Data Structures

Binary Search Trees

CS 225

September 22, 2025

Harsha Tirumala

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Tree Search tradeoffs

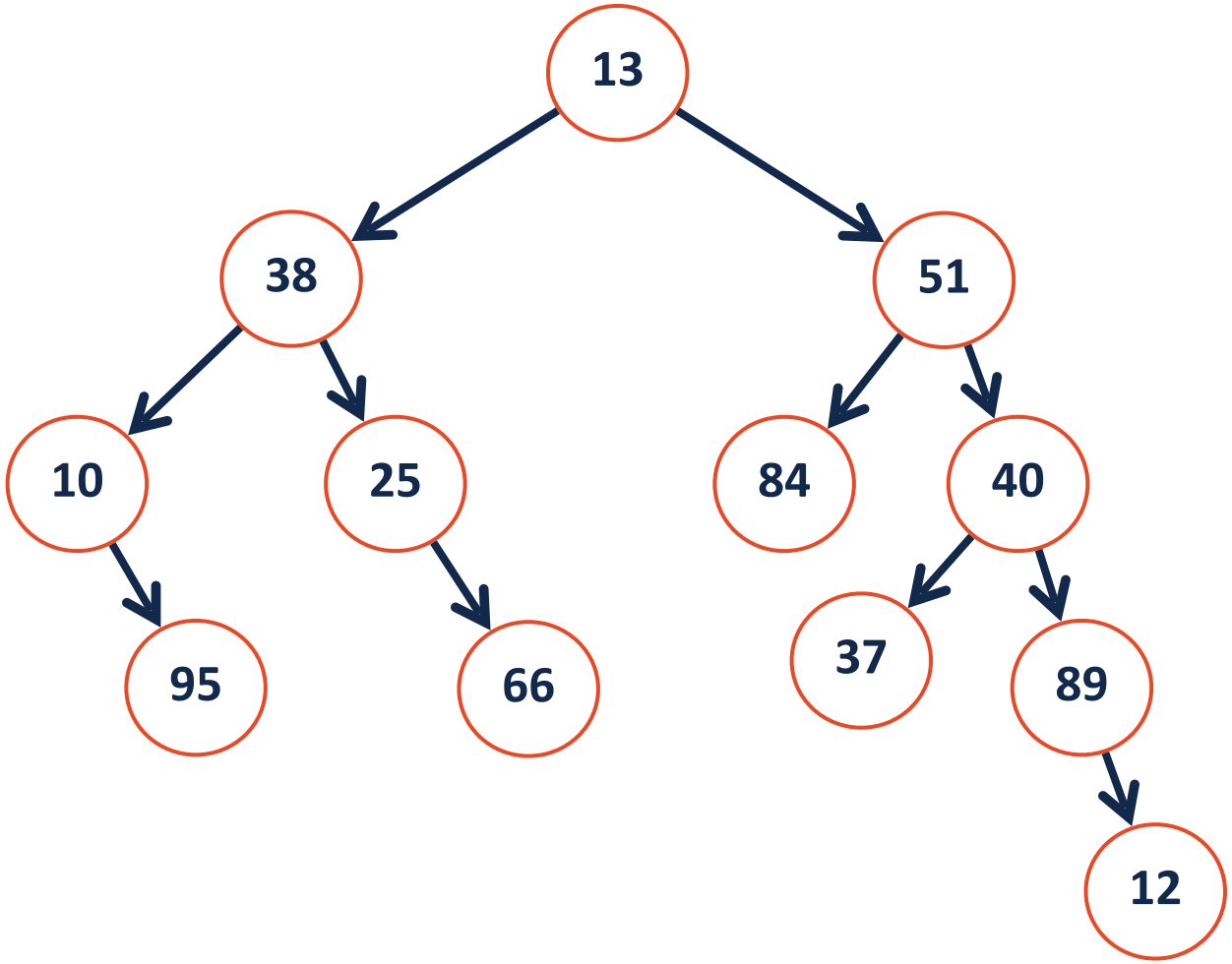
Extend binary trees into binary *search* trees

Build conceptual and coding understanding of BST

Binary Trees

Operation	Implementation	Time complexity
Insert	Can Insert anywhere - so insert as new root with previous root as left/right child	O(1)
Remove	Implemented as find and remove - find requires traversal + remove can be done by **swap**	O(n) for find + remove
Traverse	Can use any tree traversal - pre/in/post/ level order	O(n)
Find	DFS/BFS	O(n)

Unstructured Binary Tree



Tree Search Tradeoffs

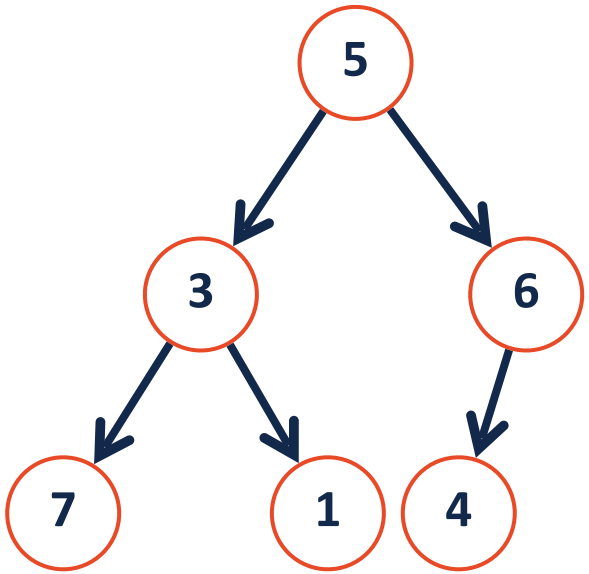
How can we improve our ability to search a binary tree?

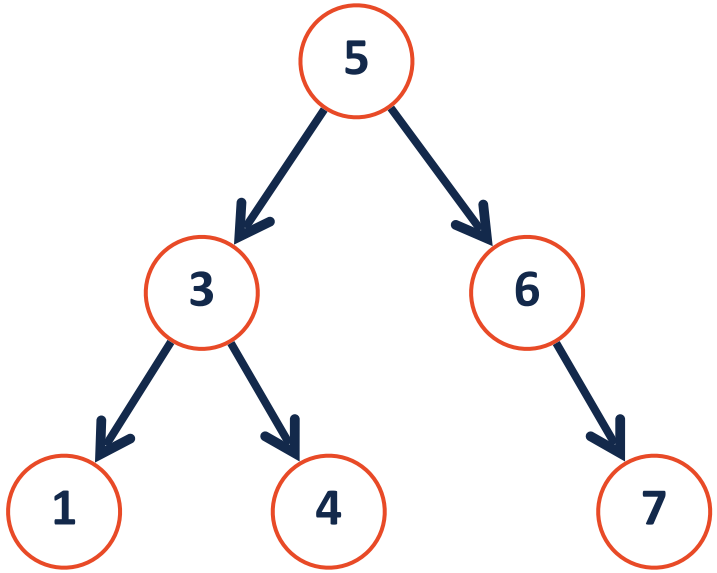
What do we trade in order to do so?

Improved Search - Unstructured vs Structured BT

5	3	6	7	1	4

1	3	4	5	6	7





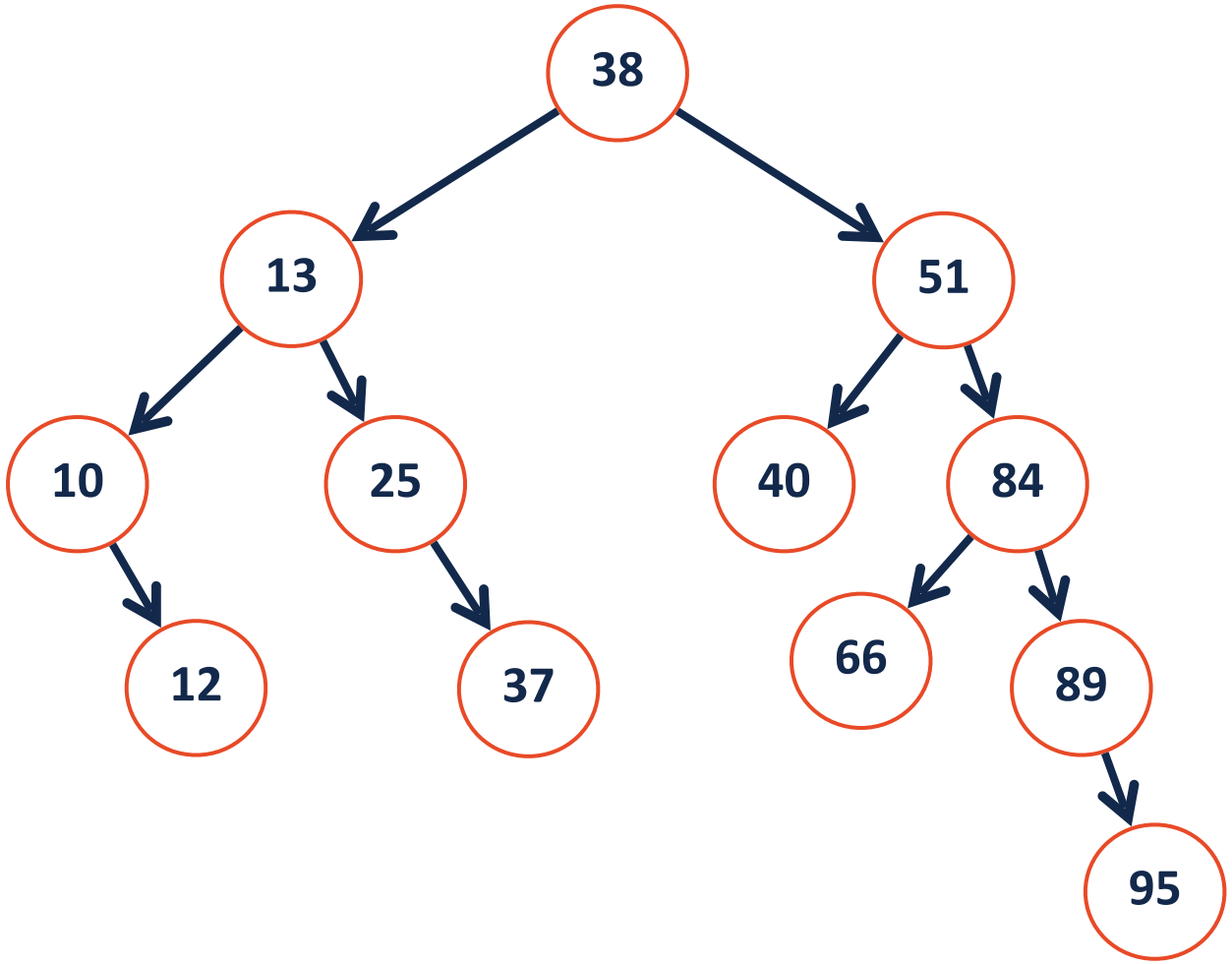
Dictionary ADT

Real (meaningful) Data is often organised into key/value pairs:

Word ➔ Definition
Course Number ➔ Lecture/Lab Schedule
Node ➔ Incident Edges
Flight Number ➔ Arrival Information
URL ➔ HTML Page
Net id ➔ Student name, DOB, Degree, Department, etc.
…

Binary Search Tree (BST)

A **BST** is a binary tree $T = TreeNode(key, T_L, T_r)$ such that:



BST.h

```
1  #pragma once
2
3  template <typename K, typename V>
4  class BST {
5    public:
6     /* ... */
7    private:
8      class TreeNode {
9         K & key;
10        V & value;
11
12        TreeNode *left, *right;
13
14        TreeNode(K & k, V & v) :
15 key(k), value(v),
16 left(NULL), right(NULL) { }
17     };
18
19     TreeNode *root_;
20    /* ... */
21 };
22
23
```

Tree.h

```
1  #pragma once
2
3  template <typename T>
4  class BinaryTree {
5    public:
6      /* ... */
7    private:
8      class TreeNode {
9        T & data;
10
11       TreeNode * left;
12
13       TreeNode * right;
14
15       TreeNode(T & data) :
16        data(data), left(NULL),
17 right(NULL) { }
18
19     };
20
21     TreeNode *root_;
22     /* ... */
23 };
```

Binary Search Tree ADT

Insert

Remove

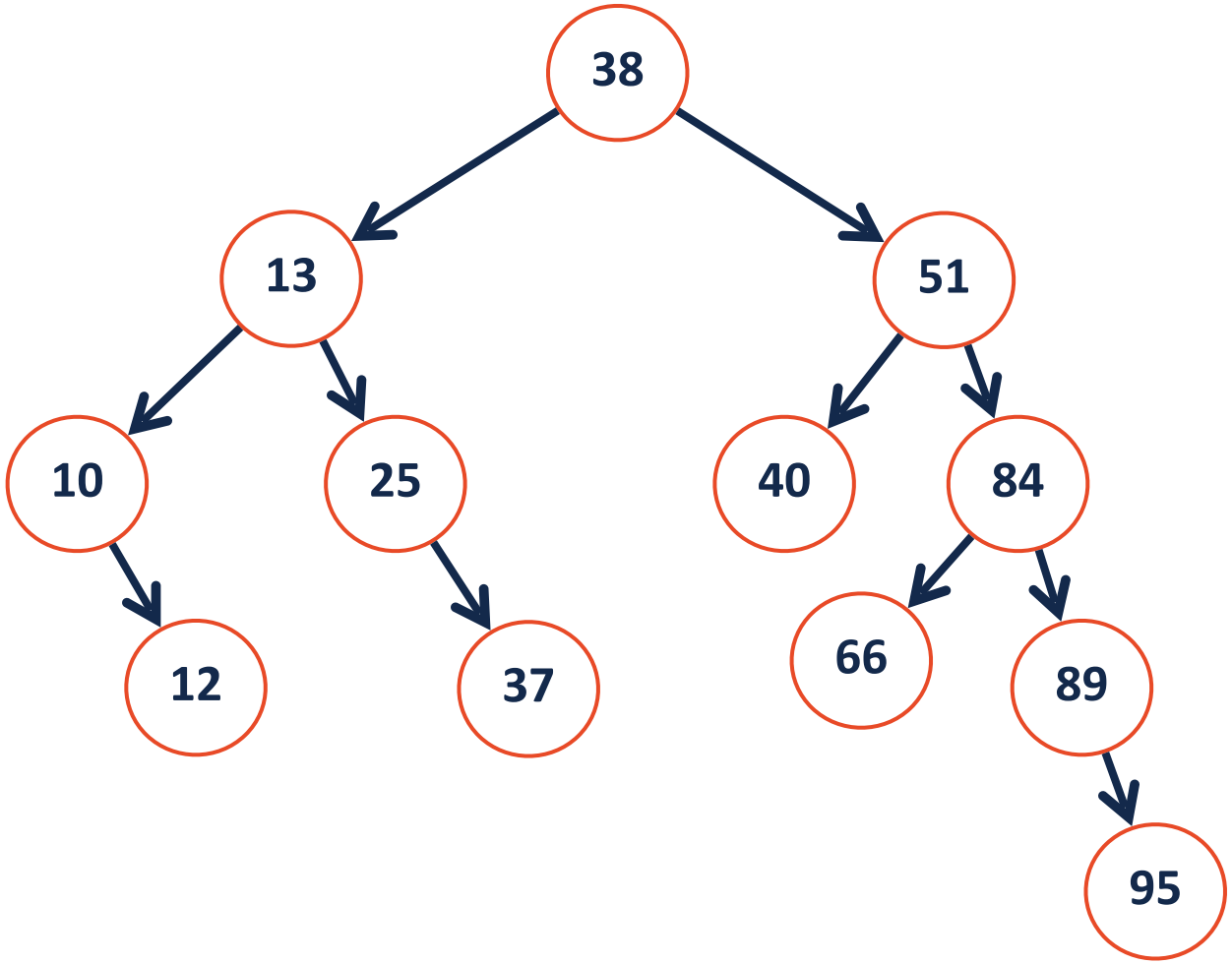
Traverse

Find

Constructor

BST Find

`find(66)`



BST Find

`find(66)`

A recursive function based around value of root:

Base Case: If root is `null`, return root

Let `tmp = root->key()`

`tmp == query`, return root

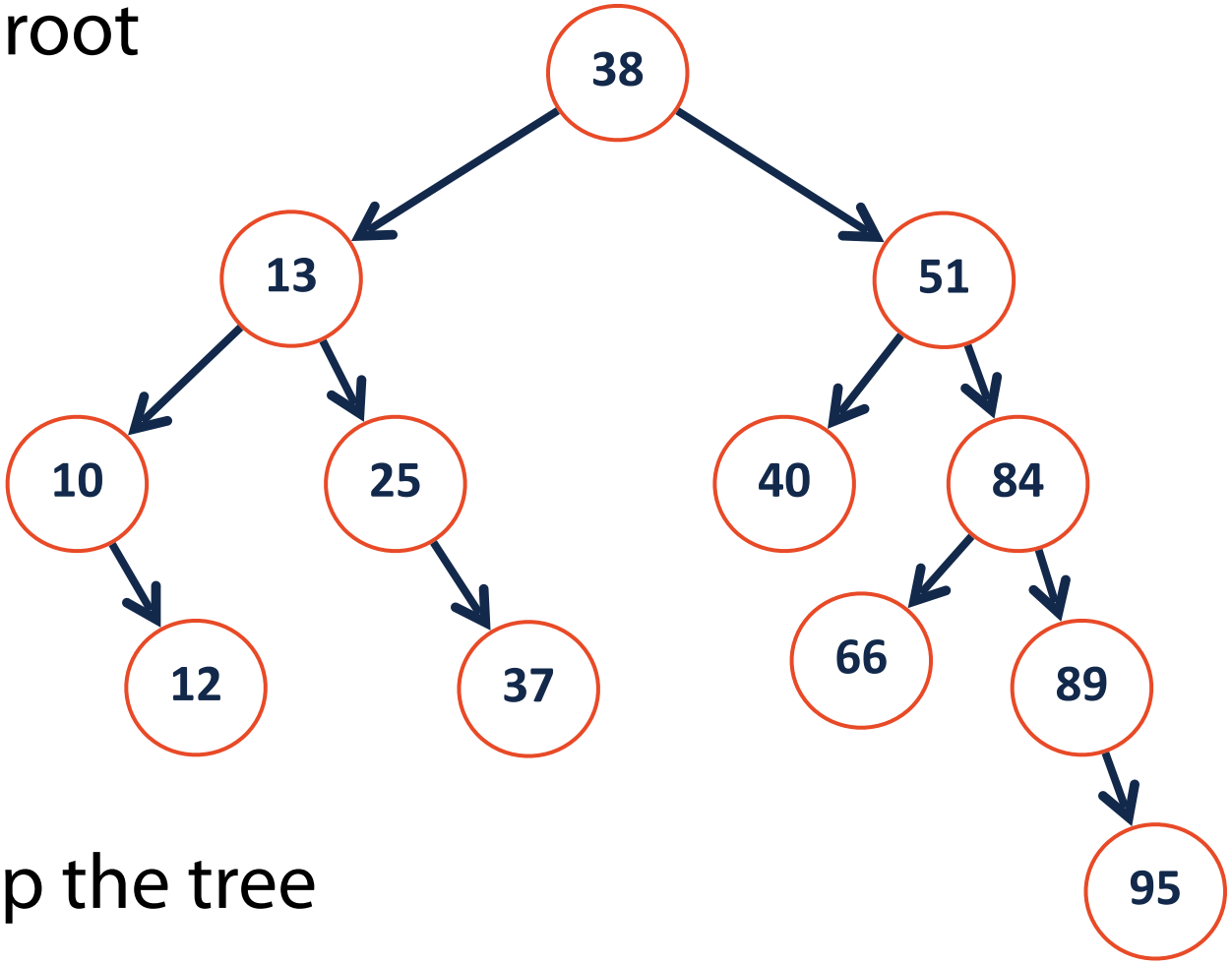
Recursion:

`tmp < query`, recurse right

`tmp > query`, recurse left

Combining:

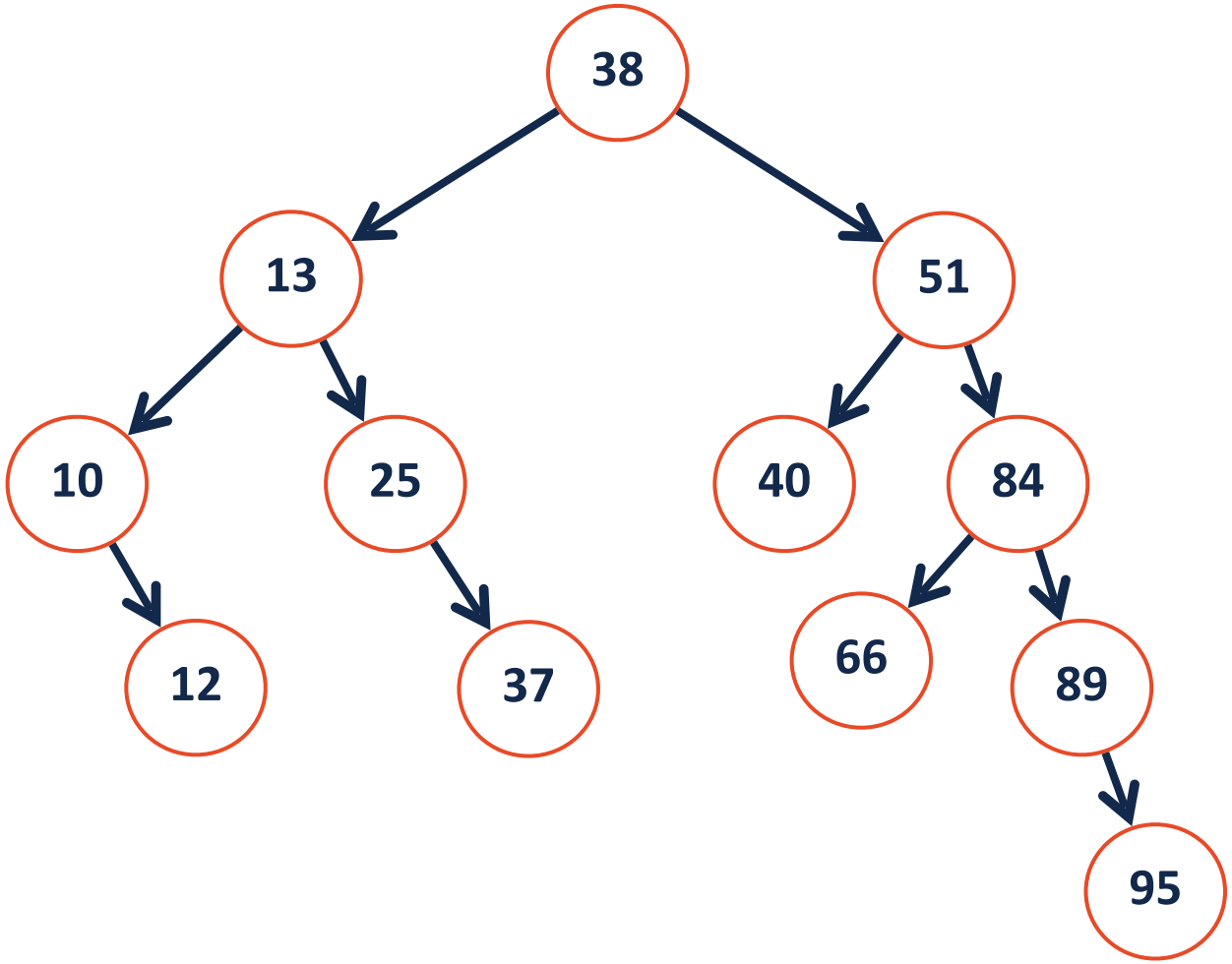
Return the recursive value back up the tree



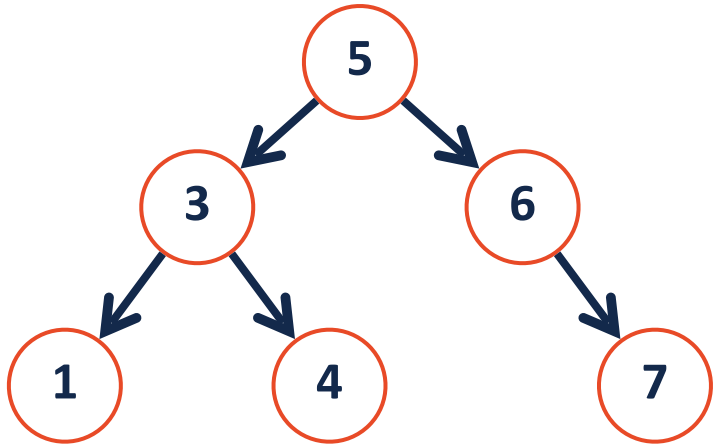
BST Find

`find(9)`

Make sure you can follow the search path!

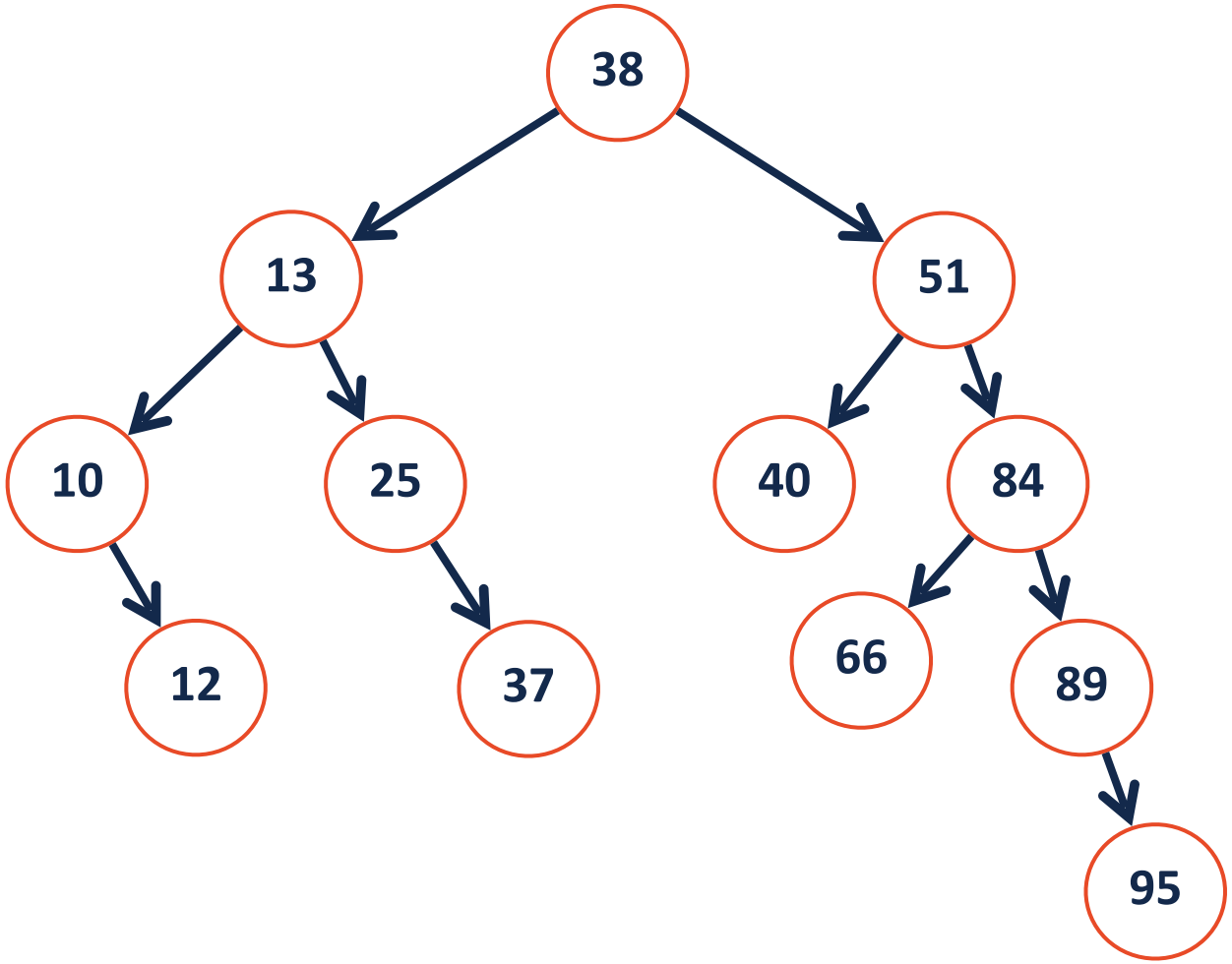


```
1  template<typename K, typename V>
2
3        TreeNode *&          _find(TreeNode *& root, const K & key) {
4
5
6  // Base Case
7  if(root == nullptr || root->data == key){
8      return root;
9  }
10
11 // Recursive Step ("Combining step" is 'return')
12 if (root->data > key){
13     return _find(root->left, key);
14 }
15
16 return _find(root->right, key);
17
18
19 }
20
21
22
23
```



BST Insert

`insert(33, v)`

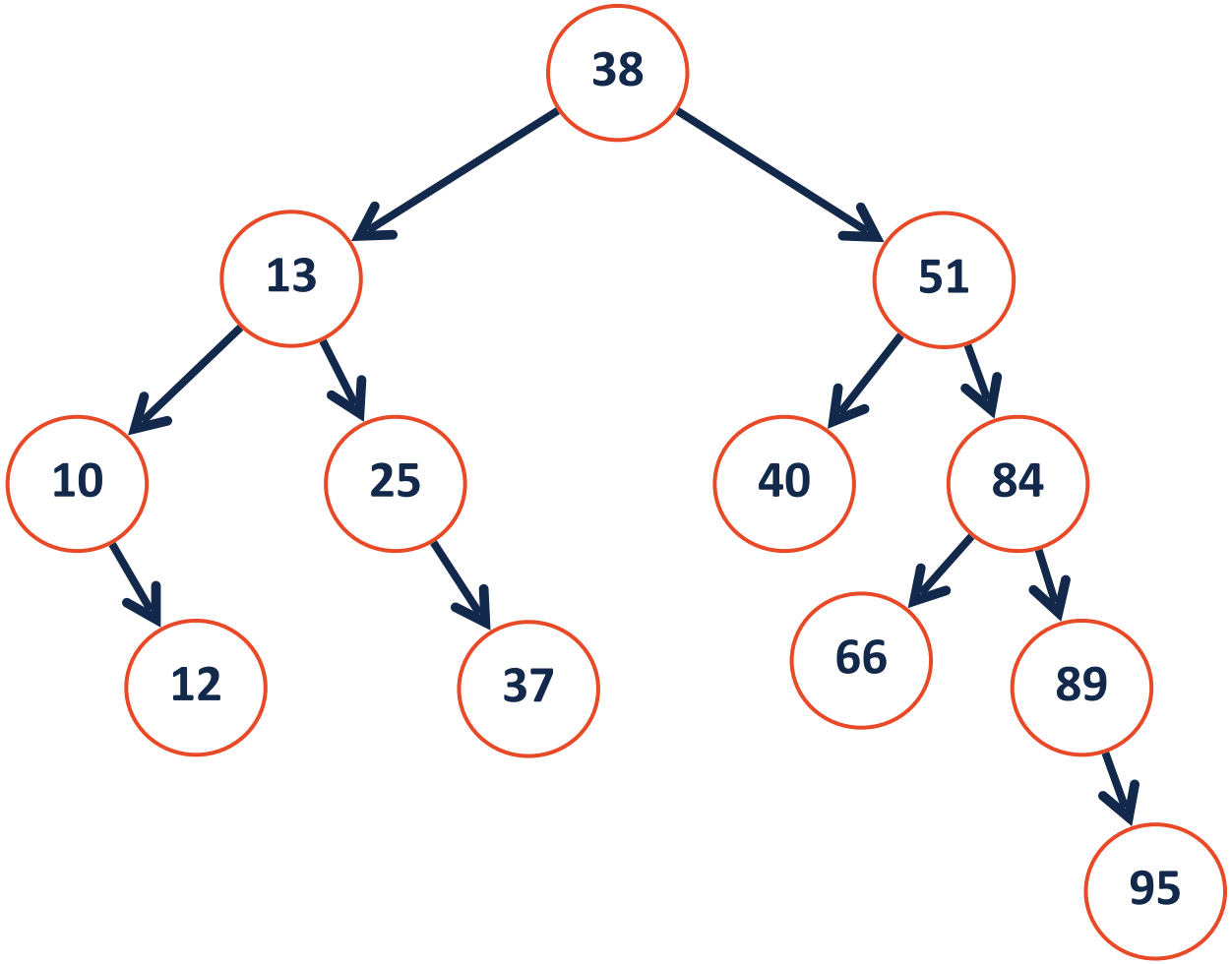


BST Insert

`insert(33, v)`

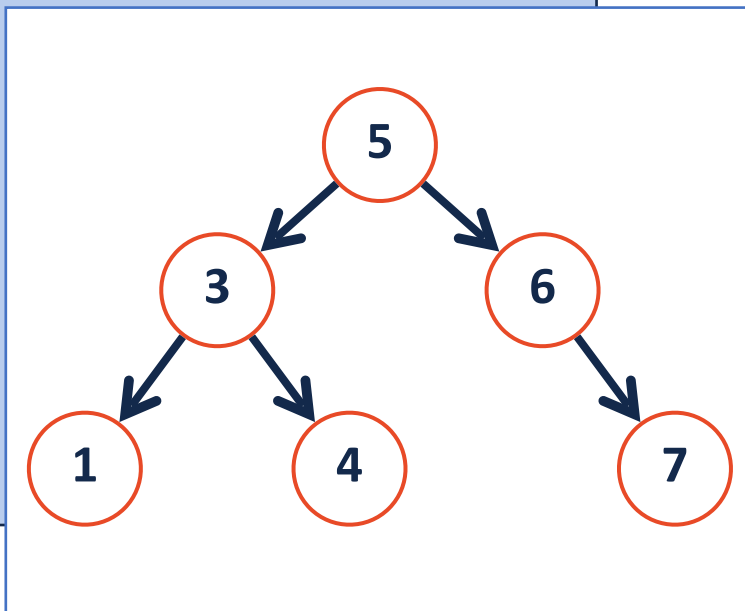
Find the insert location using `_find()`

Trivially insert at location



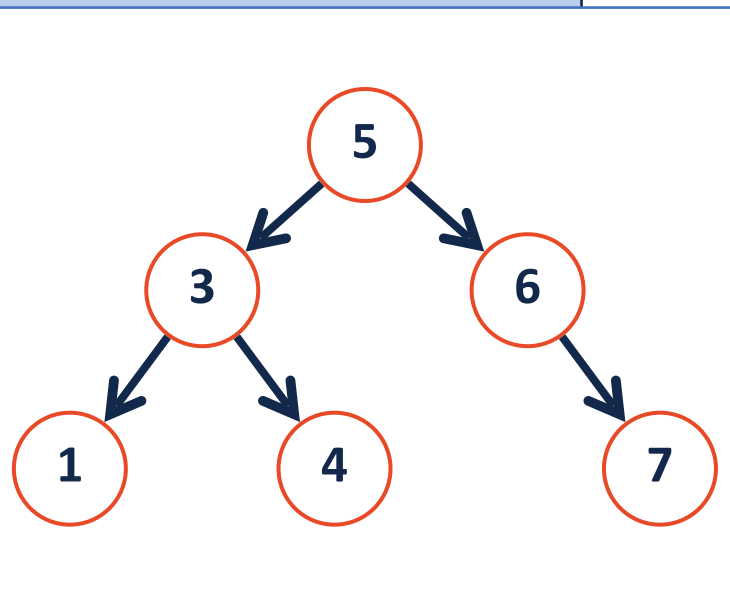
```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5
6
7
8
9
10
11
12
13
14
15
16 }
```



```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5  TreeNode *& tmp = _find(root, key);
6
7
8  tmp = new treeNode(key, val);
9
10
11
12
13 }
14
15
16
```



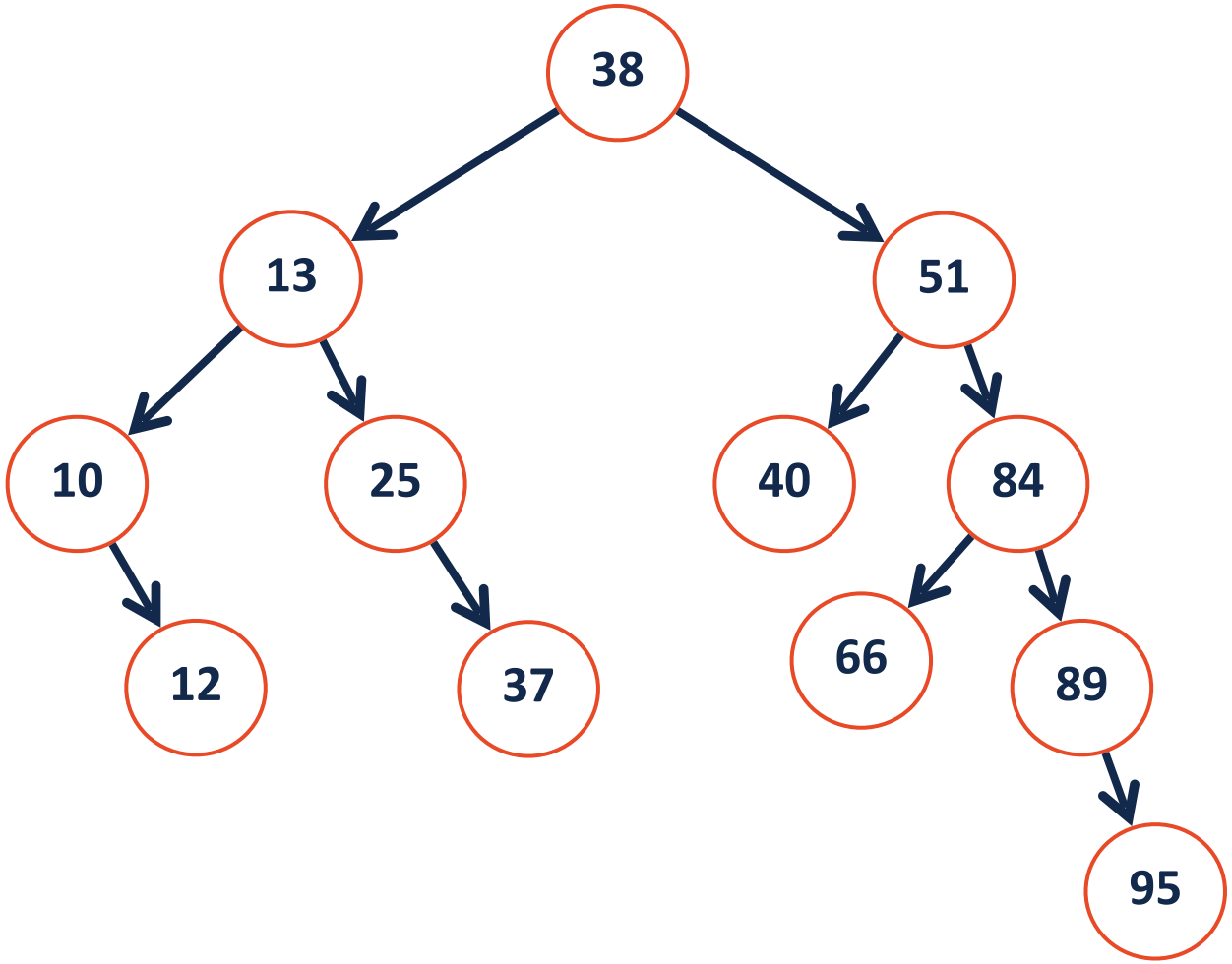
BST Insert

What binary tree would be formed by inserting the following sequence of integers: `[3, 7, 2, 1, 4, 8, 0]`

BST Remove

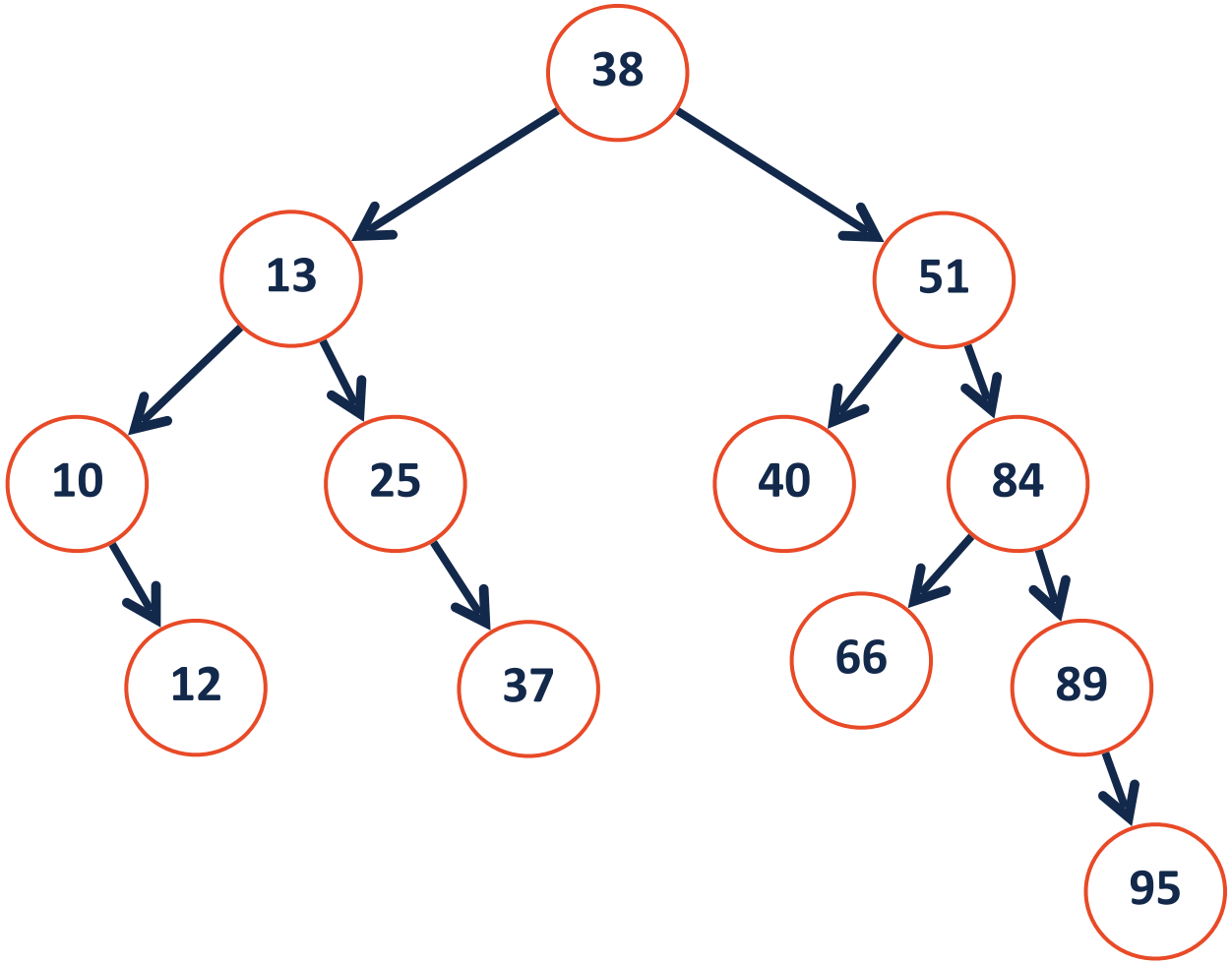
What should our tree look like after the following…

```
remove(40)
remove(25)
remove(51)
```



BST Remove

`remove(40)`

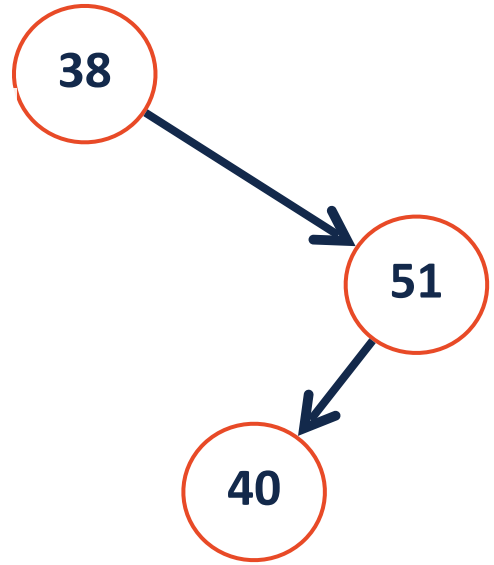


BST Remove

`remove(40)`

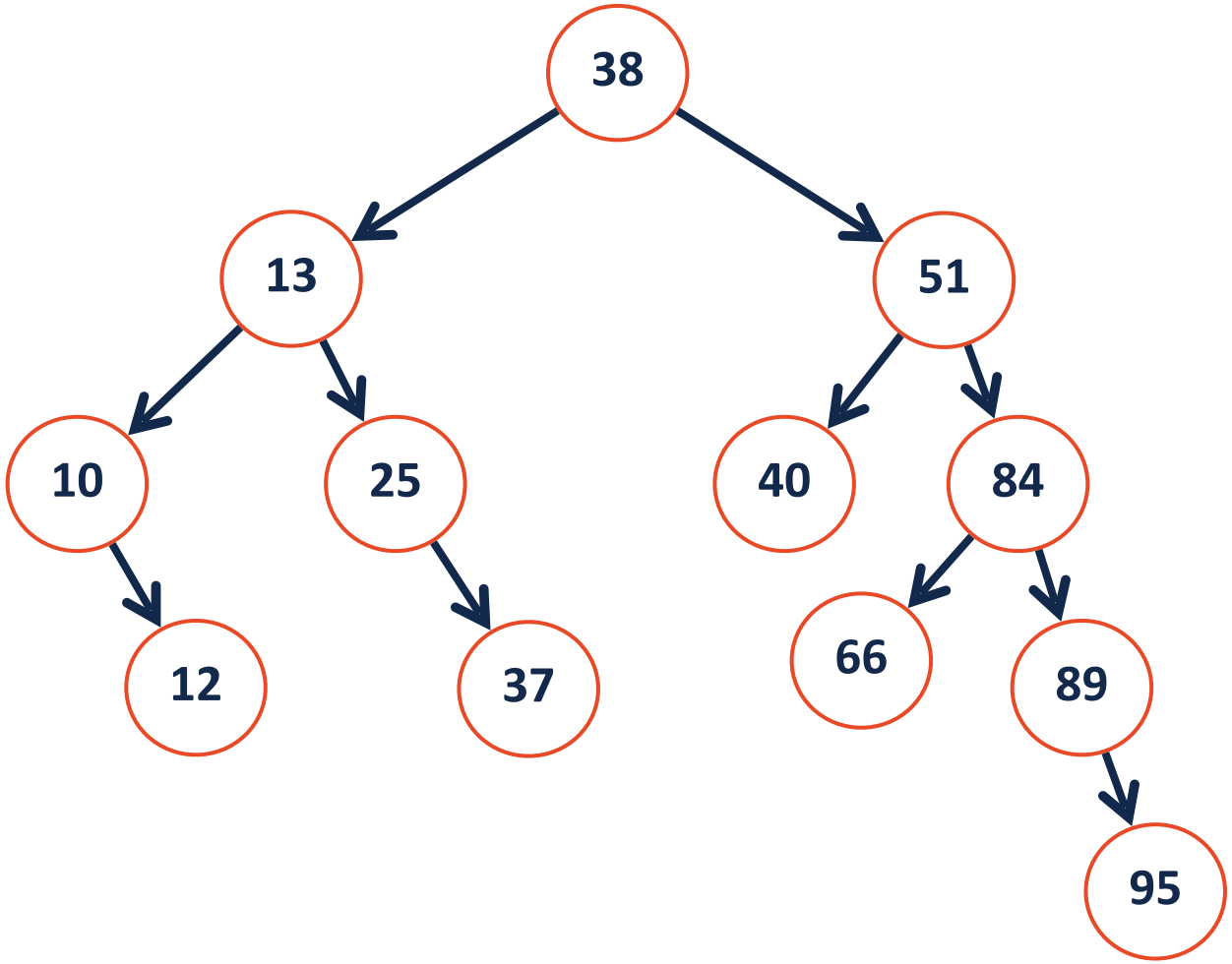
0-Child Case

```
TreeNode *& t = _find(root, 40);

delete t;

t = nullptr;
```



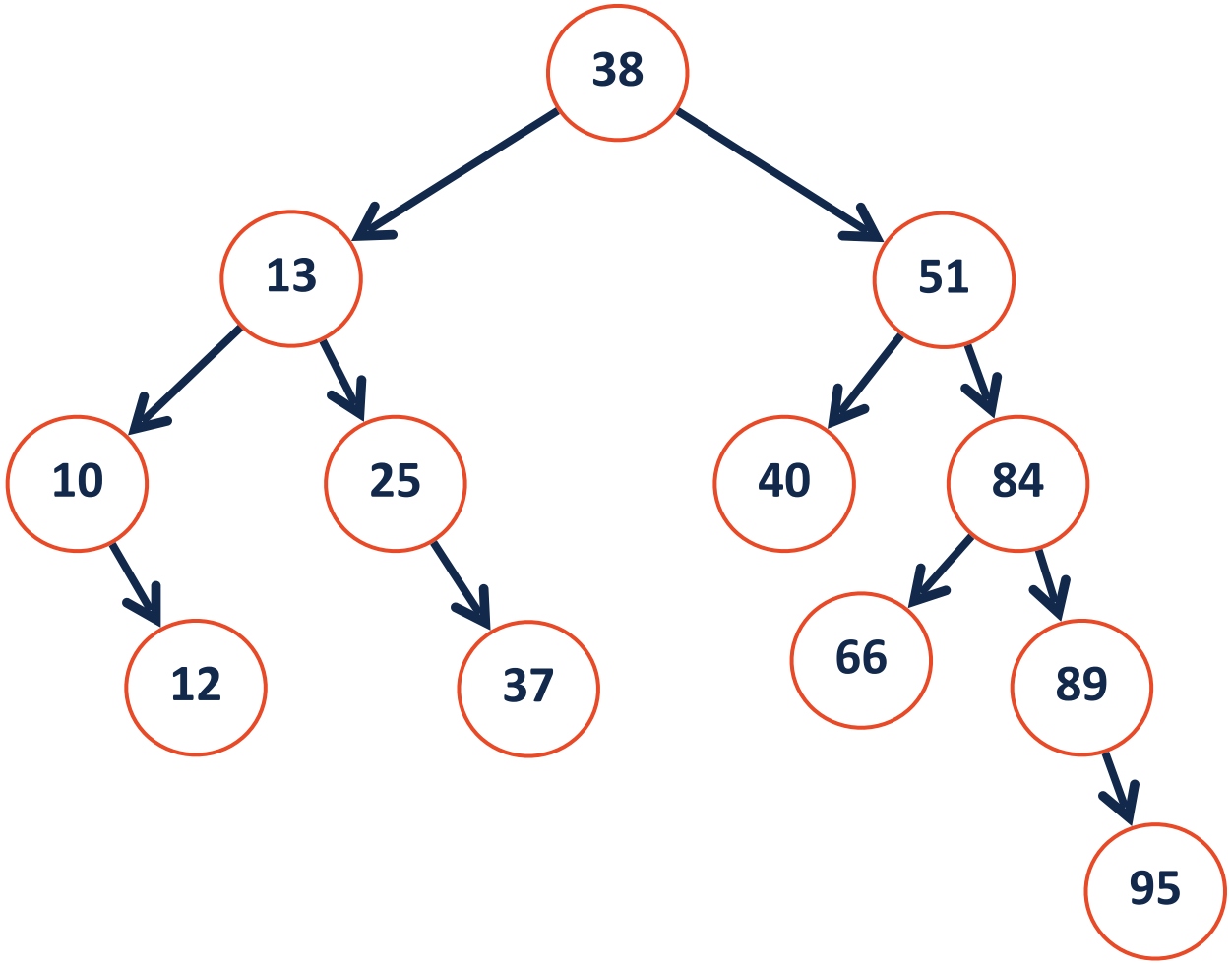
BST Remove

`remove(25)`



BST Remove

`remove(25)`

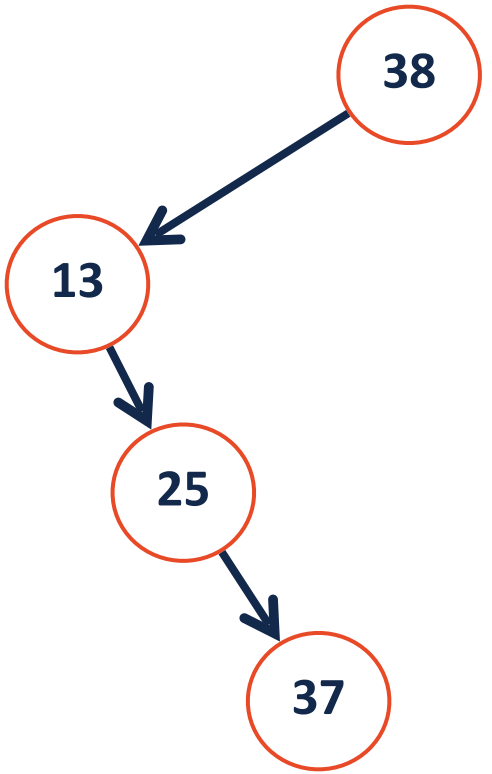


BST Remove

`remove(25)`

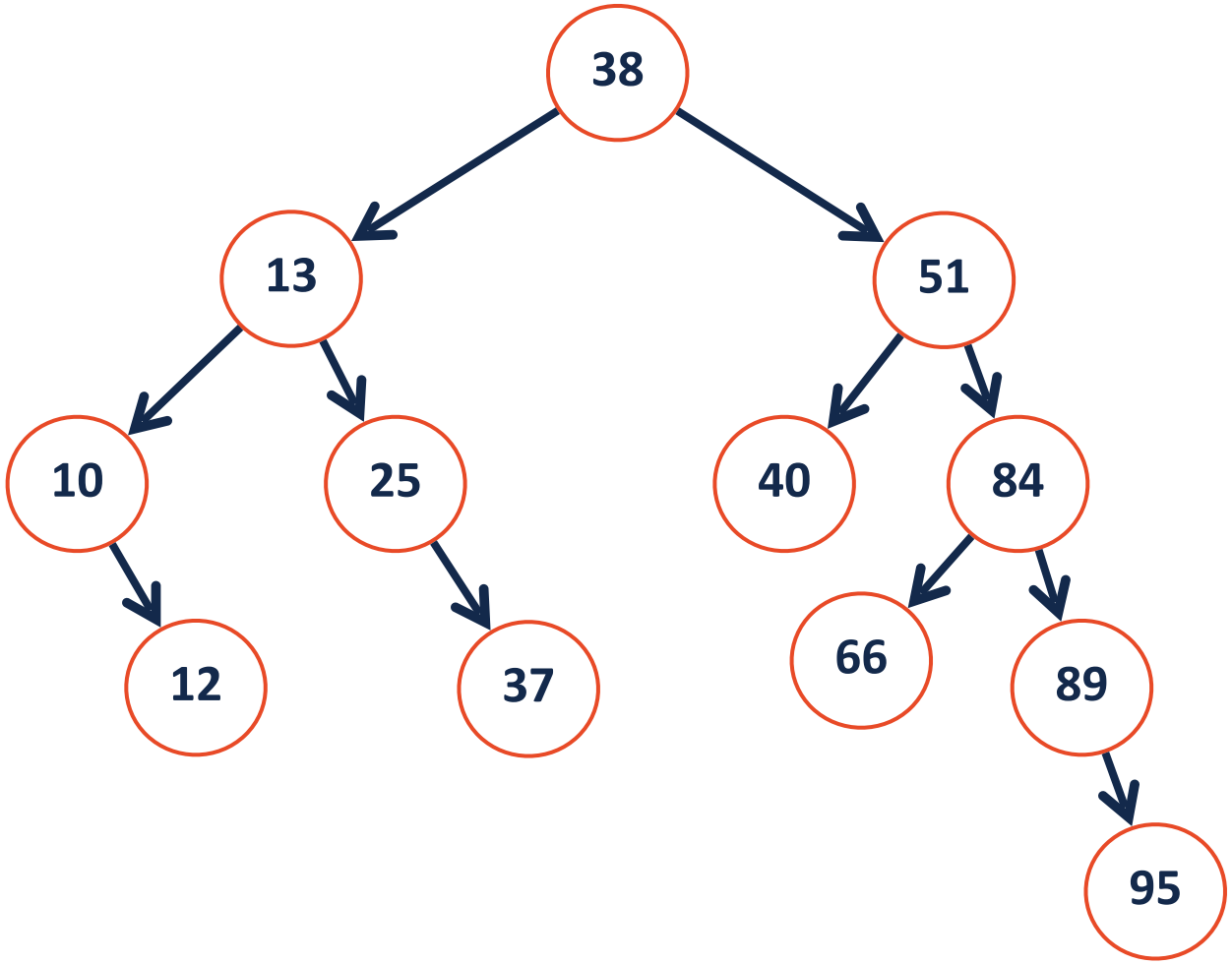
1-Child Case

```
TreeNode *& t = _find(root, 25);

TreeNode * tmp = t;

t = t->right;

delete tmp;
```



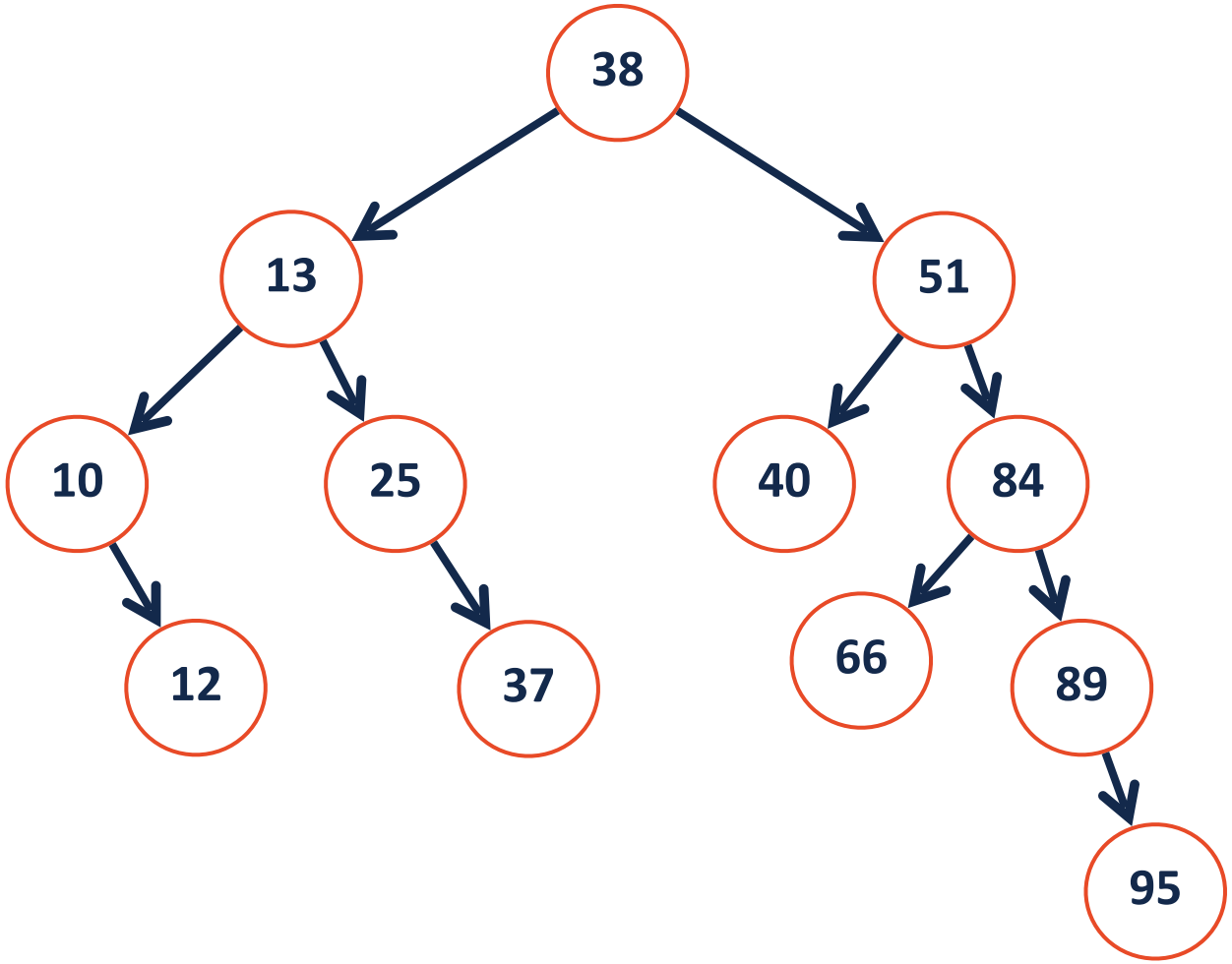
BST Remove

`remove(13)`



BST Remove

`remove(13)`



BST In-Order _____

In-Order Predecessor

Rightmost left child

IOP(38) =

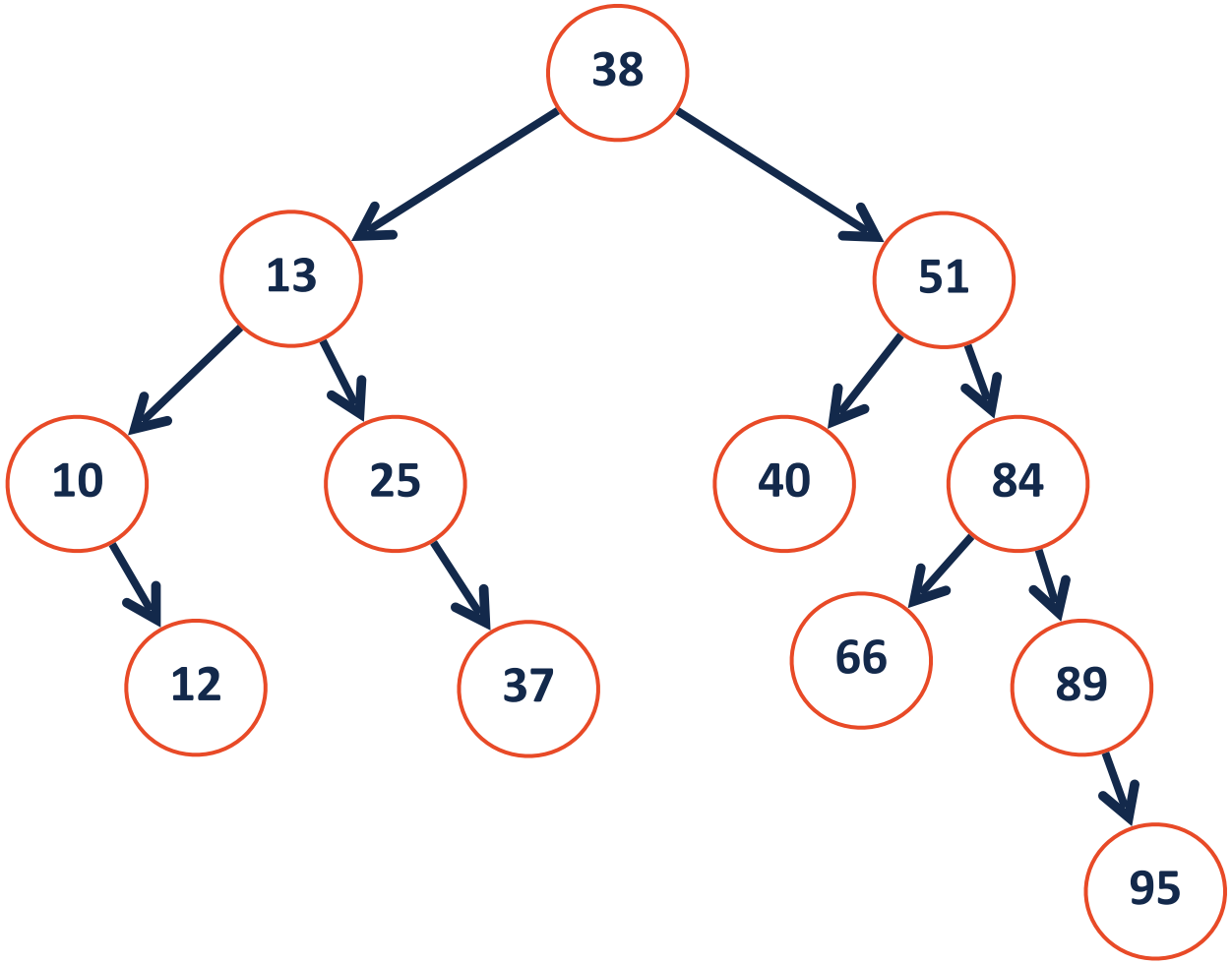
IOP(84) =

In-Order Successor

Leftmost right child

IOS(38) =

IOS(84) =



BST Remove

`remove(13)`

2-Child Case

```
TreeNode *& t = _find(root, 13);

TreeNode * IOP = getIOP(t);

swap(t, iop);

remove(13); //starting from t
```