Data Structures
Exam 3 Review

CS 225
Brad Solomon

March 13, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Spring Break Logistics

Nothing is due over spring break

Spring break doesn’t count as a ‘week’ for assignments

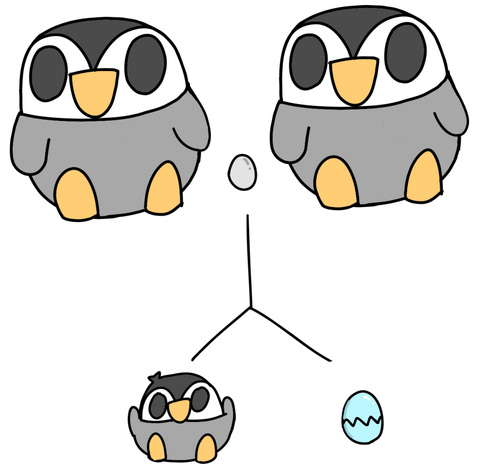
No office hours over spring break

Learning Objectives

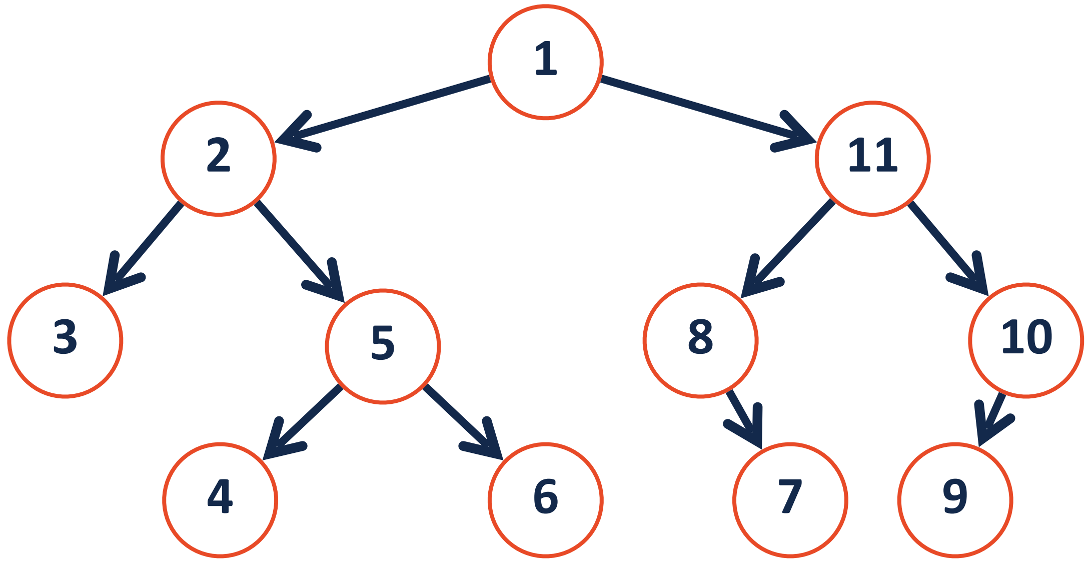
Review concepts on exam 3

Enjoy your spring break!

Trees



Tree Traversals

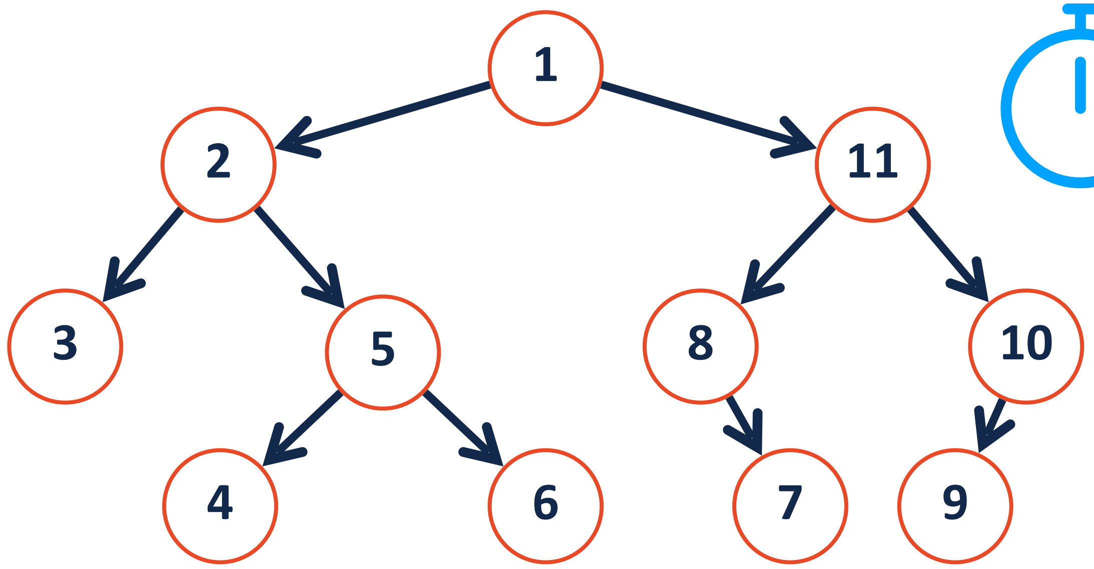


Pre-order:

In-order:

Post-order:

Tree Traversals



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

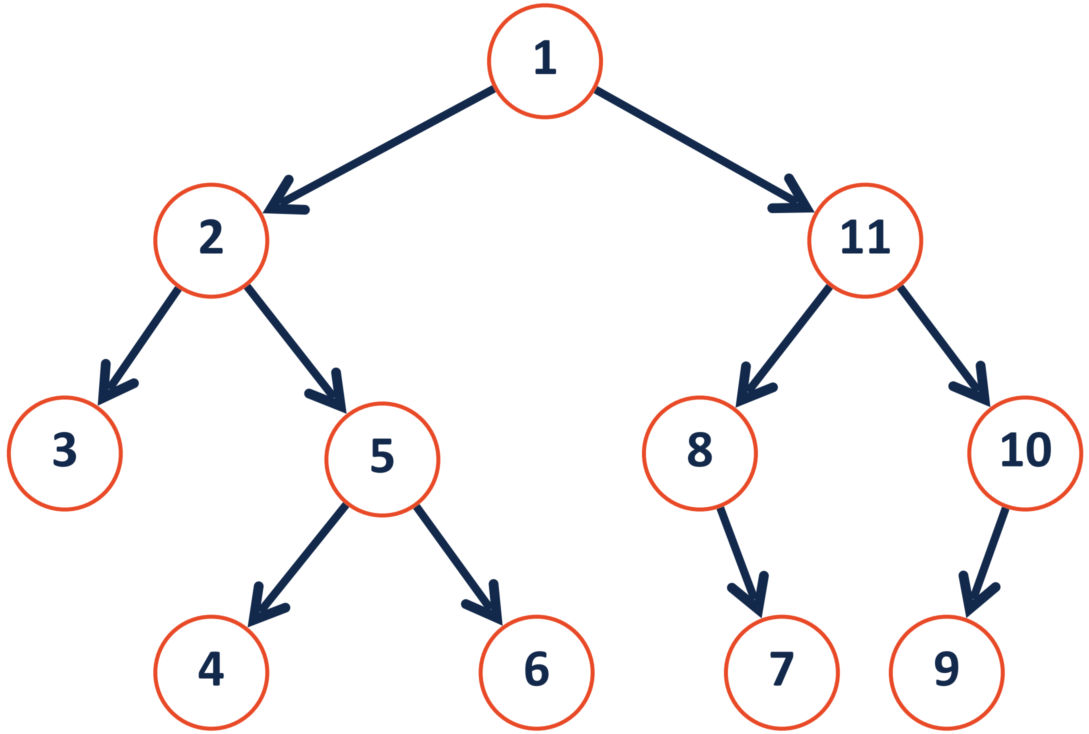
In-order: 3, 2, 4, 5, 6, **1**, 8, 7, 11, 9, 10

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

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)
  Print tmp
  Push tmp->right to stack
  Push tmp->left to stack
```



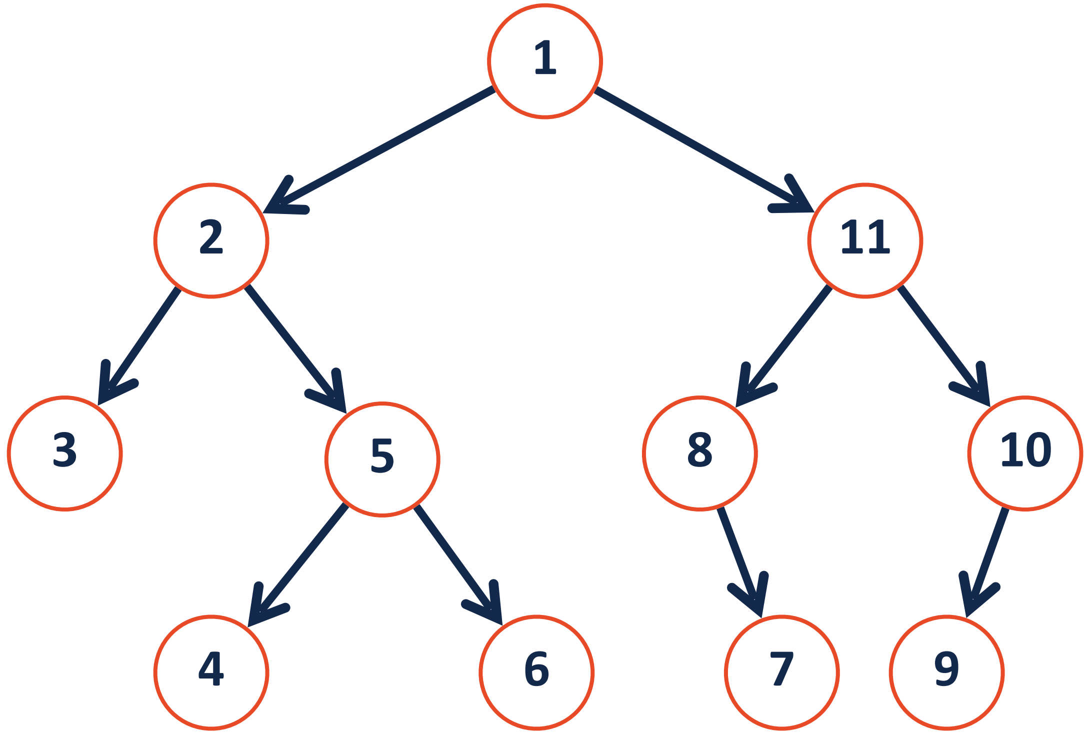
Stack:

Print:

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)
  Print tmp
  Push tmp->right to stack
  Push tmp->left to stack
```



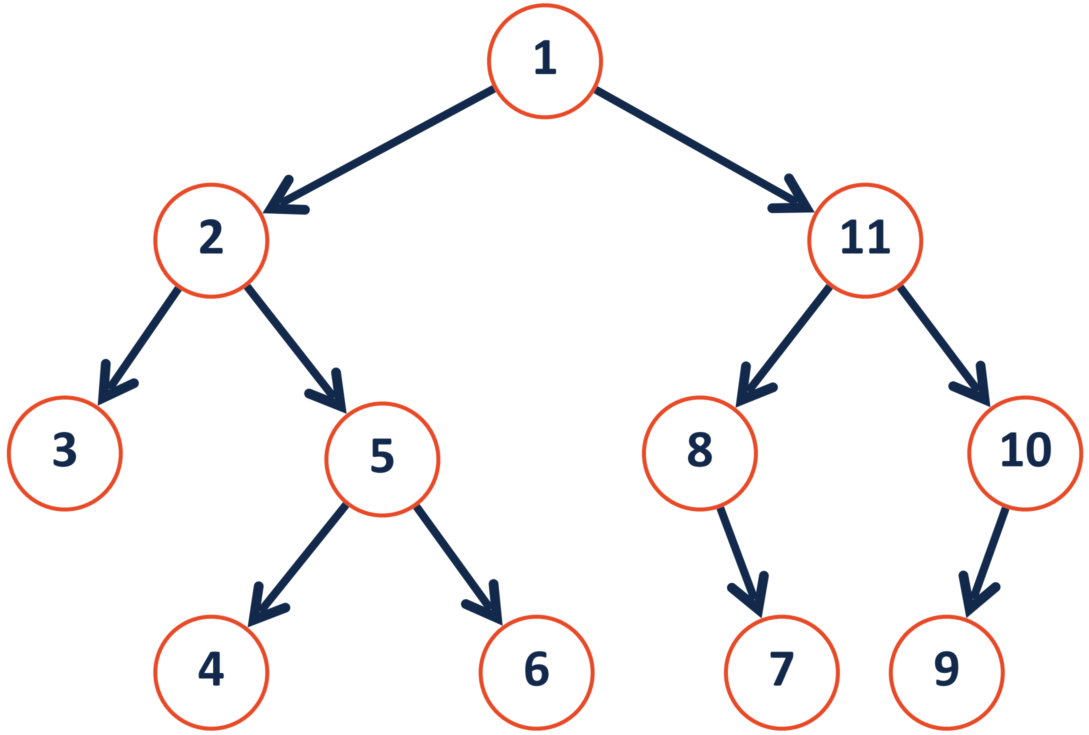
Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

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

Breadth First Search

Fully explore depth i before exploring depth i+1

```
Make a queue initialized with root
While queue isn't empty:
  Dequeue front element (as tmp)
  Print tmp
  Enqueue tmp->left
  Enqueue tmp->right
```



Queue:

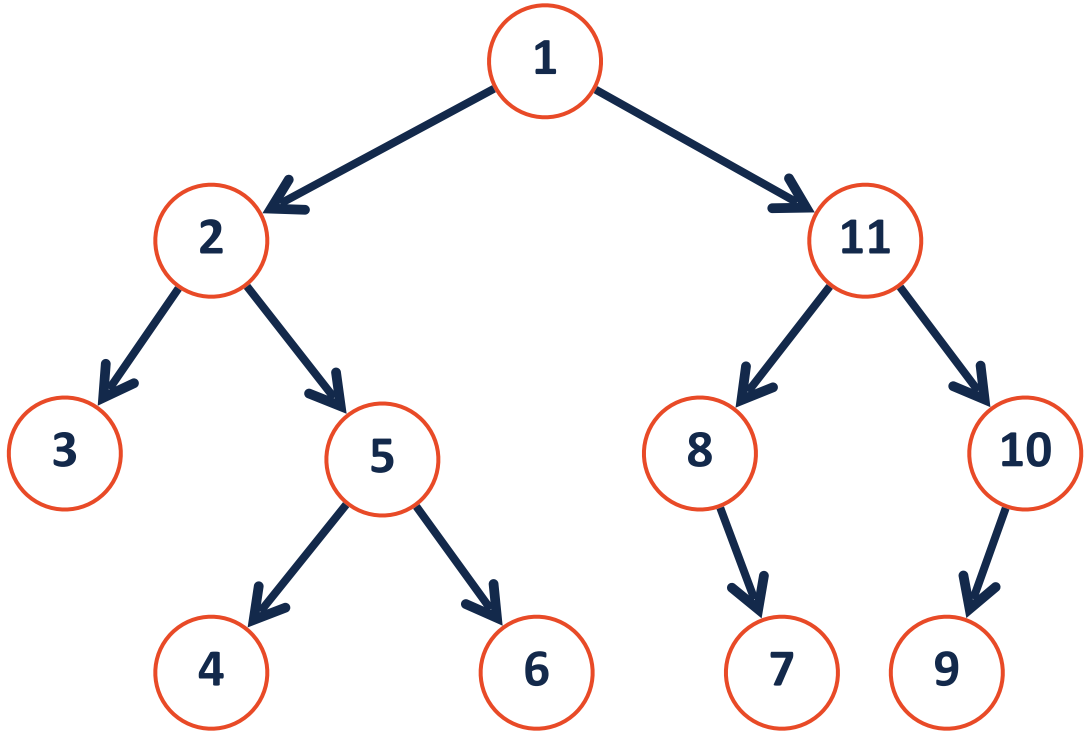
Print:

Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue initialized with root

```
While queue isn't empty:
  Dequeue front element (as tmp)
  Print tmp
  Enqueue tmp->left
  Enqueue tmp->right
```

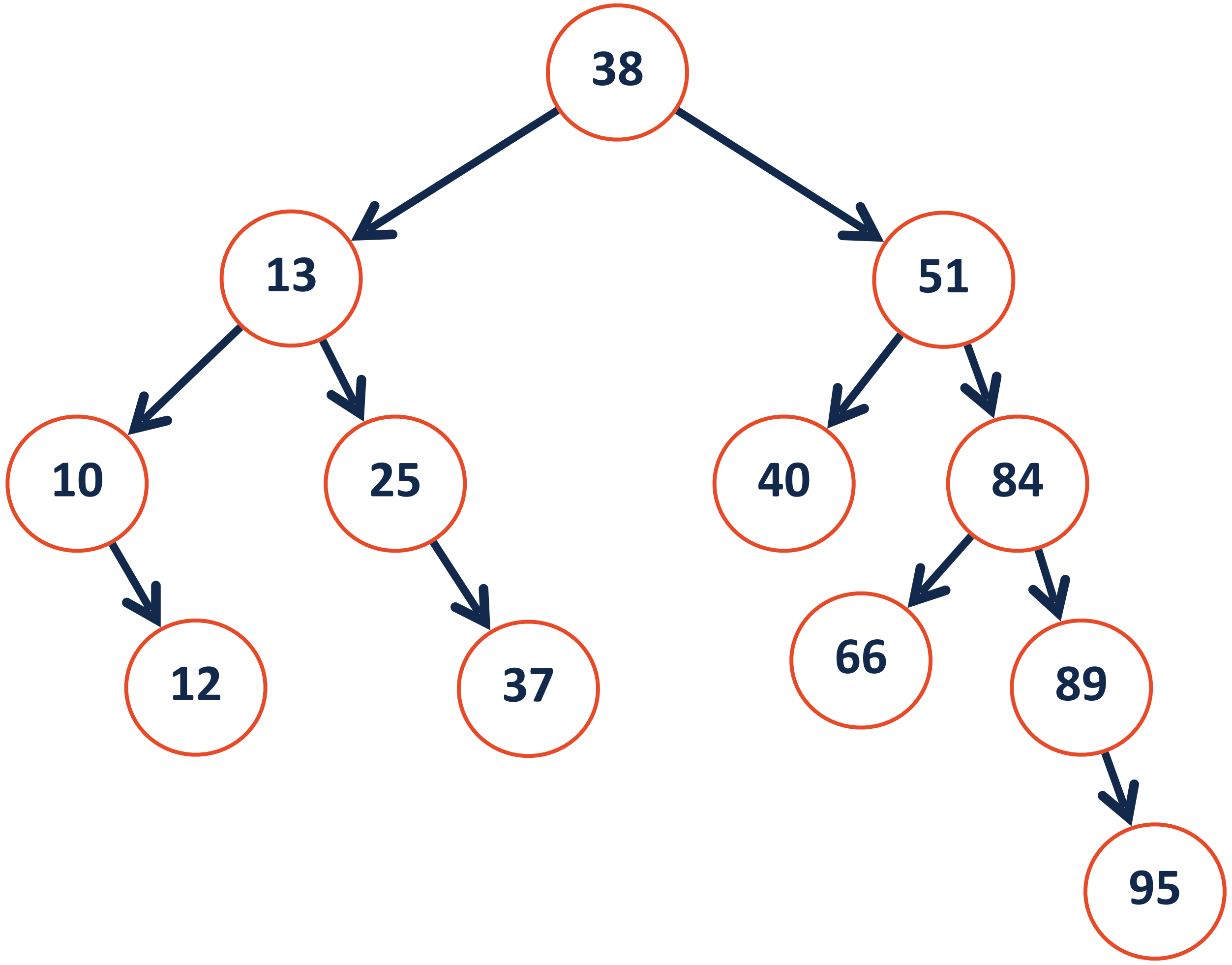


Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

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

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

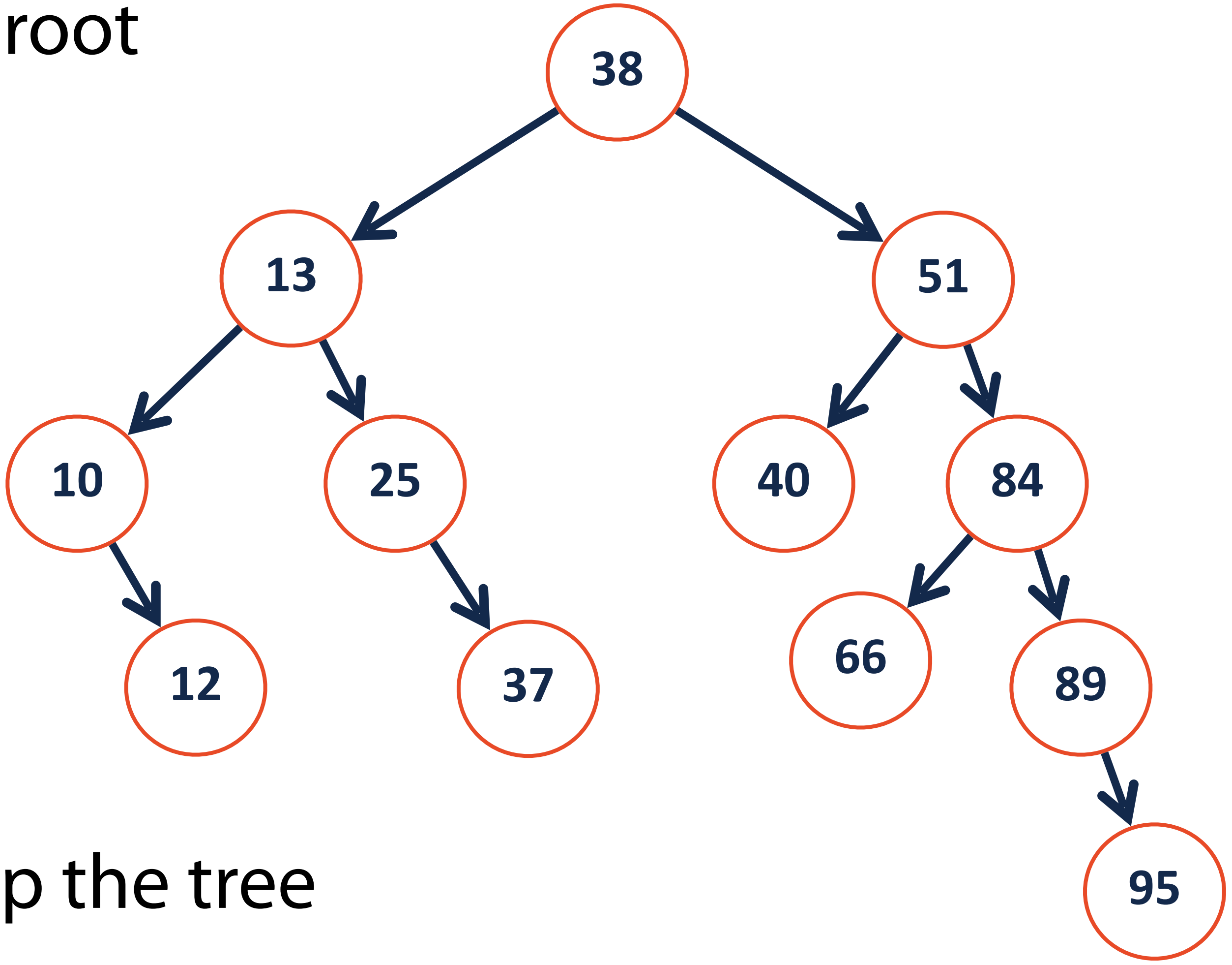
Recursion:

`tmp <  query`, recurse right

`tmp >  query`, recurse left

Combining:

Return the recursive value back up the tree

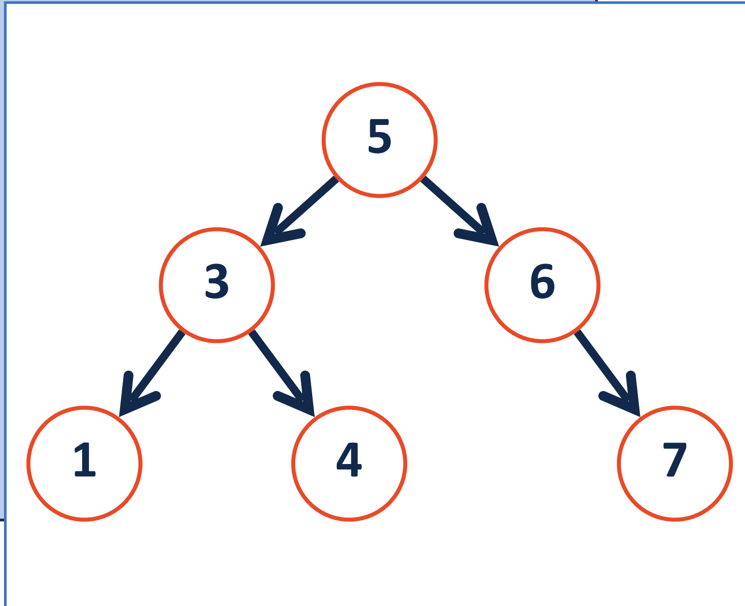


```
template<typename K, typename V>

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


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

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

return _find(root->right, key);


}
```

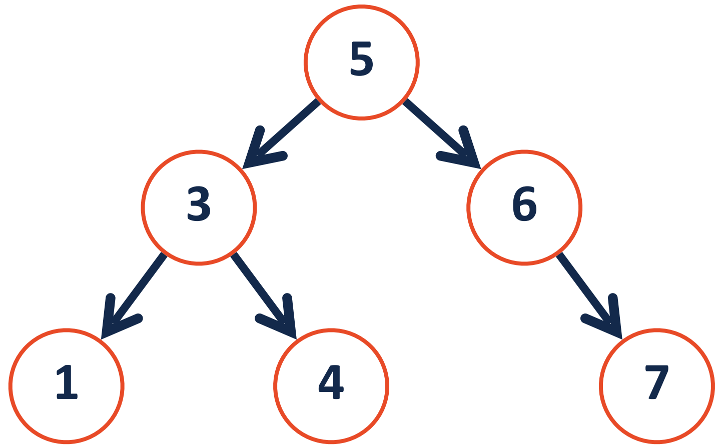
$$\begin{tikzcd}
 & & 5 \arrow[dl] \arrow[dr] & \\
 & 3 \arrow[dl] \arrow[d] & & 6 \arrow[d] \\
1 & 4 & & 7
\end{tikzcd}$$

```
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);




}



```



```
template<typename K, typename V>

void _remove(TreeNode *& root, const K & key) {




















}
```



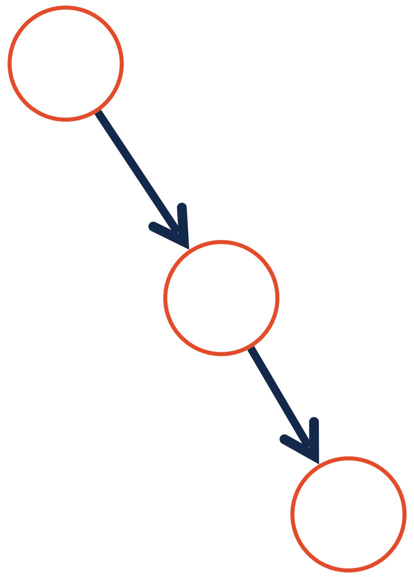
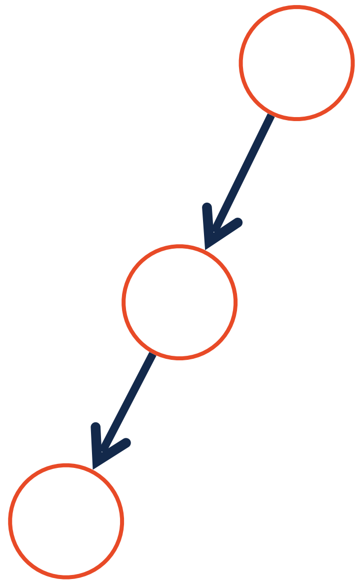
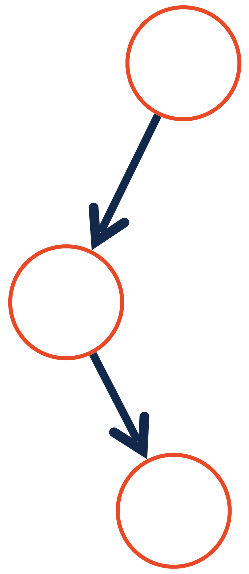
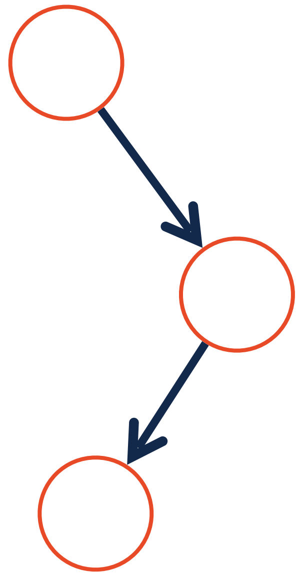
BST Analysis – Running Time

Operation	BST Worst Case
find	
insert	
remove	
traverse	

BST Analysis – Running Time

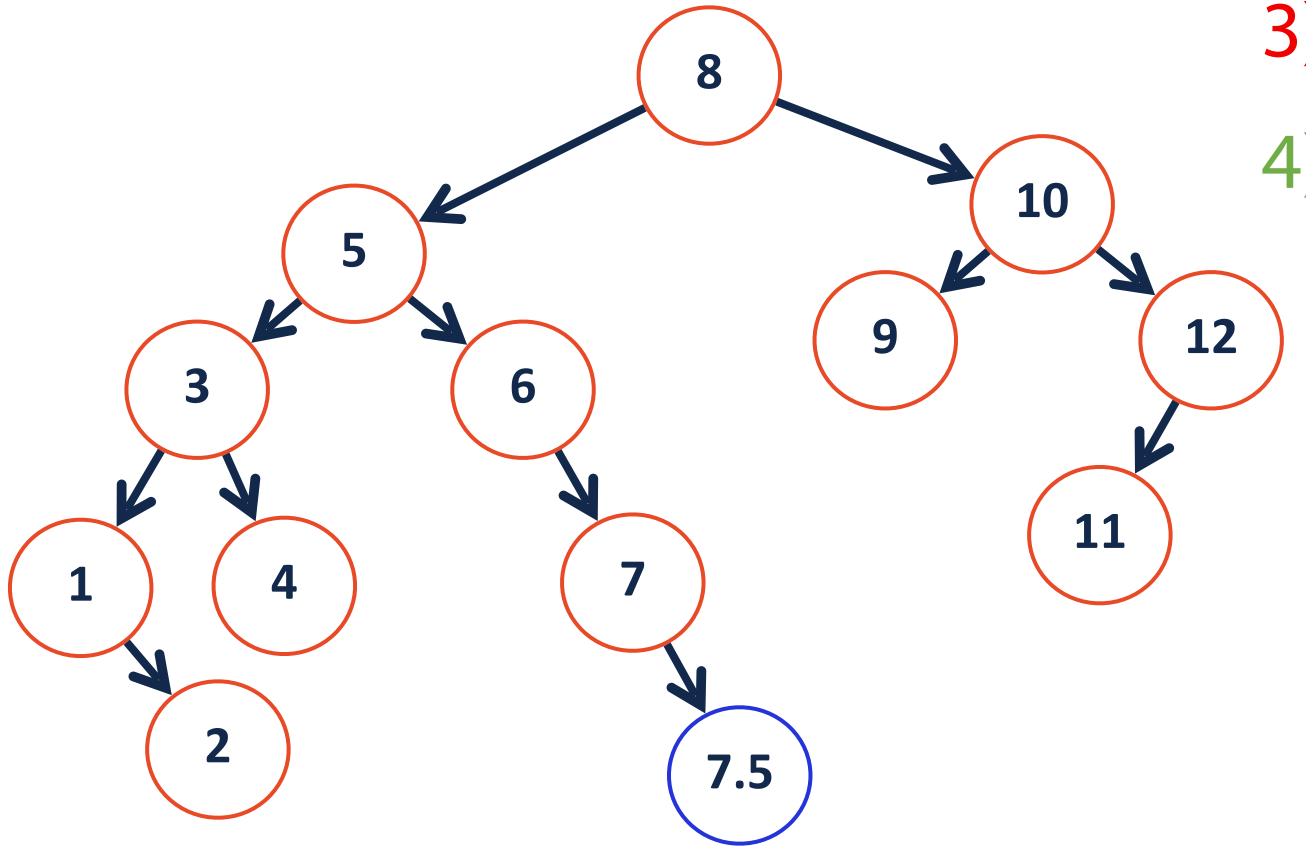
Operation	BST Worst Case
find	O(h) = O(n)
insert	O(h) = O(n)
remove	O(h) = O(n)
traverse	O(n)

AVL Rotations

	Left	**Right**	**LeftRight**	**RightLeft**
				
Root Balance:	2	-2	-2	2
Child Balance:	1	-1	1	-1

Left Rotation

1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->right = root->left
4) root->left = tmp



AVL Rotations

Four kinds of rotations: (L, R, LR, RL)

1. All rotations are local (subtrees are not impacted)

2. The running time of rotations are constant

3. The rotations maintain BST property

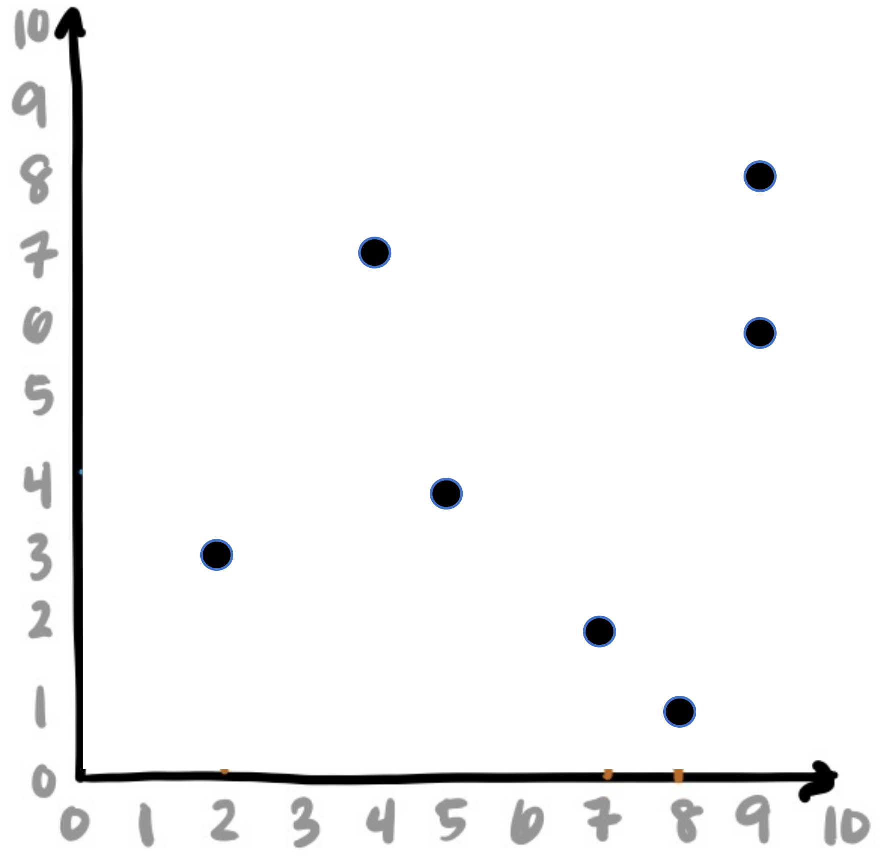
Goal: Maintain AVL Tree balance

Implied (now Proved): If the tree is balanced, height is bounded by O(log n)

Nearest Neighbor: k-d tree

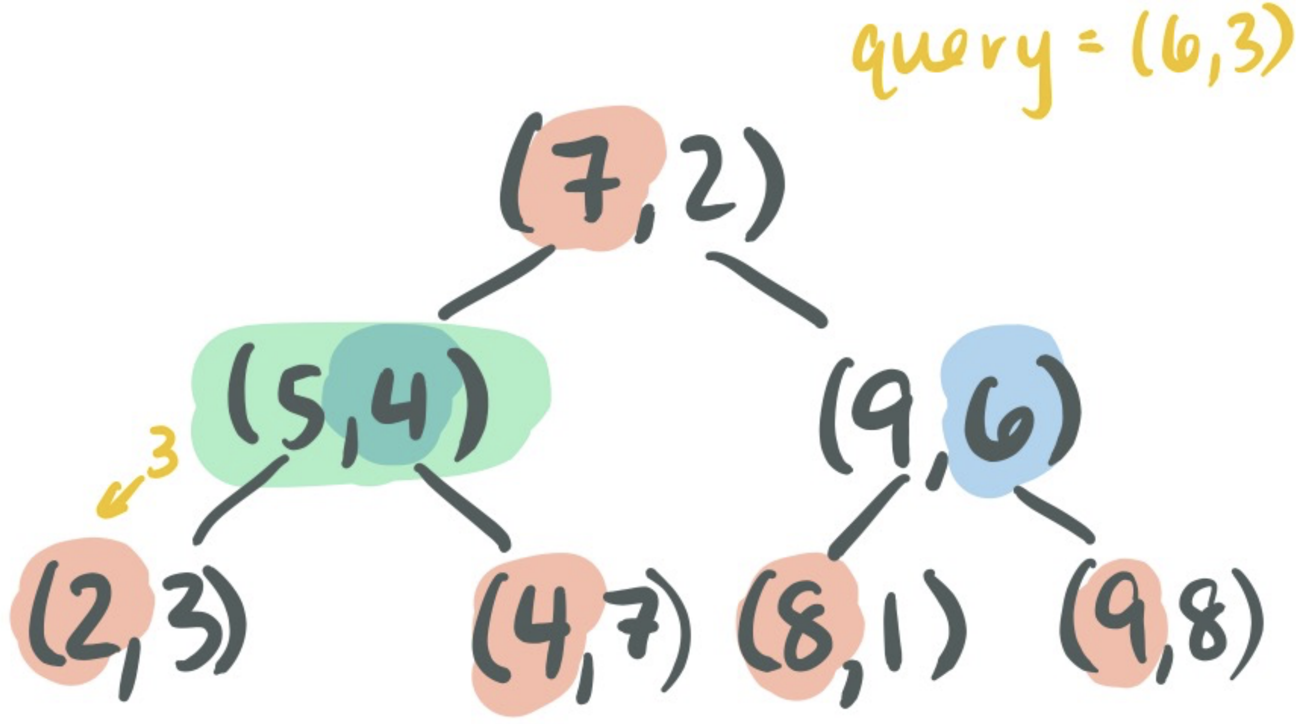
A **k-d tree** is similar but splits on points:

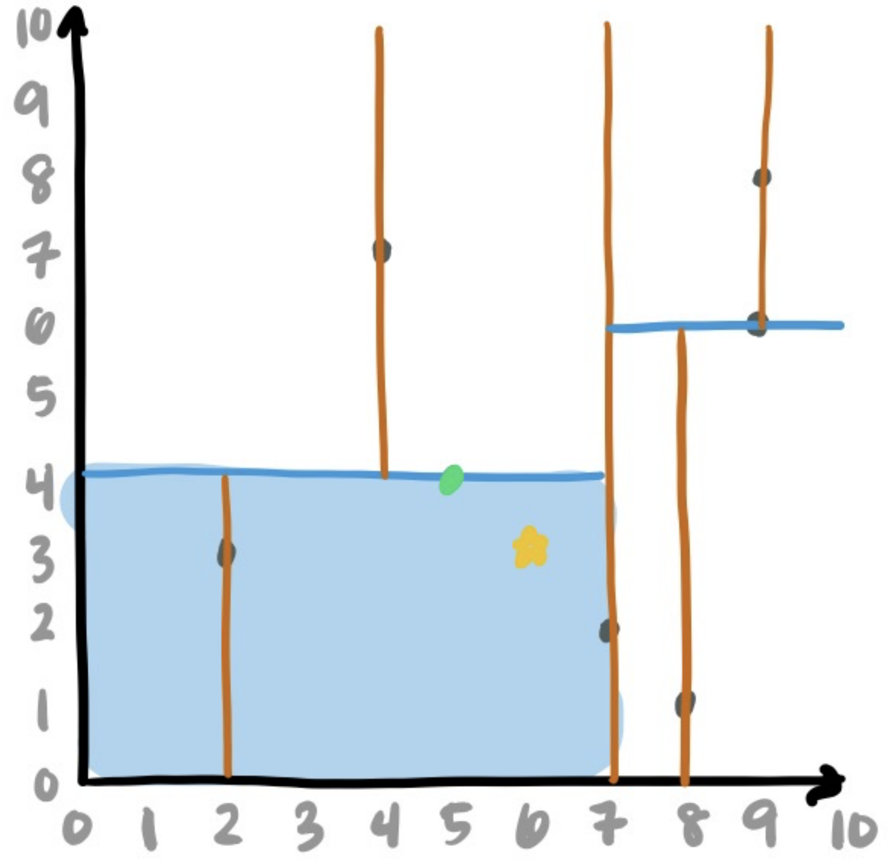
```
(7,2), (5,4), (9,6), (4,7), (2,3), (8,1), (9,8)
```



Nearest Neighbor: k-d tree

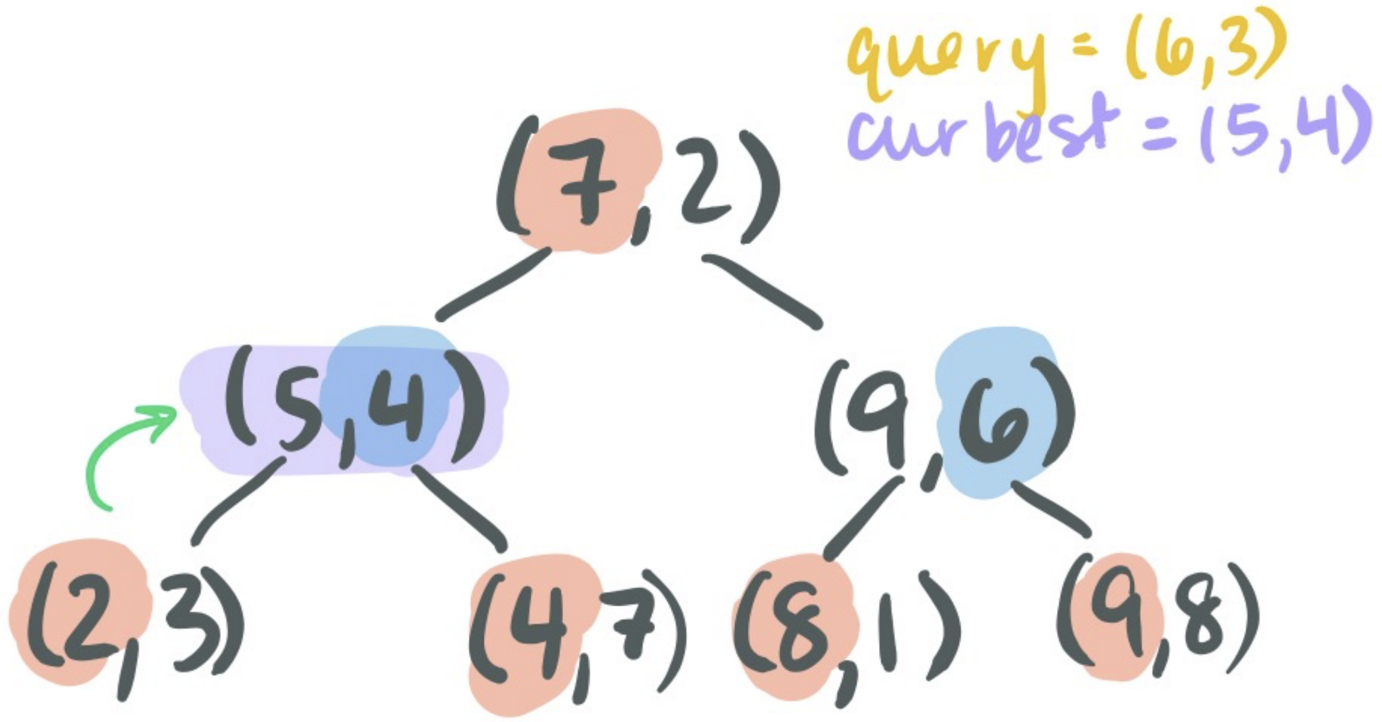
Search by comparing query and node in single **alternating** dimension

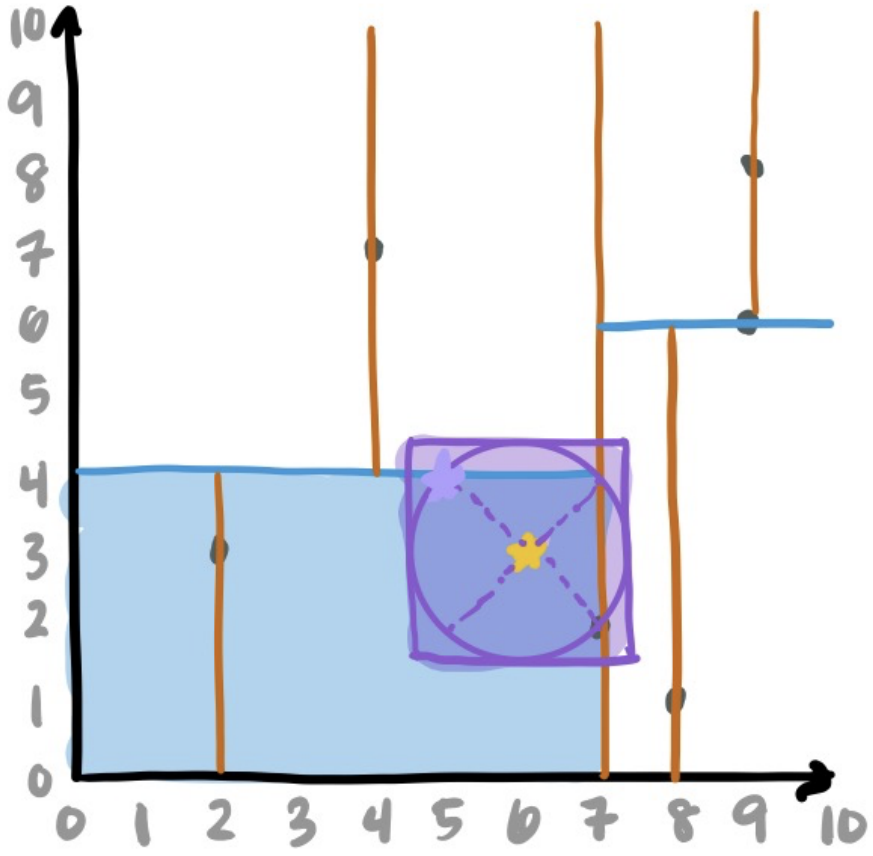




Nearest Neighbor: k-d tree

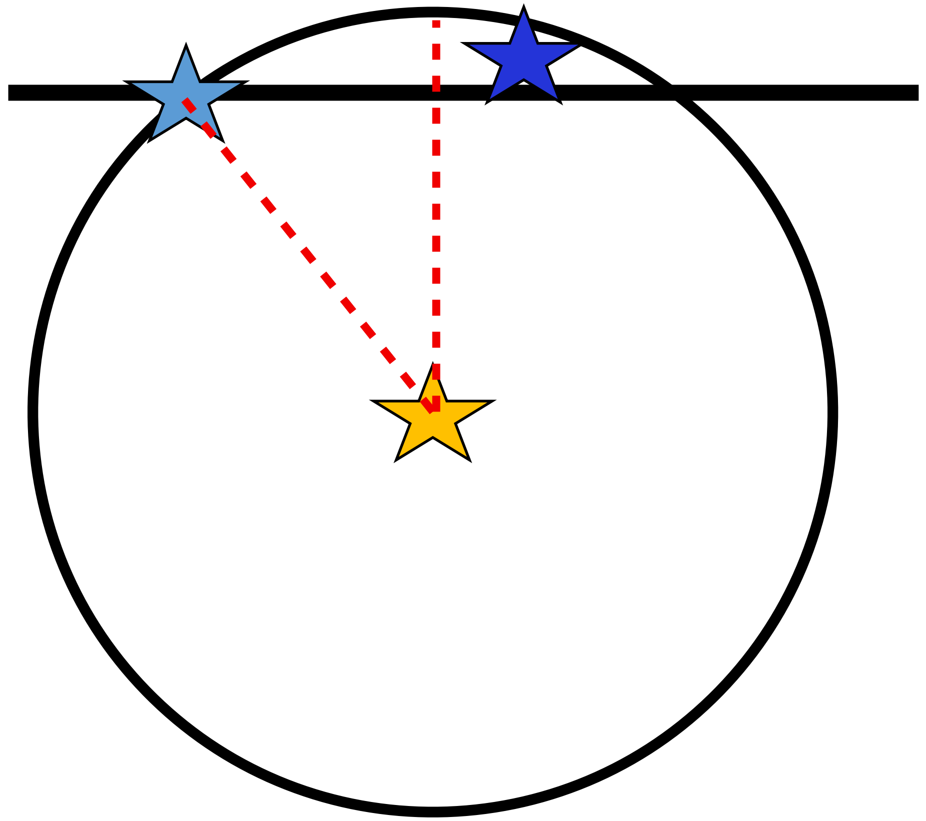
Backtracking: start recursing backwards -- store "best" possibility as you trace back

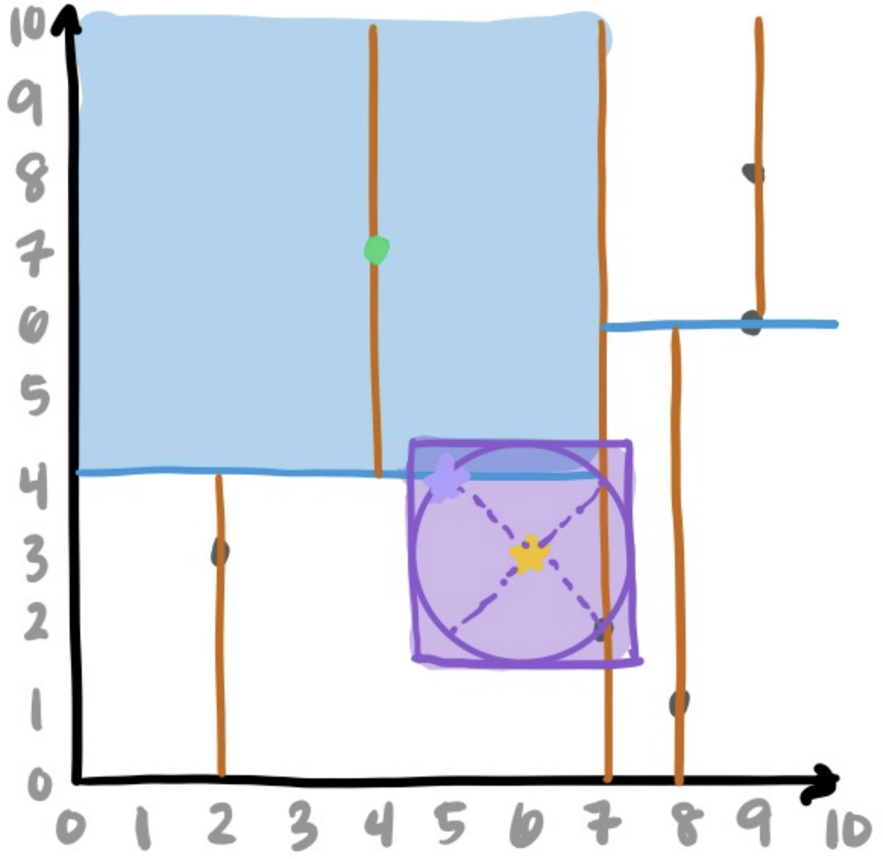




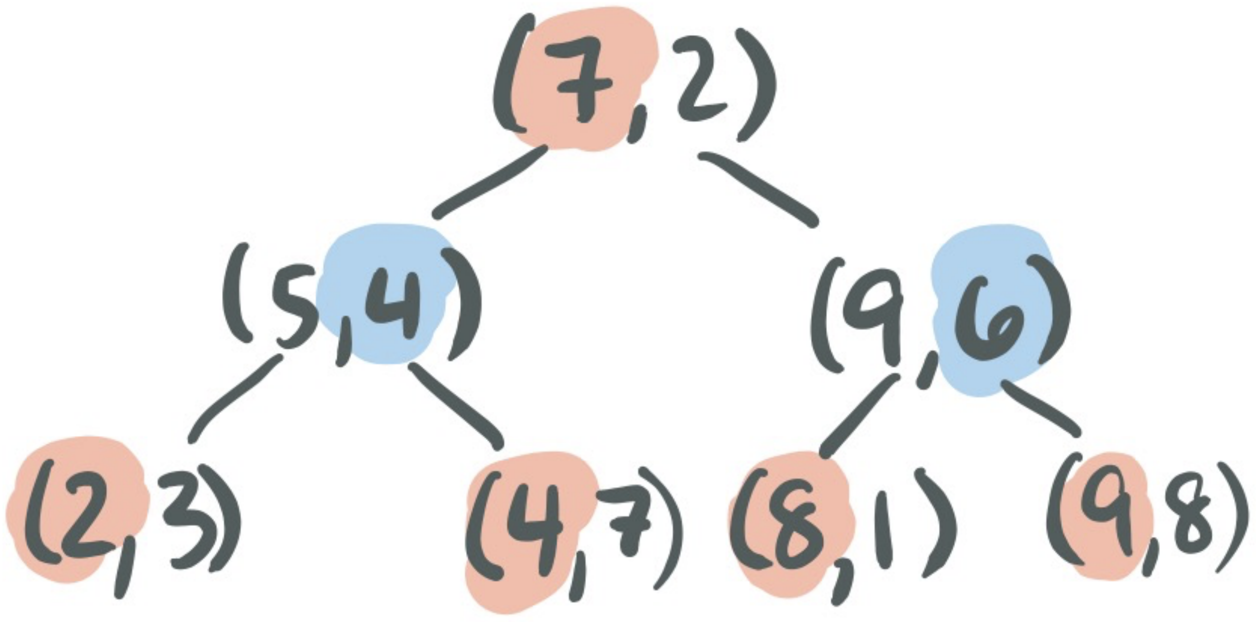
Nearest Neighbor: k-d tree

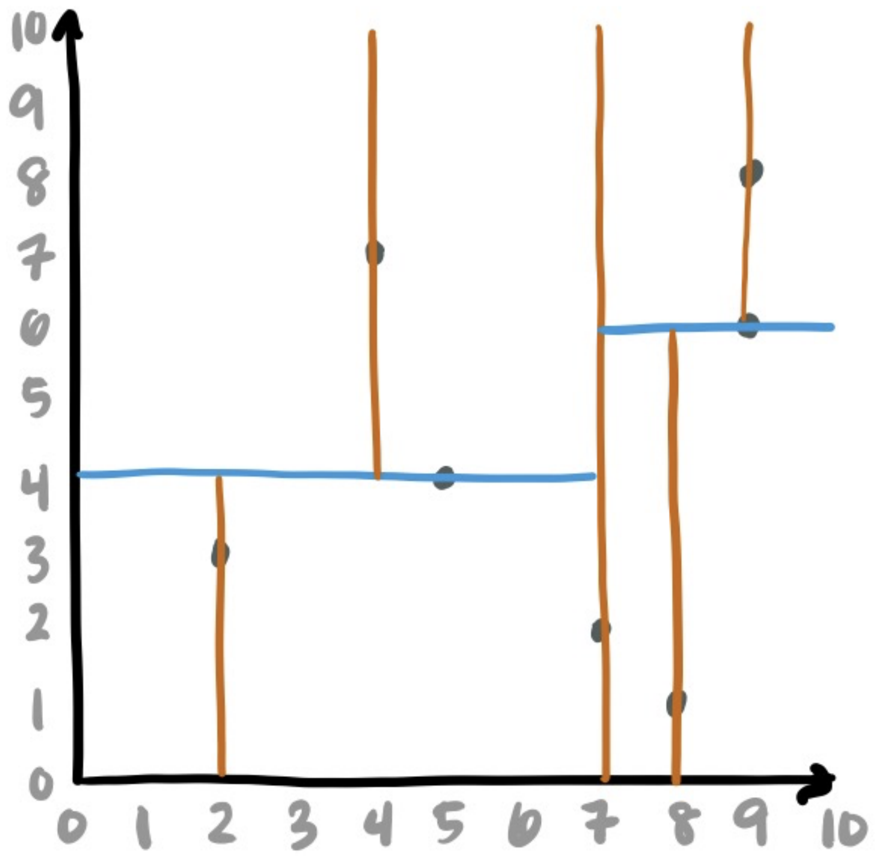
May have to recursively check other branches of tree — **why?**





Nearest Neighbor: k-d tree





BTree Properties

A **BTrees** of order **m** is an m-ary tree and by definition:

- All keys within a node are ordered
- All nodes contain no more than **m-1** keys.
- All internal nodes have exactly **one more child than keys**

Root nodes can be a leaf or have _____________ children.

All non-root, internal nodes have ___________________ children.

All leaves in the tree are at the same level.

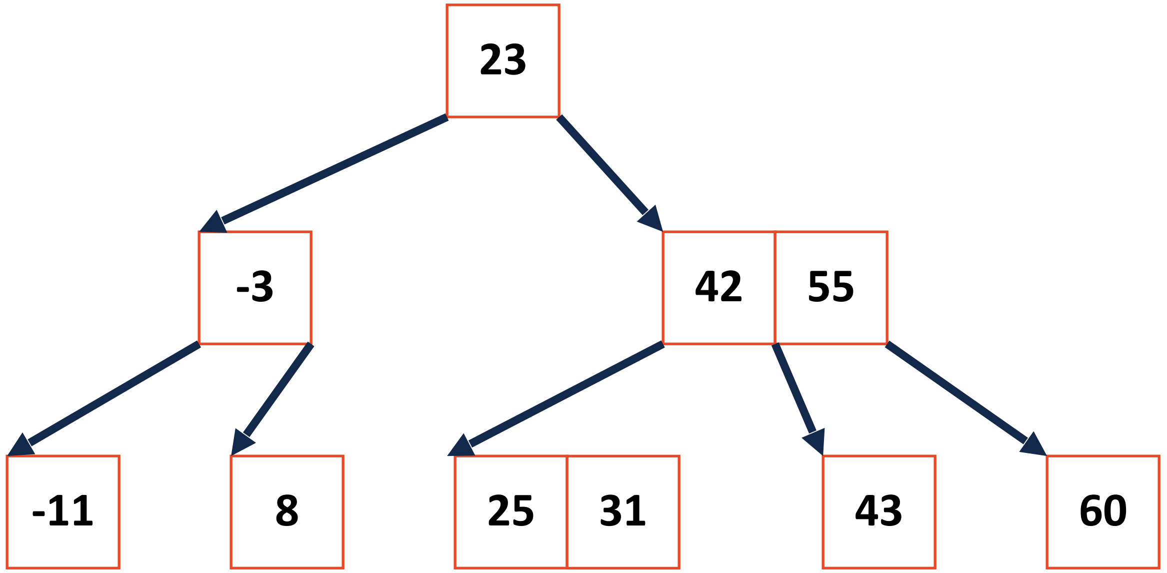
BTree Find

`Find(7)`

Base Case:

If root is empty, return

If leaf, do array find() and return



Recursive Step:

Array find() for match or first greater value

Recurse on appropriate child

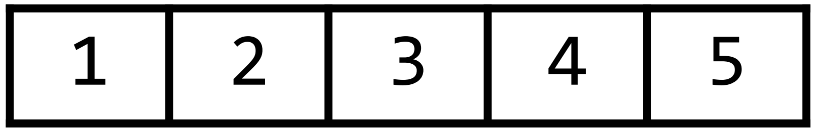
Tip: Index of first greater value is index of child we want to visit!

BTree Insertion

M = 5

When we hit **M** items, split into three nodes!

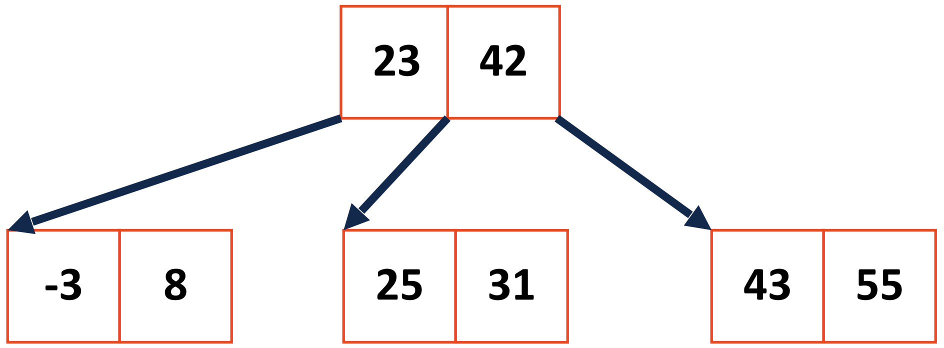
```
Insert(1)
Insert(2)
Insert(3)
Insert(4)
Insert(5)
Insert(6)
Insert(7)
Insert(8)
```



BTree Recursive Insert

Insert(56), M = 3

Insert always starts at a leaf but can propagate up repeatedly.



Final thoughts on Trees

Trees have a large space of **possible coding questions**

The practice exam question was build mirror tree

What concepts was this question reinforcing?

Final thoughts on Trees

Trees have a large space of **possible coding questions**

The practice exam question was build mirror tree

What similar problems with a twist might you want to prep for?