

Data Structures

Tree Search & BST

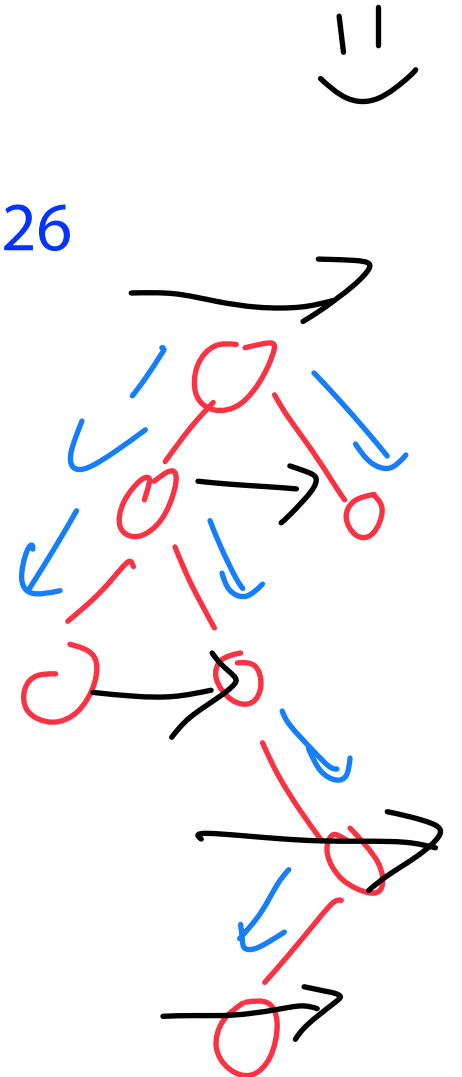
CS 225
Brad Solomon

September 18, 2026



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



HACK TO THE FUTURE

PRESENTED BY PRODUCT SPACE UIUC
STARRING YOU



APPLY NOW

PRODUCTSPACEHACKATHON.COM

OCTOBER 2-3

📍 CAMPUS INSTRUCTIONAL FACILITY

ALL MAJORS. FREE FOOD. INDUSTRY JUDGES

📷 @PRODUCTSPACEUIUC

SPONSORS

Flagship Sponsor



Questions, comments or concerns? Email productspaceui@gmail.com

Learning Objectives

Review Tree Traversals

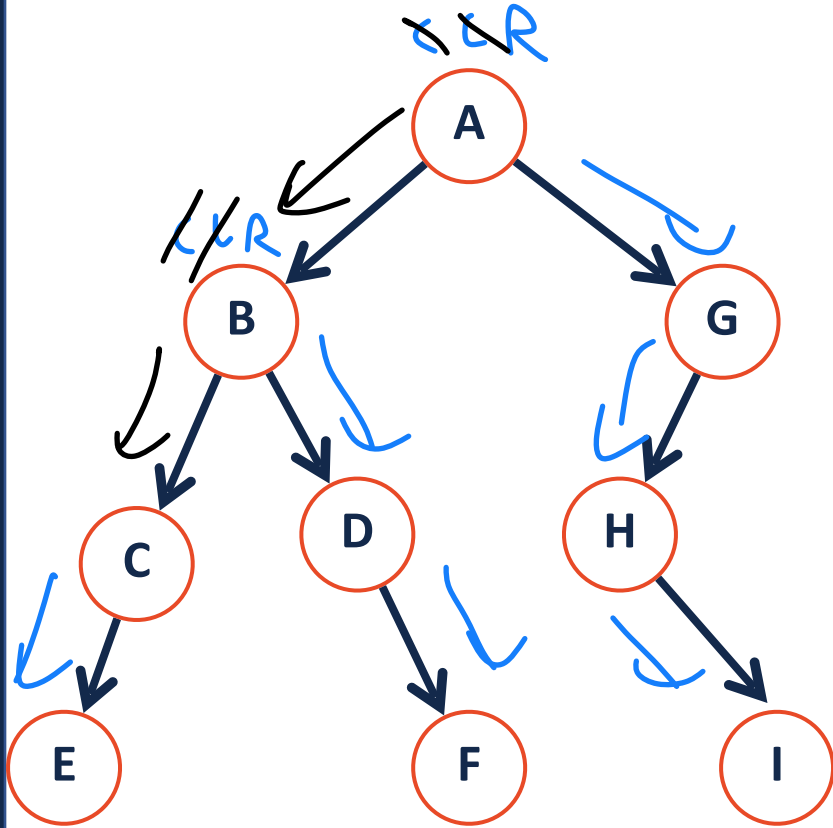
Search

Explore implementations of DFS and BFS on binary trees

Extend binary trees into binary *search* trees

Build conceptual and coding understanding of BST

Traversals - Pre Order



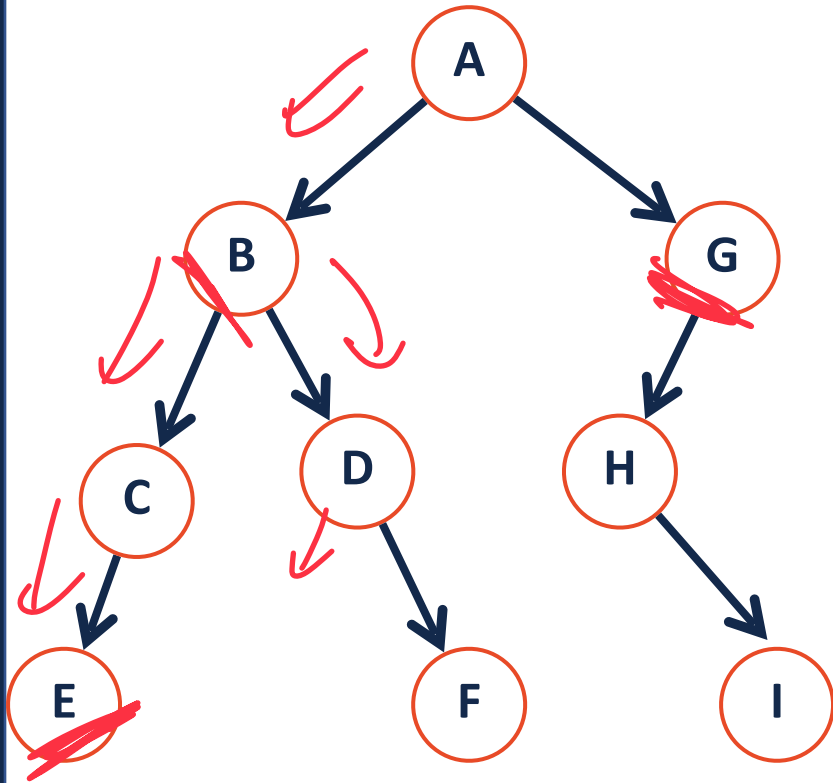
A B C E D F G H I

```
1 template<class T>
2 void BinaryTree<T>:: PreOrder(TreeNode * root)
3 {
4
5     if (root) {
6         print (root);
7
8         preOrder(root->left);
9
10
11         preOrder(root->right);
12
13
14     }
15
16
17
18
19
20
21 }
```

Join Code: 225



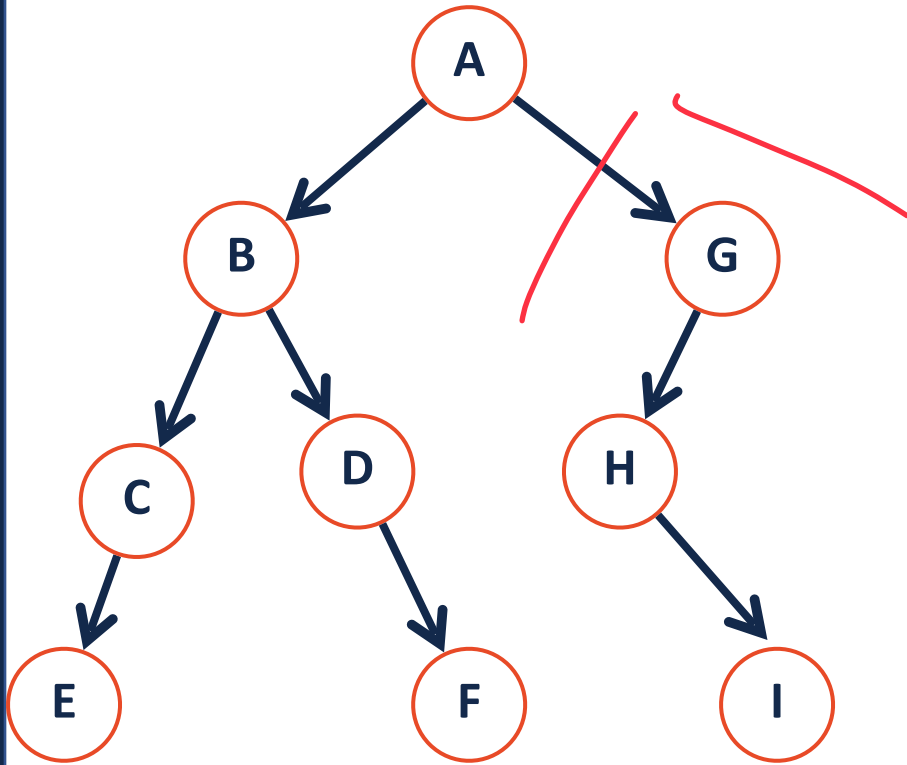
Traversals - In order



E C B D F A H I G

```
1 template<class T>
2 void BinaryTree<T>:: In Order(TreeNode * root)
3 {
4
5     if (root) {
6
7         _____;
8
9         _____ Order(root->left);
10
11         Print (root) _____;
12
13         _____ Order(root->right);
14
15         _____;
16
17     }
18
19
20
21 }
```

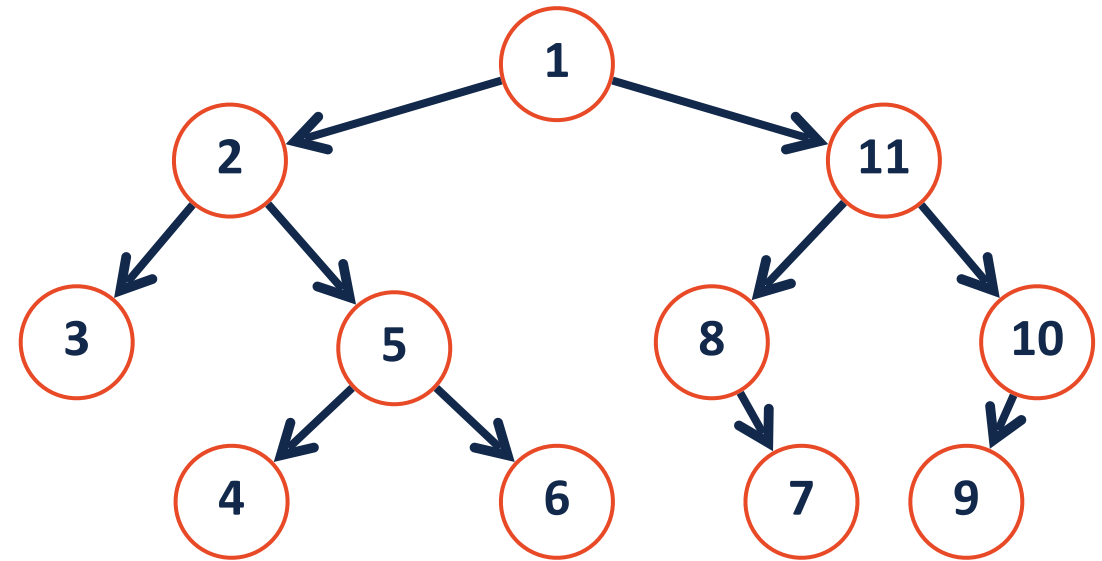
Traversals - Post Order



```
template<class T>
void BinaryTree<T>:: Post Order (TreeNode * root)
{
    if (root) {
        _____;
        _____ Order (root->left) ;
        _____;
        _____ Order (root->right) ;
        print (root) _____;
    }
}
```

E C F D B I H G A

Tree Traversals (Solution)



Updated 9/21: Postorder tip corrected

Pre-order: **1** 2 3 5 4 6 11 8 7 10 9

Tip: Preorder always starts with root!

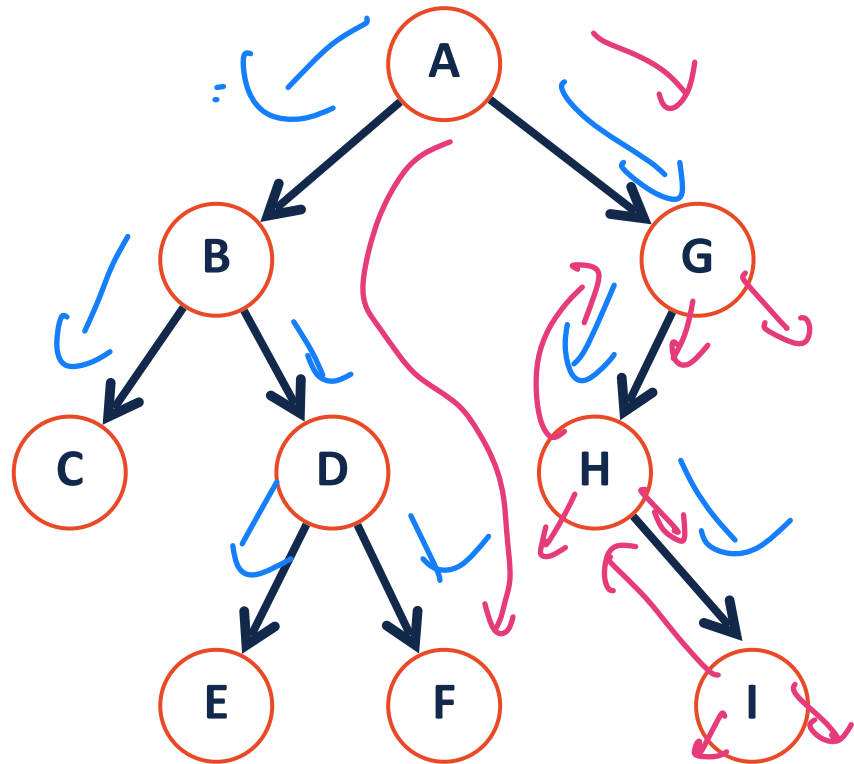
In-order: **3** 2 4 5 6 **1** 8 7 11 9 10

Tip: Inorder always starts with leftmost node. Root is after all left nodes!

Post-order: **3** 4 6 5 2 7 8 9 10 11 **1**

Tip: Root is always LAST node! First node depends on tree structure

Traversal vs Search



Post order (right to left) I ?

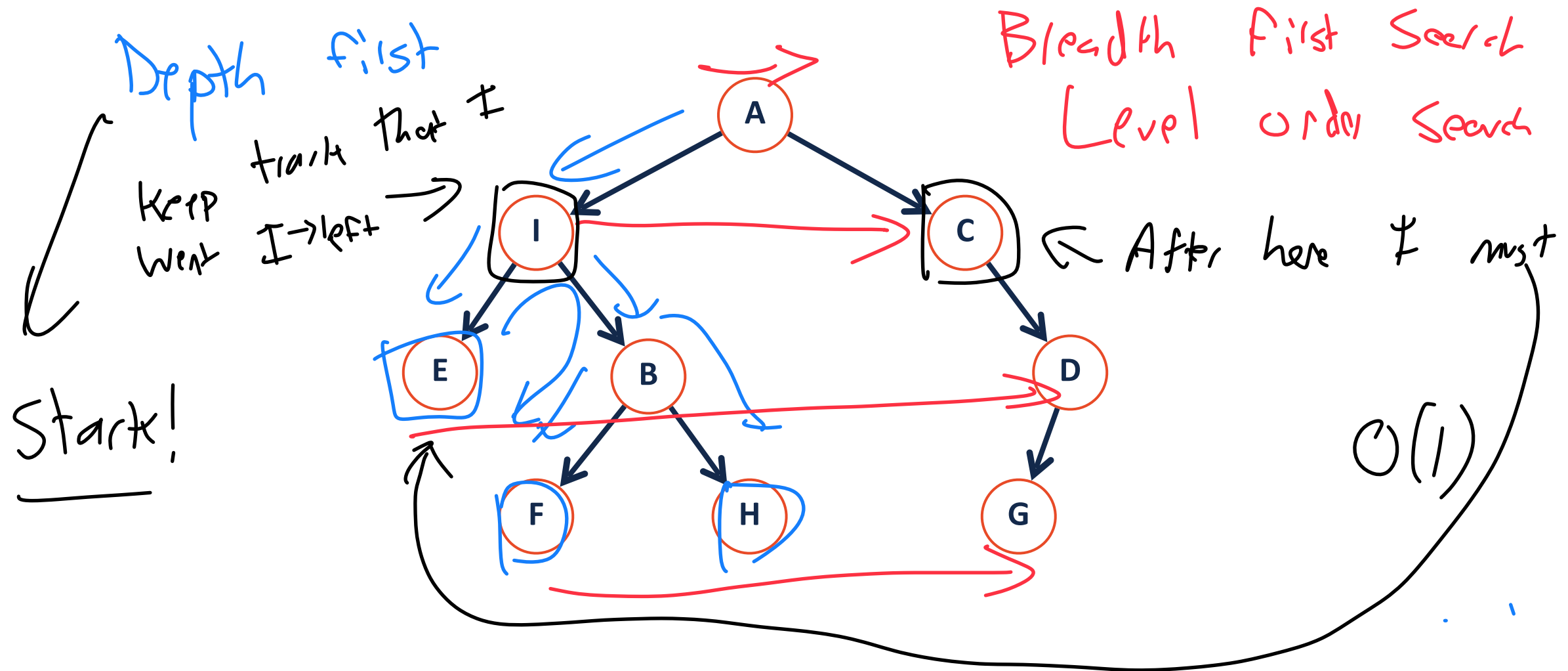
I H G F E D C B A

TRAVERSAL	SEARCH
Objective: Must visit every node	Objective: Needs to visit a specific node
Runtime: Always n nodes [$O(n)$ but always worst case]	Runtime: $O(n)$ for "unstructured" trees (We will see better runtimes later)
Examples : Pre-order In-order Post-order	Examples : Depth First Search Breadth First Search

Tree Search

There are two main approaches to searching a binary tree:

↑ Queue



Depth First Search

Explore as far along one path as possible before backtracking

1) Make a stack initialized to root

while my stack is not empty

tmp = stack.pop()

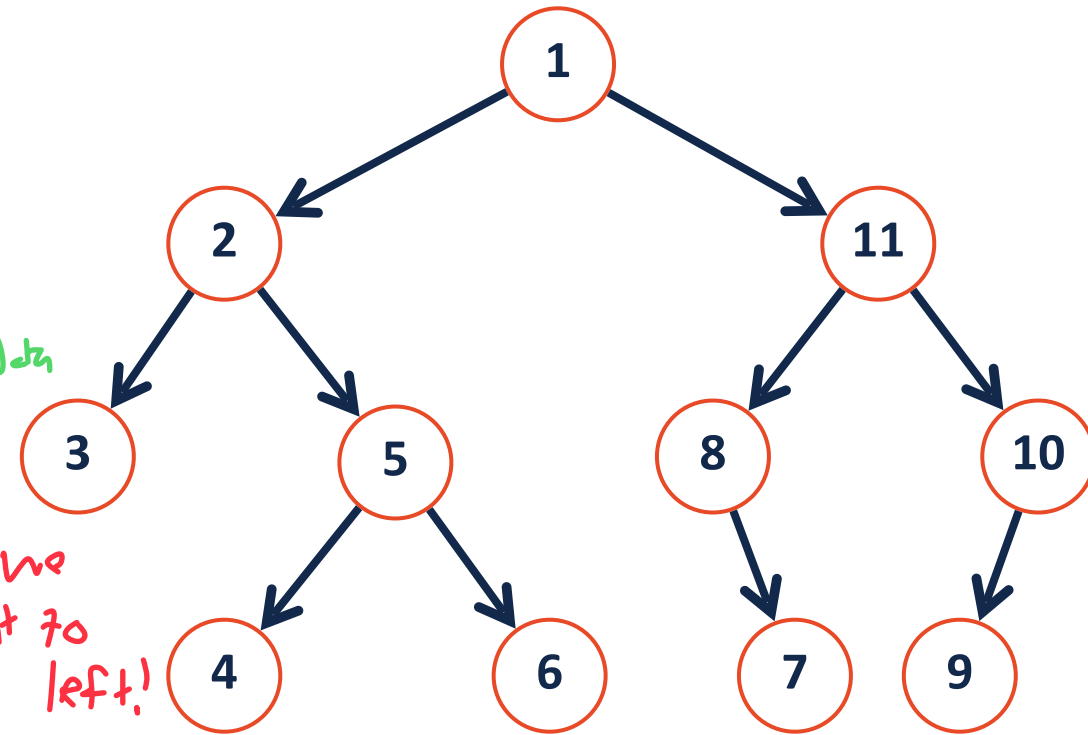
check if tmp.data is query //print tmp.data

Stack.push(tmp->right)

Stack.push(tmp->left)

If we want to go left!

Stack: ~~1~~, 11, ~~2~~, ~~5~~, ~~3~~, 6, 4
Print: 1, 2, 3, 5, 4, 6



Depth First Search

Explore as far along one path as possible before backtracking

Make a stack initialized with root

While stack isn't empty:

- Pop top element (as tmp)

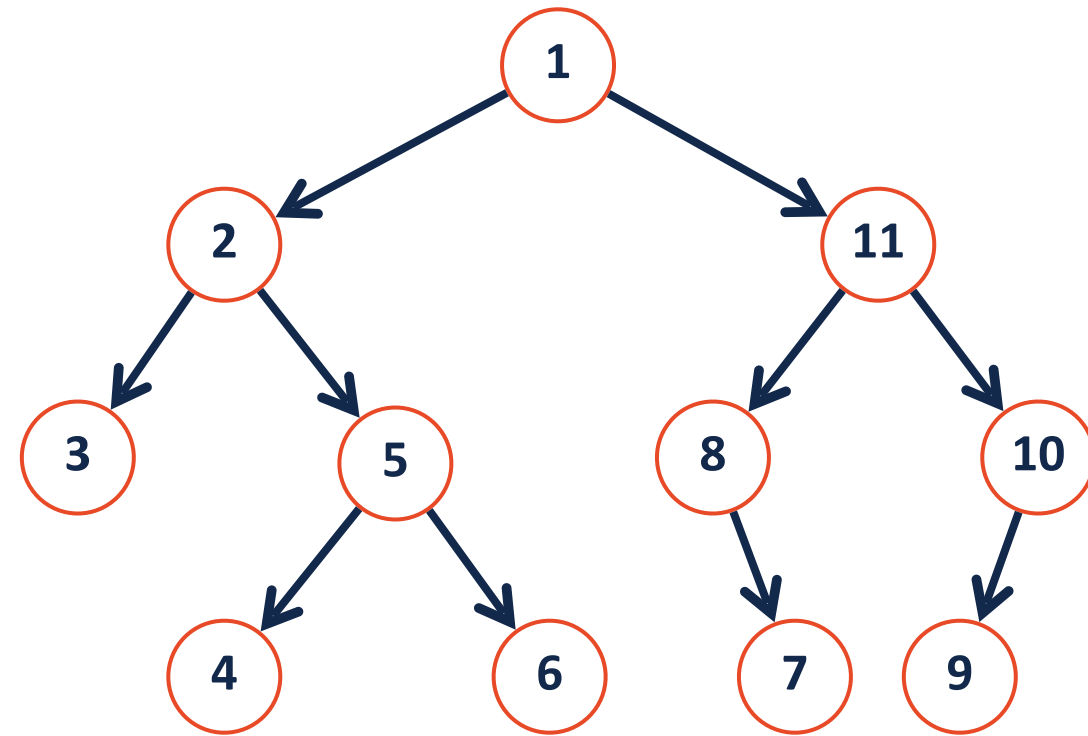
- Check if tmp is query (visual: 'print')

- Push tmp->right to stack

- Push tmp->left to stack

Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9



Pre-order traversal!

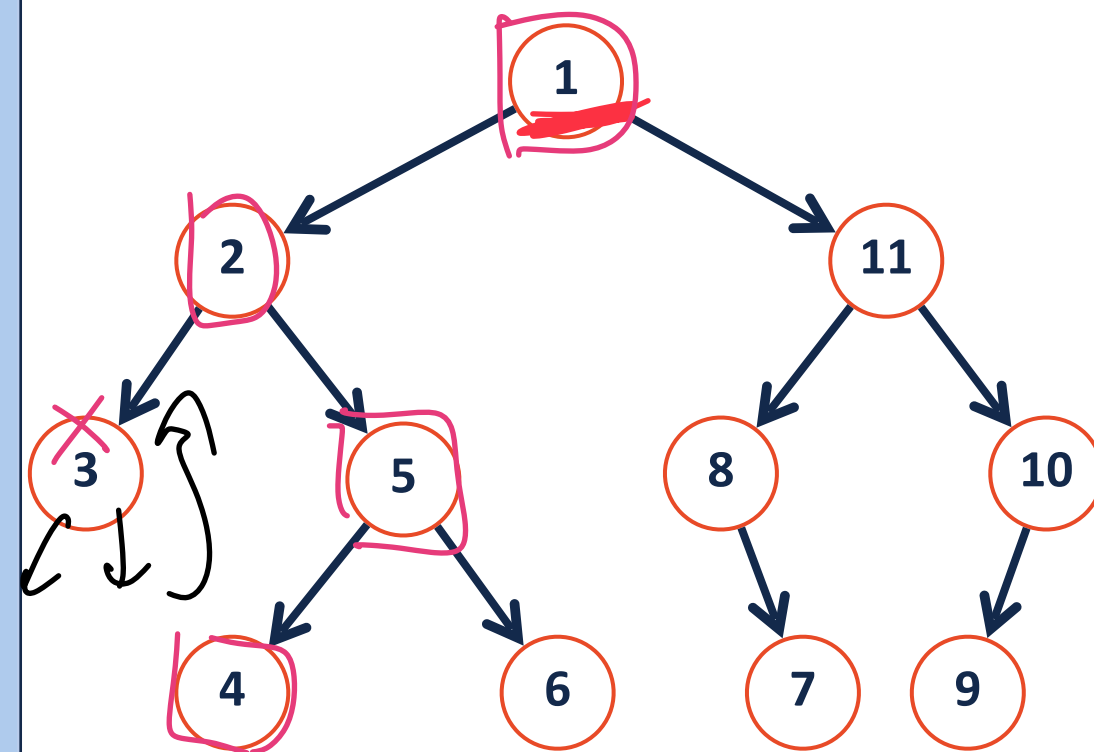
Depth First Search

How many processes are paused?
↳ height processes waiting at one time



Explore as far along one path as possible before backtracking

```
1 template<class T>
2 bool BinaryTree<T>::DFS_search(TreeNode*
  root, T& query)
3 {
4     if (root) {
5         if (process(root->data, query)) {
6             return true;
7         }
8         if (DFS_search(root->left, query)) {
9             return true;
10        }
11        if (DFS_search(root->right, query)) {
12            return true;
13        }
14    }
15    return false;
16 }
17
18
```



1 call (frozen @ line 8)

Breadth First Search

Fully explore level i before exploring level i+1 width of tree limitation!

1) Make a queue init to root

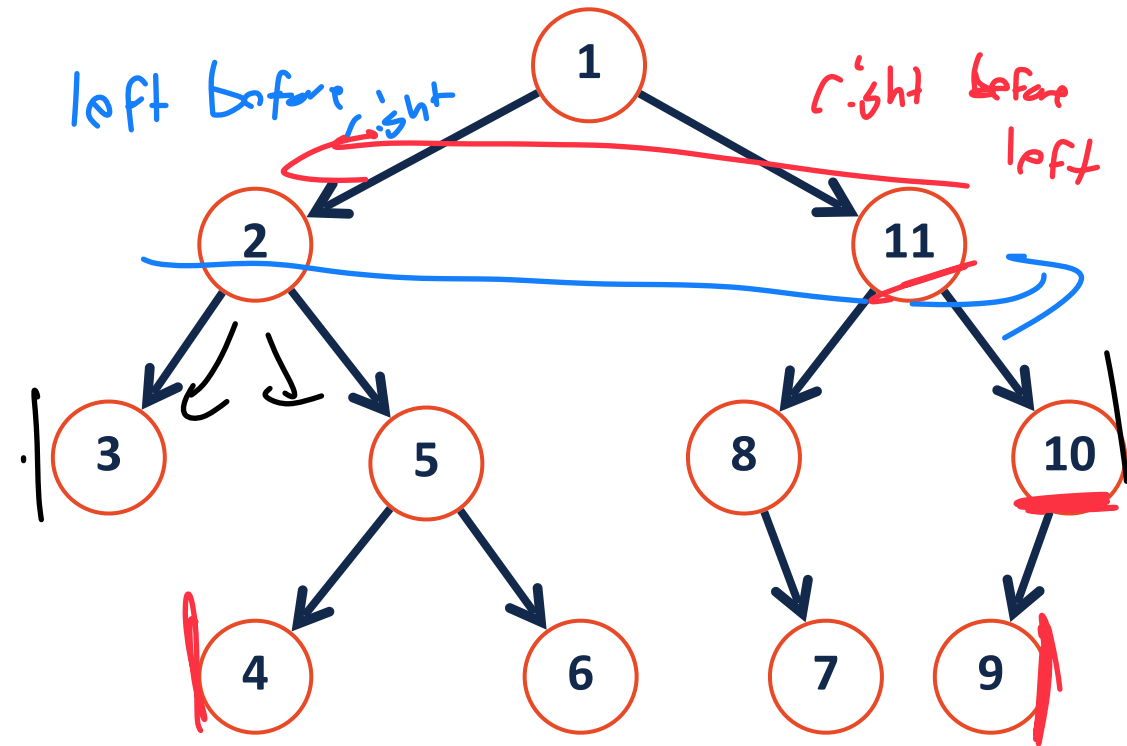
while queue not empty:

tmp = queue.dequeue()

check if tmp is query

queue.enqueue(tmp.left); right

queue.enqueue(tmp.right); left



Queue: ~~1~~, ~~2~~, ~~11~~, ~~3~~, ~~5~~, ~~8~~, ~~10~~, 4, 6, 7, 9

Print: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Breadth First Search

Fully explore level i before exploring level $i+1$

Make a queue initialized with root

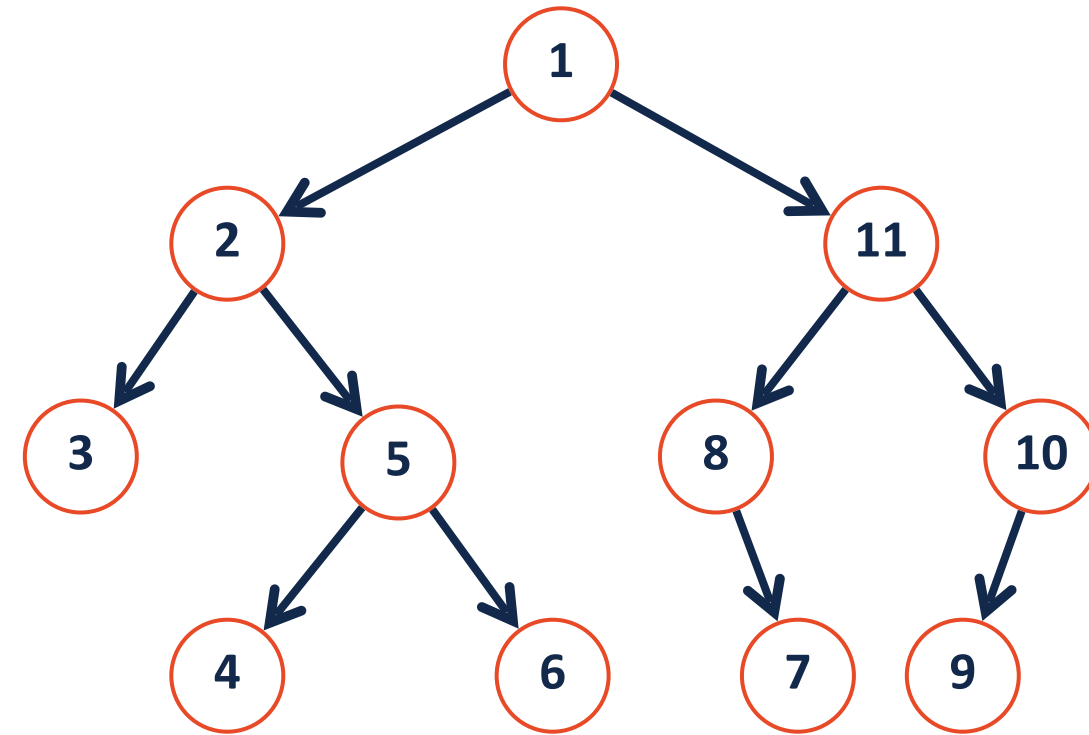
While queue isn't empty:

- Dequeue front element (as tmp)

- Check if tmp is query (visual: 'print')

- Enqueue tmp->left

- Enqueue tmp->right



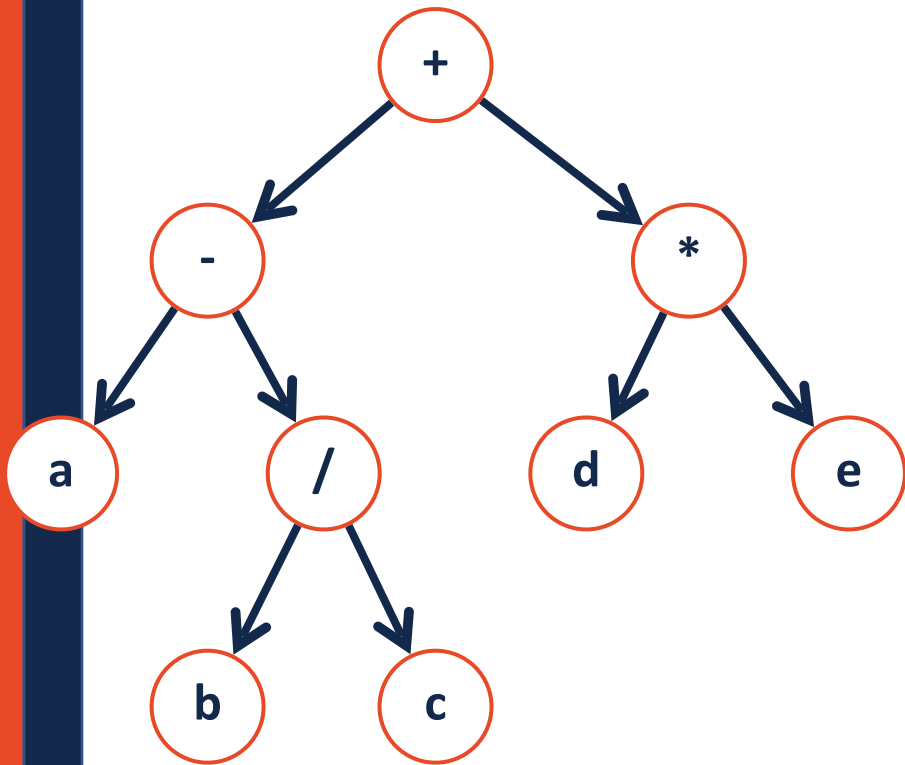
Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Print: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Updated 9/21: Print was DFS not BFS

Level-Order Traversal

Try recursive version on own!



```
1 template<class T>
2 bool BinaryTree<T>::BFS_search(TreeNode* root, T&
  query)
3 {
4     if (!root) return false;
5
6     std::queue<TreeNode*> q;
7     q.push(root); // enqueue == push
8
9     while (!q.empty()) {
10         TreeNode* temp = q.front();
11         q.dequeue();
12
13         if (proc(temp->data, query)) return true;
14
15         if (temp->left) q.enqueue(temp->left);
16         if (temp->right) q.enqueue(temp->right);
17     }
18
19     return false;
20 }
```

Handwritten annotations on the code:

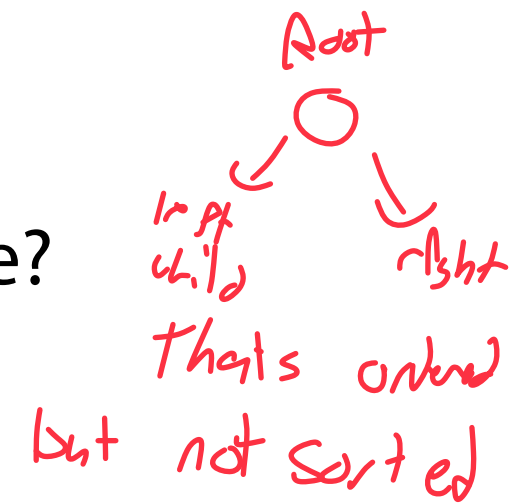
- Next to line 4: *else case* with an arrow pointing to the `if (!root)` condition.
- Next to line 6: *init* with a bracket spanning lines 6 and 7.
- Next to line 13: `return true;`
- Next to line 19: `return false;`

Tree Search Tradeoffs

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

We want $O(\log n)$!

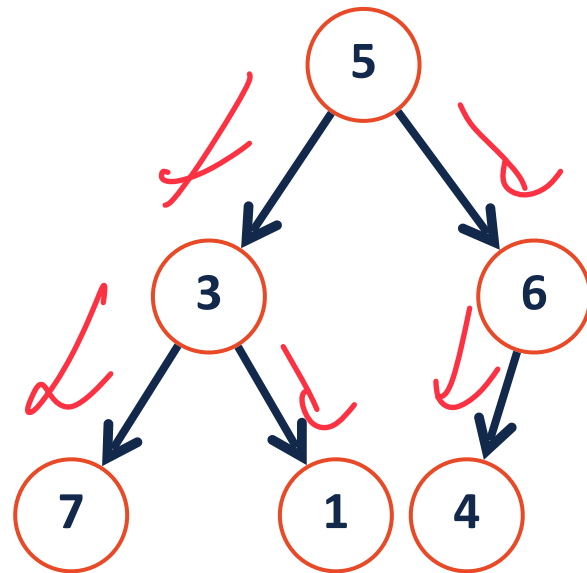
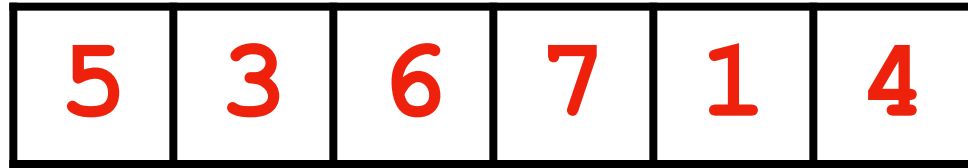
What do we trade in order to do so?



Add Sorting!

Improved search on a binary tree

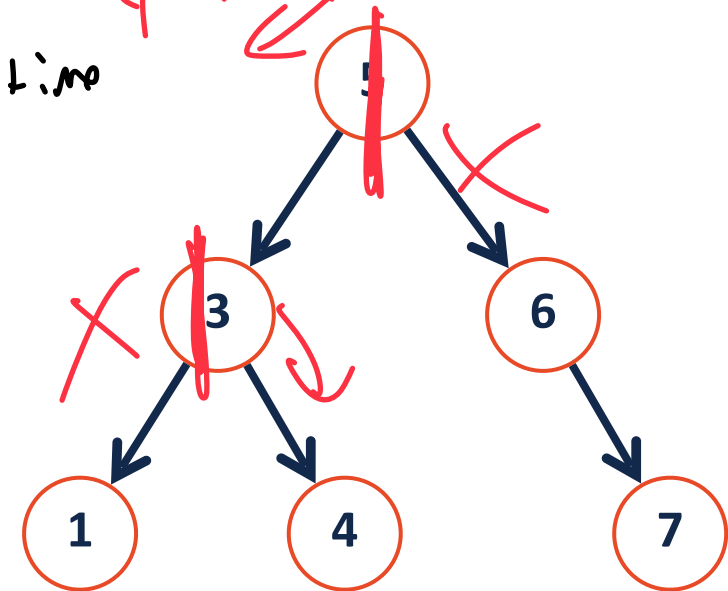
we've seen this



An analogy / visualization

Improves search time

at what cost?



Dictionary ADT

Data is often organized into key/value pairs:

Word → Definition

Course Number → Lecture/Lab Schedule

Node → Incident Edges

Flight Number → Arrival Information

URL → HTML Page

...

$D[\text{key}] \rightarrow \text{return the value}$



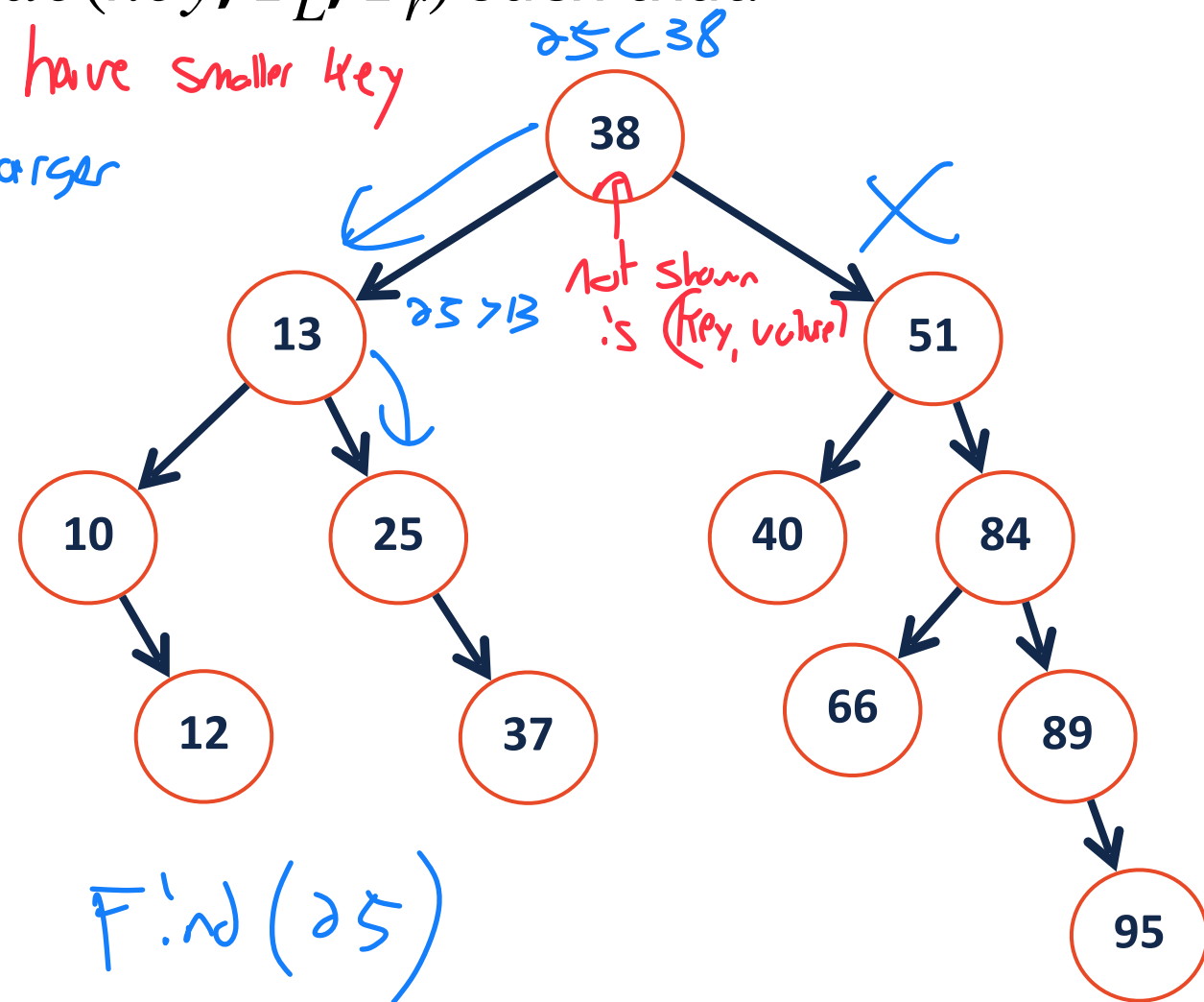
Binary Search Tree (BST)

TreeNode
↳ T & key
↳ T & value
TreeNode * left, right

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

All nodes to the left of root have smaller key
right

larger



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         TreeNode *root_;
21         /* ... */
22     };
23
```

Binary Search Tree ADT



The BST Node is not meaningfully different

Our changes are purely in the interface

Insert

Remove

Traverse

Find

Constructor

BST Find

lets write recursively

Base case

If root is empty, return ~~_____~~

If root is query, return _____

Recursive Step

if query is smaller than root.key

↳ Recurse left

else (larger)

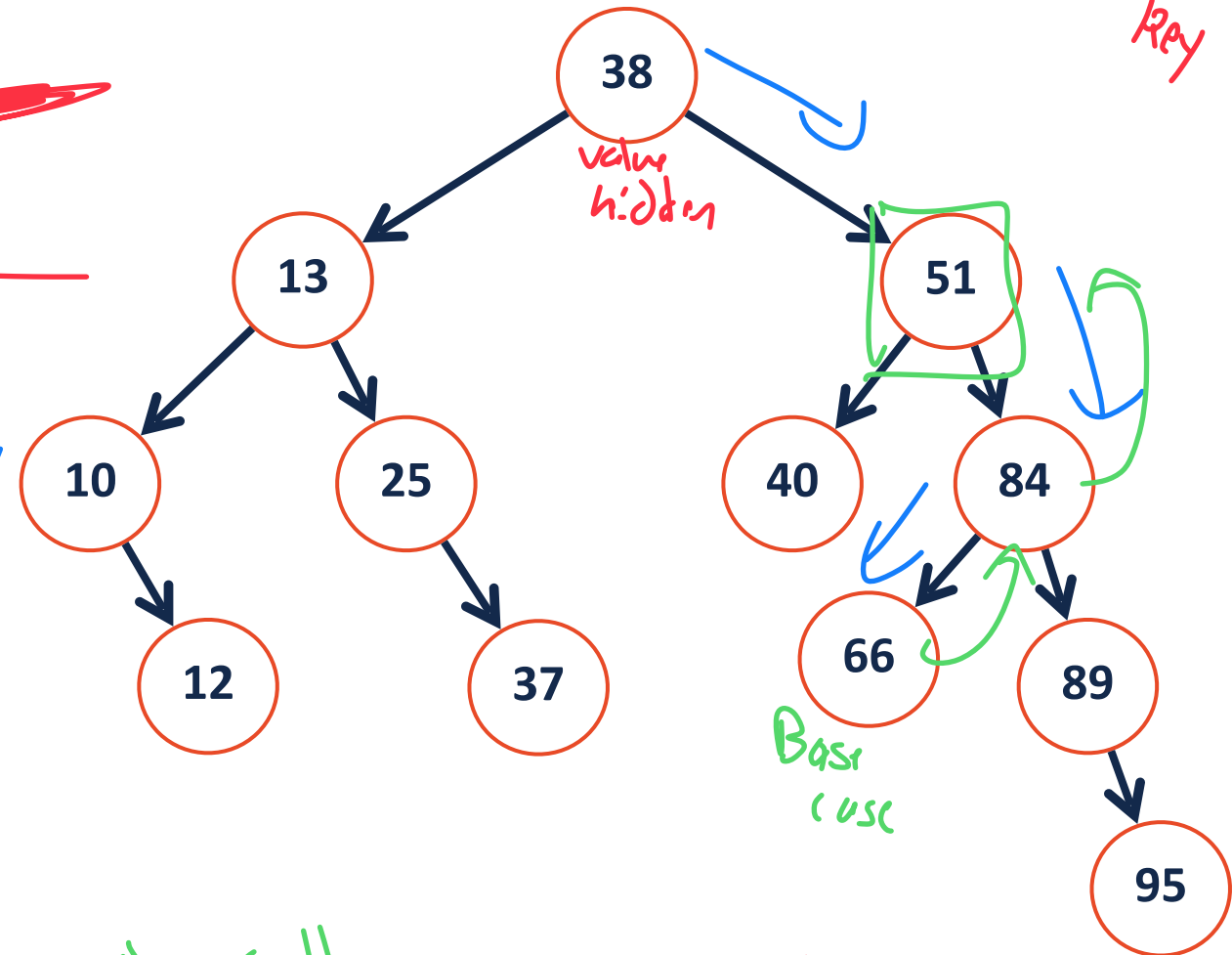
↳ Recurse right

Combining Step

return the result of recursive call

find(66)

↑
always key



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

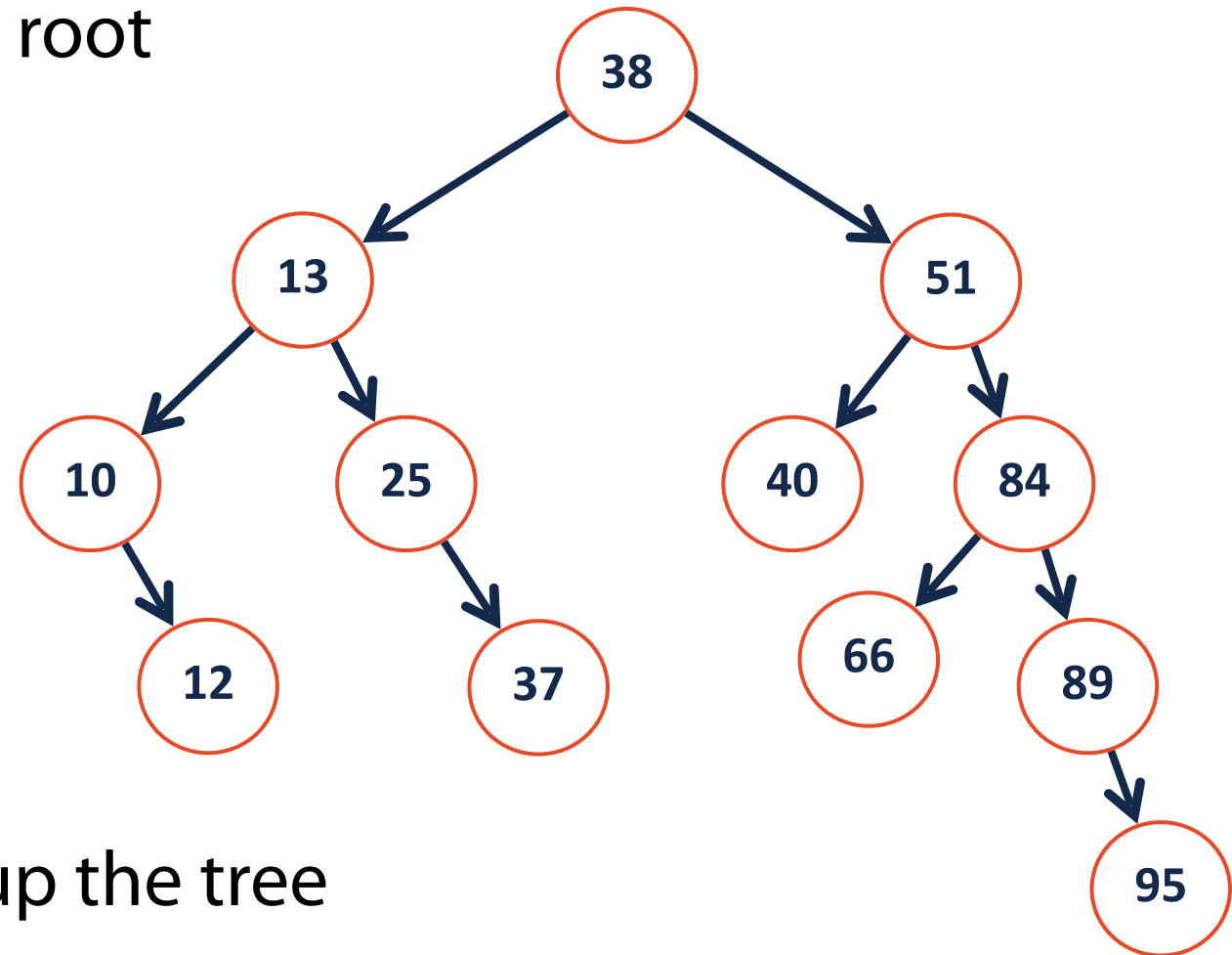
Recursion:

tmp < query, recurse right

tmp > query, recurse left

Combining:

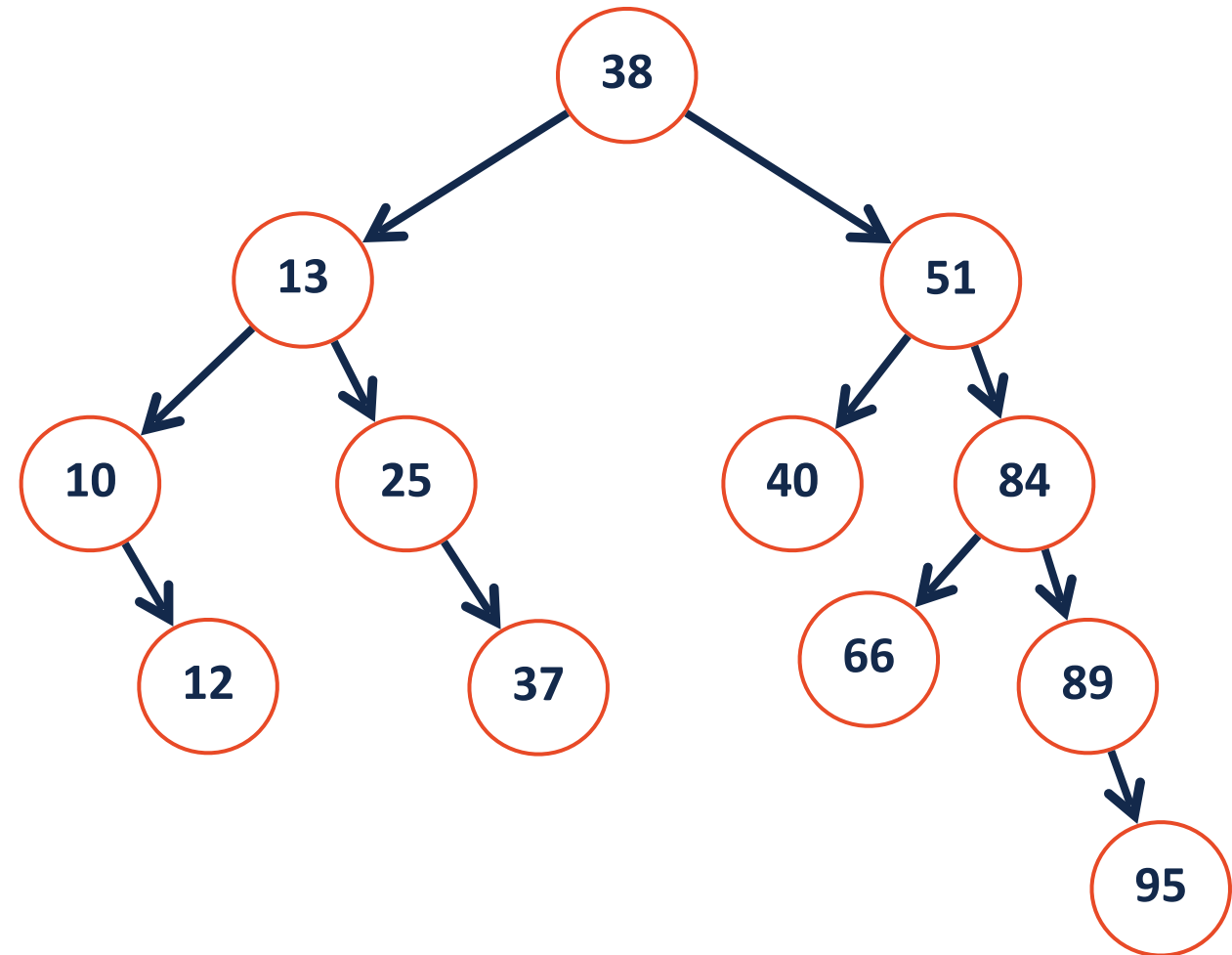
Return the recursive value back up the tree



BST Find

find(9)

Make sure you can follow the search path!





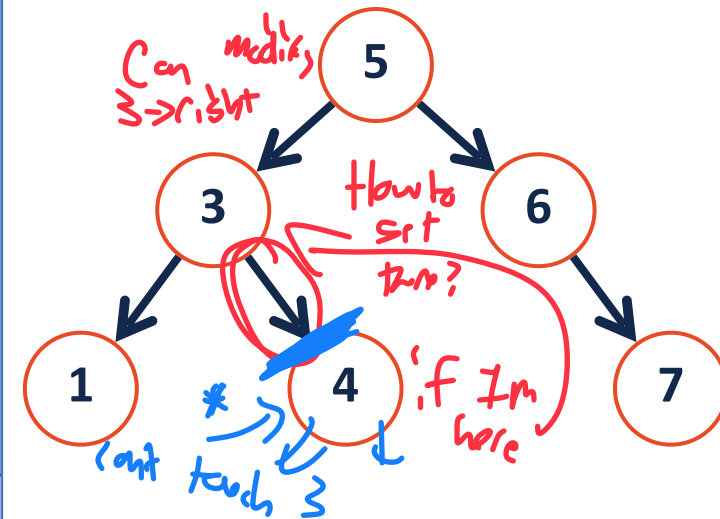
```

1 template<typename K, typename V>
2
3     __find(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }

```

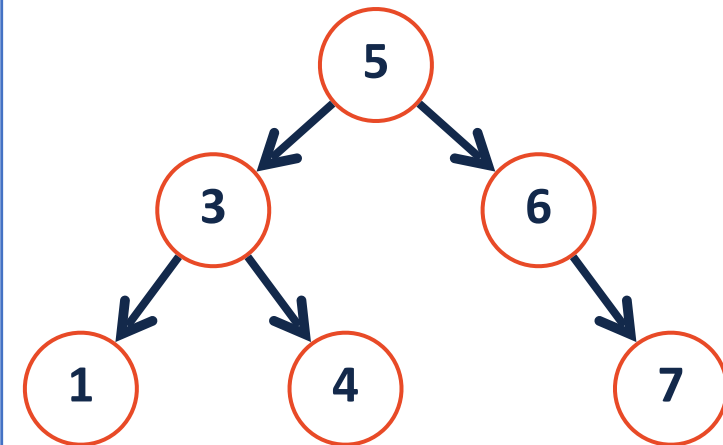
Handwritten notes on the slide:

- 4 / V & (with a slash and a female symbol)
- 15%
- TreeNode *
- 30% (example in the)
- 55% (example in red)
- TreeNode *
- if to type





```
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
```



Question about Big O

$f(k)$

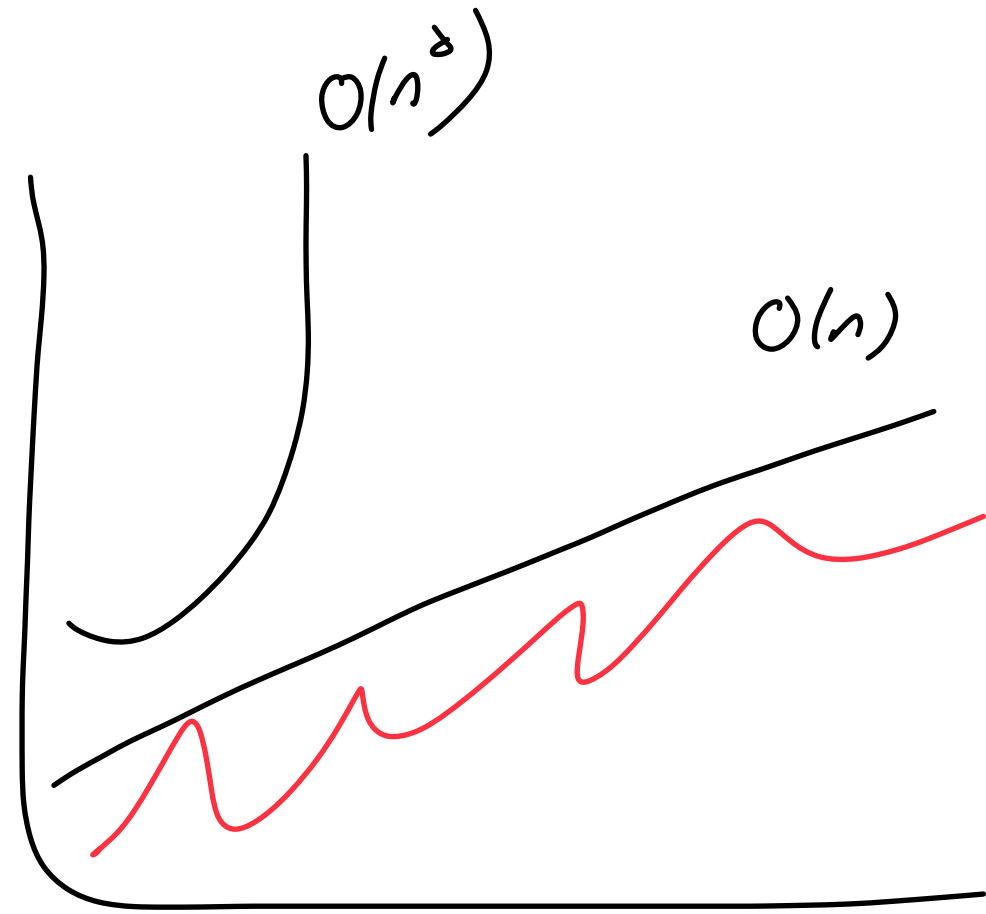
If k is small & n is big
 $n \gg k$

If $k \approx n$

" $O(k)$ where $k \leq$ at most n "

$\hookrightarrow$ It depends on k but k can be as large as n

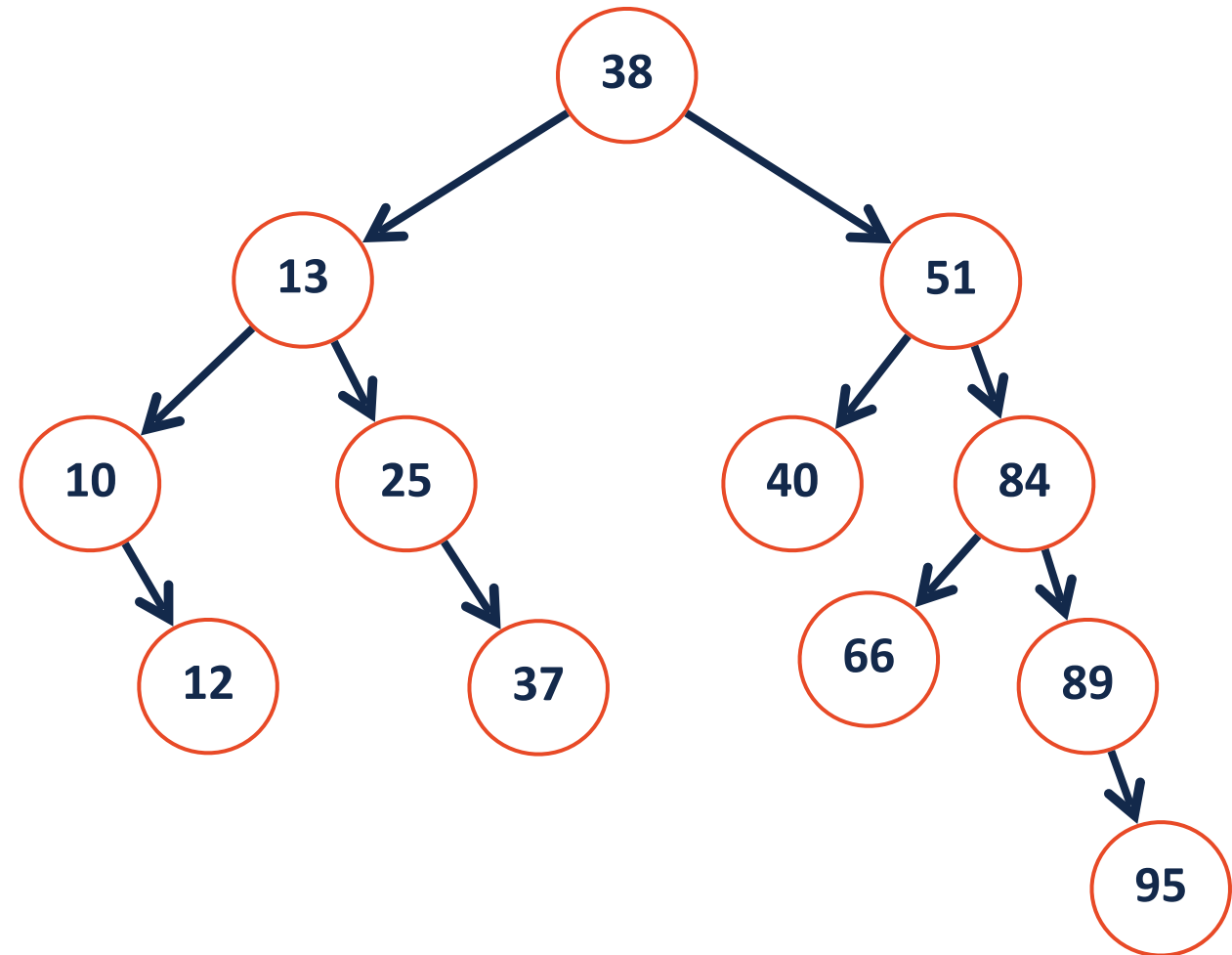
↑
ops



$n \rightarrow$

BST Insert

insert(33, v)

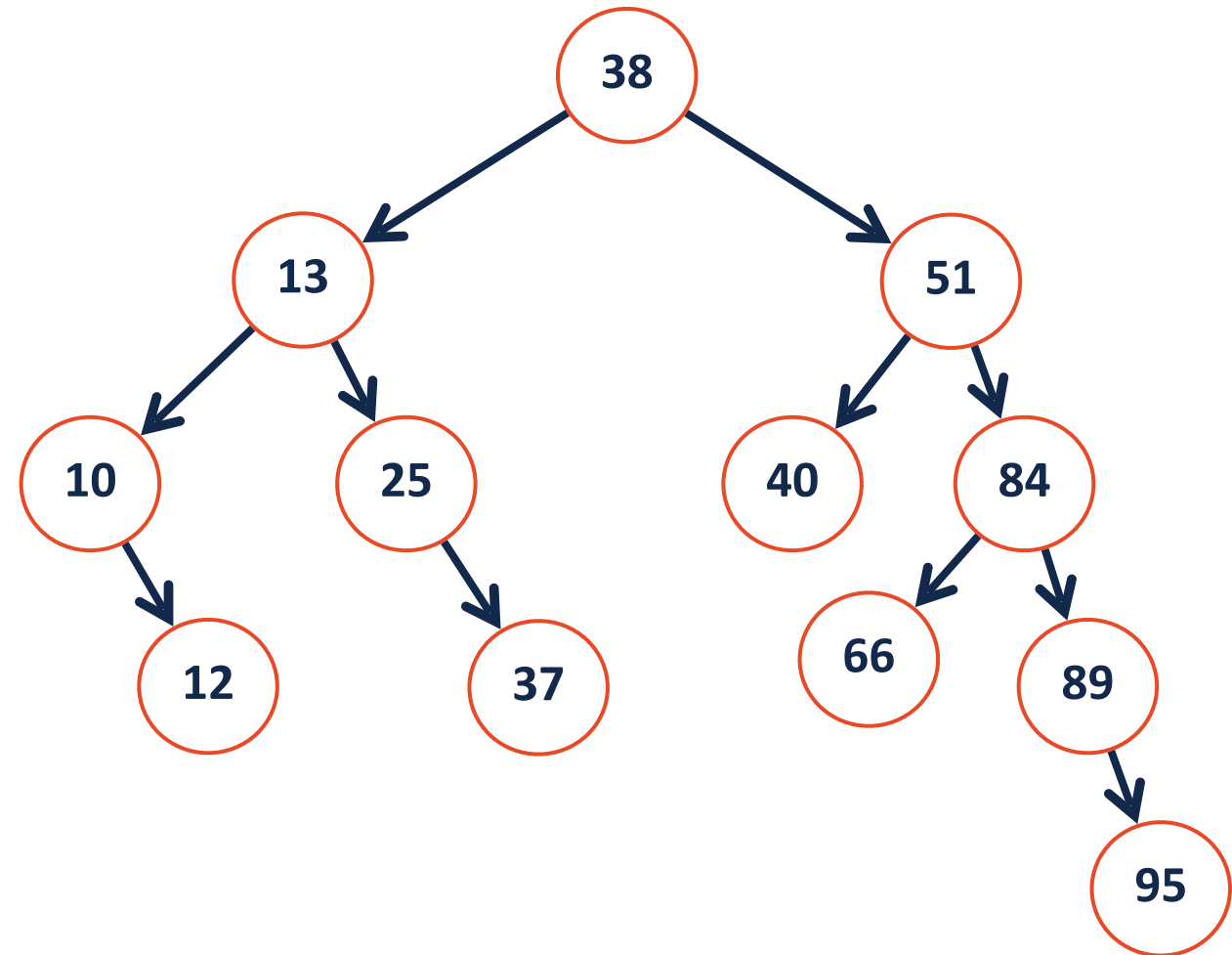


BST Insert

insert(15, v)

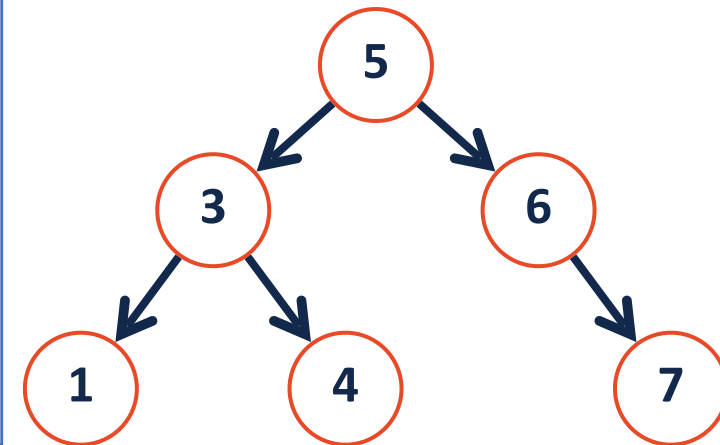
Find the insert location using `_find()`

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
```

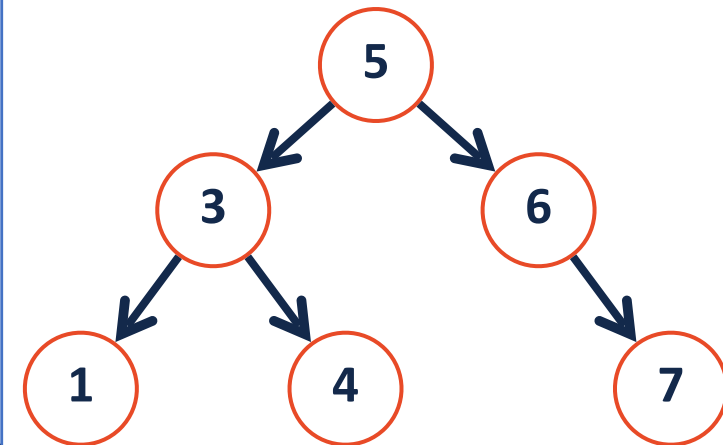
```
1 template<typename K, typename V>
2
3 void _insert(TreeNode *& root, const K & key, const V & val) {
4
5
6
7
8
9
10
11
12
13
14
15
16 }
```





```
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)

