CSCI 232:
Data Structures and Algorithms

Binary Search Trees (BST)

Reese Pearsall

Spring 2024

Announcements

Lab 3 due **tomorrow** at 11:59

Program 1 will be posted very soon

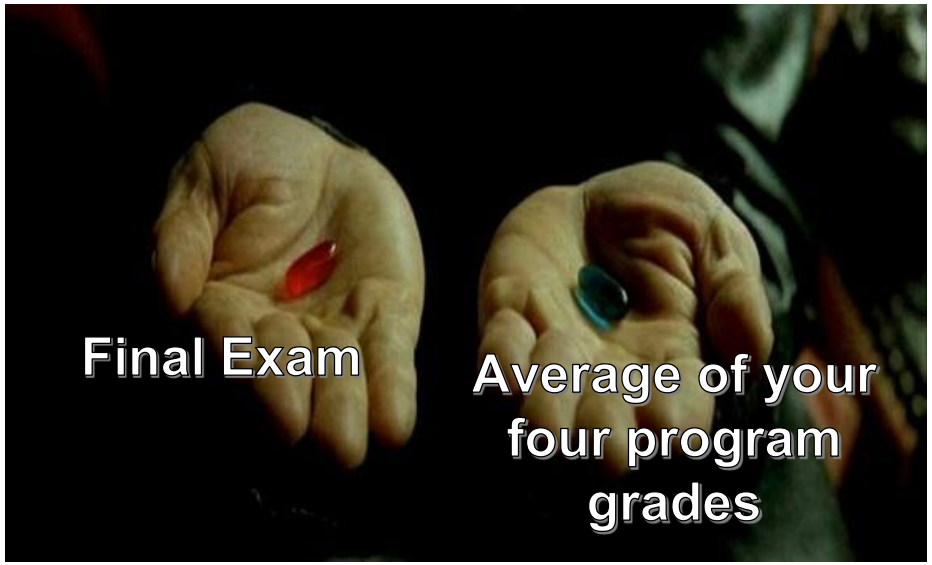
Final Exam

The final exam for 232 will be **optional**.

You can either:

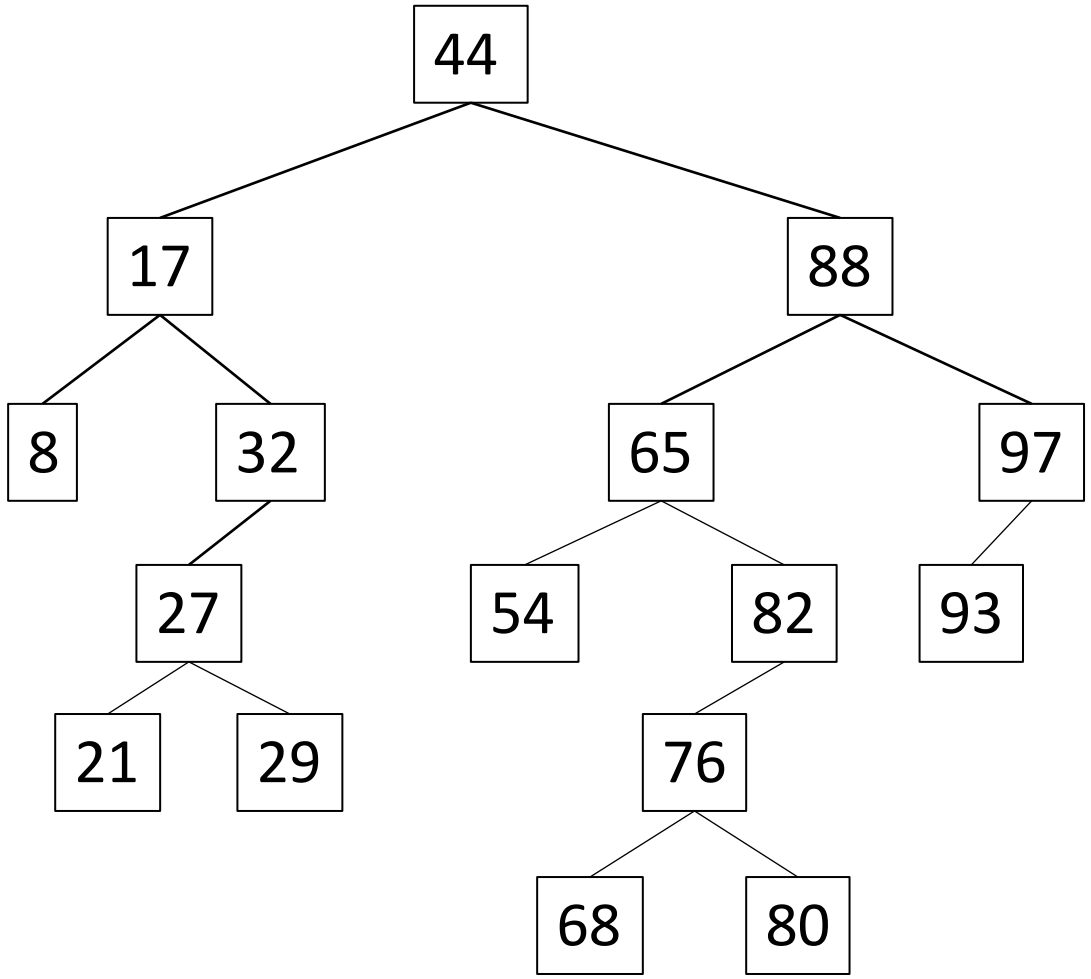
1. Show up and take the final exam normally

2. Don't show up, and your final exam grade will be the average of your four program grades



(If you show up, and you still do worse than the average of your four exam grades, I will give you the average of your program grades)

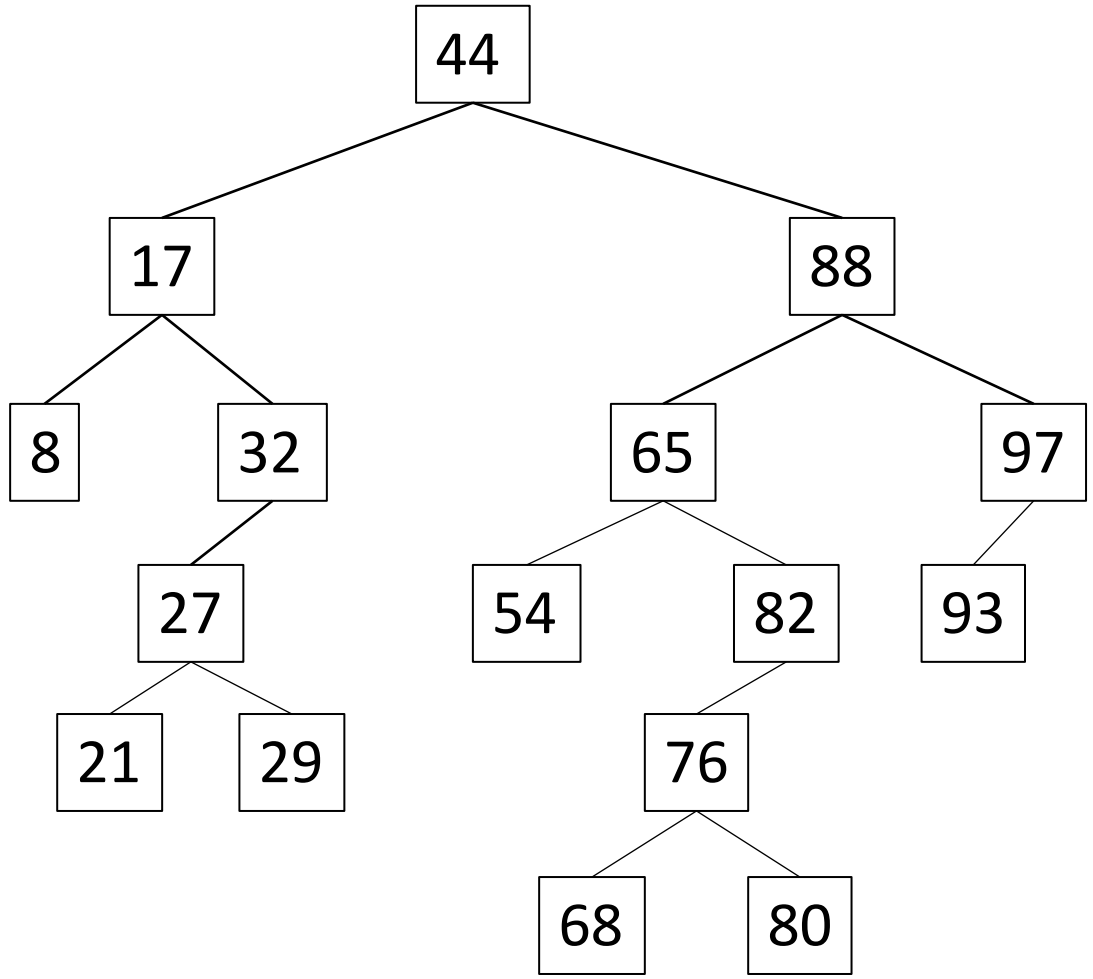
Binary Search Tree



Binary Search Tree

Binary Search Tree (BST) properties:

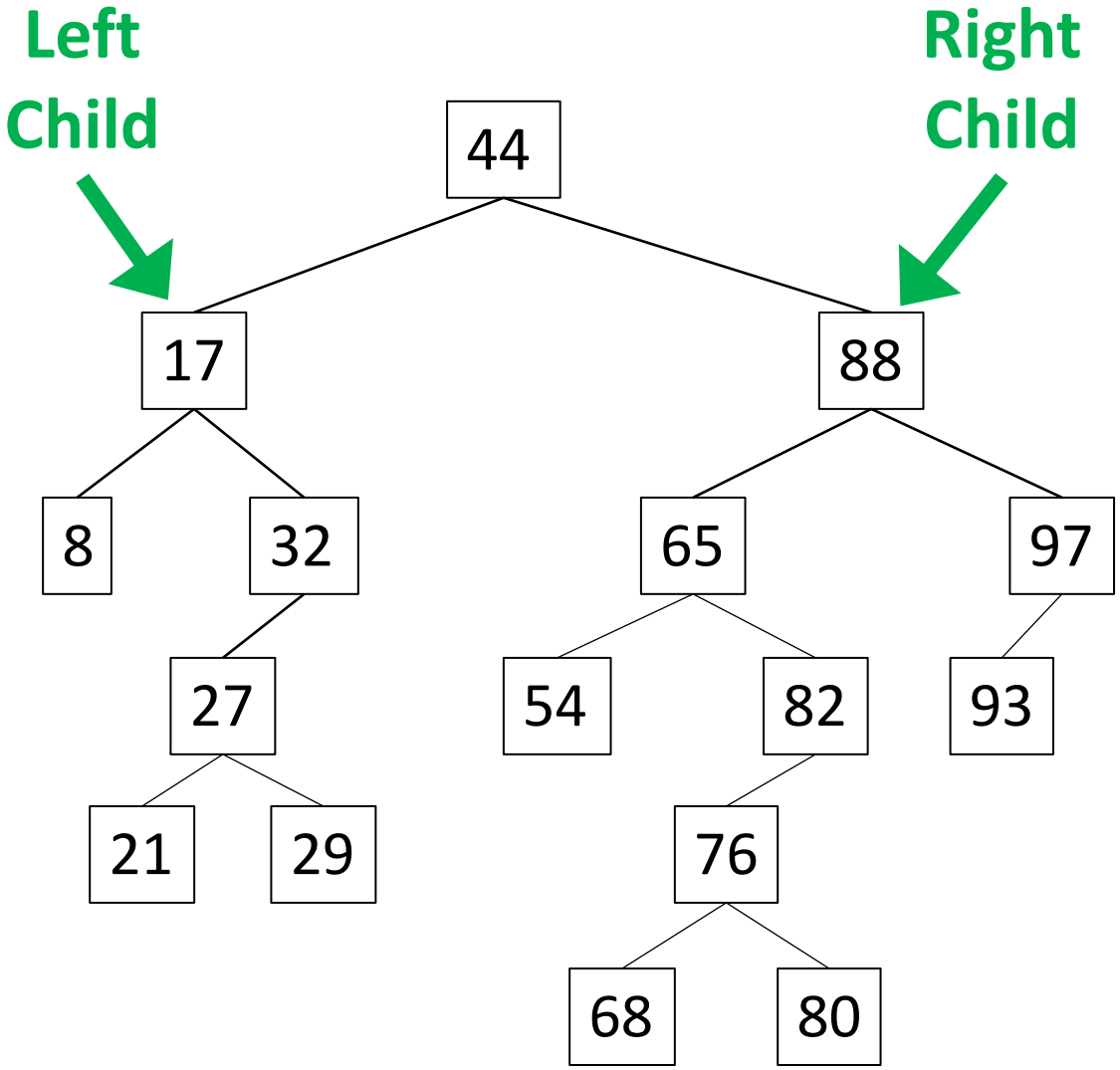
- A BST is composed of `Comparable` data elements.



Binary Search Tree

Binary Search Tree (BST) properties:

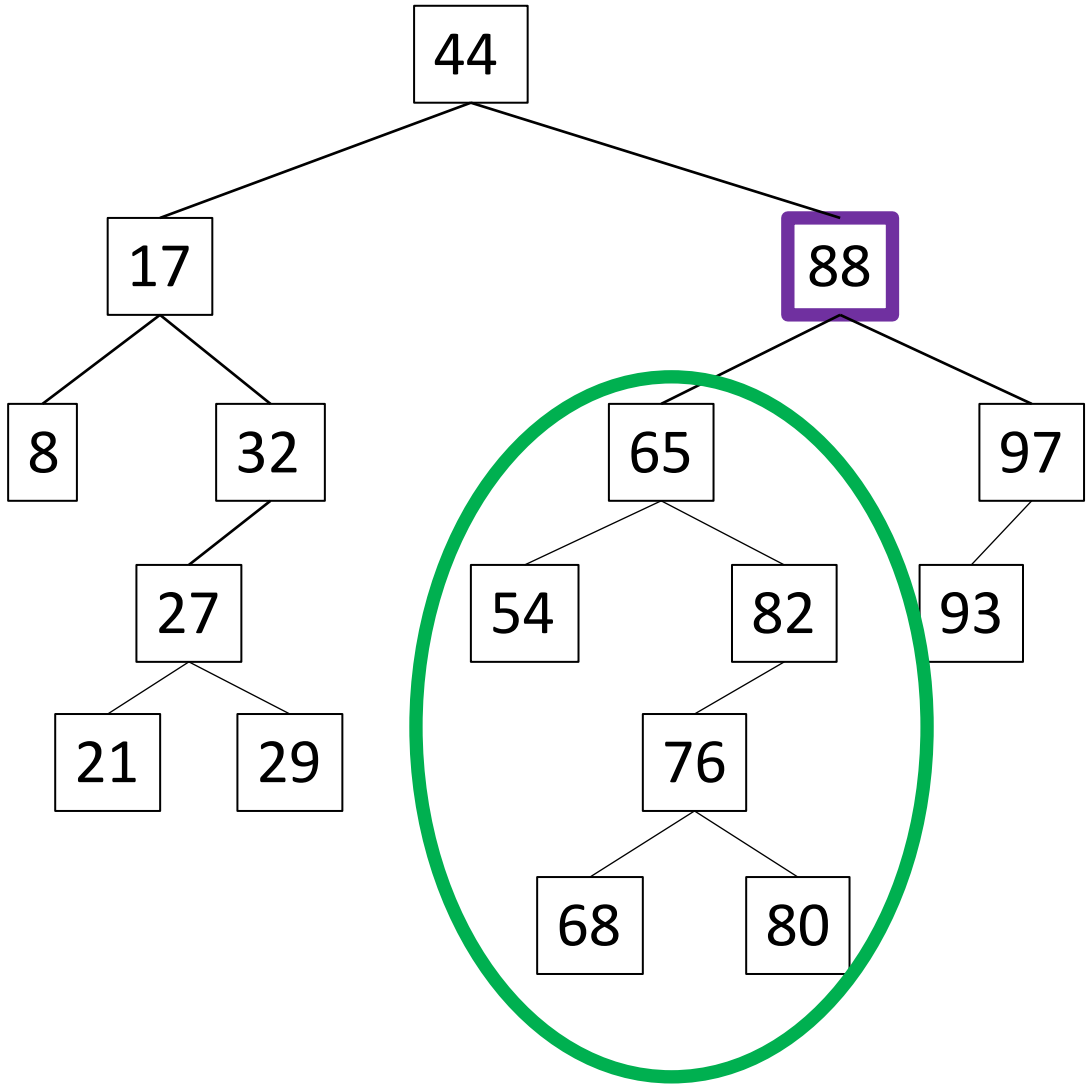
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).



Binary Search Tree

Binary Search Tree (BST) properties:

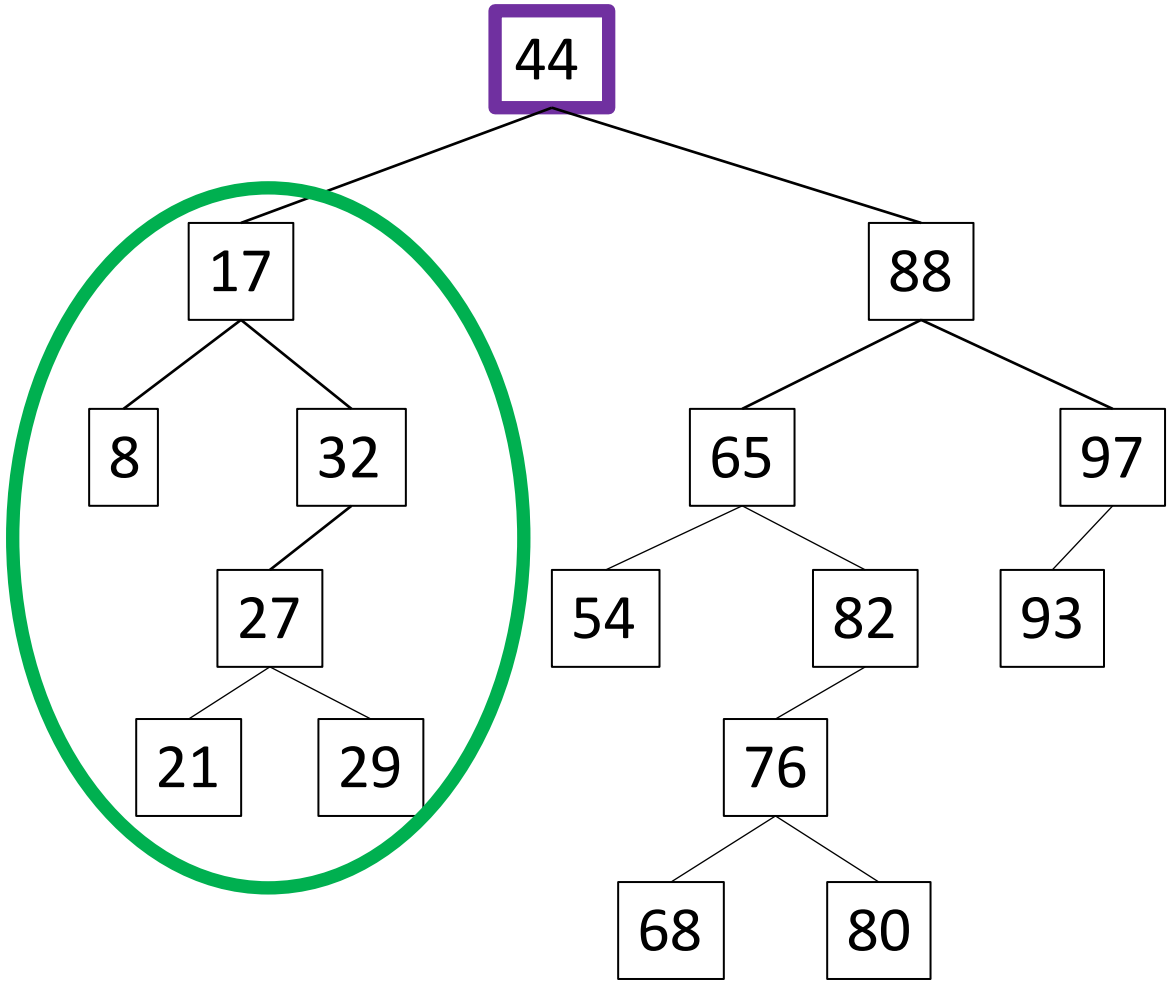
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).
- For each **node**, all **left-hand descendants** have values that are less that the node.



Binary Search Tree

Binary Search Tree (BST) properties:

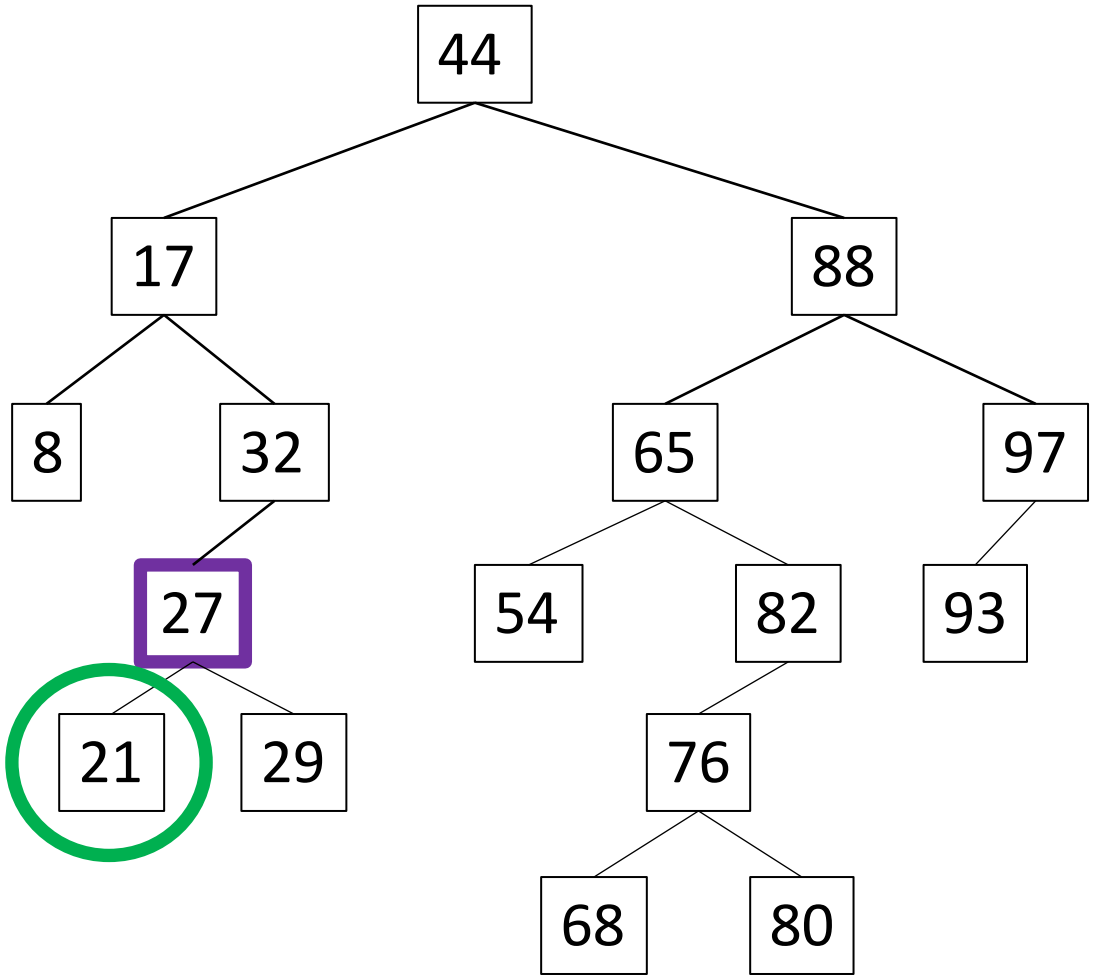
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).
- For each **node**, all **left-hand descendants** have values that are less that the node.



Binary Search Tree

Binary Search Tree (BST) properties:

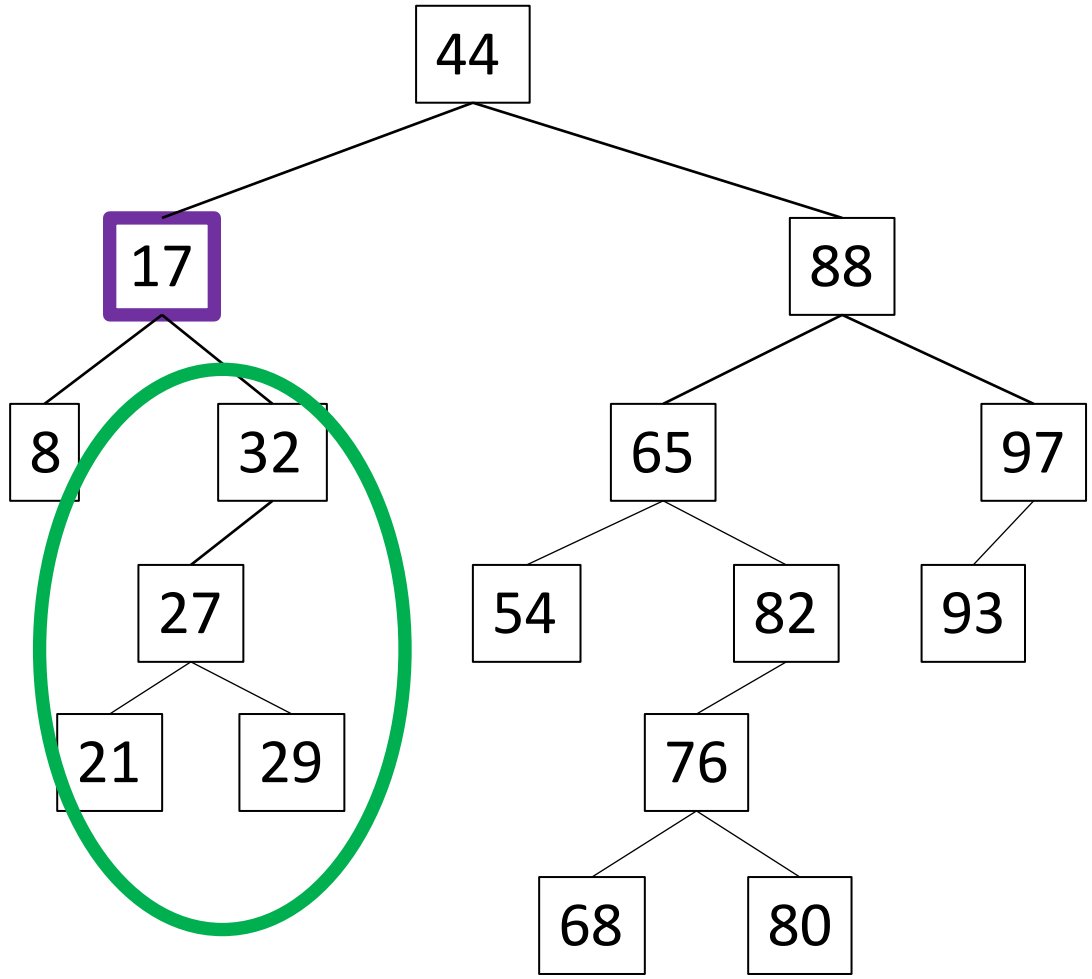
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).
- For each **node**, all **left-hand descendants** have values that are less that the node.



Binary Search Tree

Binary Search Tree (BST) properties:

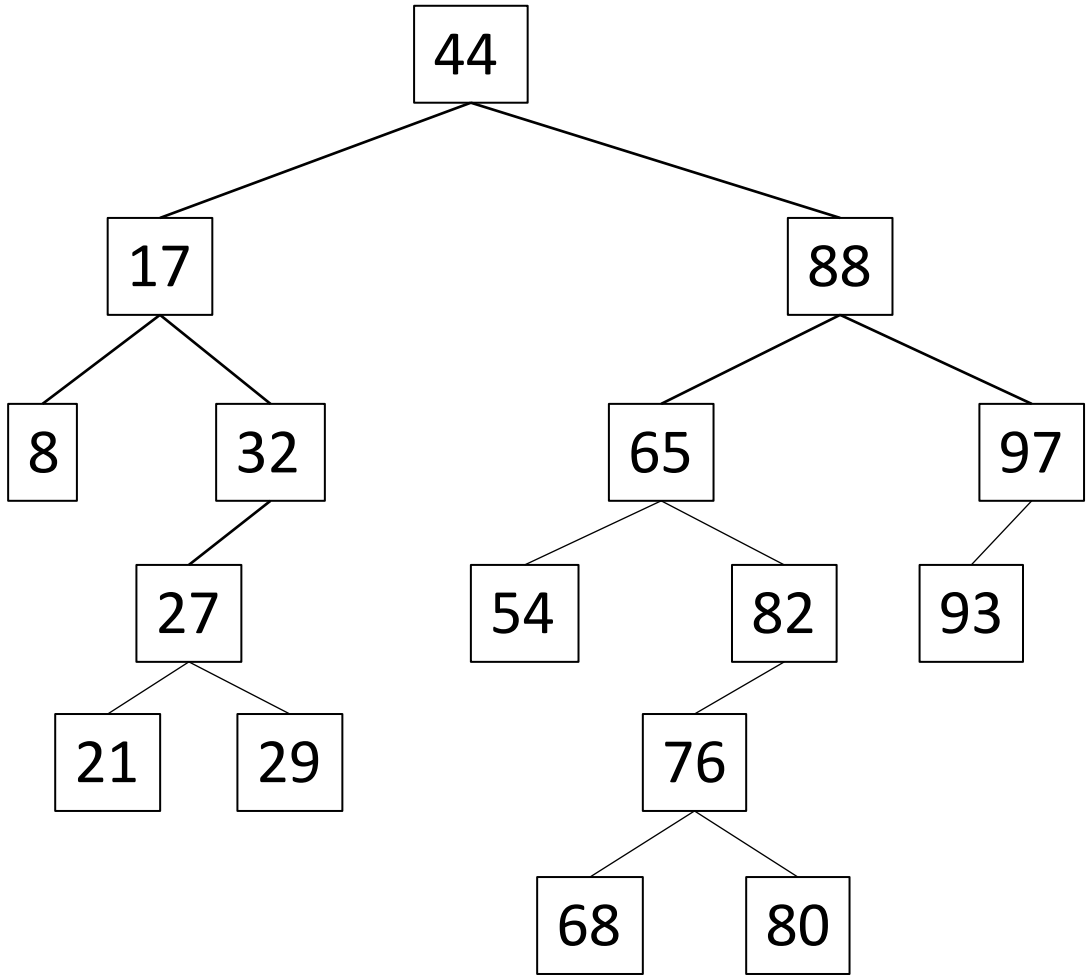
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).
- For each node, all left-hand descendants have values that are less that the node.
- For each node, all right-hand descendants have values that are larger than the node.



Binary Search Tree

Binary Search Tree (BST) properties:

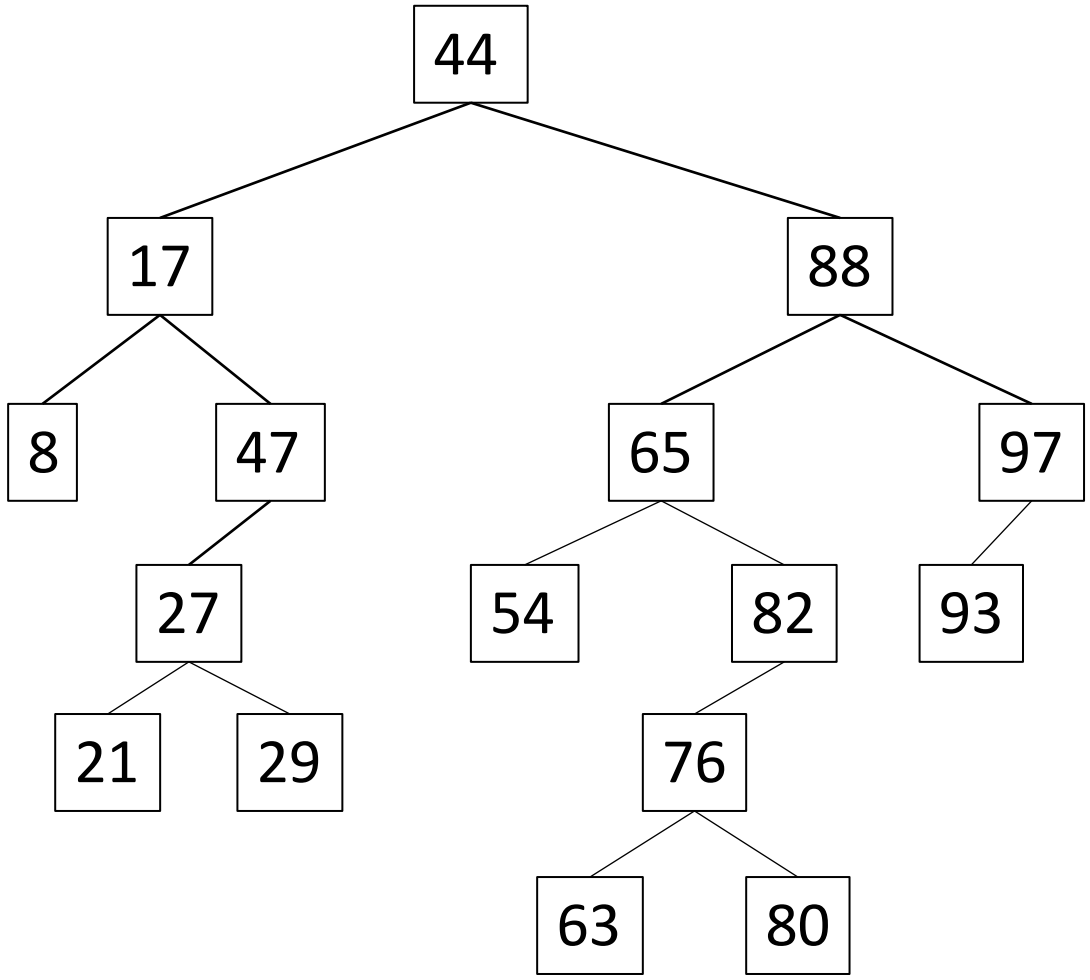
- A BST is composed of `Comparable` data elements.
- A BST is a binary tree (each node has at most two children).
- For each node, all left-hand descendants have values that are less that the node.
- For each node, all right-hand descendants have values that are larger than the node.
- There are no duplicate values (definitions vary).



Binary Search Tree

Binary Search Tree (BST) properties:

- **A BST is composed of `Comparable` data elements.**
- A BST is a binary tree (each node has at most two children).
- For each node, all left-hand descendants have values that are less that the node.
- For each node, all right-hand descendants have values that are larger than the node.
- There are no duplicate values (definitions vary).

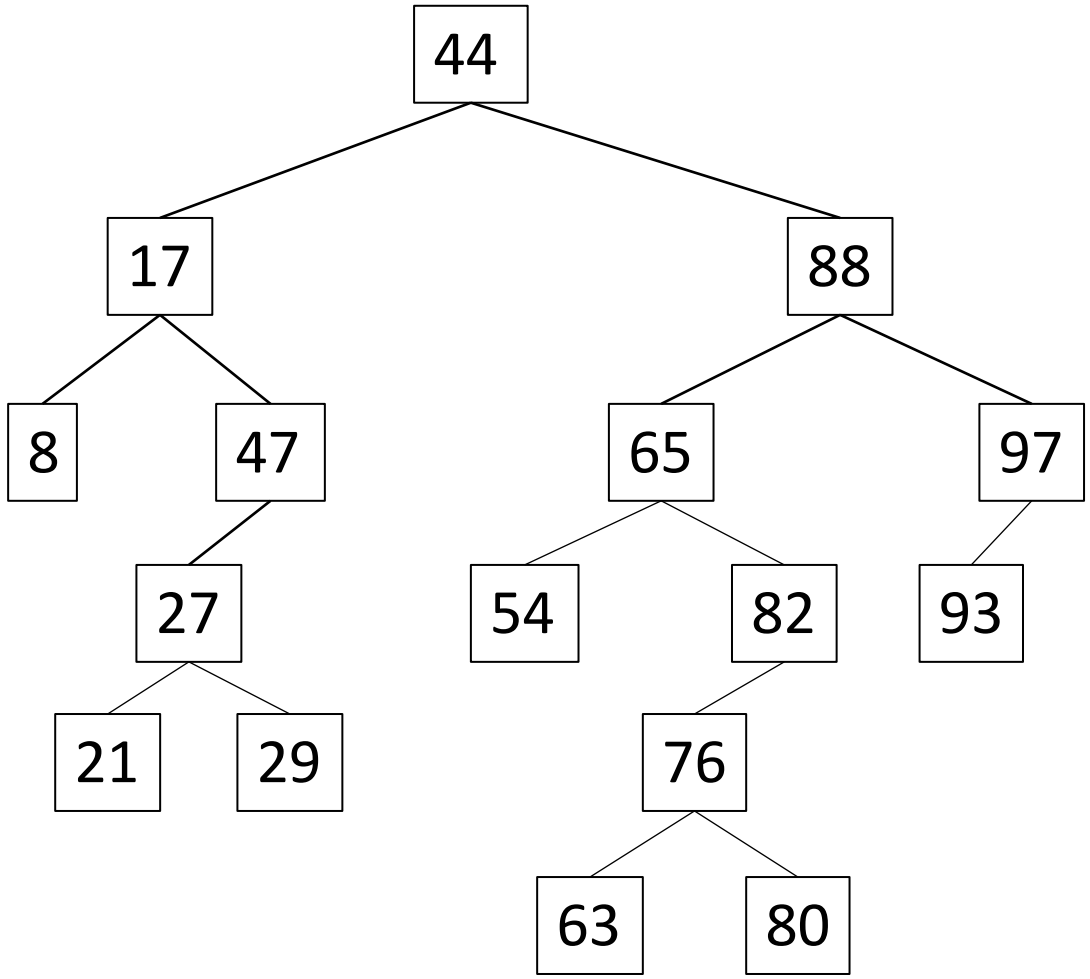


Is it a BST?

Binary Search Tree

Binary Search Tree (BST) properties:

- **A BST is composed of `Comparable` data elements.**
- **A BST is a binary tree (each node has at most two children).**
- For each node, all left-hand descendants have values that are less that the node.
- For each node, all right-hand descendants have values that are larger than the node.
- There are no duplicate values (definitions vary).

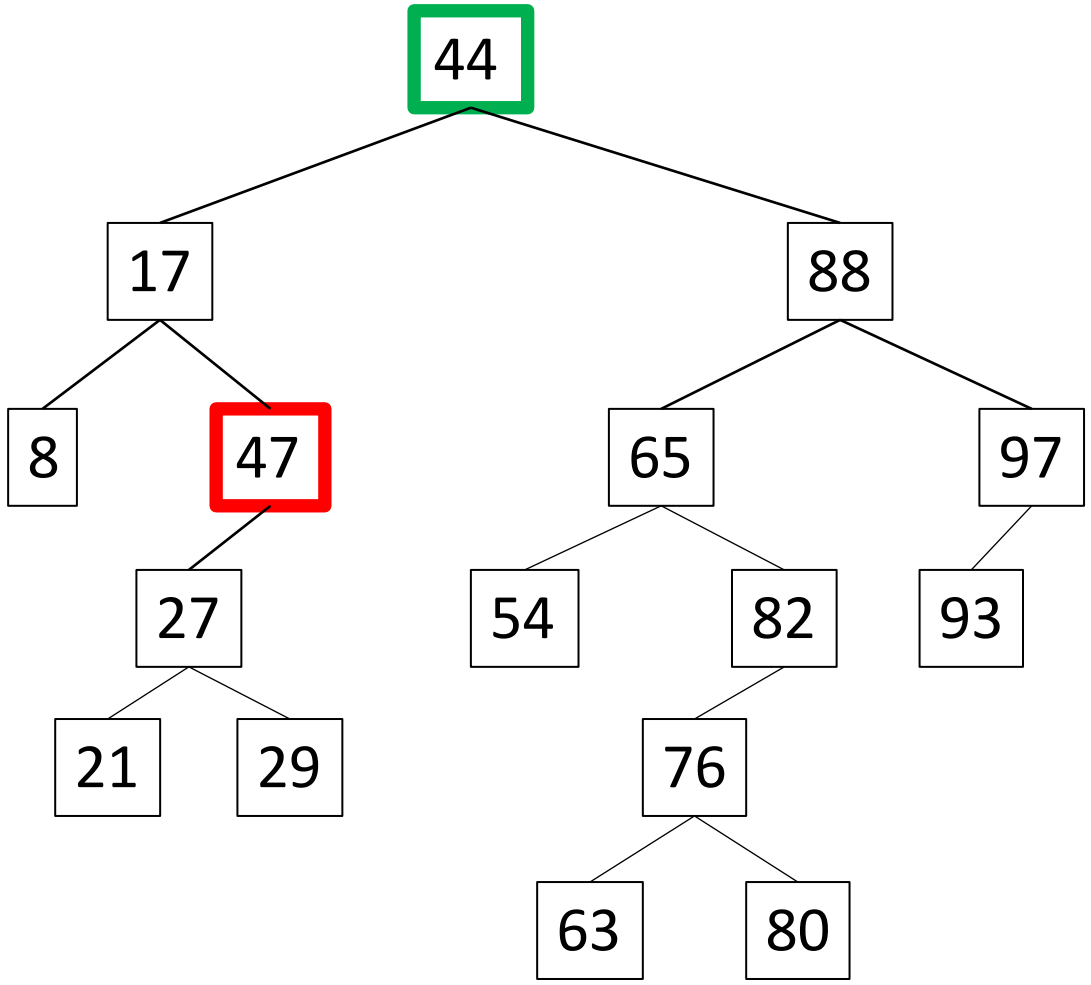


Is it a BST?

Binary Search Tree

Binary Search Tree (BST) properties:

- **A BST is composed of `Comparable` data elements.**
- **A BST is a binary tree (each node has at most two children).**
- **For each node, all left-hand descendants have values that are less that the node.**
- For each node, all right-hand descendants have values that are larger than the node.
- There are no duplicate values (definitions vary).

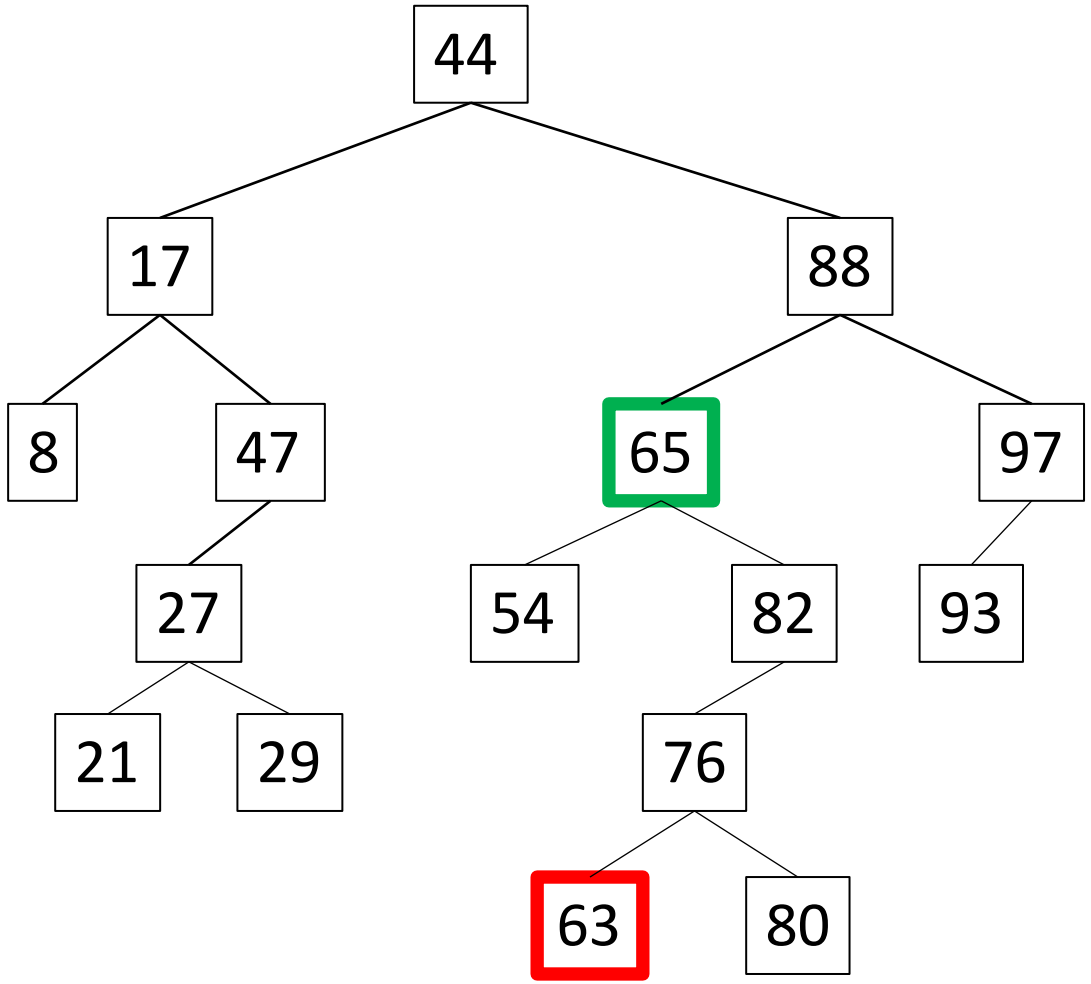


Is it a BST?

Binary Search Tree

Binary Search Tree (BST) properties:

- **A BST is composed of `Comparable` data elements.**
- **A BST is a binary tree (each node has at most two children).**
- **For each node, all left-hand descendants have values that are less that the node.**
- **For each node, all right-hand descendants have values that are larger than the node.**
- There are no duplicate values (definitions vary).

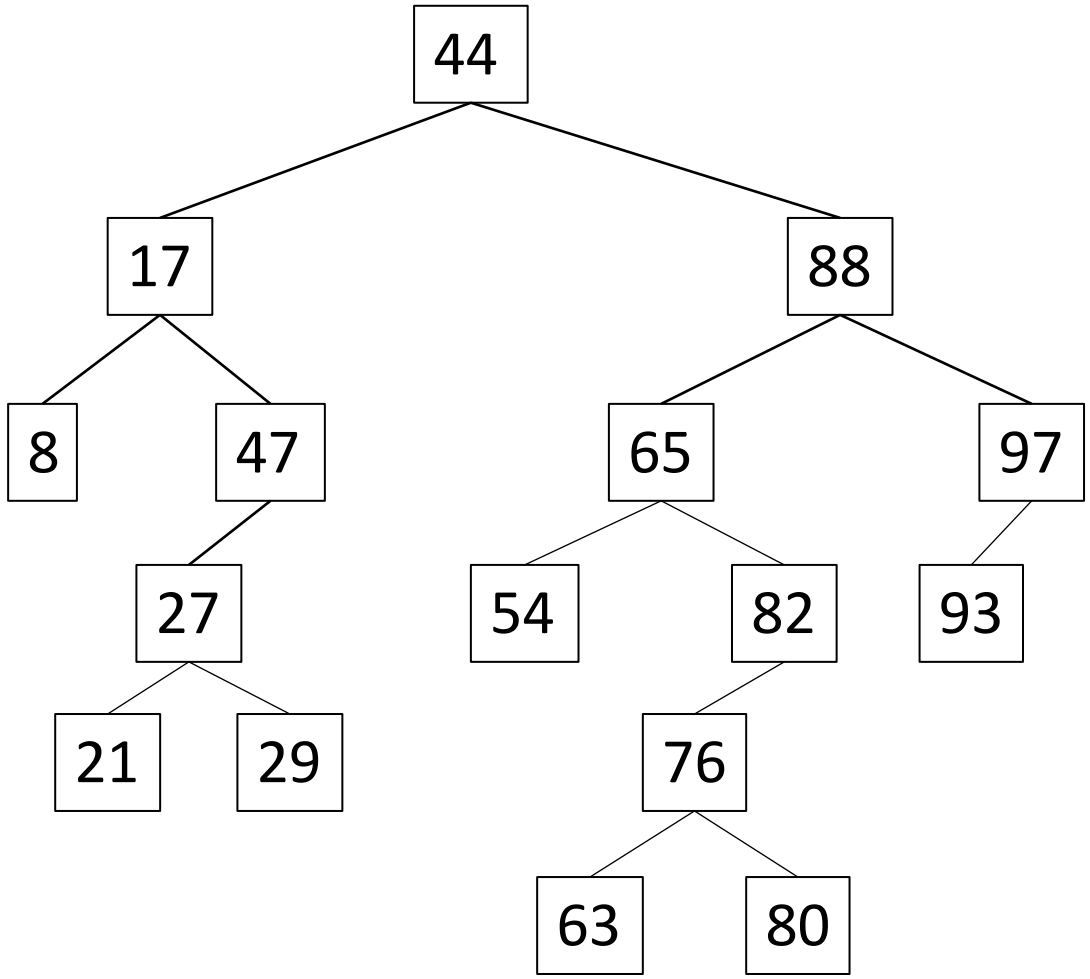


Is it a BST?

Binary Search Tree

Binary Search Tree (BST) properties:

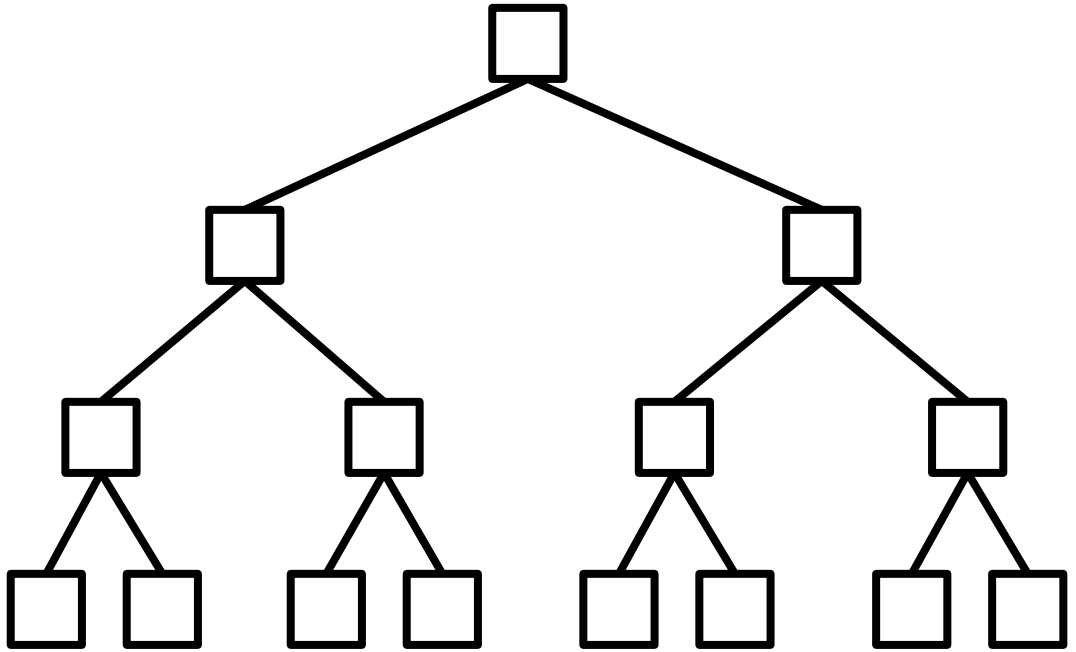
- **A BST is composed of `Comparable` data elements.**
- **A BST is a binary tree (each node has at most two children).**
- **For each node, all left-hand descendants have values that are less that the node.**
- **For each node, all right-hand descendants have values that are larger than the node.**
- **There are no duplicate values (definitions vary).**



Is it a BST?

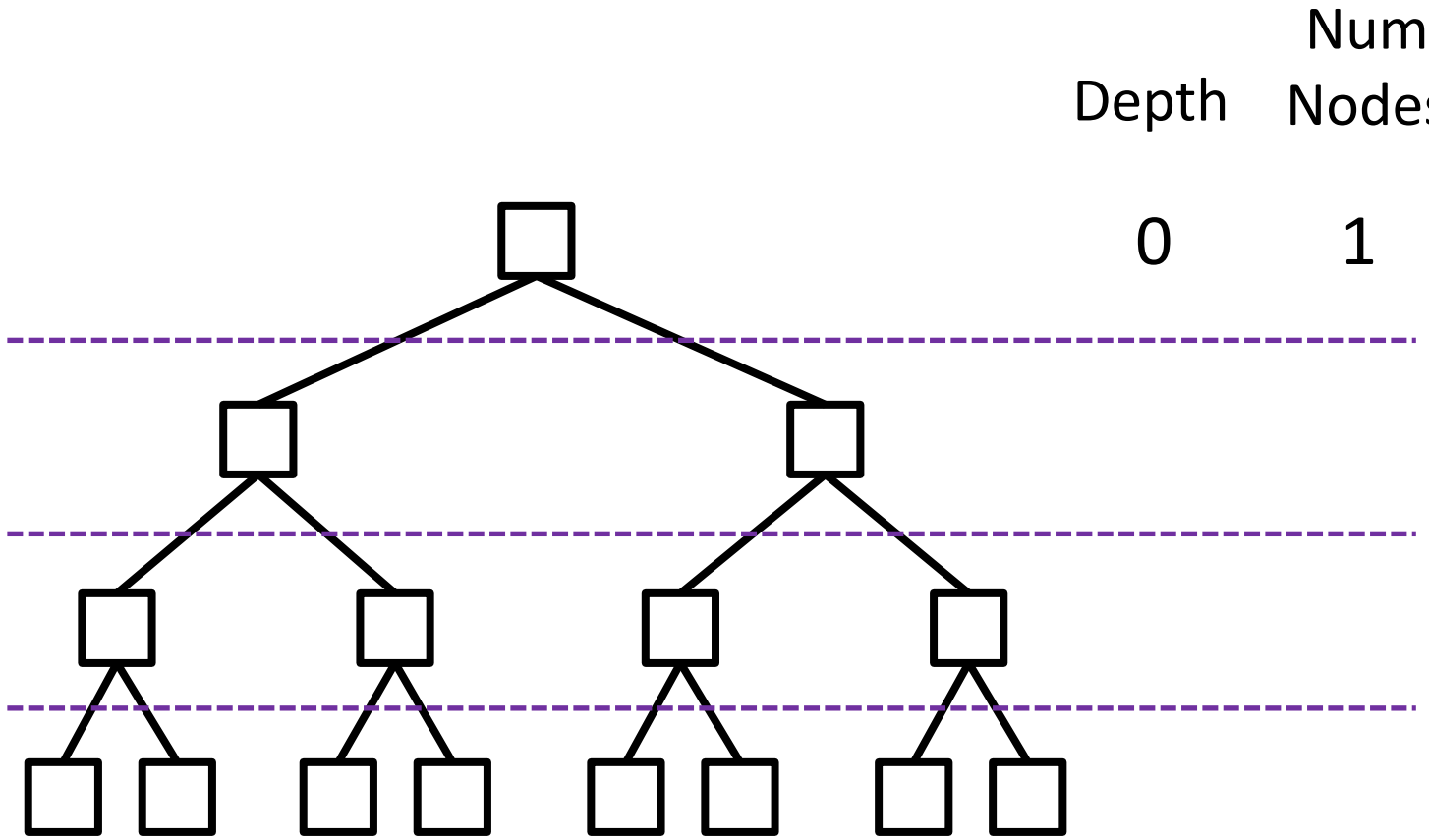
Binary Search Tree

What is the point? Why use a BST?



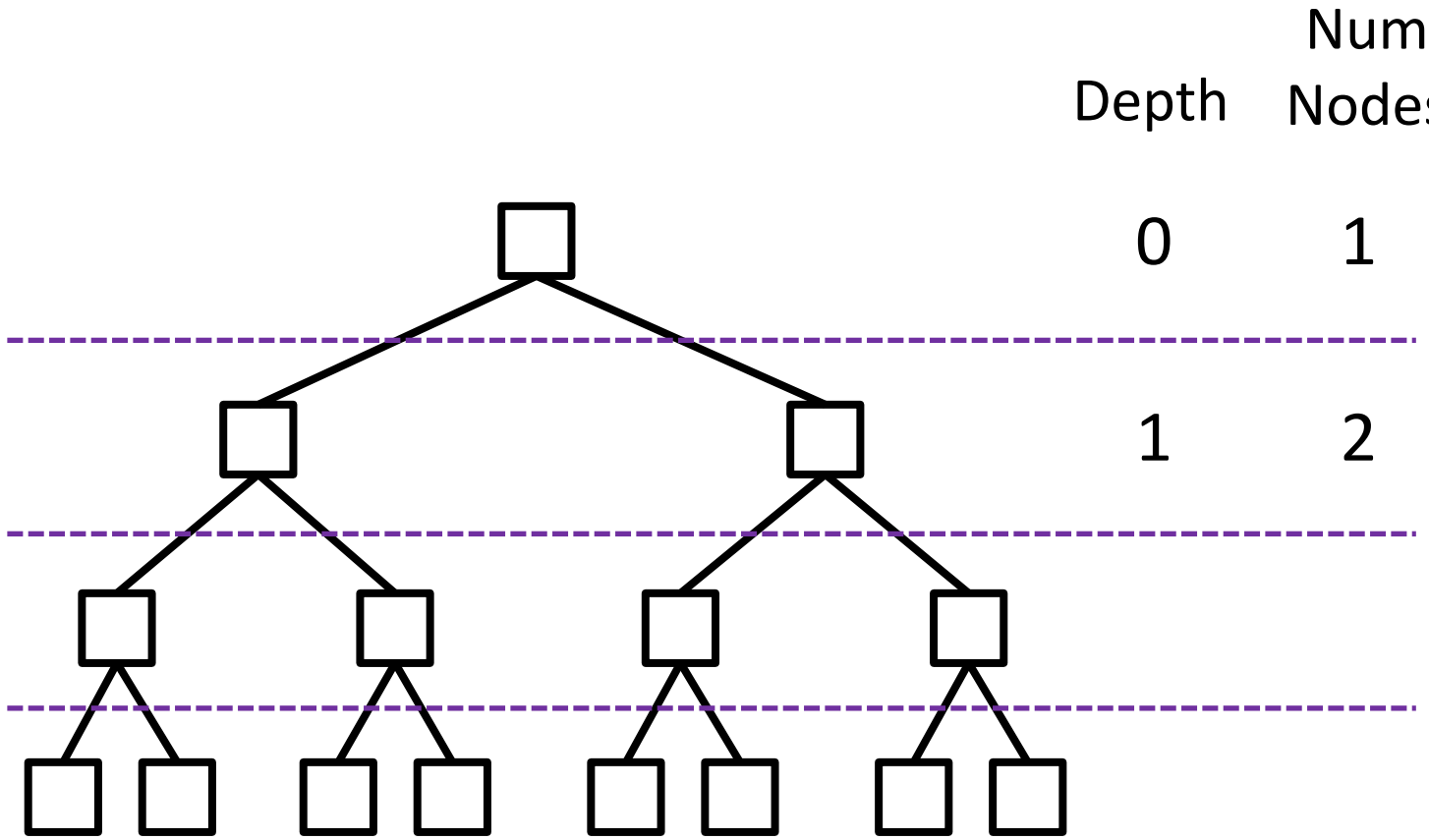
Binary Search Tree

What is the point? Why use a BST?



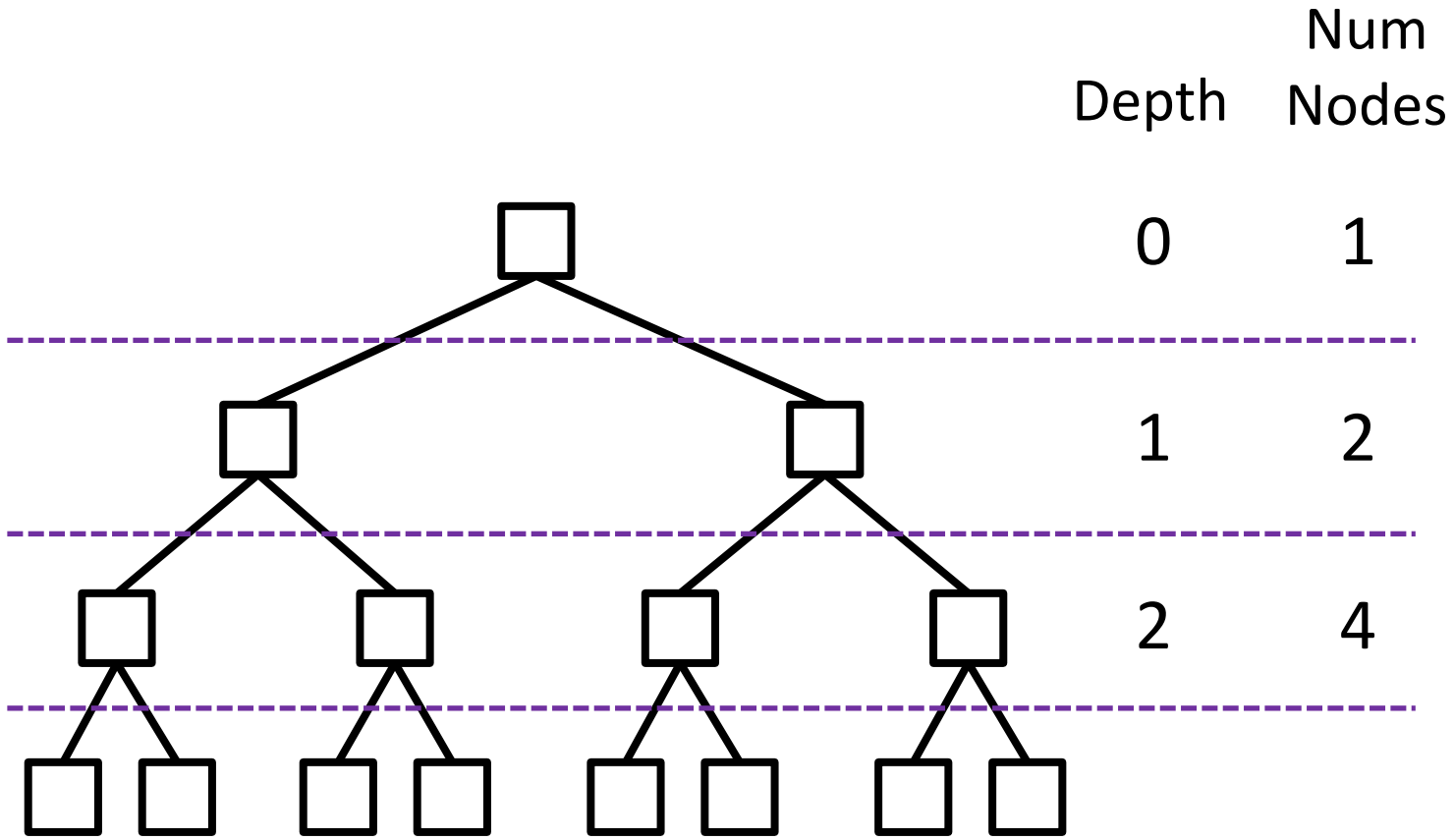
Binary Search Tree

What is the point? Why use a BST?



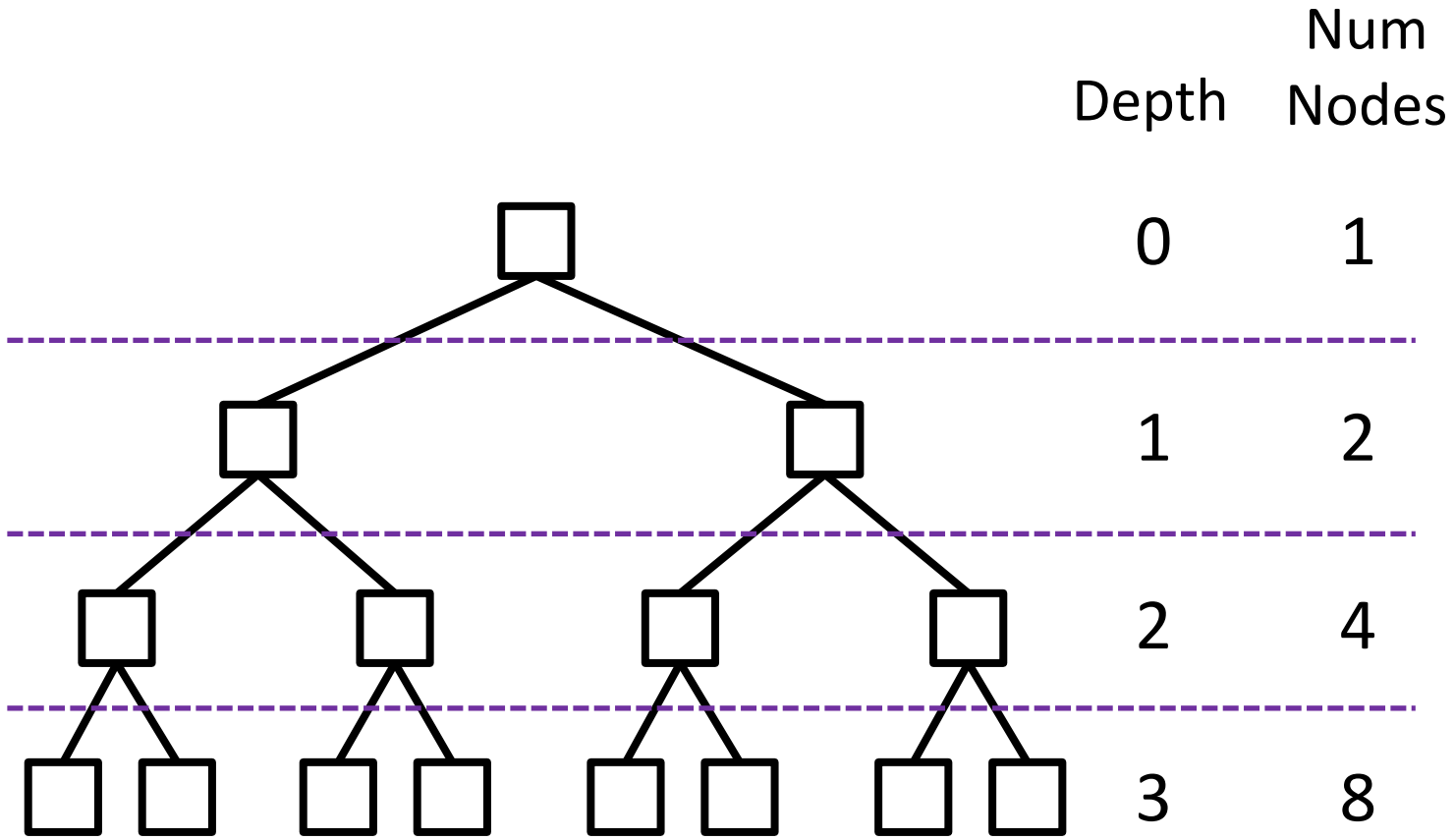
Binary Search Tree

What is the point? Why use a BST?



Binary Search Tree

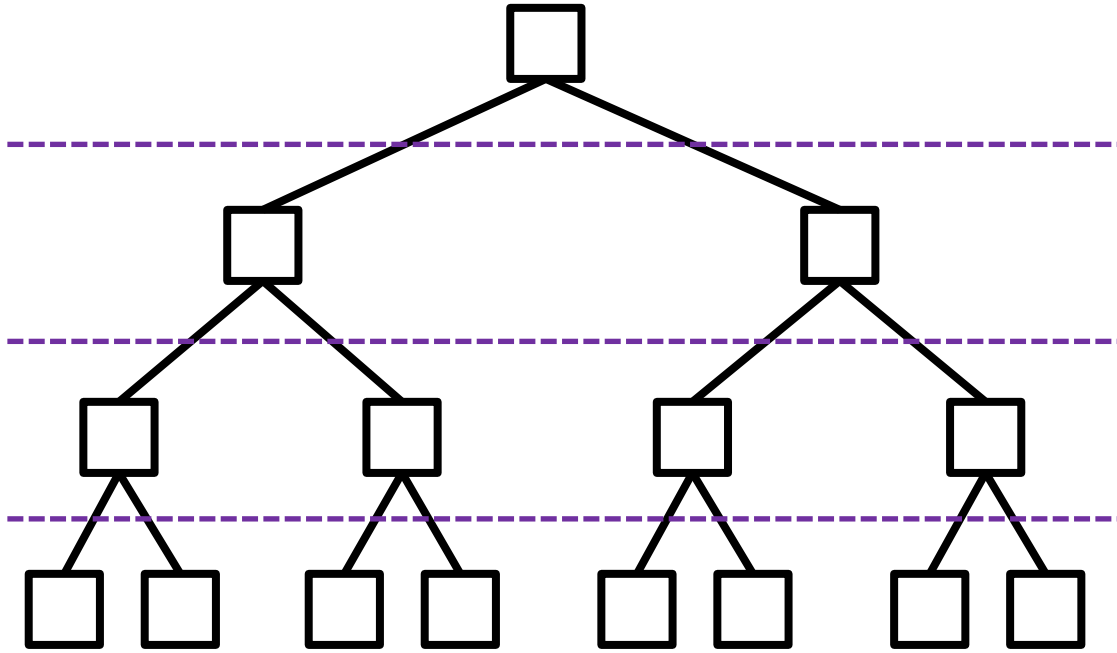
What is the point? Why use a BST?



Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most $??$ nodes.

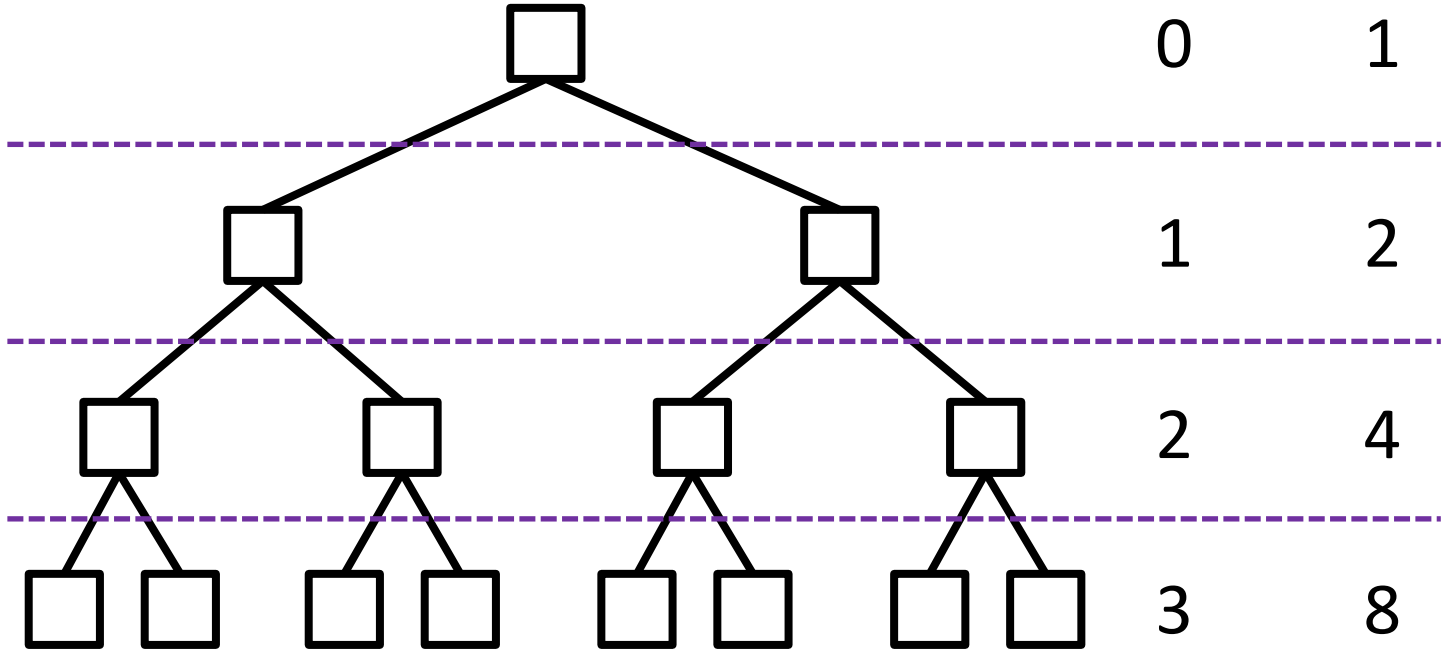


Depth	Num Nodes
0	1
1	2
2	4
3	8

Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

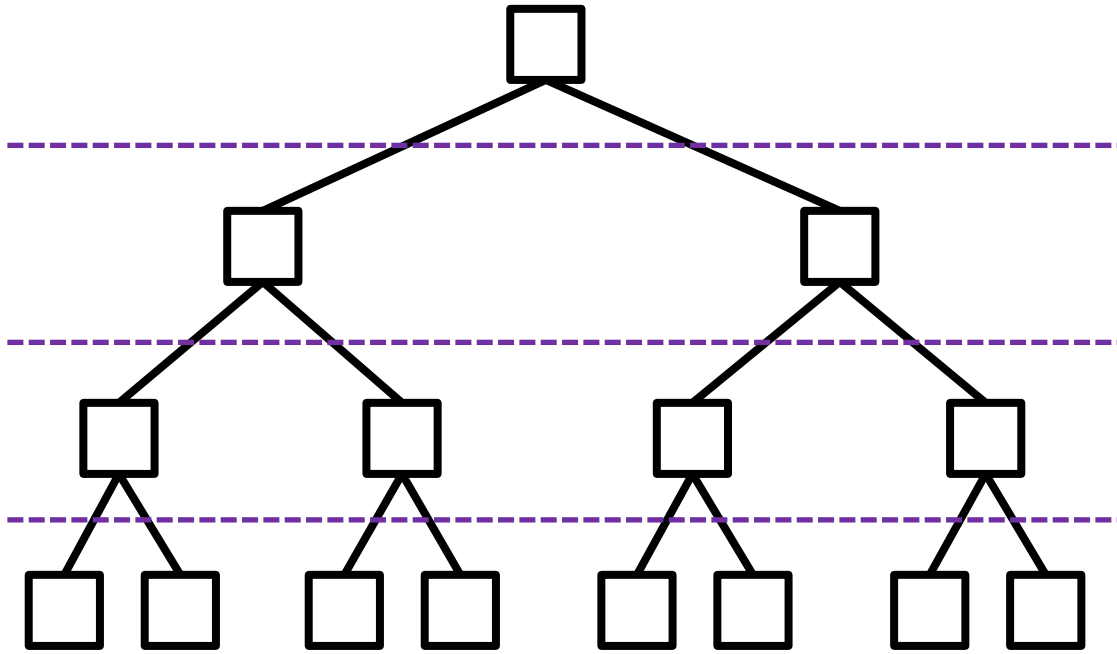


Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf?



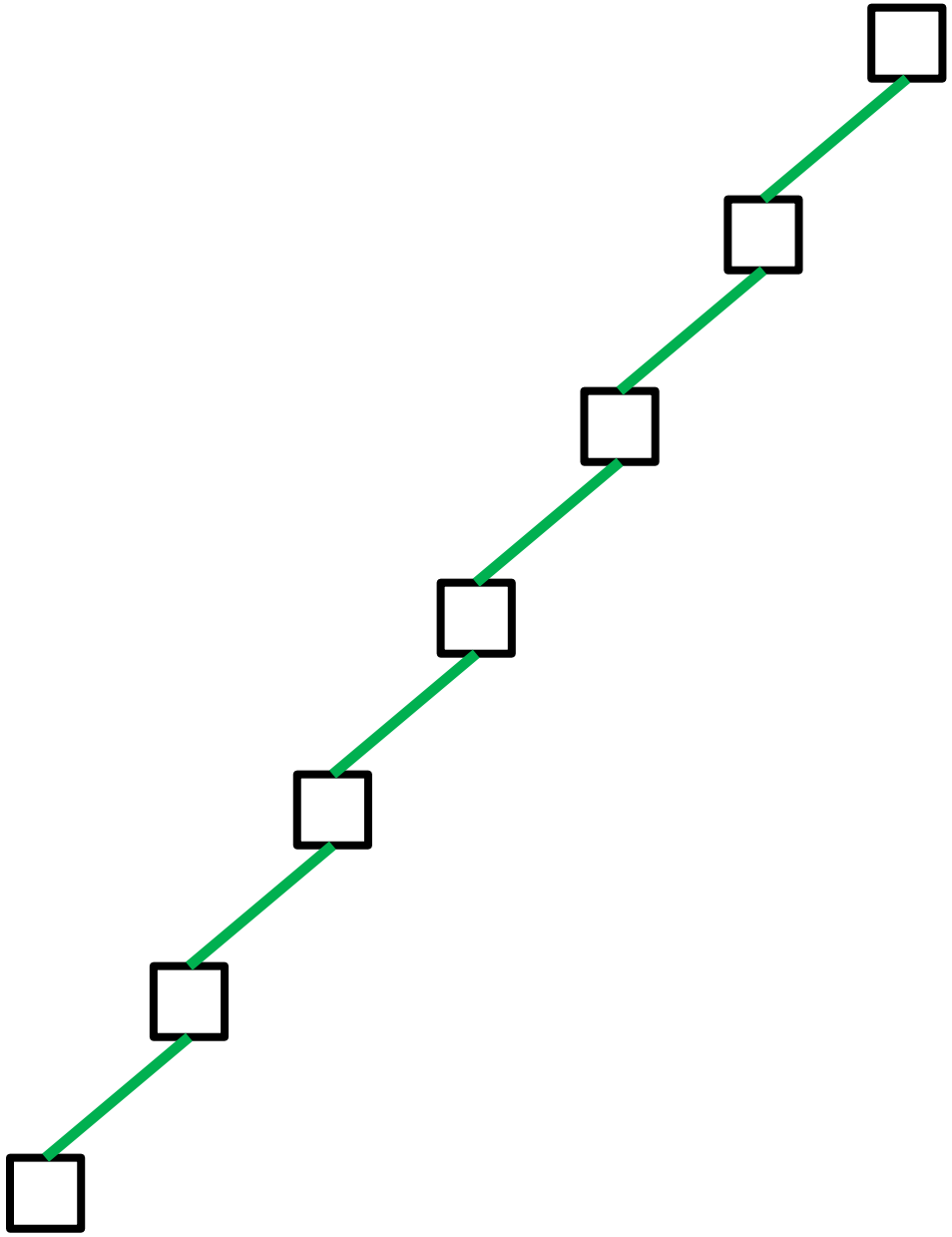
Depth	Num Nodes
0	1
1	2
2	4
3	8

Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? $n - 1$

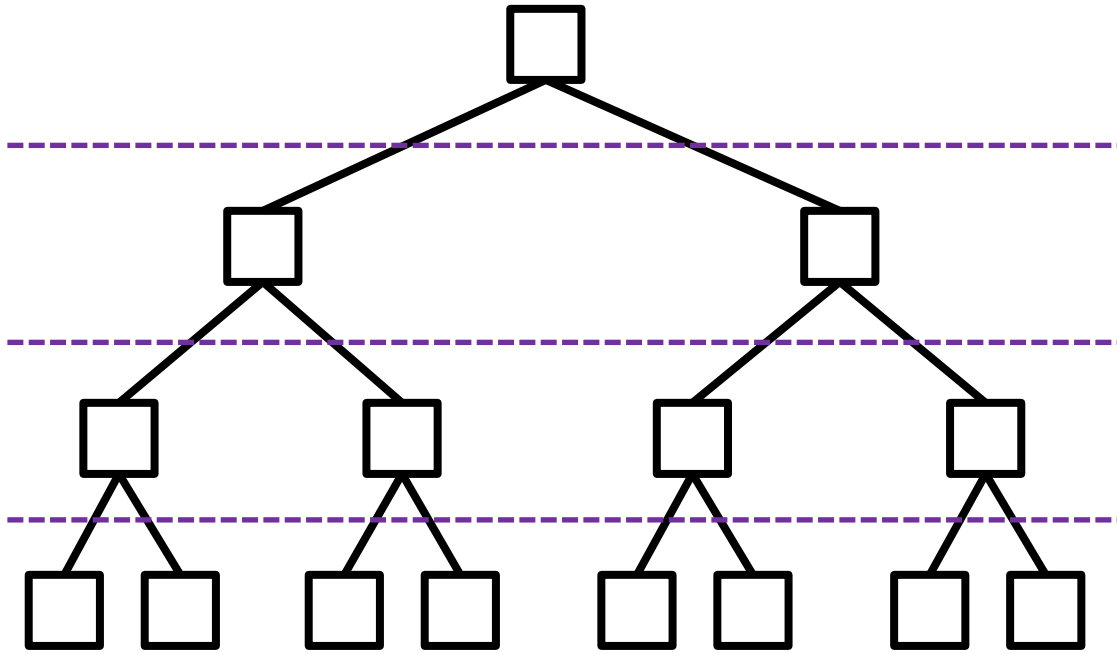


Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf?



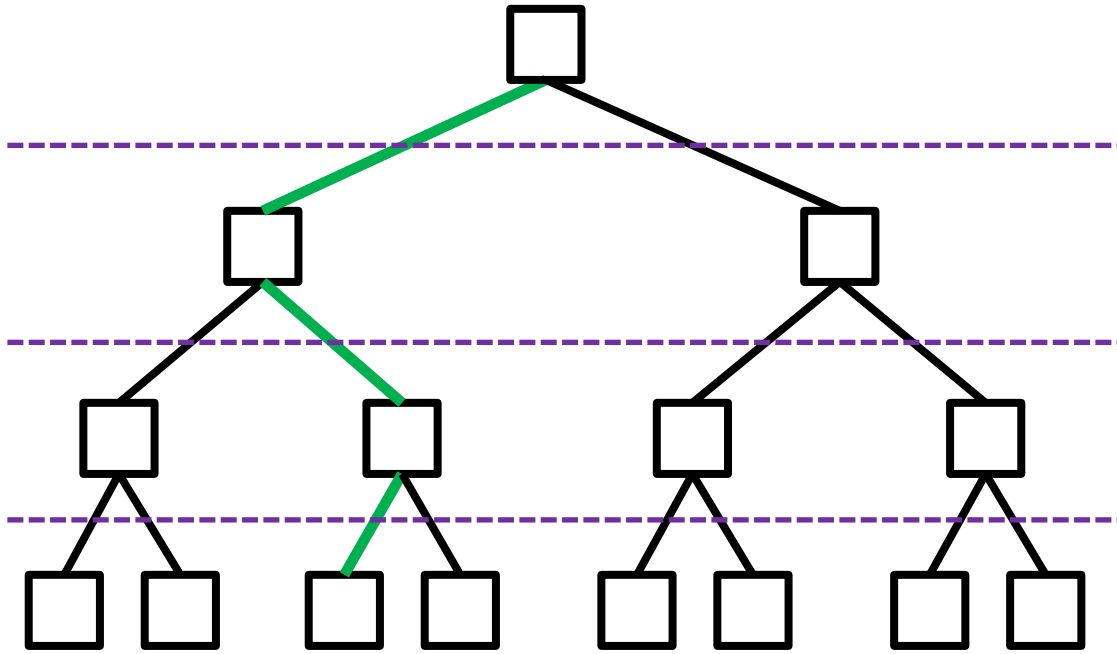
Depth	Num Nodes
0	1
1	2
2	4
3	8

Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*



Depth	Num Nodes
0	1
1	2
2	4
3	8

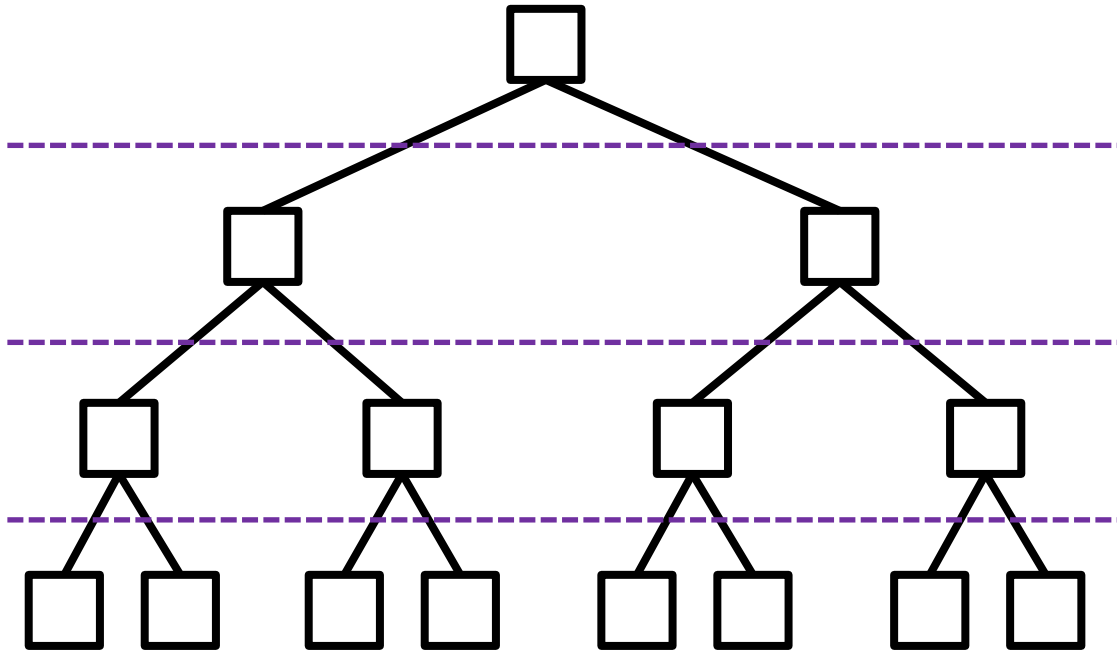
Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?



Depth	Num Nodes
0	1
1	2
2	4
3	8

Binary Search Tree

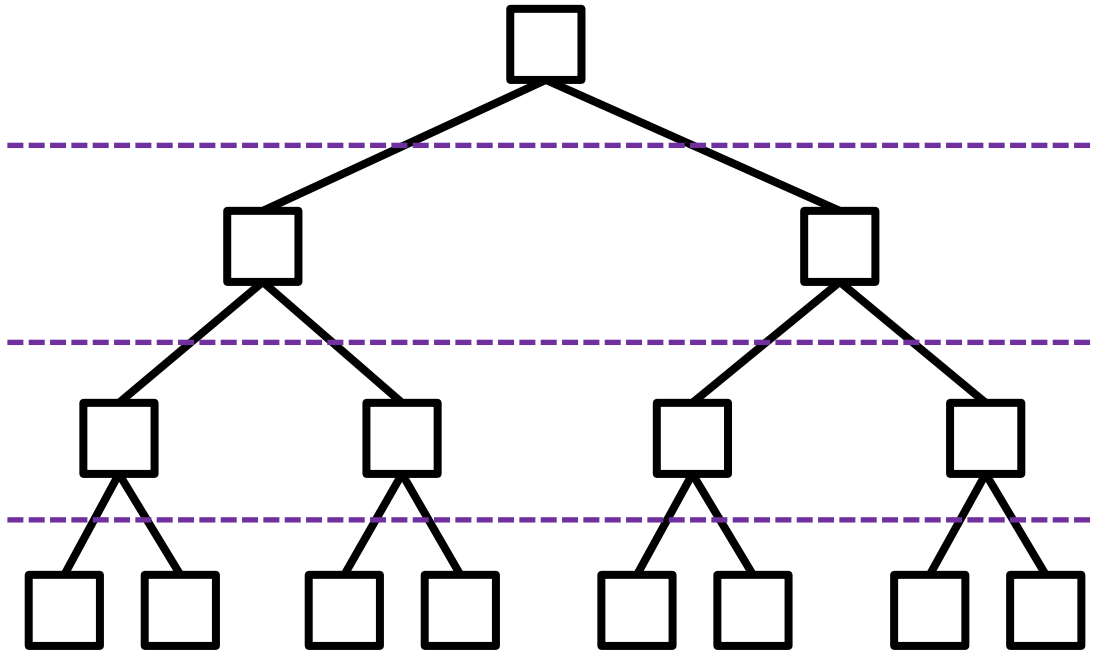
What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h$$



Depth	Num Nodes
0	1
1	2
2	4
3	8

Binary Search Tree

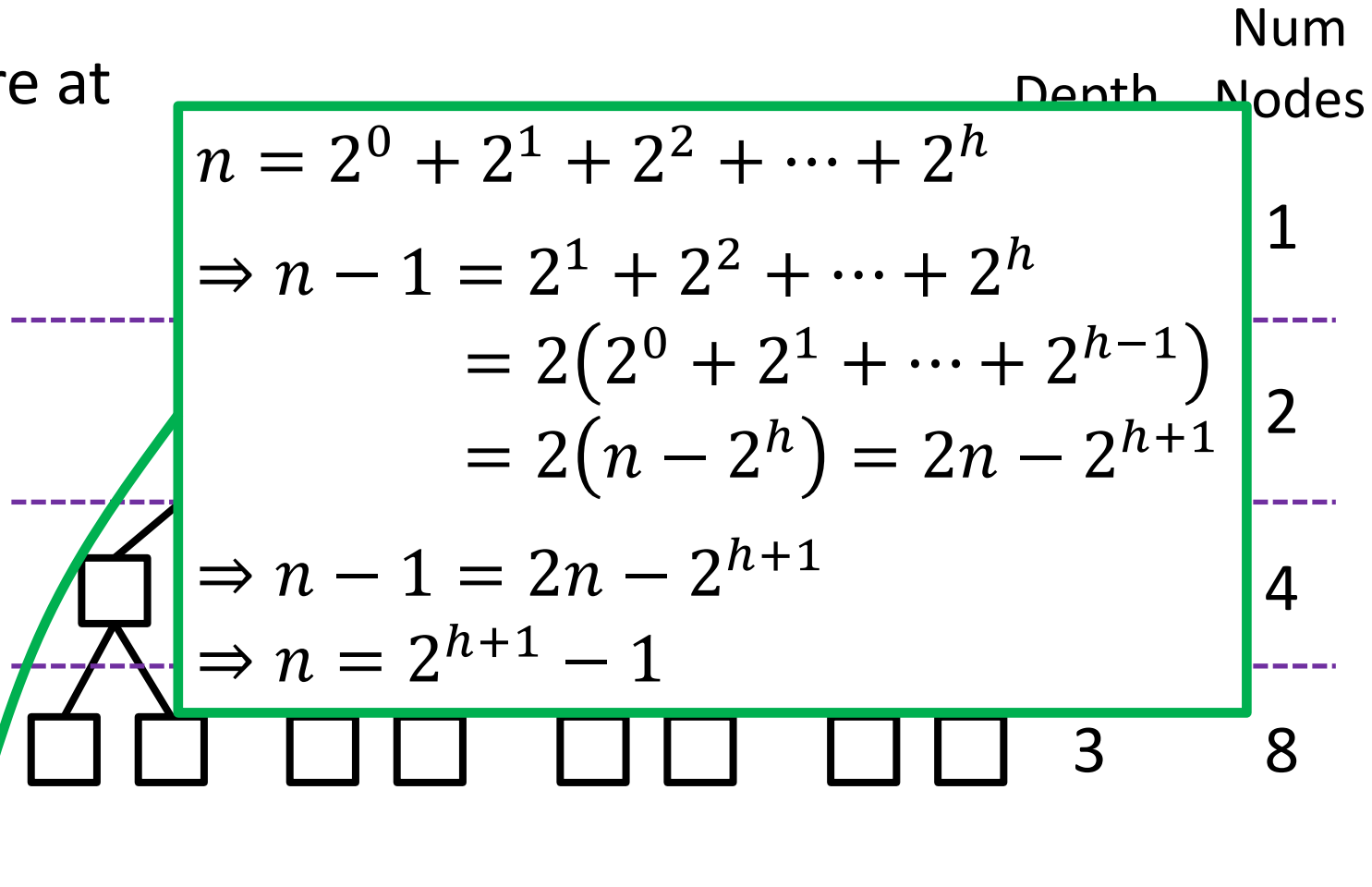
What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$$



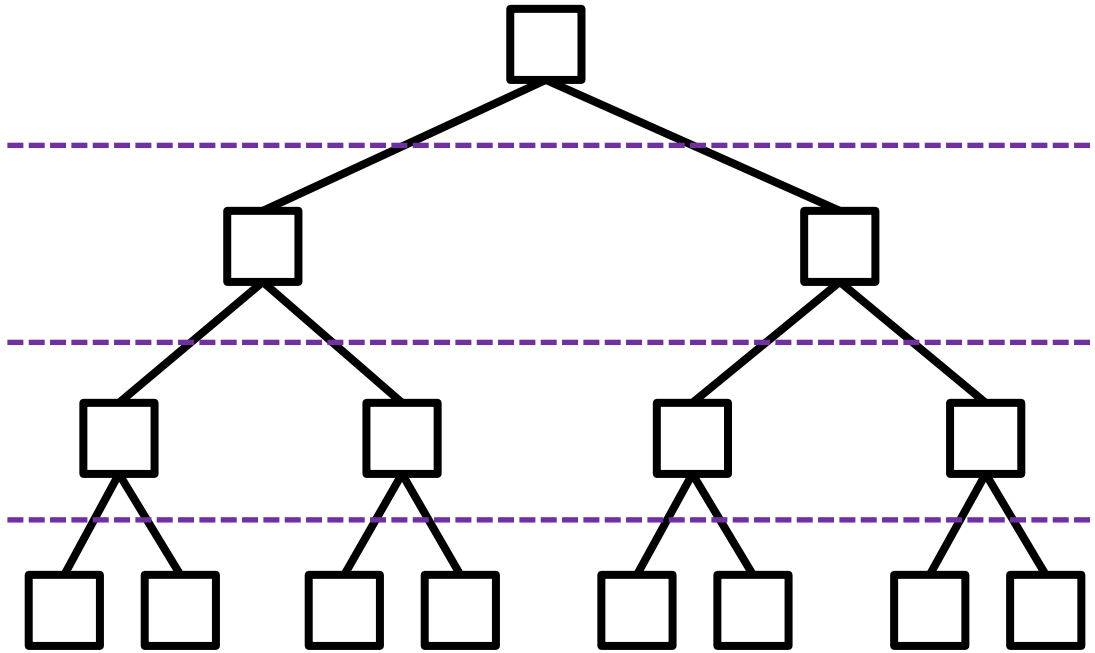
Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?



Depth	Num Nodes
0	1
1	2
2	4
3	8

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 \Rightarrow n + 1 = 2^{h+1}$$

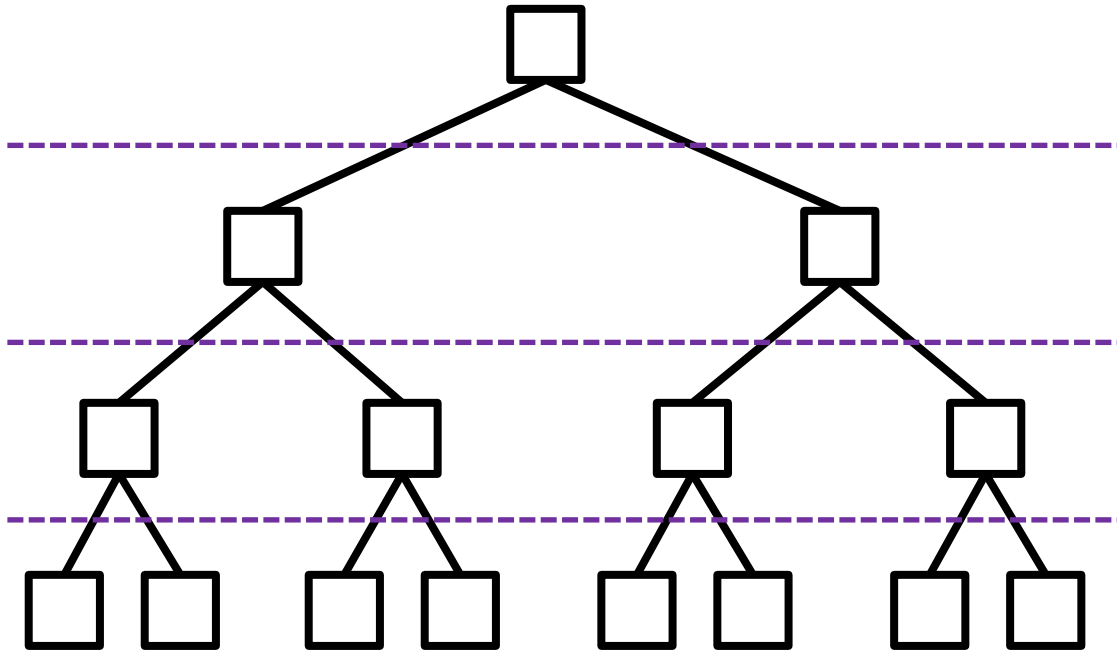
Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?



Depth	Num Nodes
0	1
1	2
2	4
3	8

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 \Rightarrow \begin{matrix} n + 1 = 2^{h+1} \\ \log_2(n+1) = h + 1 \end{matrix}$$

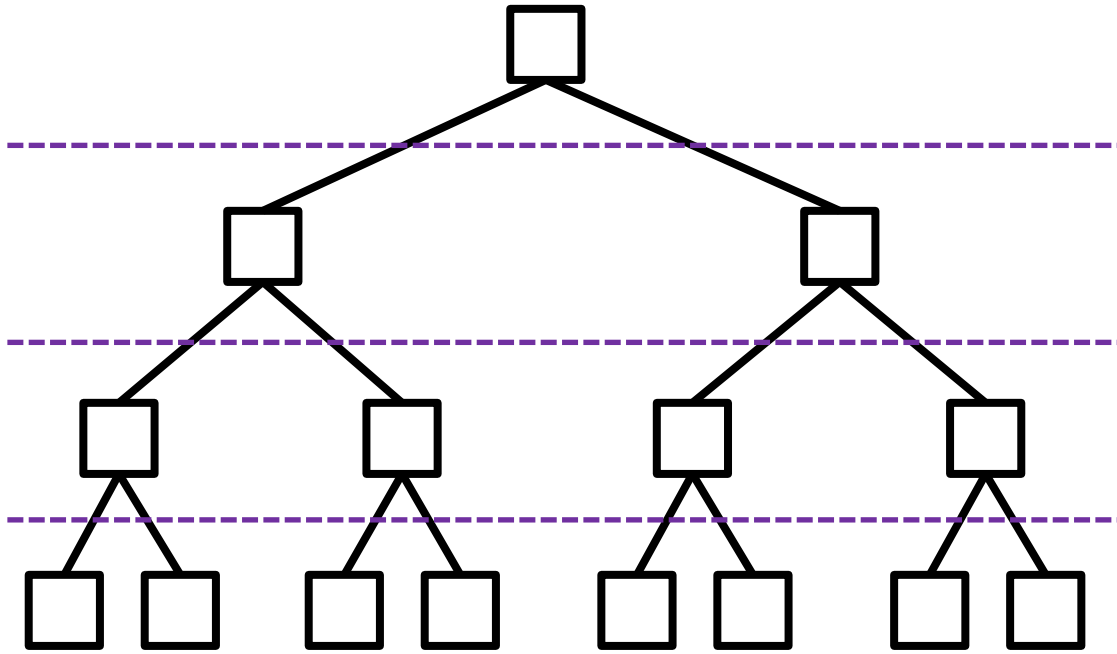
Binary Search Tree

What is the point? Why use a BST?

In general, at depth d, there are at most 2^d nodes.

Given a BST with n nodes, what is the greatest number of edges we would have to traverse to go from the root to a leaf? *height of tree.*

Given n nodes, what is the smallest height (h) of the BST?



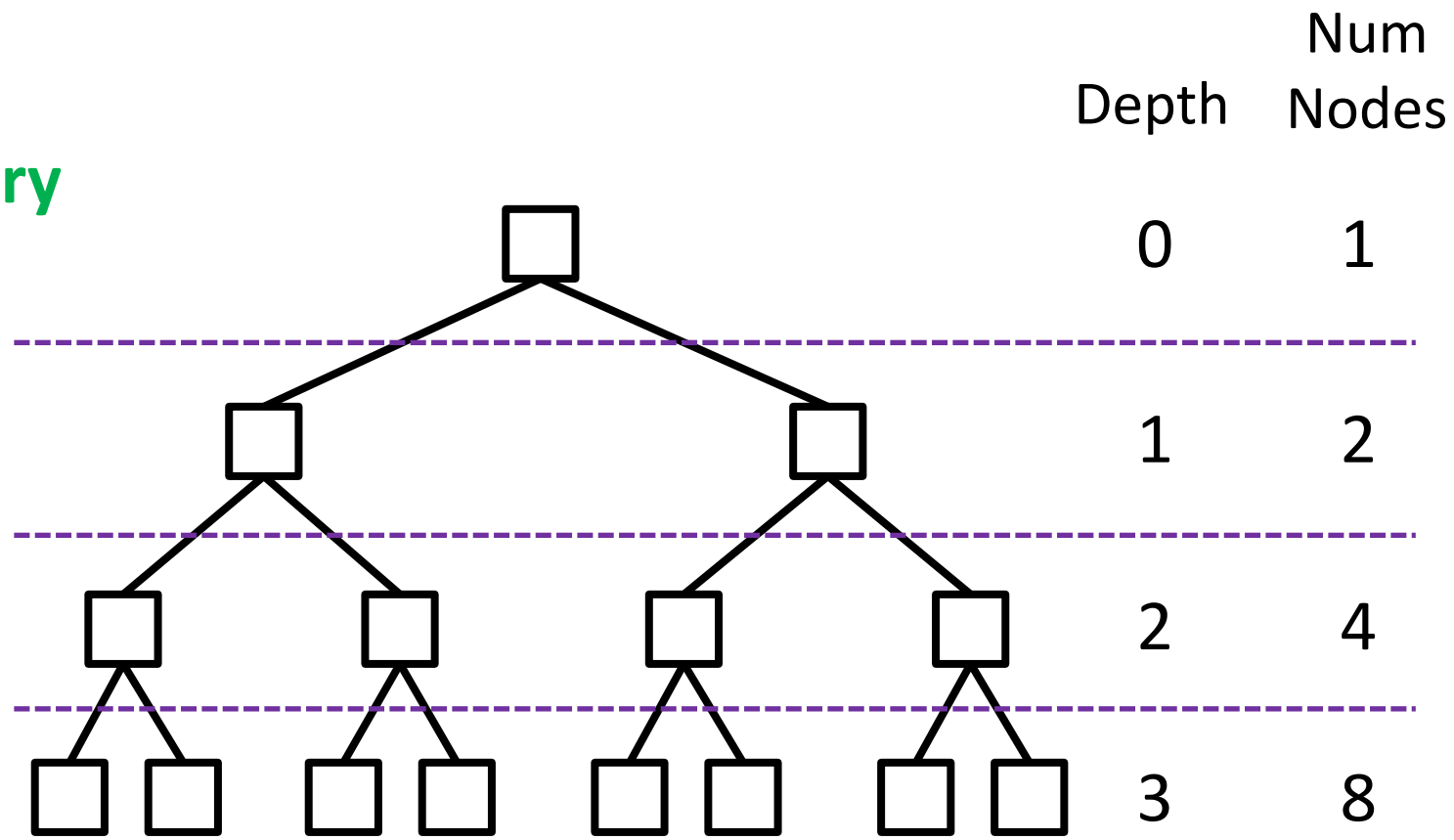
Depth	Num Nodes
0	1
1	2
2	4
3	8

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 \Rightarrow \begin{matrix} n+1 = 2^{h+1} \\ \log_2(n+1) = h+1 \end{matrix} \Rightarrow \boldsymbol{h \in O(\log n)}$$

Binary Search Tree

What is the point? Why use a BST?

This means we can access any node in a specific type of binary tree in $\log n$ time.

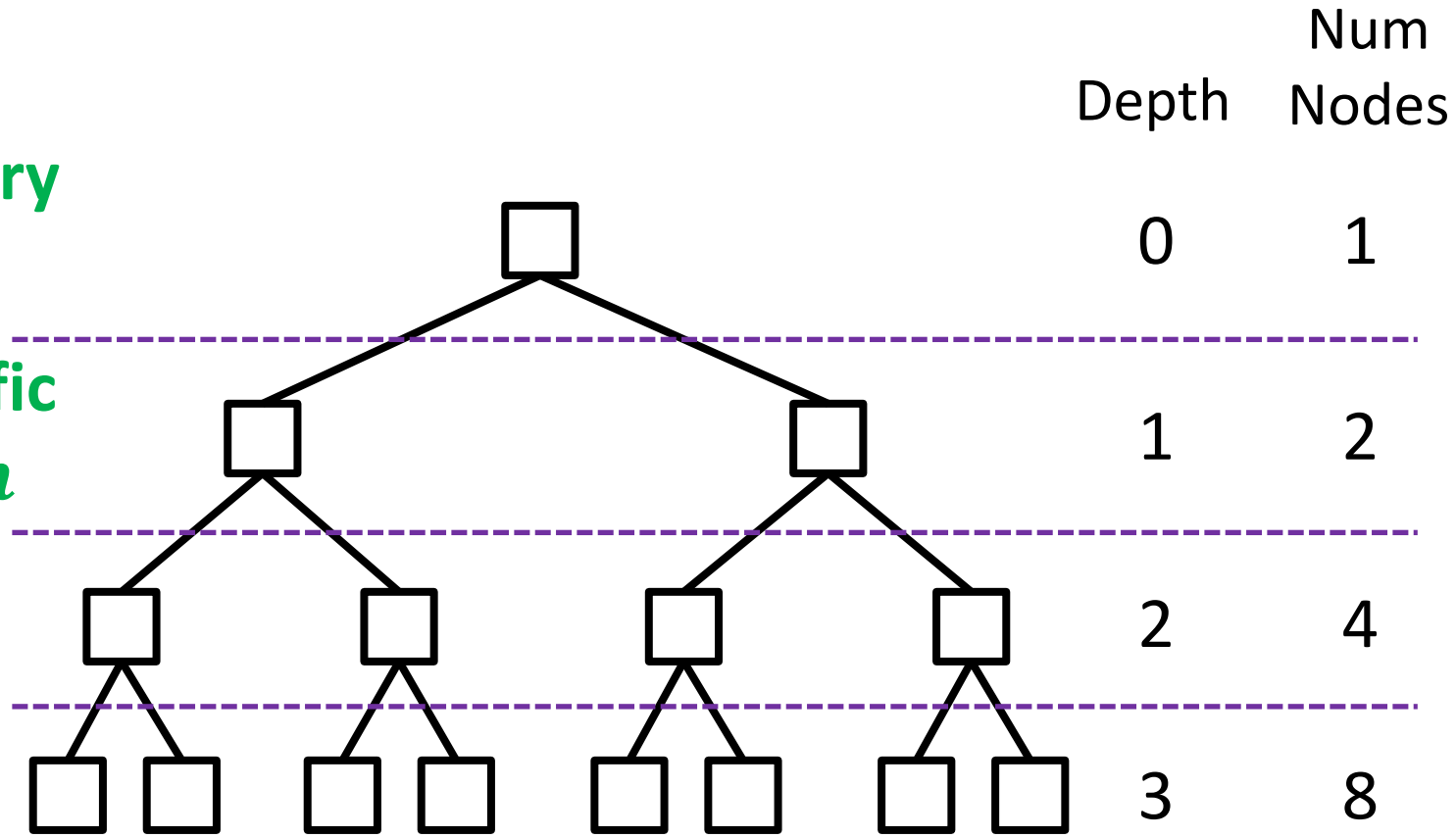


Binary Search Tree

What is the point? Why use a BST?

This means we can access any node in a specific type of binary tree in $\log n$ time.

Of note, we can test if a specific value is in a collection in $\log n$ time.



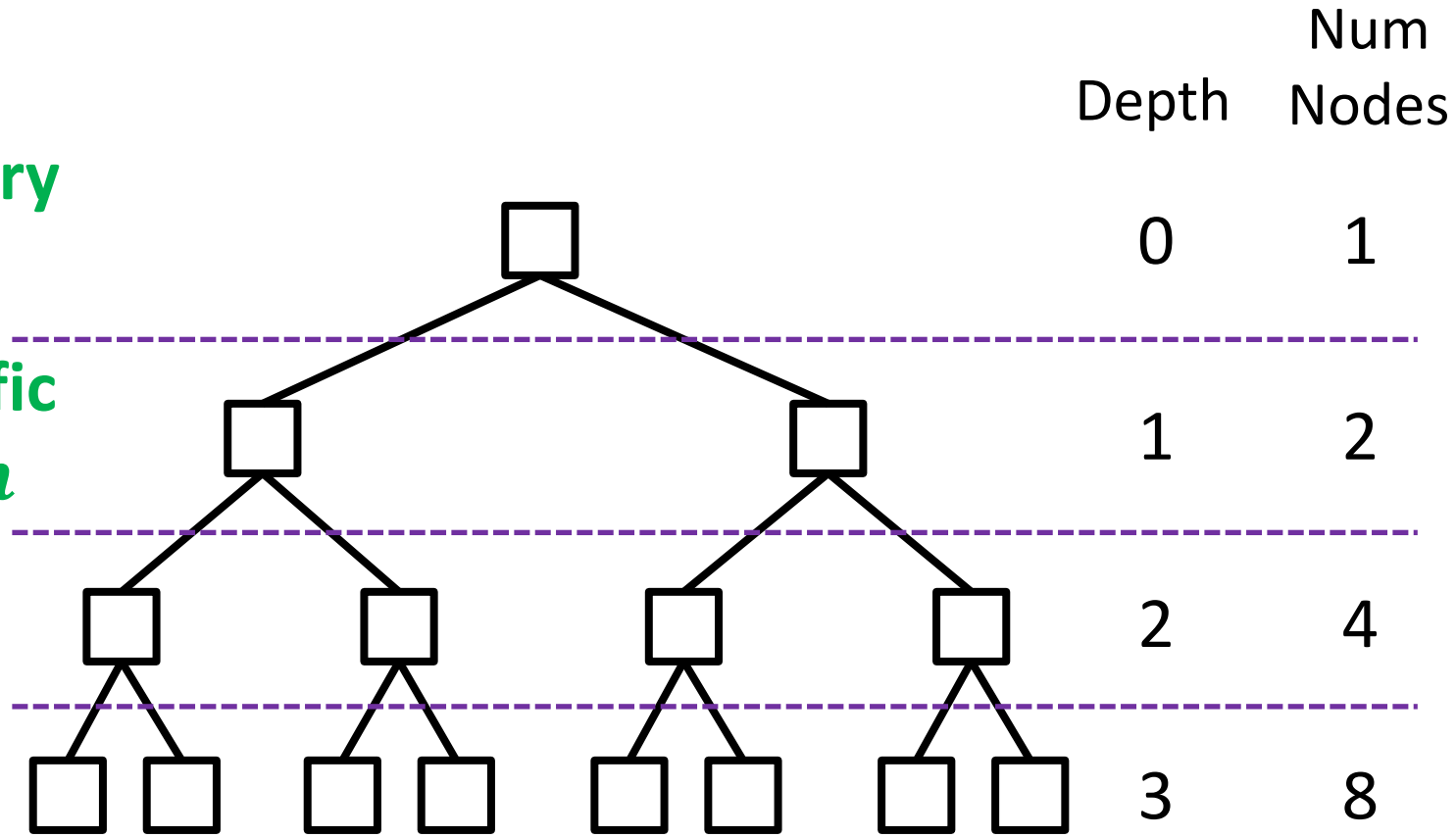
Binary Search Tree

What is the point? Why use a BST?

This means we can access any node in a specific type of binary tree in $\log n$ time.

Of note, we can test if a specific value is in a collection in $\log n$ time.

But we can already do that with a sorted array and Binary Search!



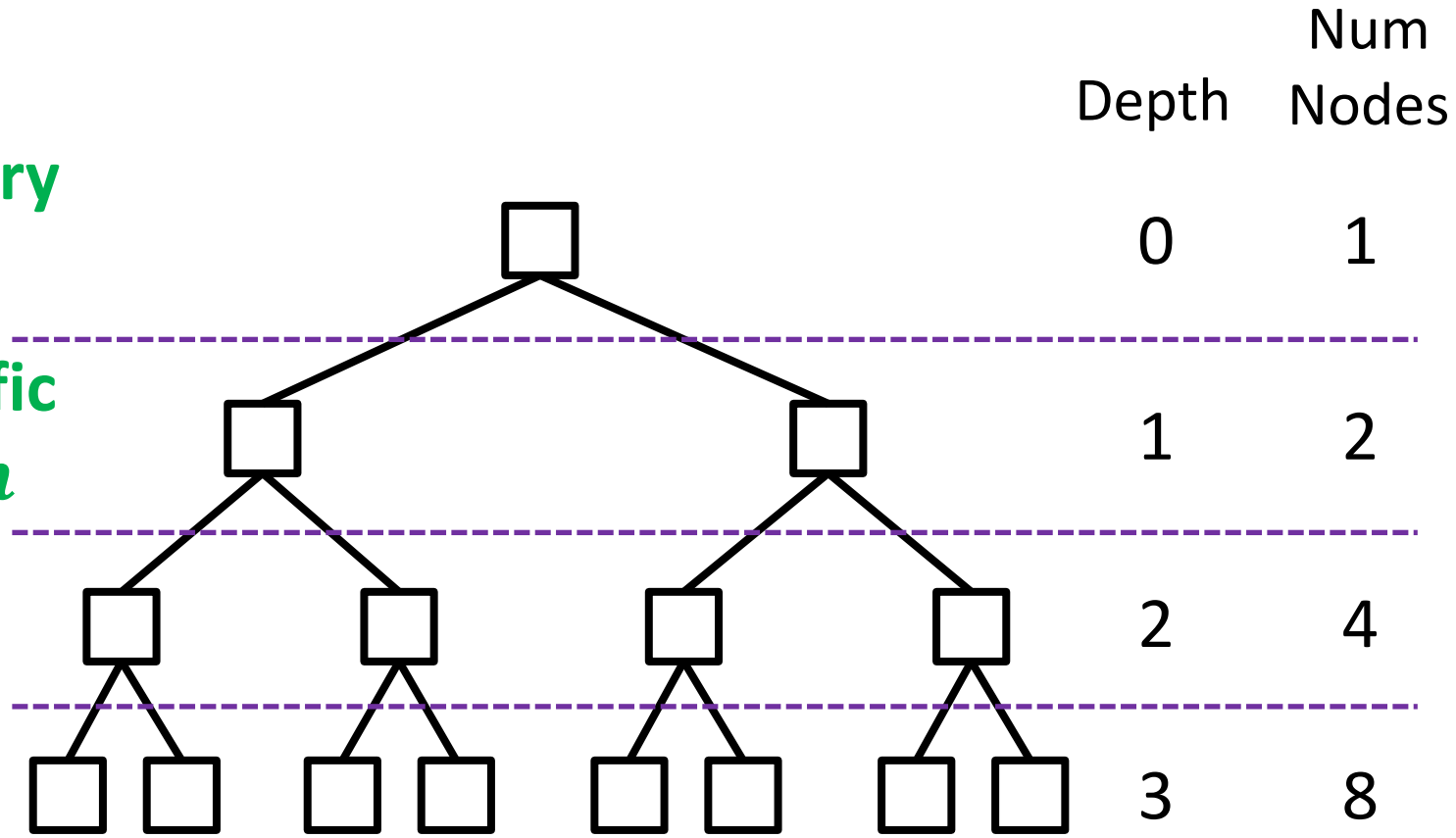
Binary Search Tree

What is the point? Why use a BST?

This means we can access any node in a specific type of binary tree in $\log n$ time.

Of note, we can test if a specific value is in a collection in $\log n$ time.

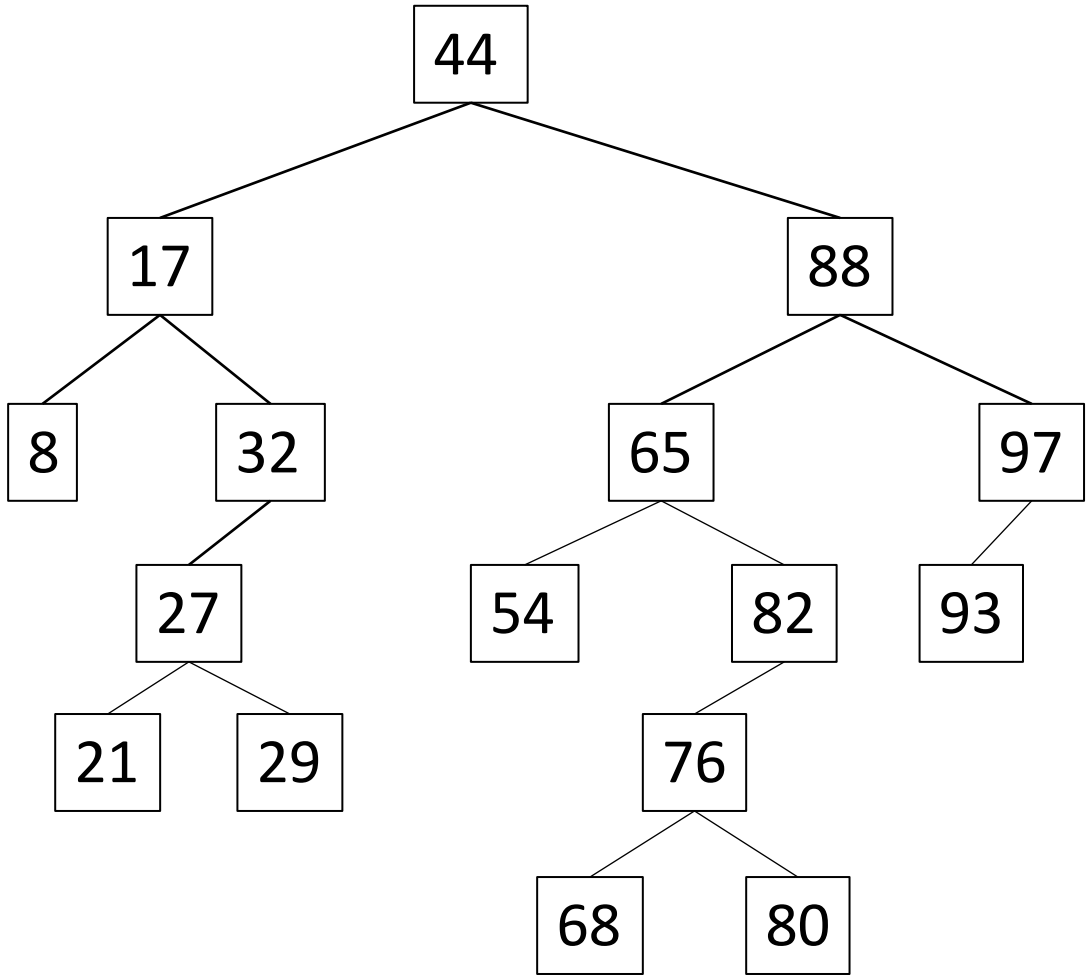
But we can already do that with a sorted array and Binary Search!



Perhaps managing a BST is more efficient than managing an array.

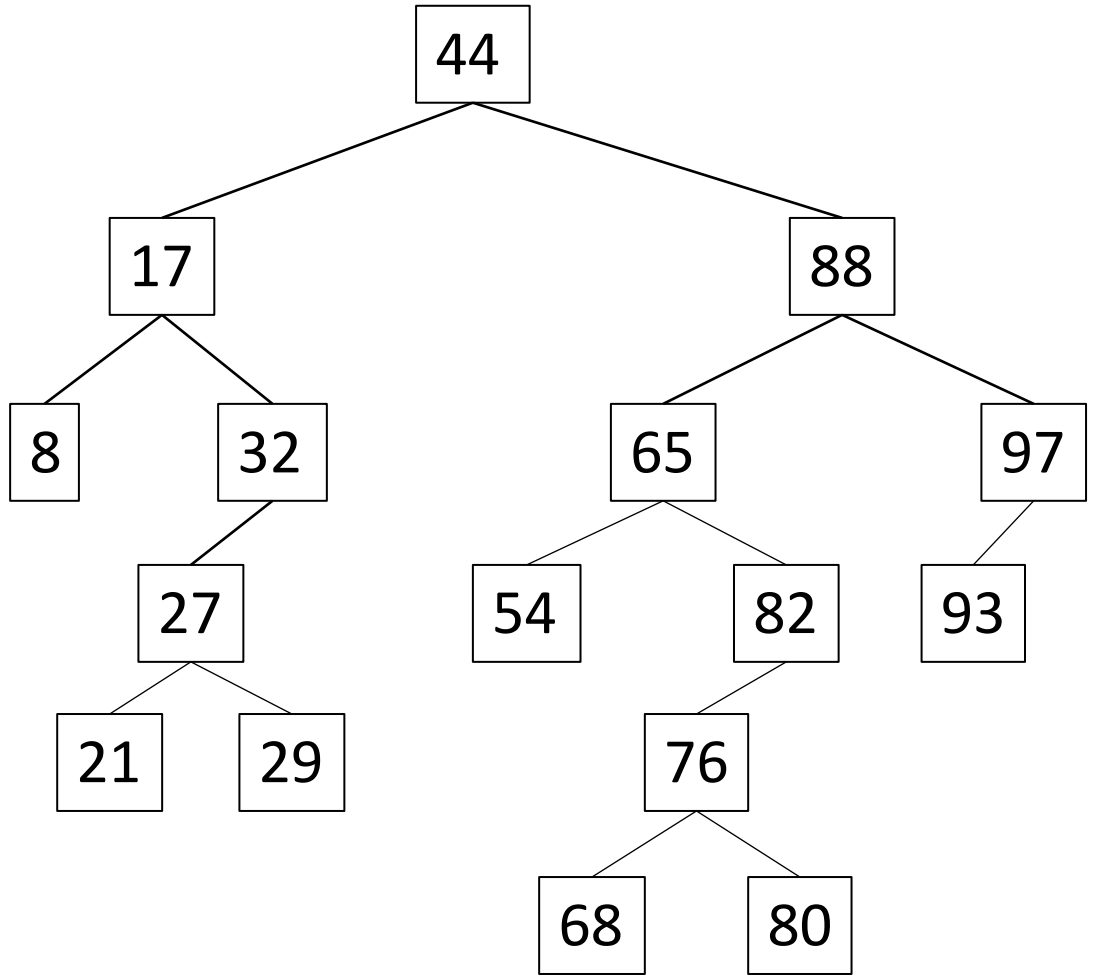
Binary Search Tree - Insertion

```
public class Node {

    private int value;
    private Node left;
    private Node right;
    private Node parent;

    public Node(int value) {
        this.value = value;
    }

    // getValue()
    // getLeft(), getRight()
    // getParent()

    // setLeft(), setRight()
    // setParent()
}
```



Binary Search Tree - Insertion

```
insert(31);
```

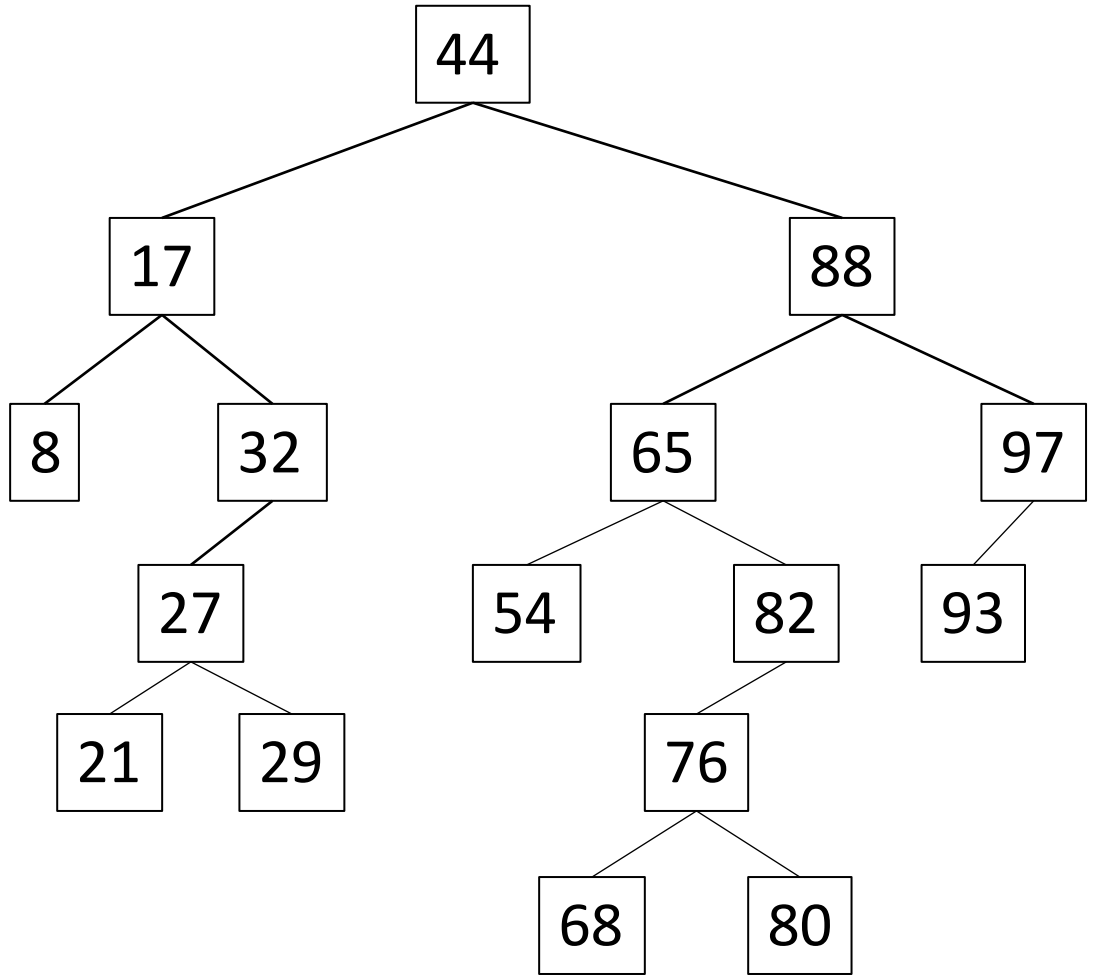


Binary Search Tree - Insertion

```
insert(31);
```

Step 1: Find where it should go.

Step 2: Modify pointers.

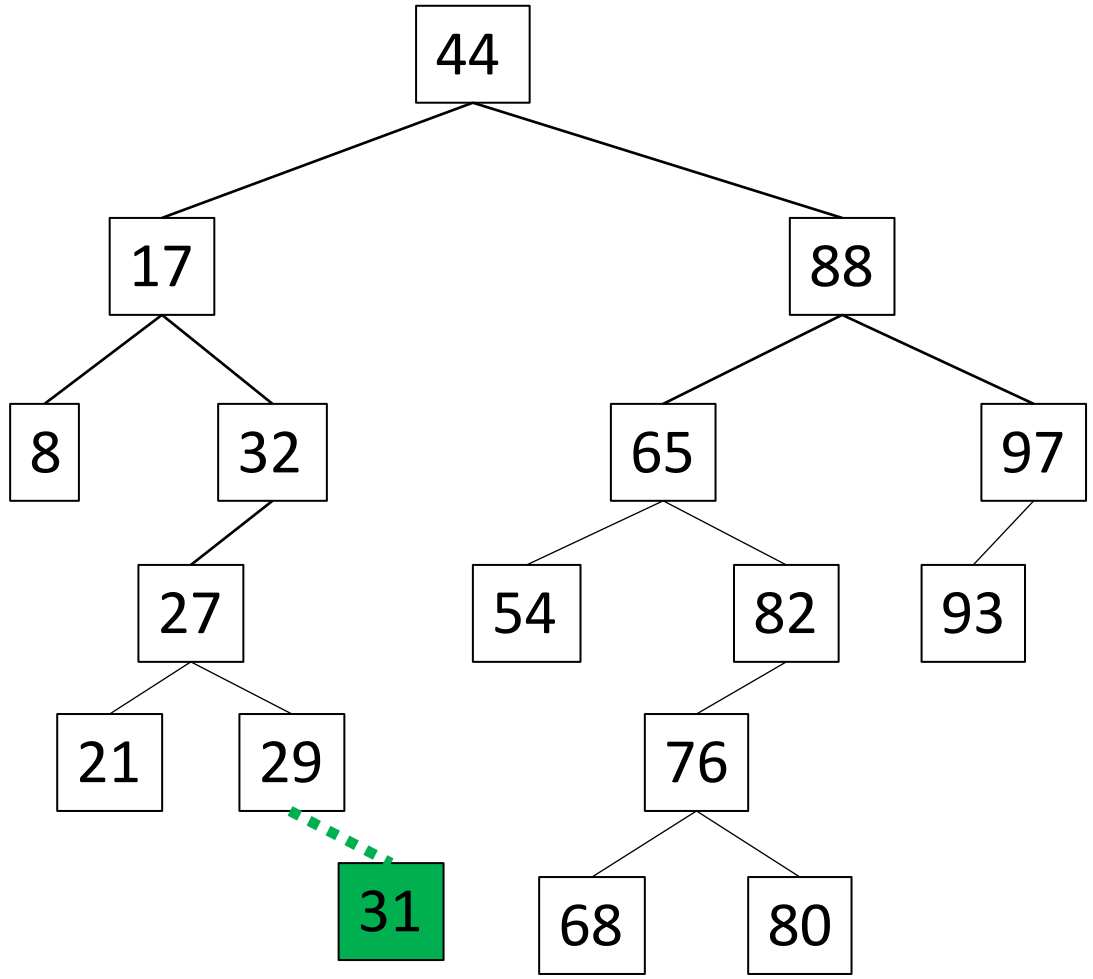


Binary Search Tree - Insertion

```
insert(31);
```

Step 1: Find where it should go.

Step 2: Modify pointers.

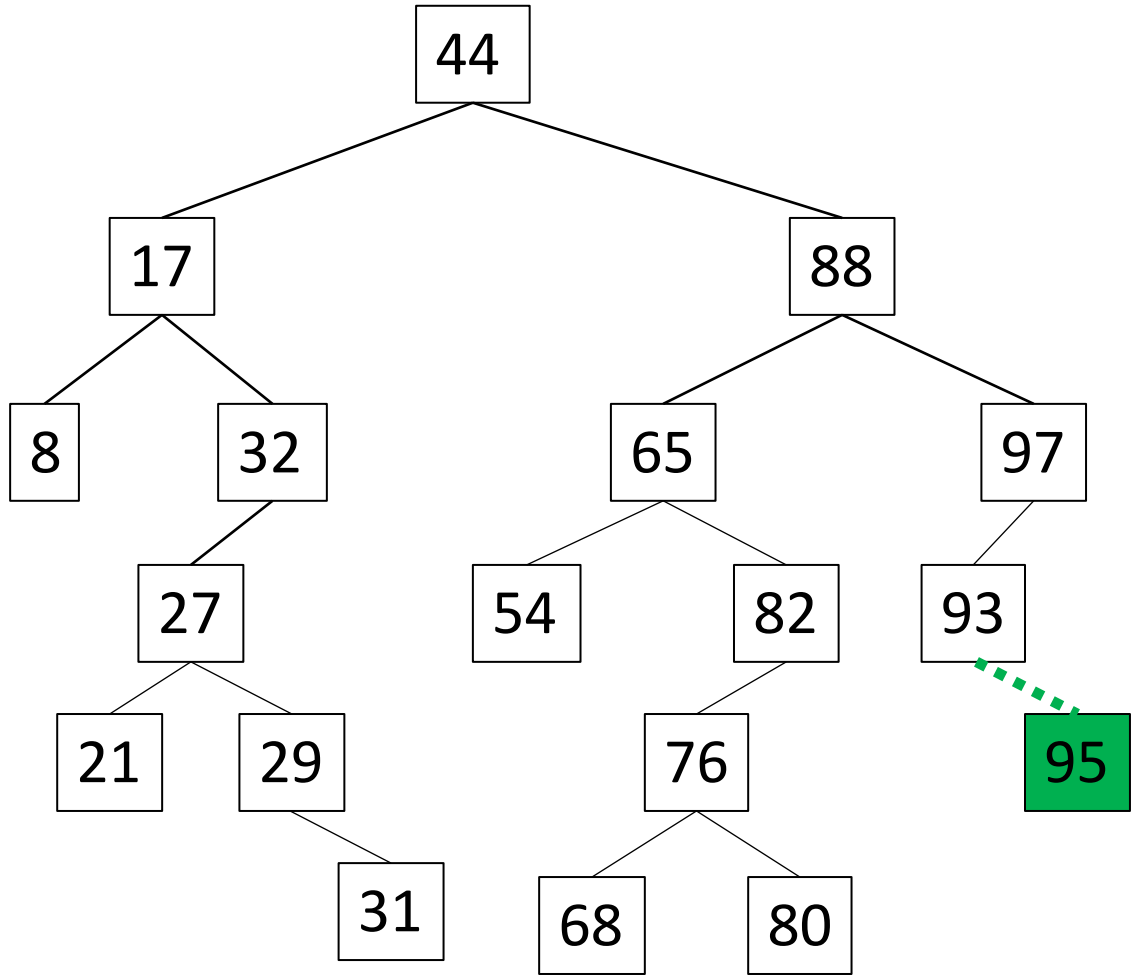


Binary Search Tree - Insertion

```
insert(95);
```

Step 1: Find where it should go.

Step 2: Modify pointers.



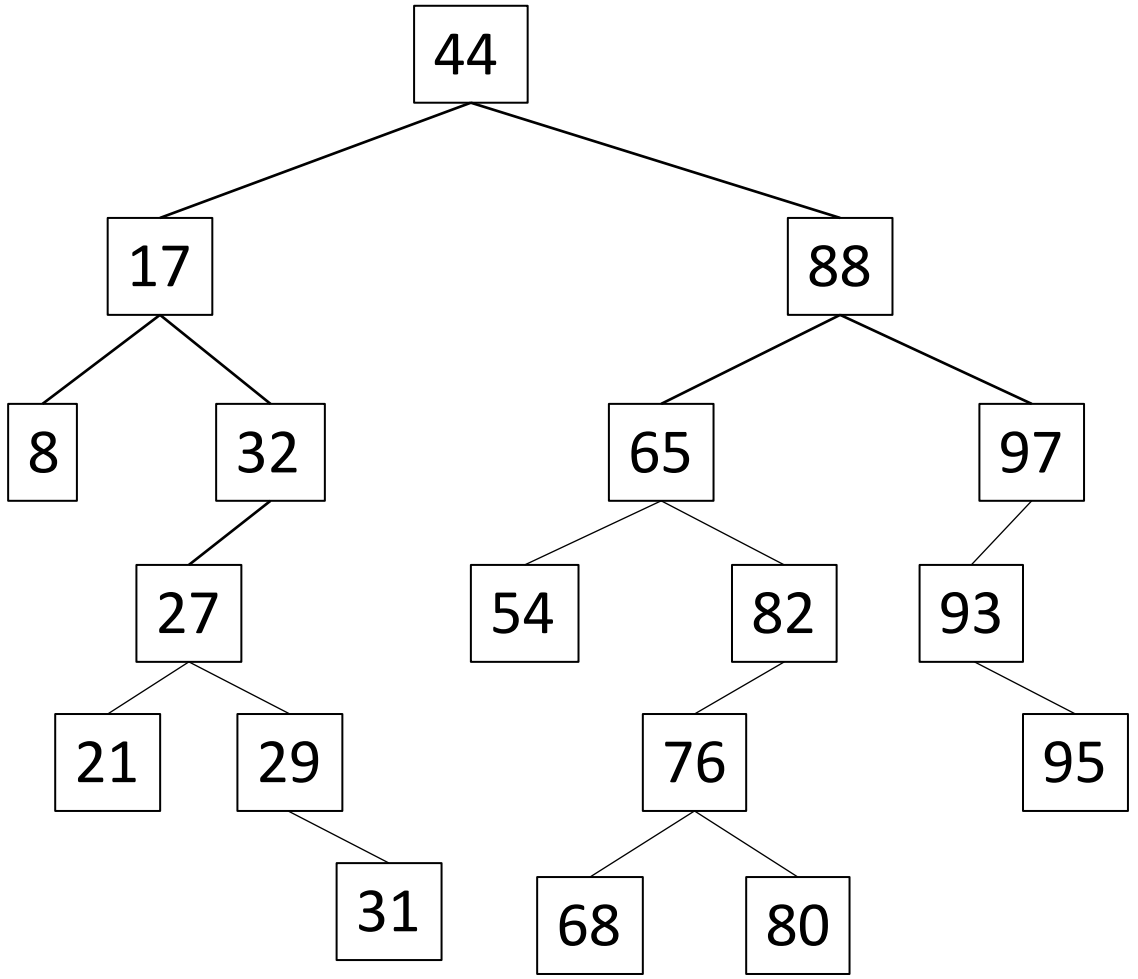
Binary Search Tree - Insertion

```
insert(95);
```

Step 1: Find where it should go.

Step 2: Modify pointers.

Any trends??



Binary Search Tree - Insertion

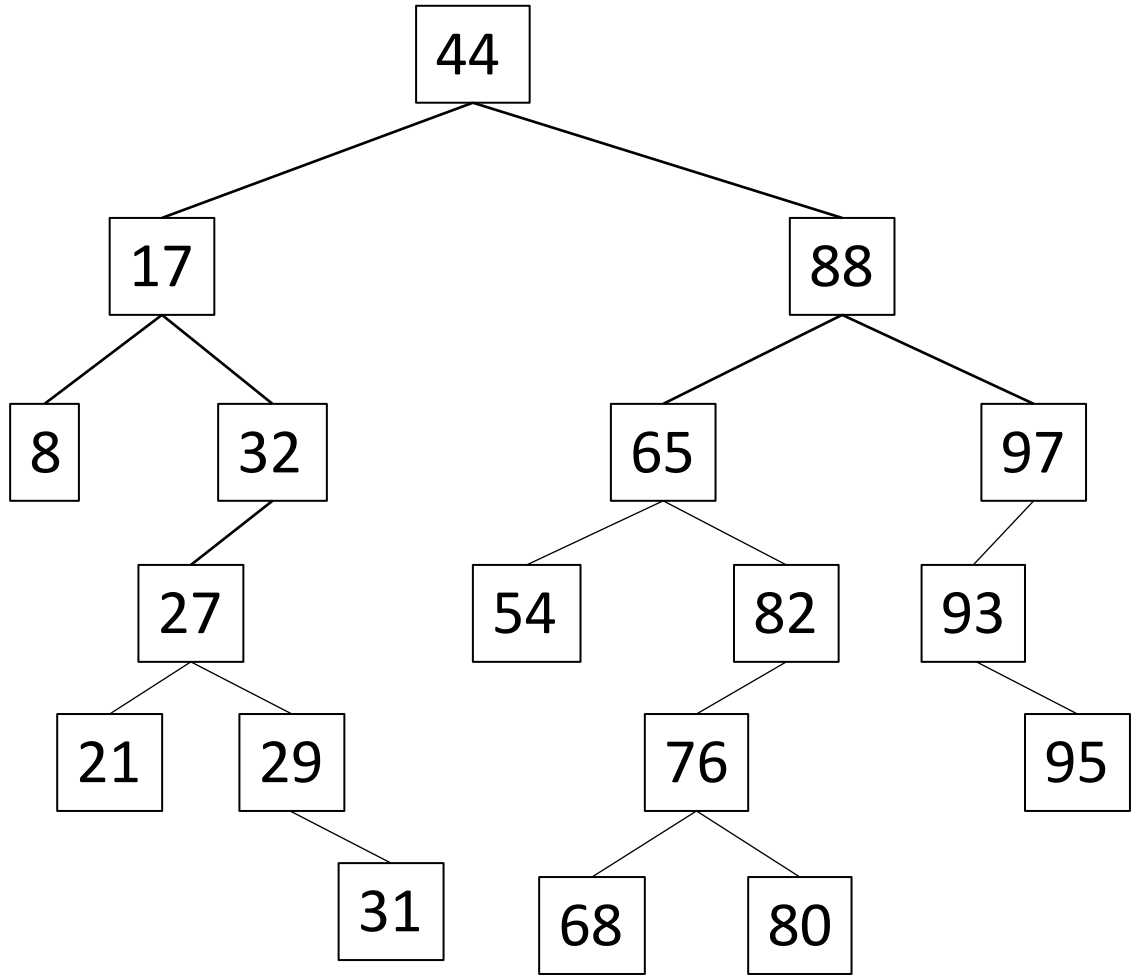
```
insert(95);
```

Step 1: Find where it should go.

Step 2: Modify pointers.

Any trends??

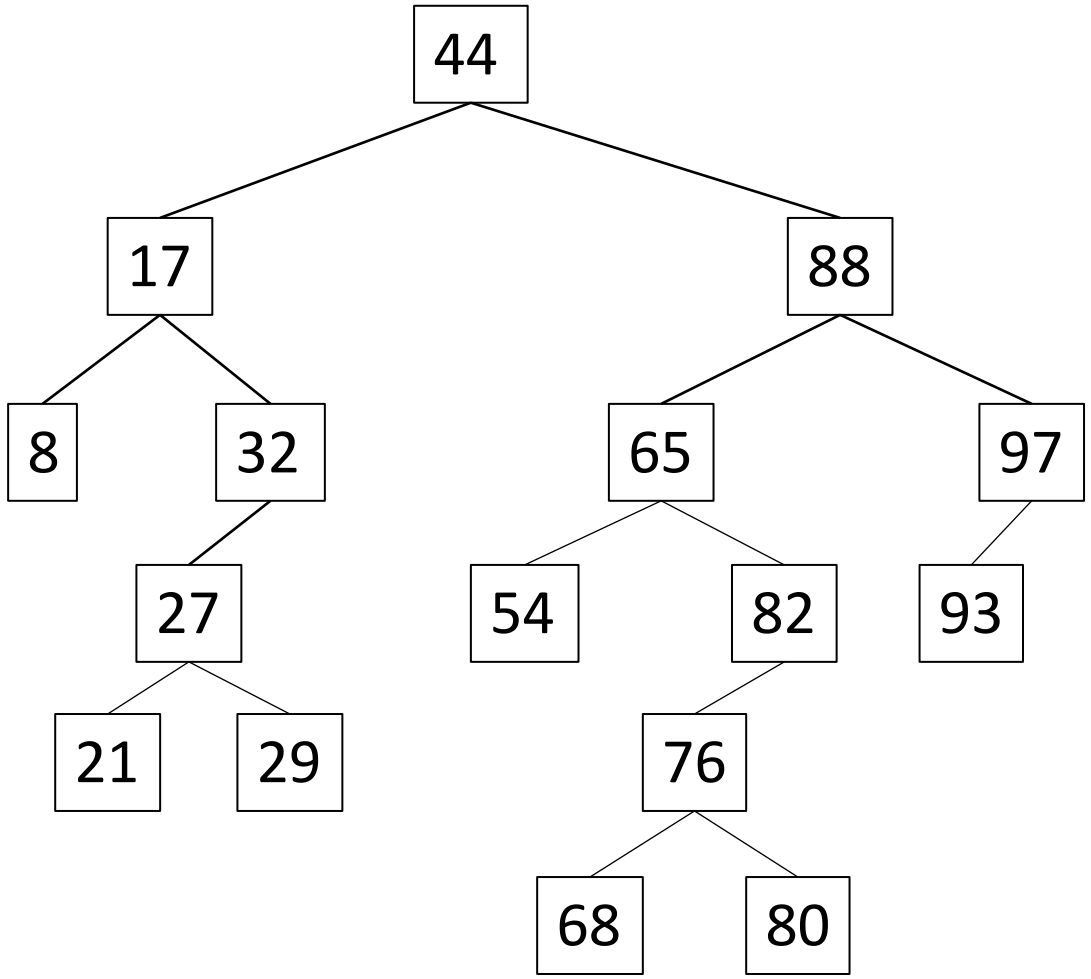
Always insert a new leaf!



Binary Search Tree - Insertion

```
insert(28);
```

```
public void insert(int newValue) {

}
```



Binary Search Tree - Insertion

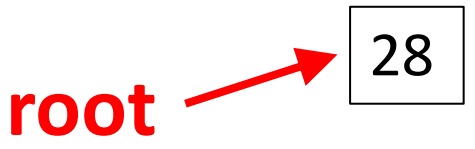
```
insert(28);
```

```
public void insert(int newValue) {
  if (root == null) {

  } else {

}
```

Binary Search Tree - Insertion

```
insert(28);
```

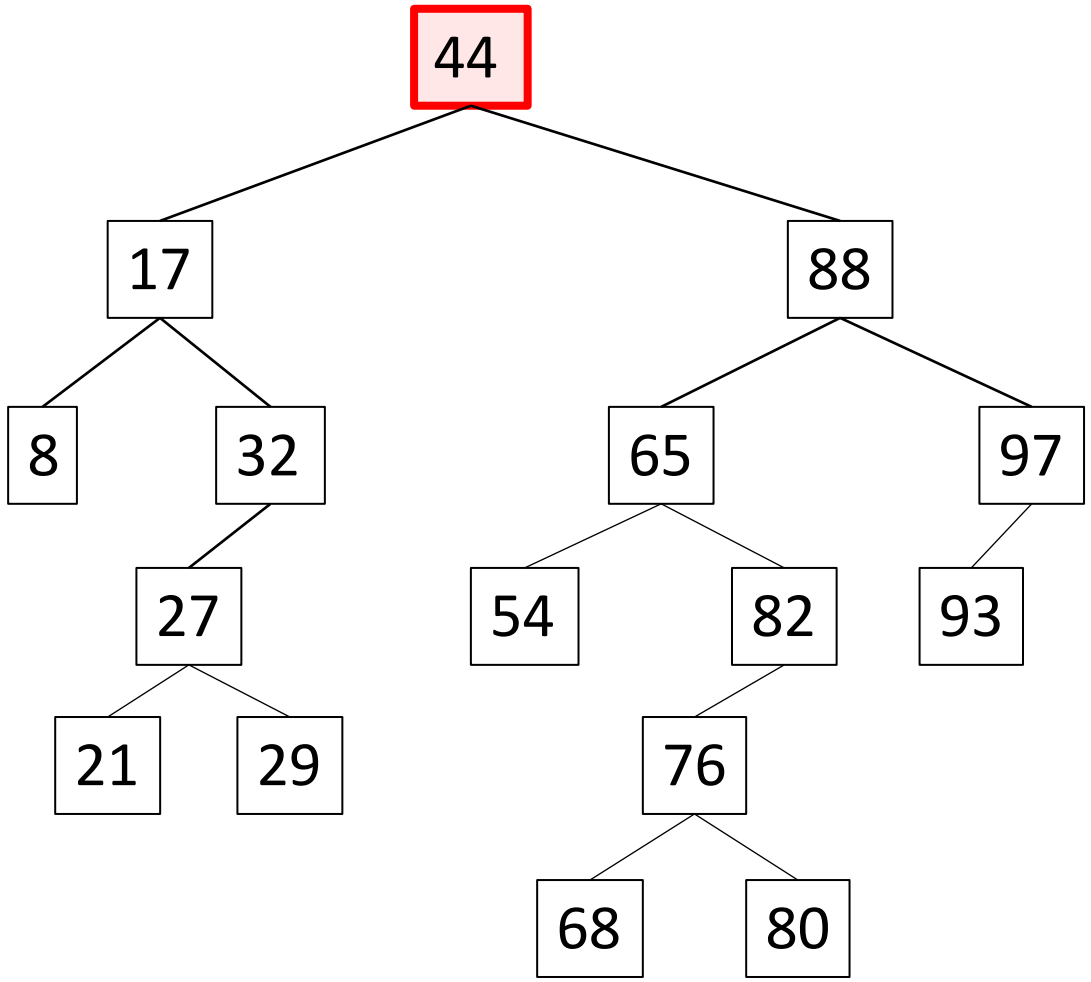
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {




}
```



Binary Search Tree - Insertion

```
insert(28);
```

```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
```

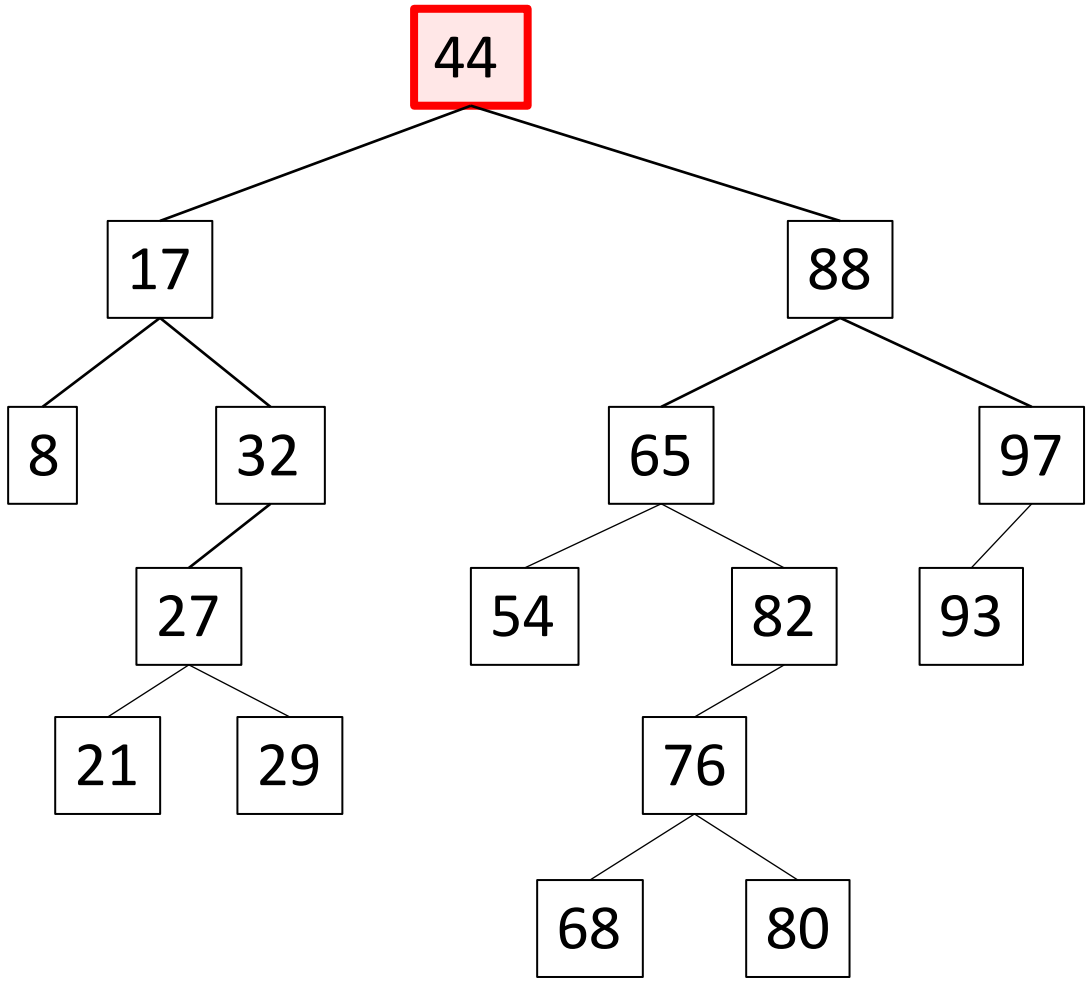



```
}
```

Binary Search Tree - Insertion

```
insert(28);
```

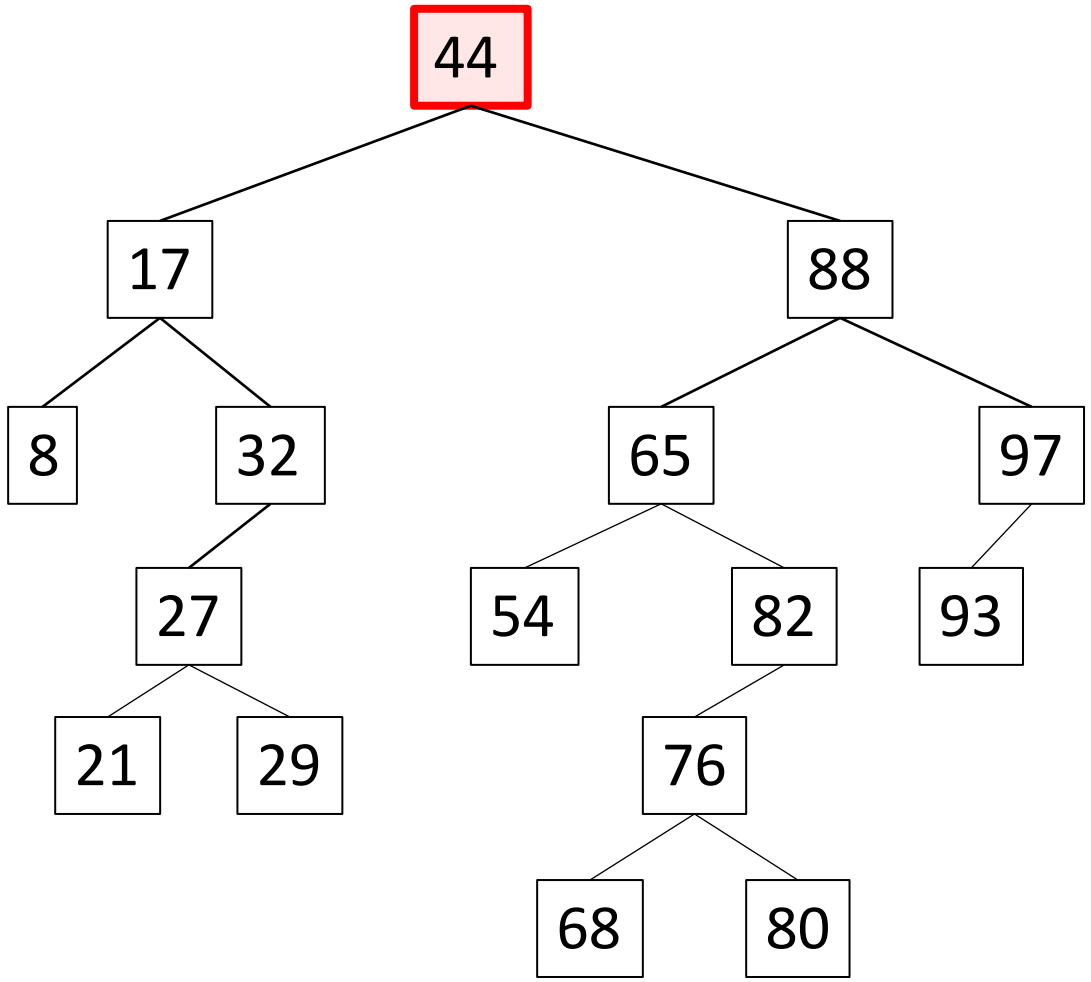
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;




}
```



Binary Search Tree - Insertion

```
insert(28);
```

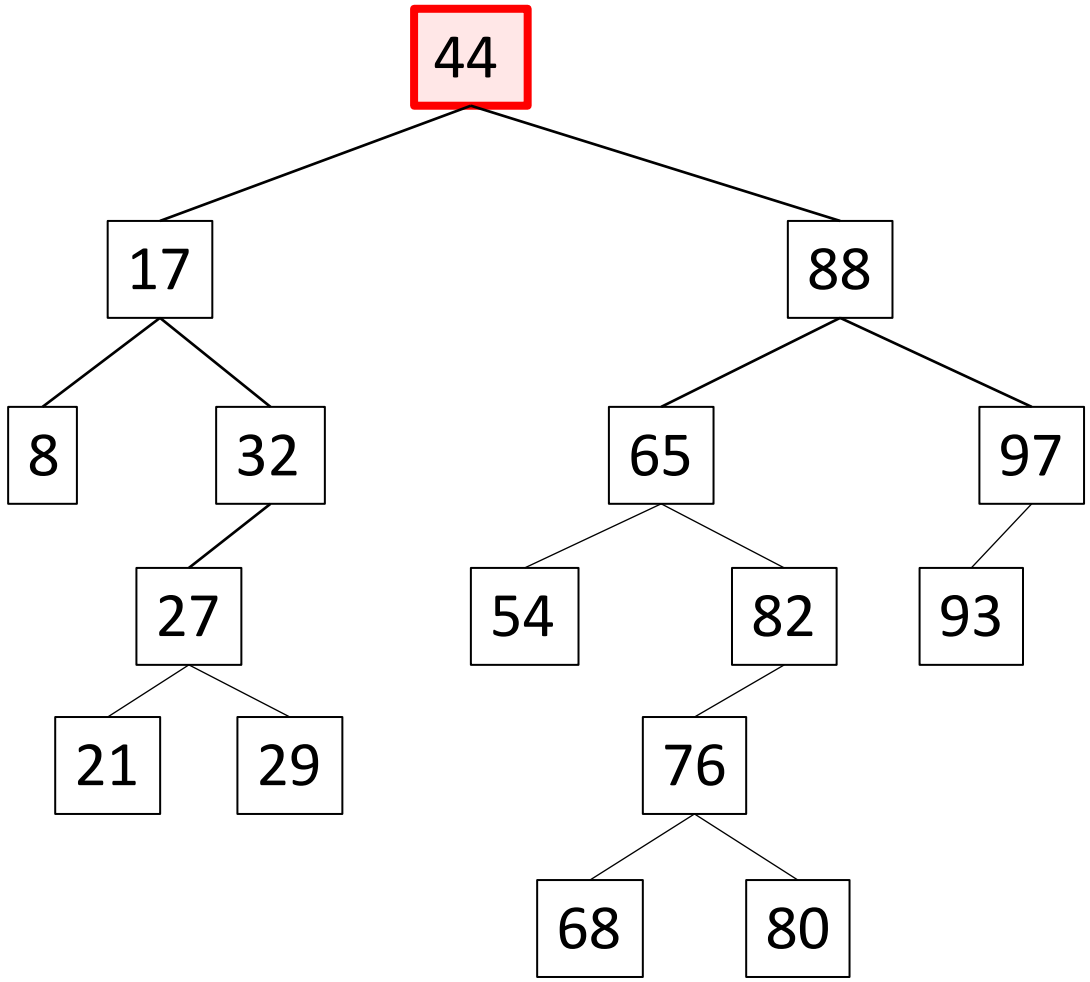
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {




}
```



Binary Search Tree - Insertion

```
insert(28);
```

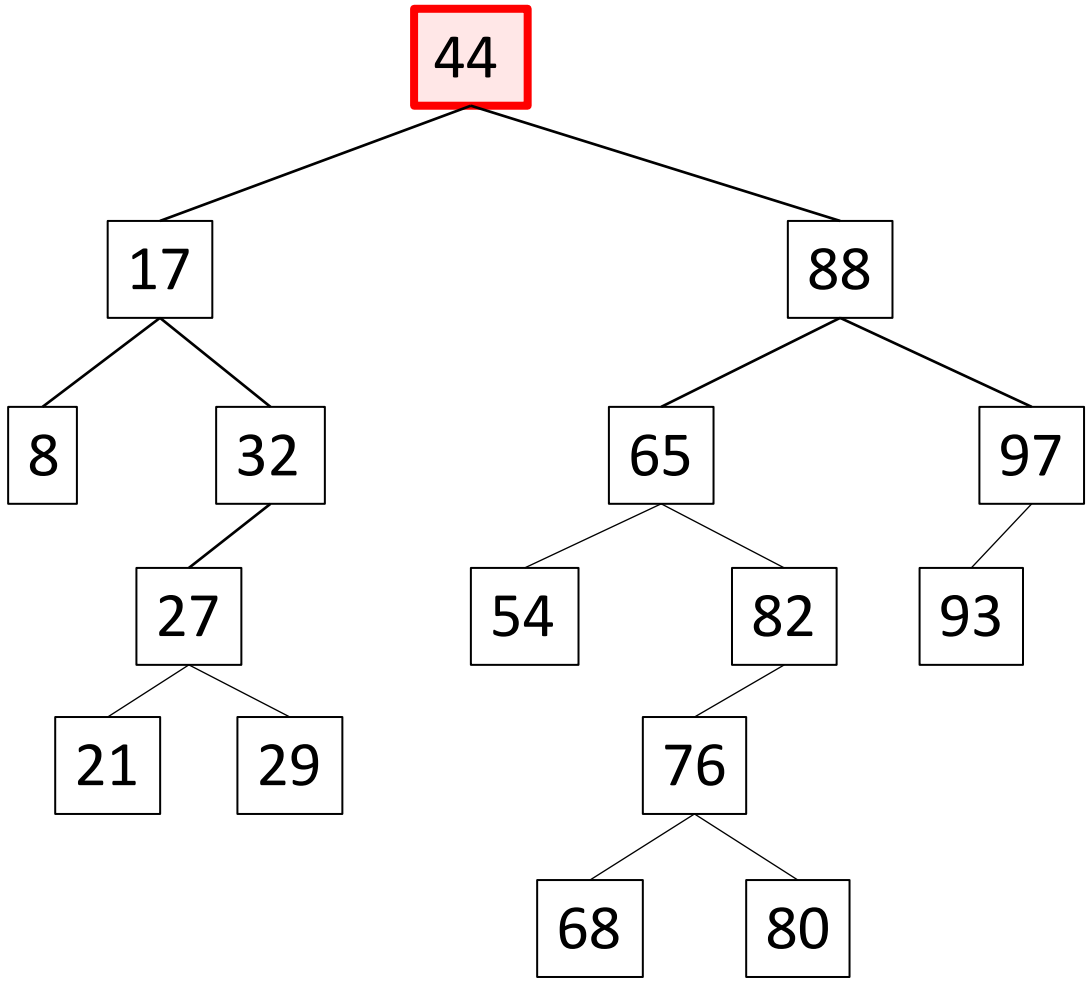
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {









      } else {









}
```



Binary Search Tree - Insertion

```
insert(28);
```

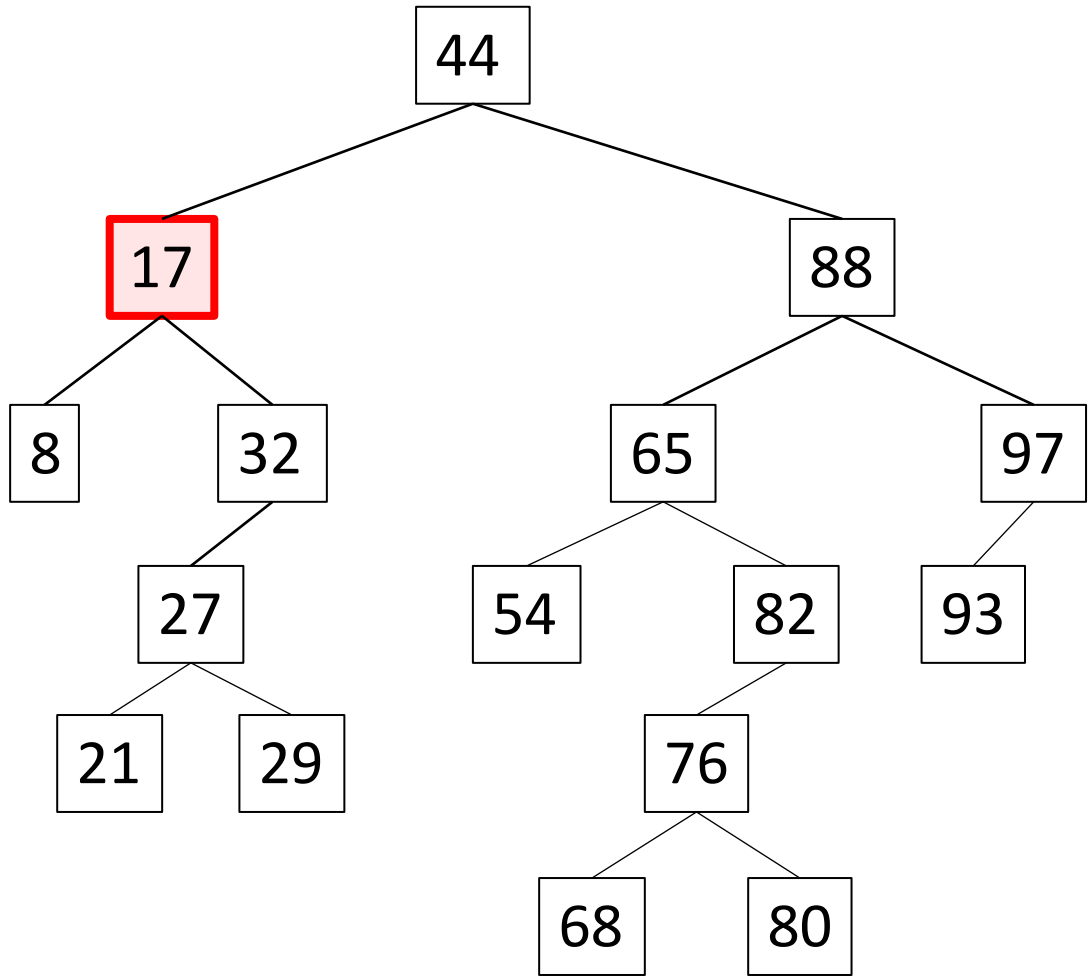
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {

        } else {




        }
      } else {









}
```



Binary Search Tree - Insertion

```
insert(28);
```

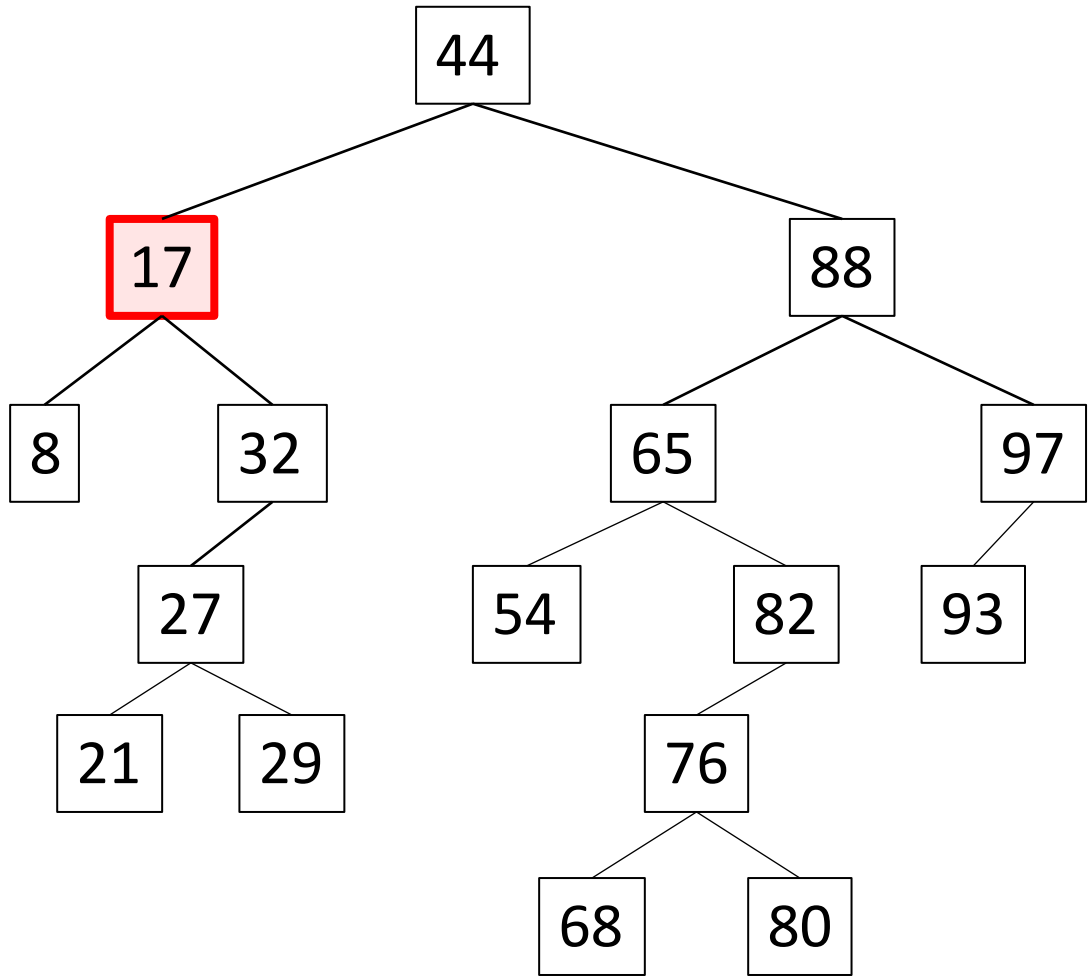
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {









}
```



Binary Search Tree - Insertion

```
insert(28);
```

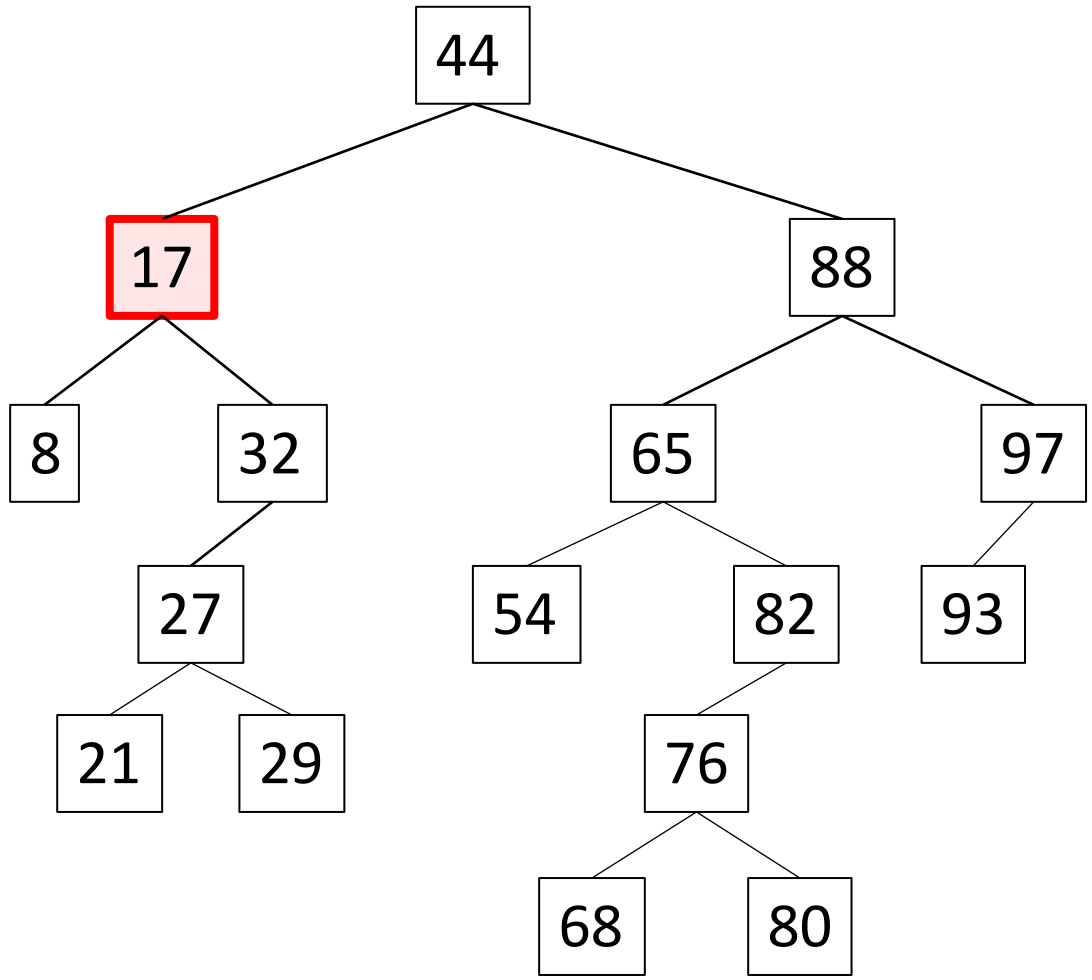
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {




}
```



Binary Search Tree - Insertion

```
insert(28);
```

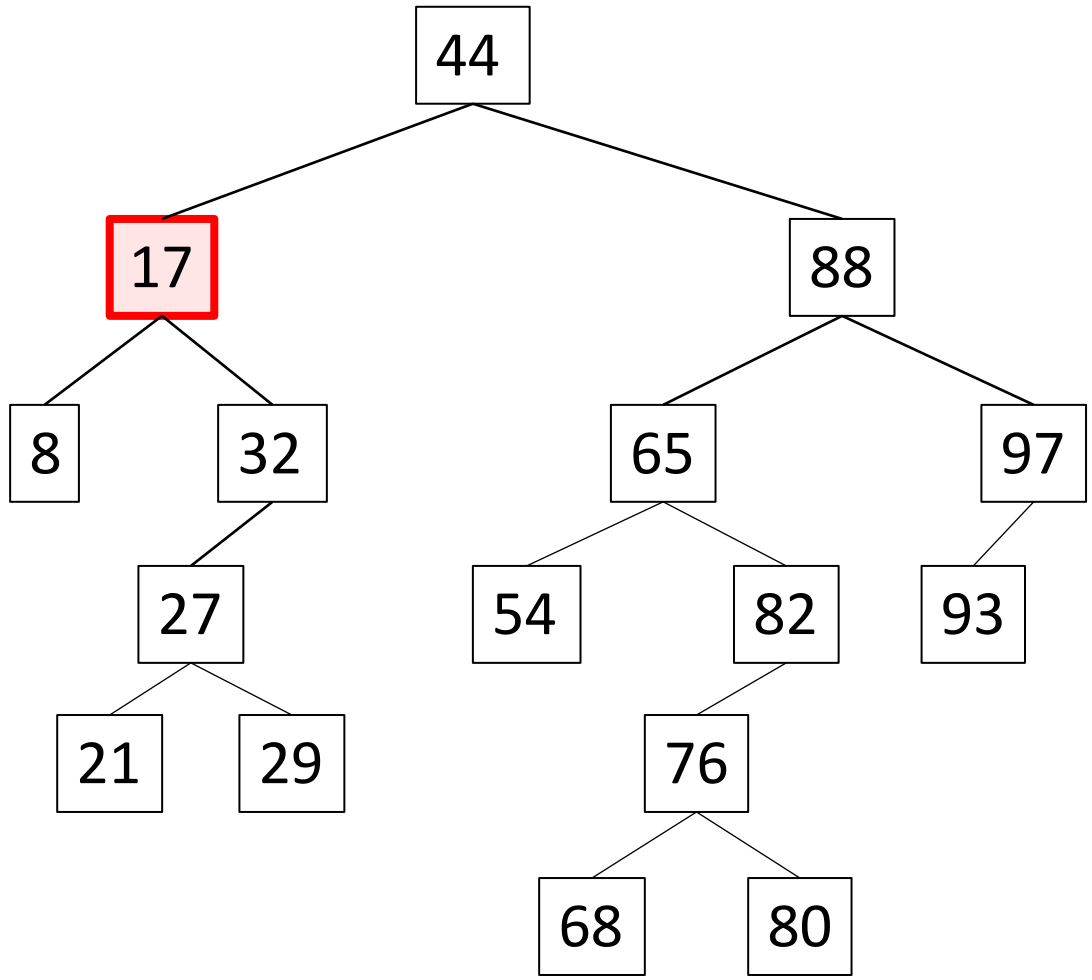
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {



        }
      } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

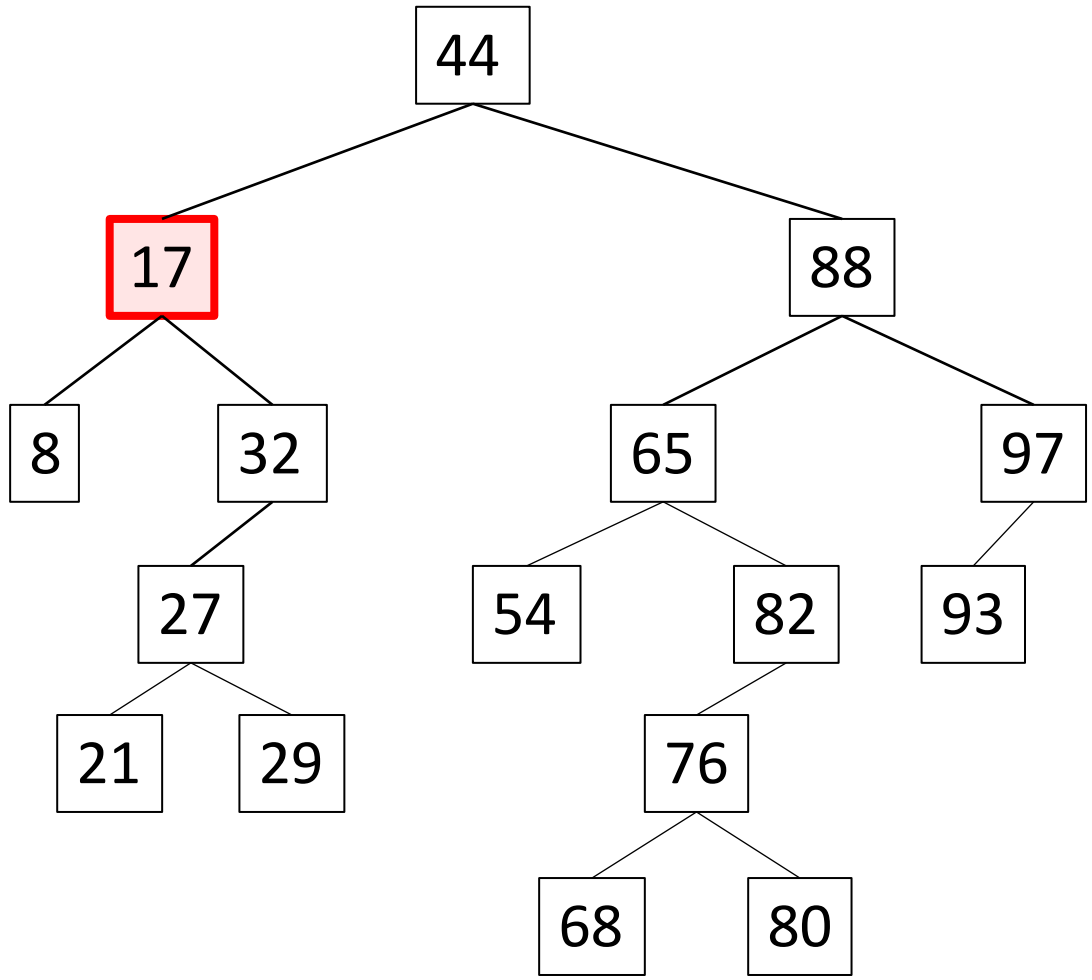
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {



        }
      } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

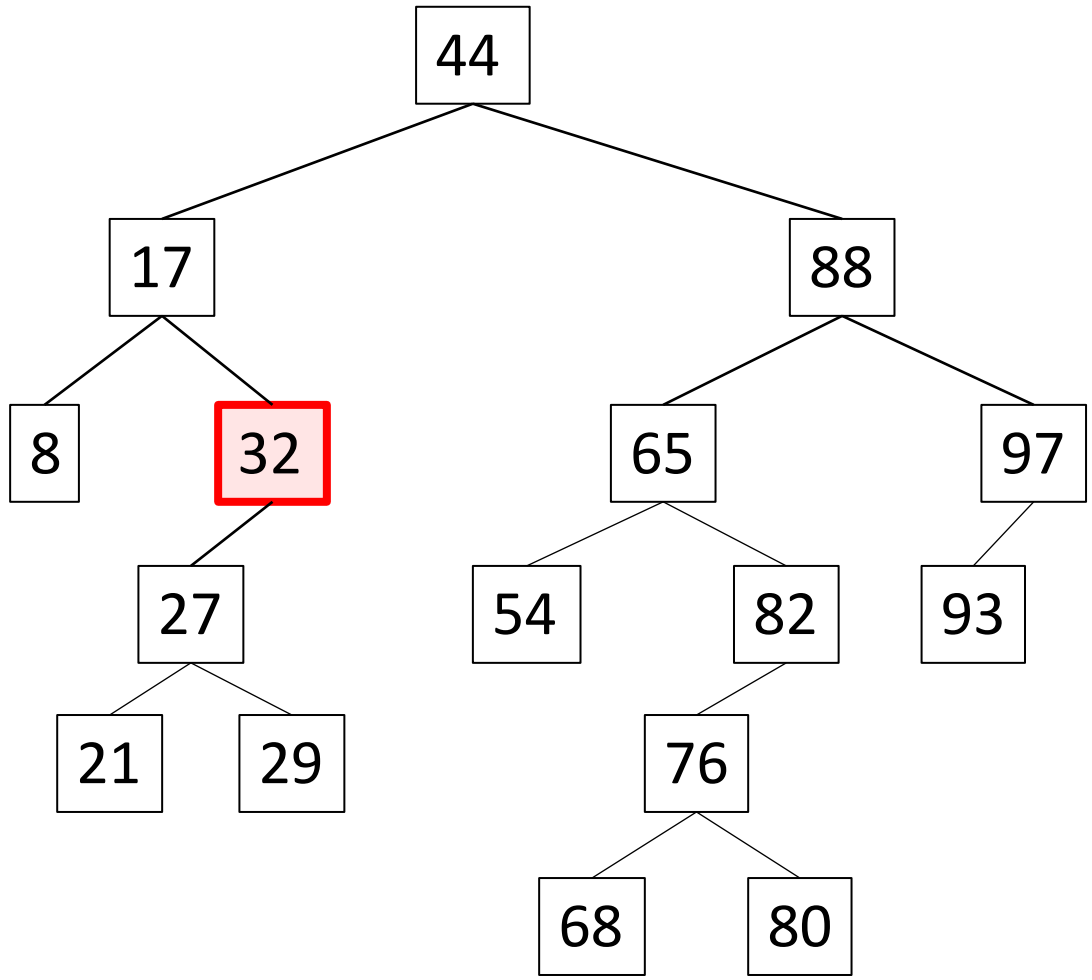
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {



        }
      } else {
        if (currentNode.getRight() != null) {

        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

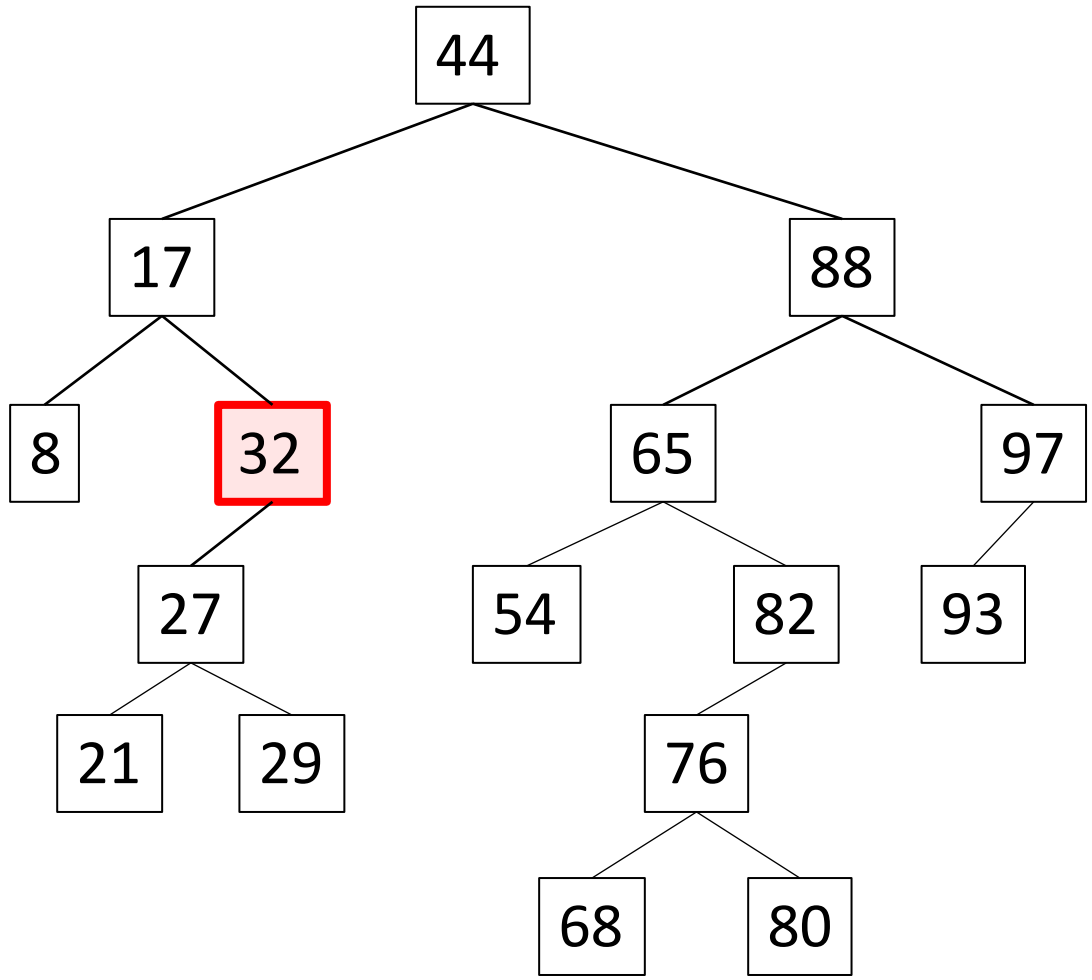
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

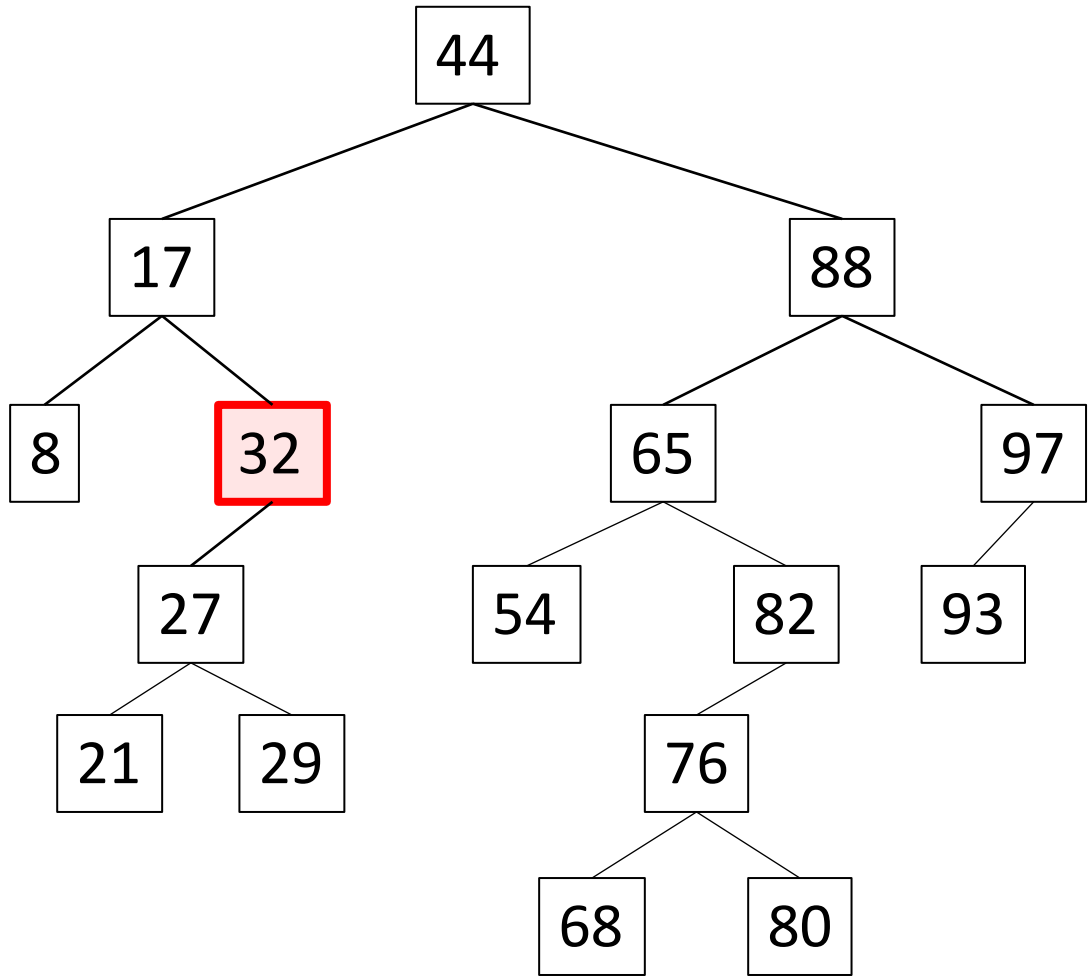
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

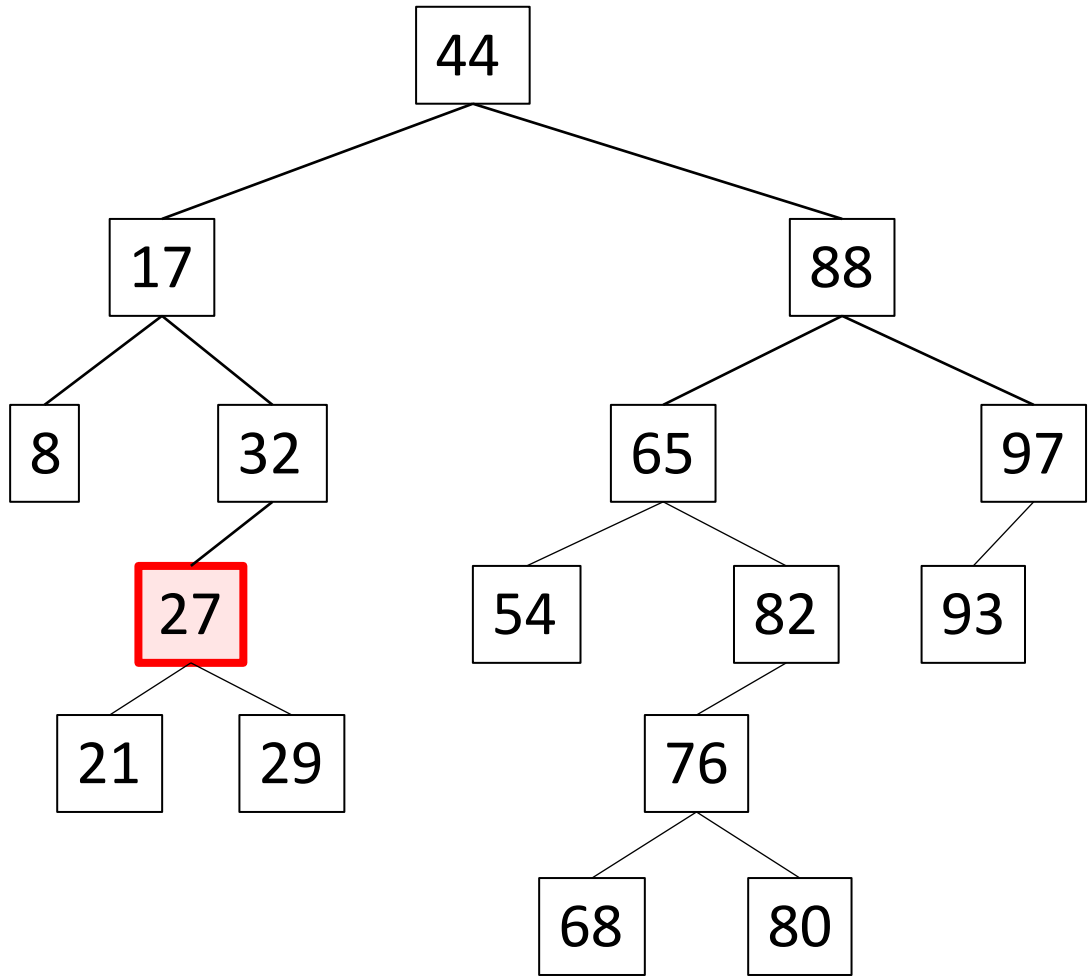
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

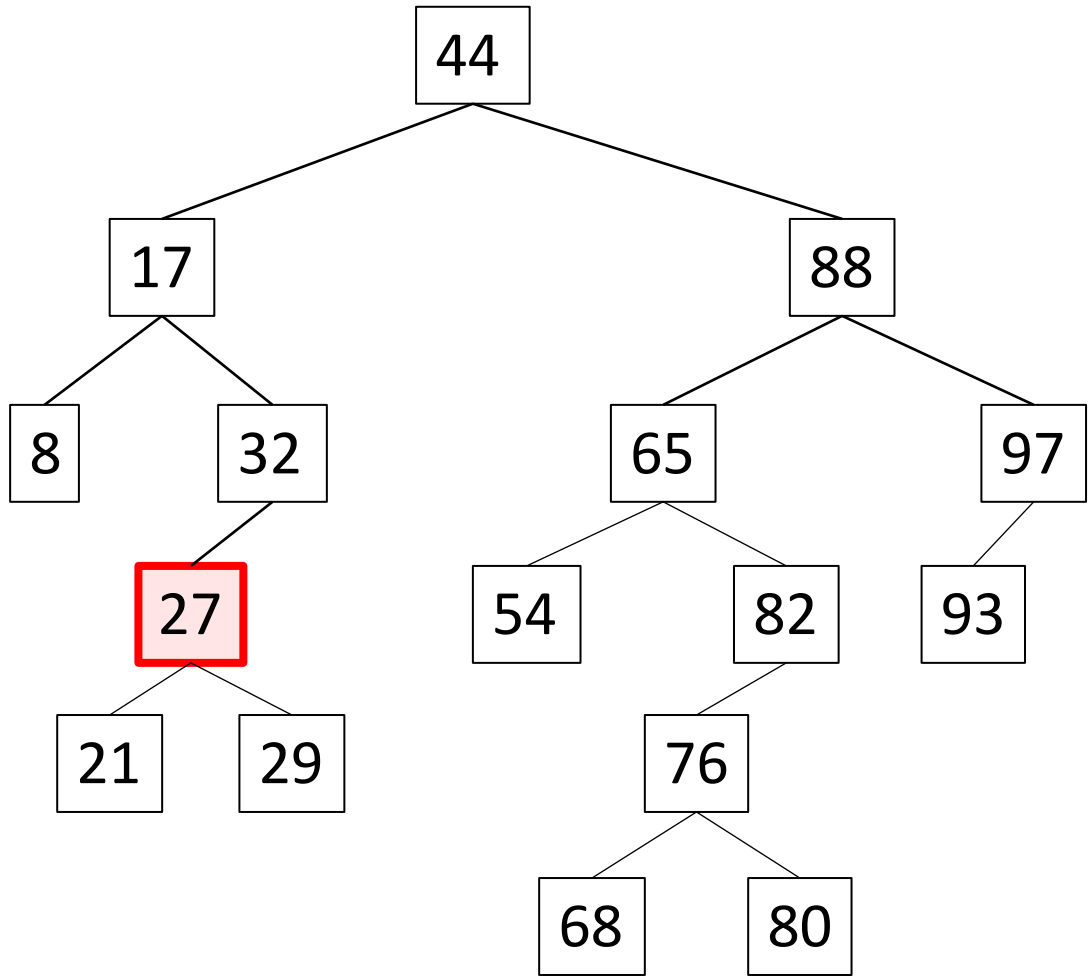
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

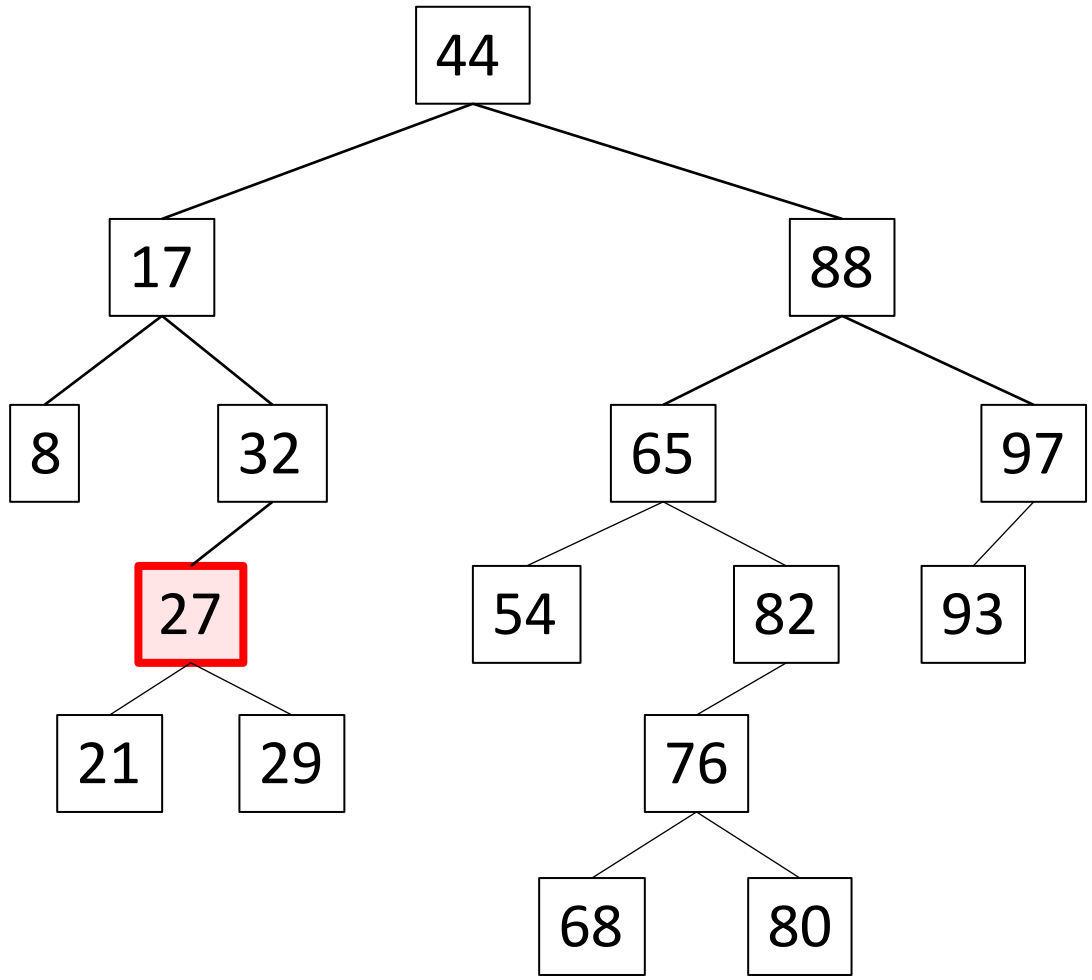
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

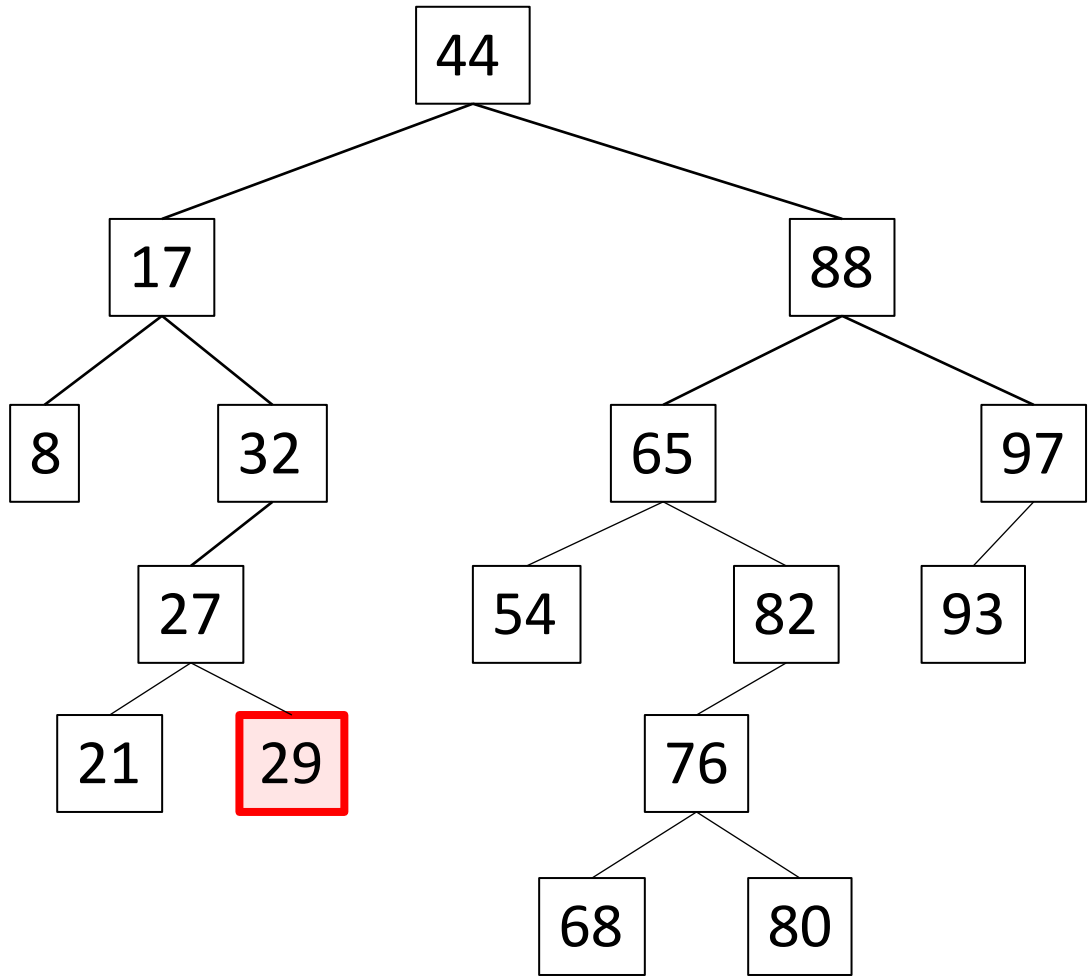
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {


}
```



Binary Search Tree - Insertion

```
insert(28);
```

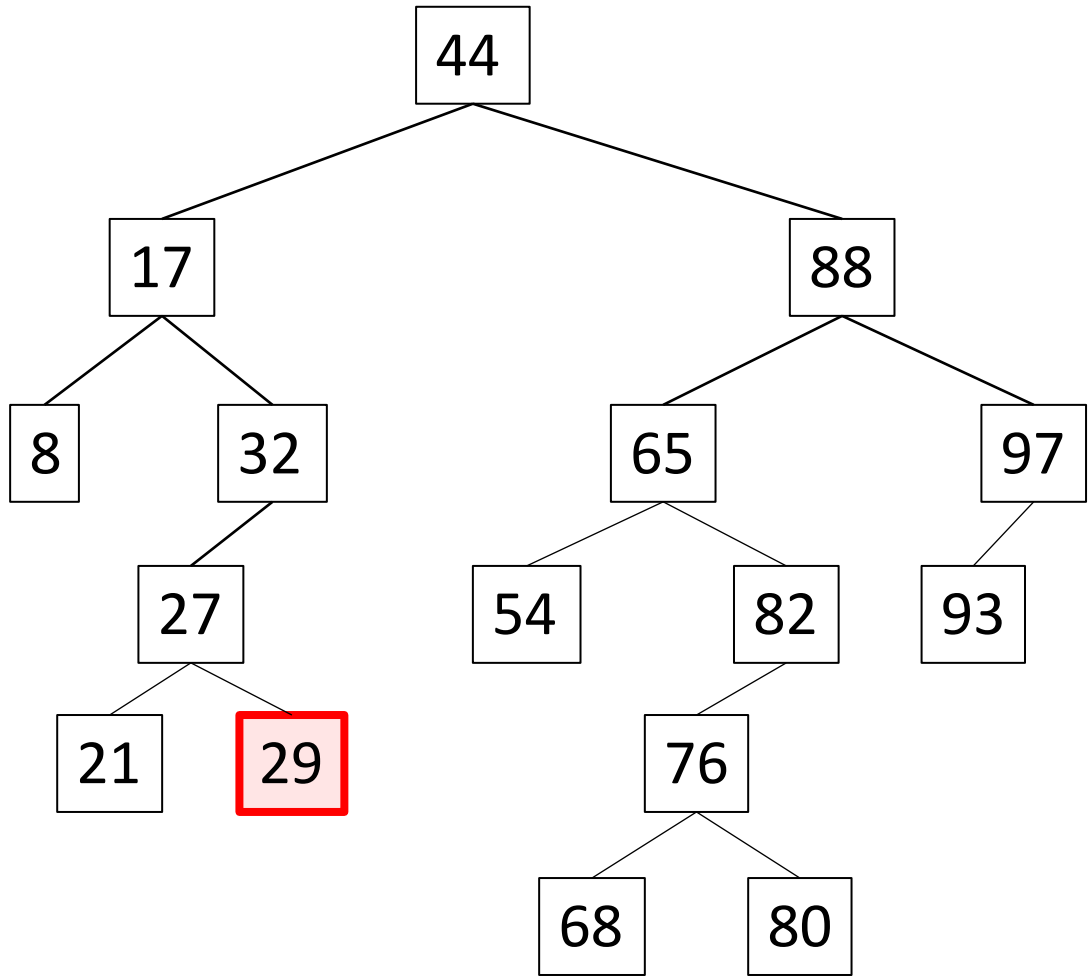
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

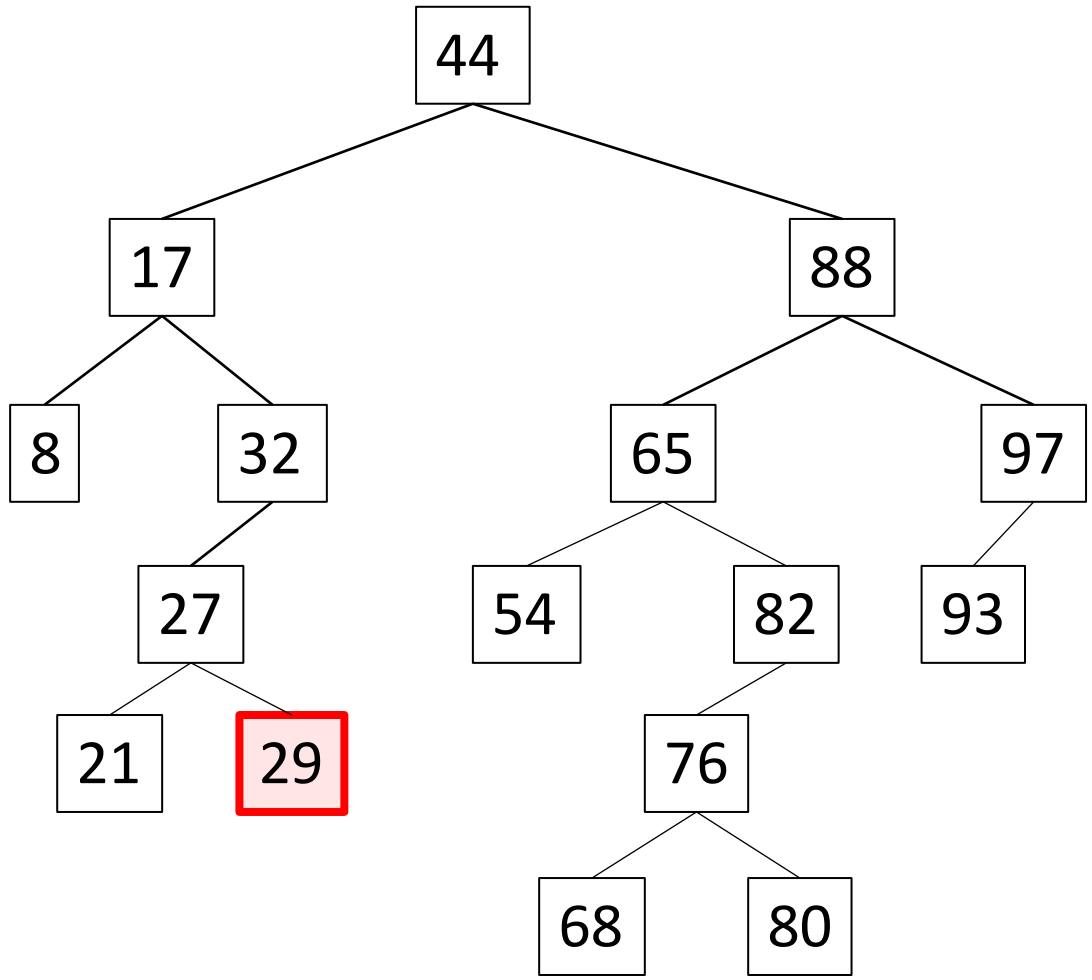
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

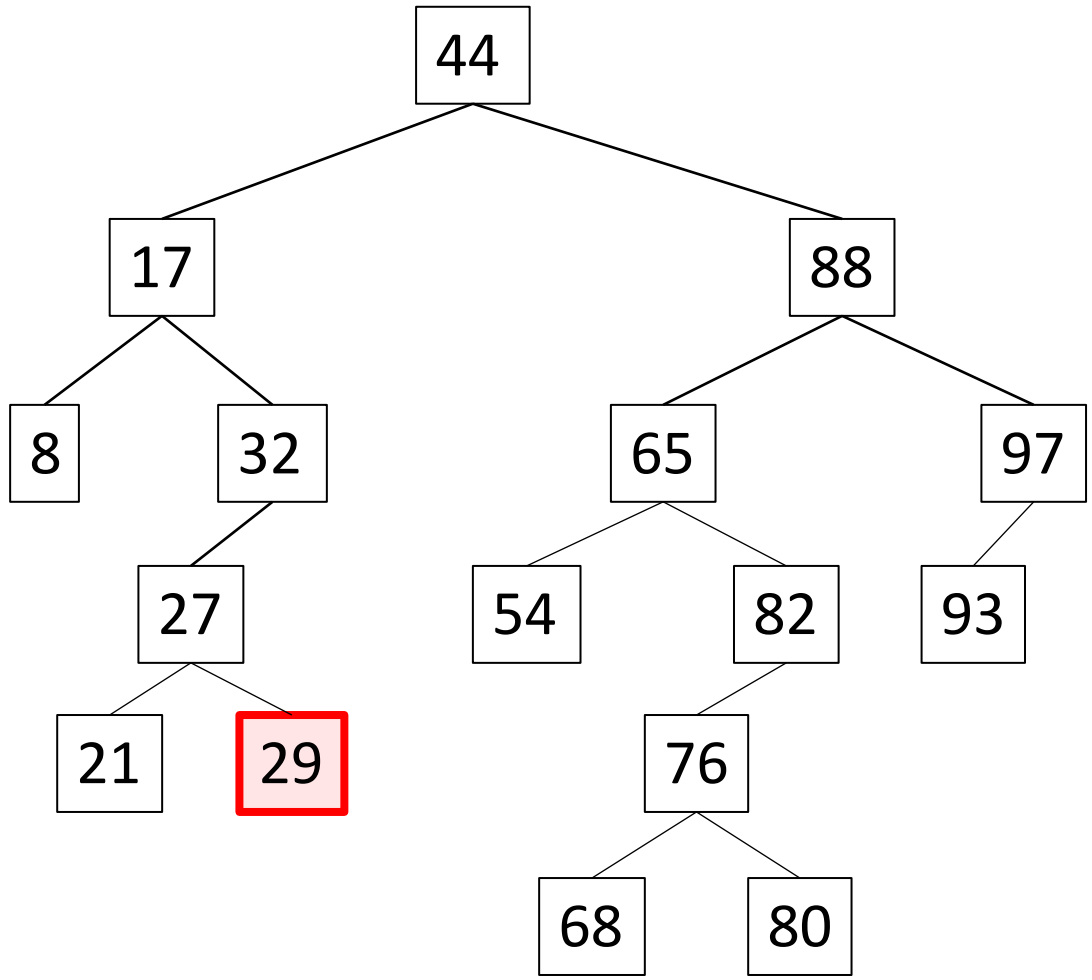
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

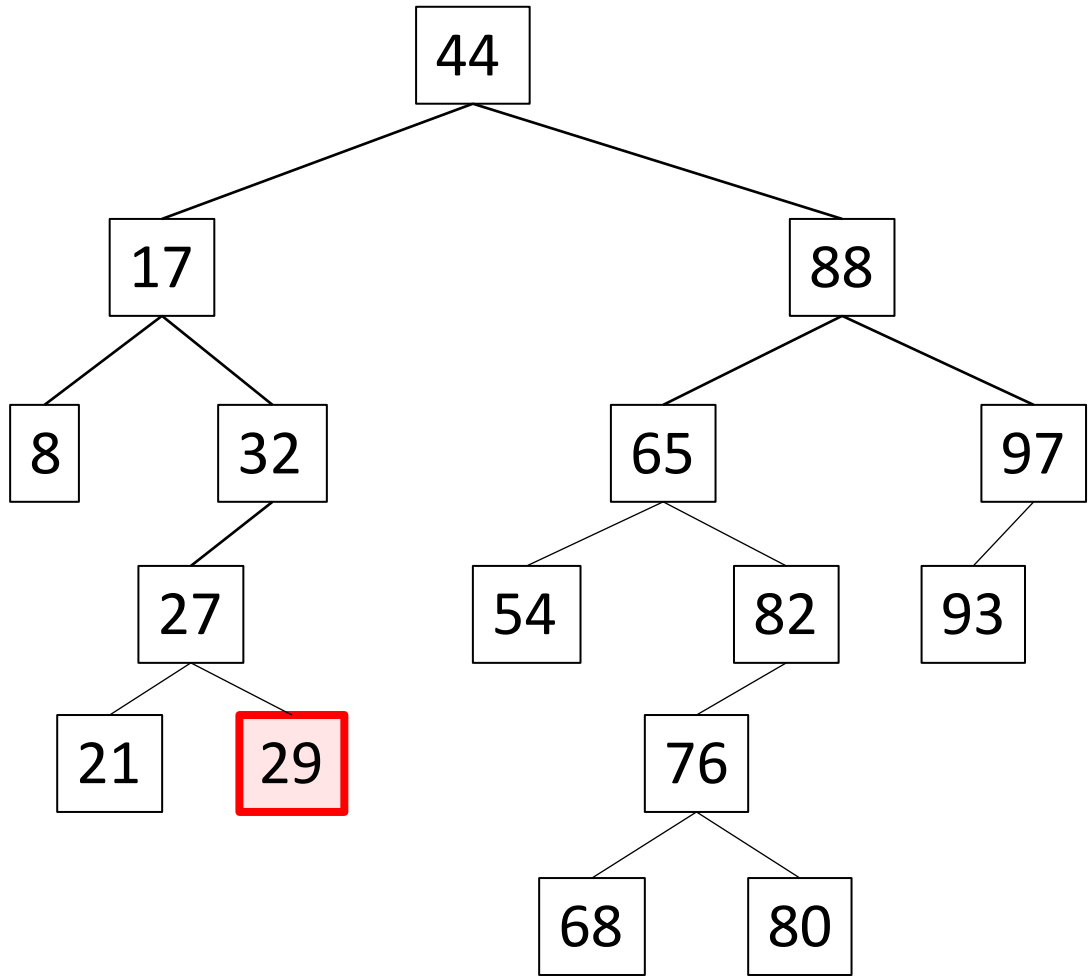
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {




        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

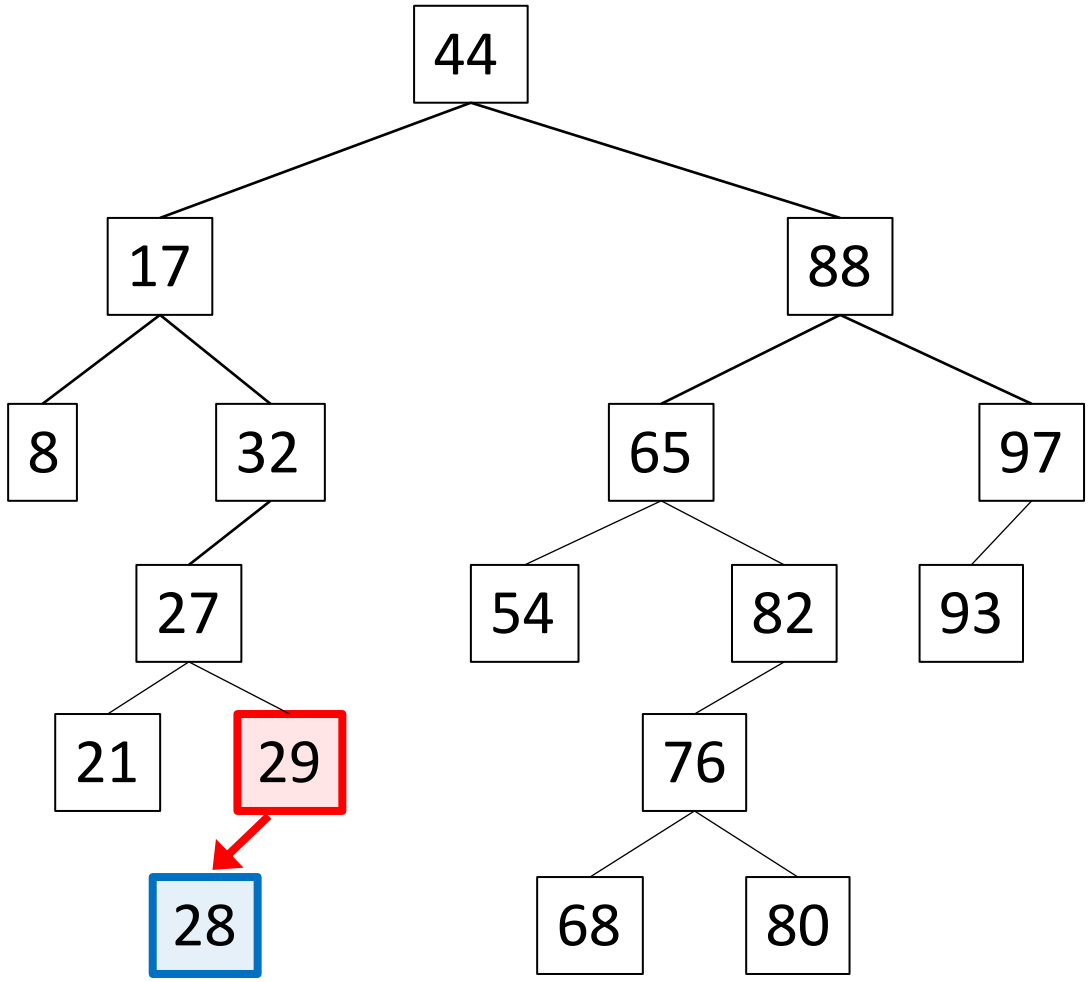
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {



        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {


}
```



Binary Search Tree - Insertion

```
insert(28);
```

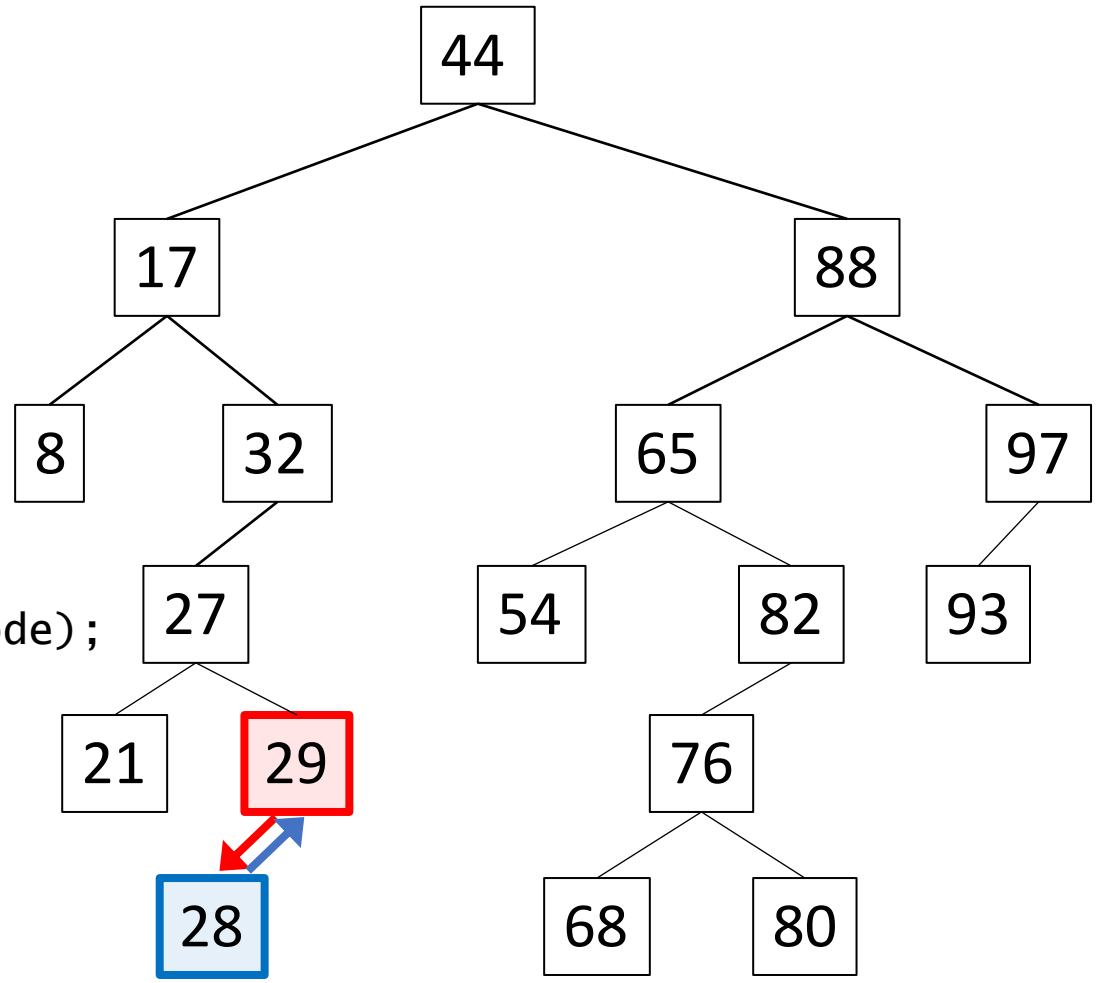
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));



        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

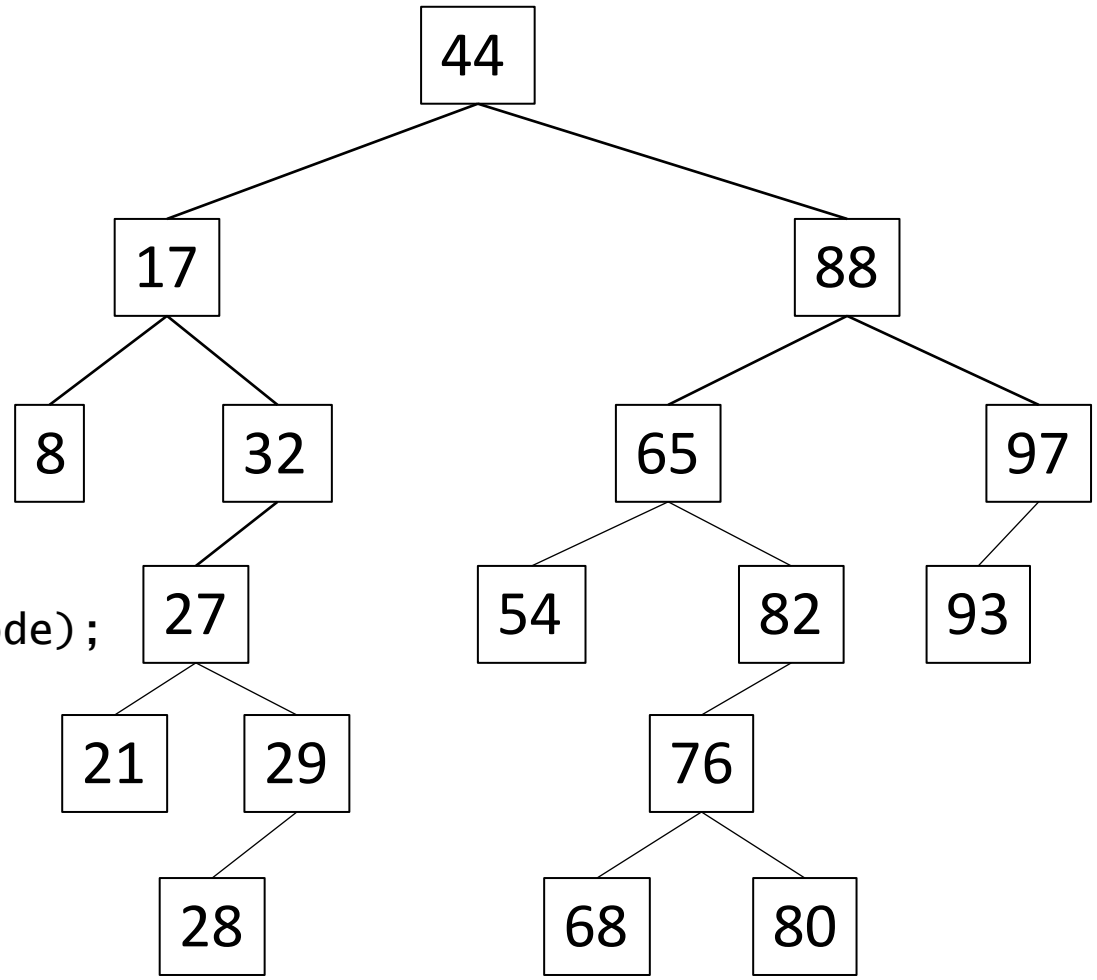
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);

        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {



}
```



Binary Search Tree - Insertion

```
insert(28);
```

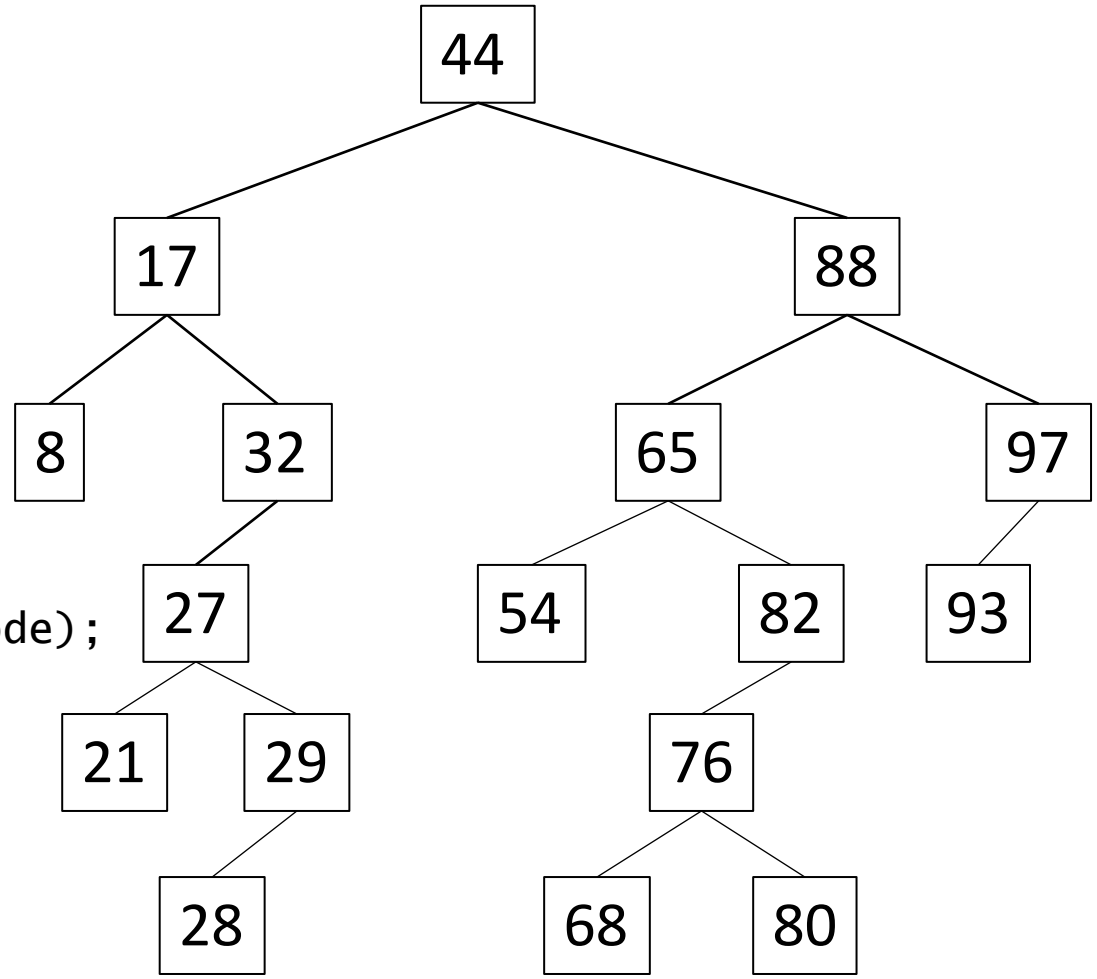
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);
          placed = true;
        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {


}
```



Binary Search Tree - Insertion

```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);
          placed = true;
        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {

}
```

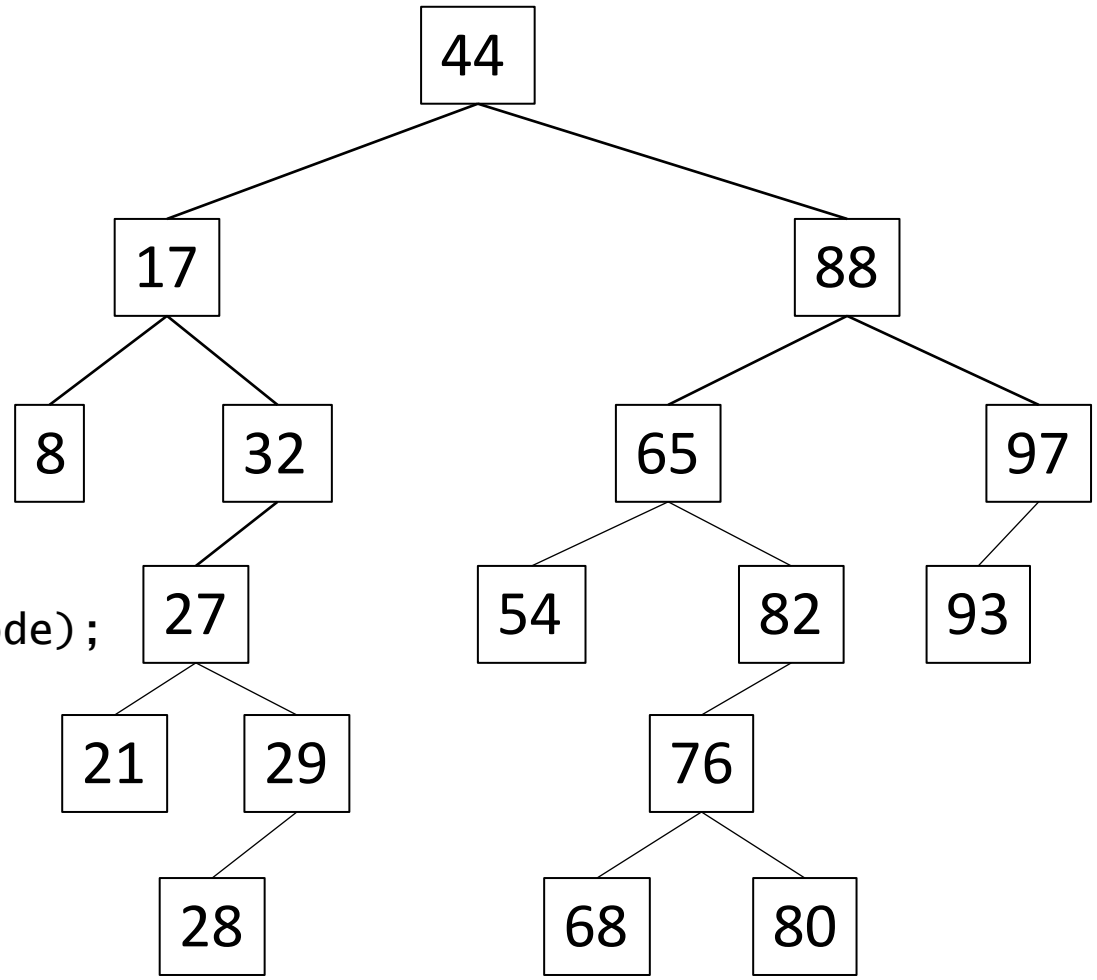
```
insert(28);
```



Binary Search Tree - Insertion

```
insert(28);
```

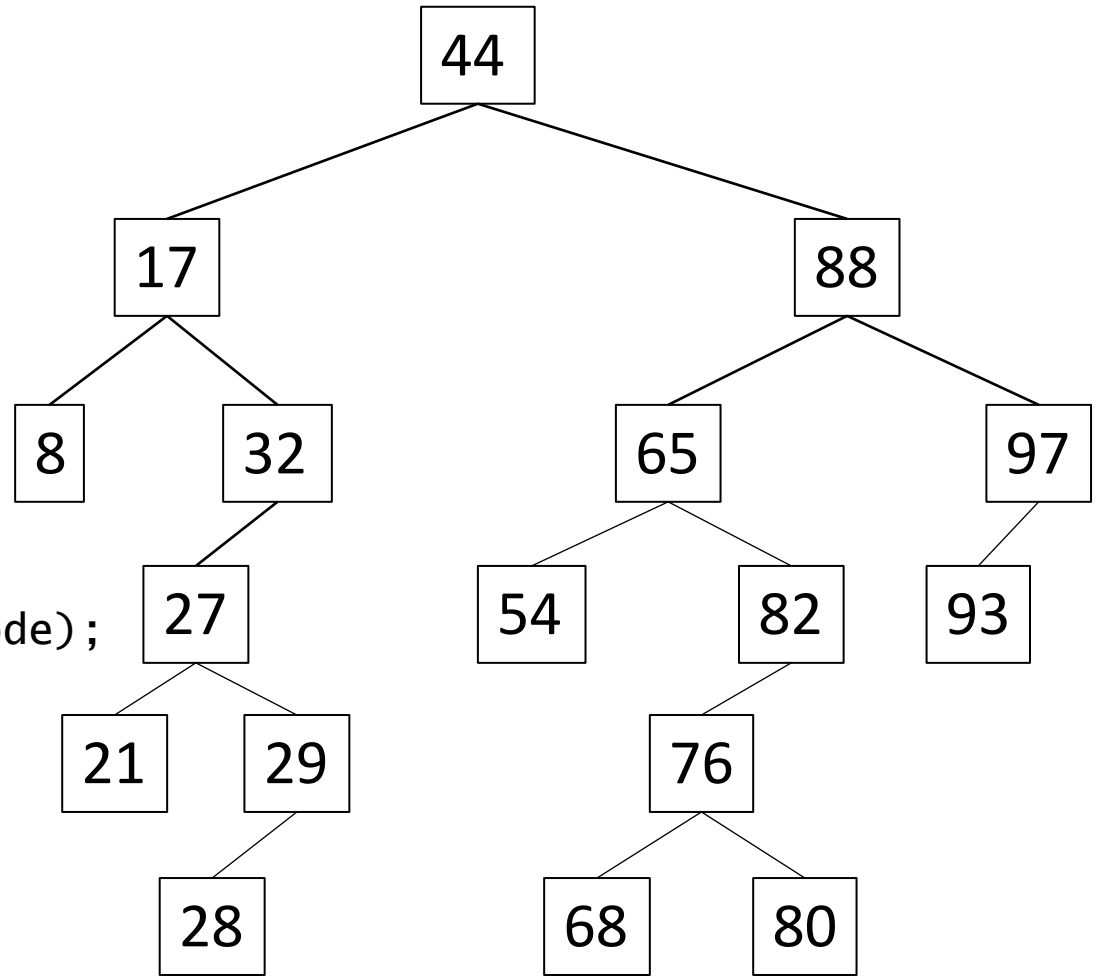
```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);
          placed = true;
        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {
          currentNode.setRight(new Node(newValue));


}
```



Binary Search Tree - Insertion

```
insert(28);
```

```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);
          placed = true;
        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {
          currentNode.setRight(new Node(newValue));
          currentNode.getRight().setParent(currentNode);

}
```



Binary Search Tree - Insertion

```
insert(28);
```

```
public void insert(int newValue) {
  if (root == null) {
    root = new Node(newValue);
  } else {
    Node currentNode = root;
    boolean placed = false;
    while (!placed) {
      if (newValue < currentNode.getValue()) {
        if (currentNode.getLeft() != null) {
          currentNode = currentNode.getLeft();
        } else {
          currentNode.setLeft(new Node(newValue));
          currentNode.getLeft().setParent(currentNode);
          placed = true;
        }
      } else {
        if (currentNode.getRight() != null) {
          currentNode = currentNode.getRight();
        } else {
          currentNode.setRight(new Node(newValue));
          currentNode.getRight().setParent(currentNode);
          placed = true;
}
```