Data Structures

Tree Search & BST

CS 225

Brad Solomon

September 18, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

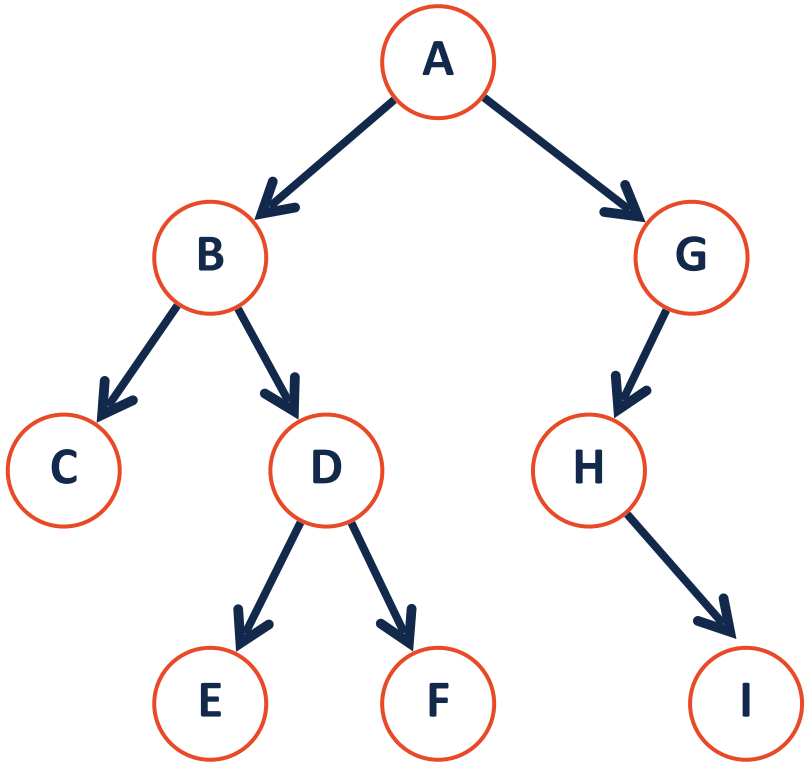
Review Tree Traversals

Explore implementations of DFS and BFS on binary trees

Extend binary trees into binary *search* trees

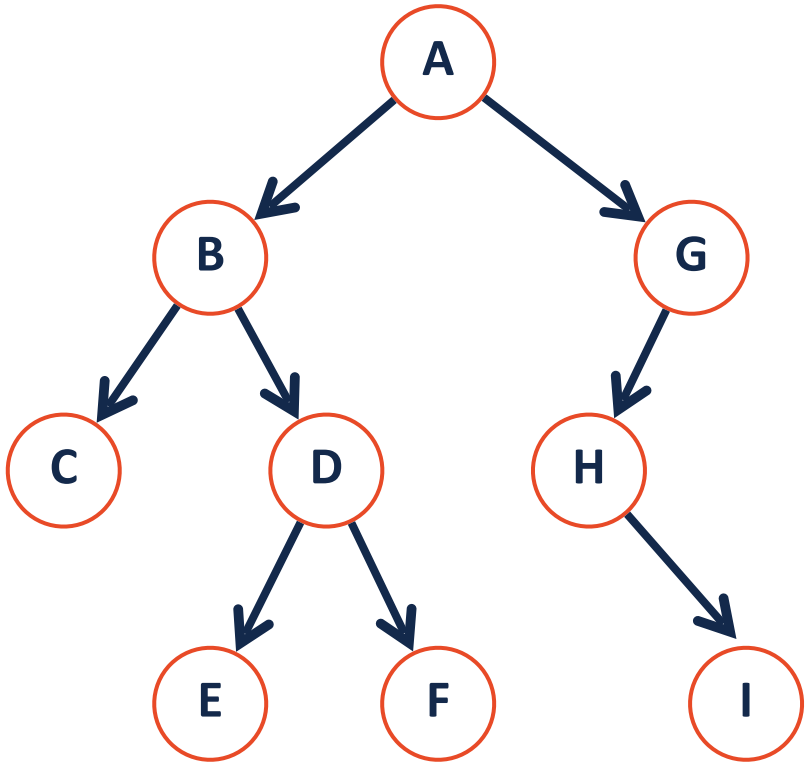
Build conceptual and coding understanding of BST

Traversals - Pre Order



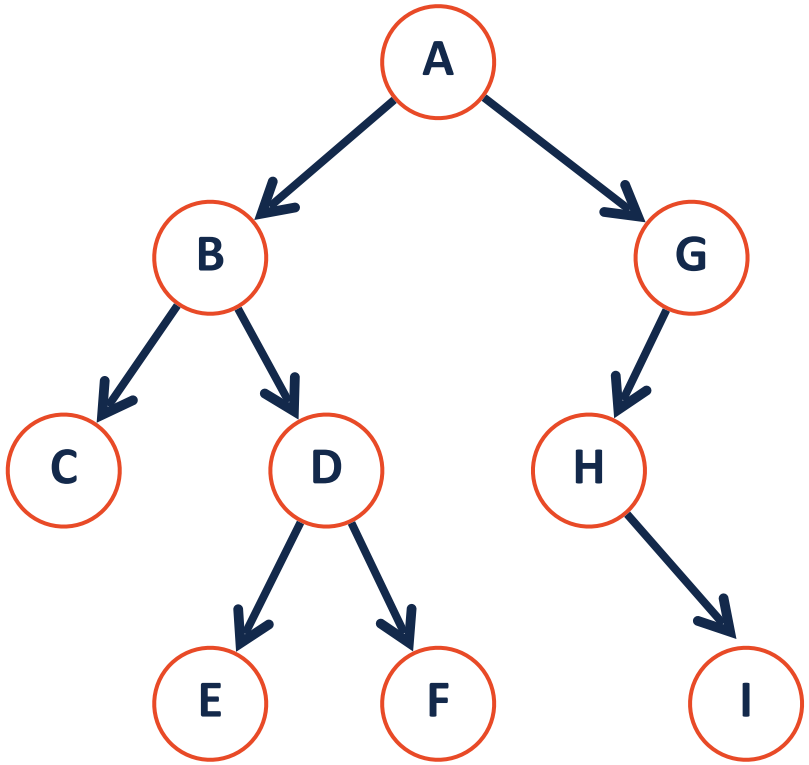

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7        ____________________________;
8
9        _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16
17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

Traversals - In order



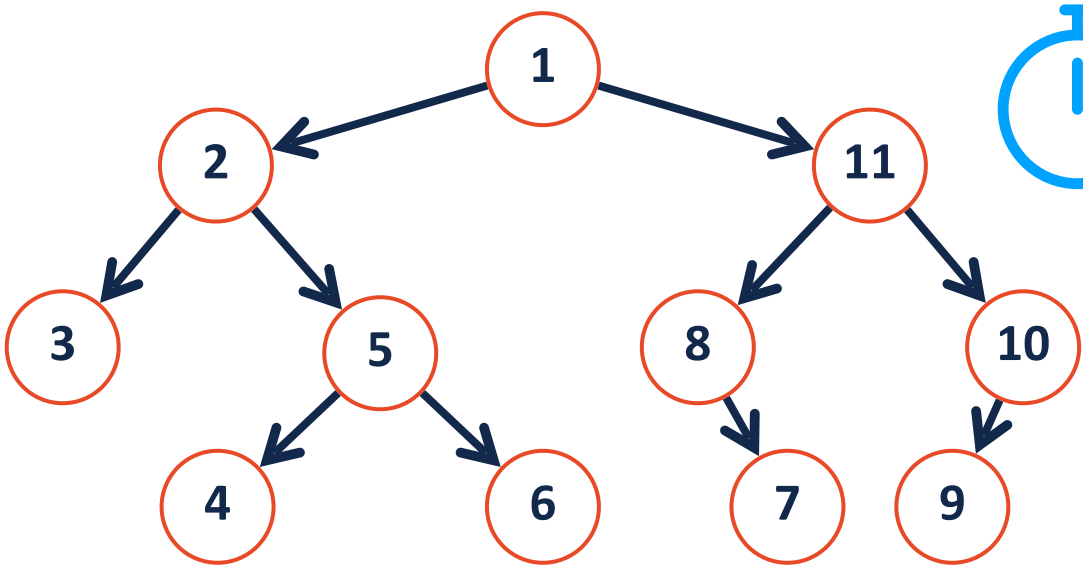

```
 1 template<class T>
 2 void BinaryTree<T>::_____Order(TreeNode * root)
 3 {
 4
 5   if (root) {
 6
 7       ____________________________;
 8
 9       _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16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

Traversals - Post Order




```
template<class T>
void BinaryTree<T>::_____Order(TreeNode * root)
{

  if (root) {

      ____________________________;

      _____Order(root->left);

      ____________________________;

      _____Order(root->right);

      ____________________________;

  }



}
```

Tree Traversals

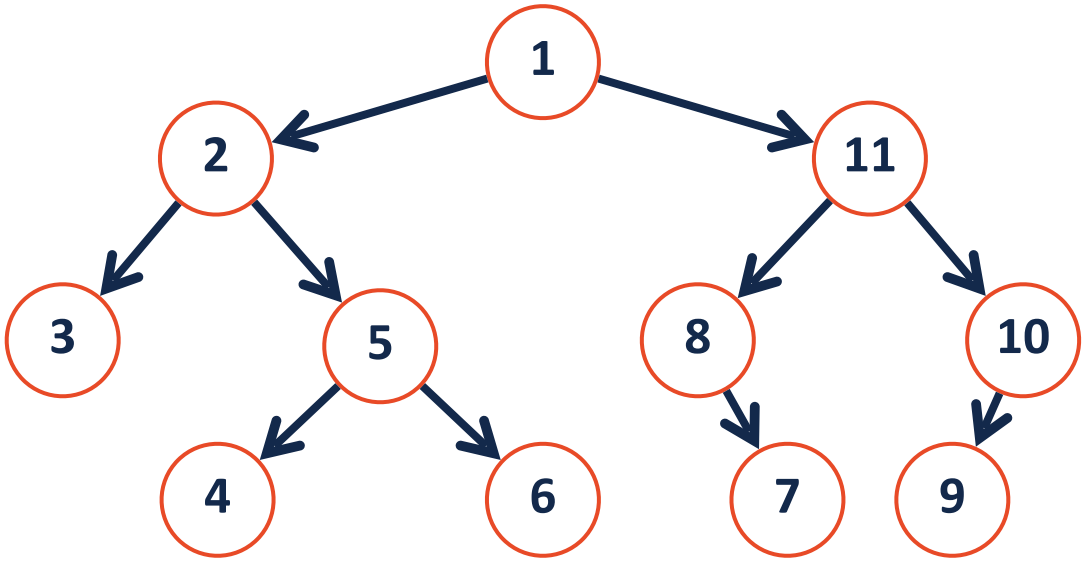


Pre-order:

In-order:

Post-order:

Tree Traversals (Solution)



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

Tip: Preorder always starts with root!

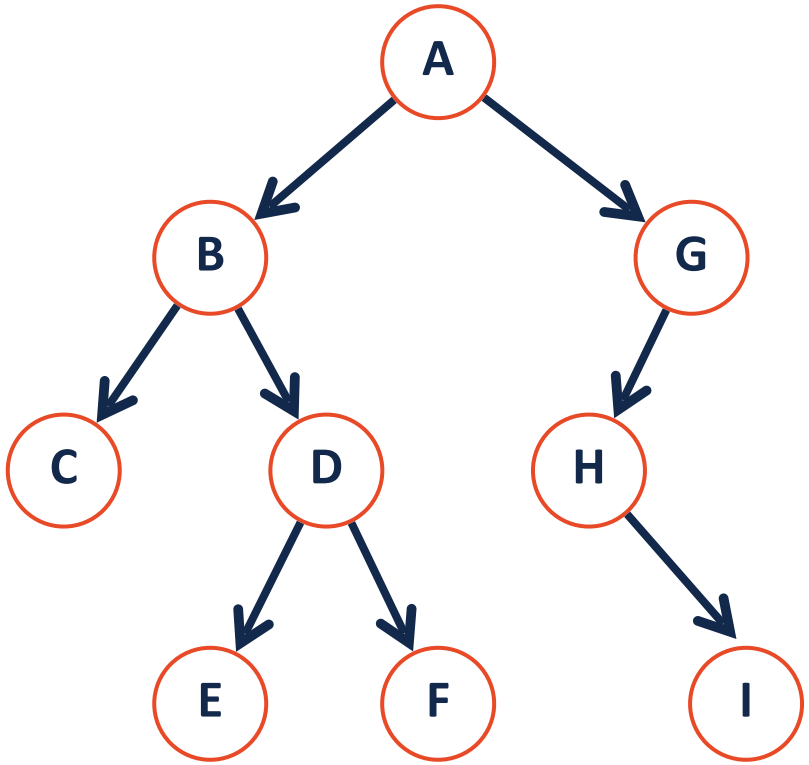
In-order: **3** 2 4 5 6 **1** 8 7 11 9 10

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: Post always starts with leftmost node. Root is always LAST node!

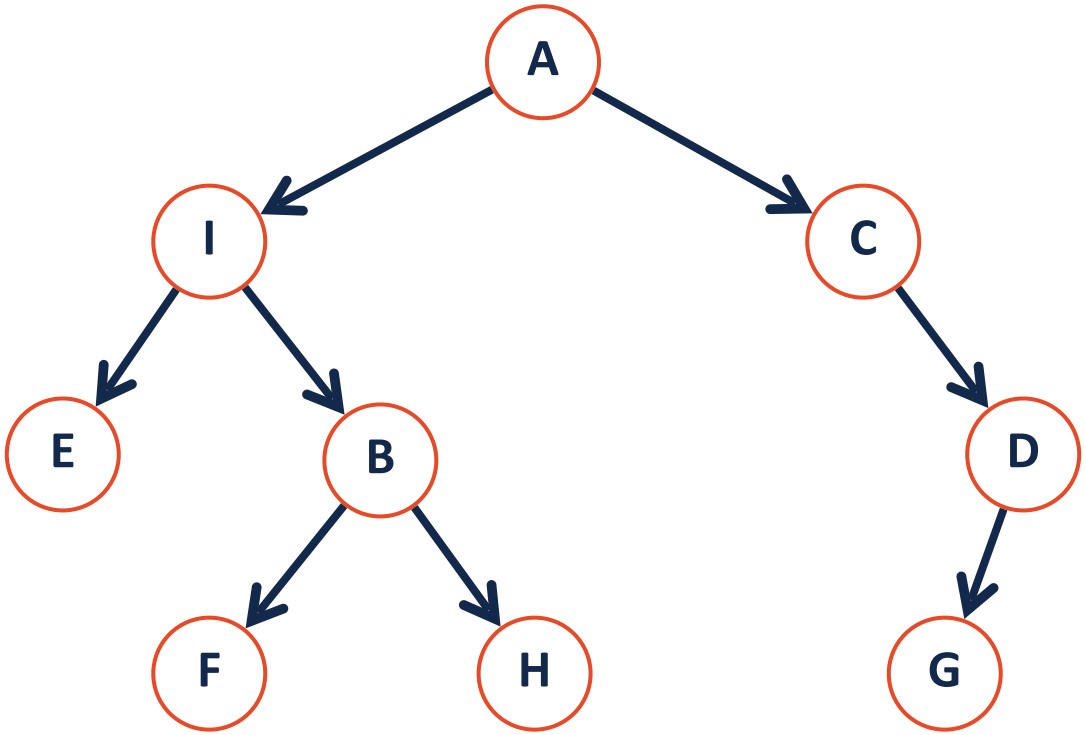
Traversal vs Search



TRAVERSAL	SEARCH
Objective - Must visit every node	Objective - Needs to visit a specific node
Runtime is always O(n)	Runtime is O(n) for “unstructured” trees (We will see better runtimes later)
Types : Pre-order In-order Post-order	Types : Depth First Search Breadth First Search

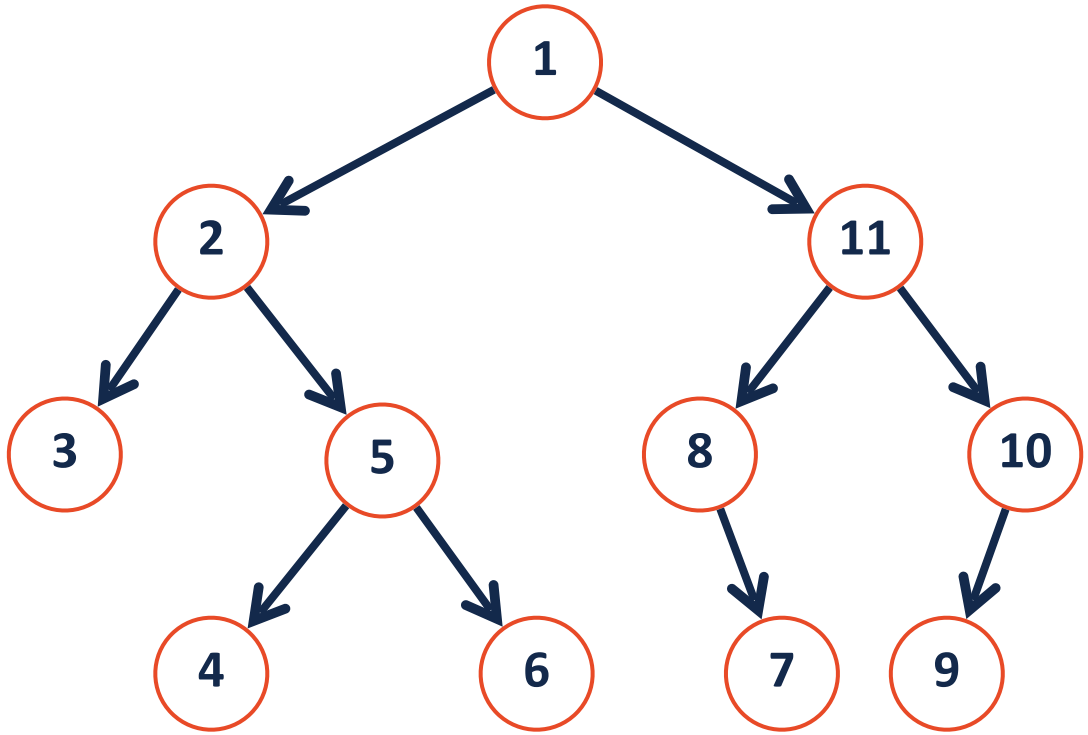
Tree Search

There are two main approaches to searching a binary tree:



Depth First Search

Explore as far along one path as possible before backtracking



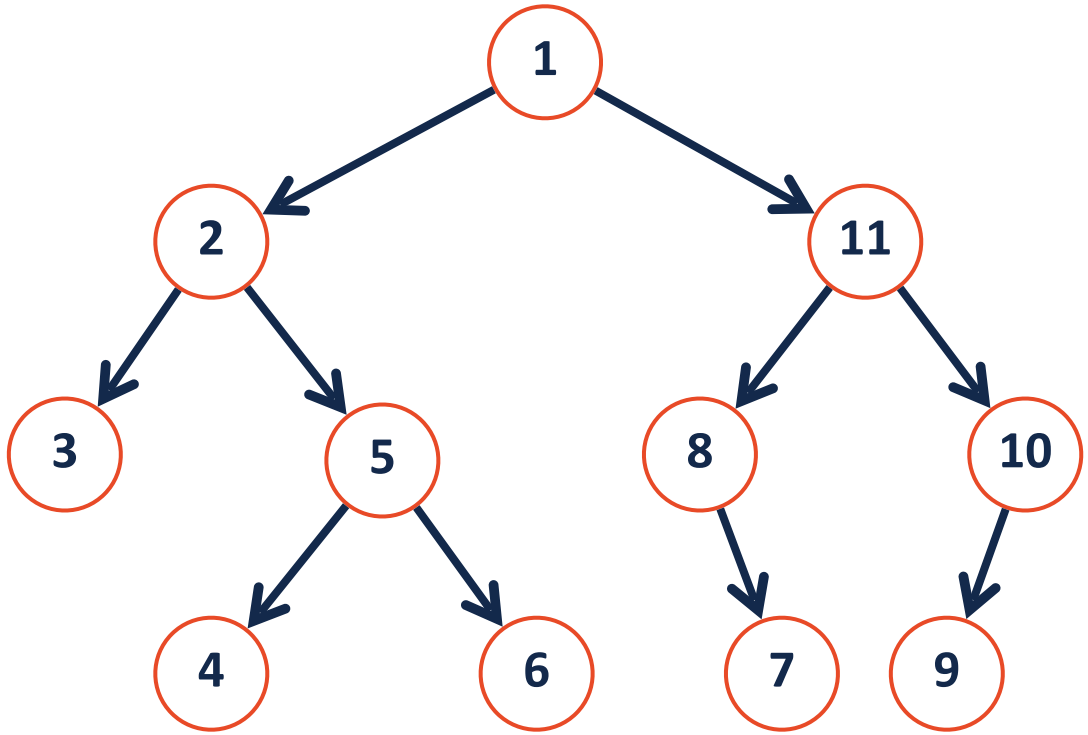
Depth First Search

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



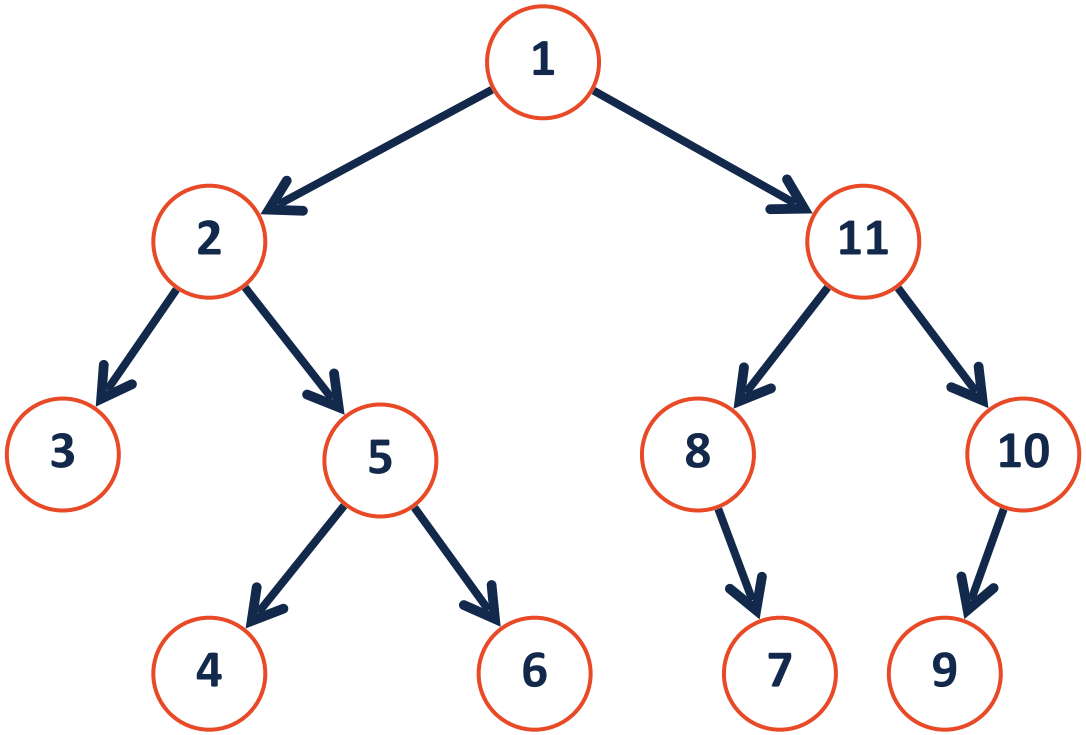
Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

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

Depth First Search

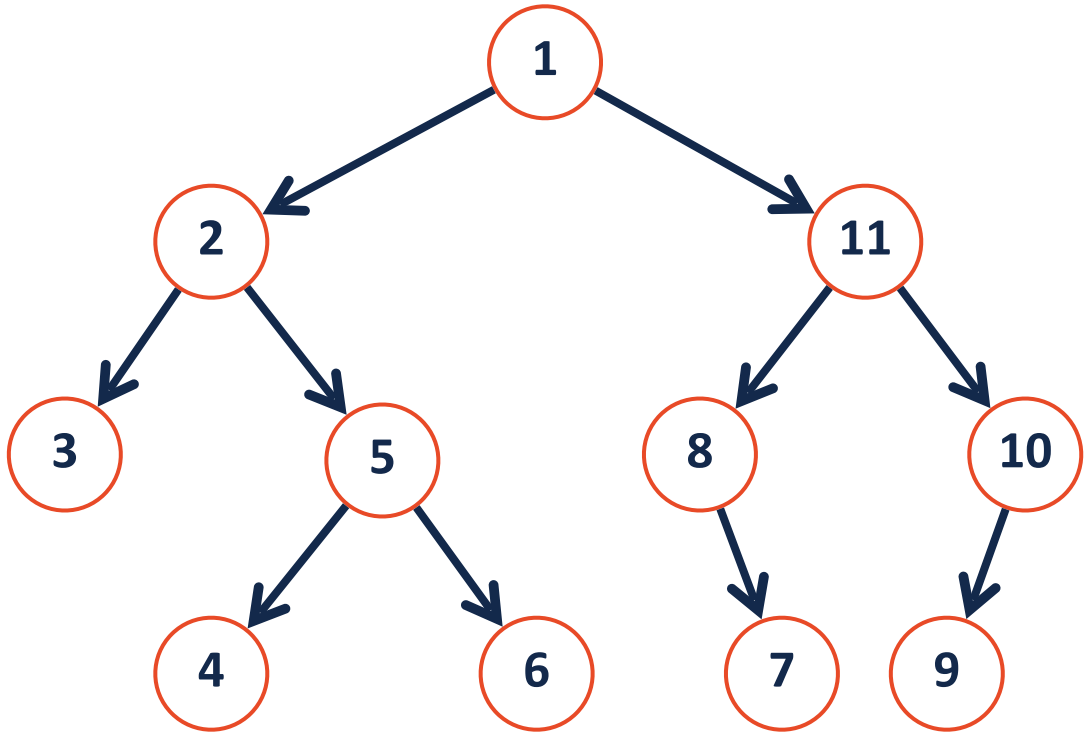
Explore as far along one path as possible before backtracking

```
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
```



Breadth First Search

Fully explore level i before exploring level i+1



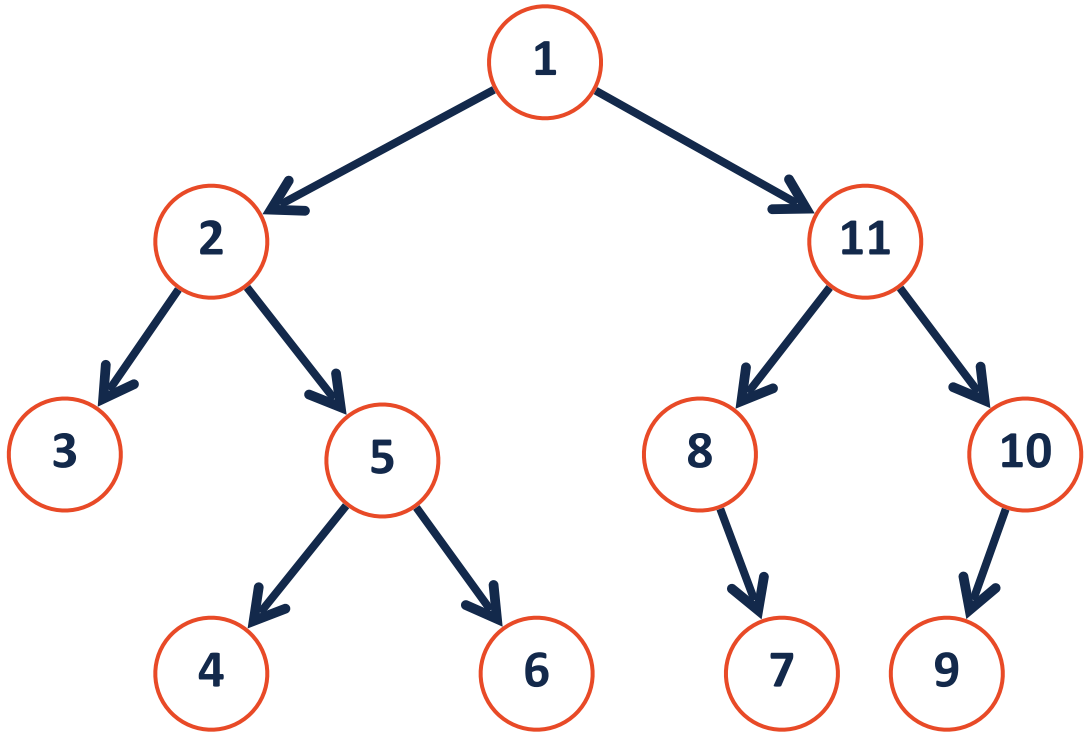
Breadth First Search

Fully explore level i before exploring level i+1

Make a queue initialized with root

While queue isn't empty:

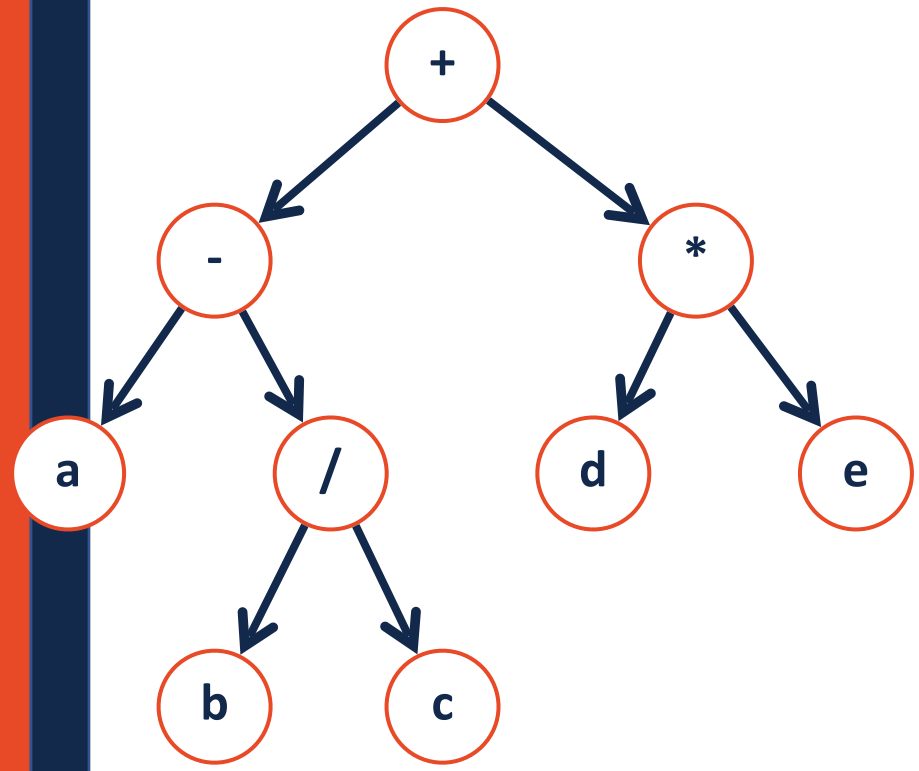
- Dequeue front element (as `tmp`)
- Check if tmp is query (visual: 'print')
- Enqueue `tmp->left`
- Enqueue `tmp->right`



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

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

Level-Order Traversal




```
 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 }
```

Tree Search Tradeoffs

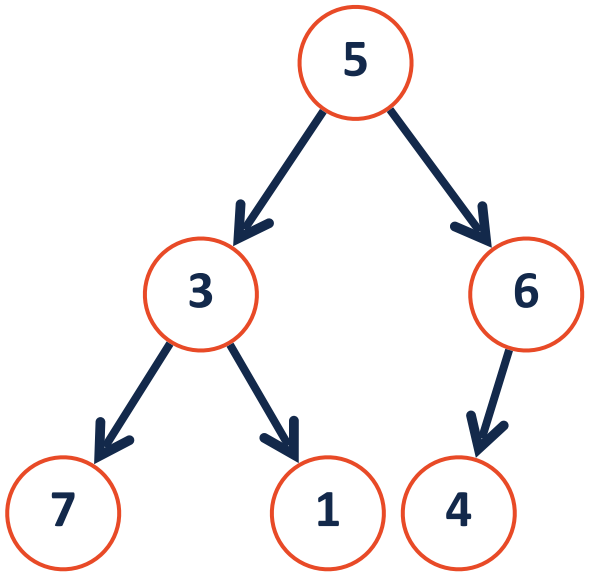
How can we improve our ability to search a binary tree?

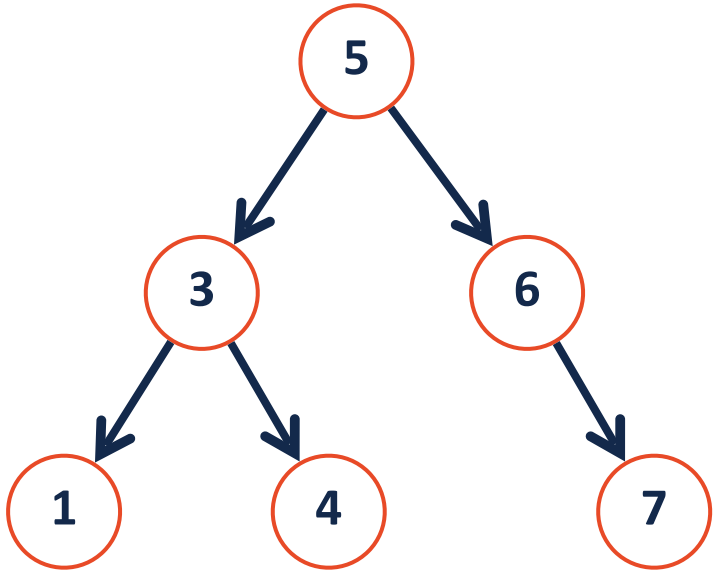
What do we trade in order to do so?

Improved search on a binary tree

5	3	6	7	1	4

1	3	4	5	6	7





Dictionary ADT

Data is often organized into key/value pairs:

Word → Definition

Course Number → Lecture/Lab Schedule

Node → Incident Edges

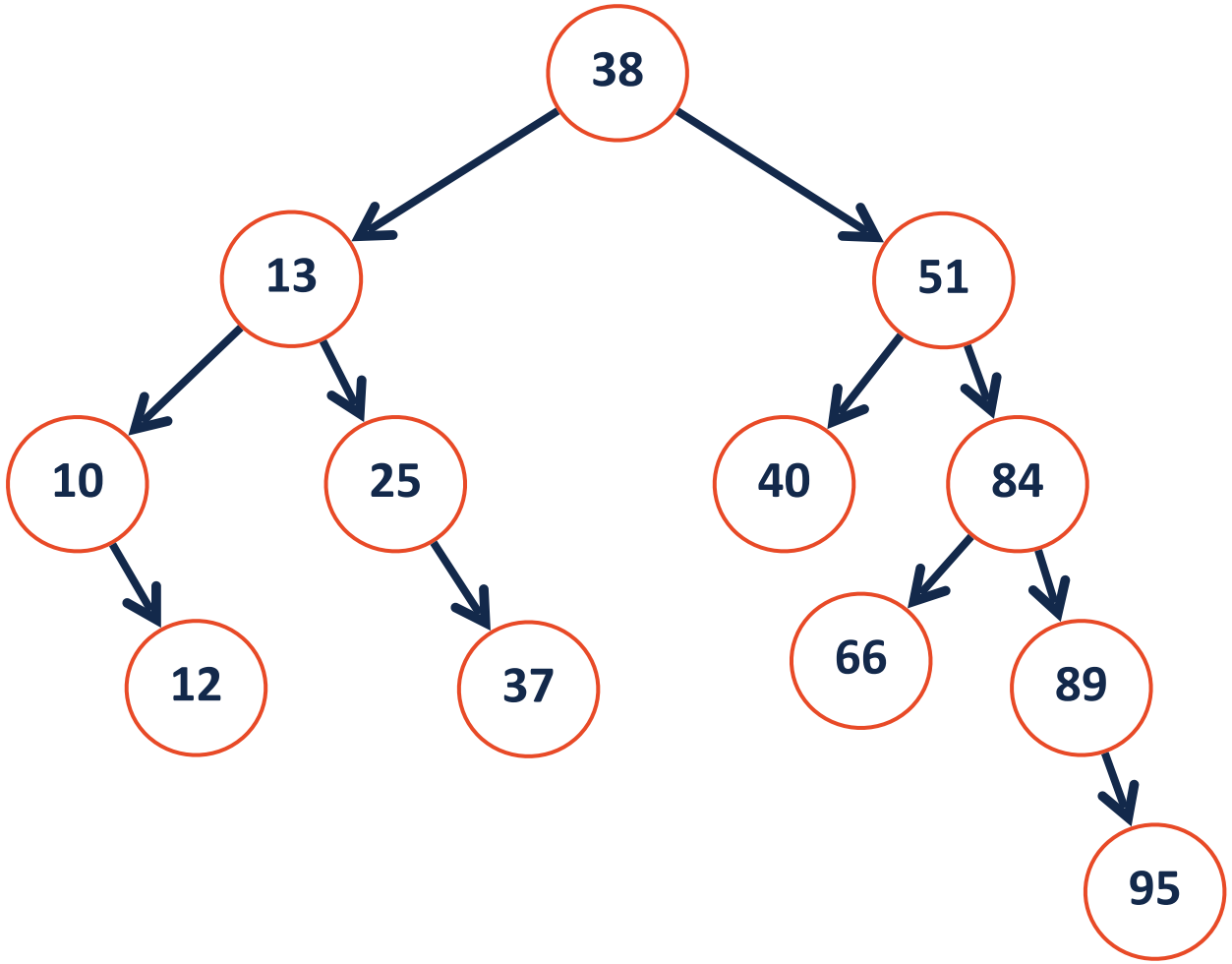
Flight Number → Arrival Information

URL → HTML Page

…

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

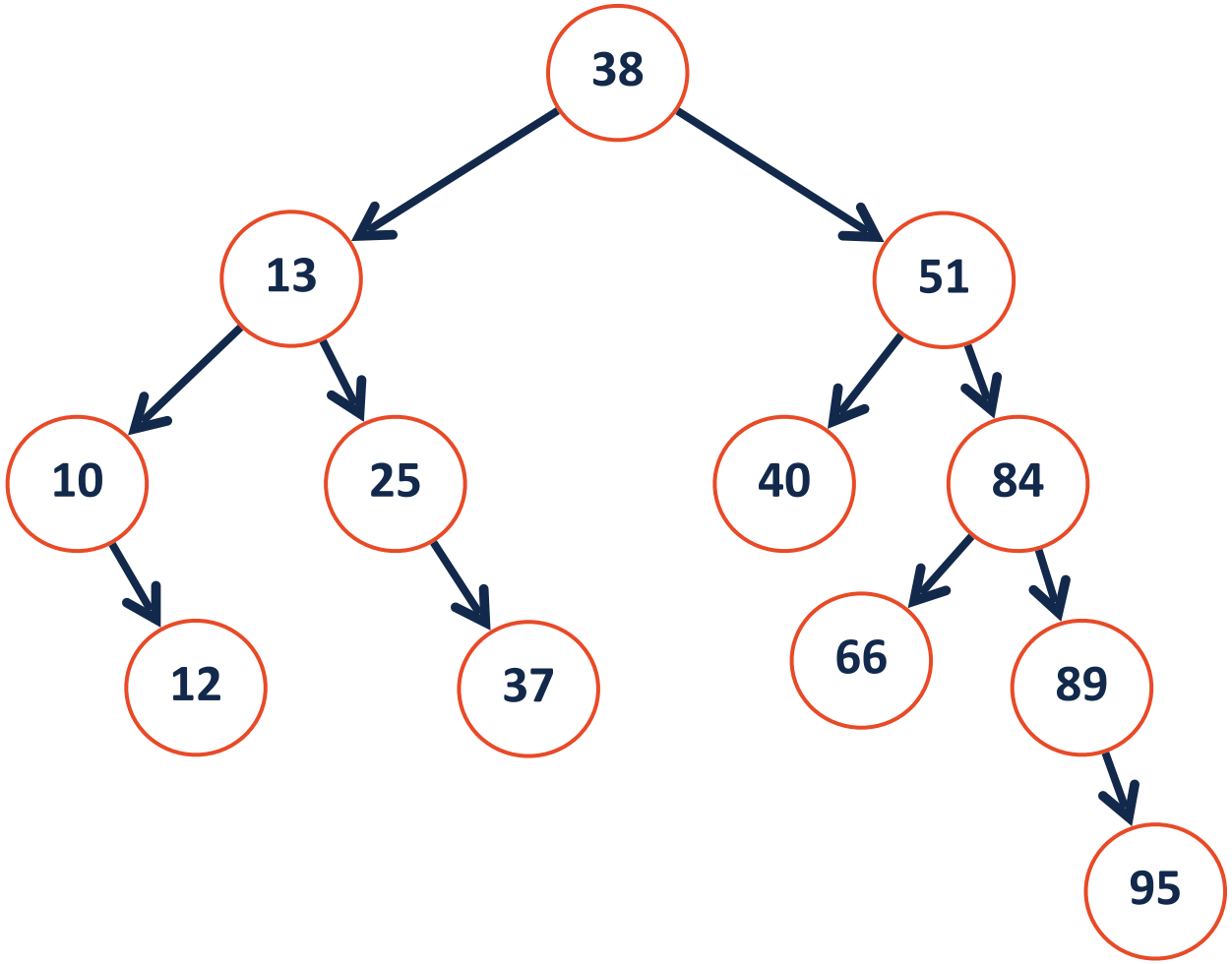
Traverse

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

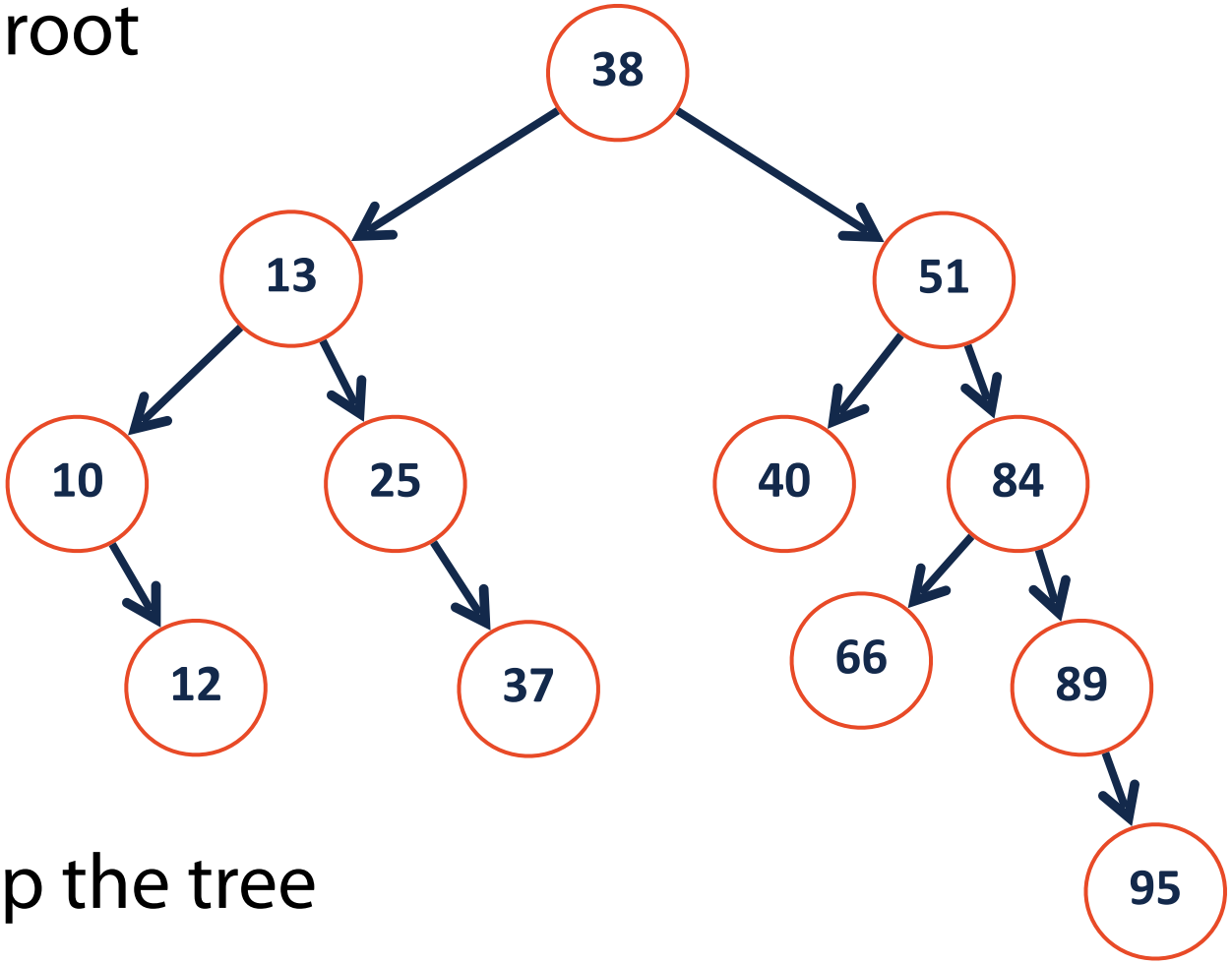
Recursion:

`tmp < query`, recurse right

`tmp > query`, recurse left

Combining:

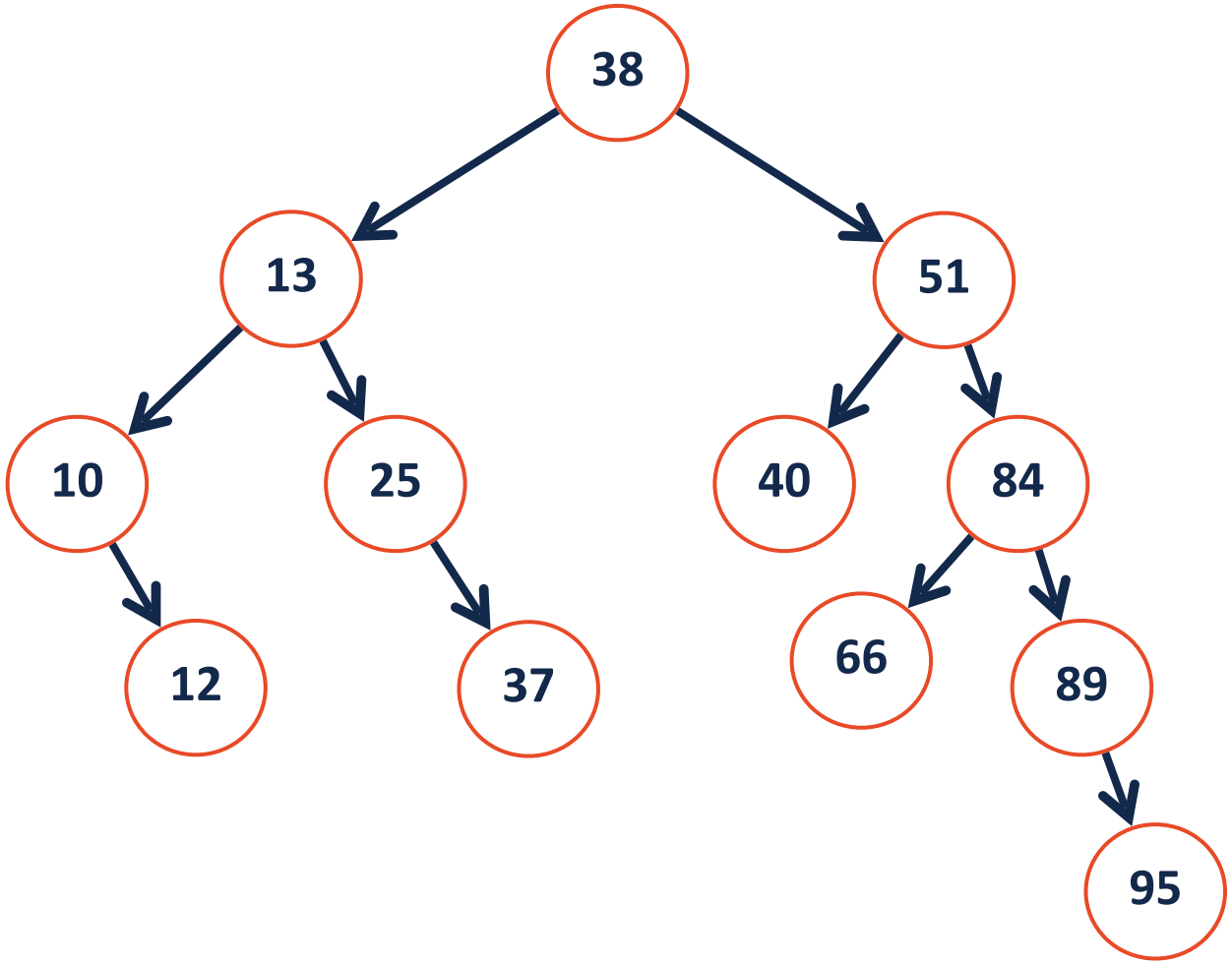
Return the recursive value back up the tree



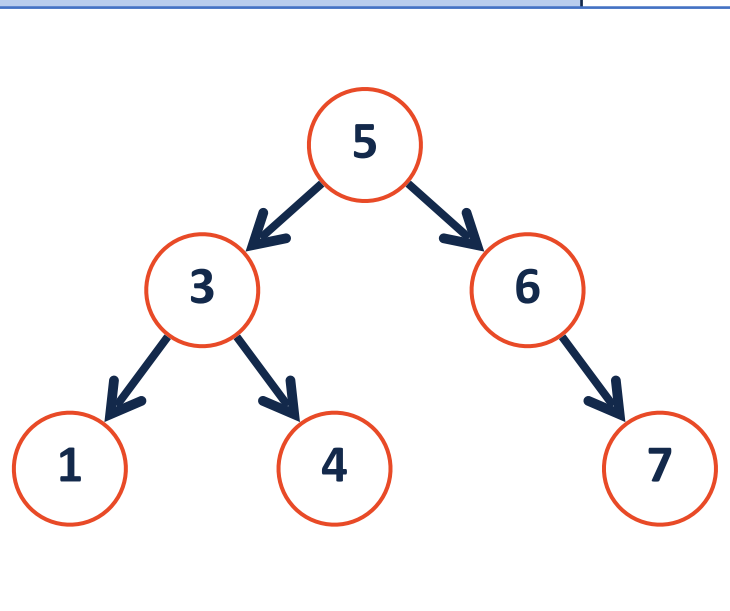
BST Find

`find(9)`

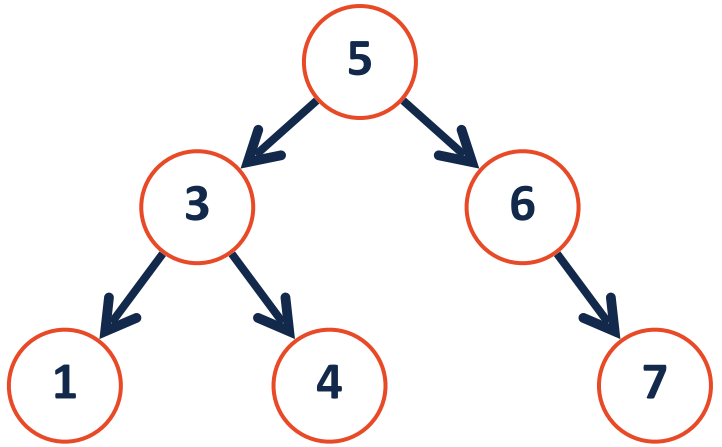
Make sure you can follow the search path!



```
1  template<typename K, typename V>
2
3  ____________________________ _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 }
```

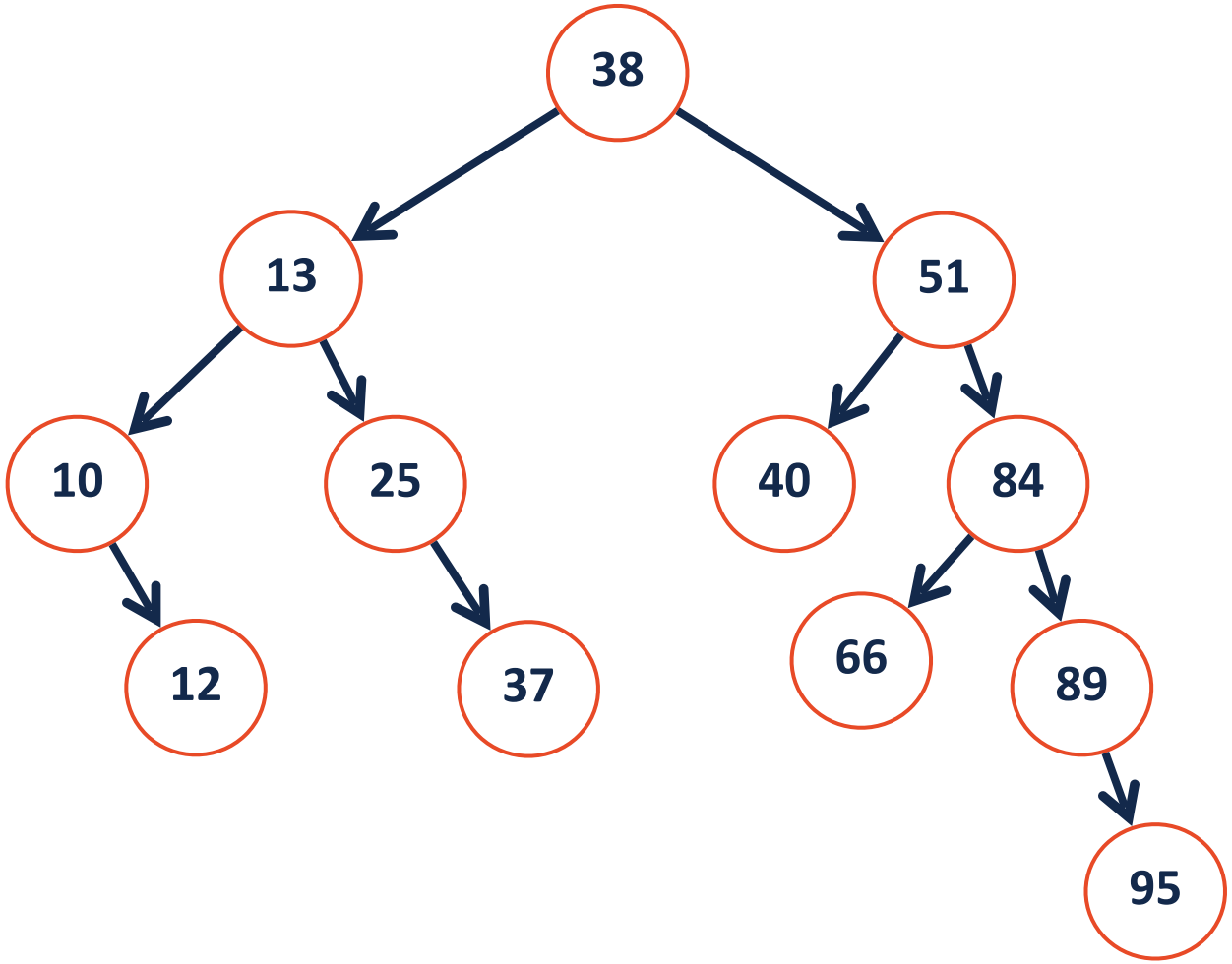


```
1  template<typename K, typename V>
2
3        TreeNode *&          _find(TreeNode *& root, const K & key) {
4
5
6  // Base Case
7  if(root == nullptr || root->data == key){
8      return root;
9  }
10
11 // Recursive Step ("Combining step" is 'return')
12 if (root->data > key){
13     return _find(root->left,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



BST Insert

`insert(33, v)`

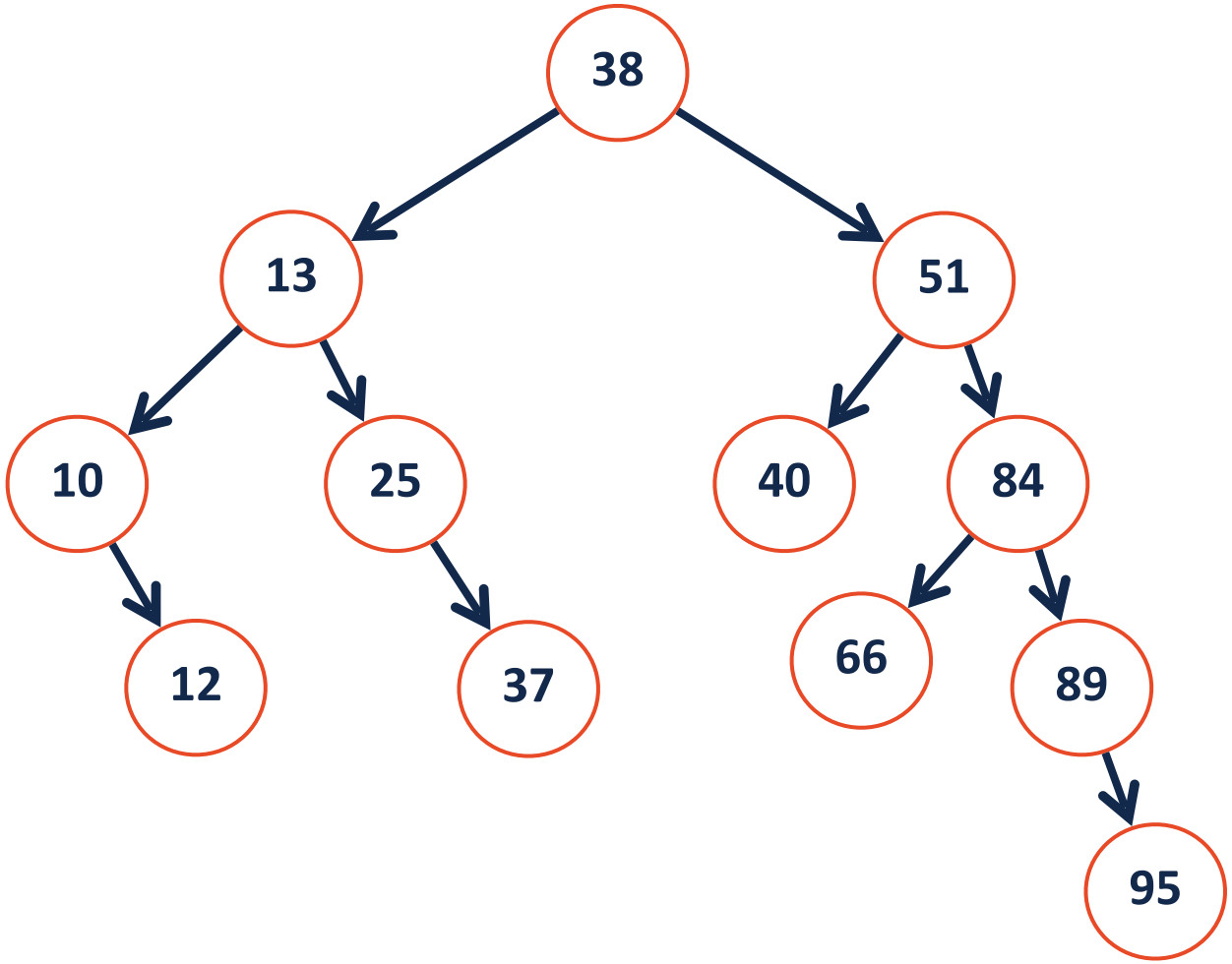


BST Insert

`insert(15, v)`

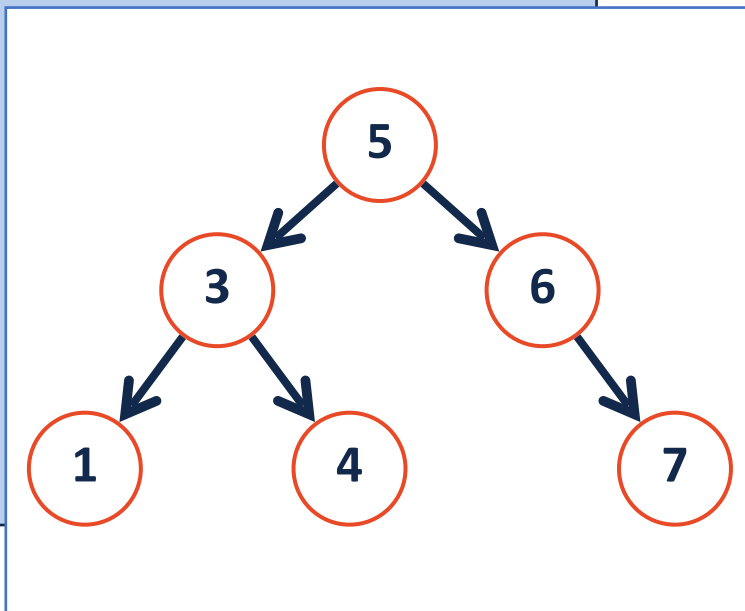
Find the insert location using `_find()`

Trivially insert at location



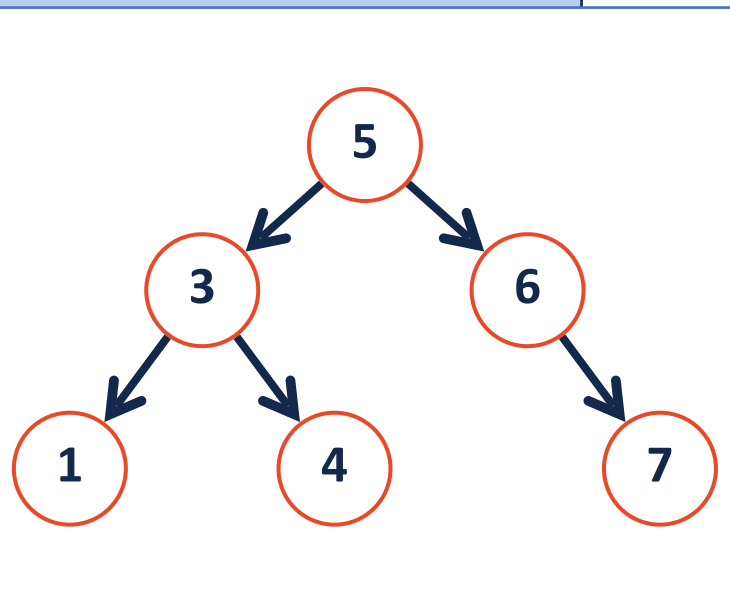
```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

```
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)
```