Data Structures

CSCI 2270-202: REC 08

Sanskar Katiyar

Logistics

Office Hours at ECAE 128

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Recitation Outline

1. Binary Tree: Quick Recap
2. Binary Search Tree (BST)
3. BST: Search
4. BST: Insertion
5. BST: Deletion
6. BST: Analysis
7. Exercise

Binary Tree

Quick Recap

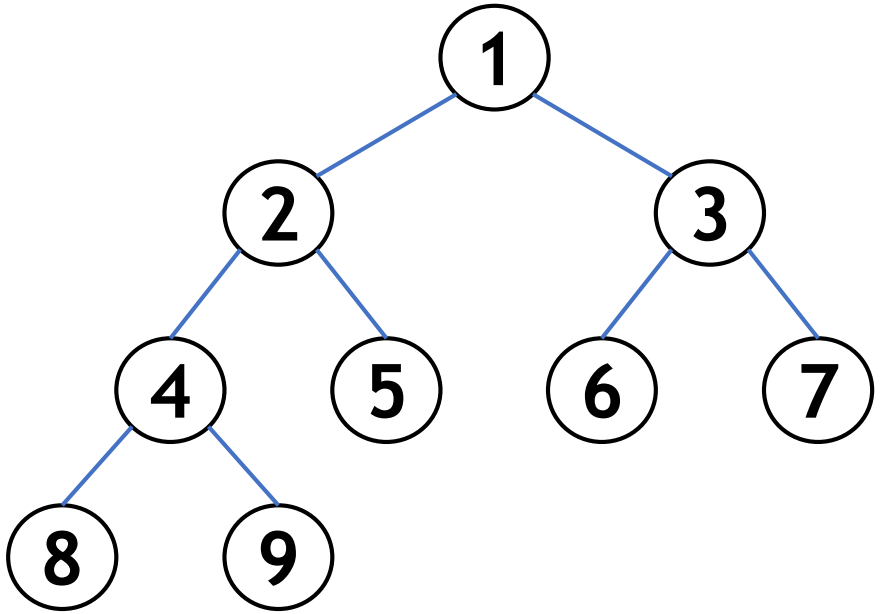
Tree ADT

Composed of **Nodes, Edges**

Non-linear, Hierarchical

n-ary: Each node can have ***n*** children

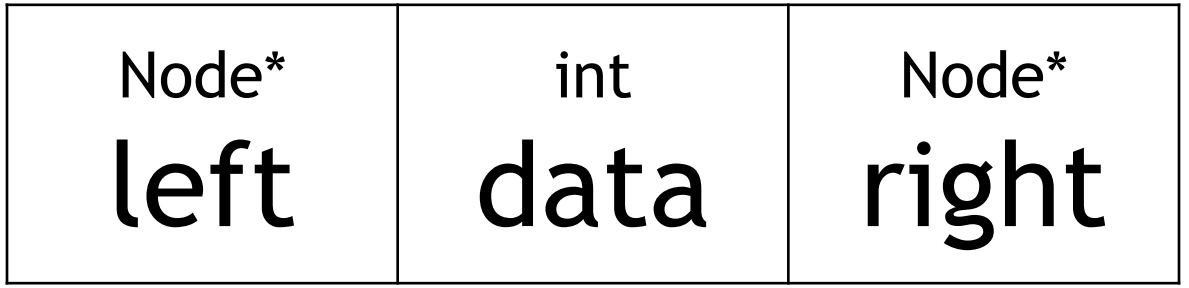
Example(s), Application(s):
Search, Expression Trees, Directory Structure



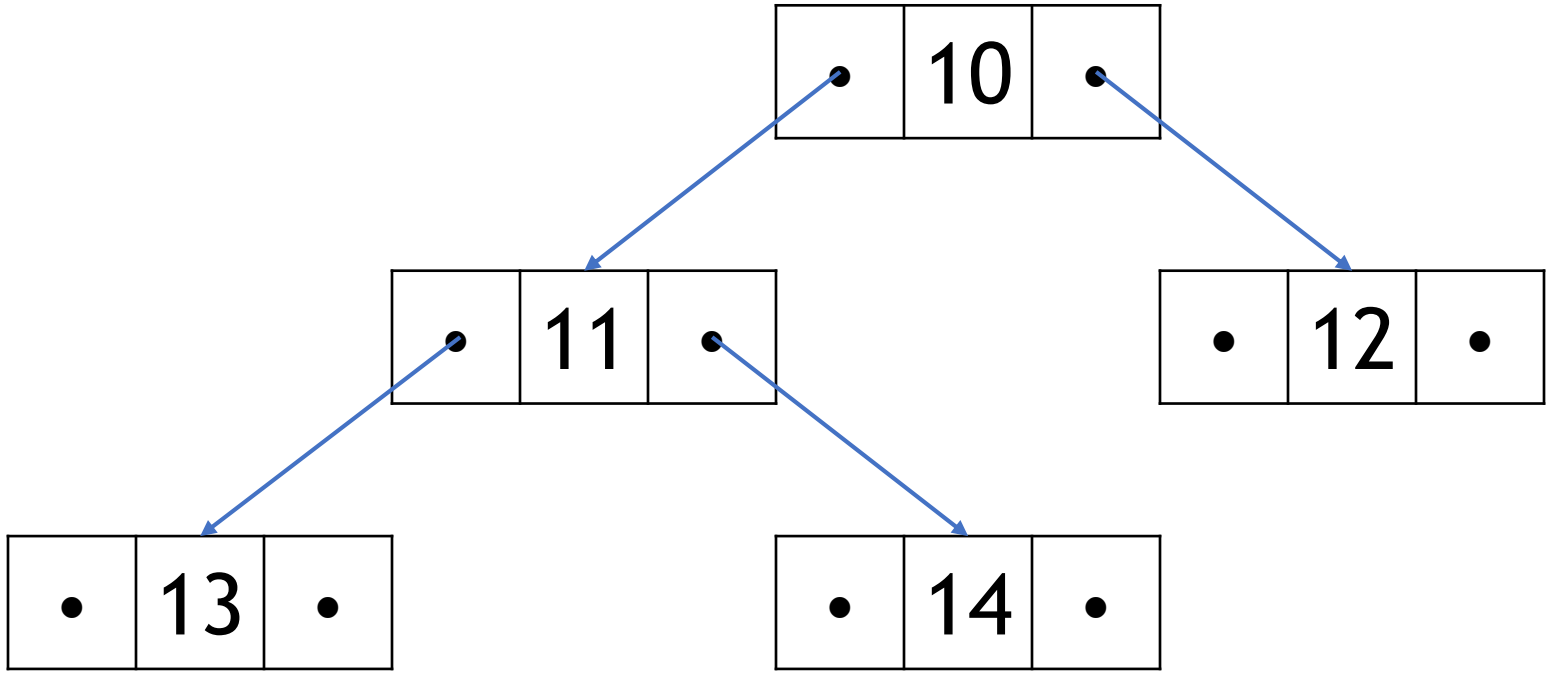
Binary Tree: Implementation (1)

Recall: Implementation of a Node in a Linked List

```
struct Node
{
    int data;
    Node *left;
    Node *right;
};
```



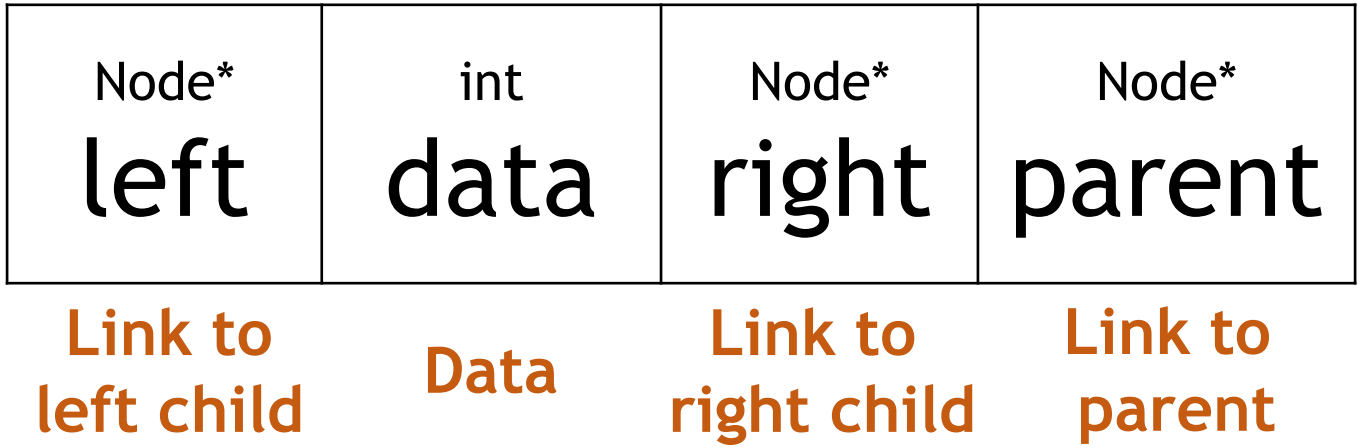
Binary Tree: Implementation (1)



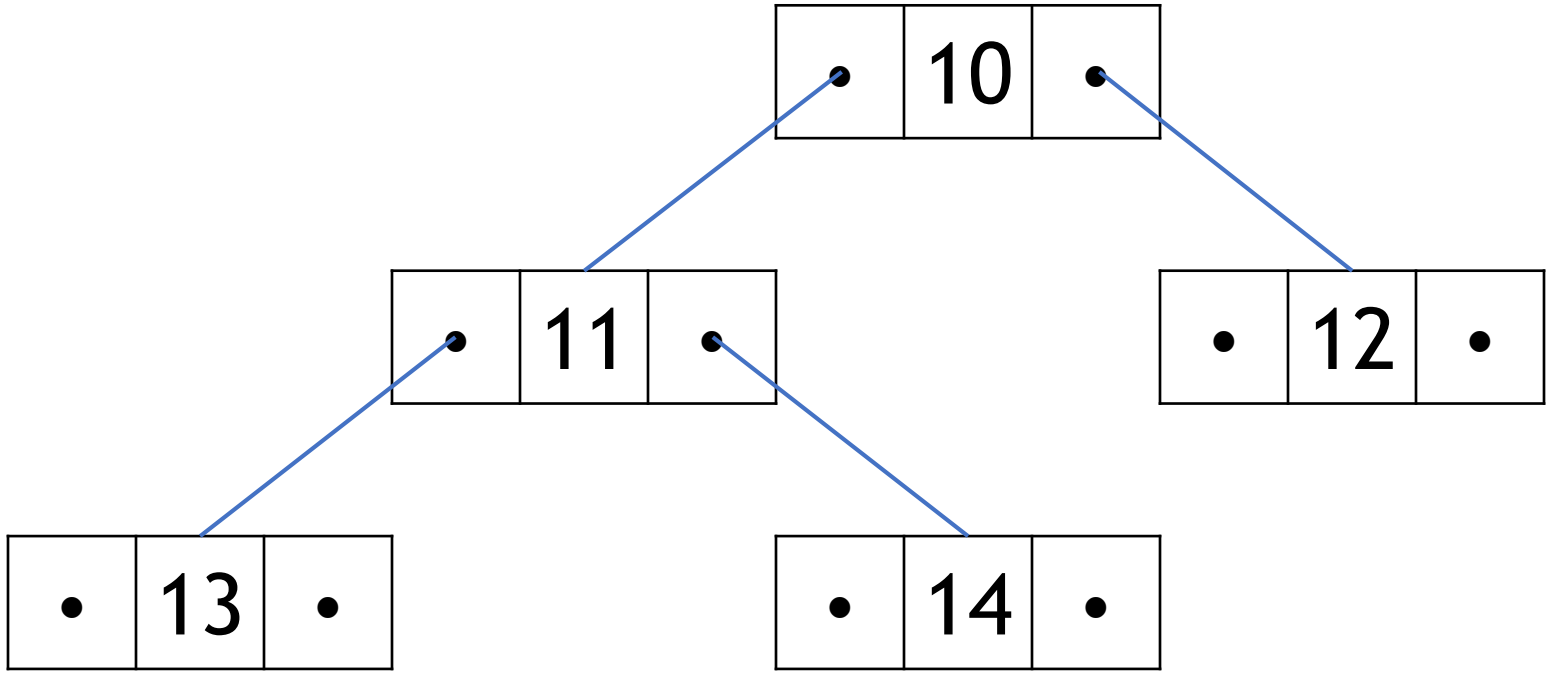
Binary Tree: Implementation (2)

Recall: Implementation of a Node in a Linked List

```
struct Node
{
    int data;
    Node *parent;
    Node *left;
    Node *right;
};
```



Binary Tree: Implementation (2)



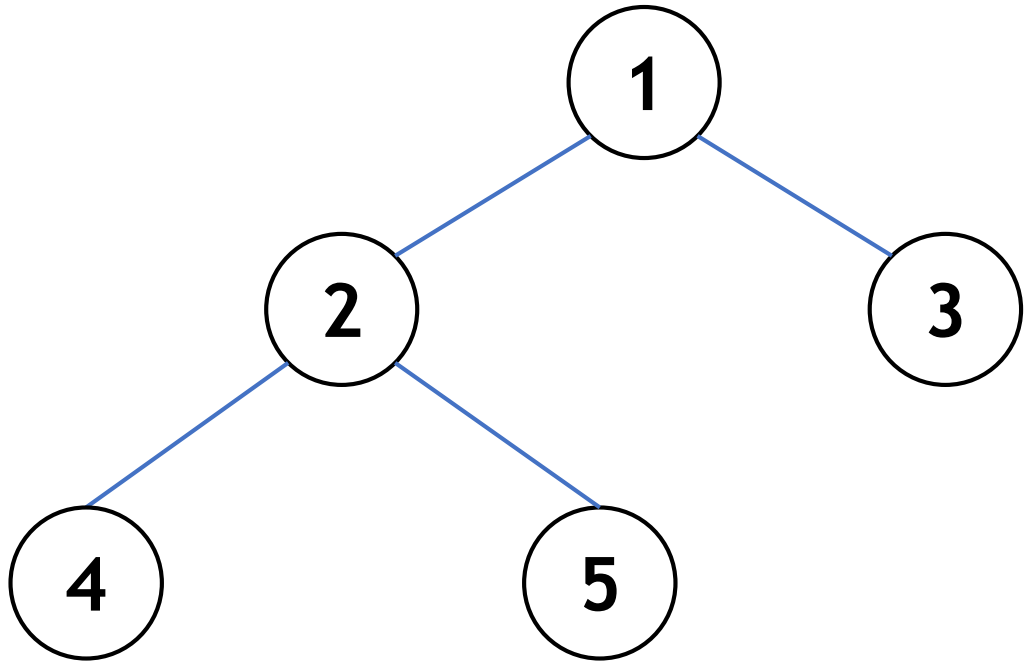
Traversal in a Binary Tree

Classification of traversal on basis of current node

Preorder	DLR	1, 2, 4, 5, 3
Inorder	LDR	4, 2, 5, 1, 3
Postorder	LRD	4, 5, 2, 3, 1

Level order traversal

1, 2, 3, 4, 5



Binary Tree: Notes

Iterative Solution vs Recursive Solution

- Problem with deep recursion: Limited space in stack
- Will still stick with the latter for most part

Problem Solving in Trees

- Think in terms of left and right subtrees
- Consider all possible configurations of subtrees to form base case
- Individual nodes are also trees

Binary Search Tree (BST)

Invariant: Some property that must be maintained at all times

BST: Property

BST = BT + *Special Property (Invariant)*

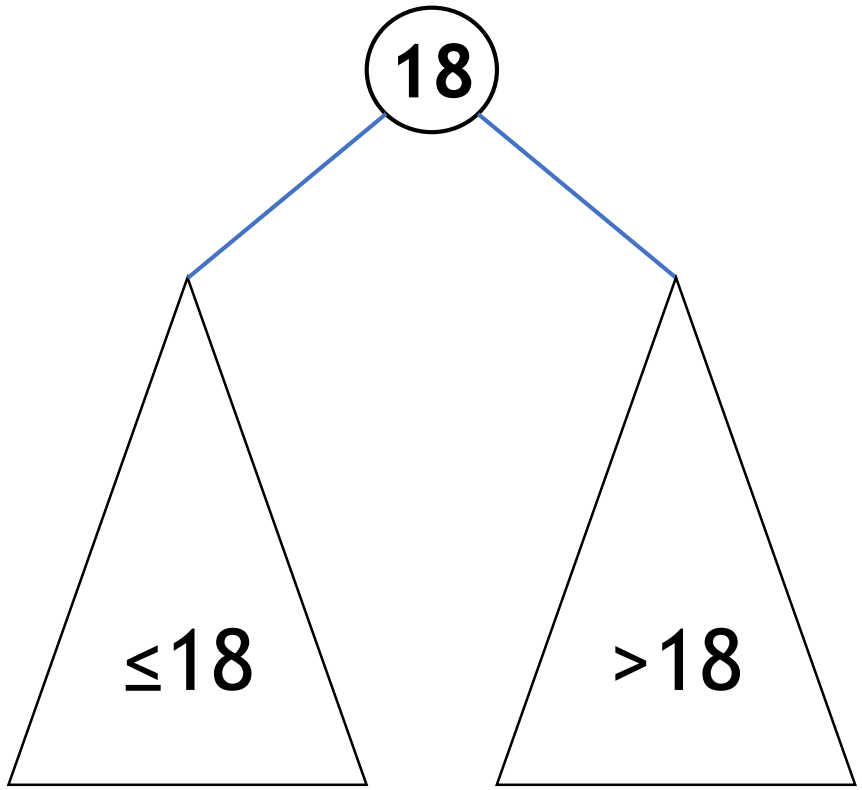
For any non-leaf node *N*:

$$\boldsymbol{X}.data \leq \boldsymbol{N}.data \; \forall \; \boldsymbol{X} \in \boldsymbol{N}.left\_subtree$$

$$\boldsymbol{X}.data > \boldsymbol{N}.data \; \forall \; \boldsymbol{X} \in \boldsymbol{N}.right\_subtree$$

Dealing with repeated values

= on left subtree (*convention*)



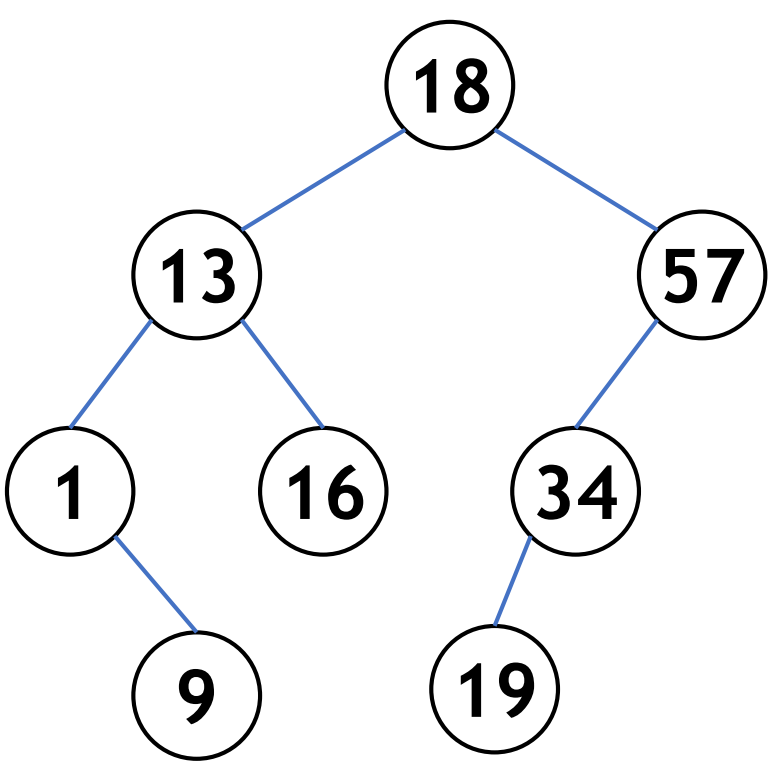
BST: Property

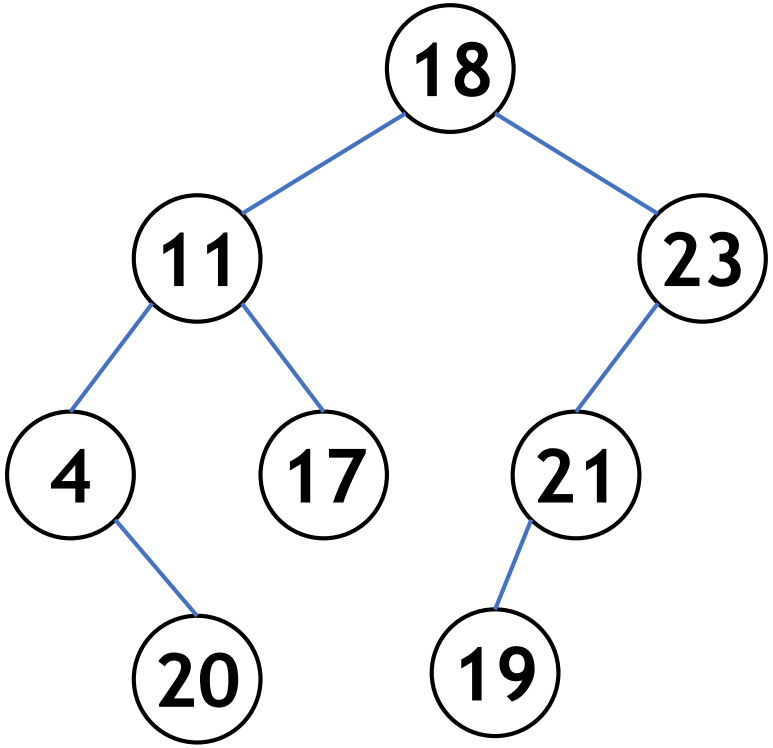
Why this invariant?

Allows us to add meaningful structure to the tree

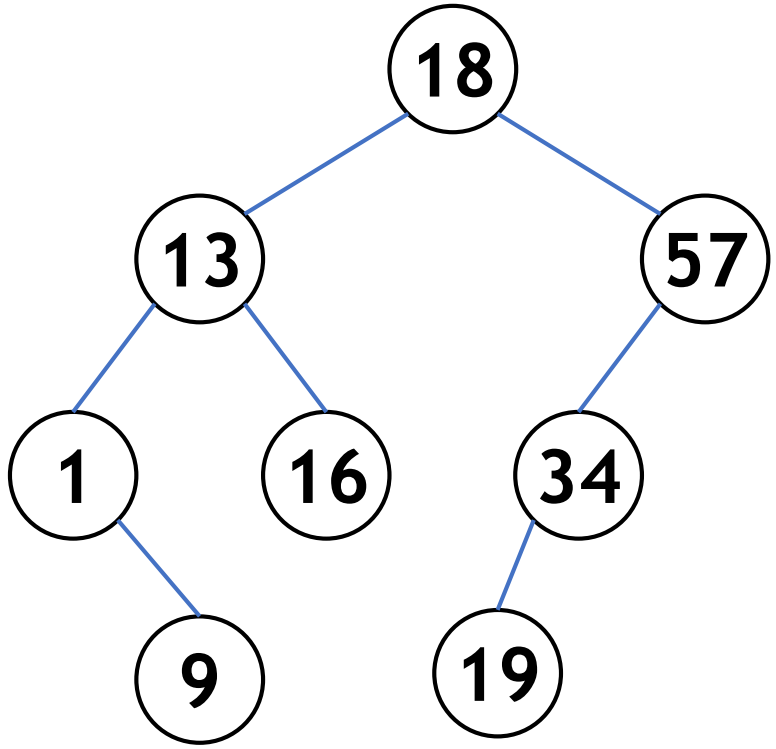
This makes several operations faster than a simple binary tree
Search, *Insert*, *Delete*

BST: Property





BST: Traversal



BT Traversals are valid

Inorder Traversal of a BST:
Results in a sorted order

Preorder	DLR	18, 13, 1, 9, 16, 57, 34, 19
Inorder	LDR	1, 9, 13, 16, 18, 19, 34, 57
Postorder	LRD	9, 1, 16, 13, 19, 34, 57, 18

Remember this! We will use this concept later.

BST: Find minimum node

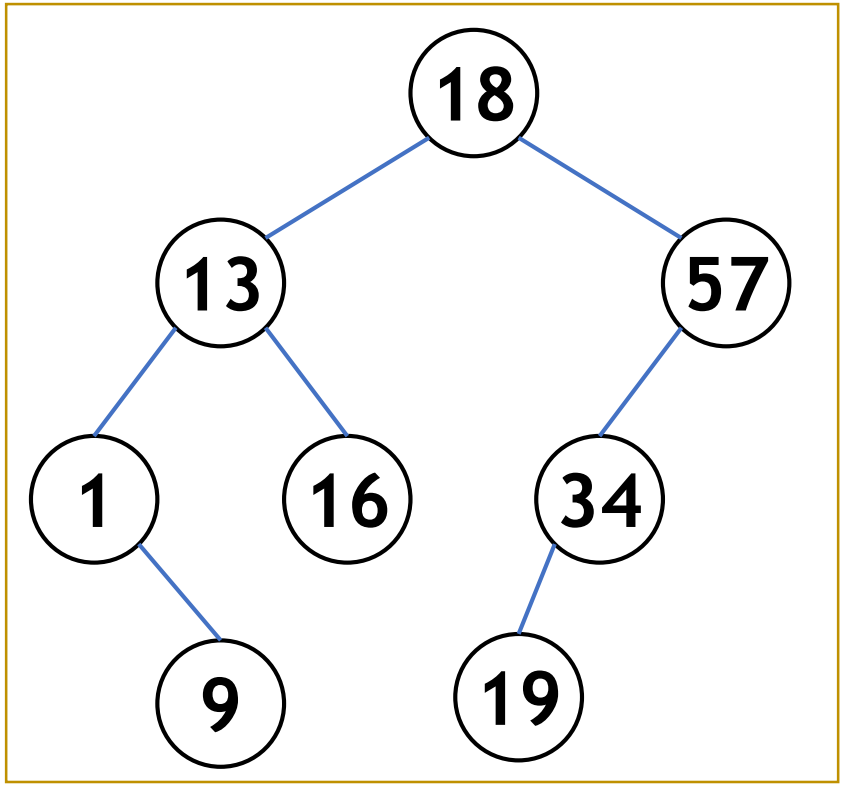
How to find the minimum node in a BST?

Exploit the structure of a BST:

- Left subtree contains all the nodes with values <= root.value
- Right subtree contains all the nodes with values > root.value

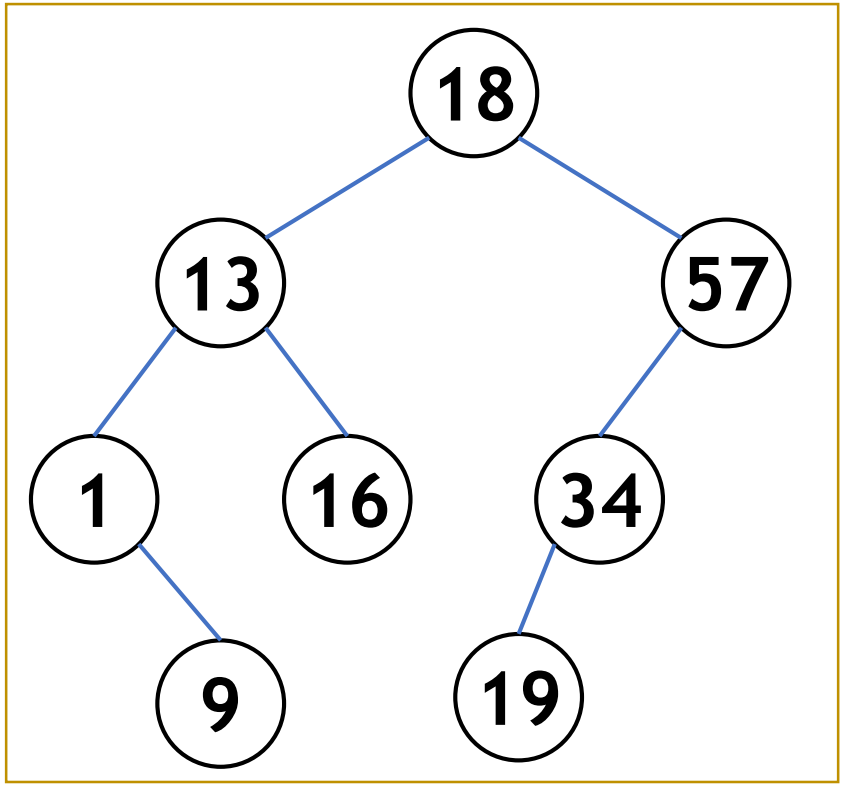
The leftmost node!

BST: Find minimum node



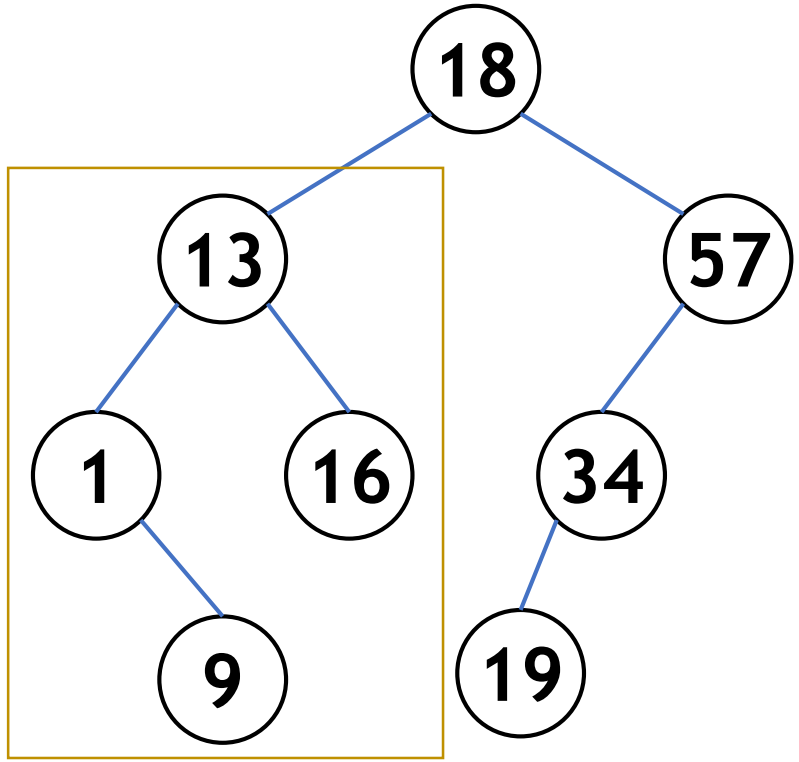

```
Node* minNode(Node* root) {
   Node* temp = root;
   while (temp && temp->left != NULL)
        temp = temp->left;
   return temp;
}
```

BST: Find minimum node



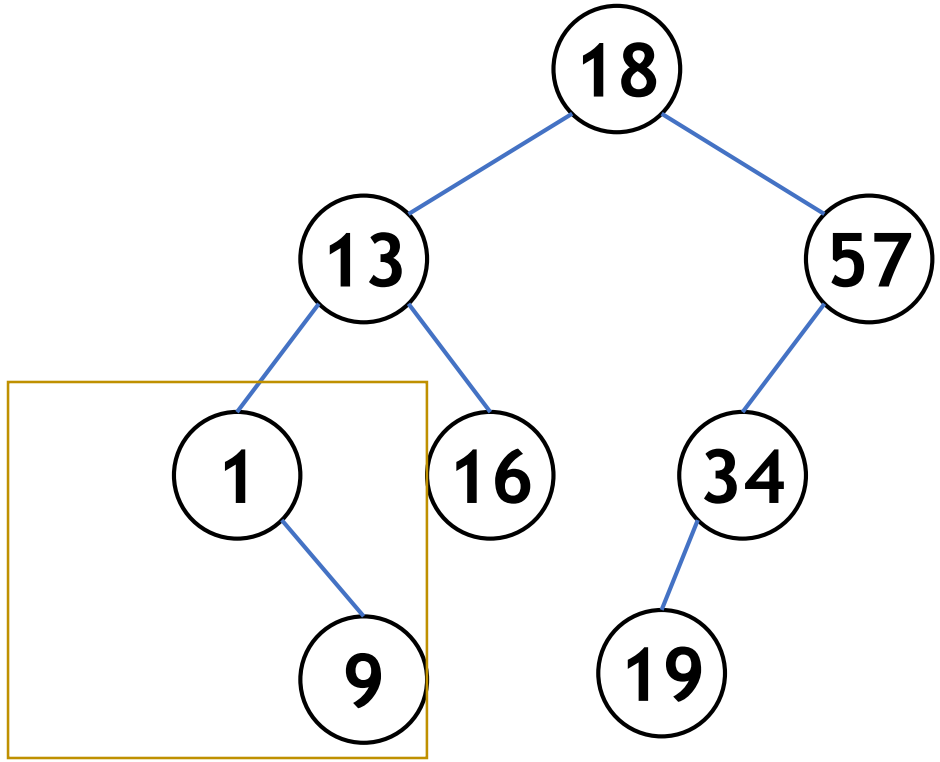

```
Node* minNode(Node* root) {
   Node* temp = root;
   while (temp && temp->left != NULL)
        temp = temp->left;
   return temp;
}
```

BST: Find minimum node



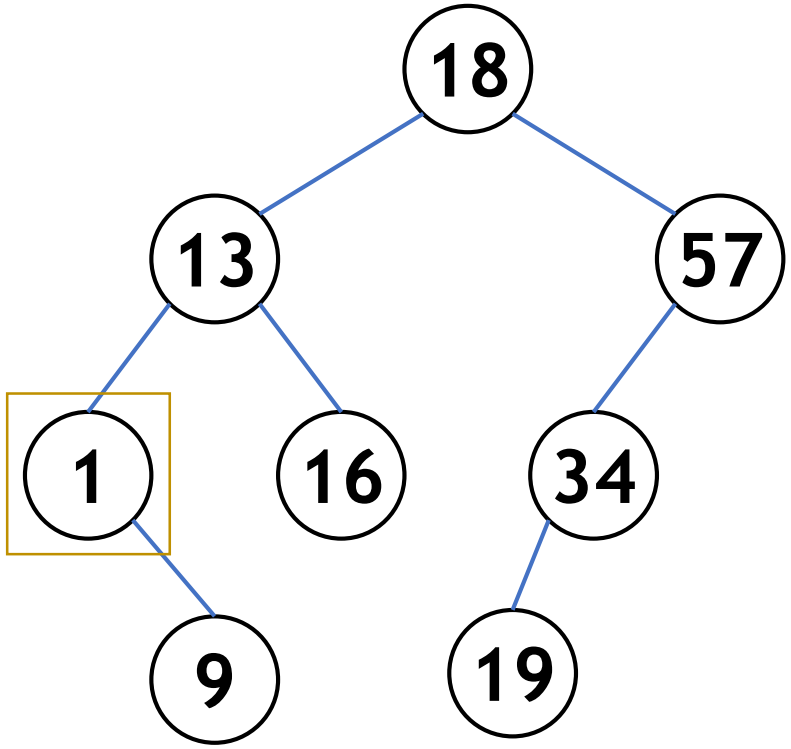

```
Node* minNode(Node* root) {
   Node* temp = root;
   while (temp && temp->left != NULL)
        temp = temp->left;
   return temp;
}
```

BST: Find minimum node




```
Node* minNode(Node* root) {
   Node* temp = root;
   while (temp && temp->left != NULL)
        temp = temp->left;
   return temp;
}
```

BST: Find minimum node




```
Node* minNode(Node* root) {
   Node* temp = root;
   while (temp && temp->left != NULL)
        temp = temp->left;
   return temp;
}
```

BST: Find minimum node

```
// iterative
Node* minNode(Node* root) {
  Node* temp = root;
  while(temp&&temp->left != NULL)
      temp = temp->left;
  return temp;
}
```

```
// recursive
Node* minNode(Node* root) {
  if (root && !(root->left))
      return root;
  return minNode(root->left);
}
```

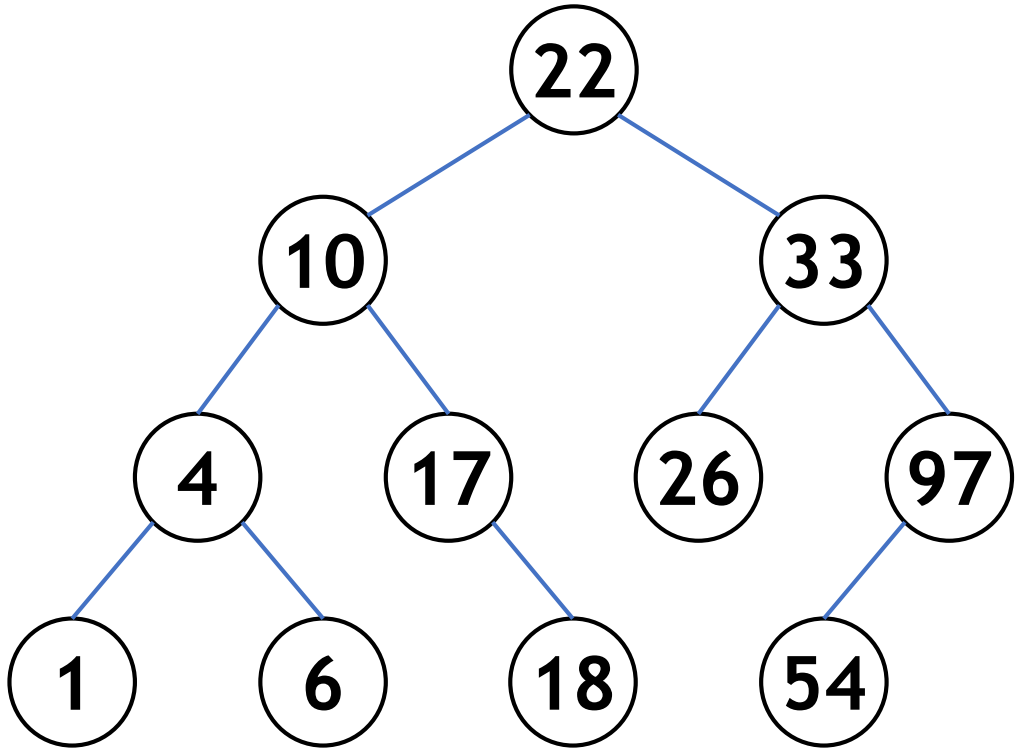
BST: Search

BST: Search

```
Node* search(Node* root, int key)
if (!root || key == root->data)
    return root;
else if (key <= root->data)
    return search(root->left, key);
else
    return search(root->right, key);
```

BST: Search (1)

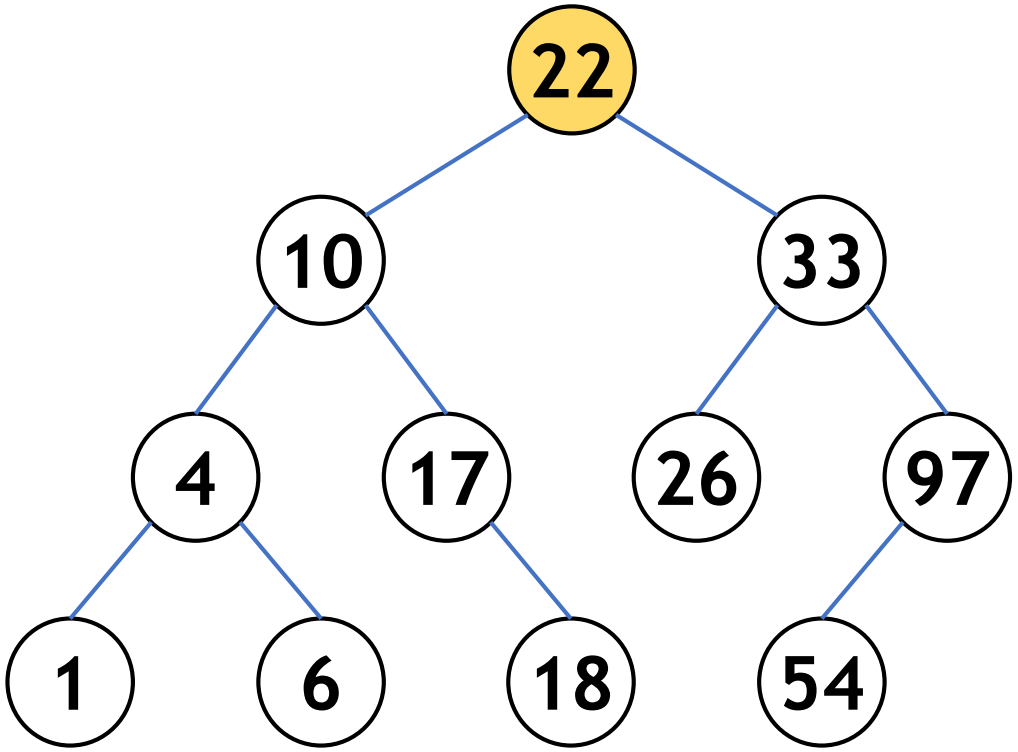
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

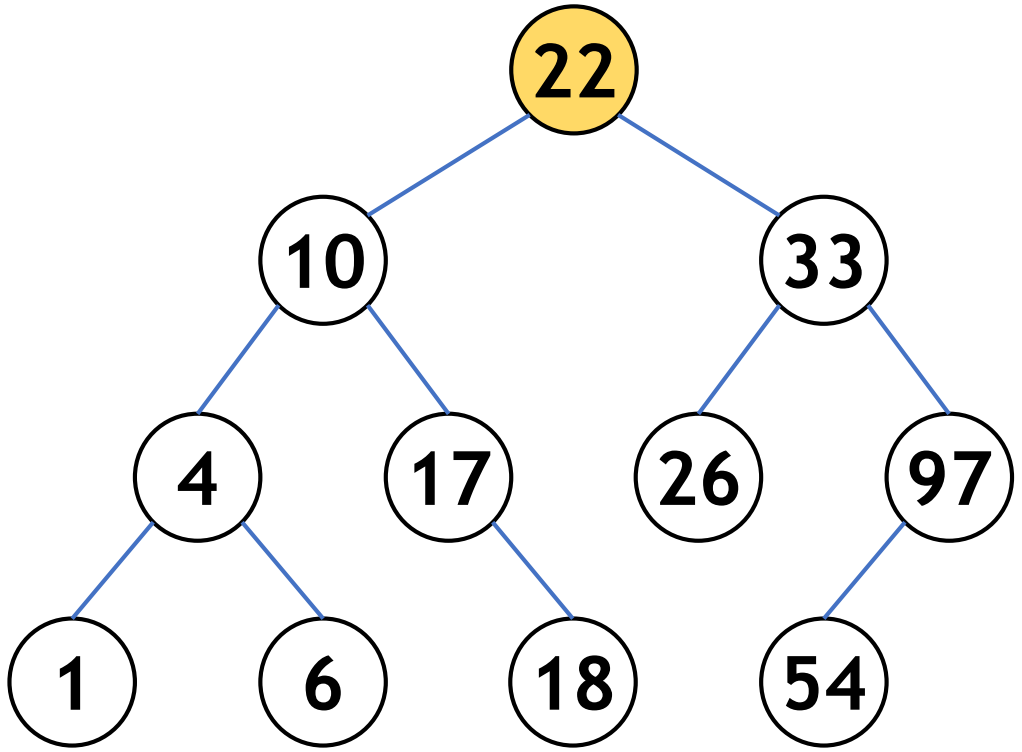
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

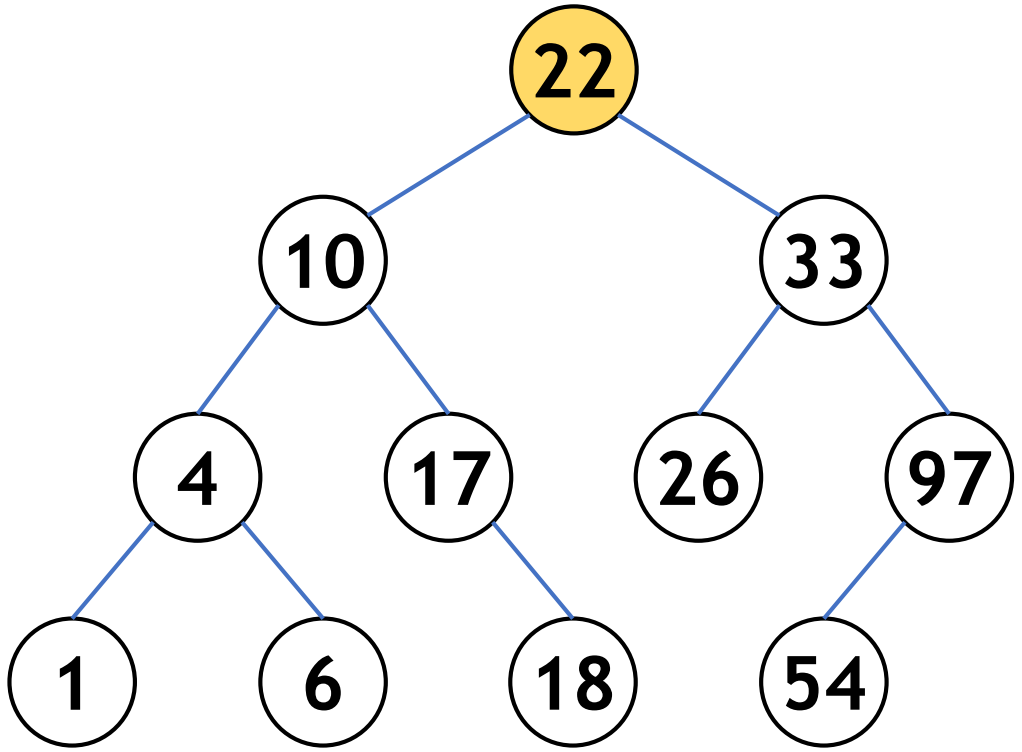
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

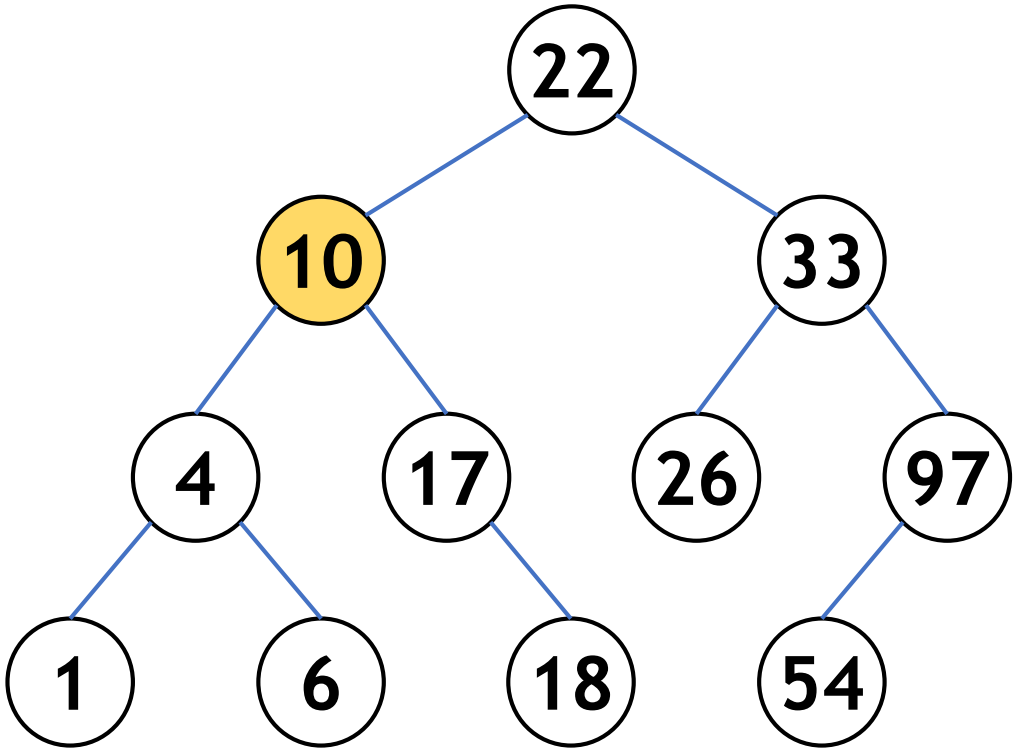
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

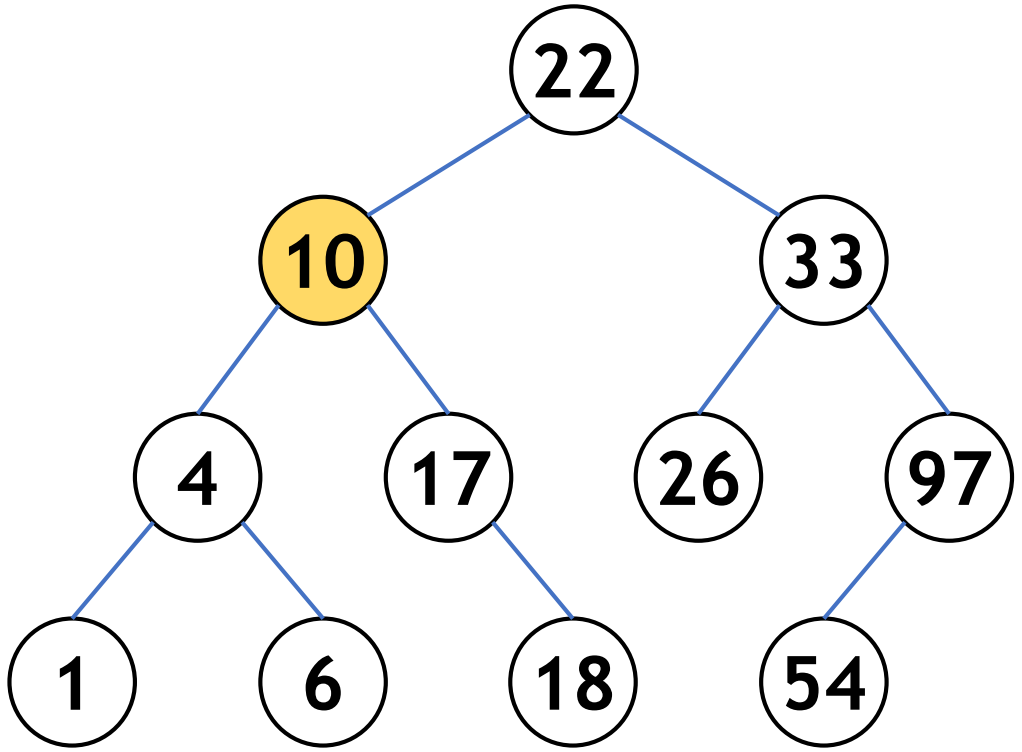
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

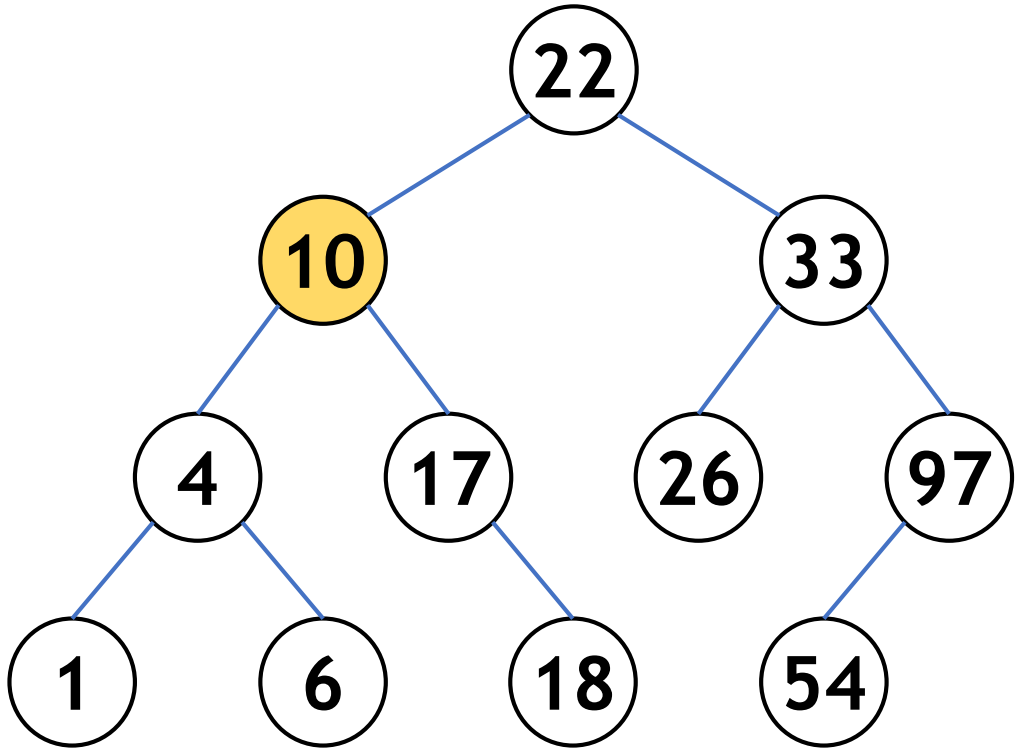
`search(root, 6)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

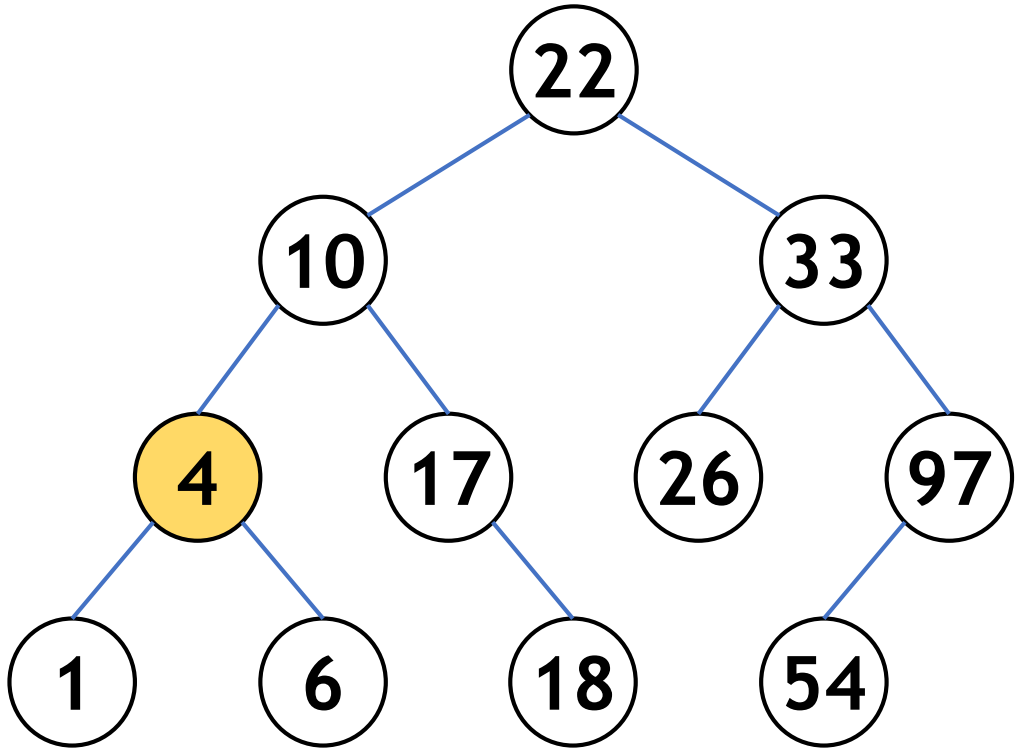
`search(root, 6)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

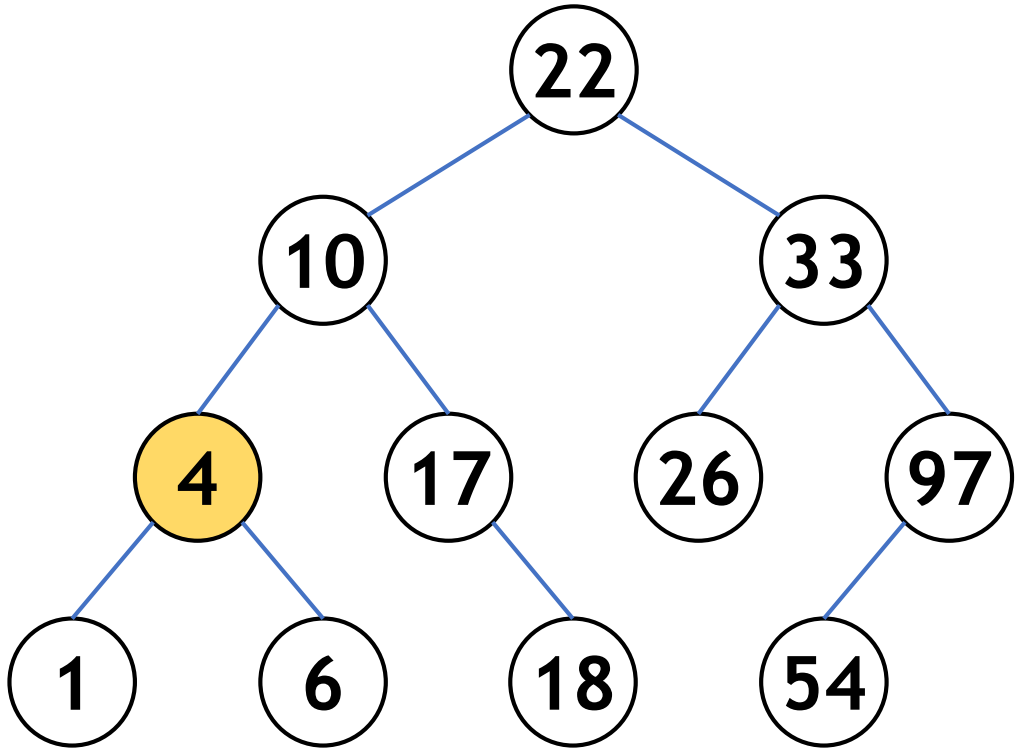
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

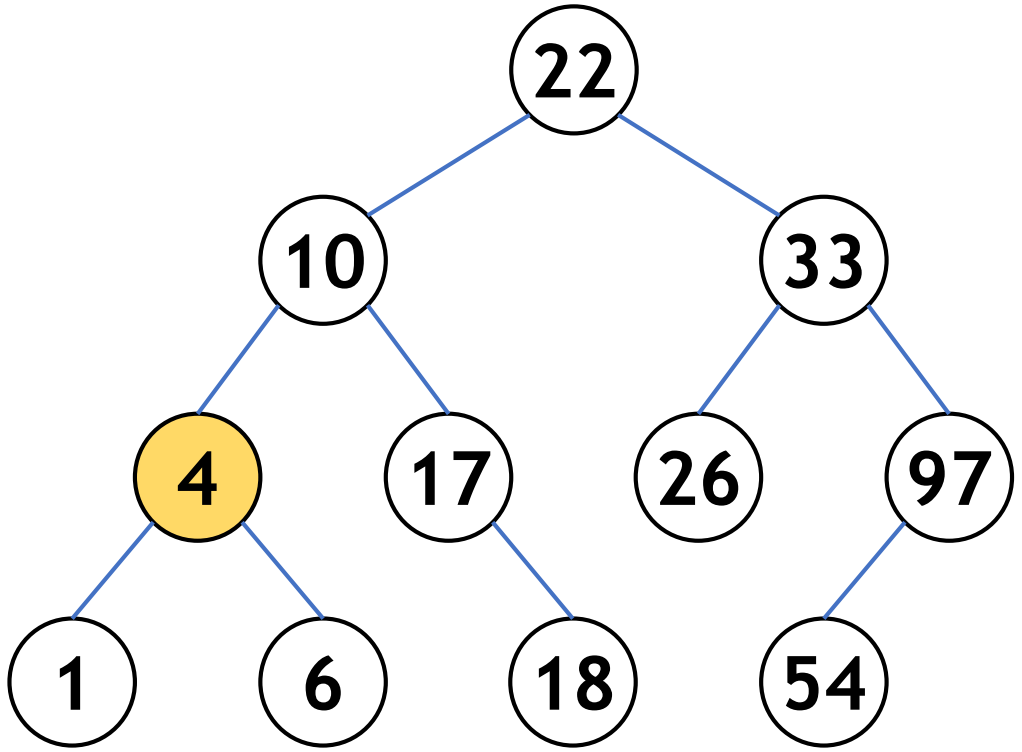
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

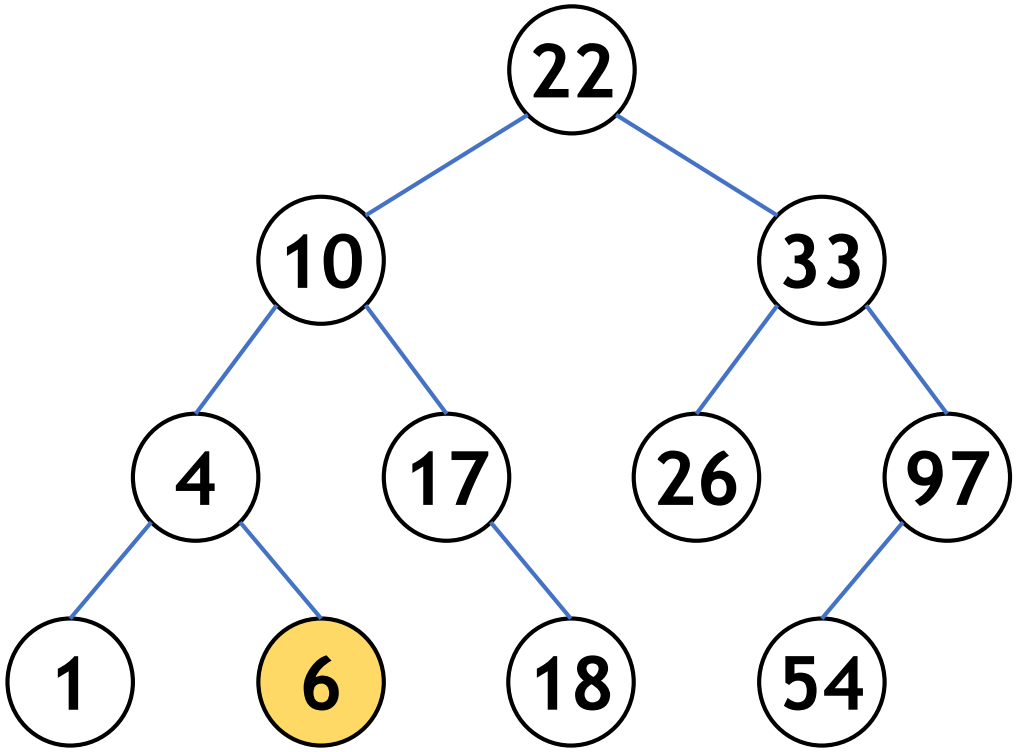
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

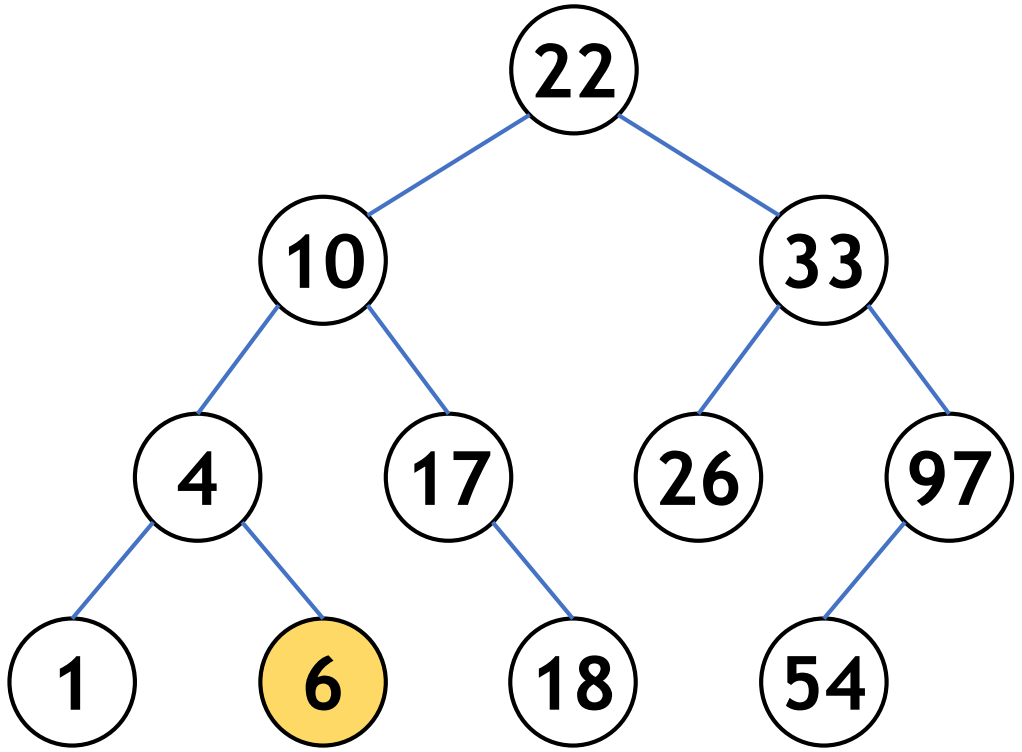
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

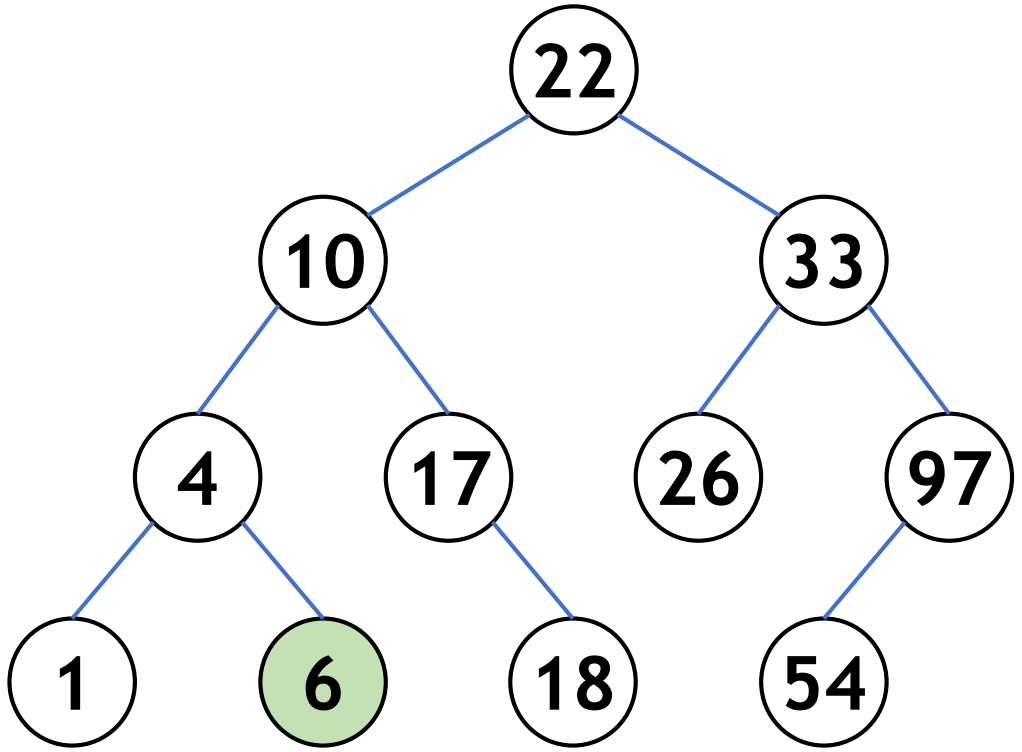
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

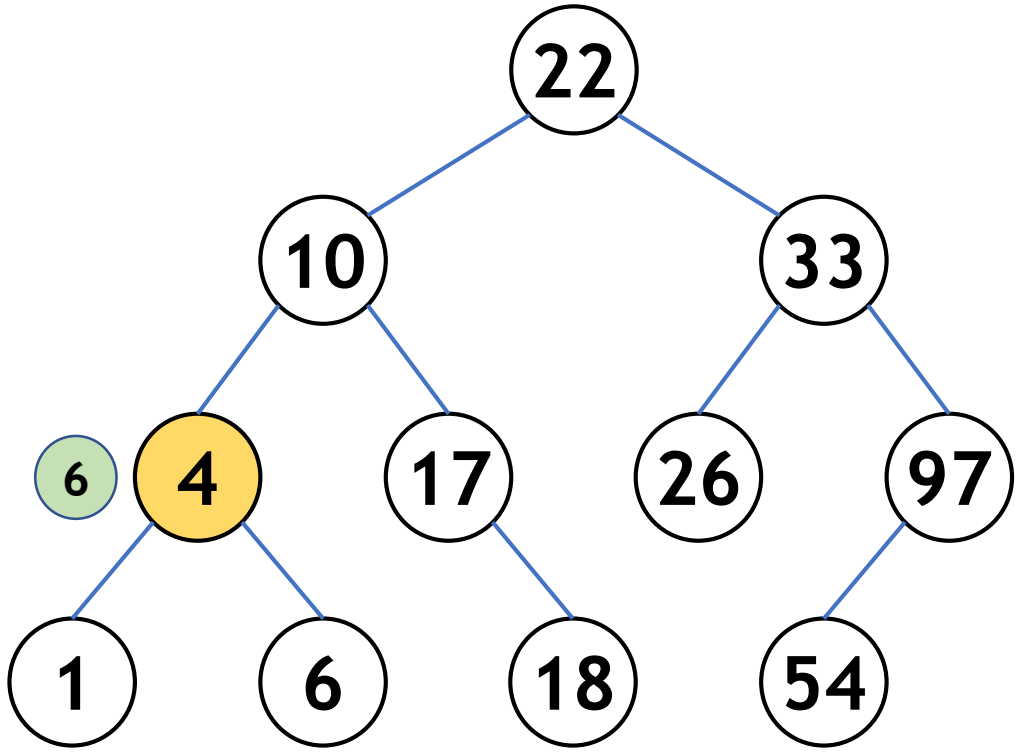
`search(root, 6)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

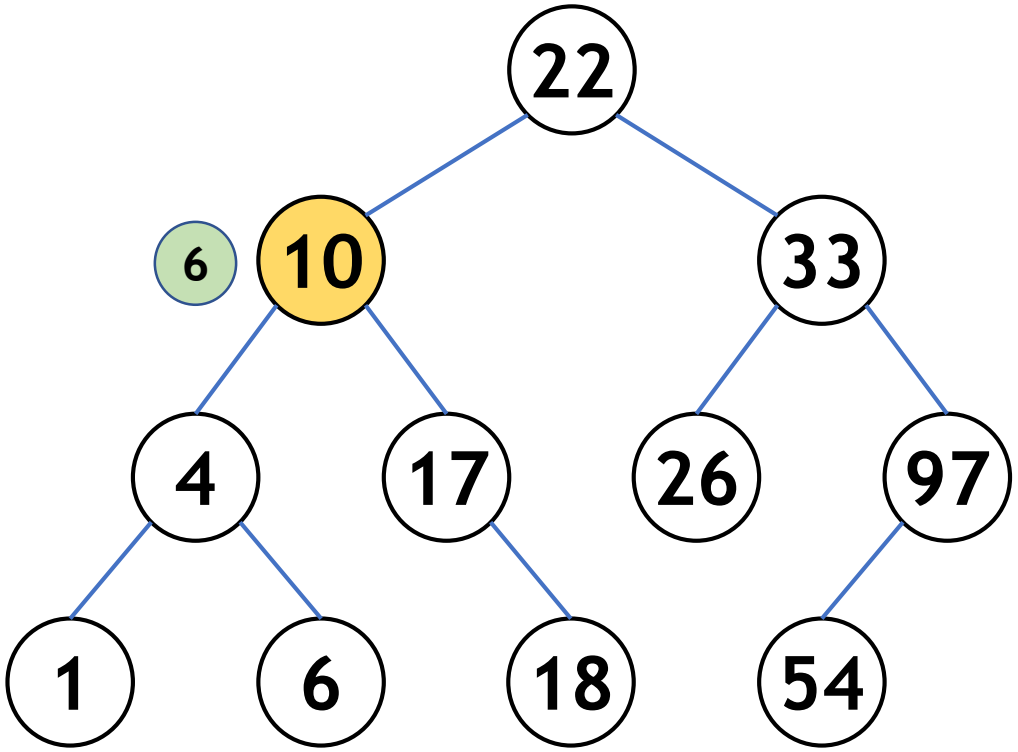
search(root, 6)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

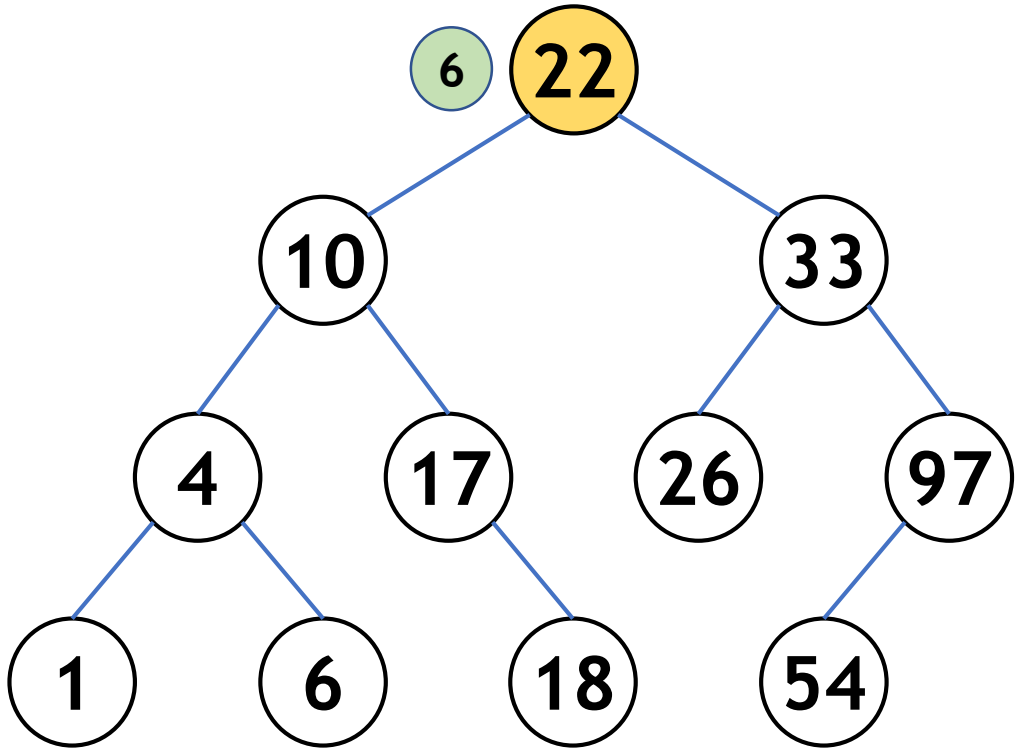
`search(root, 6)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (1)

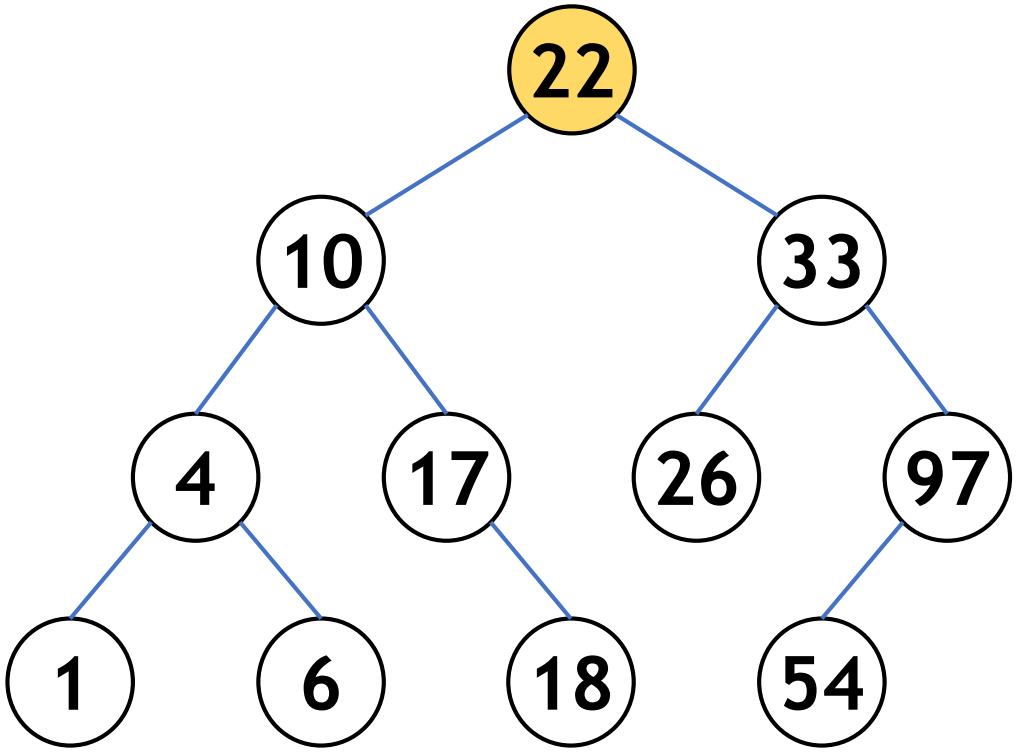
`search(root, 6)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

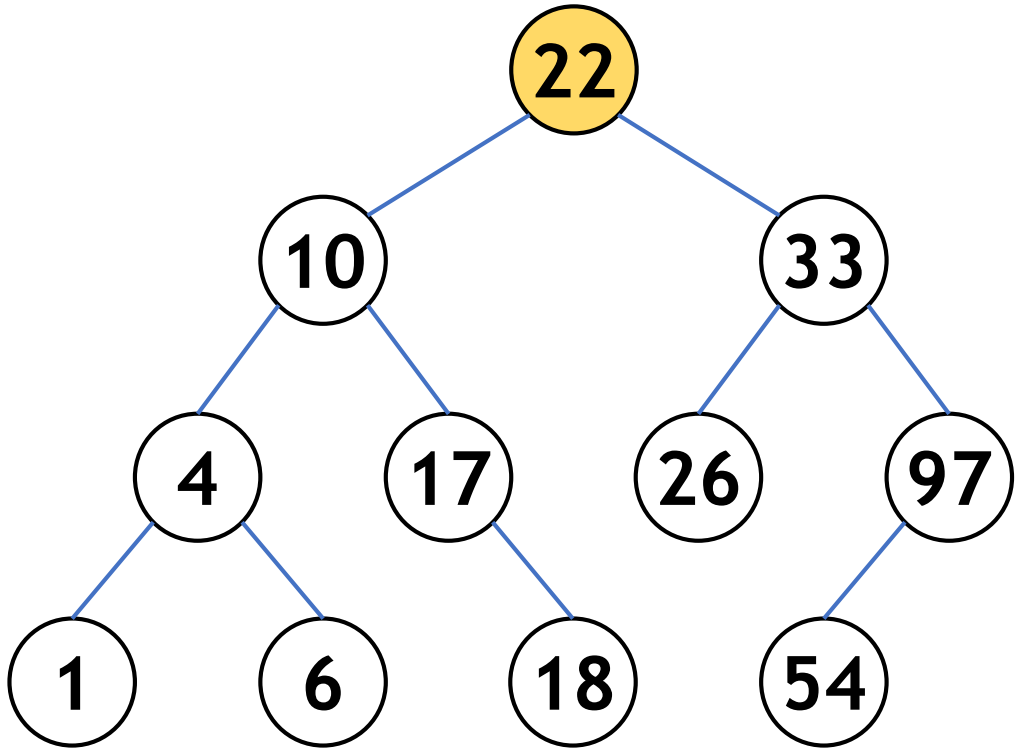
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

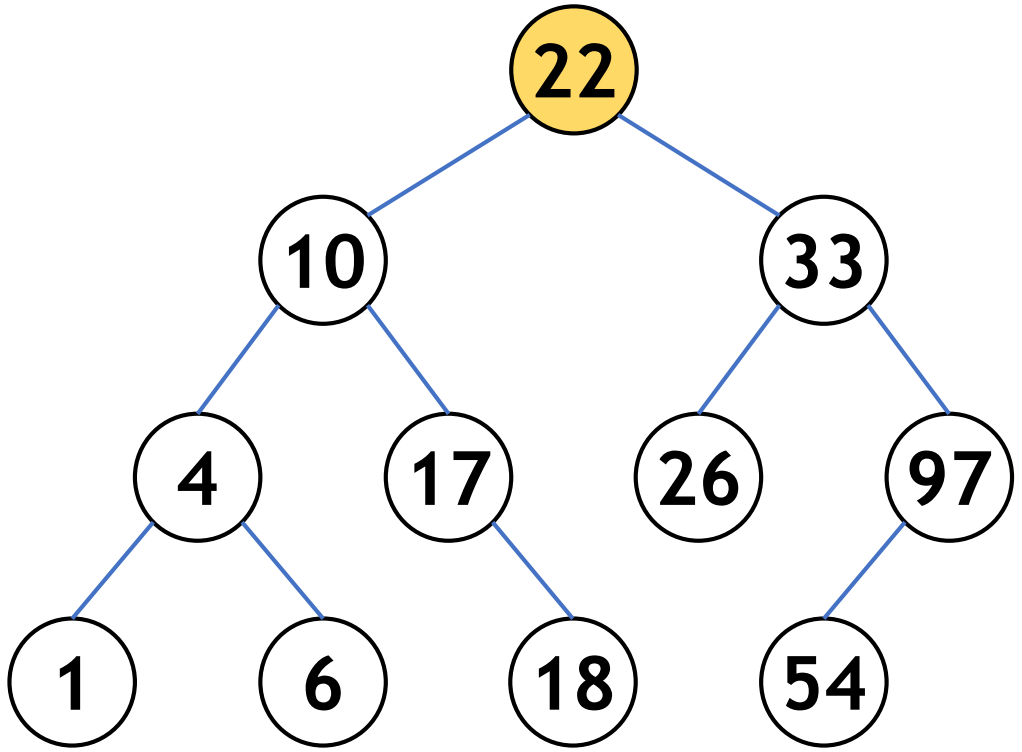
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

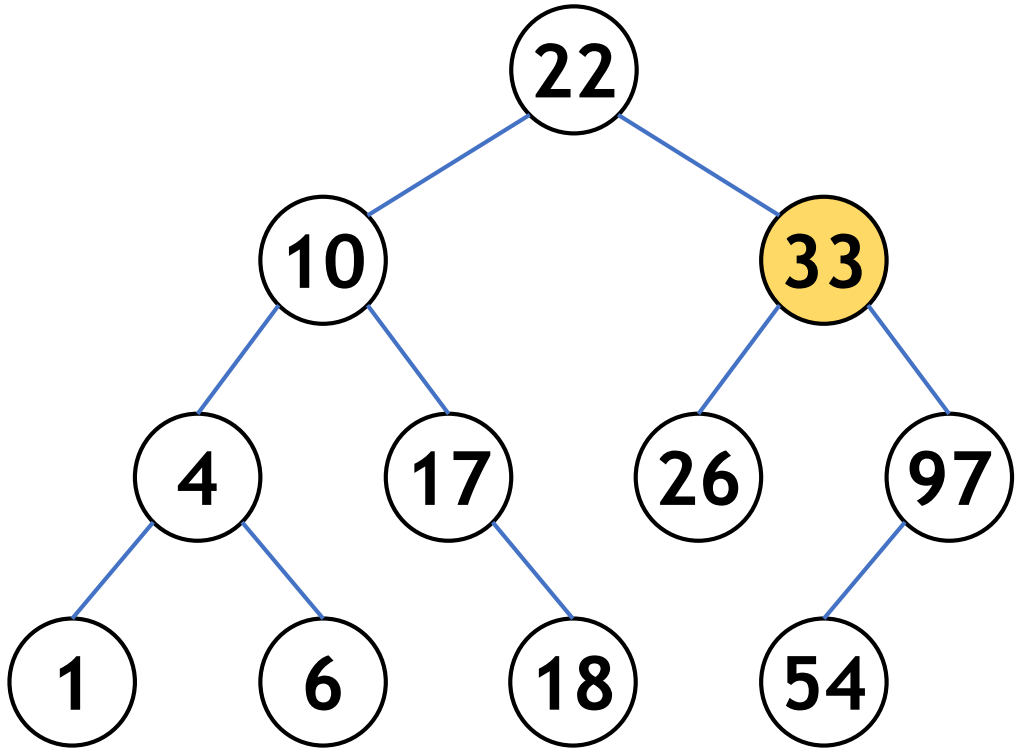
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

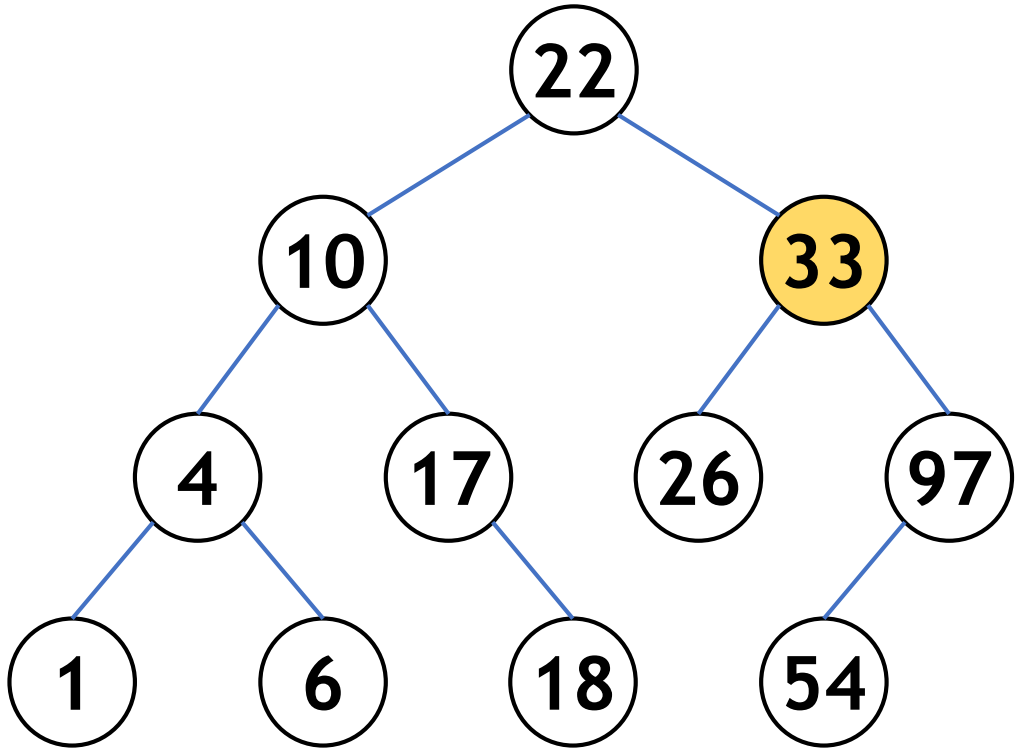
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

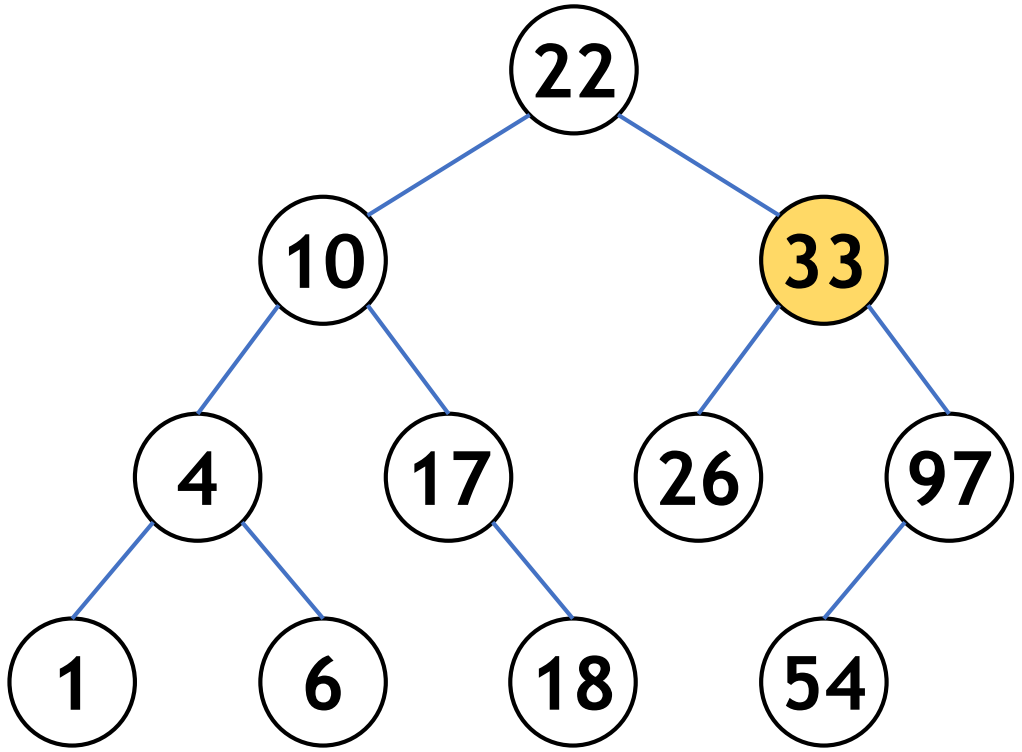
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

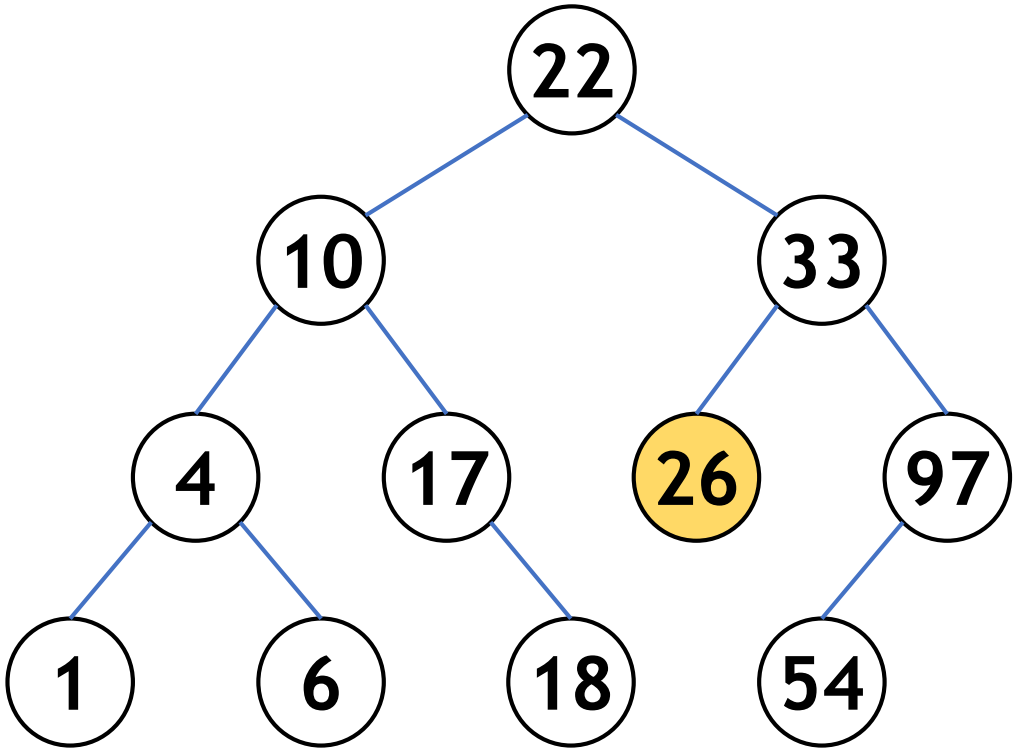
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

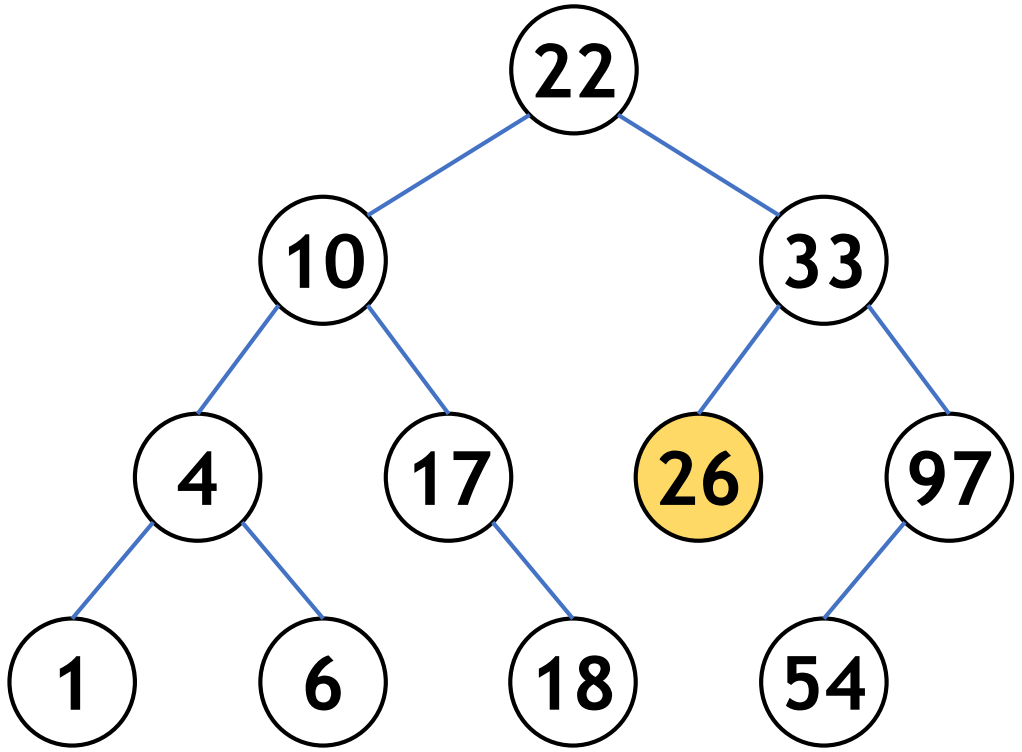
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

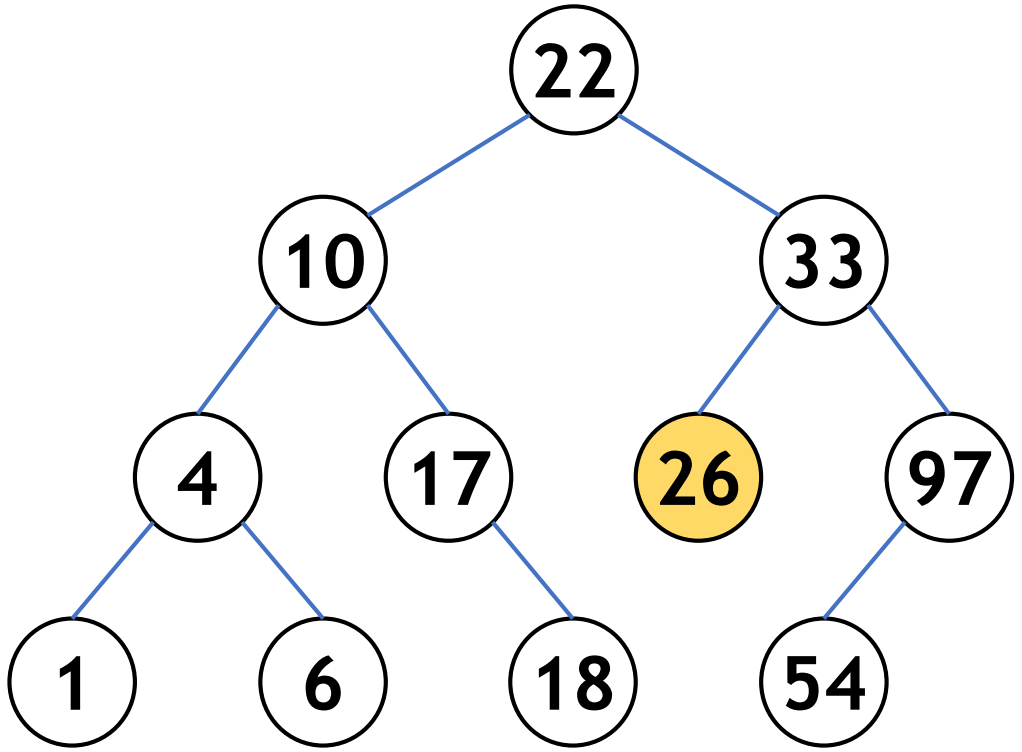
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

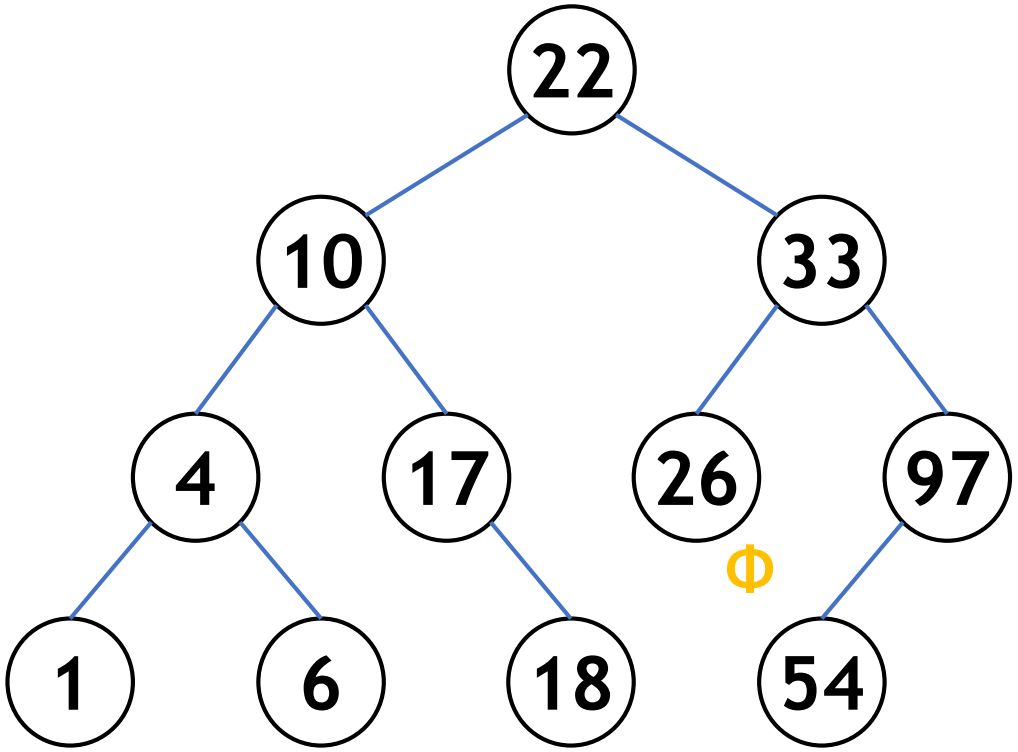
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

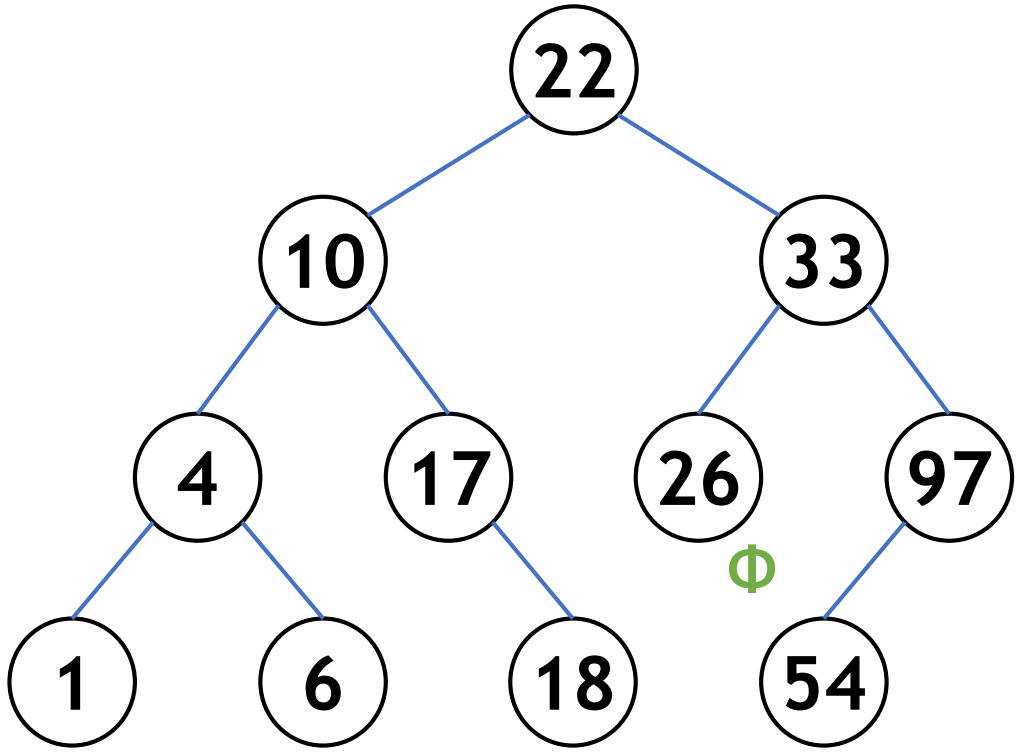
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

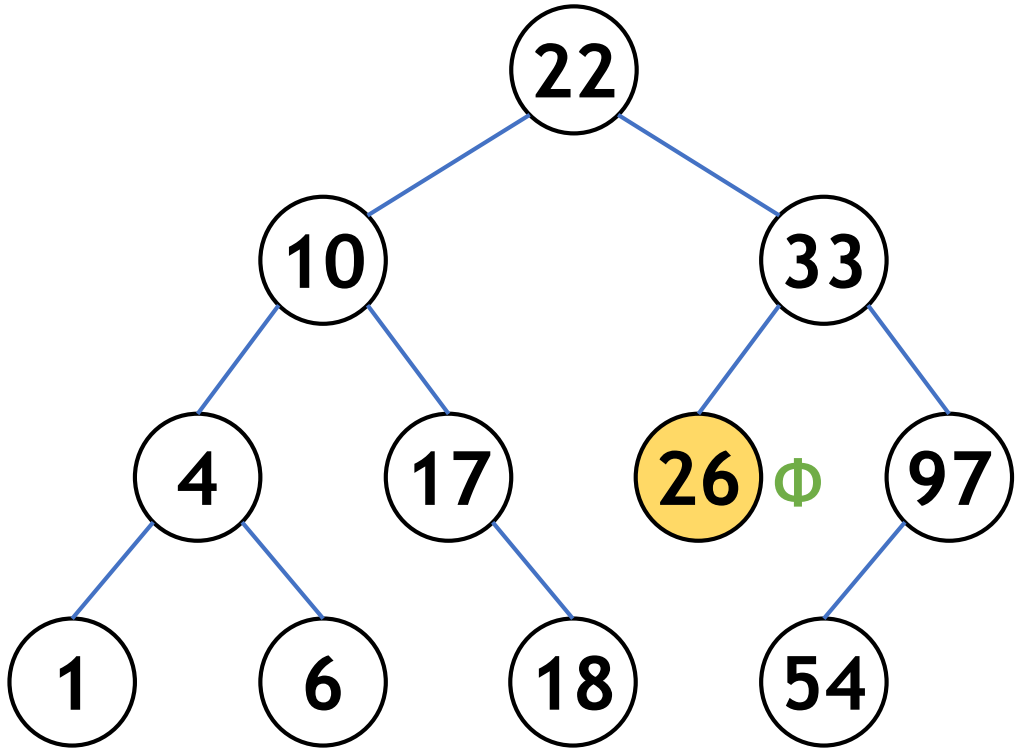
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

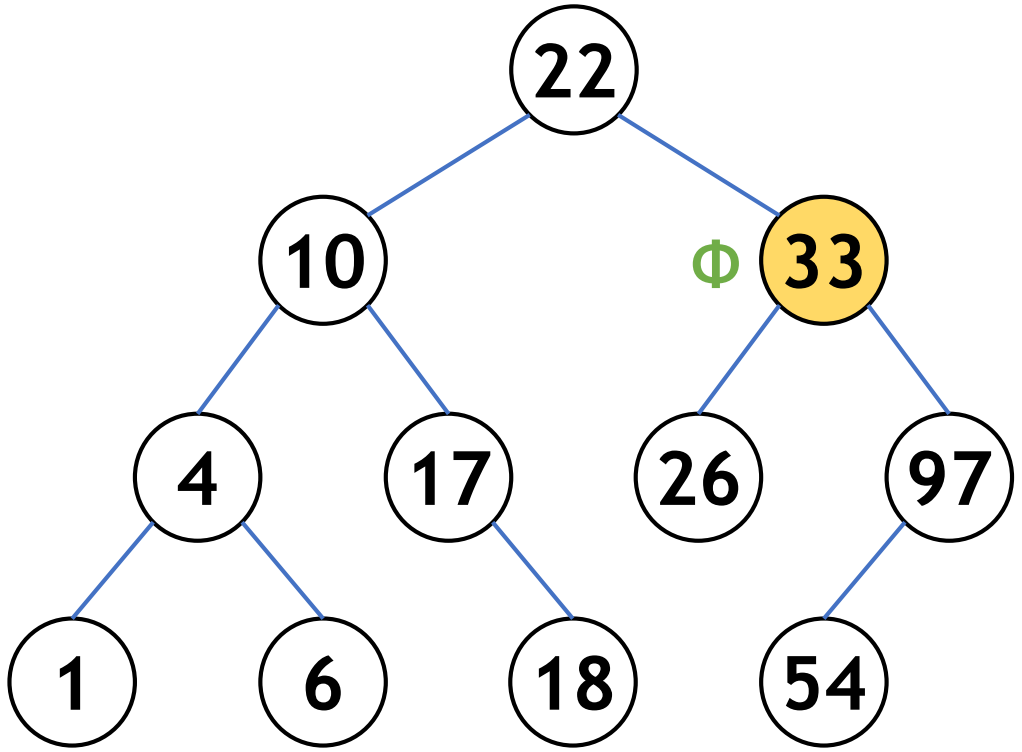
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

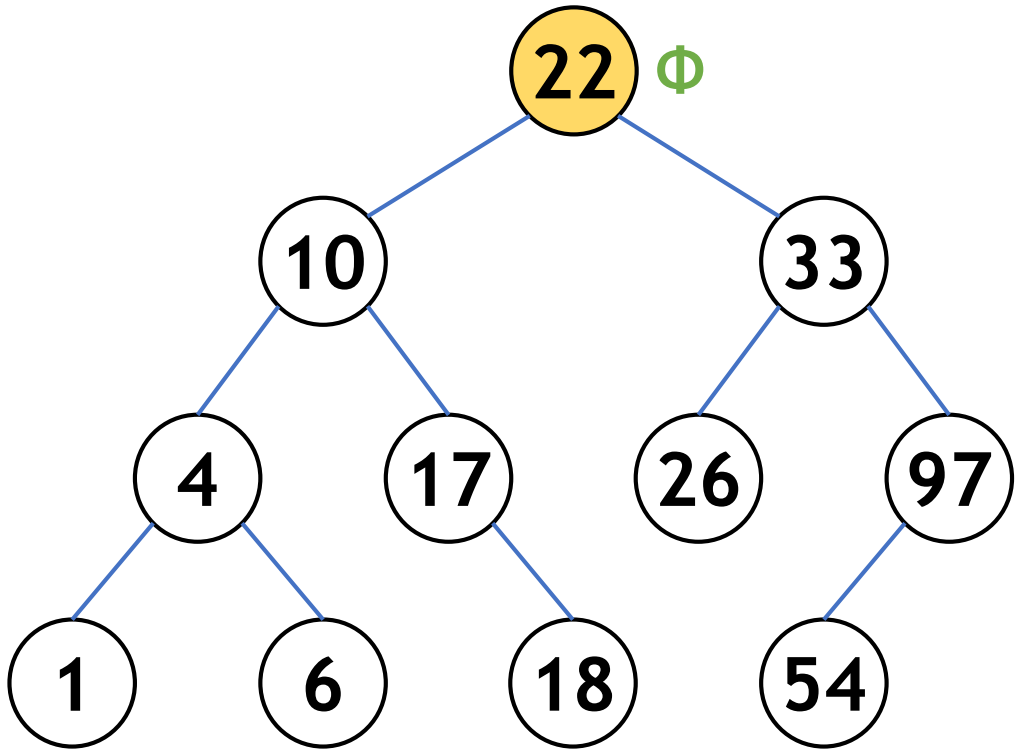
search(root, 28)




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Search (2)

`search(root, 28)`




```
search(root, key)
if (!root || key == root->data)
    return root
else if (key <= root->data)
    return search(root->left, key)
else
    return search(root->right, key)
```

BST: Analysis of Search Operation

***Recall*: Binary Search in a sorted array**

4	10	17	22	26	33	54	97

Complexity of Search operation:

Binary Tree: **O(n)**

Binary Search Tree: **$O(\log_2 n)$**

BST: Insertion

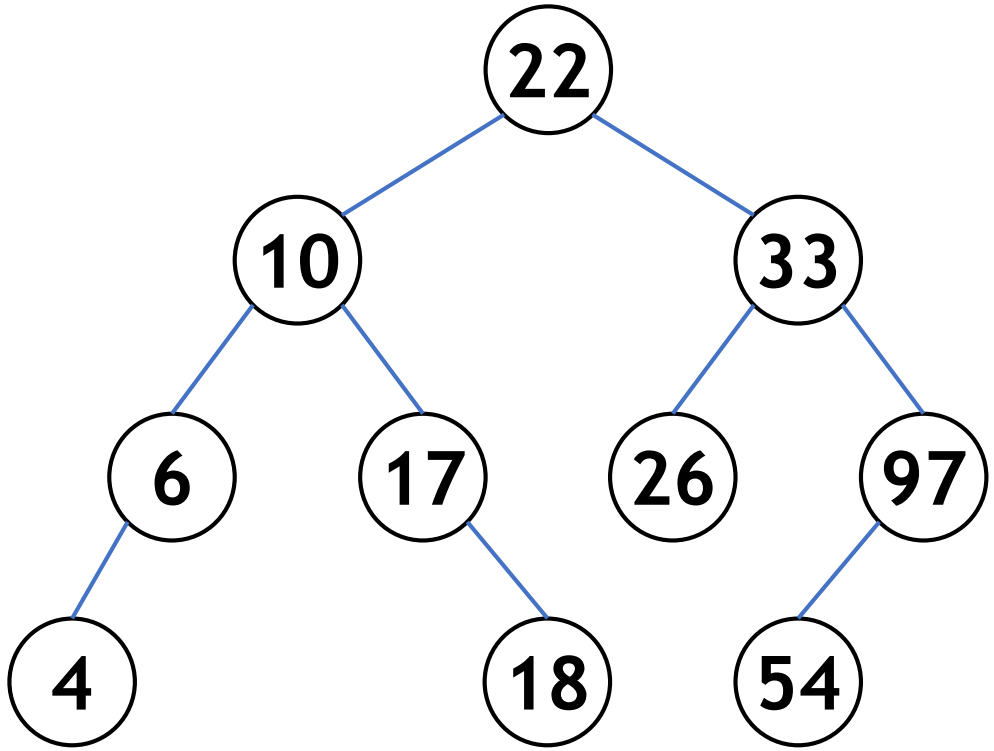
BST: Insertion

```
Node* createNode(int k) {
    Node* n = new Node;
    n->left = nullptr;
    n->right = nullptr;
    n->data = k;
    return n;
}
```

```
Node* insert(Node* root, int key)
if (!root)
    return createNode(key)
else if (key <= root->data)
    root->left = insert(root->left, key)
else
    root->right = insert(root->right, key)

return root
```

BST: Insertion

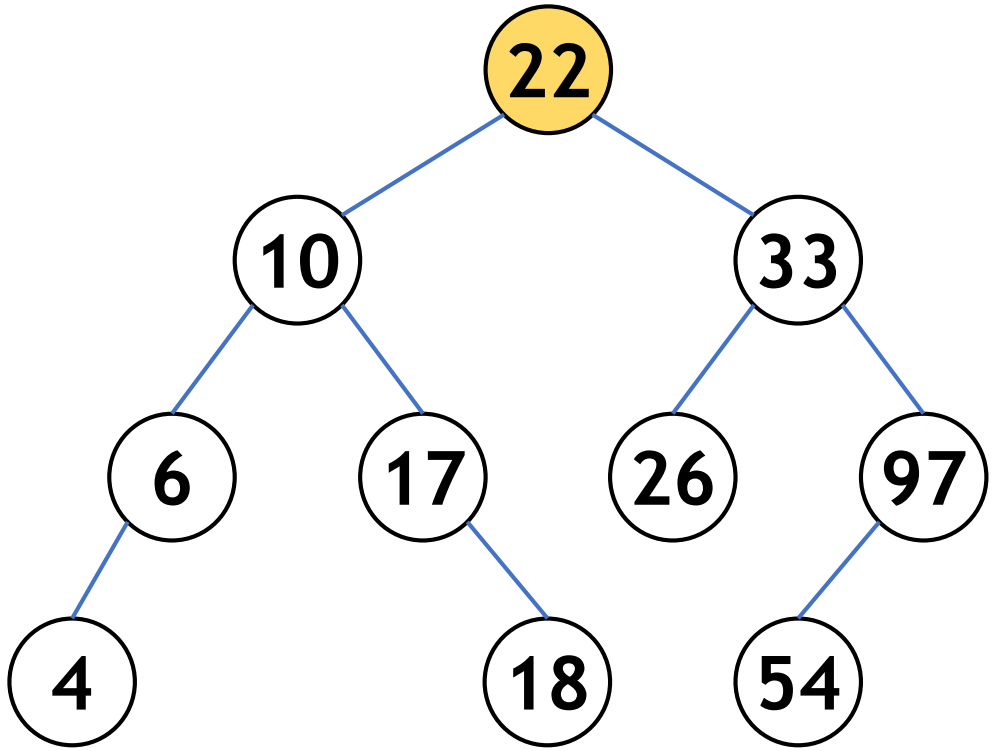
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

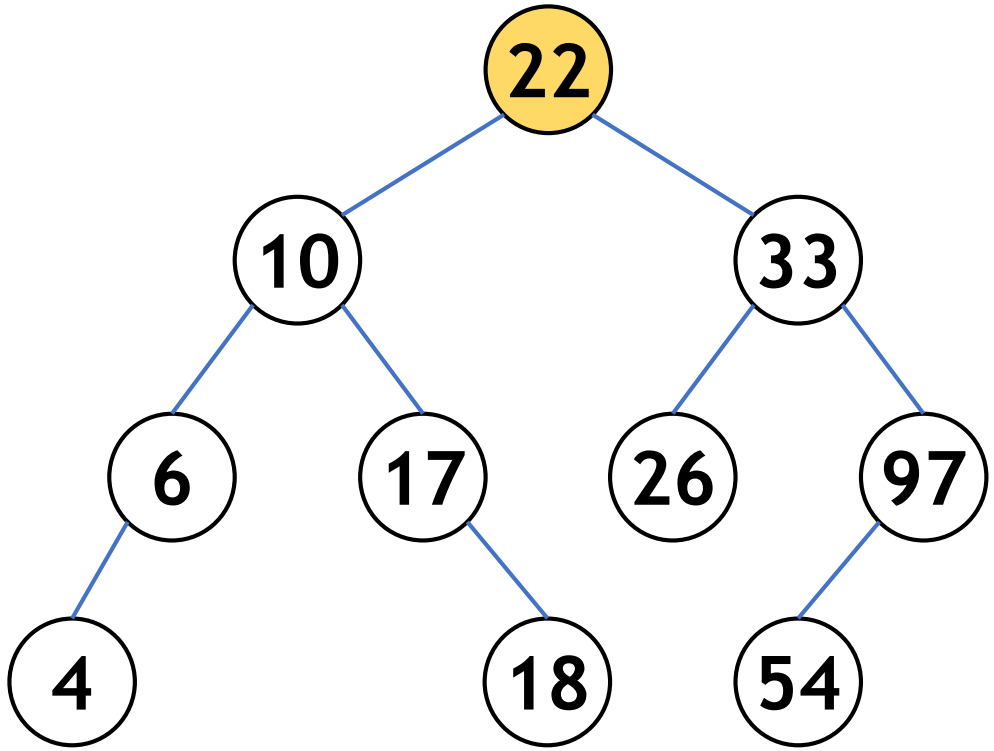
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

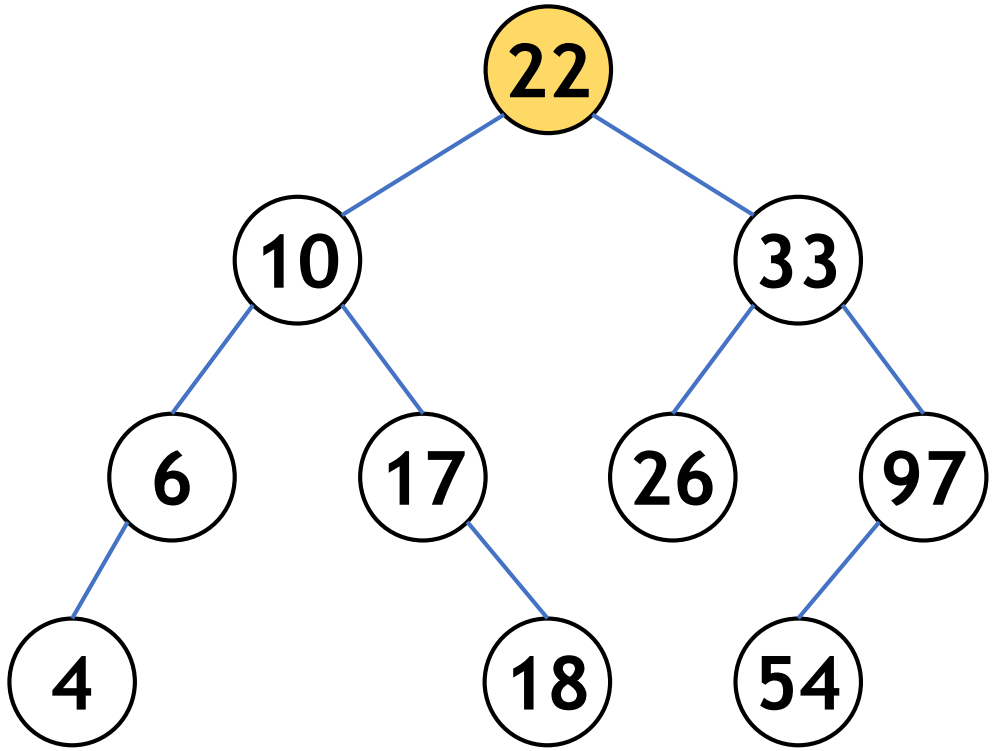
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

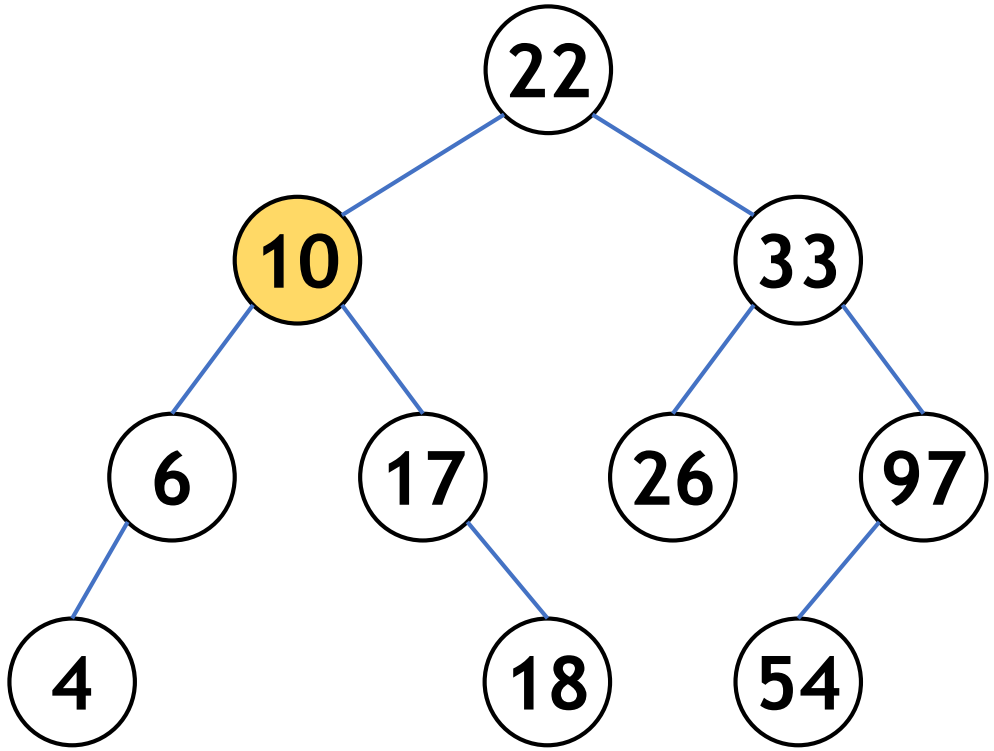
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

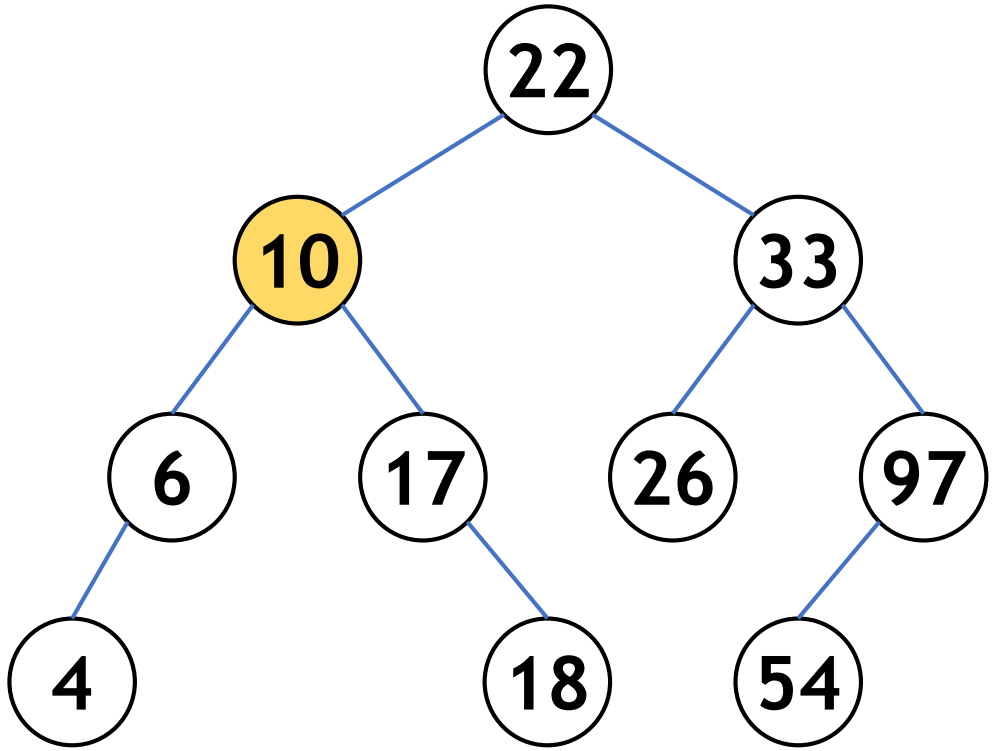
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

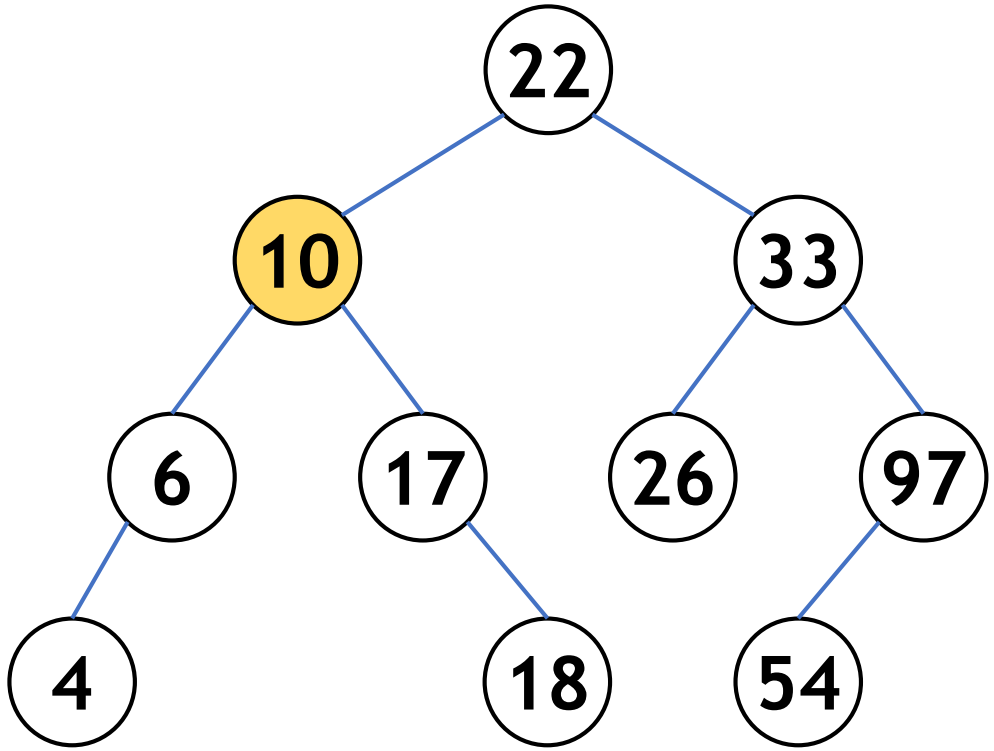
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
   return createNode(key)
else if (key <= root->data)
   root->left = insert(root->left, key)
else
   root->right = insert(root->right, key)

return root
```

BST: Insertion

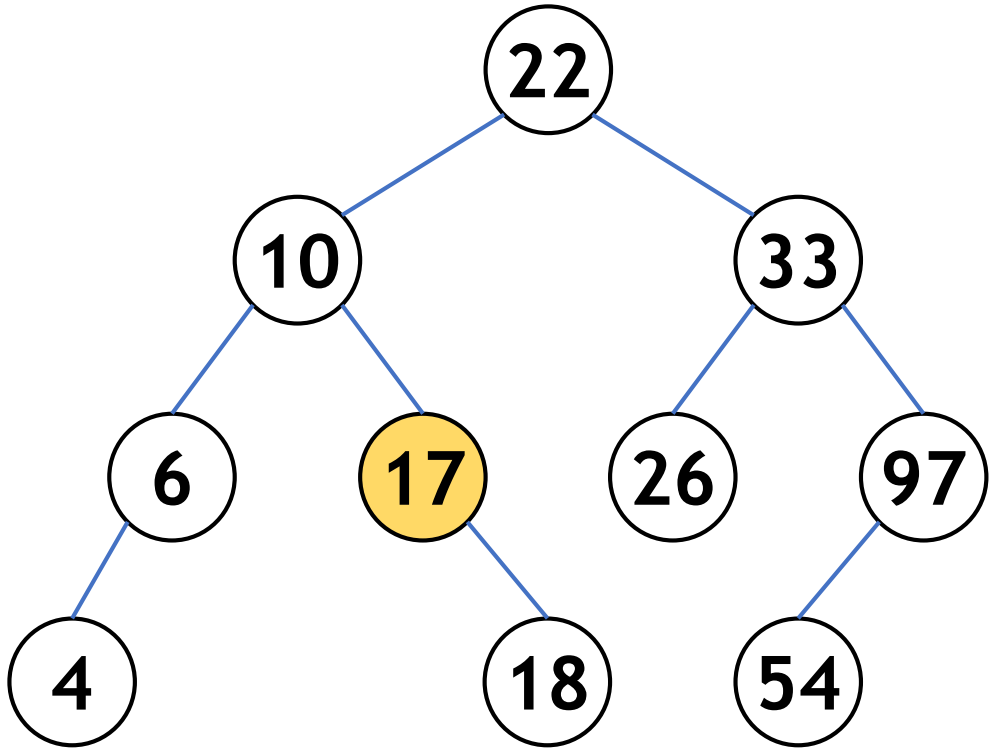
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

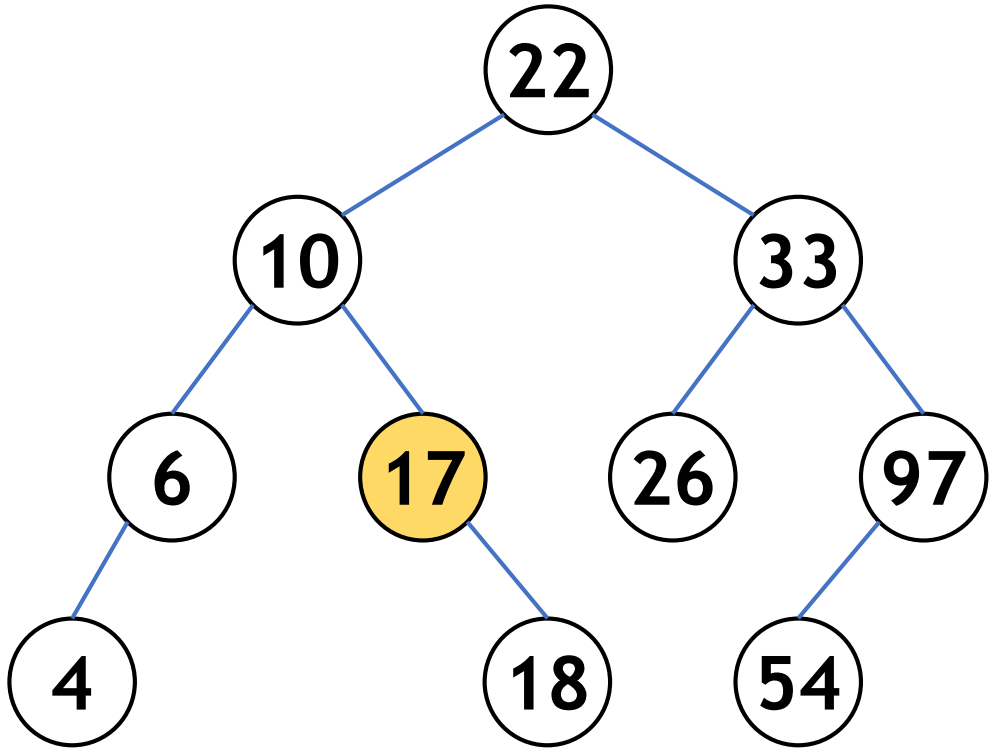
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

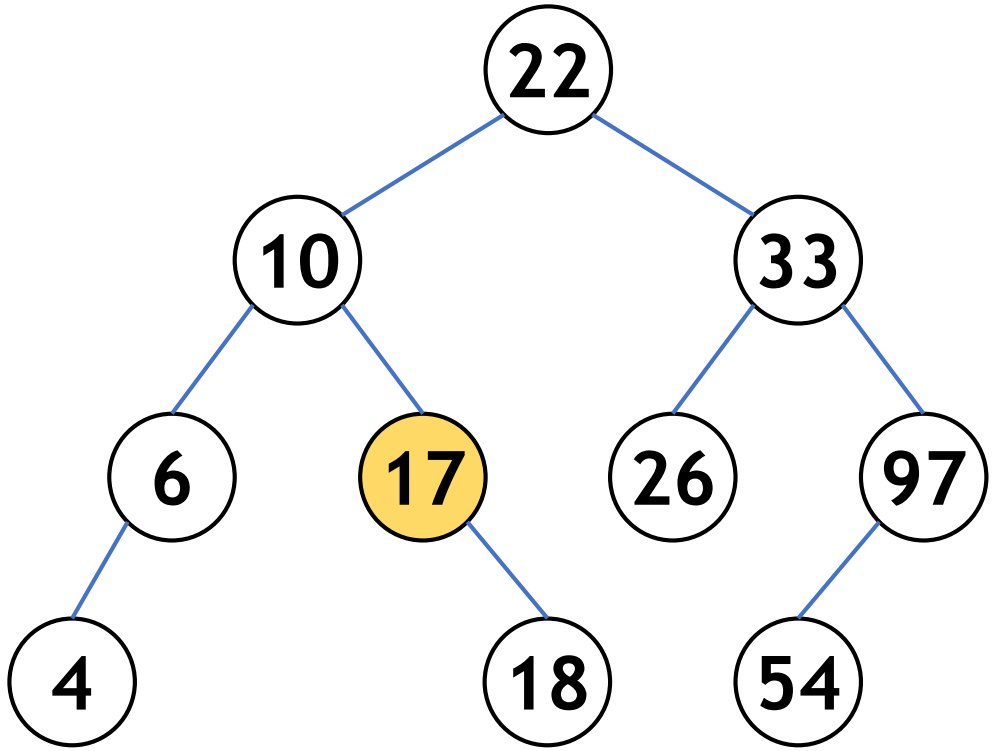
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

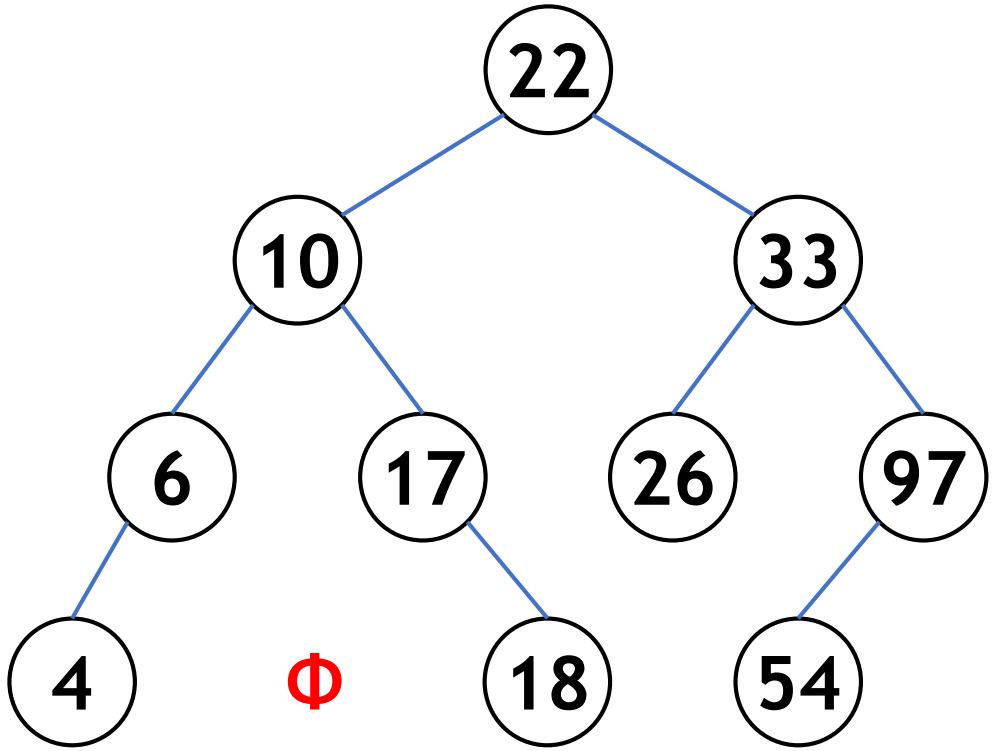
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

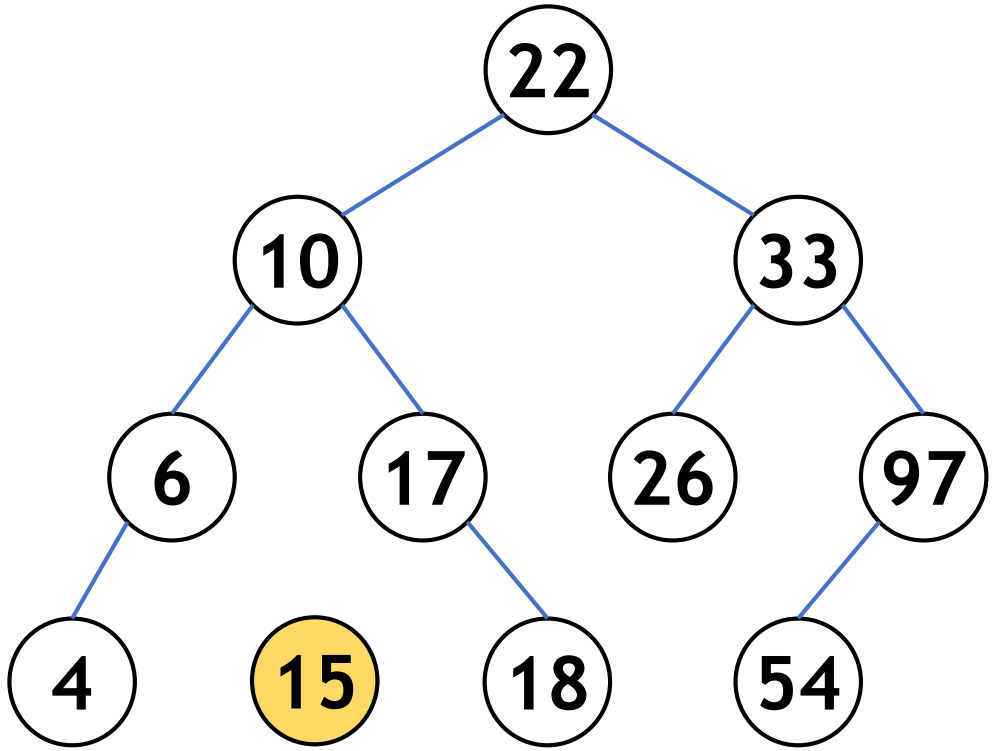
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
   return createNode(key)
else if (key <= root->data)
   root->left = insert(root->left, key)
else
   root->right = insert(root->right, key)

return root
```

BST: Insertion

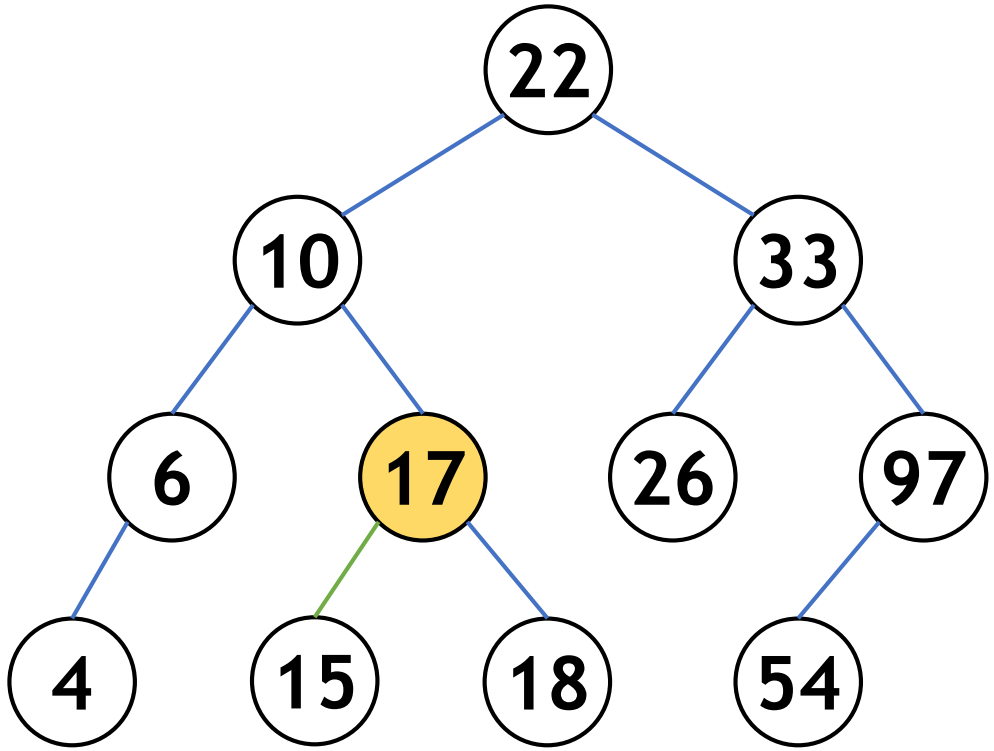
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

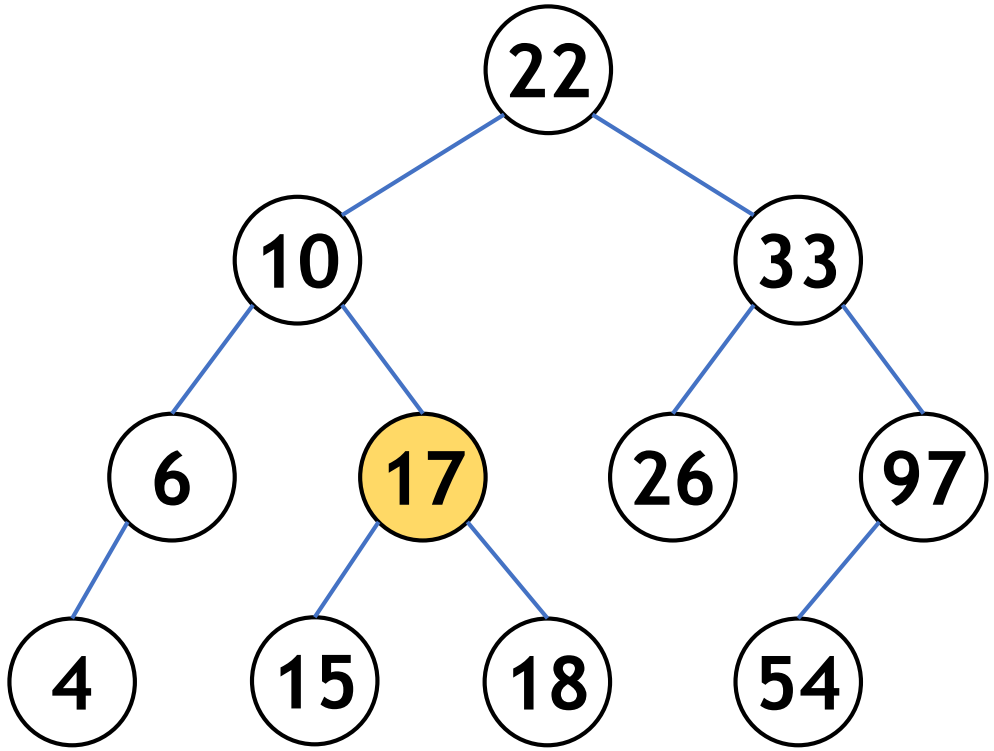
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

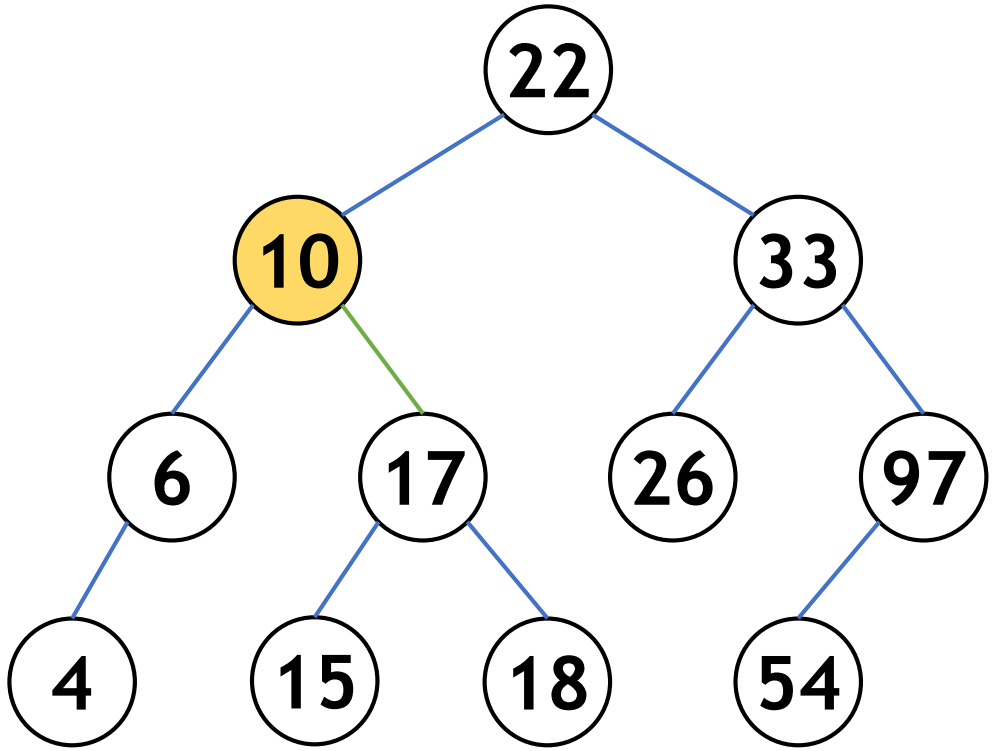
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

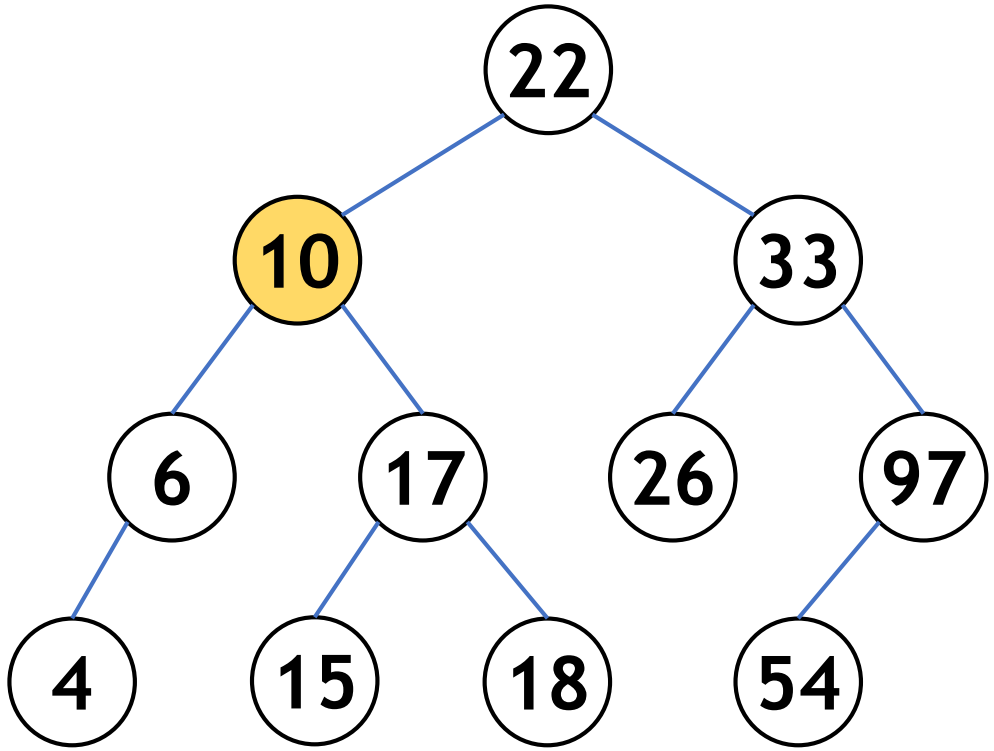
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)


return root
```

BST: Insertion

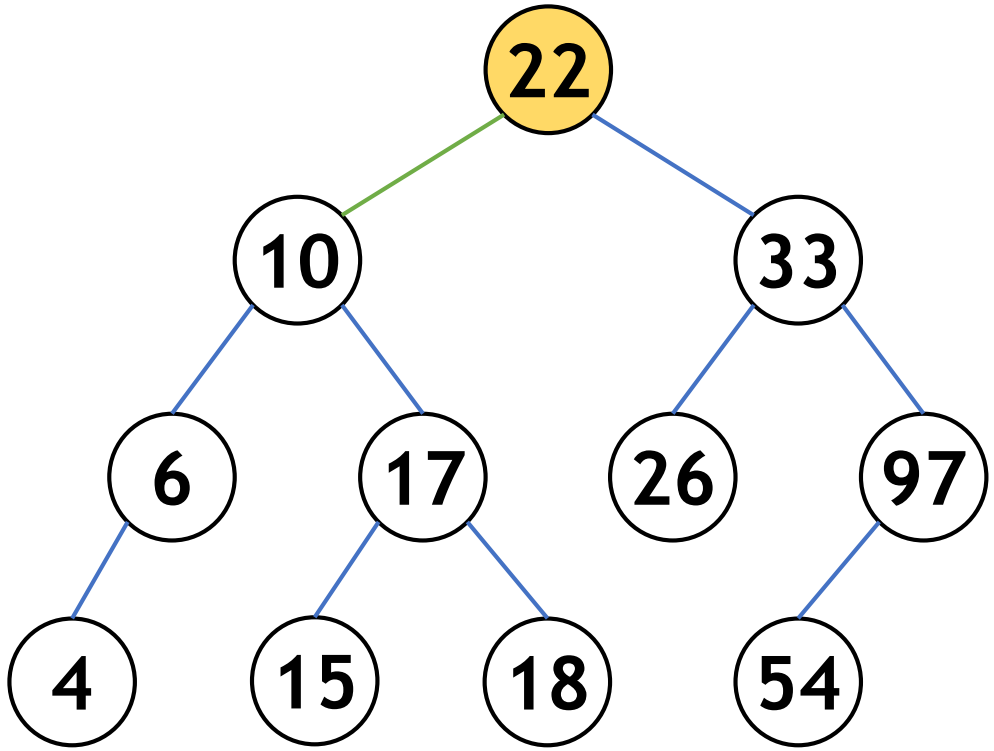
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

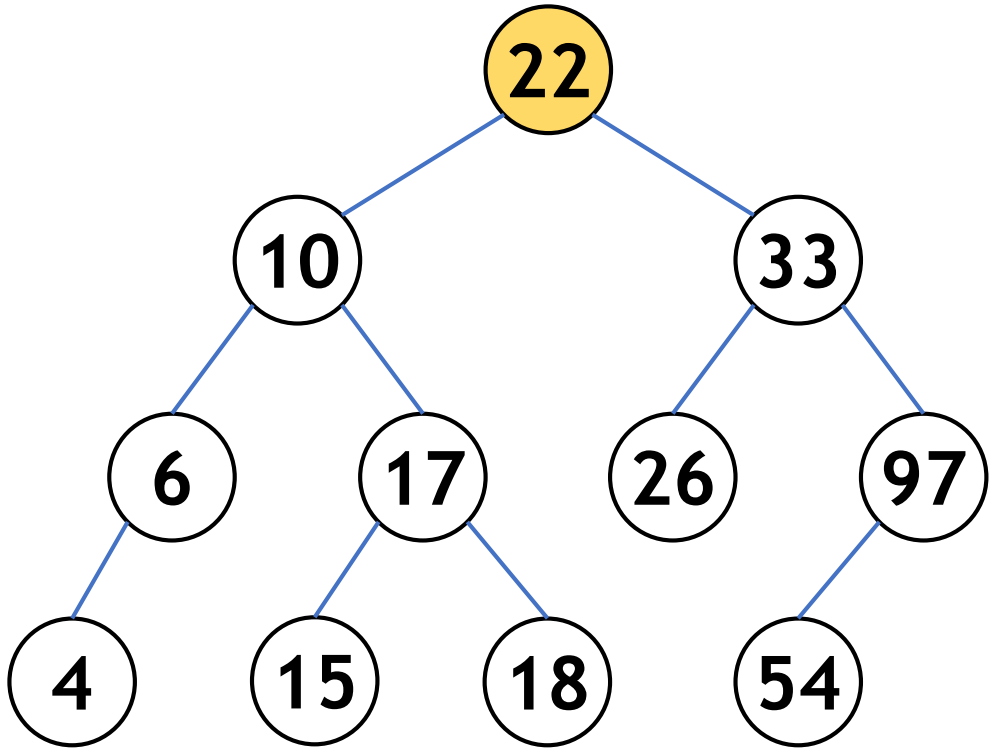
insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Insertion

insert(root, 15)




```
Node* insert(Node* root, int key)
if (!root)
  return createNode(key)
else if (key <= root->data)
  root->left = insert(root->left, key)
else
  root->right = insert(root->right, key)

return root
```

BST: Deletion

BST: Deletion

Deletion in a BST is not straightforward

Delete a node *s.t.* the given BST remains a BST post deletion

Let's consider all cases, node to be deleted:

1. **Is a leaf**: simply remove from the tree
2. **Has 1 child**: copy the child to node and delete child
3. **Has 2 children**: copy contents of inorder predecessor/successor & delete it

BST: Deletion

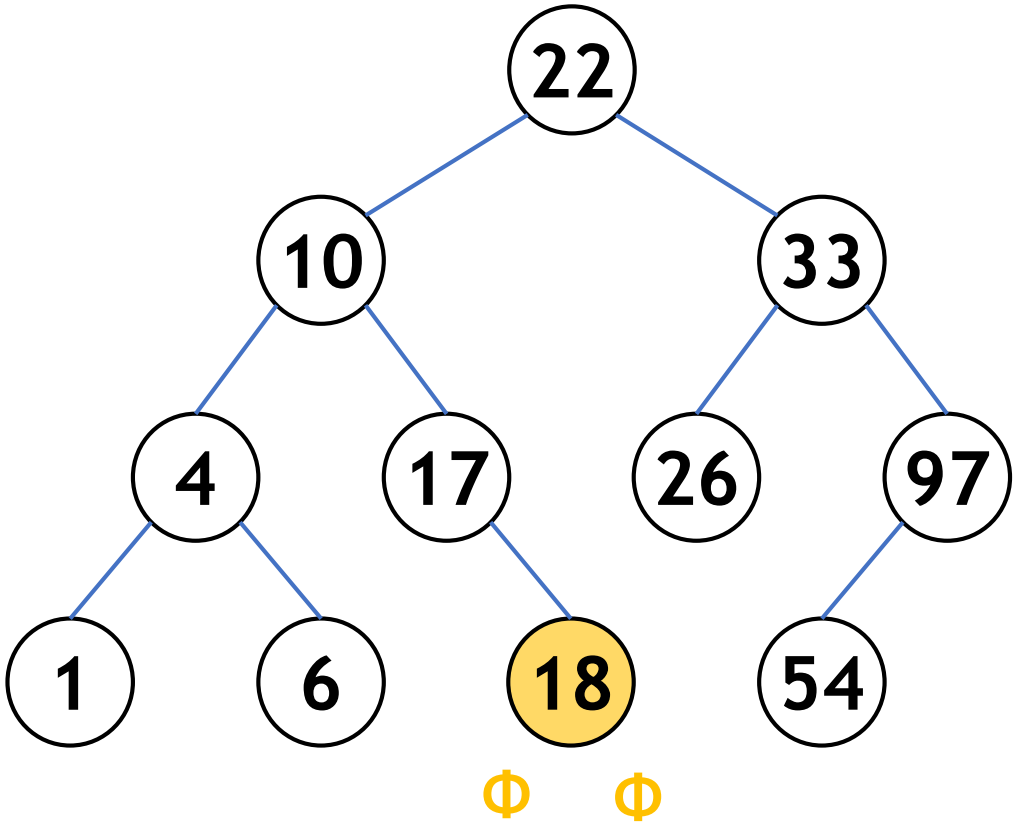
```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

```
Node* del(Node* root, int key)
if (!root)
    return root
if (key < root->data)
    root->left = del(root->left, key)
else if (key > root->data)
    root->right = del(root->right, key)
else { // node to be deleted }

return root;
```

```
if(root->left == nullptr) {
    Node* temp = root->right;
    delete root; return temp;
}
else if(root->right == nullptr) {
    Node* temp = root->left;
    delete root; return temp;
}
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 1)

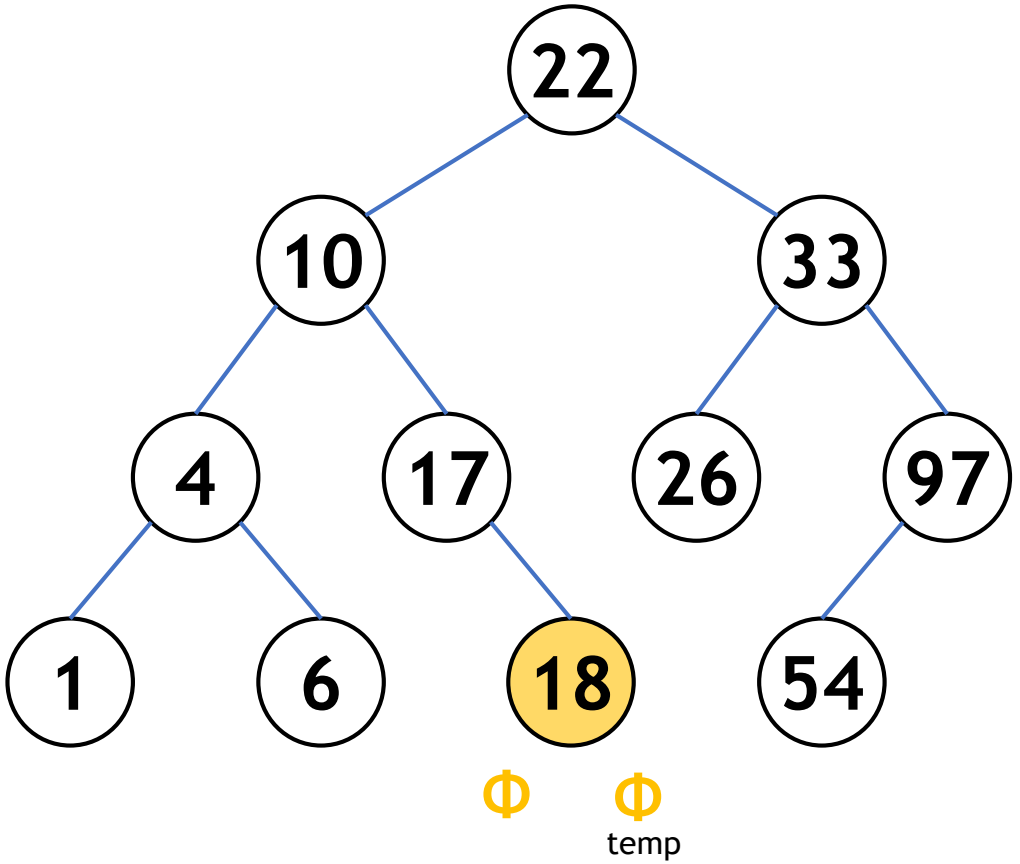
del(root, 18)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;
```

BST: Deletion (Case 1)

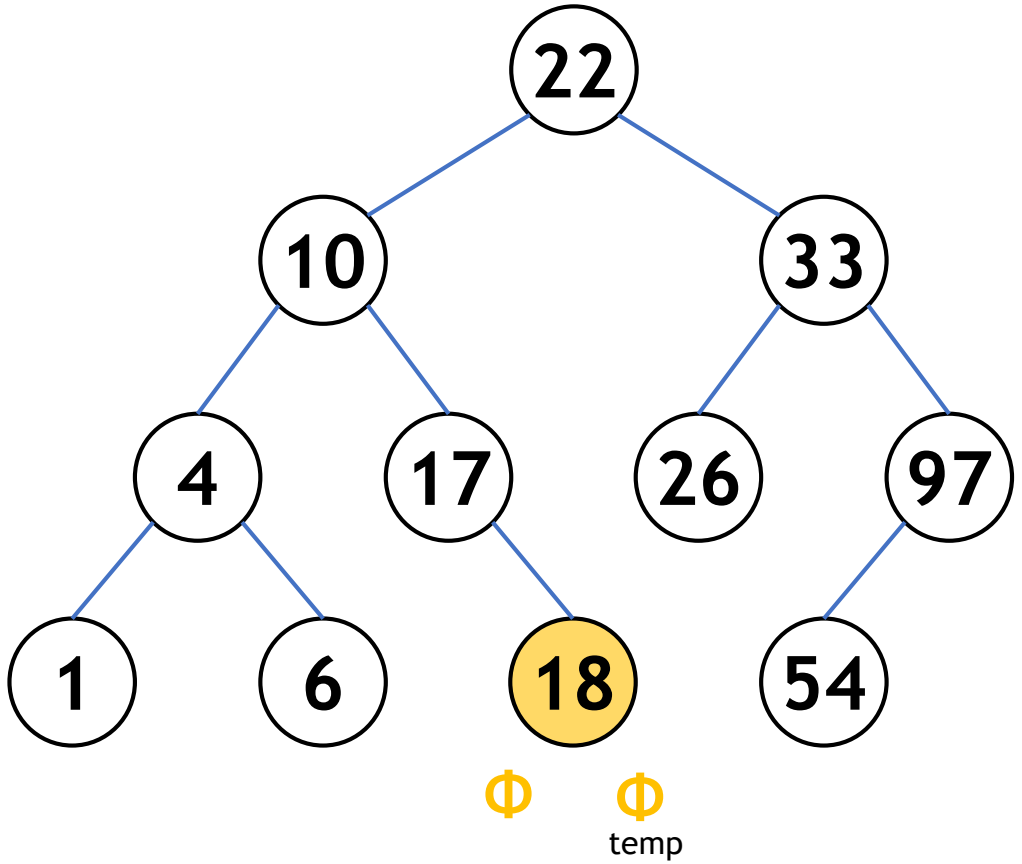
del(root, 18)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;
```

BST: Deletion (Case 1)

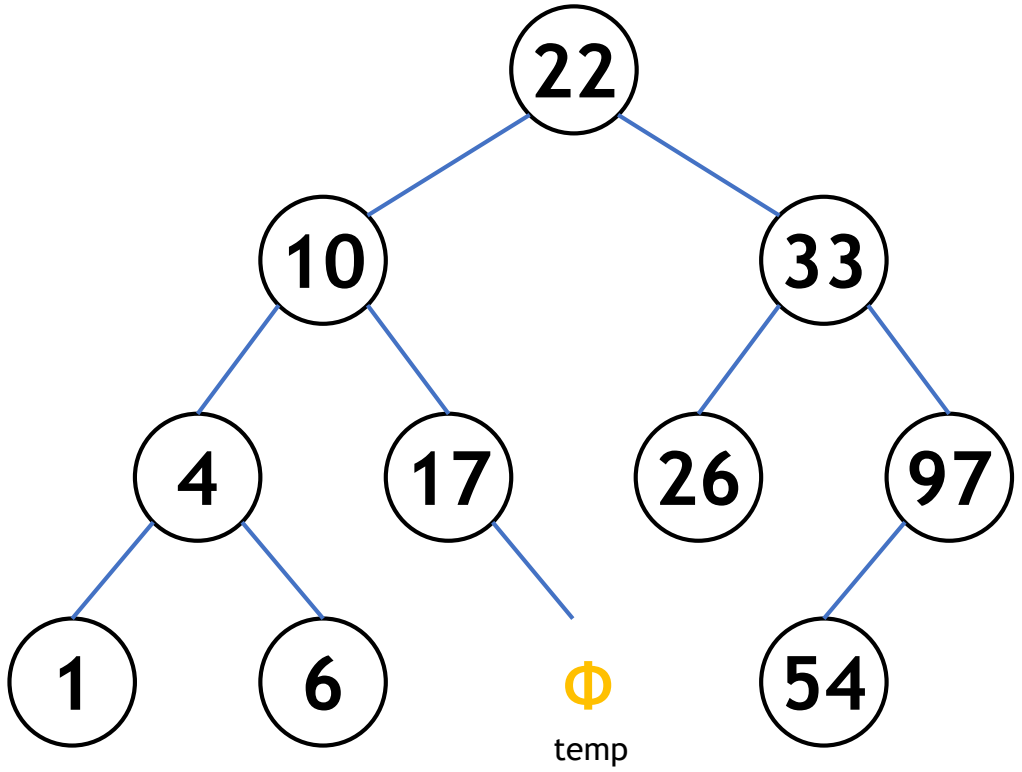
del(root, 18)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;
```

BST: Deletion (Case 1)

del(root, 18)

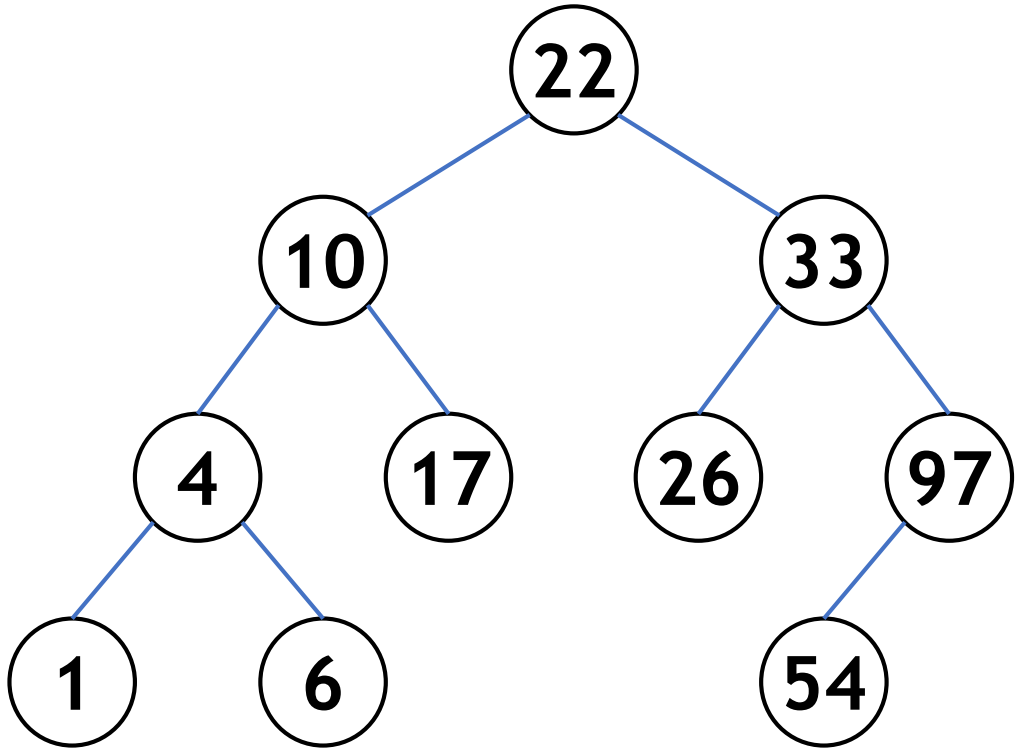



```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;
```

$root_{17}$->right = nullptr;

BST: Deletion (Case 1)

del(root, 18)

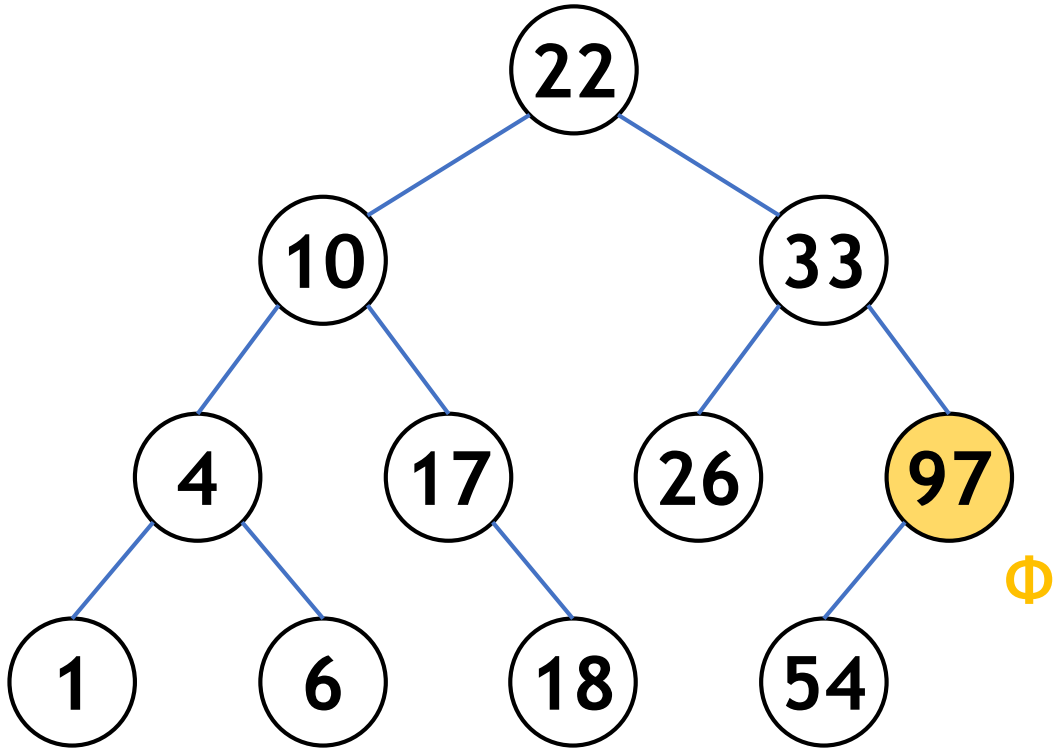



```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;
```

$root_{17}$->right = nullptr;

BST: Deletion (Case 2)

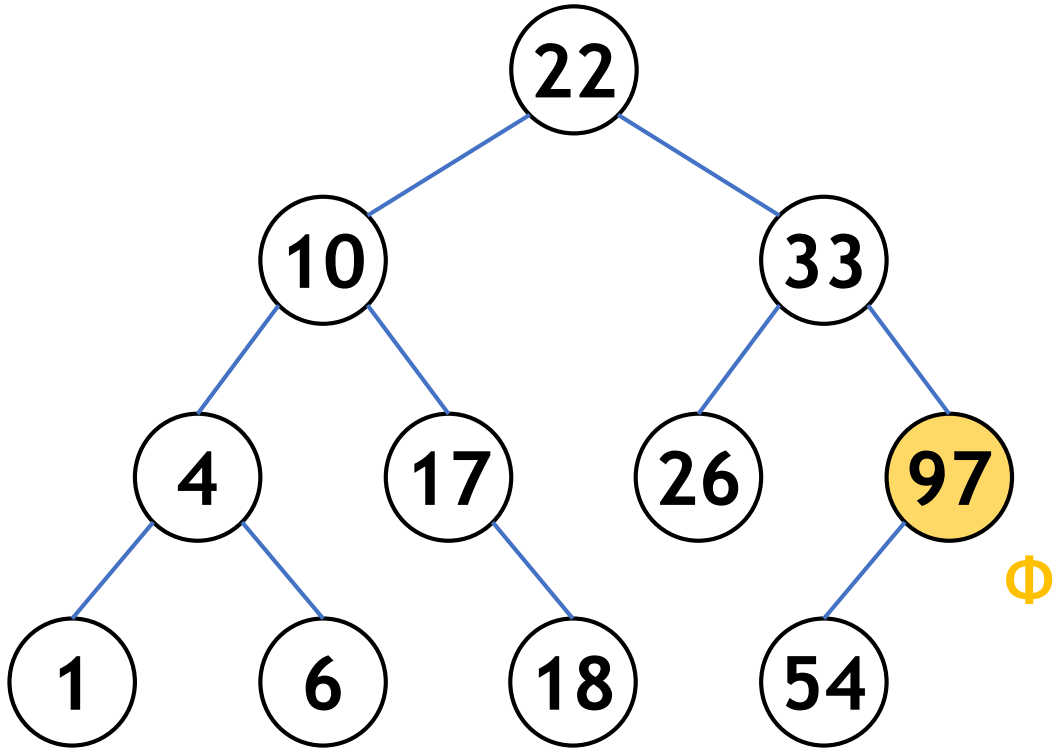
del(root, 97)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;

else if(root->right == nullptr)
    Node* temp = root->left;
    delete root;
    return temp;
```

BST: Deletion (Case 2)

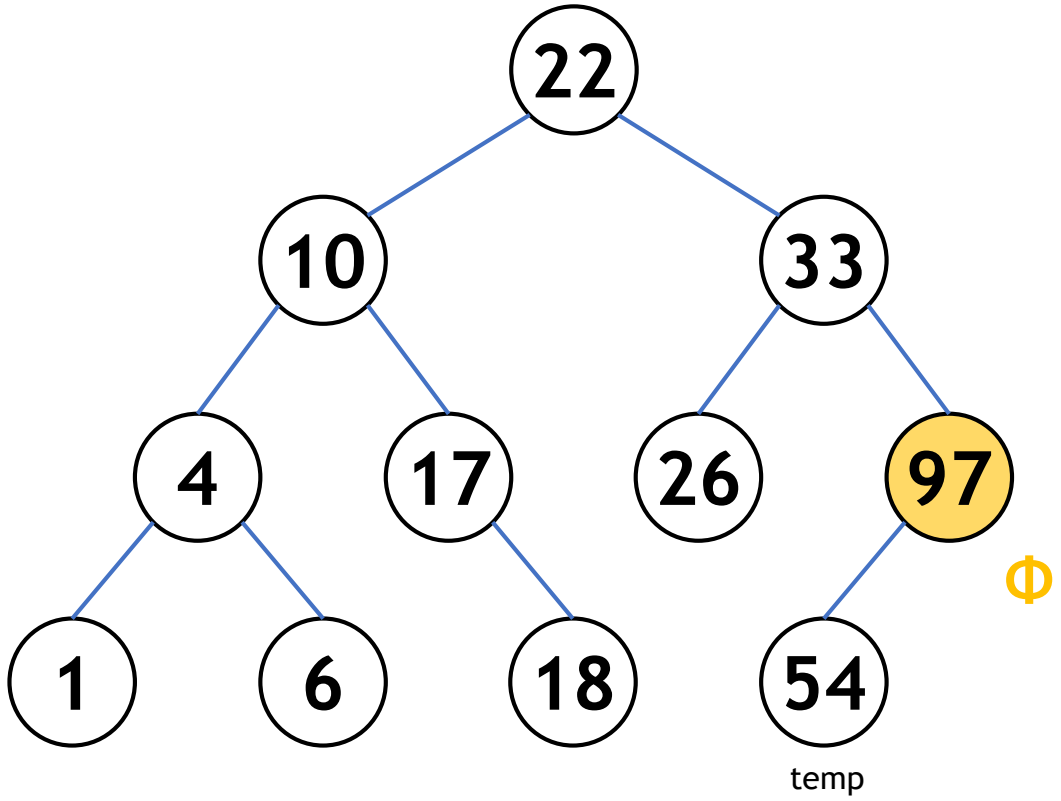
del(root, 97)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;

else if(root->right == nullptr)
    Node* temp = root->left;
    delete root;
    return temp;
```

BST: Deletion (Case 2)

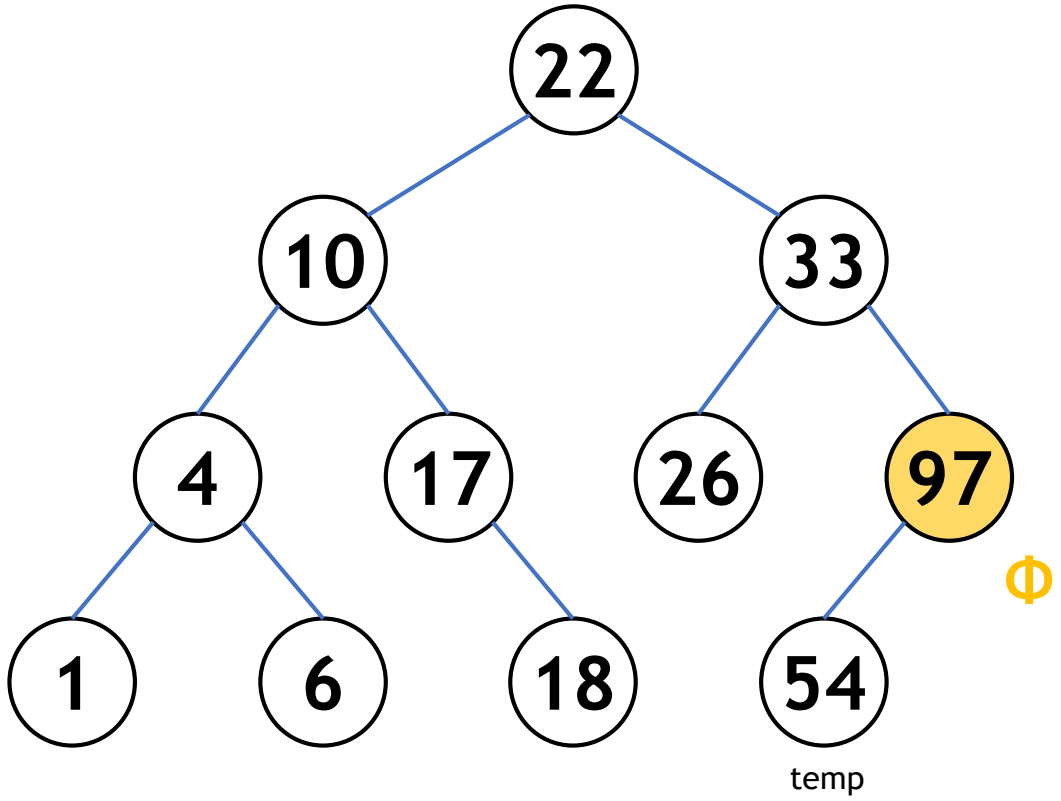
del(root, 97)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;

else if(root->right == nullptr)
    Node* temp = root->left;
    delete root;
    return temp;
```

BST: Deletion (Case 2)

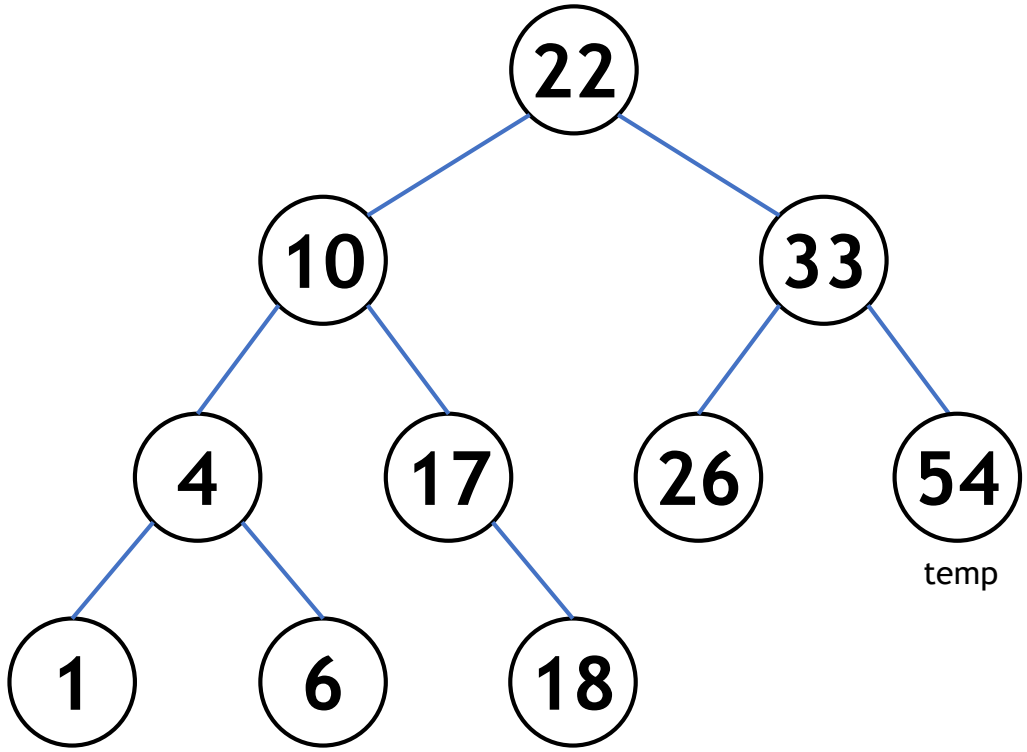
del(root, 97)




```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;

else if(root->right == nullptr)
    Node* temp = root->left;
    delete root;
    return temp;
```

BST: Deletion (Case 2)

del(root, 97)



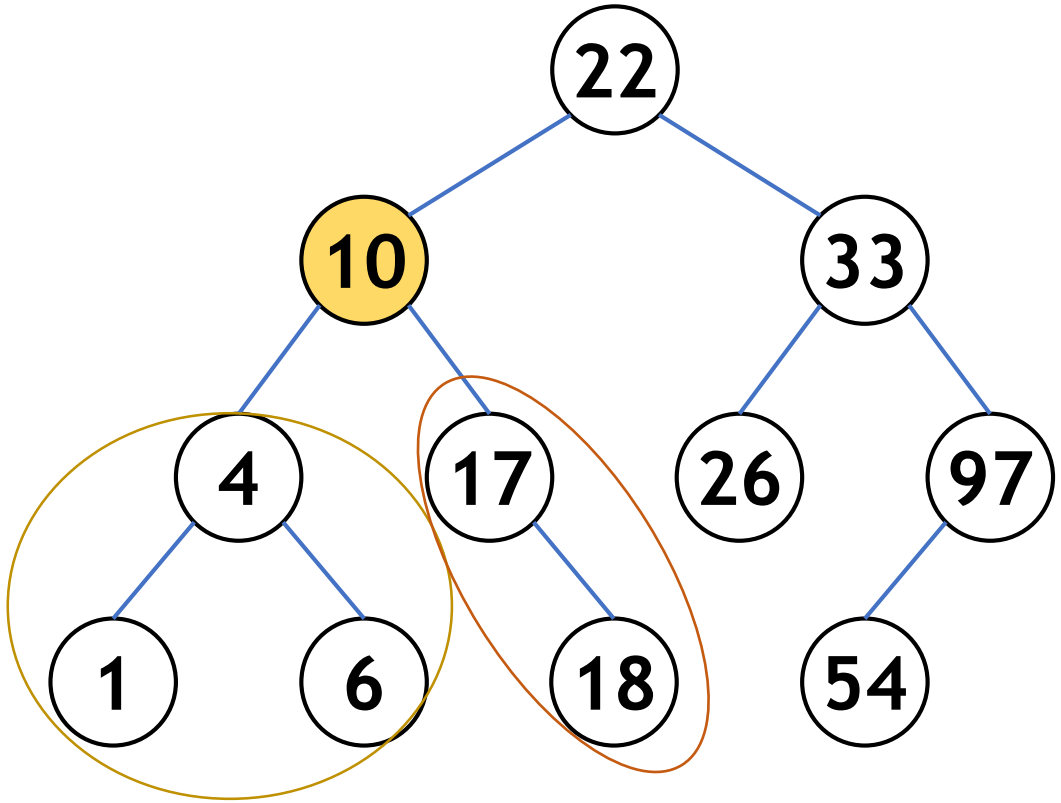

```
if(root->left == nullptr)
    Node* temp = root->right;
    delete root;
    return temp;

else if(root->right == nullptr)
    Node* temp = root->left;
    delete root;
    return temp;
```

$root_{33}$->right = del($root_{33}$->right);

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)

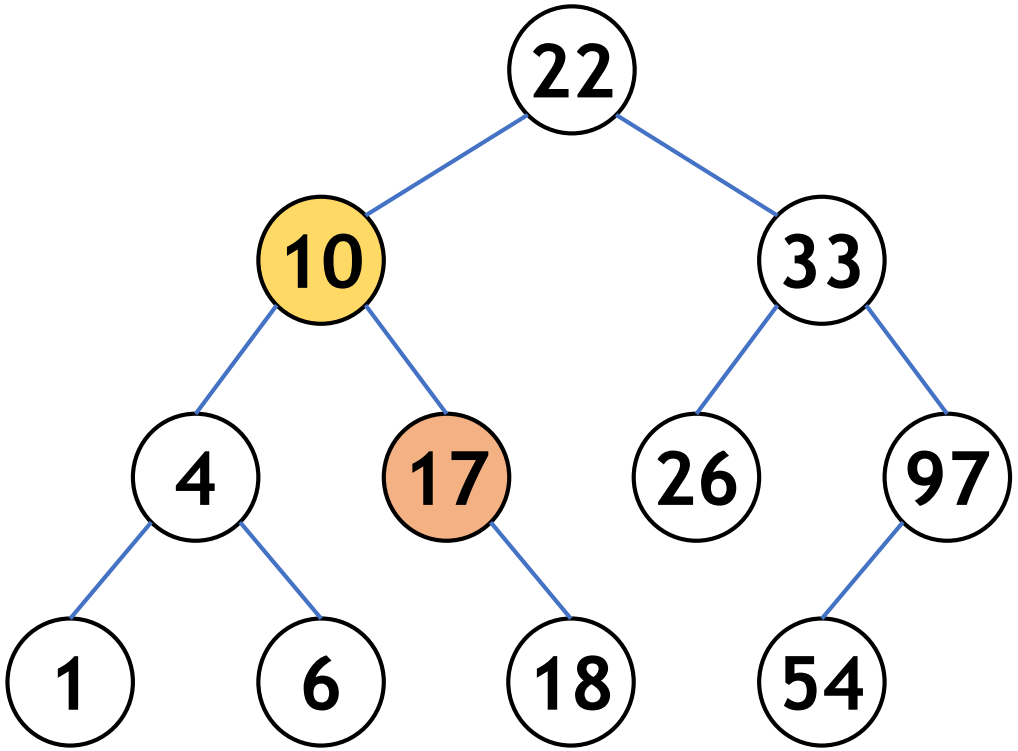



```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)

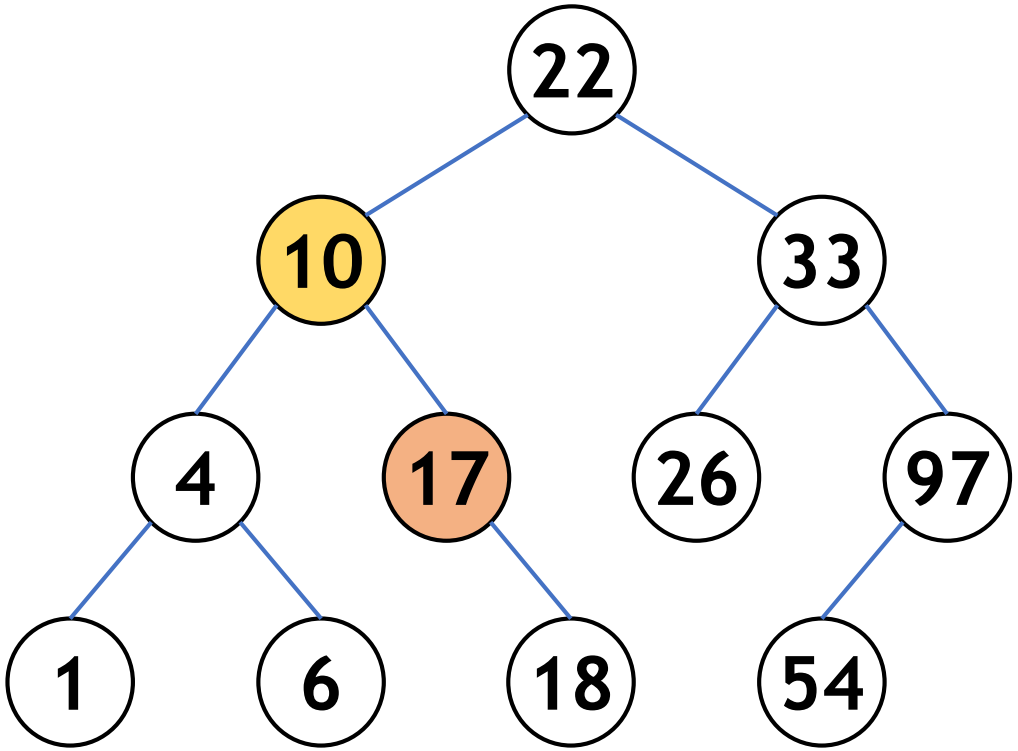



```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)

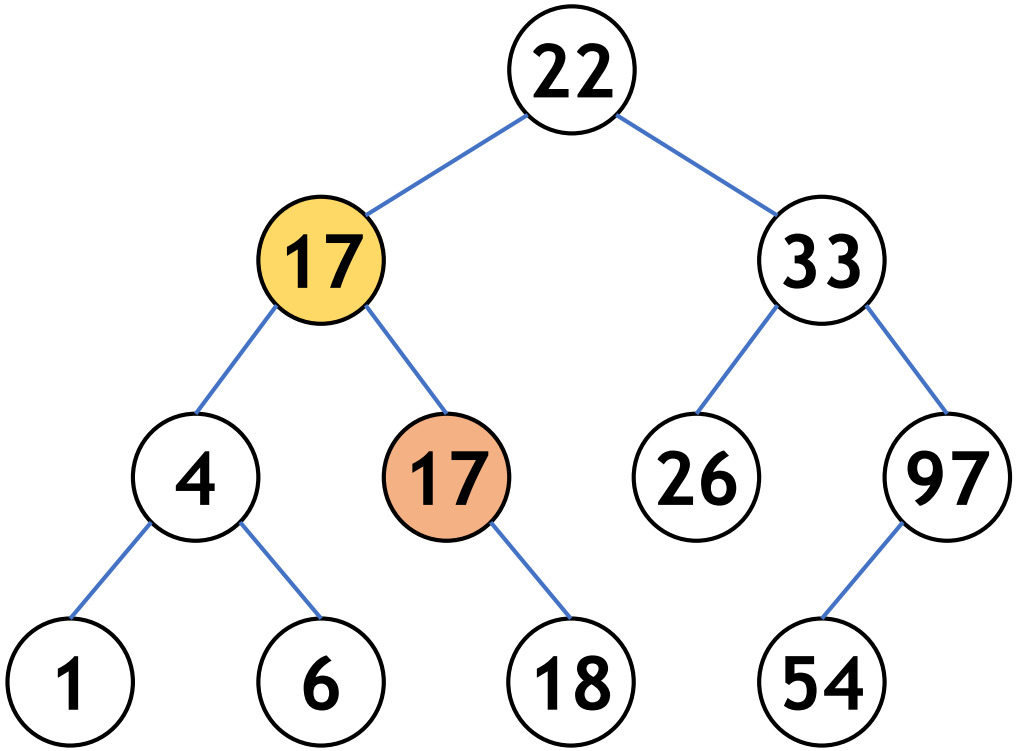



```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)

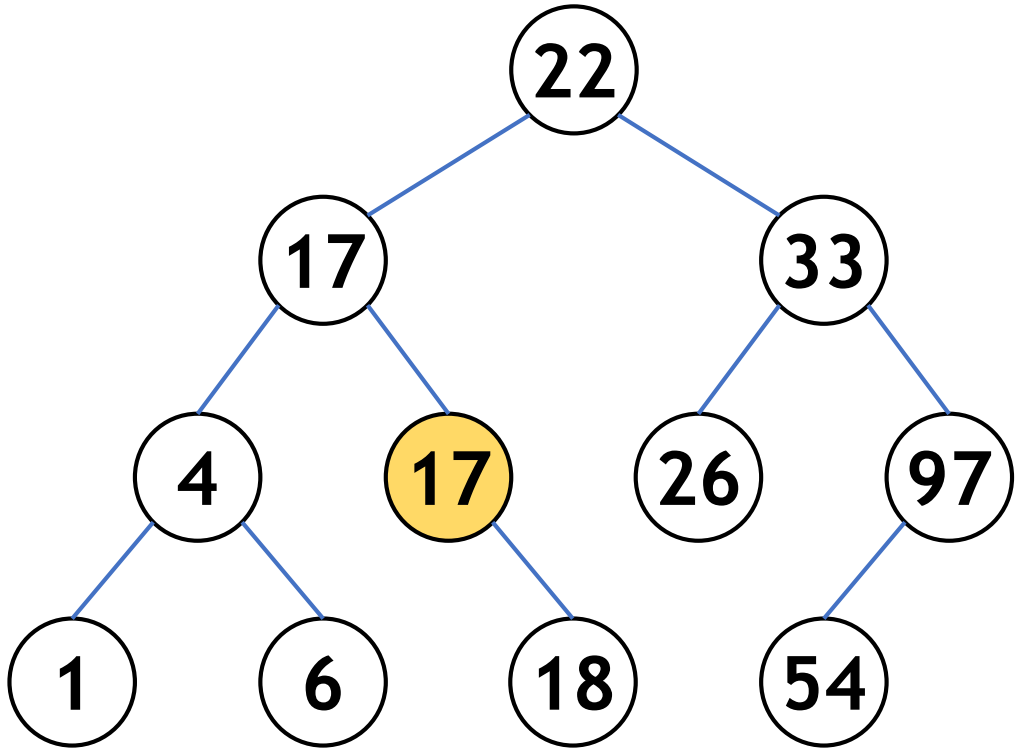



```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)

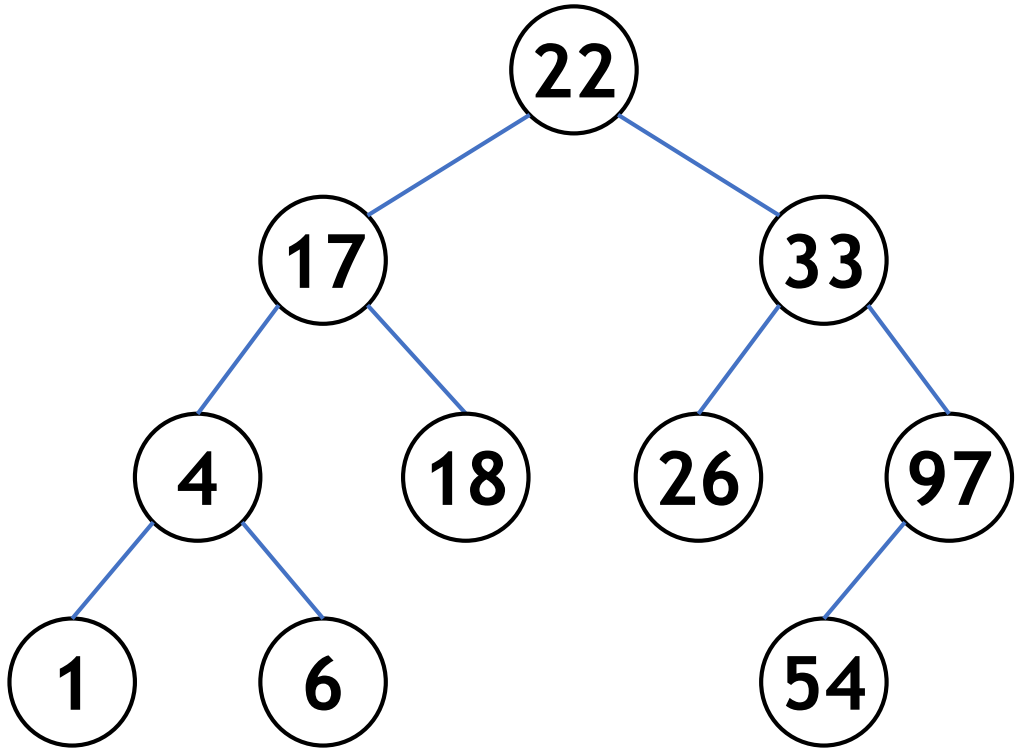



```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Deletion (Case 3)

```
Node* minNode(Node* root) {
    Node* temp = root;
    while (temp && temp->left != nullptr)
        temp = temp->left;
    return temp;
```

del(root, 10)




```
Node* temp = minNode(root->right);
root->key = temp->key;
root->right = del(root->right, temp->key);
```

BST: Analysis

BST: Analysis

Operation	*Average*	*Worst*
Search	O(logn)	O(n)
Insert	O(logn)	O(n)
Delete	O(logn)	O(n)