Data Structures

Binary Search Trees

CS 225
Harsha Tirumala

September 22, 2025

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

n

1. find(66) Lack of structure

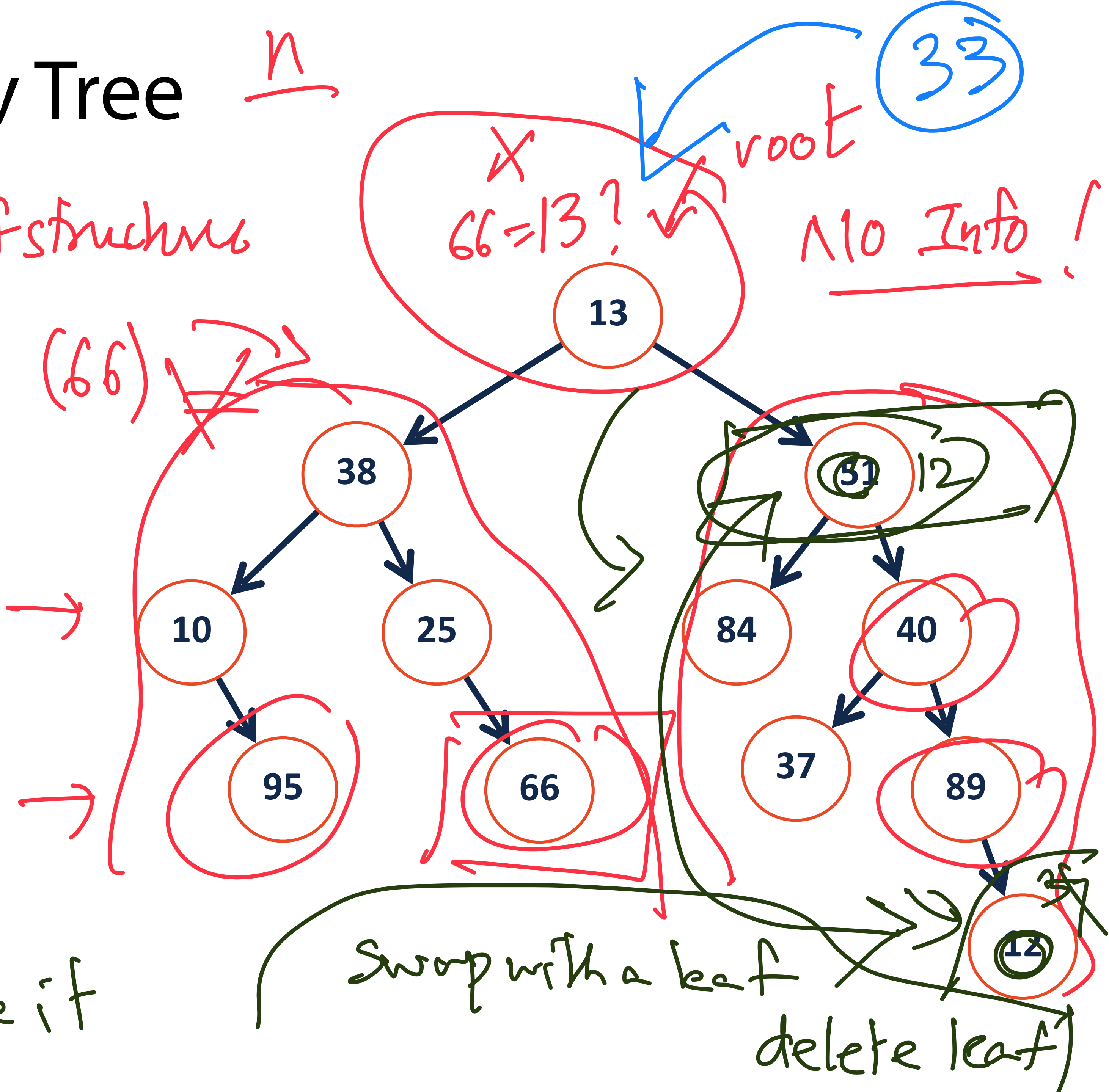
O(n)

2. Insert(33)

O(1)

3. Remove(51)

Find(51) + Remove it



Tree Search Tradeoffs

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

~~Chaos~~ Order

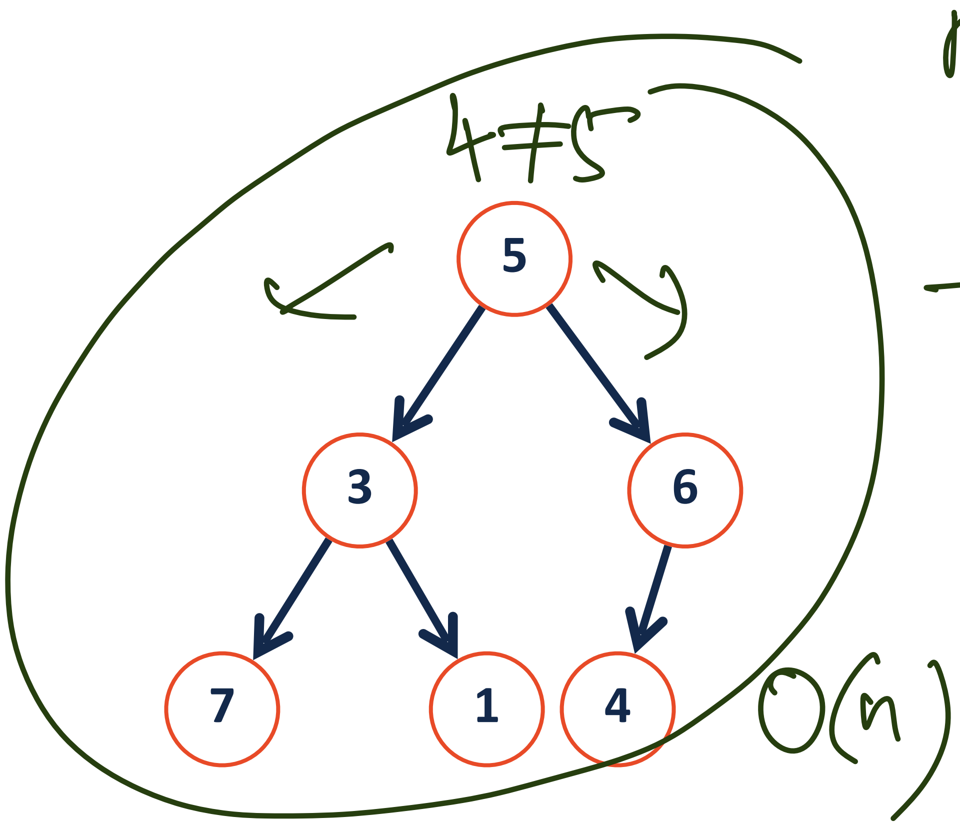
What do we trade in order to do so?

Insert ↑

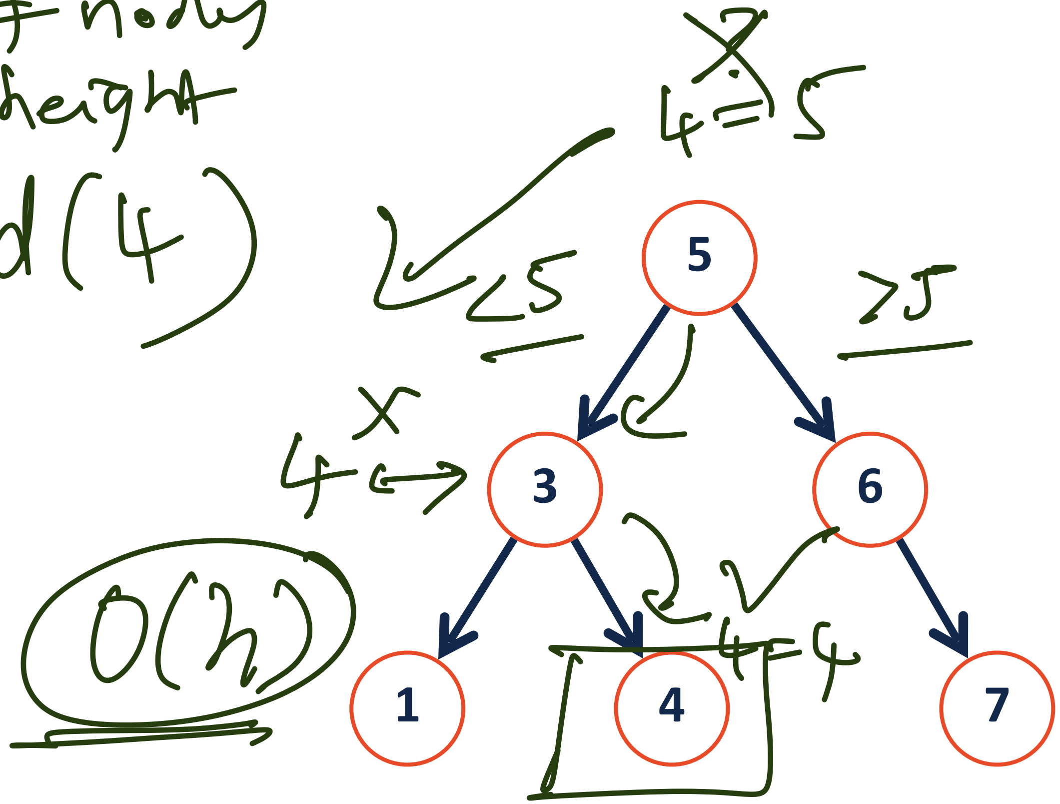
Improved Search - Unstructured vs Structured BT

5	3	6	7	1	4

1	3	4	5	6	7



n : # nodes
h : height
find(4)



Dictionary ADT

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

Key Values

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)

Unique

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

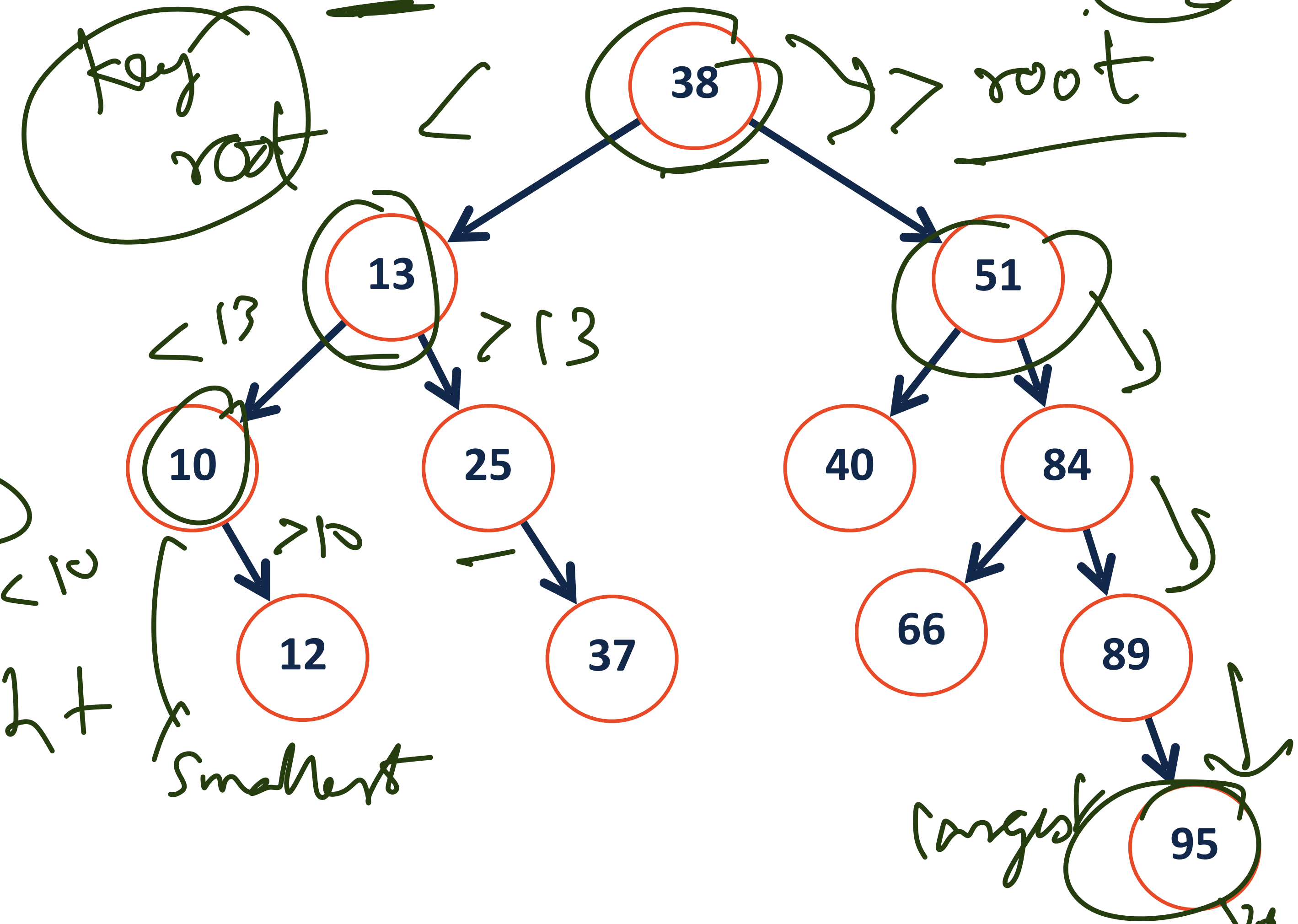
Recursive

root

left right

#node

left < root < right



BST.h

```
#pragma once

template <typename K, typename V>
class BST {
  public:
   /* ... */
  private:
    class TreeNode {
          K & key;
          V & value;

         TreeNode *left, *right;

          TreeNode(K & k, V & v)  :
key(k), value(v),
left(NULL), right(NULL) { }
    };

    TreeNode *root_;
    /* ... */
};
```

Tree.h

```
#pragma once

template <typename T>
class BinaryTree {
  public:
      /* ... */
  private:
      class TreeNode {
        T & data;

        TreeNode * left;

        TreeNode * right;

        TreeNode(T & data) :
         data(data), left(NULL),
right(NULL) { }

      };

      TreeNode *root_;
      /* ... */
};
```

Binary Search Tree ADT

requirements

Insert — insert but maintaining structure

Remove — remove but maintain structure

Traverse — pretty much same as Binary Trees.

Find —

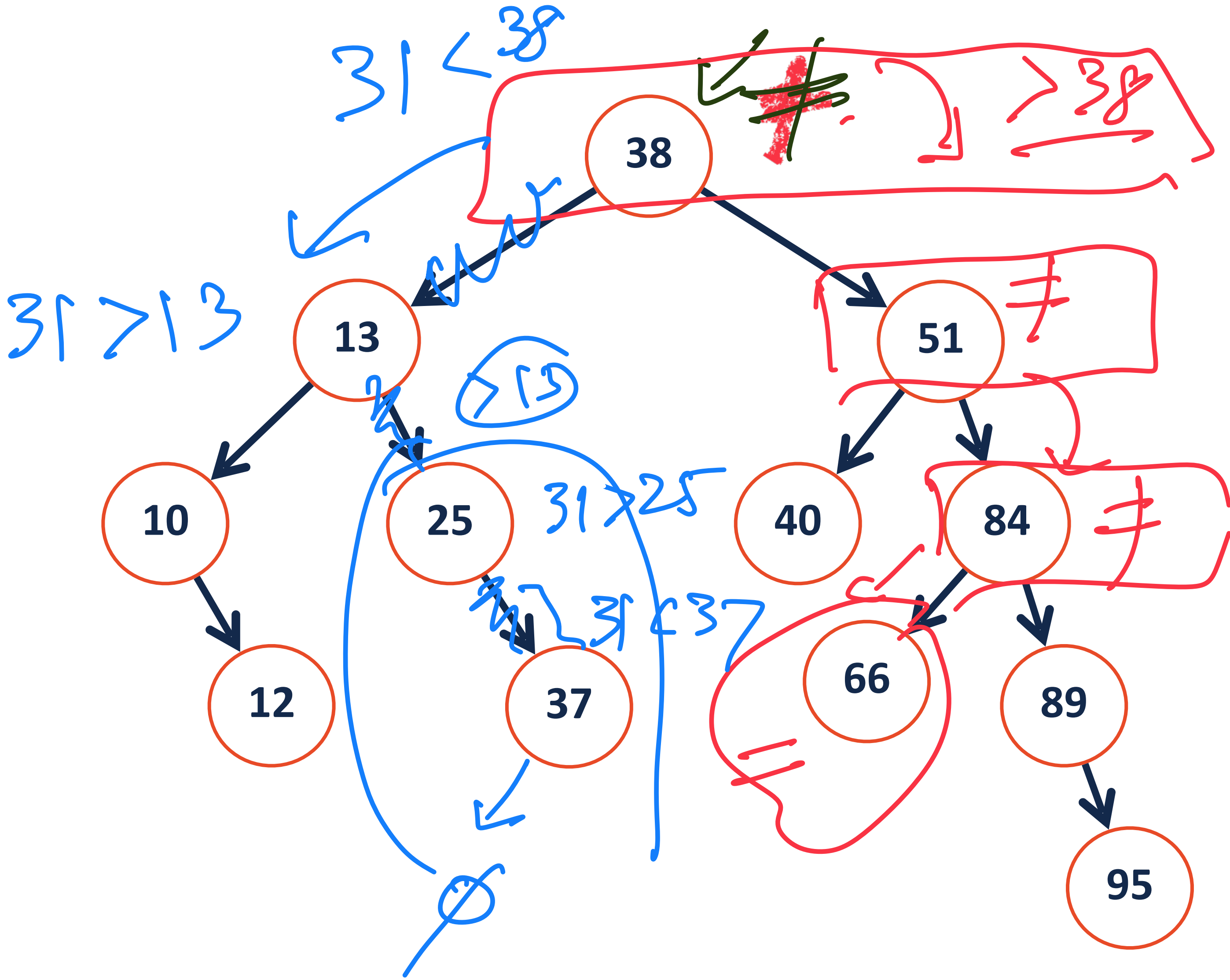
Constructor —

BST Find

find(66)

find(31)

31 not find



BST Find

find(66)

A recursive function based around value of root:

66 > 38

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

Let `tmp = root->key()`

`tmp == query`, return root

66

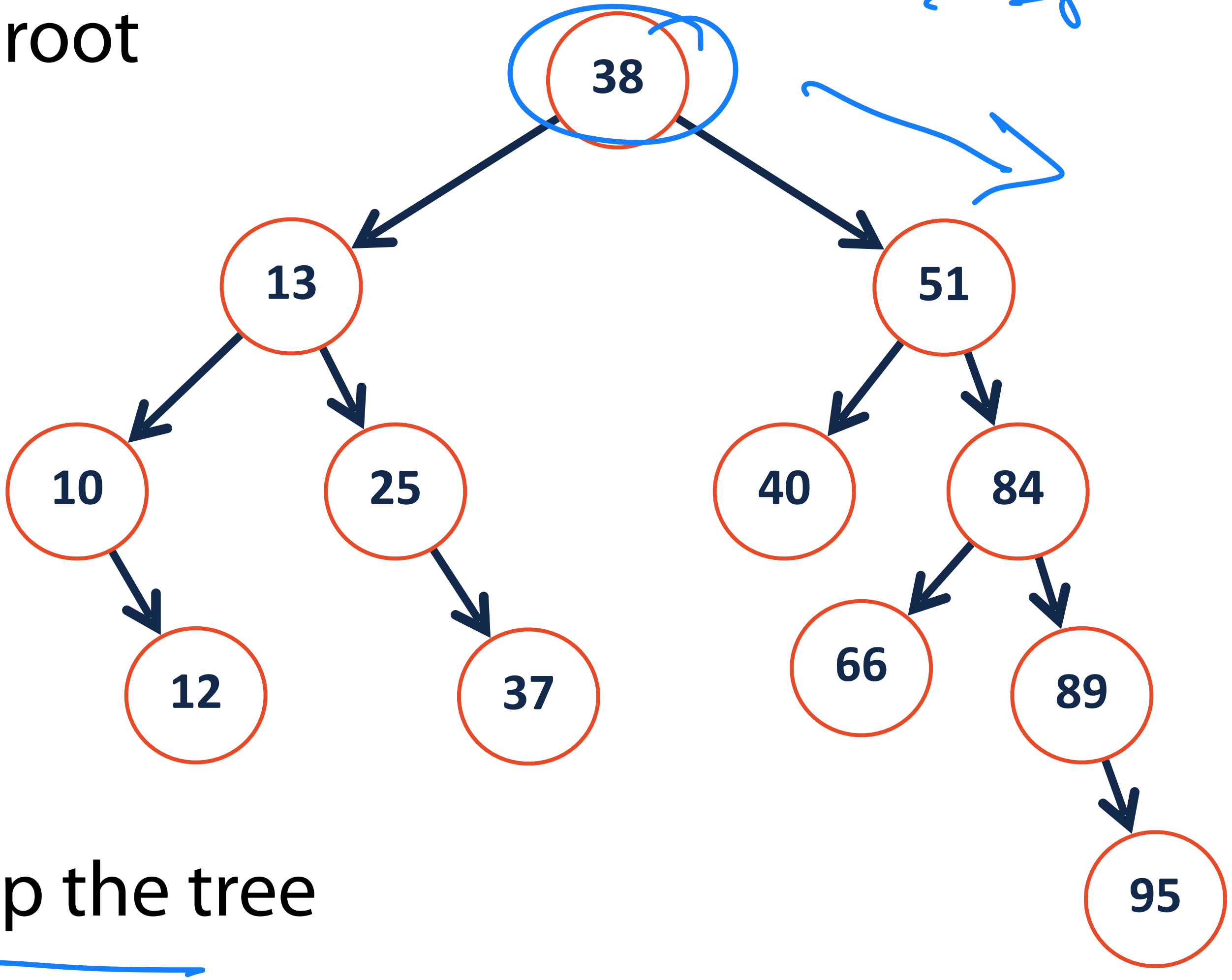
Recursion:

`tmp < query`, recurse right

`tmp > query`, recurse left

Combining:

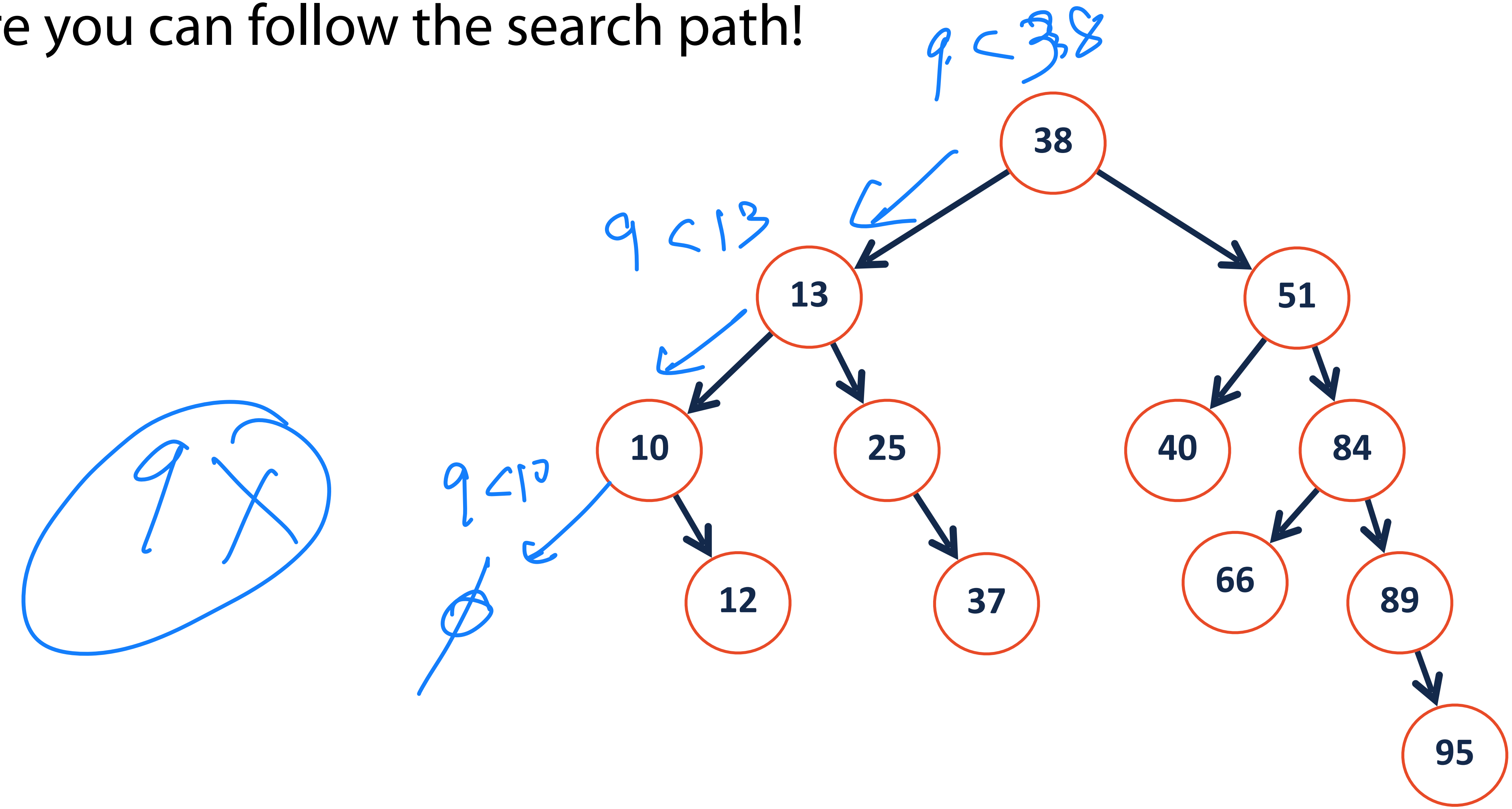
Return the recursive value back up the tree



BST Find

`find(9)`

Make sure you can follow the search path!



```
template<typename K, typename V>

_______TreeNode *&__________ find(TreeNode *& root, const K & key) {


// Base Case
if(root == nullptr || root->data == key){
    return root;
}

// Recursive Step ("Combining step" is 'return')
if (root->data > key){
    return _find(root->left, key);
}

return _find(root->right, key);


}
```

can't manipulate the ordering

tmp > query

h: O(n)

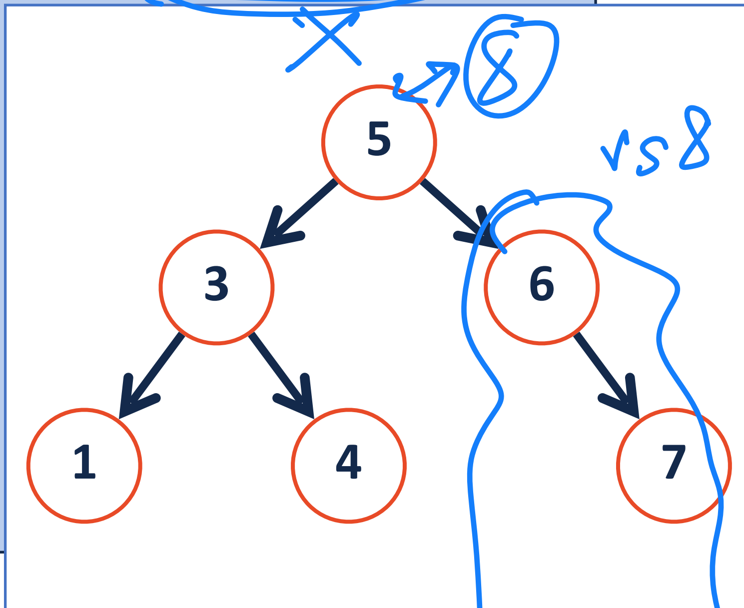
tmp < query

#nodes n

height: h

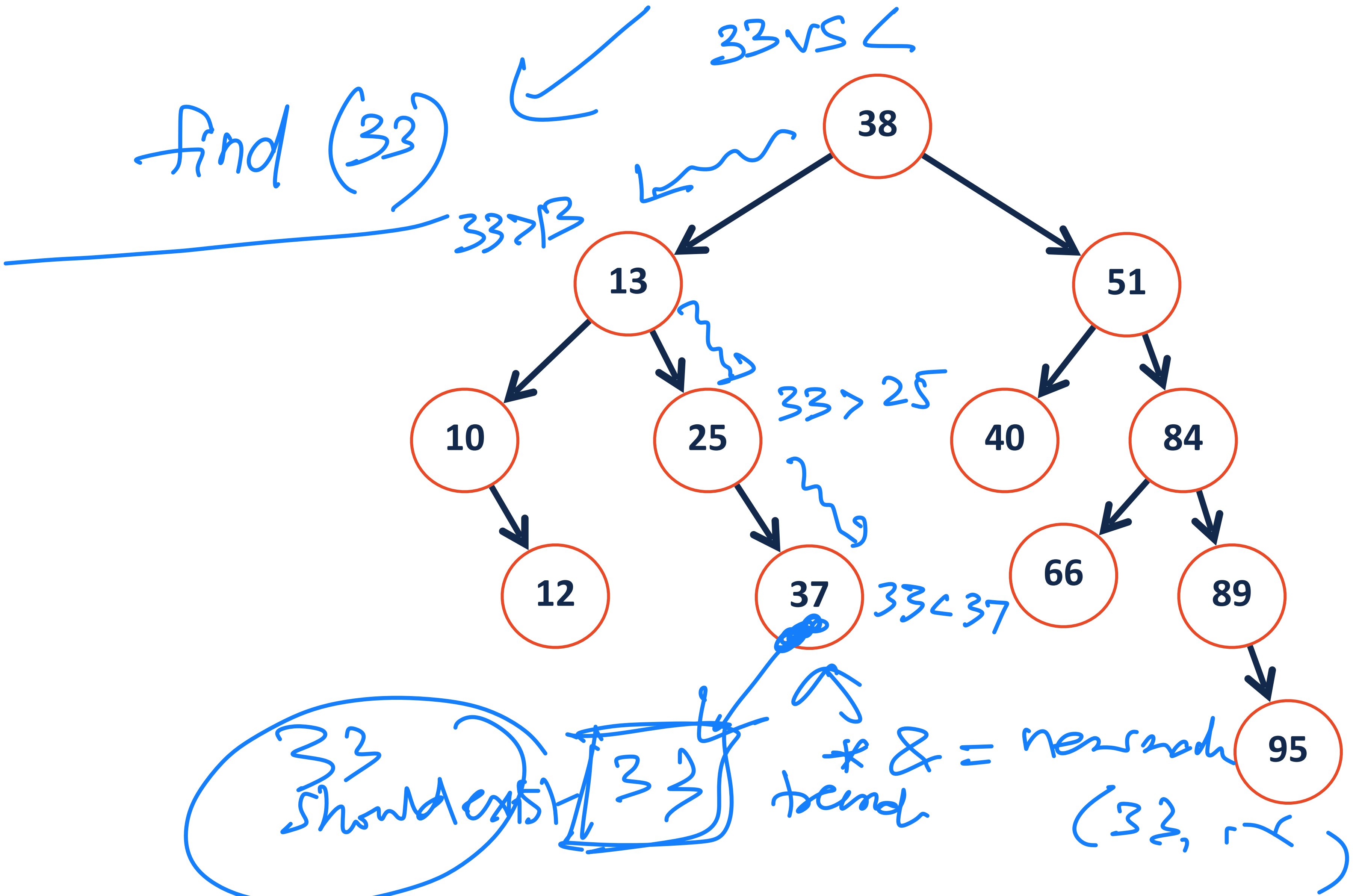
Time: O(h) <=> O(n)

BST BT



BST Insert

insert(33, v)

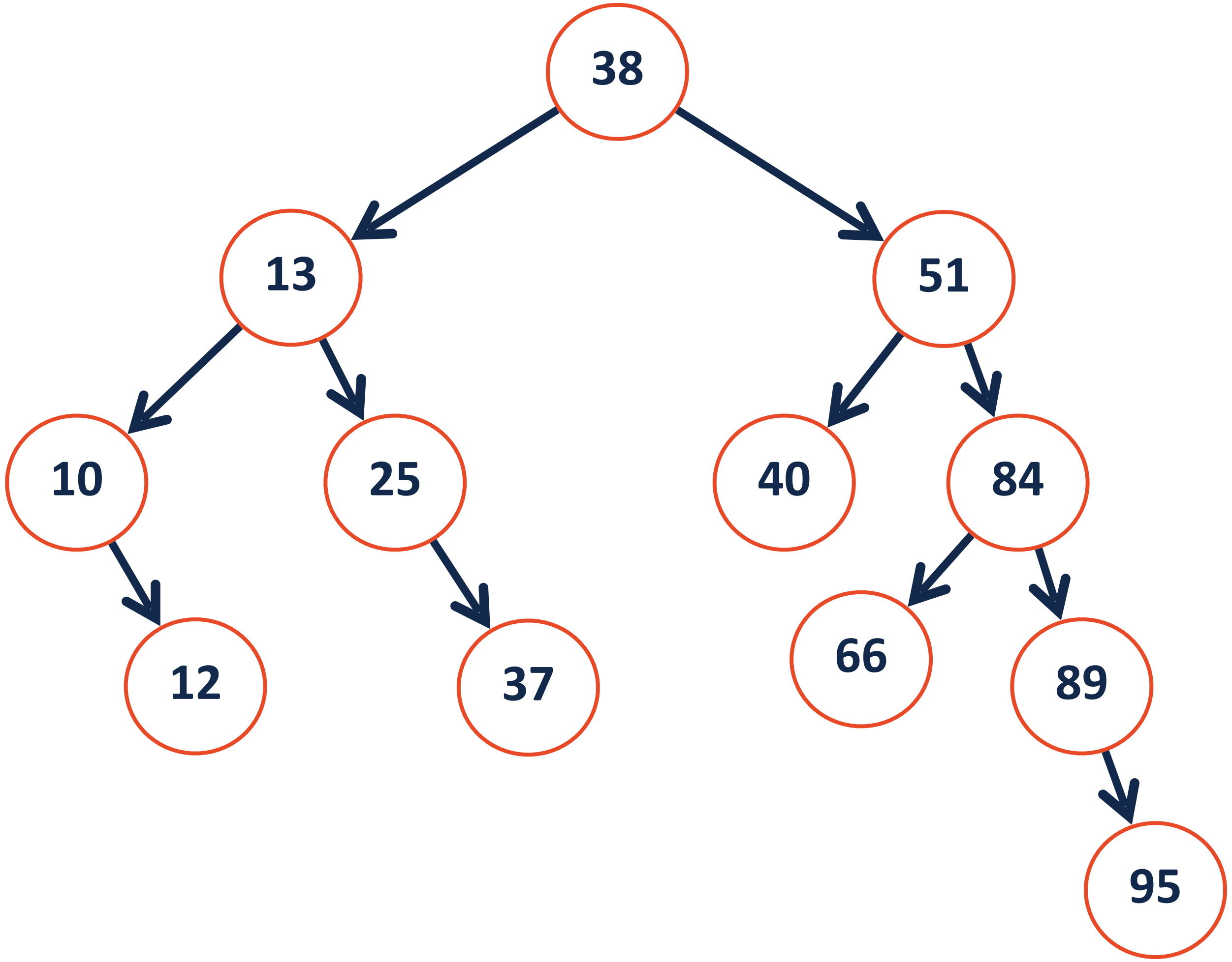


BST Insert

`insert(33, v)`

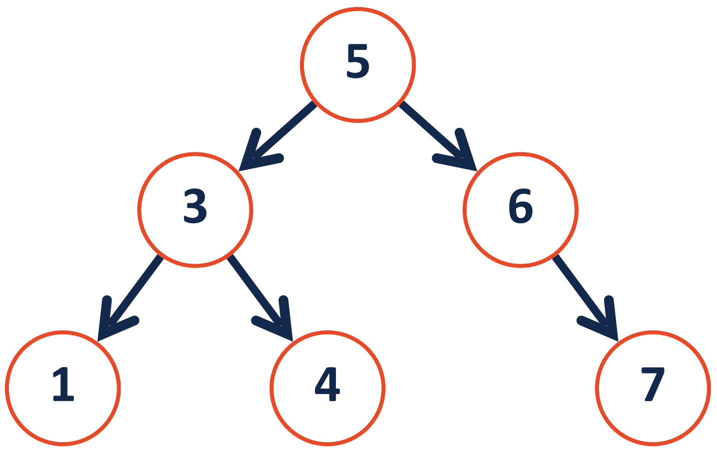
Find the insert location using `_find()`

Trivially insert at location



```
template<typename K, typename V>

void _insert(const K & key, const V & val) {

    return _insert(root, key, val);
}

```

```
template<typename K, typename V>

void _insert(TreeNode *& root, const K & key, const V & val) {












}
```



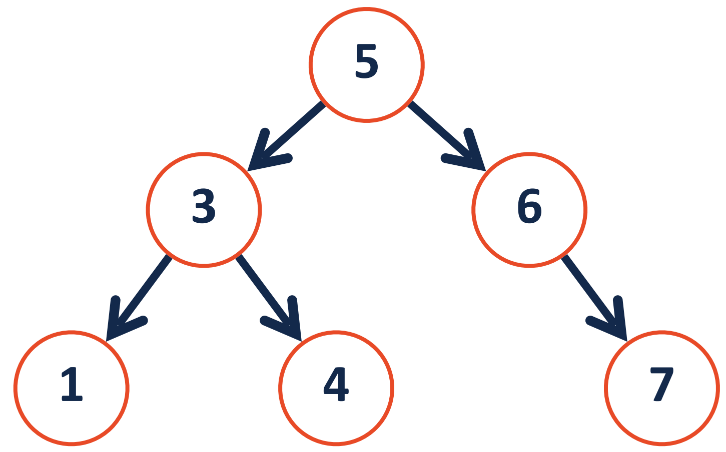
```
template<typename K, typename V>

void _insert(const K & key, const V & val) {

    return _insert(root, key, val);
}
```

Tree

```
template<typename K, typename V>

void _insert(TreeNode *& root, const K & key, const V & val) {

TreeNode *& tmp = _find(root, key);


tmp = new treeNode(key, val);




}
```

exact location to insert

new node



BST Insert

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

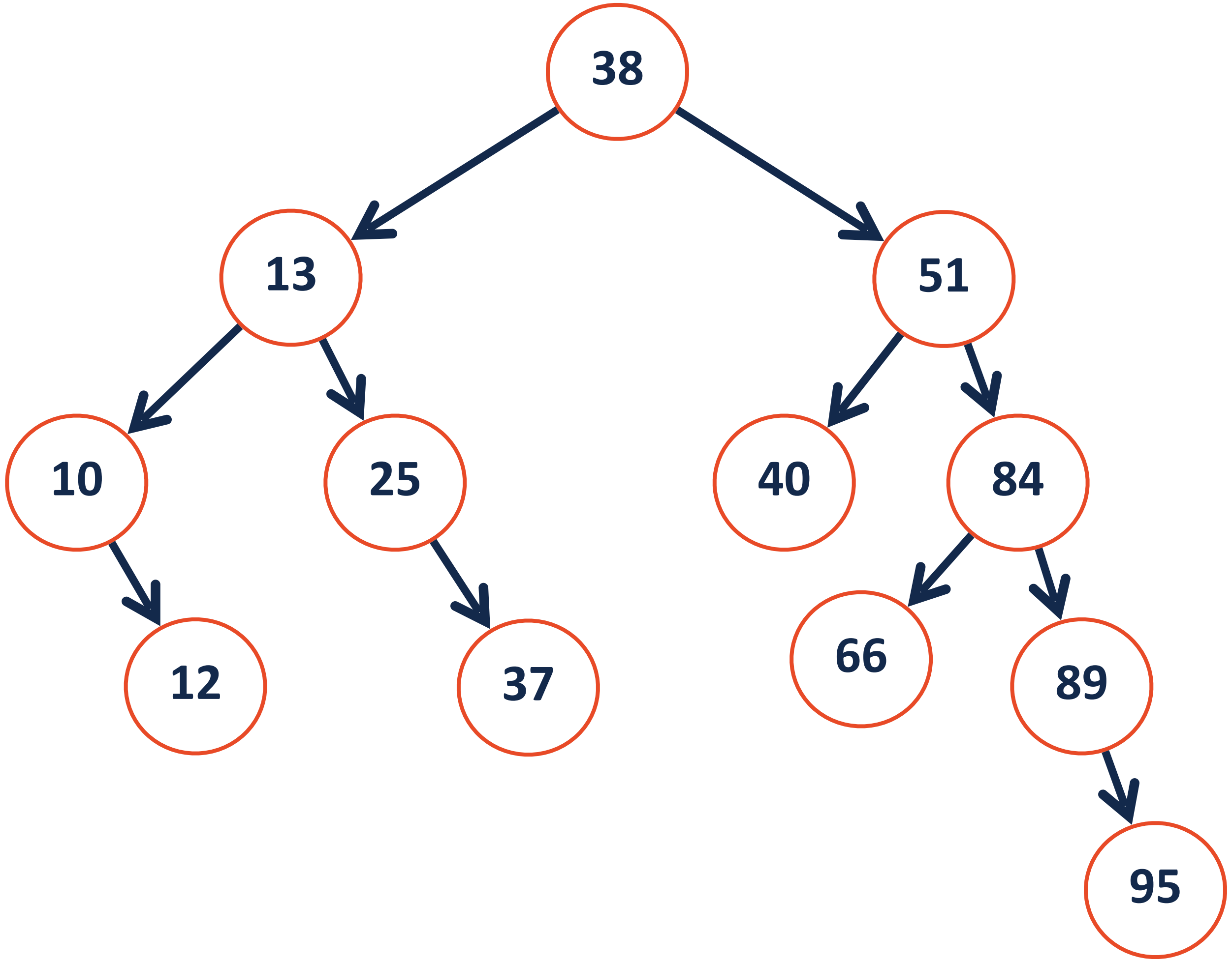
BST Remove

What should our tree look like after the following…

`remove(40)`

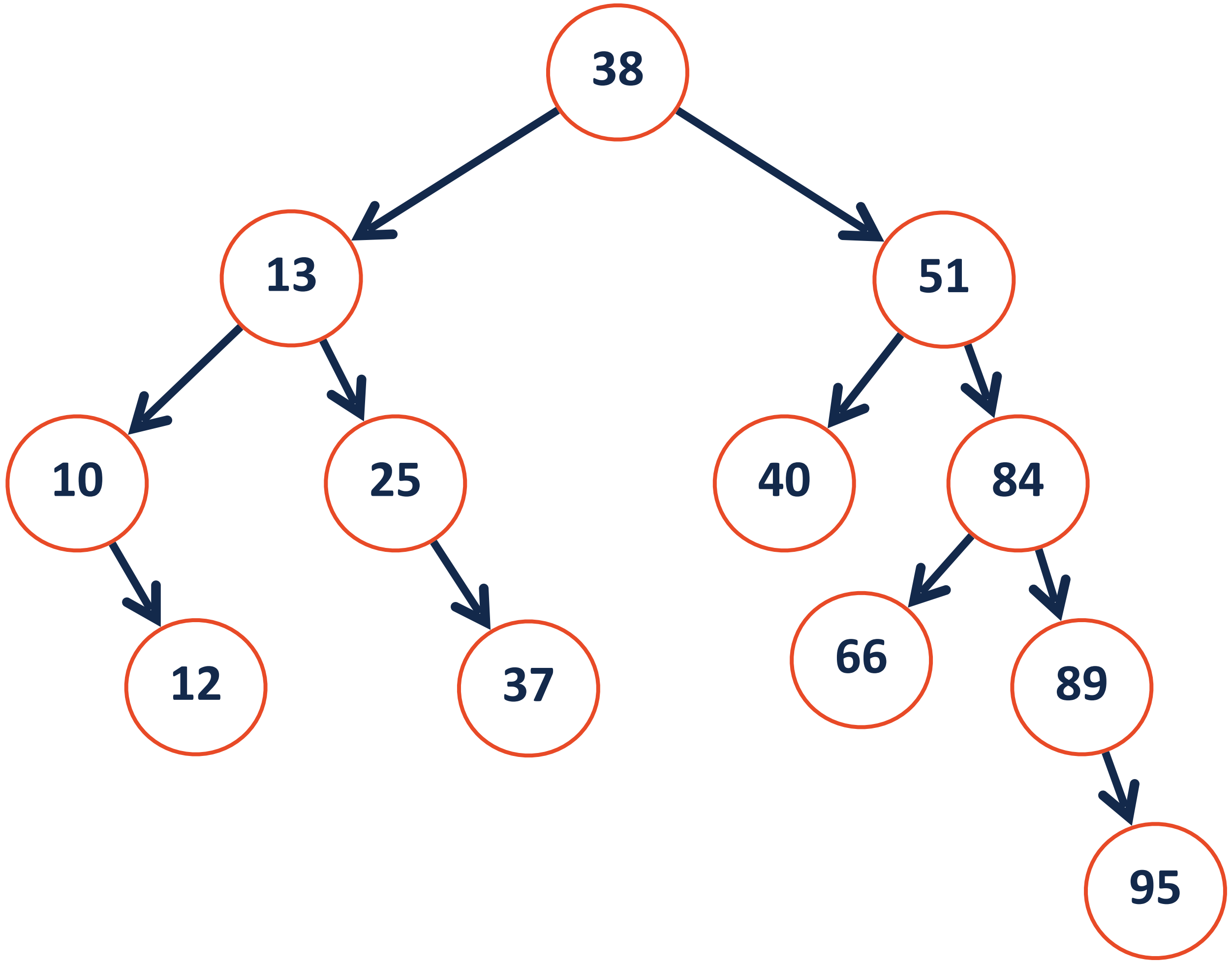
`remove(25)`

`remove(51)`



BST Remove

`remove(40)`

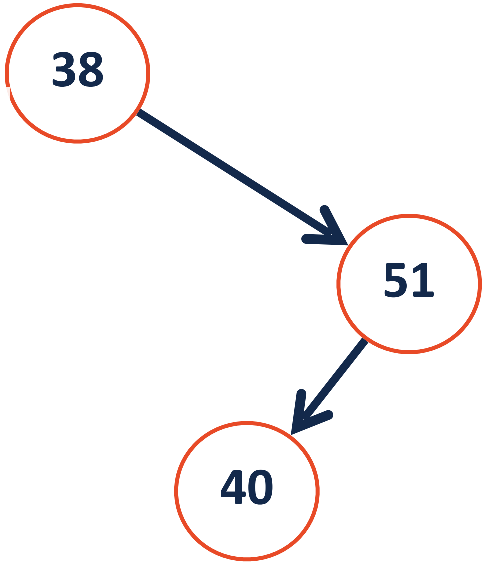


BST Remove

`remove(40)`

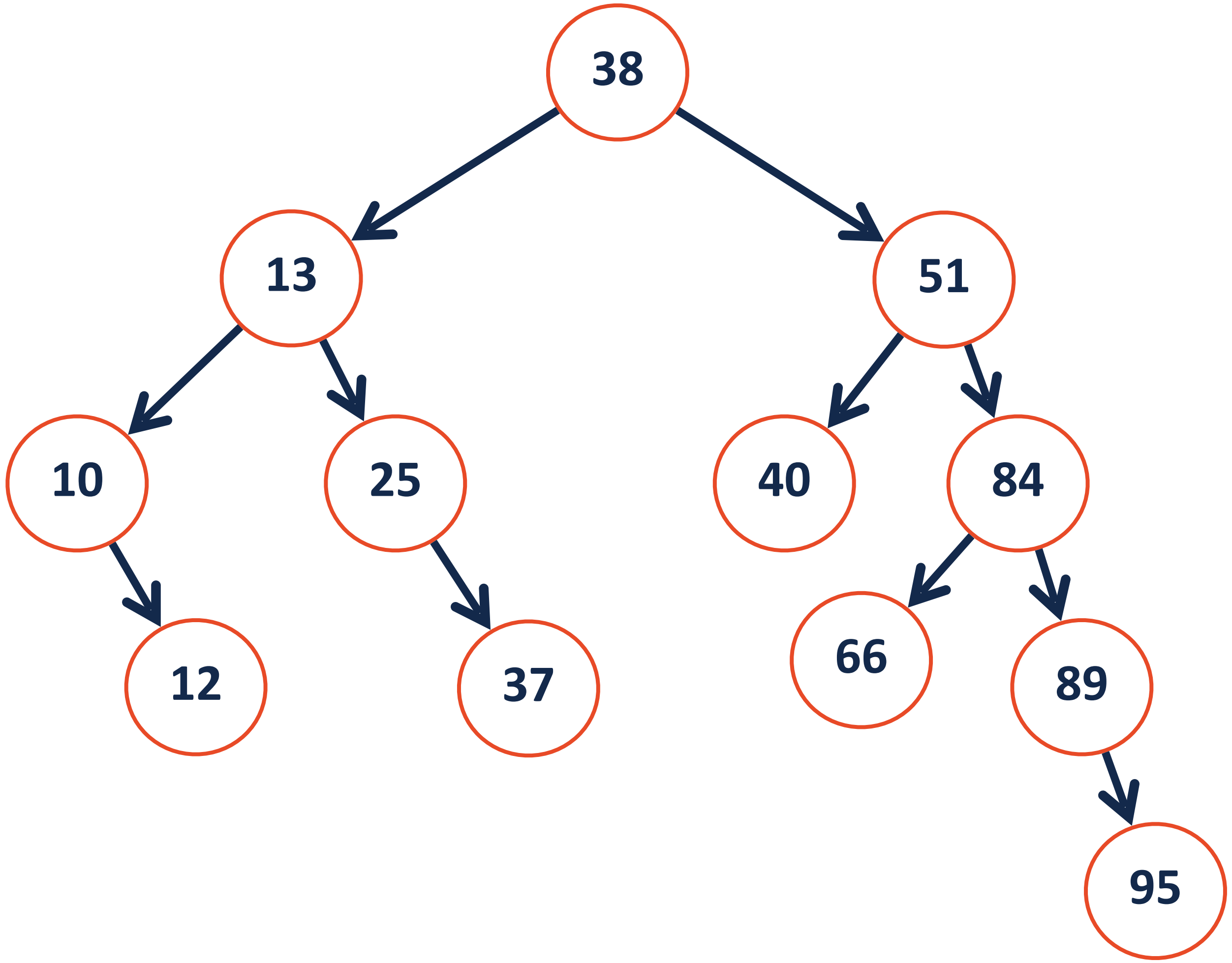
0-Child Case

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



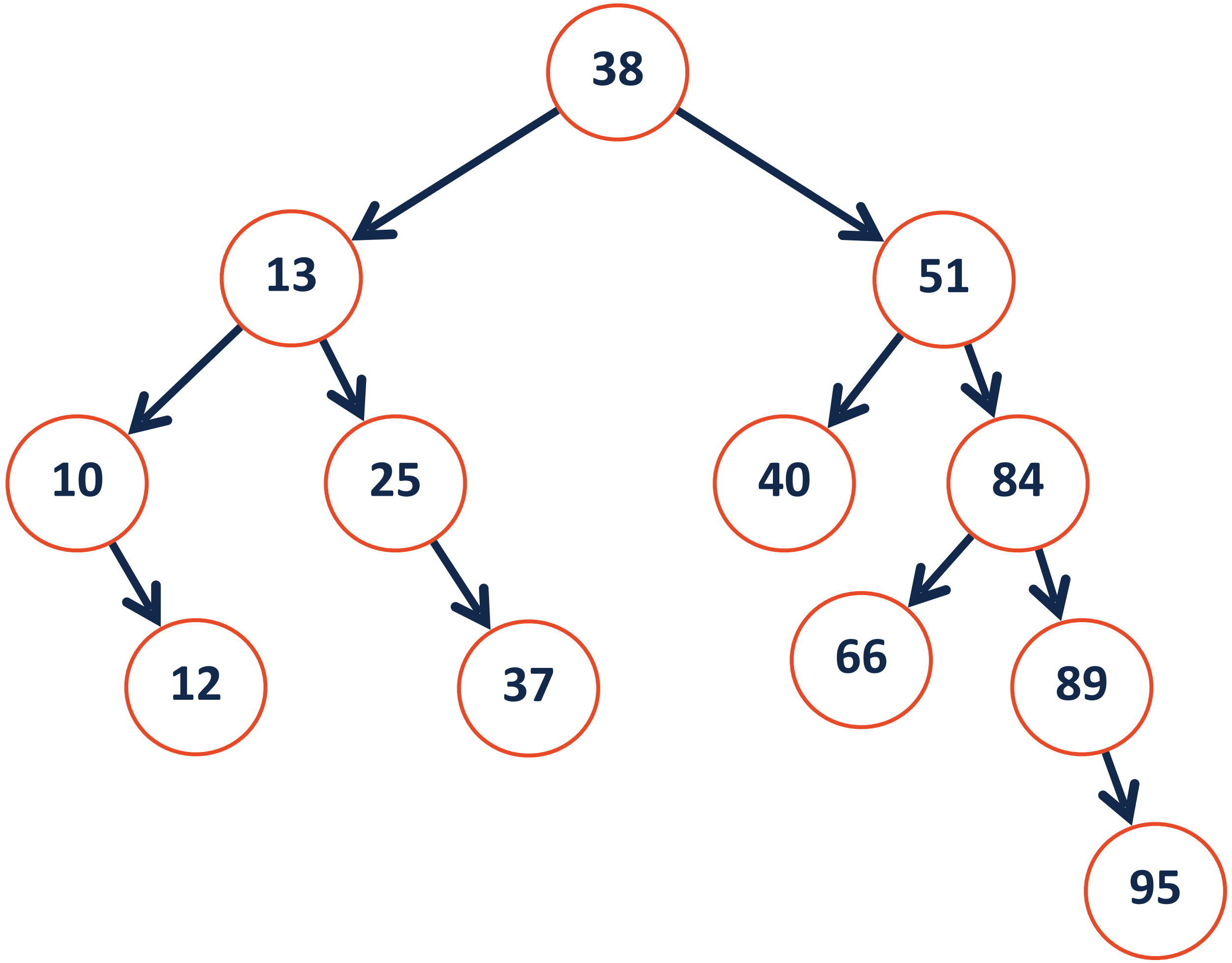
BST Remove

`remove(25)`



BST Remove

`remove(25)`

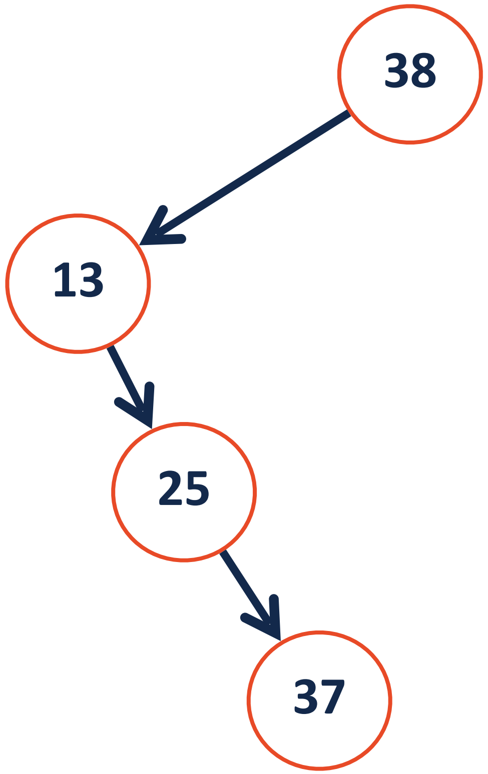


BST Remove

remove(25)

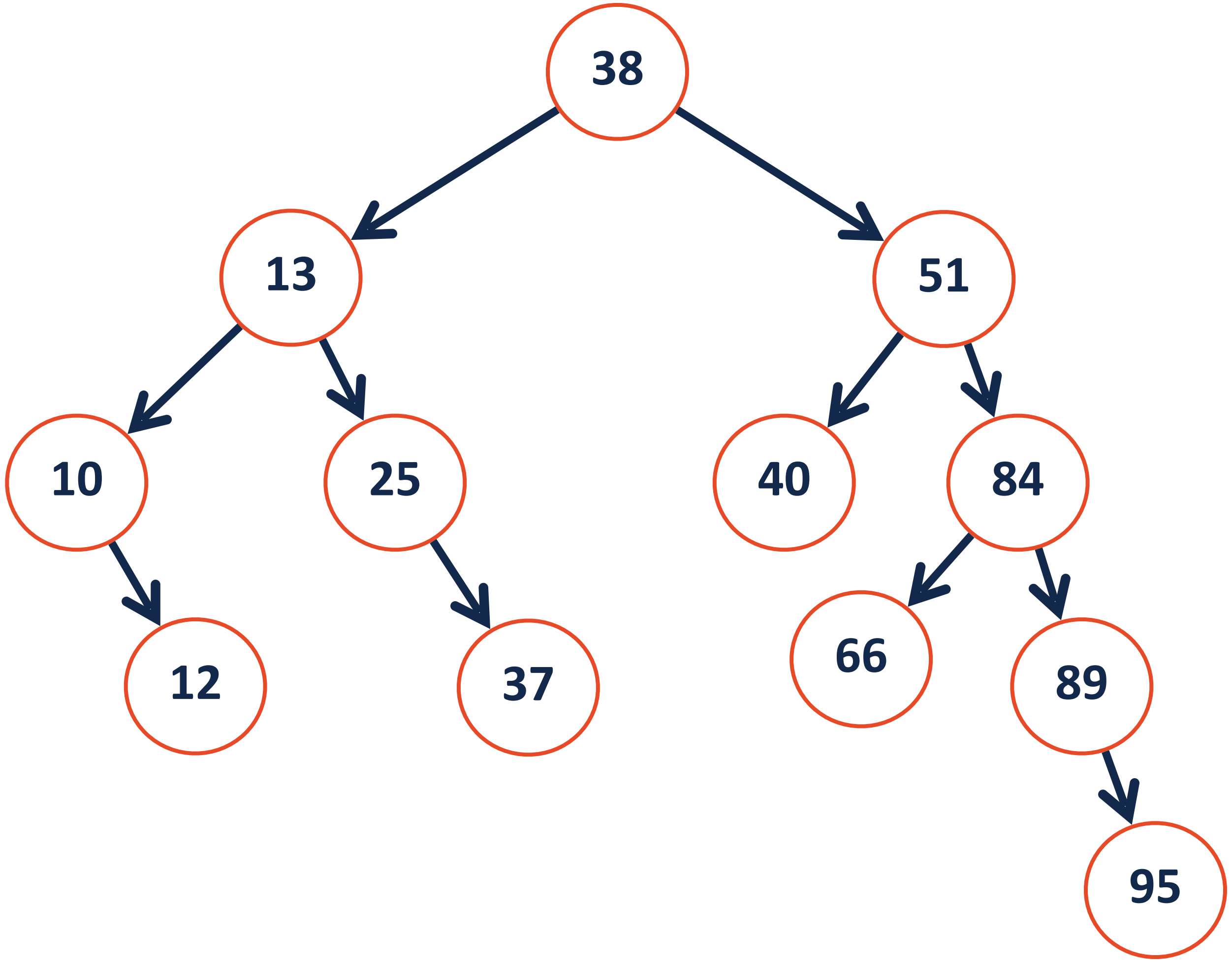
1-Child Case

```
TreeNode *& t = _find(root, 25);
TreeNode * tmp = t;
t = t->right;
delete tmp;
```



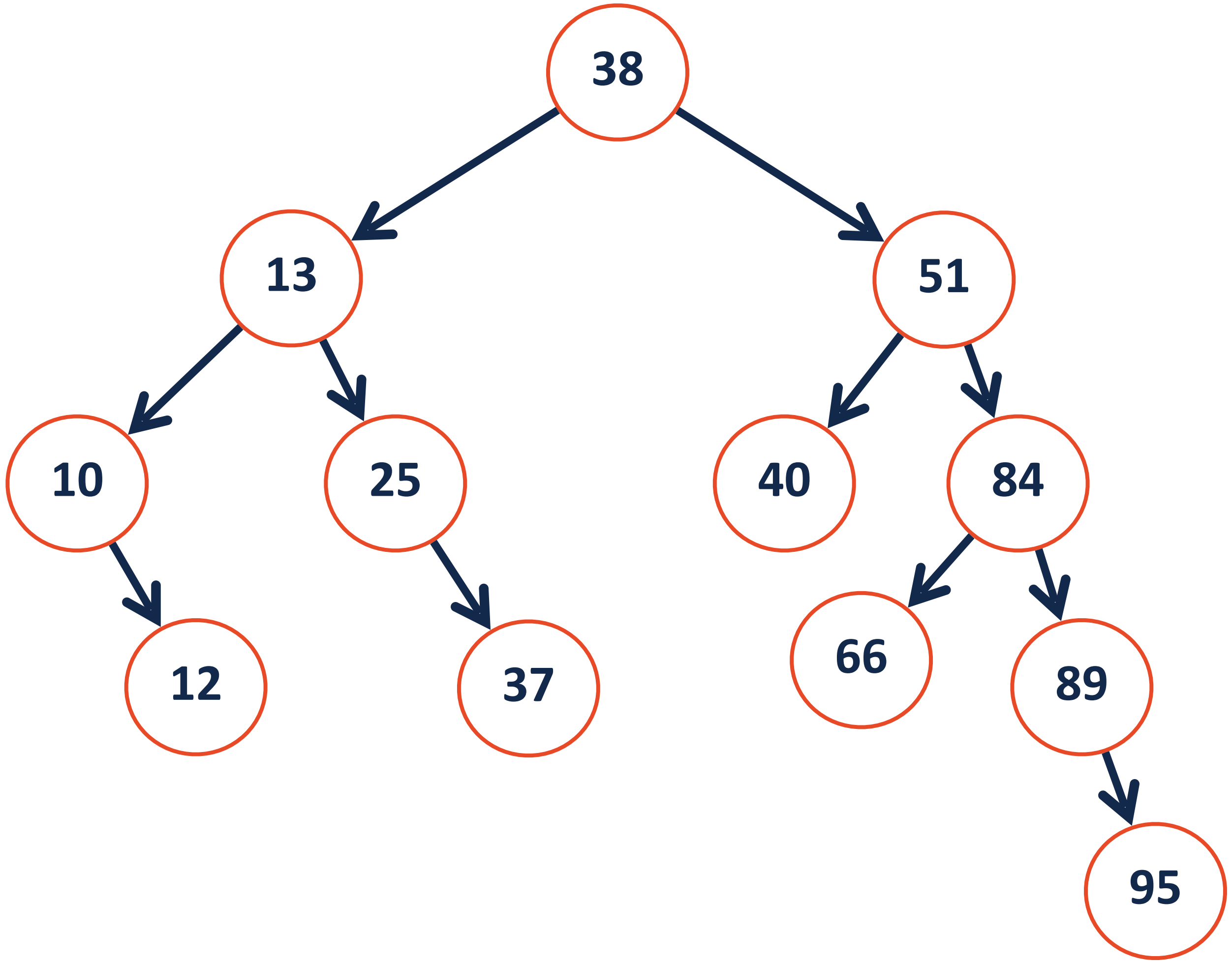
BST Remove

`remove(13)`



BST Remove

`remove(13)`



BST In-Order ________

In-Order Predecessor

Rightmost left child

IOP(38) =

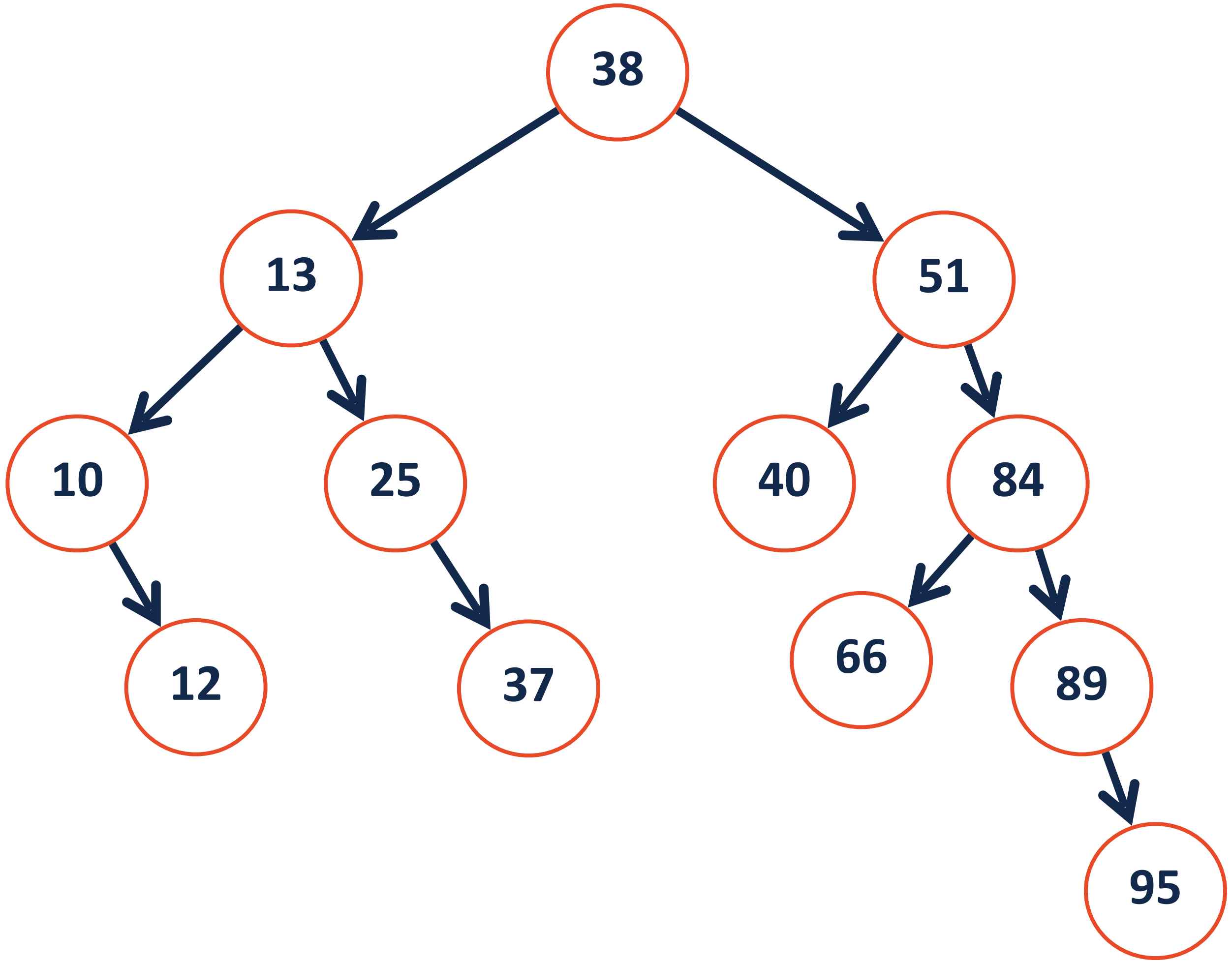
IOP(84) =

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
```