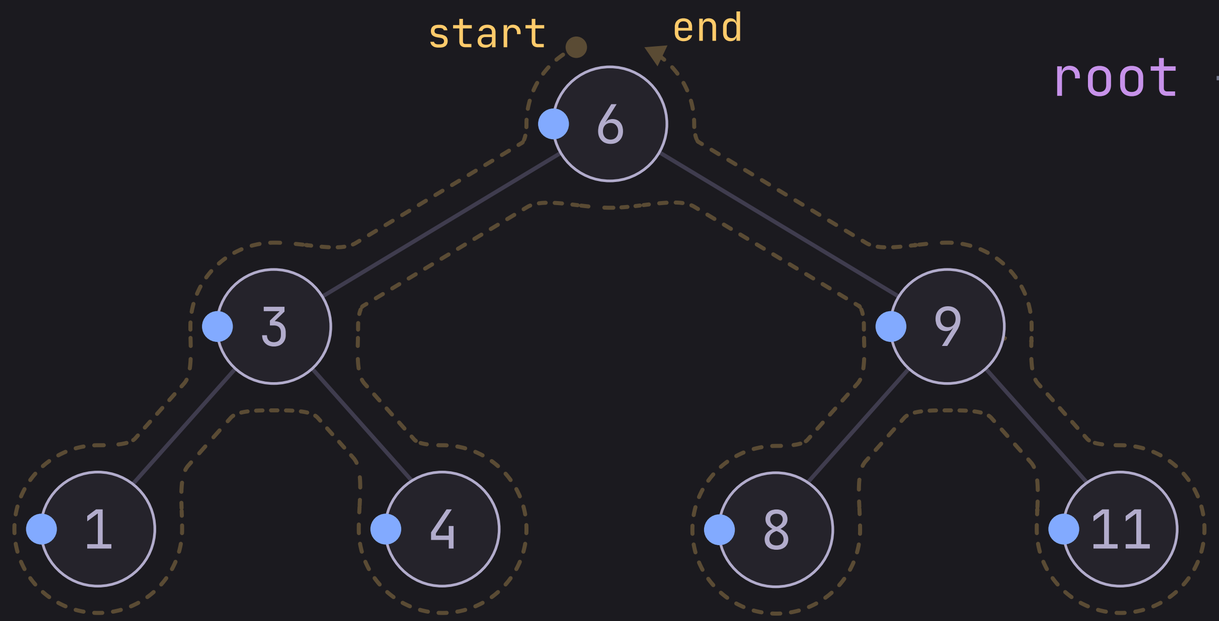
Inorder

left → root → right

Postorder

left → right → root

Binary Search Tree

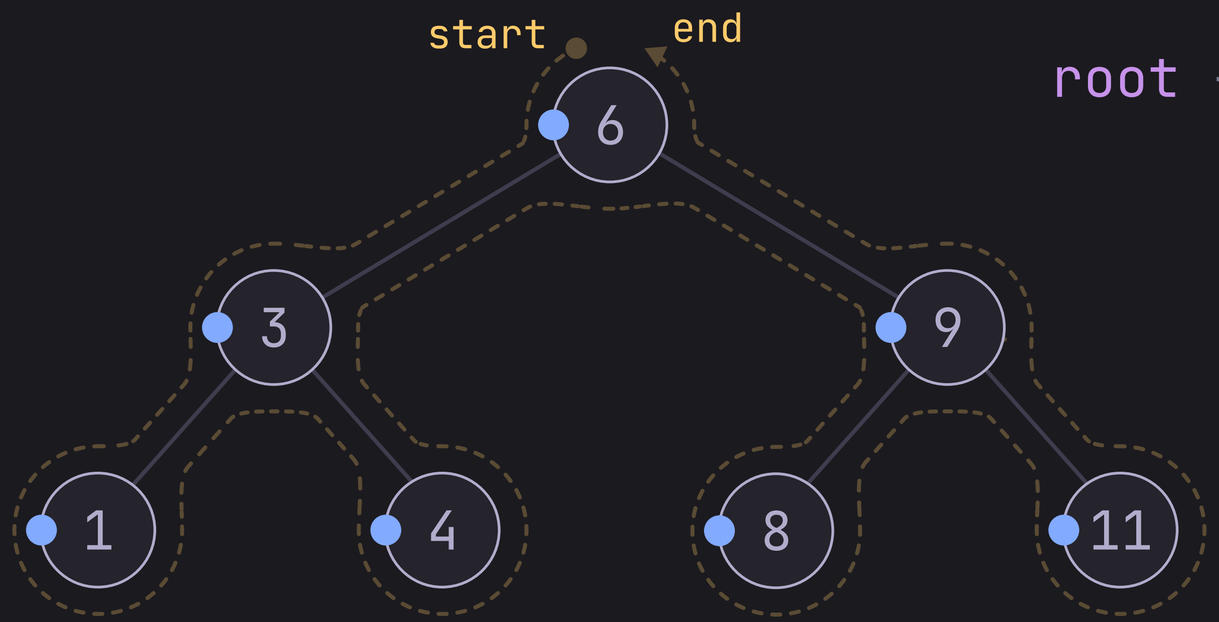


root → left → right

Preorder

To do a preorder traversal walk along the yellow line and write down node each time you encounter a blue dot

Binary Search Tree

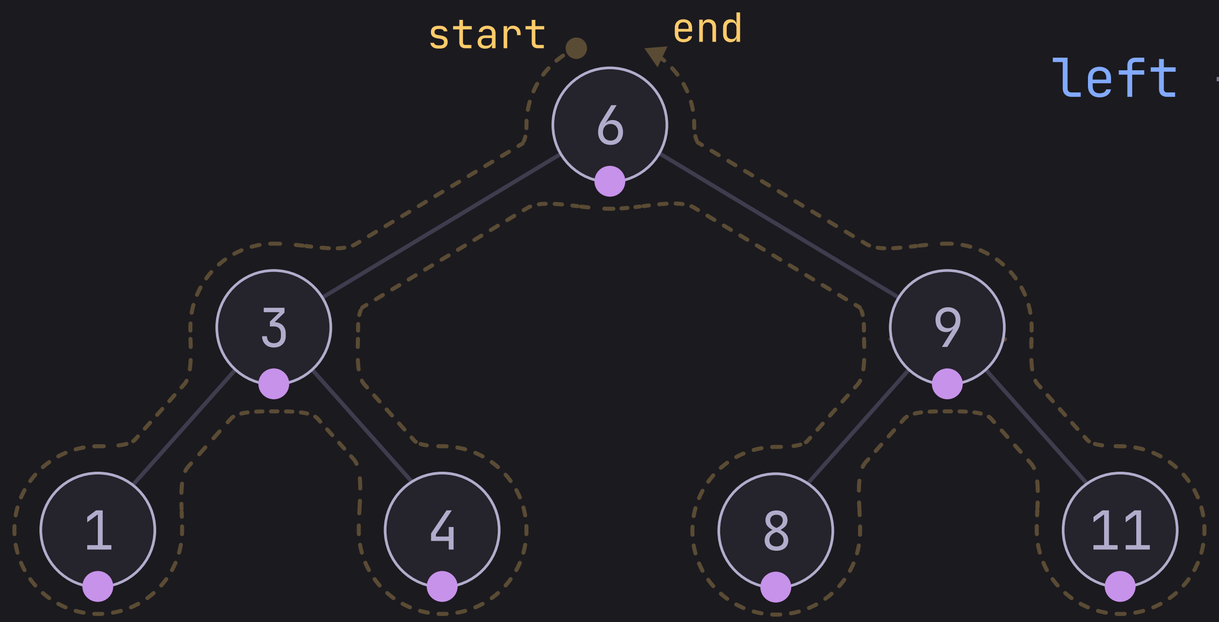


root → left → right

Preorder

[6, 3, 1, 4, 9, 8, 11]

Binary Search Tree

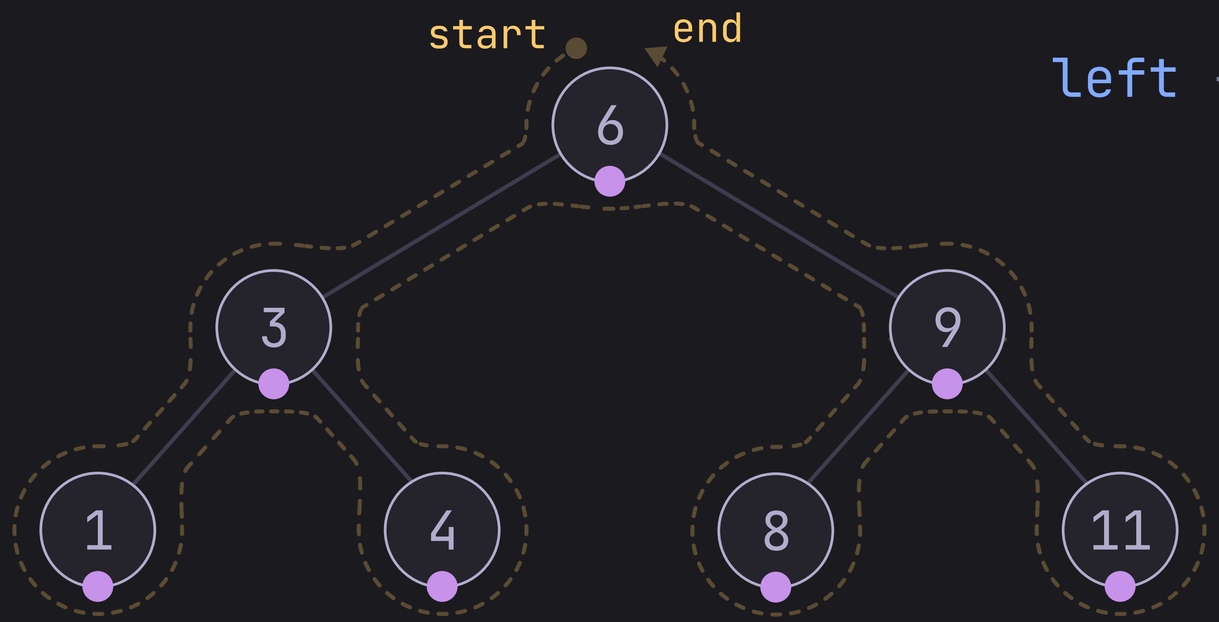


left → root → right

Inorder

To do an inorder traversal walk along the yellow line and write down node each time you encounter a purple dot

Binary Search Tree

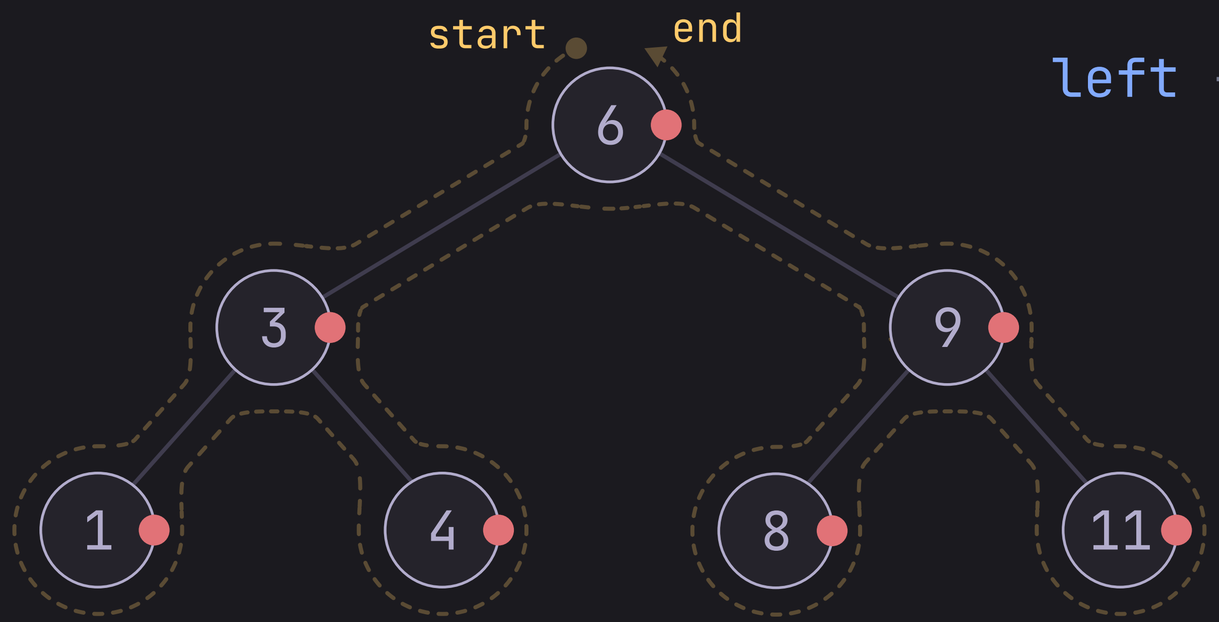


left → root → right

Inorder

[1, 3, 4, 6, 8, 9, 11]

Binary Search Tree

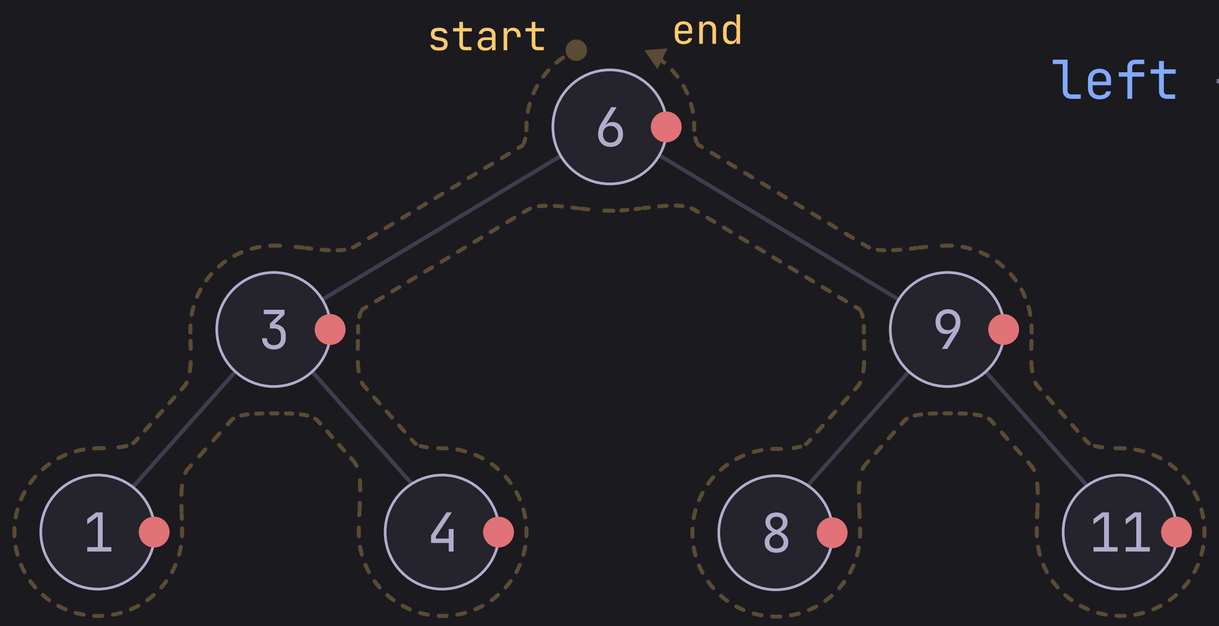


left → right → root

Postorder

To do an postorder traversal walk along the yellow line and write down node each time you encounter a red dot

Binary Search Tree

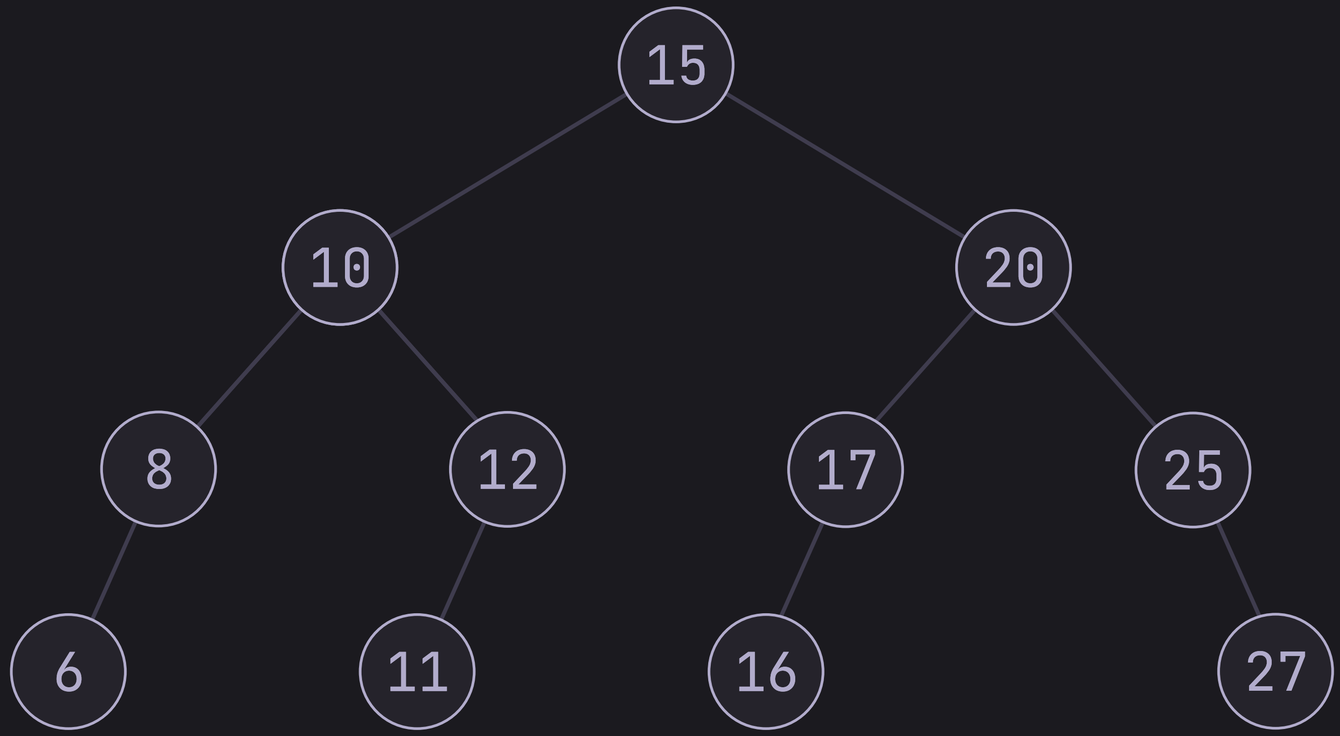


left → right → root

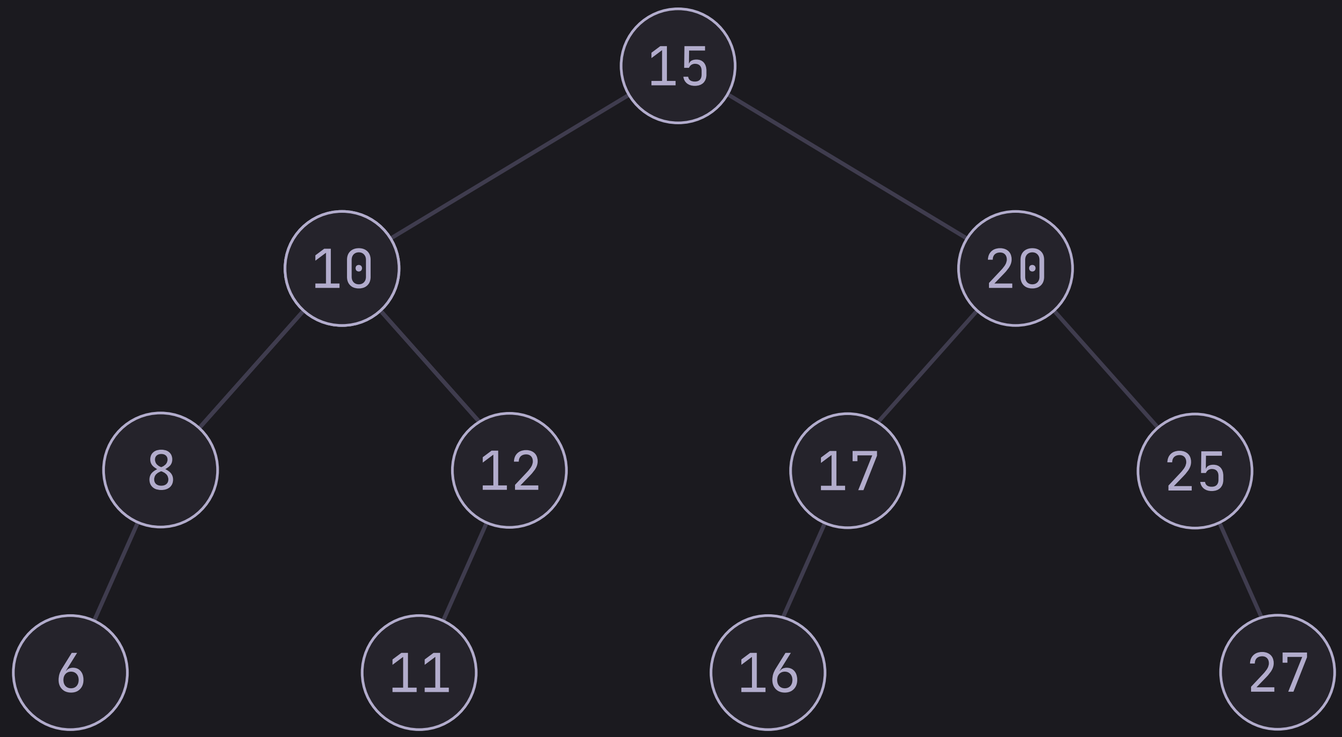
Postorder

[1, 4, 3, 8, 11, 9, 6]

Binary Search Tree

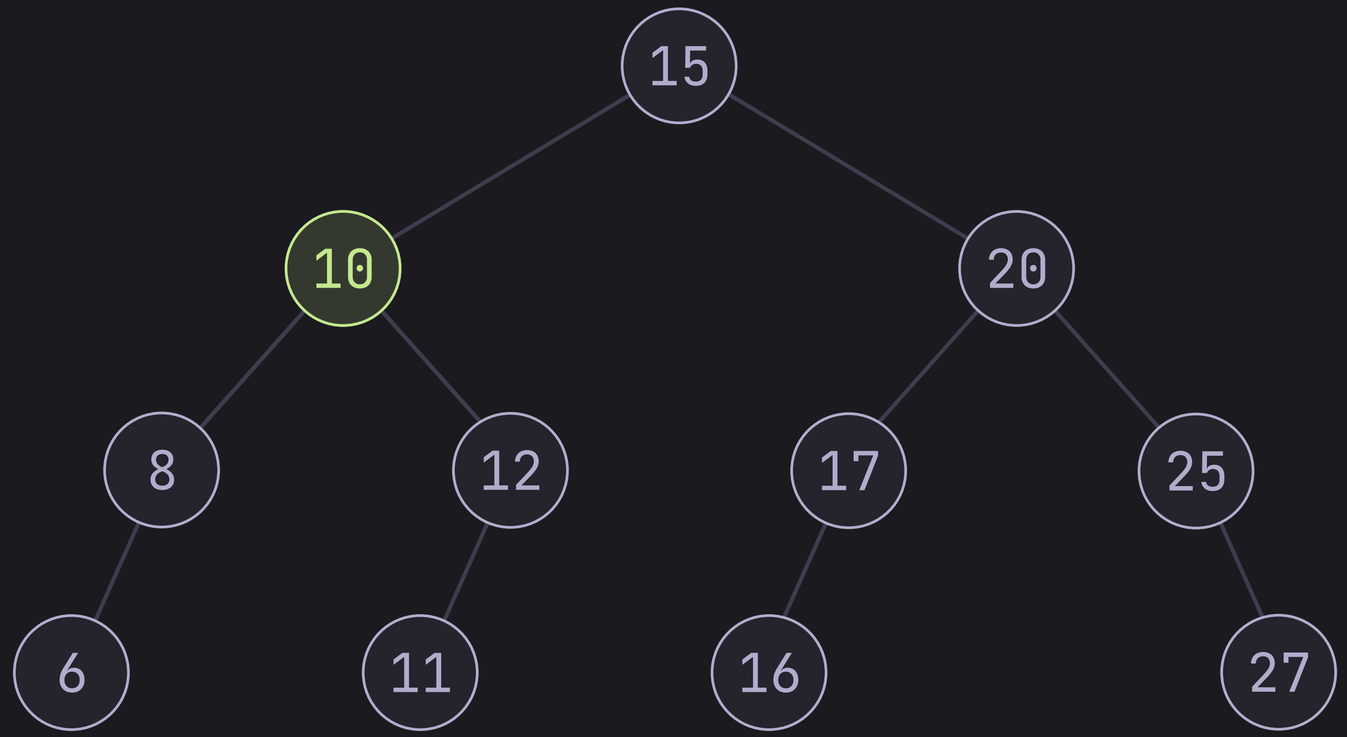


Binary Search Tree



[6 8 10 11 12 15 16 17 20 25 27]

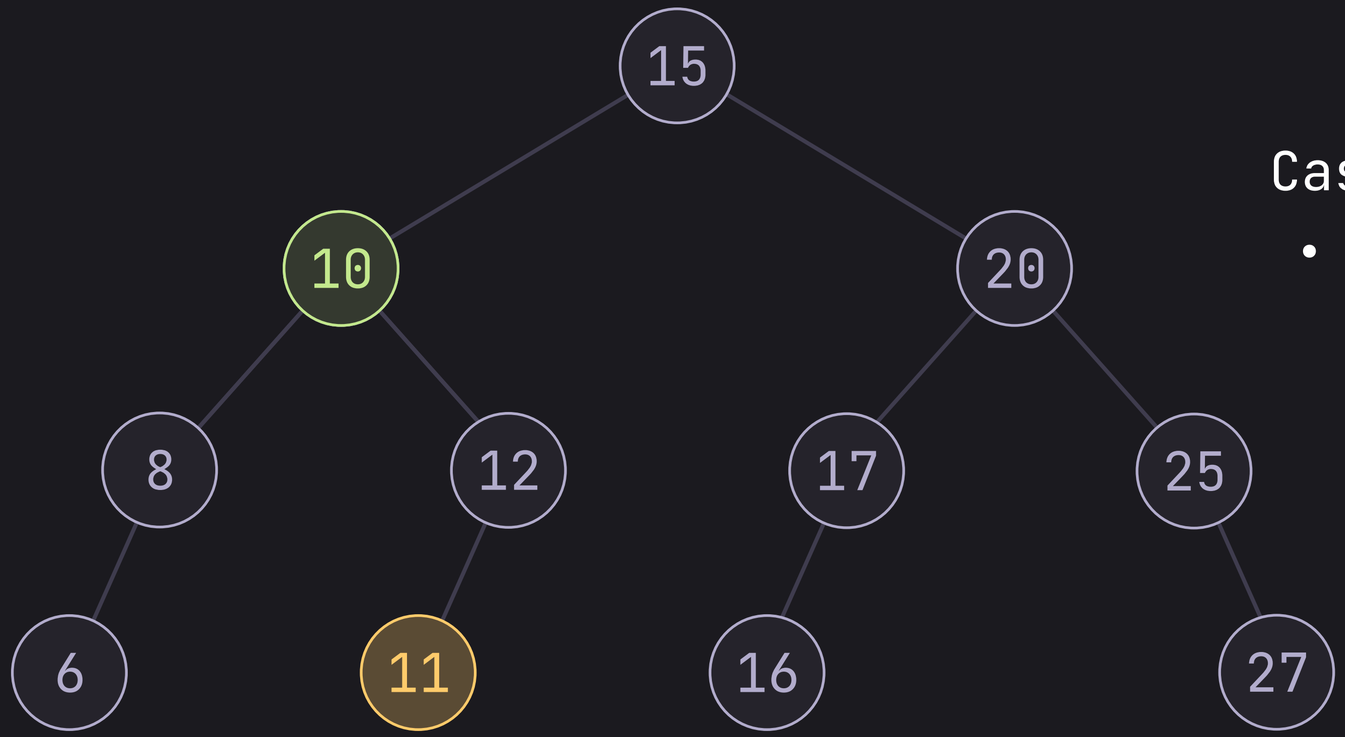
Binary Search Tree



successor(10)

[6 8 10 11 12 15 16 17 20 25 27]

Binary Search Tree



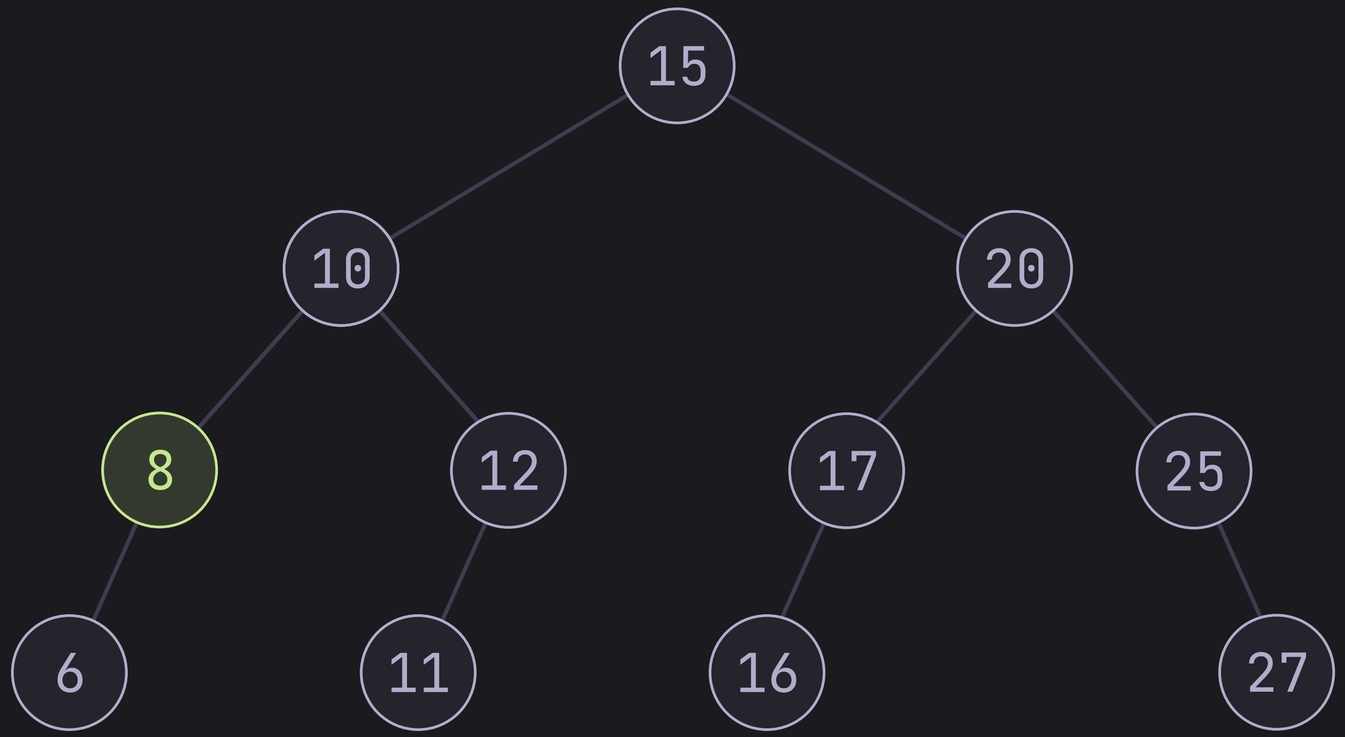
`successor(10)`

Case 1: Node has a right subtree

- Find min of right subtree

[6 8 10 11 12 15 16 17 20 25 27]

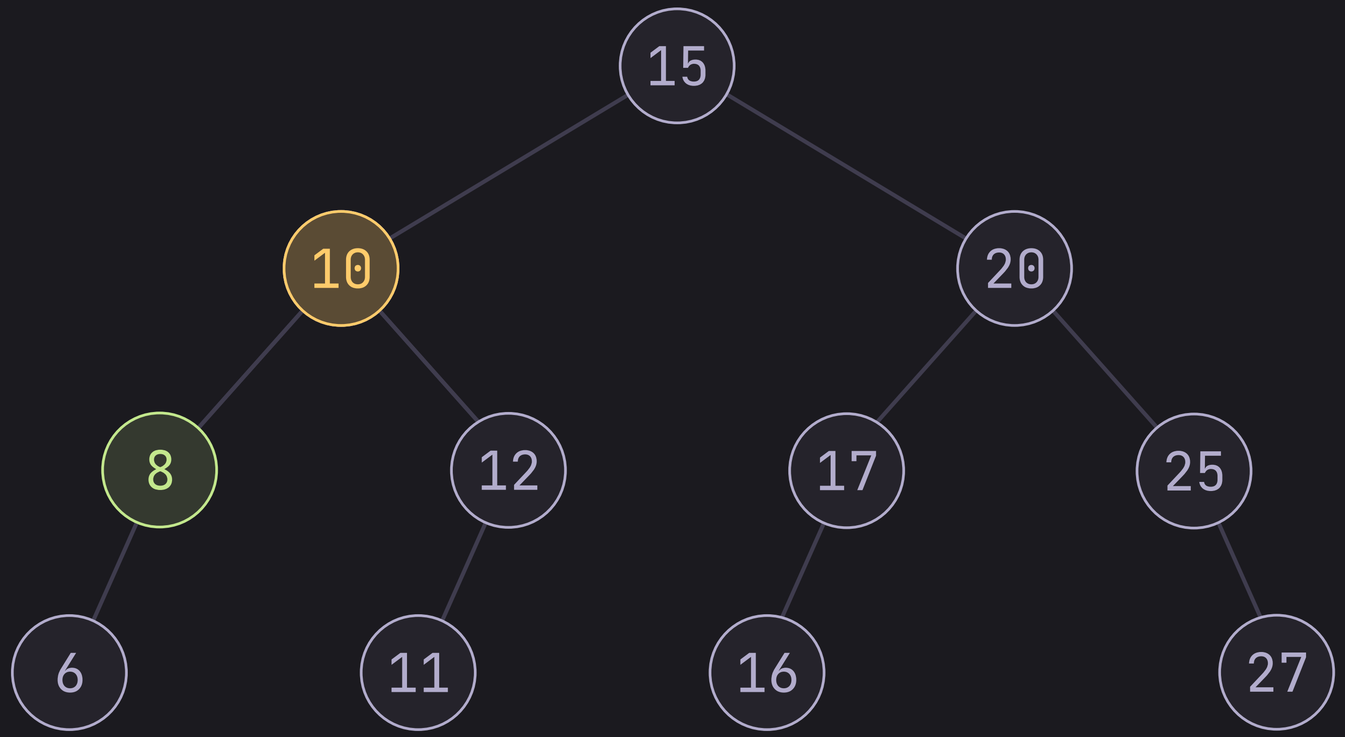
Binary Search Tree



successor(8)

[6 8 10 11 12 15 16 17 20 25 27]

Binary Search Tree



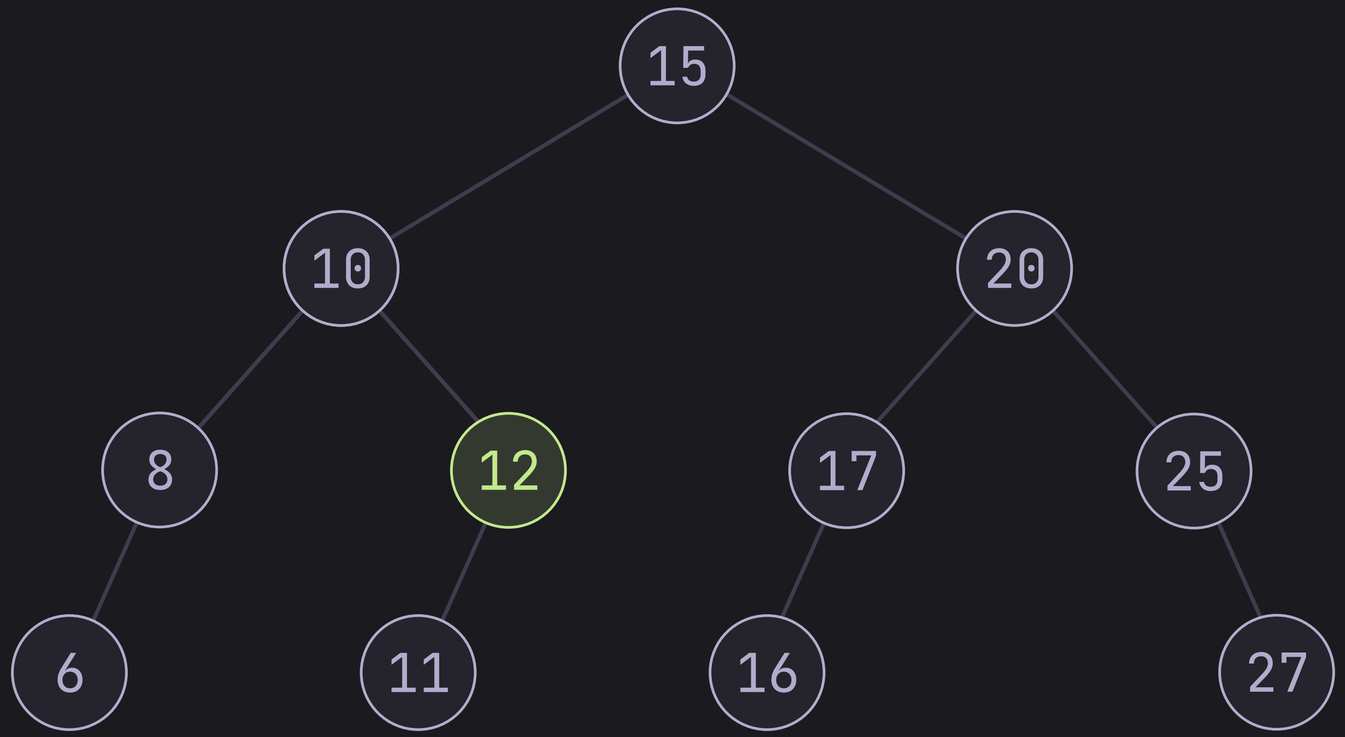
[6 8 10 11 12 15 16 17 20 25 27]

successor(8)

Case 2: Node is left child

- Return the parent

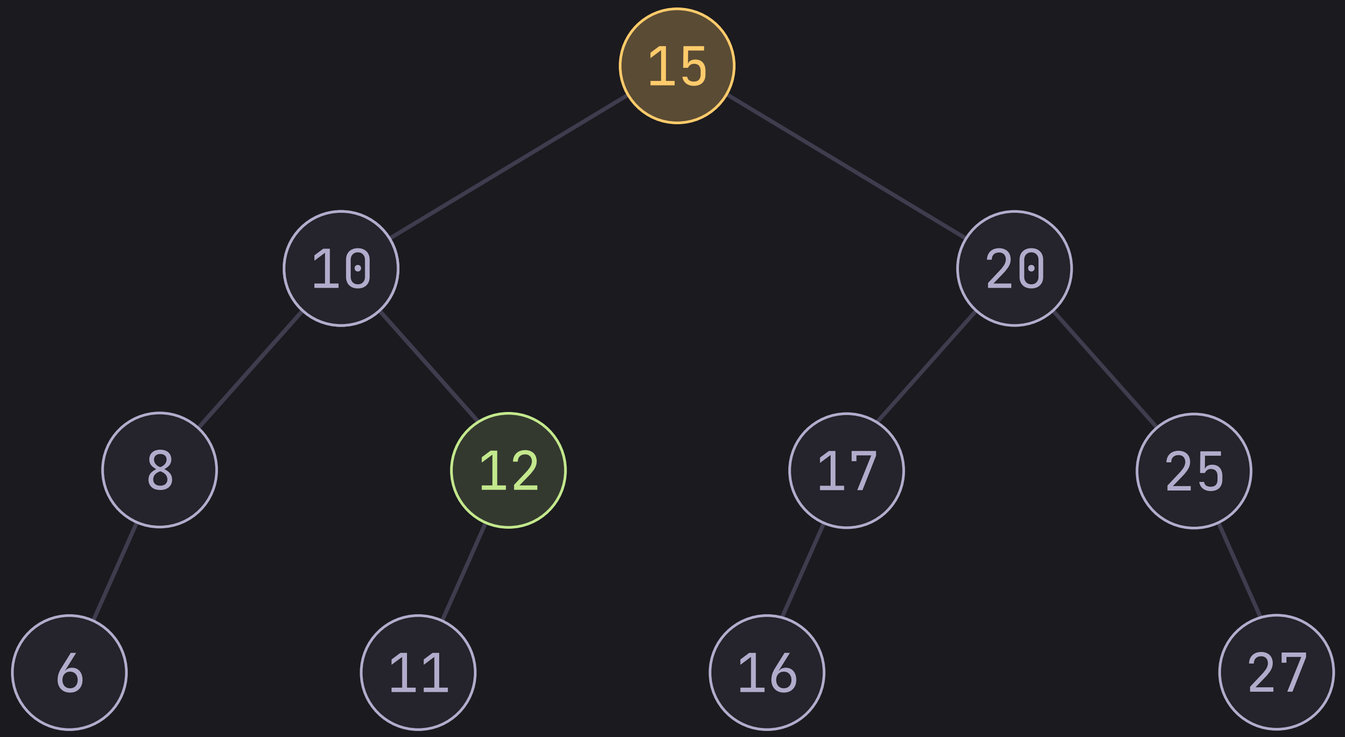
Binary Search Tree



successor(12)

[6 8 10 11 12 15 16 17 20 25 27]

Binary Search Tree



[6 8 10 11 12 15 16 17 20 25 27]

successor(12)

Case 3: Node is right child

- go to nearest ancestor for which the given node is in the left subtree