Data Structures

Binary Search Trees

CS 225
Harsha Tirumala

September 19, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

CS SAIL 2026

Illinois' Largest CS Outreach Event

APPLICATIONS OPEN NOW!

- LOGISTICS
- MARKETING
- DESIGN
- (SOFTWARE DEVELOPMENT)

DEADLINE SEPT 21, 11:59PM CT

Learning Objectives

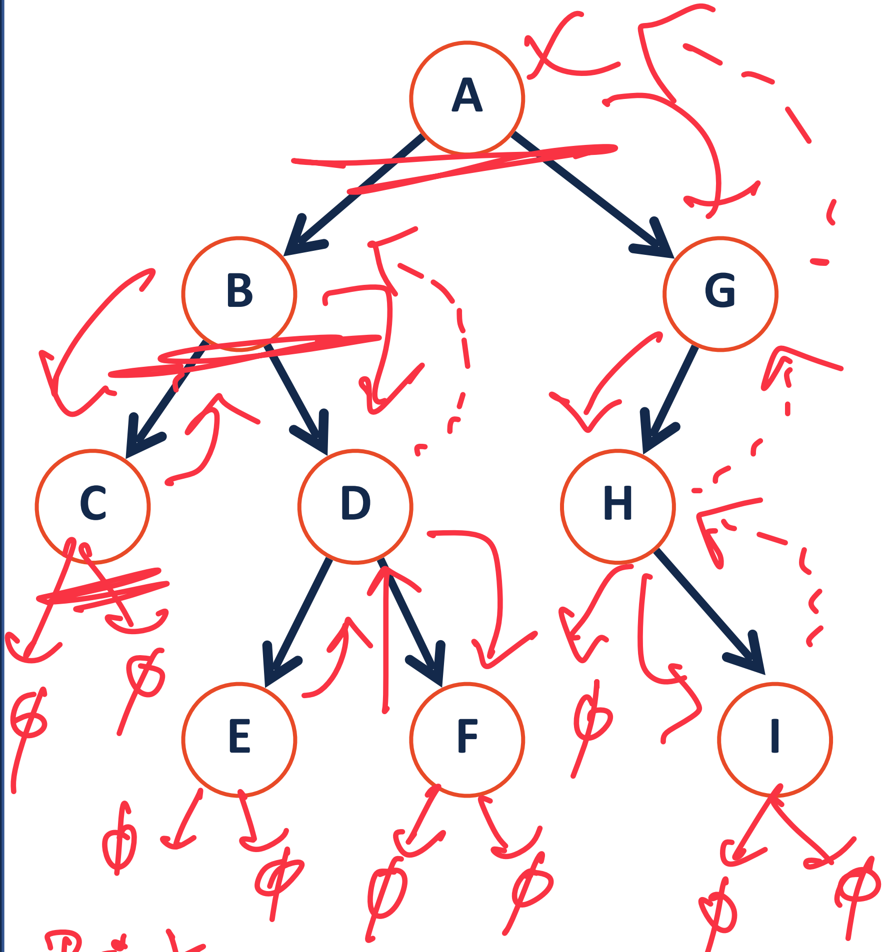
Tree Traversals - Pre, In, Post orders

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



Print

ABCDEFGHI

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

  if (root) {
     print(root)________________;

     preOrder(root->left);

     ____________________________;

     preOrder(root->right);

     ____________________________;

  }



}
```

pseudo code

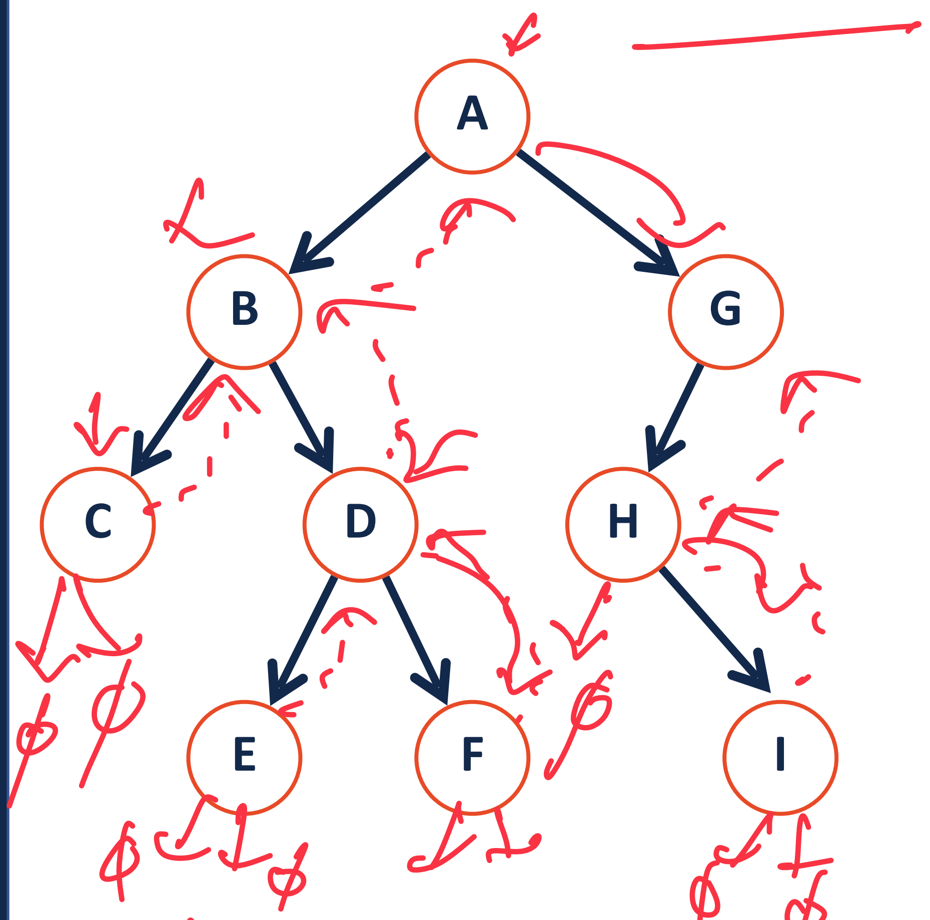
data

preorder(ϕ)

preorder(ϕ)

O(n)

Traversals - In order



Print

CBEDFAHIG

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

  if (root) {

      ______________________________;

      __in__Order(root->left);

      __print(root)_________________;

      __in__Order(root->right);

      ______________________________;

  }




}
```

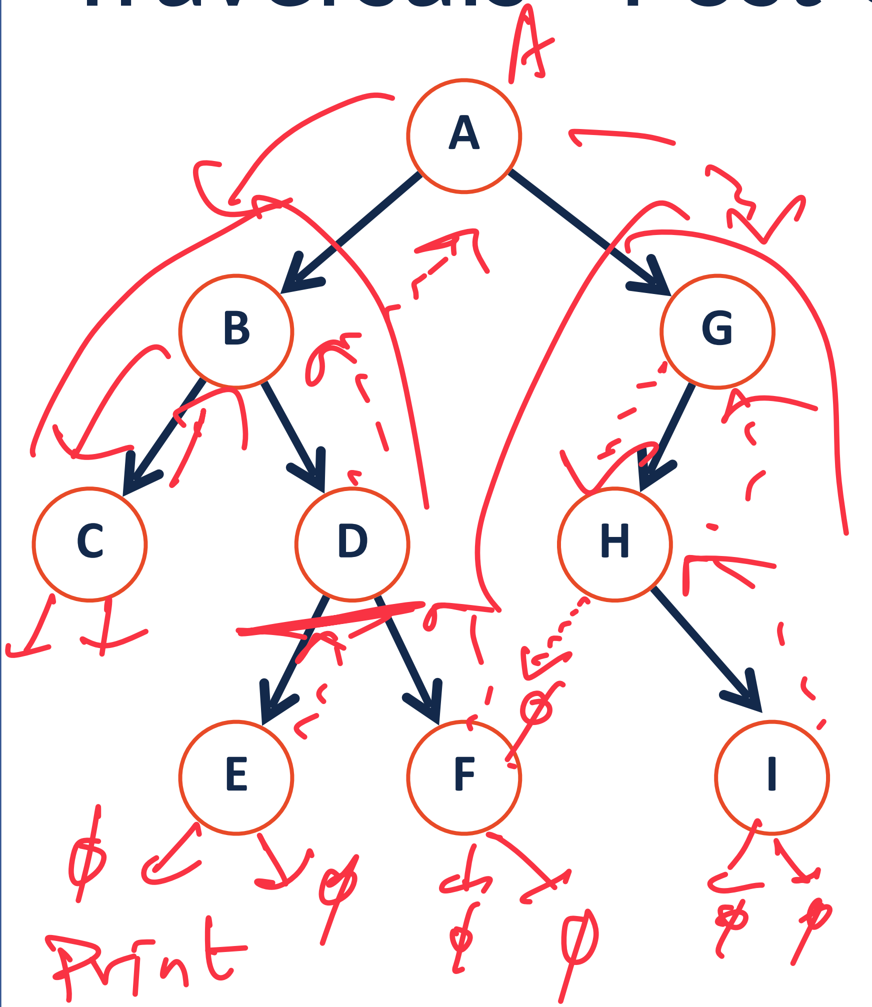
in(B)

in(C

in(D)

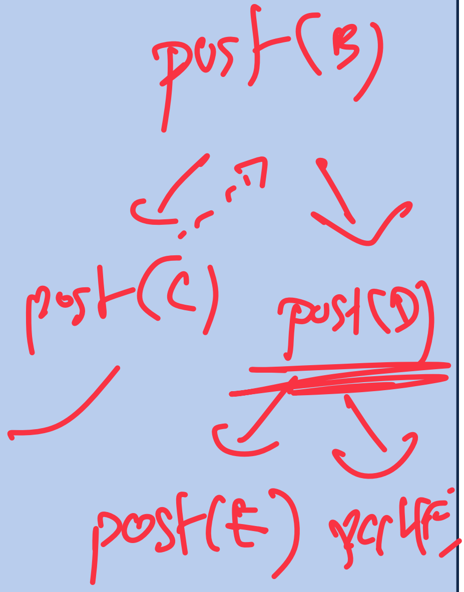
in(E)

Traversals - Post Order

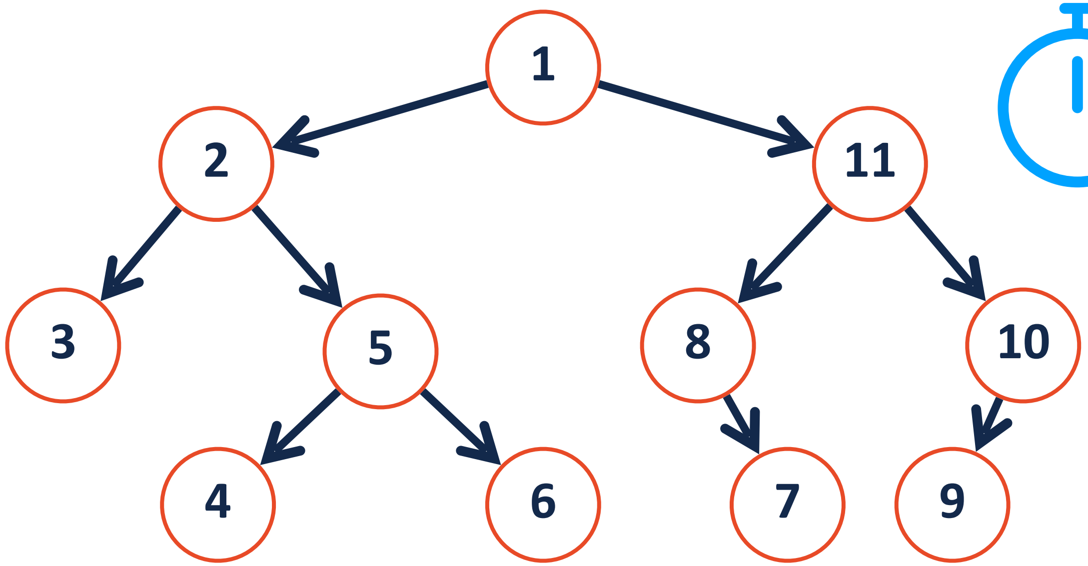


Print
CEFDBIHGA

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

   if (root) {

        ____________________________;
        postOrder(root->left);
        ____________________________;
        postOrder(root->right);
        print (root);

   }

}
```



Tree Traversals



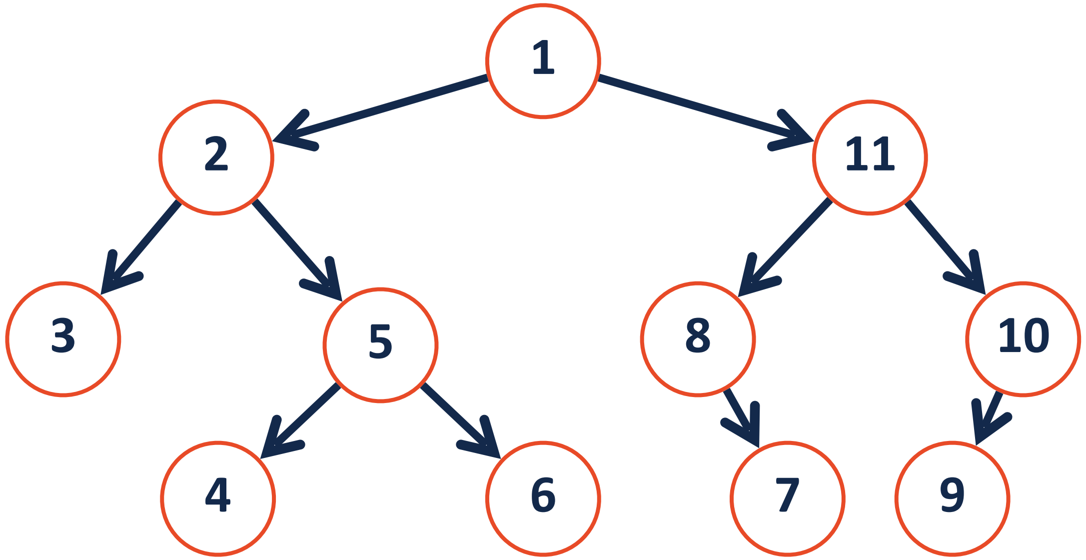
Pre-order:

left | right

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!

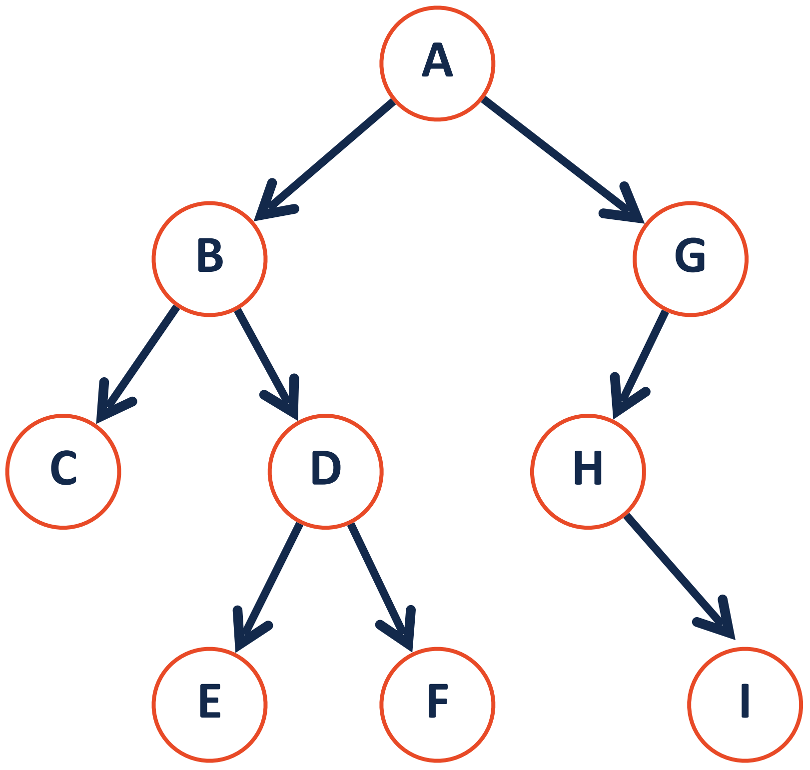
In-order: **3** 2 4 5 6 **1** 8 7 11 9 10

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



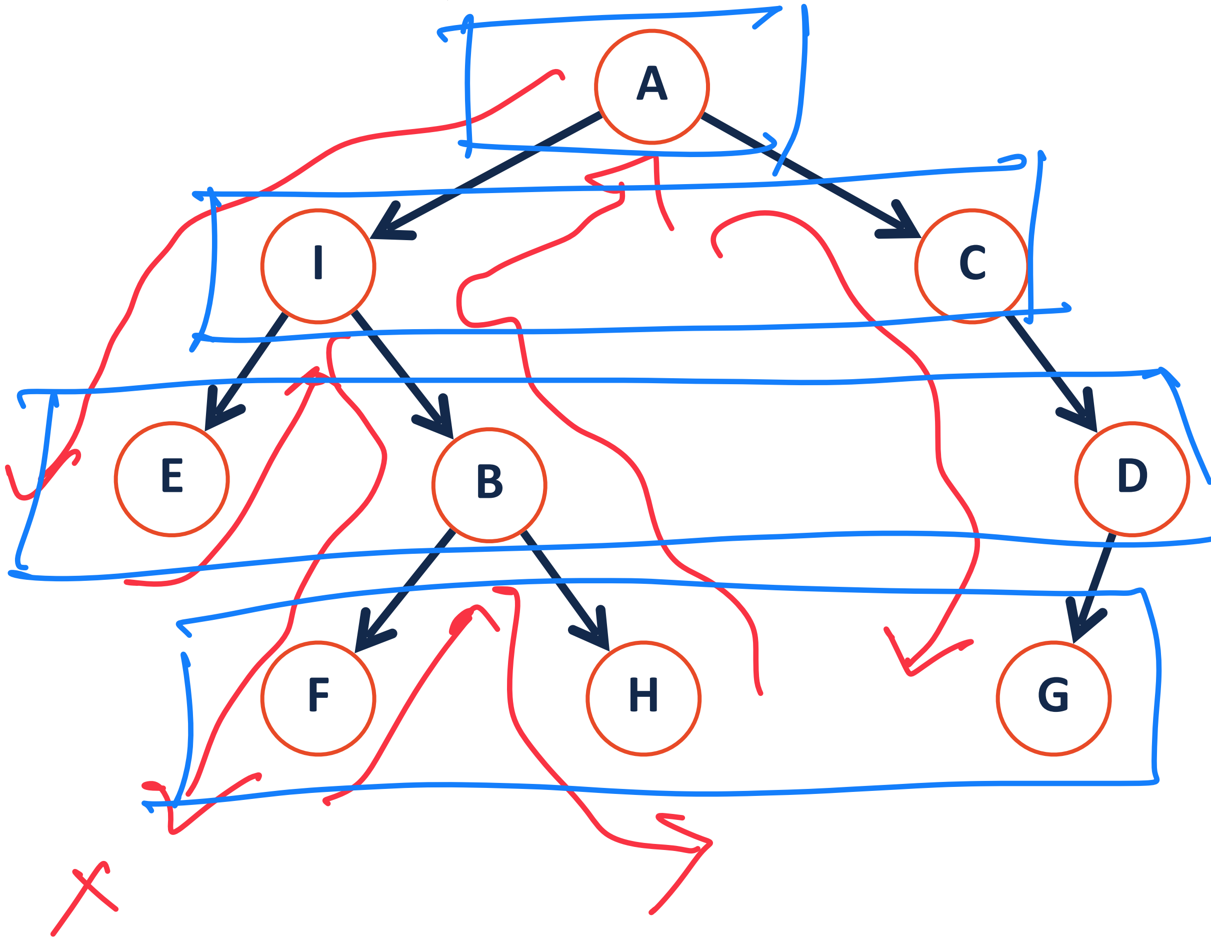
TRAVERSAL	SEARCH
Objective - Must visit every node	Objective - Needs to visit a specific node
Runtime is always O(n)	Runtime is O(n) for “unstructured” trees (We will see better runtimes later)
Types : Pre-order In-order Post-order	Types : Depth First Search Breadth First Search

Tree Search

There are two main approaches to searching a binary tree:

Depth first search

Breadth first search



Depth First Search

Explore as far along one path as possible before backtracking

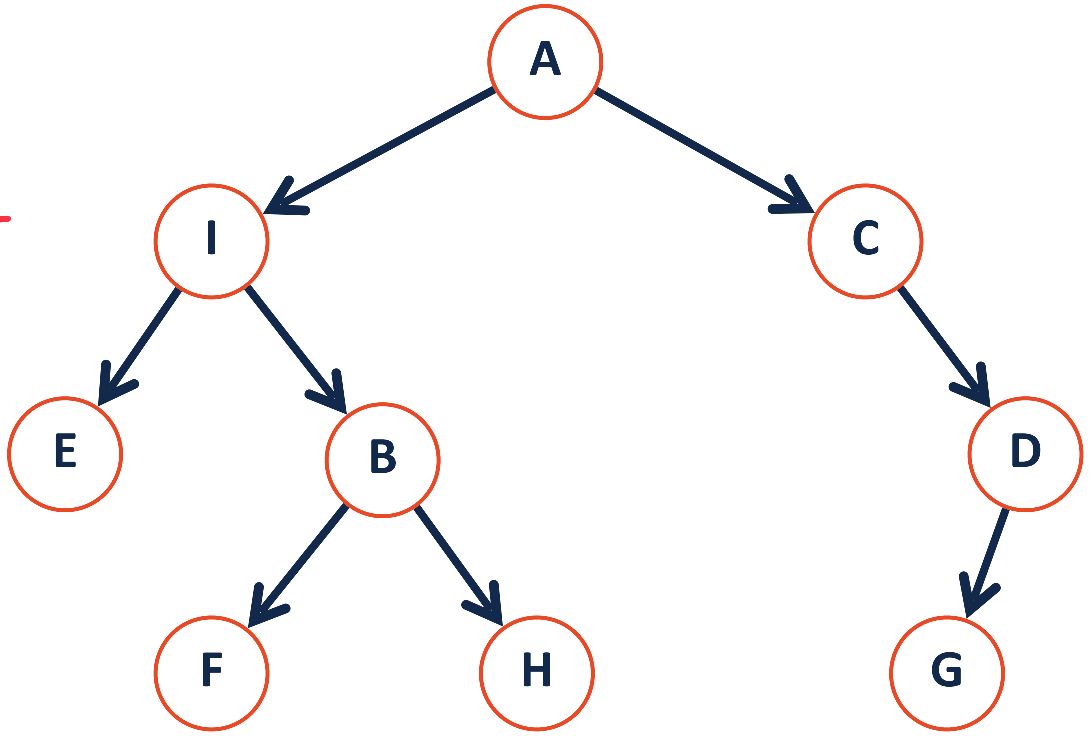
Requirements :

- Look at the end of one path before following another path
- Backtrack after each dead-end

Ideas :

Function calls?
Data Structure?

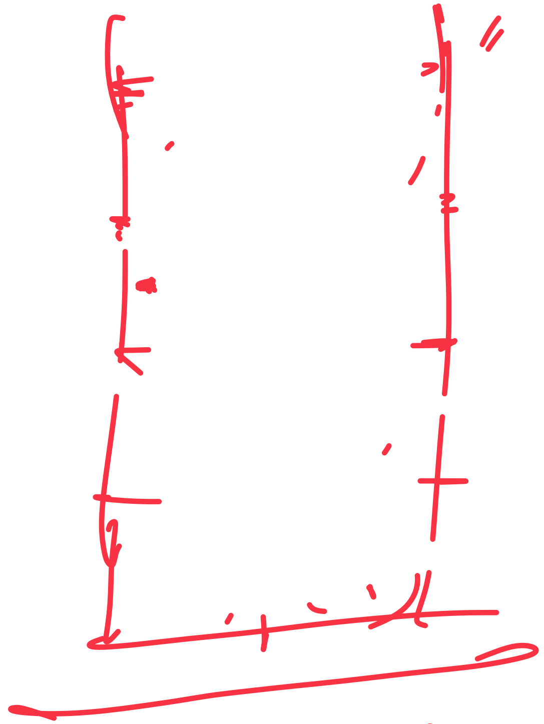
Stack

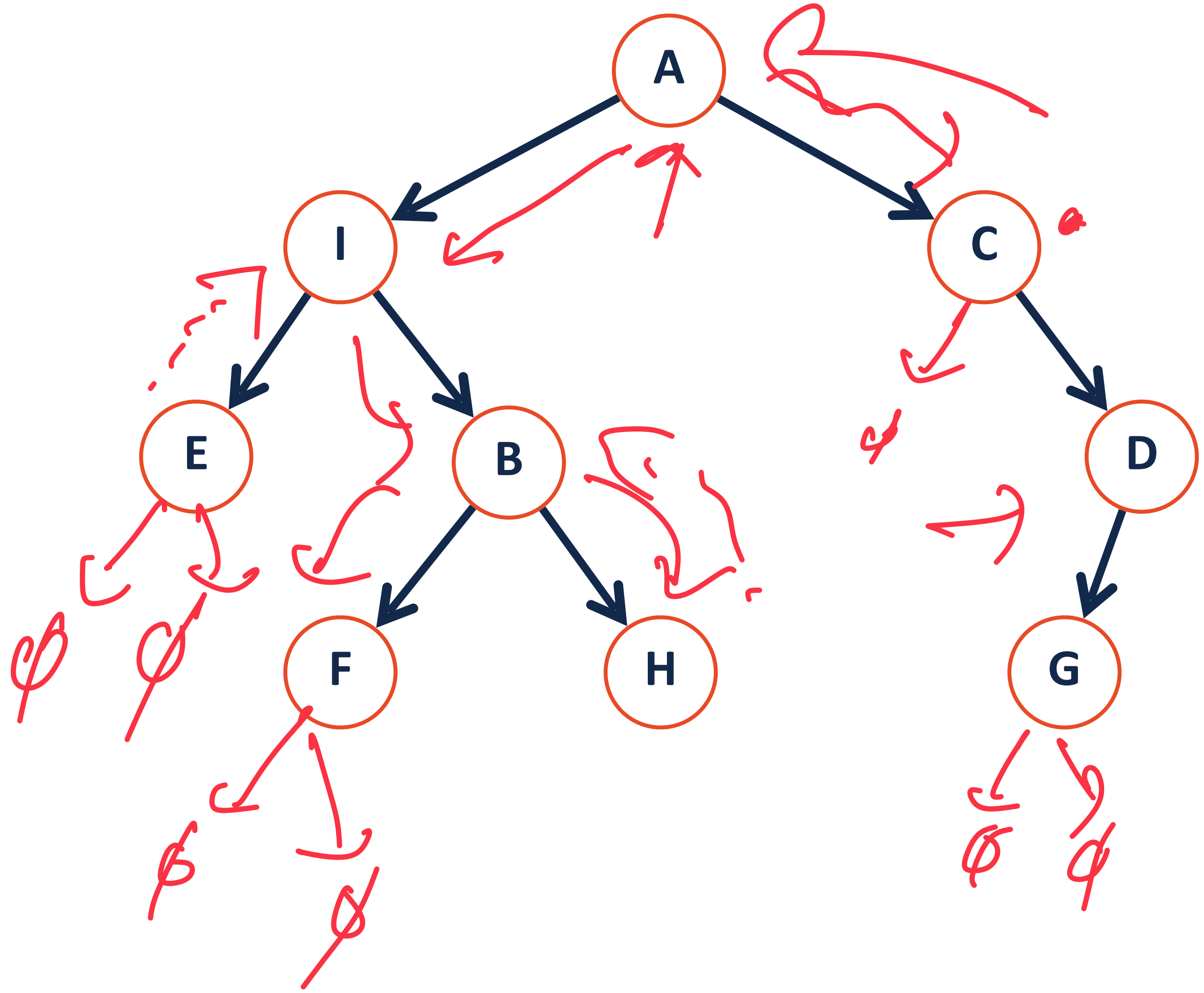


Depth First Search

left

Explore as far along one path as possible before backtracking





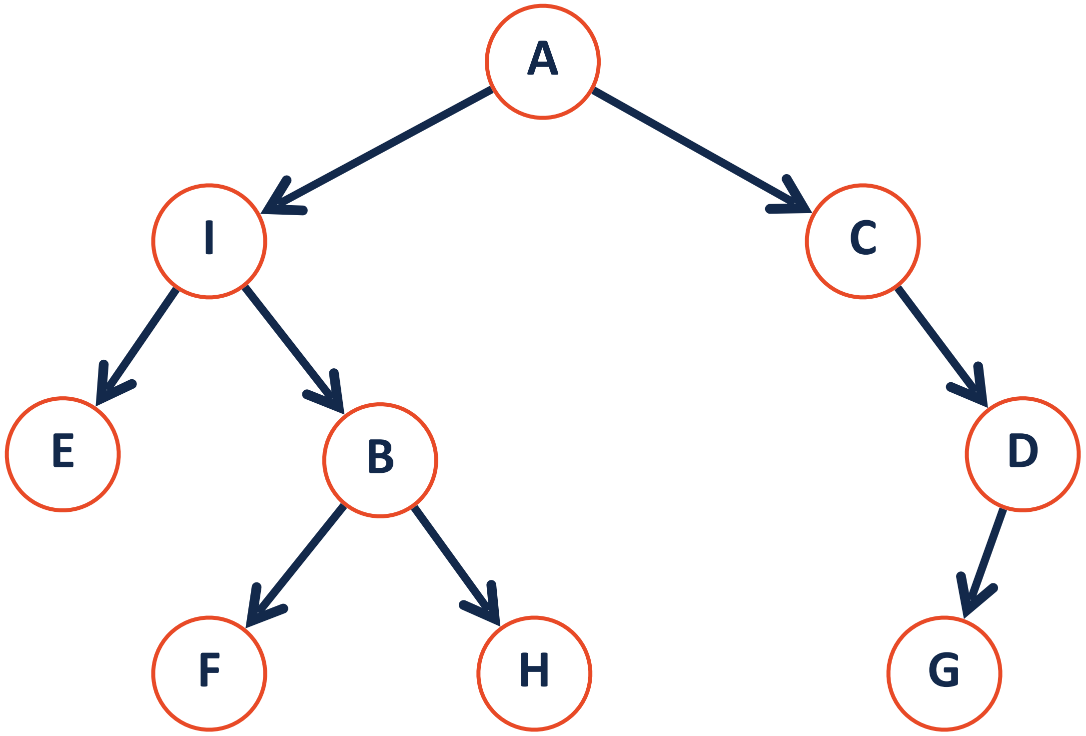
stack empty

DONE

Depth First Search

Explore as far along one path as possible before backtracking

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

  if (root) {

      Print root data;

      DFS_explore(root->left);

      DFS_explore(root->right);

     Return;

  }

Return;

}
```



O(h)

Breadth First Search

Fully explore level i before exploring level i+1

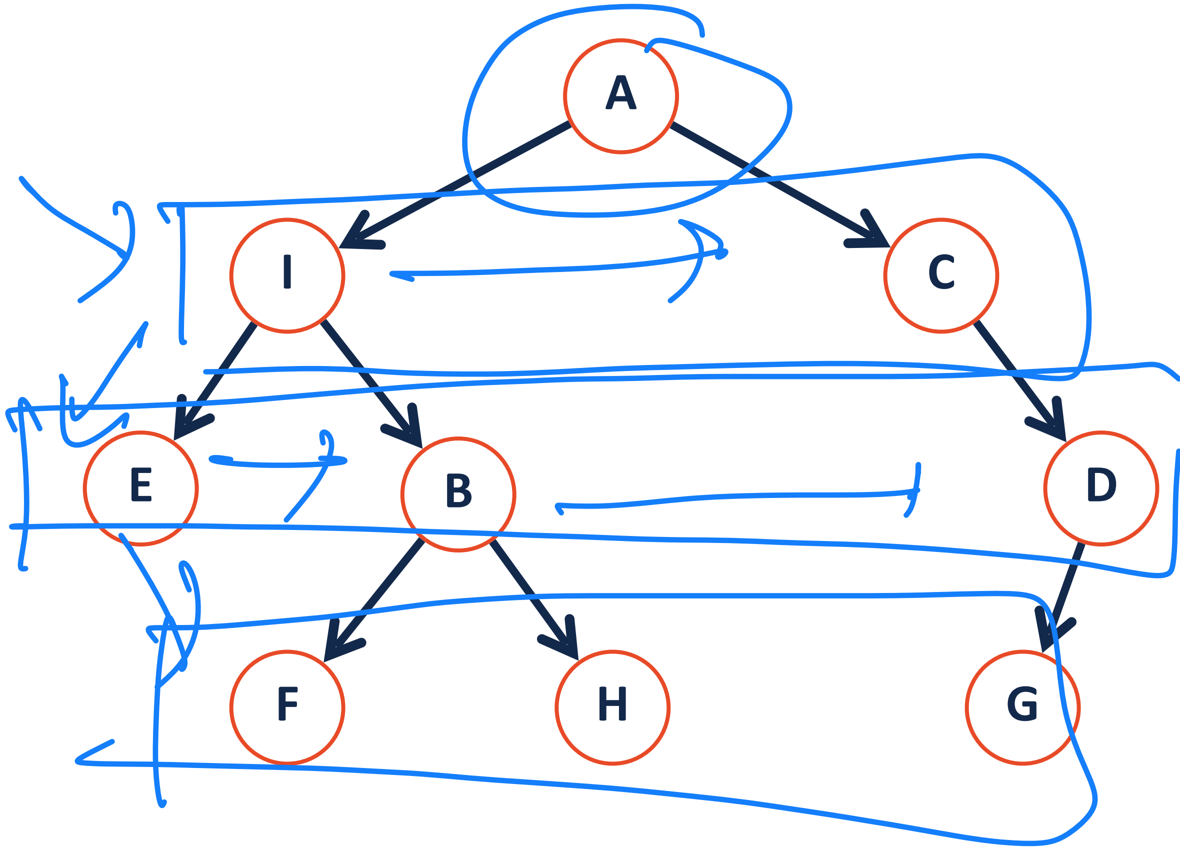
Requirements :

- Start from root - then visit level 1
- After level 1, visit level 2.
  Continue until you reach the bottom.

Ideas :

Function calls?
Data Structure?

Queue!



Breadth First Search

Fully explore level i before exploring level i+1

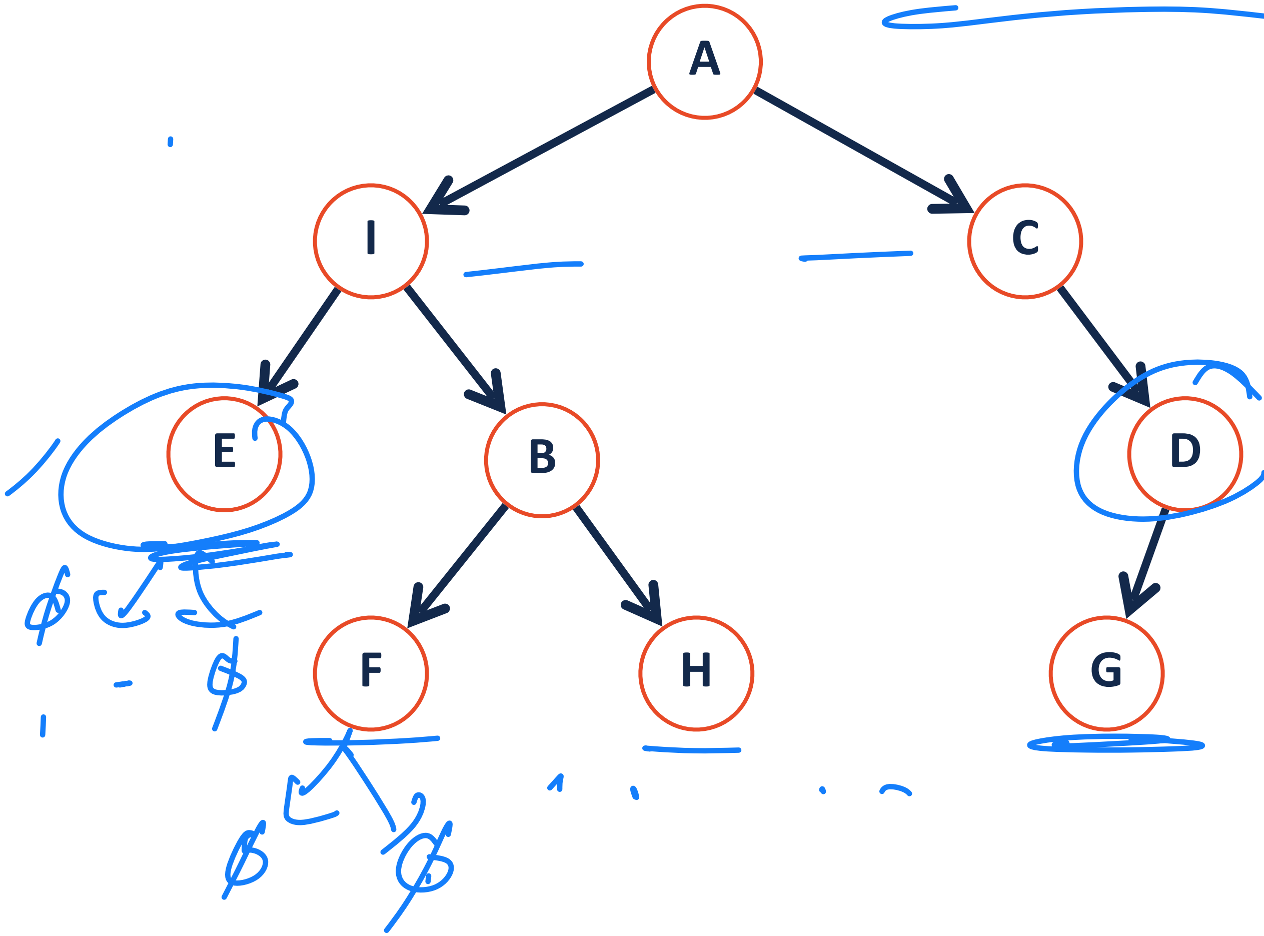
O(n)

B | D

D | F | H

G

level 4

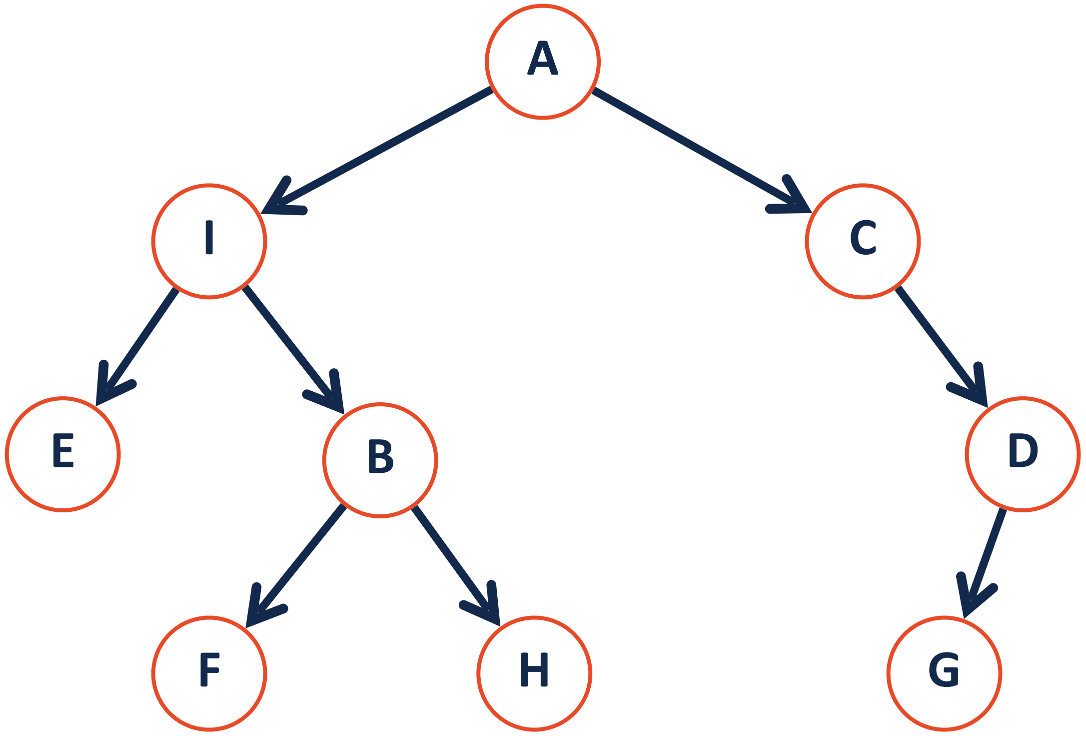


A I C E B D F H G

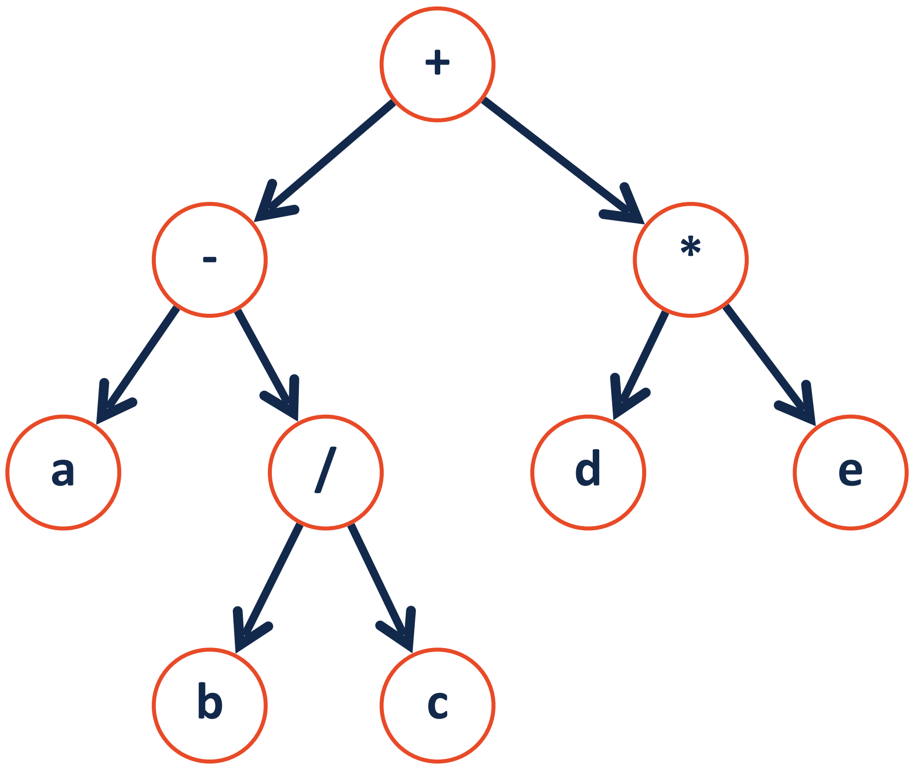
Breadth First Search

Fully explore level i before exploring level i+1

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

    Queue<TreeNode*> q;
    q.enqueue(root);

    while( q.empty() == False){

        TreeNode* temp = q.head();
        process(temp);  //can be print

        q.dequeue();

        q.enqueue(temp->left);
        q.enqueue(temp->right);

    }
}
```



Level-Order Traversal




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

    Queue<TreeNode*> q;
    q.enqueue(root);

    while( q.empty() == False){

        TreeNode* temp = q.head();
        process(temp);

        q.dequeue();

        q.enqueue(temp->left);
        q.enqueue(temp->right);

    }
}
```

Tree Search Tradeoffs

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

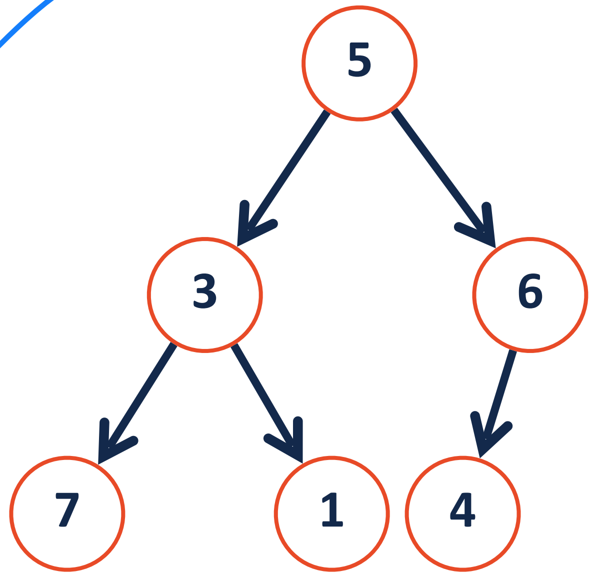
What do we trade in order to do so?

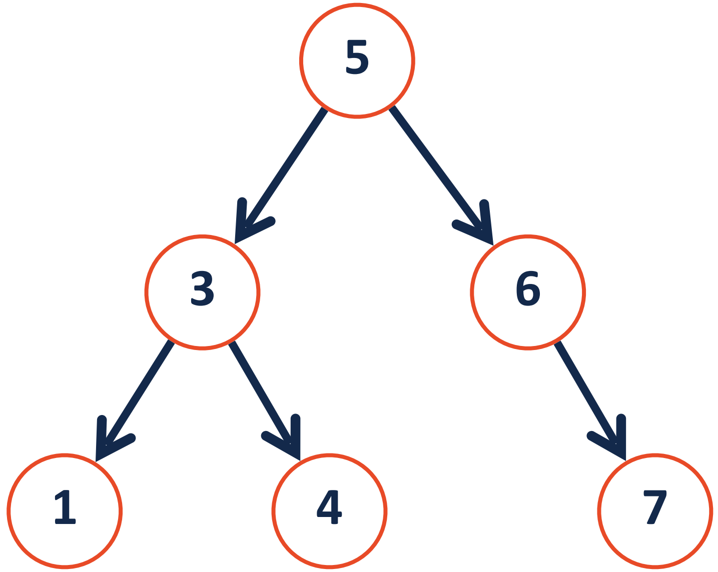
Improved search on a binary tree

5	3	6	7	1	4

1	3	4	5	6	7

2





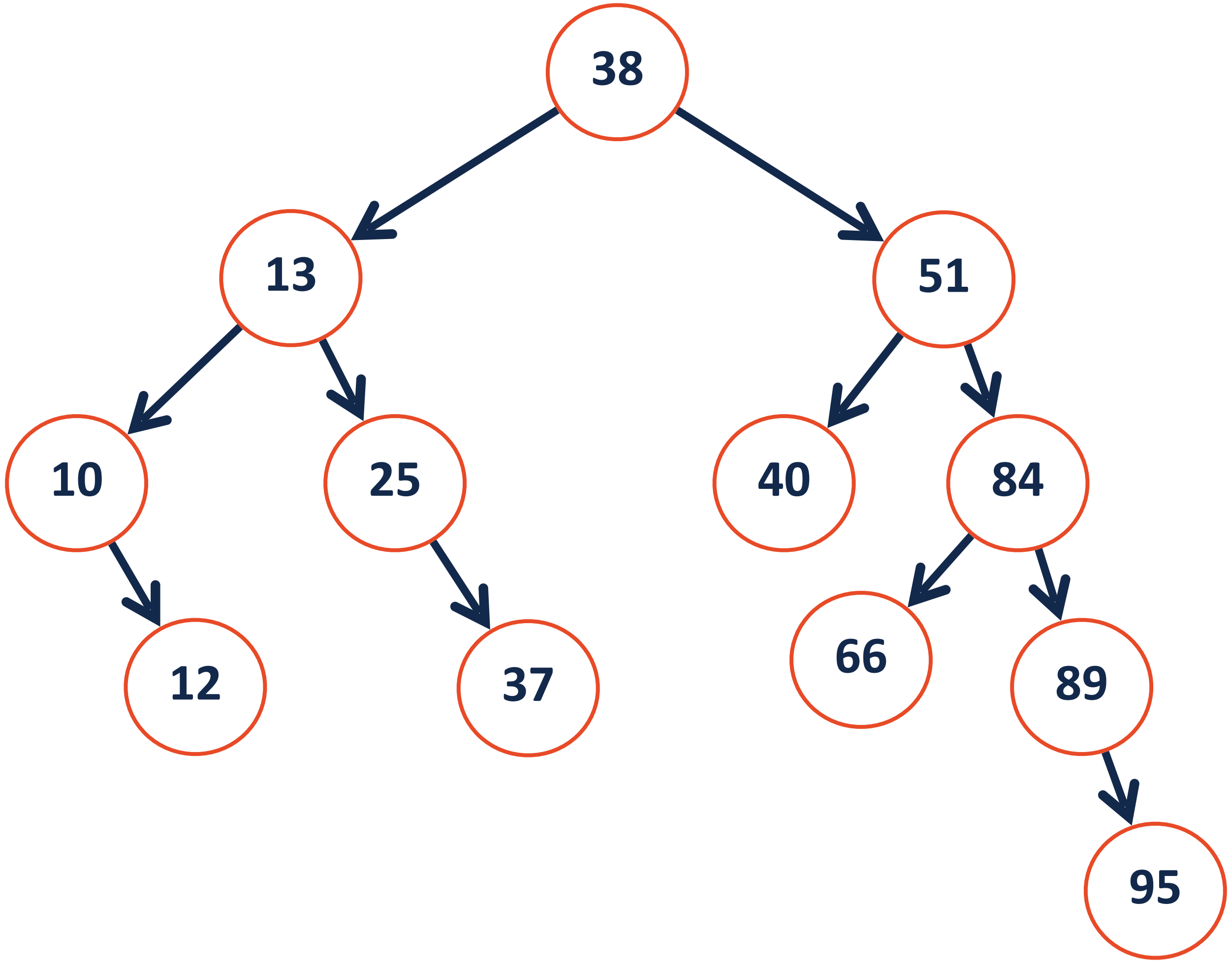
Dictionary ADT

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

```
#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

Insert

Remove

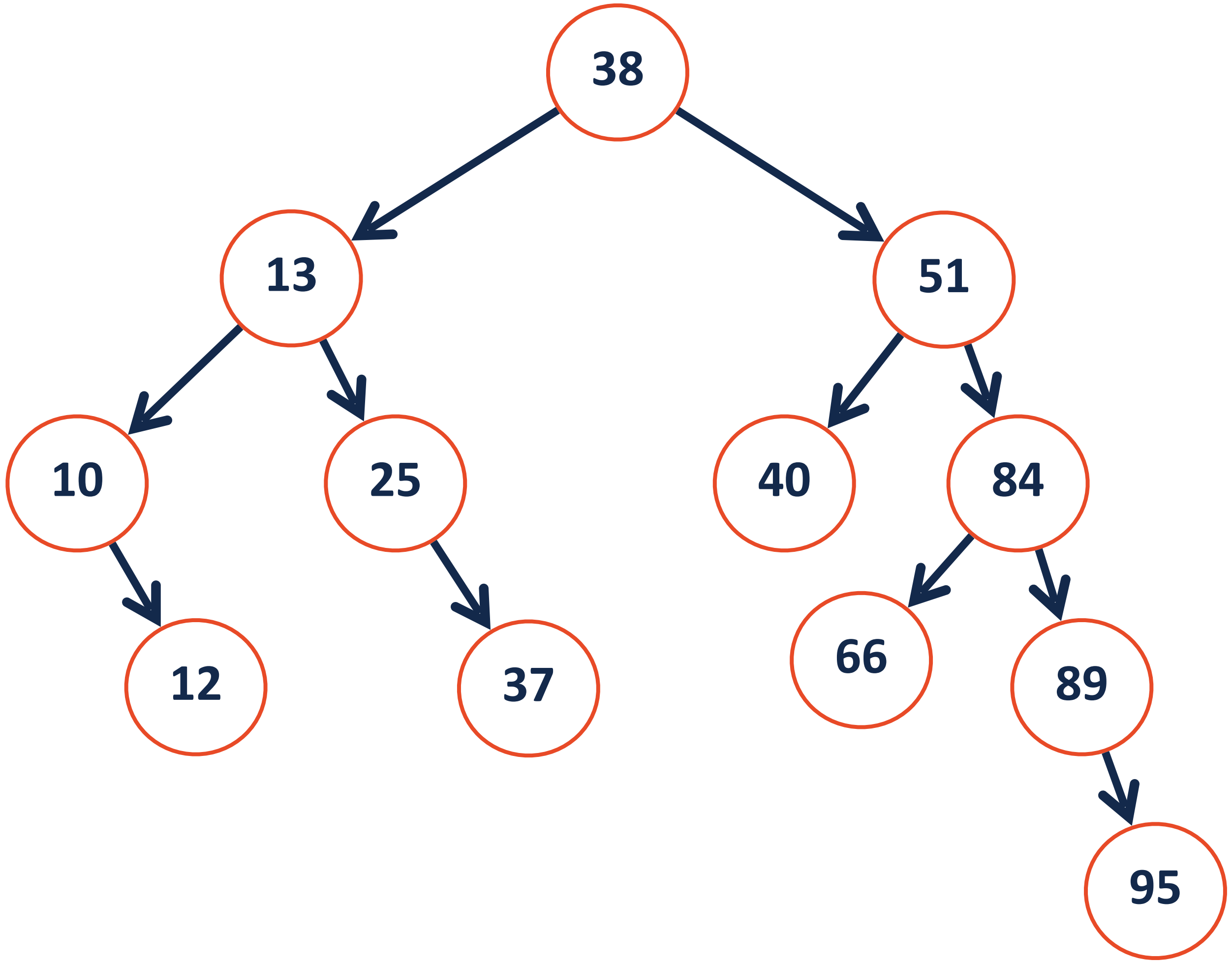
Traverse

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

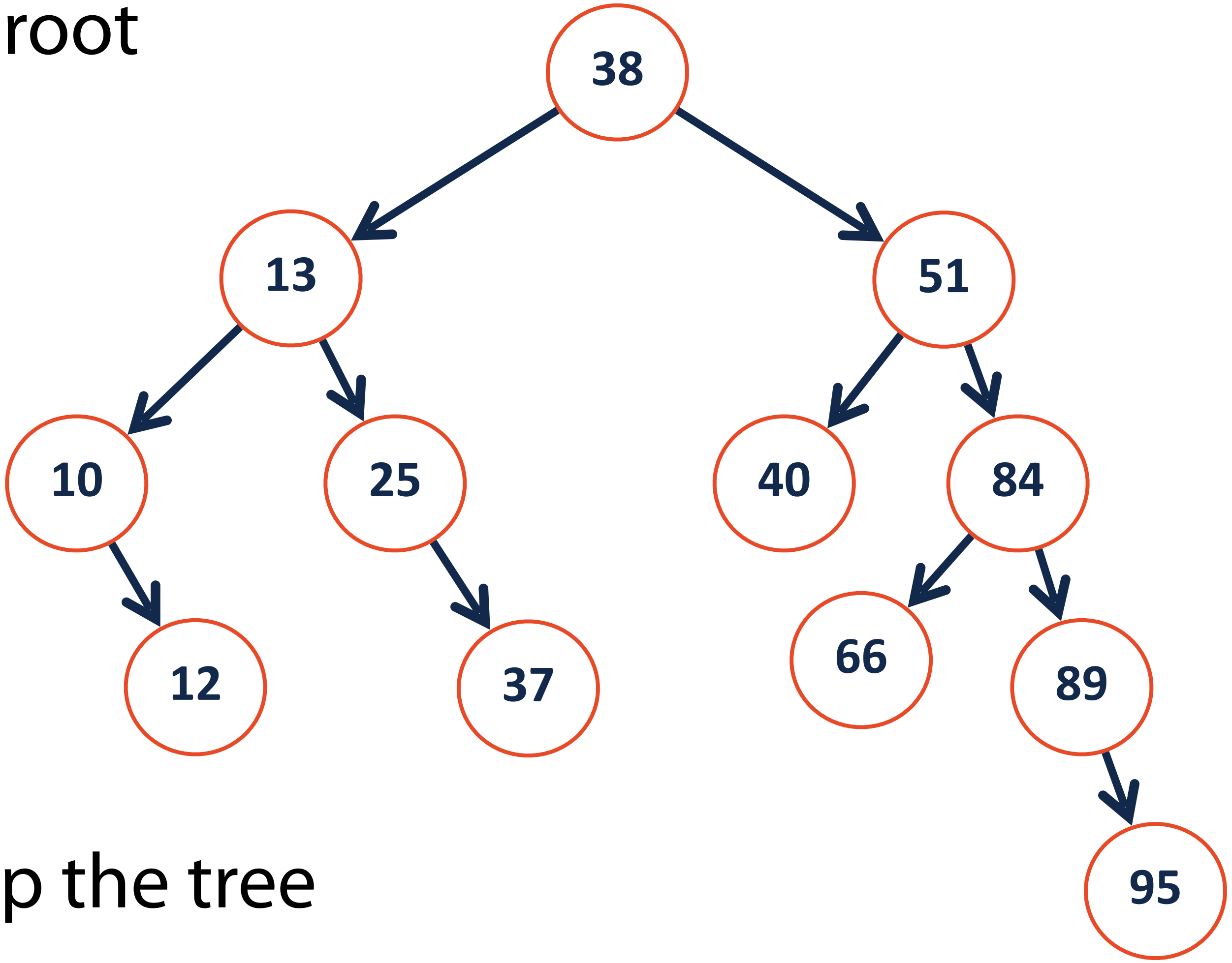
Recursion:

`tmp < query`, recurse right

`tmp > query`, recurse left

Combining:

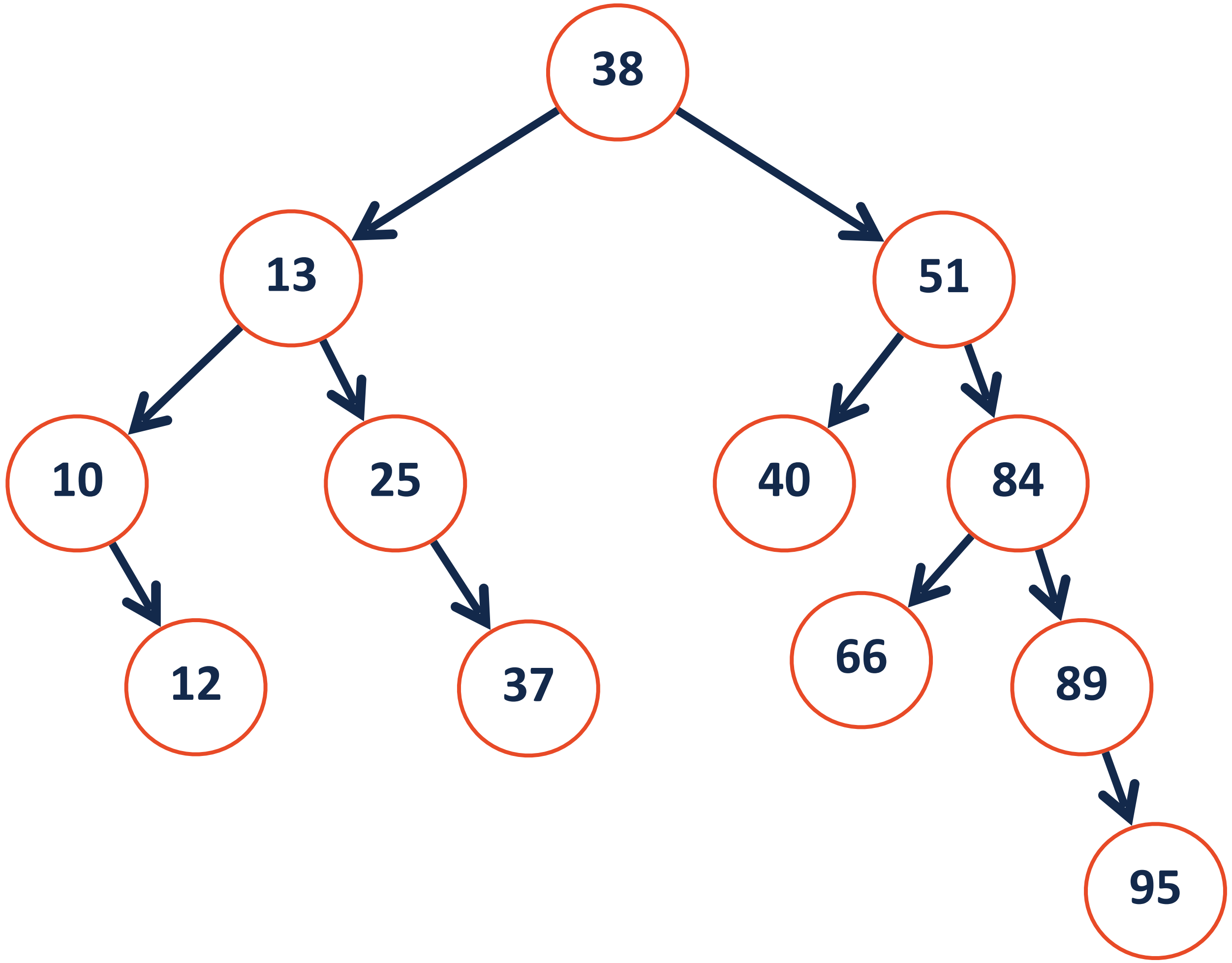
Return the recursive value back up the tree



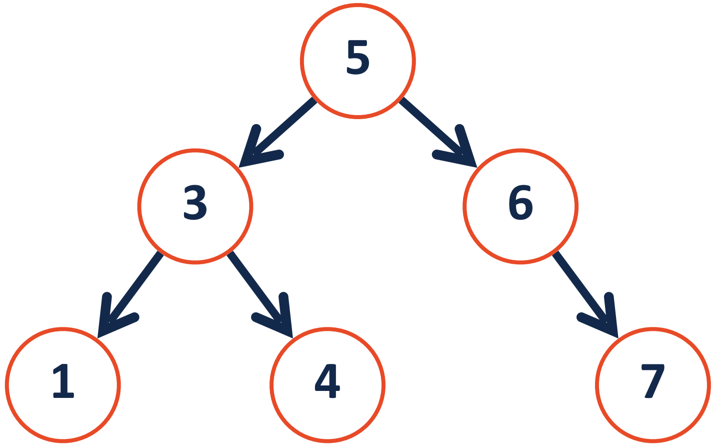
BST Find

`find(9)`

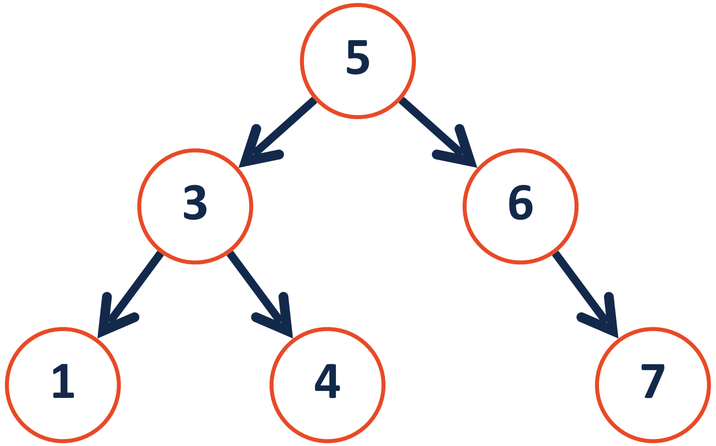
Make sure you can follow the search path!



```
template<typename K, typename V>

_______________________ _find(TreeNode *& root, const K & key) {



















}
```

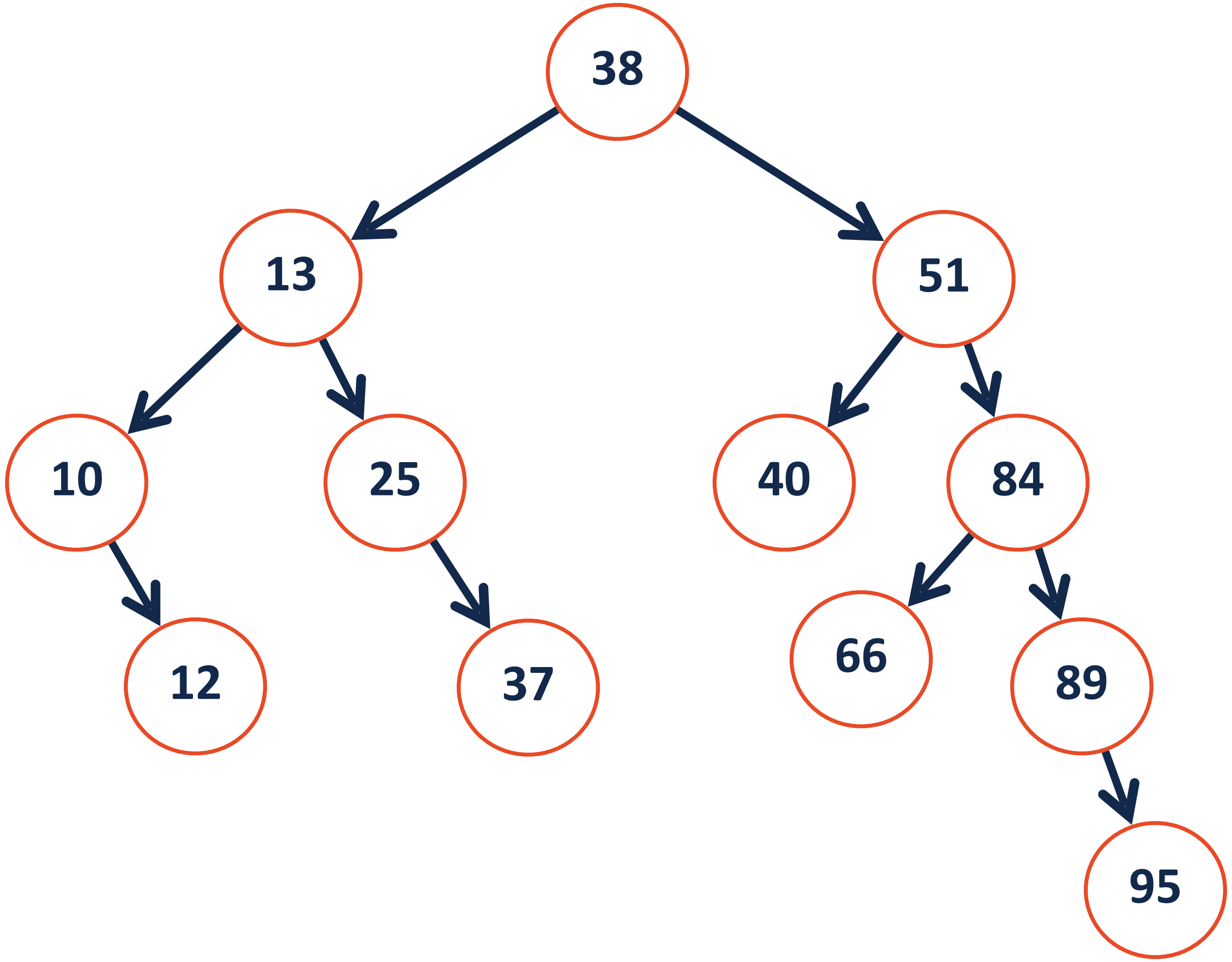


```
template<typename K, typename V>

______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);


}
```



BST Insert

`insert(33, v)`

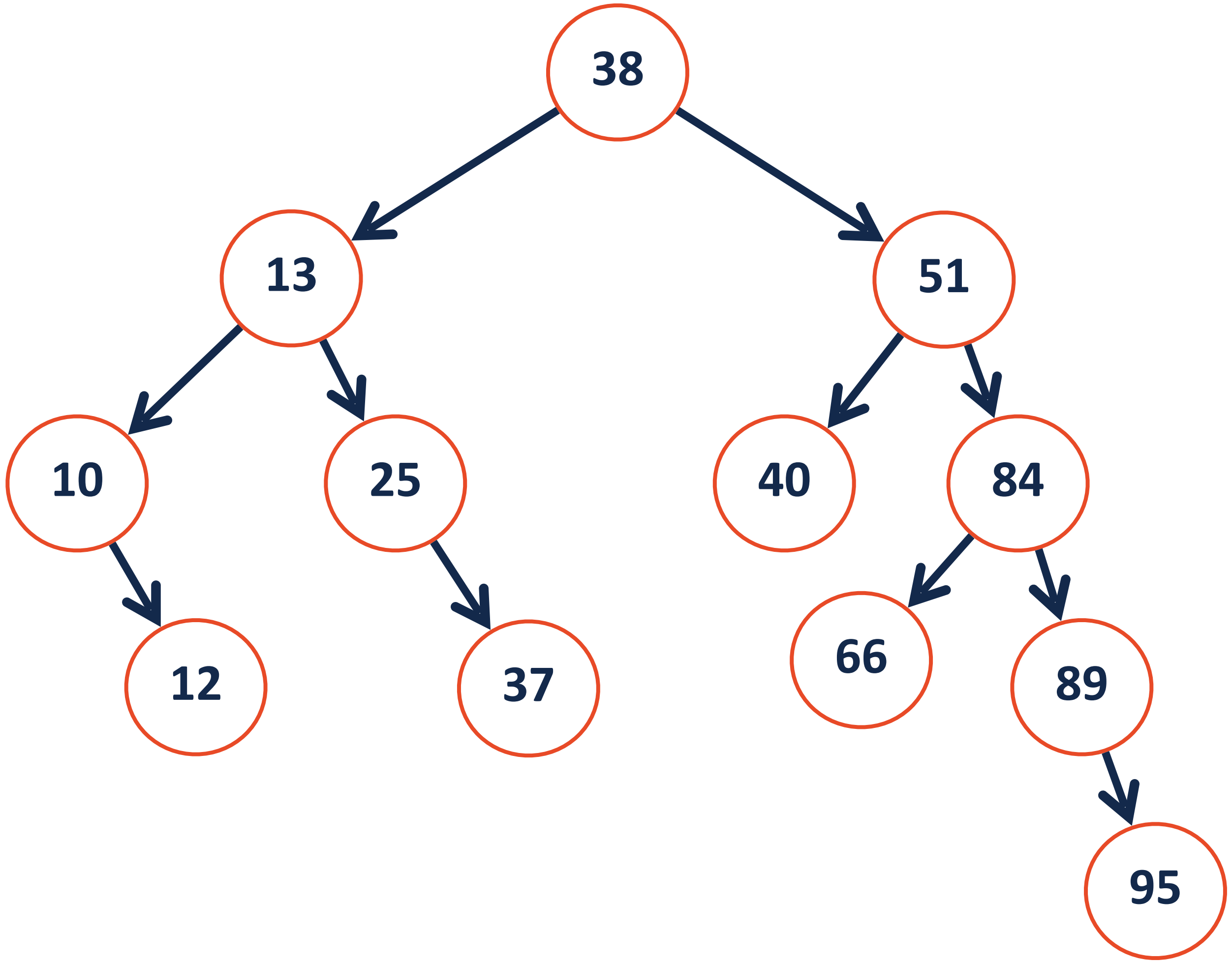


BST Insert

`insert(15, v)`

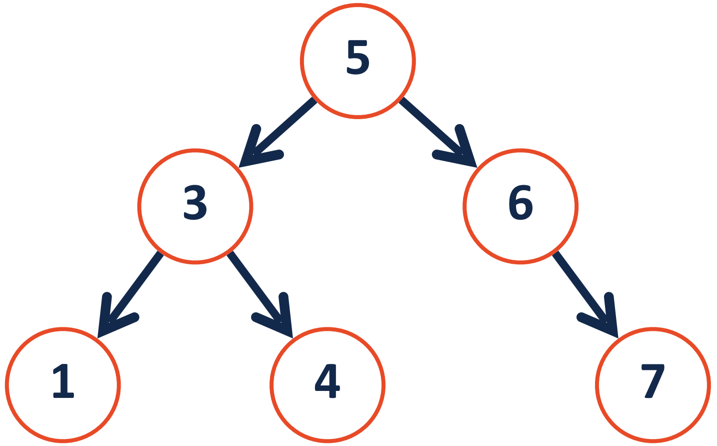
Find the insert location using `_find()`

Trivially insert at location



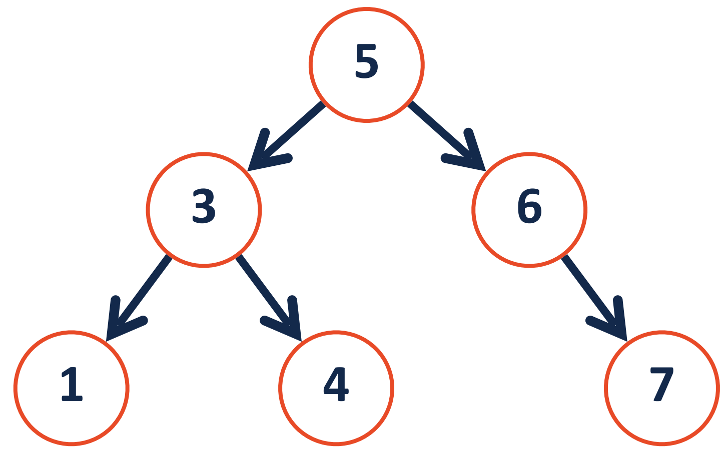
```
template<typename K, typename V>

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);
}

```

```
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);




}



```



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)`