

Data Structures

Exam 3 Review

CS 225
Brad Solomon

October 16, 2026



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Be sure to
do practice exam
& extra FRQ practice!



Learning Objectives

Any overflow information about heaps

Review concepts on exam 3



Exam 3 Concepts

Trees!

Binary Trees

Binary Search Trees

Balanced Binary Binary Trees (AVL)

B Trees

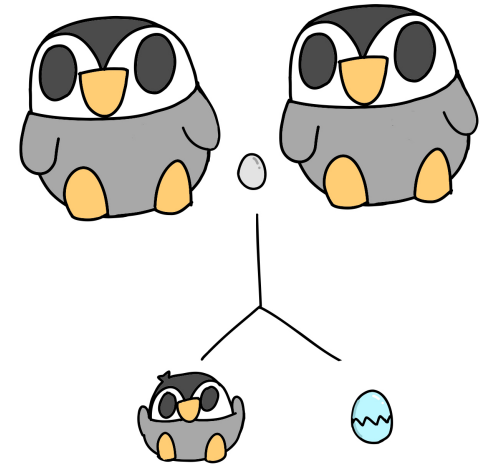
FRQs will also require previous data structure knowledge

Heaps is not on the exam

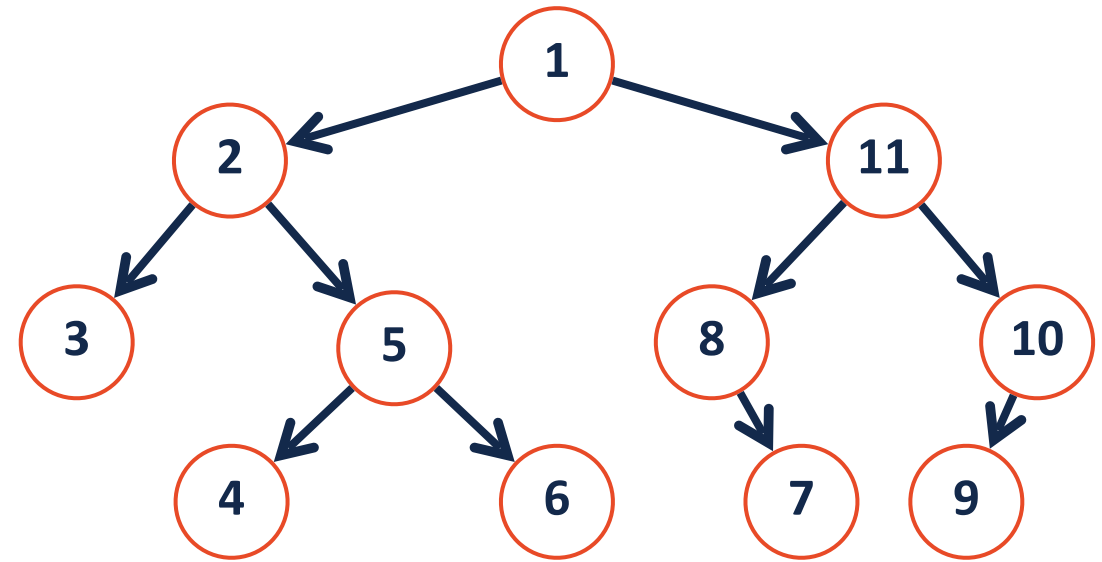
Exam 2

* Big picture topics

Trees



Tree Traversals

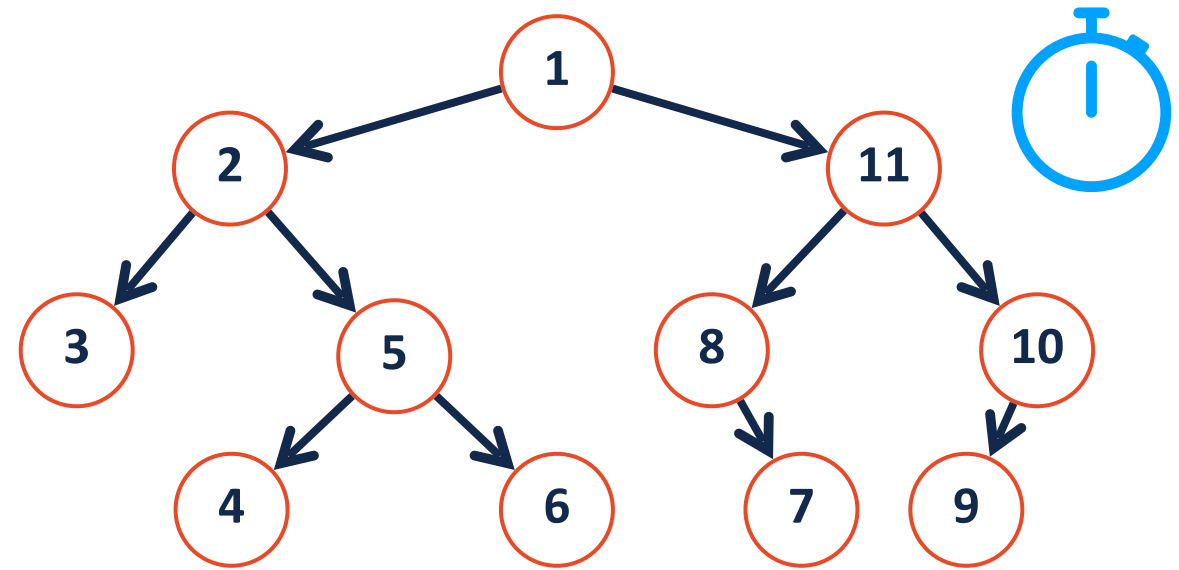


Pre-order:

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

→ Pie
in
post

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)

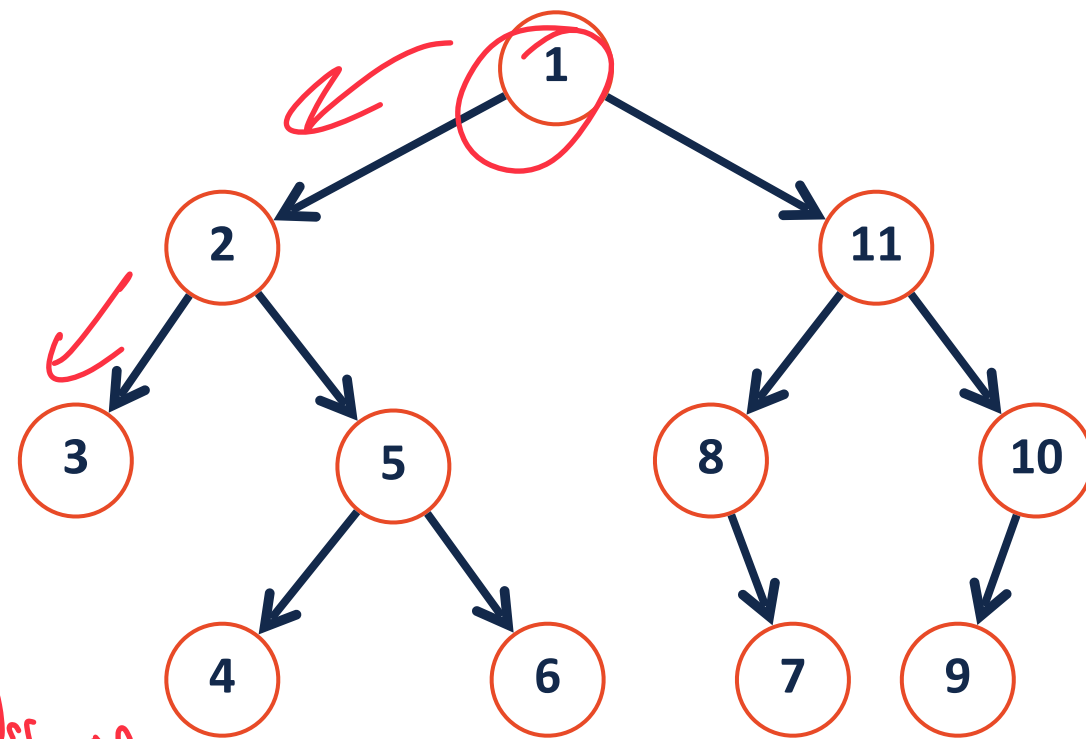
Check if tmp is query (visual: 'print')

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



order of
these three lines

Breadth First Search → Level order

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

Make a queue initialized with root

While queue isn't empty:

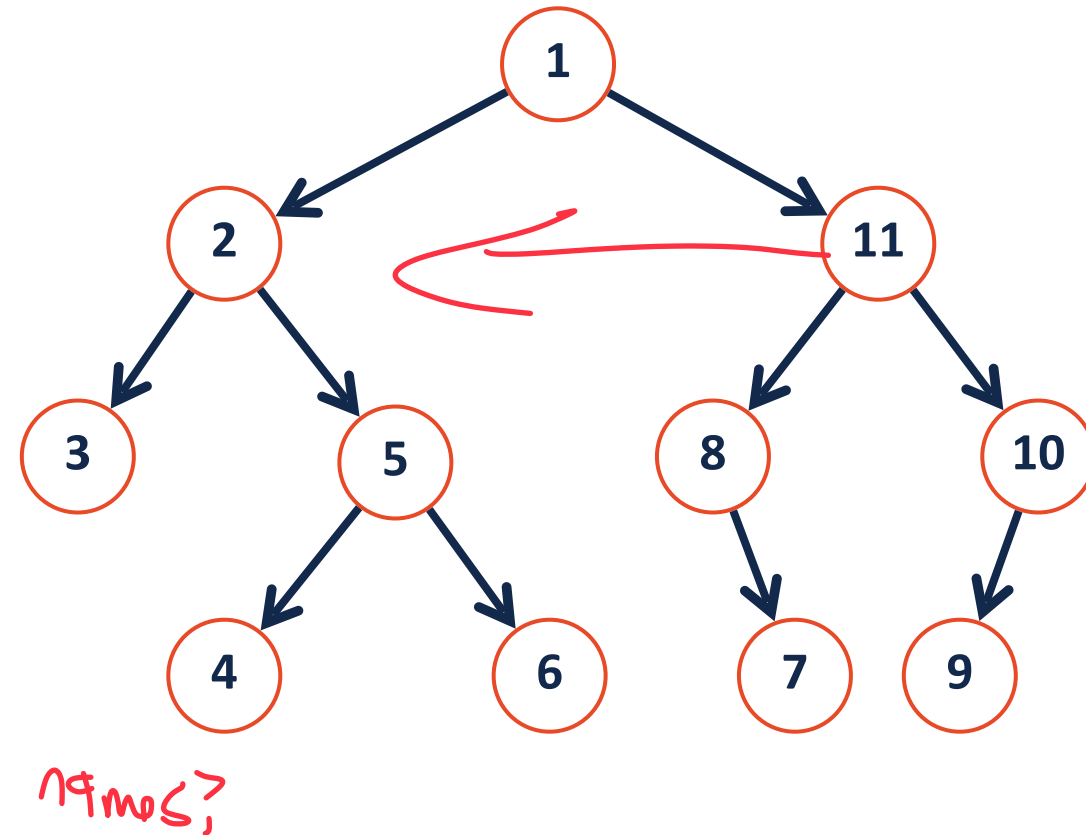
Dequeue front element (as tmp)

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

Enqueue tmp->left

Enqueue tmp->right

Different



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

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

BST Find

↳ Binary tree w/ sorted order!

find(66)

One path

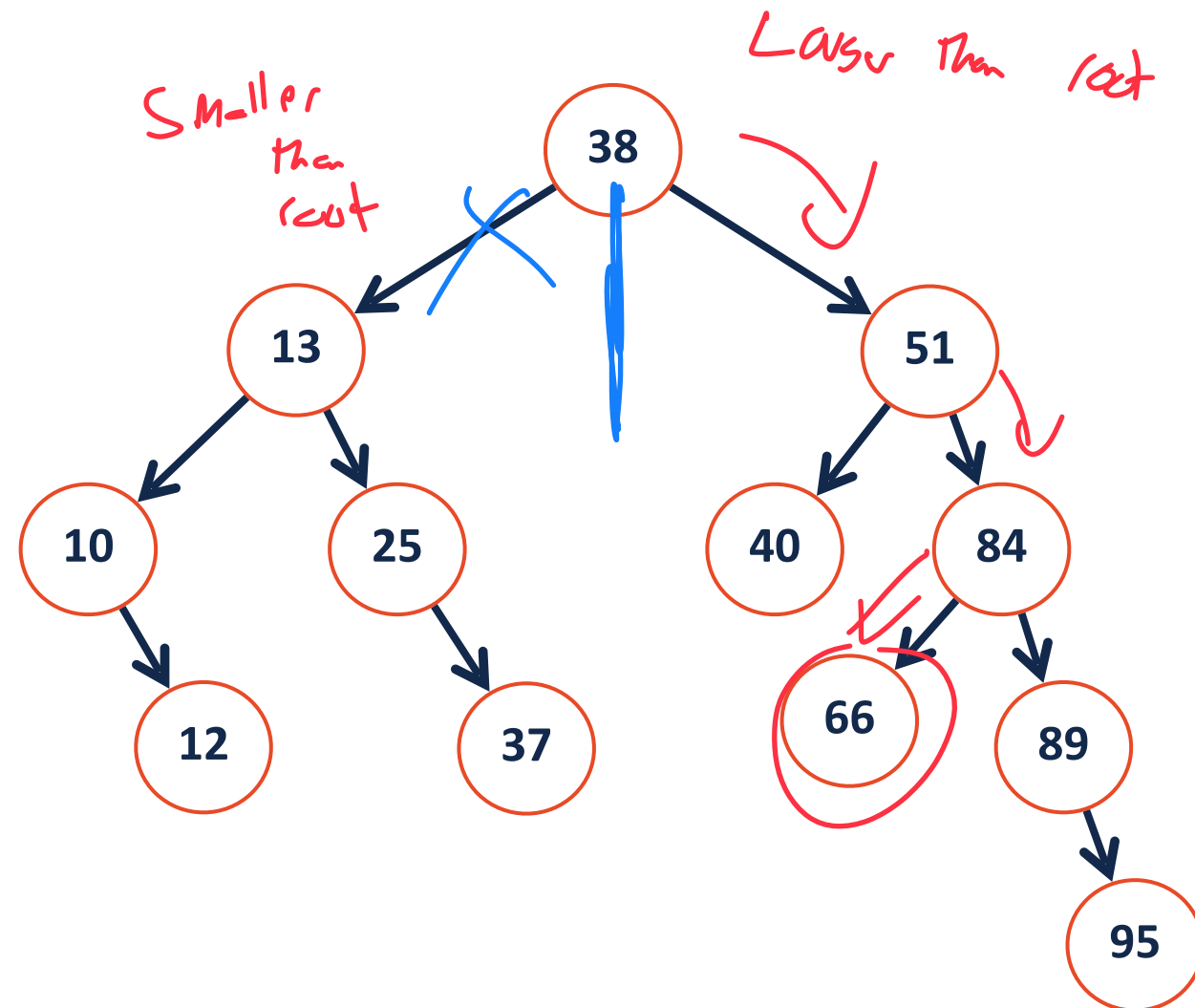
$O(h)$

is at most $O(n)$

vs

All paths

$O(n)$



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

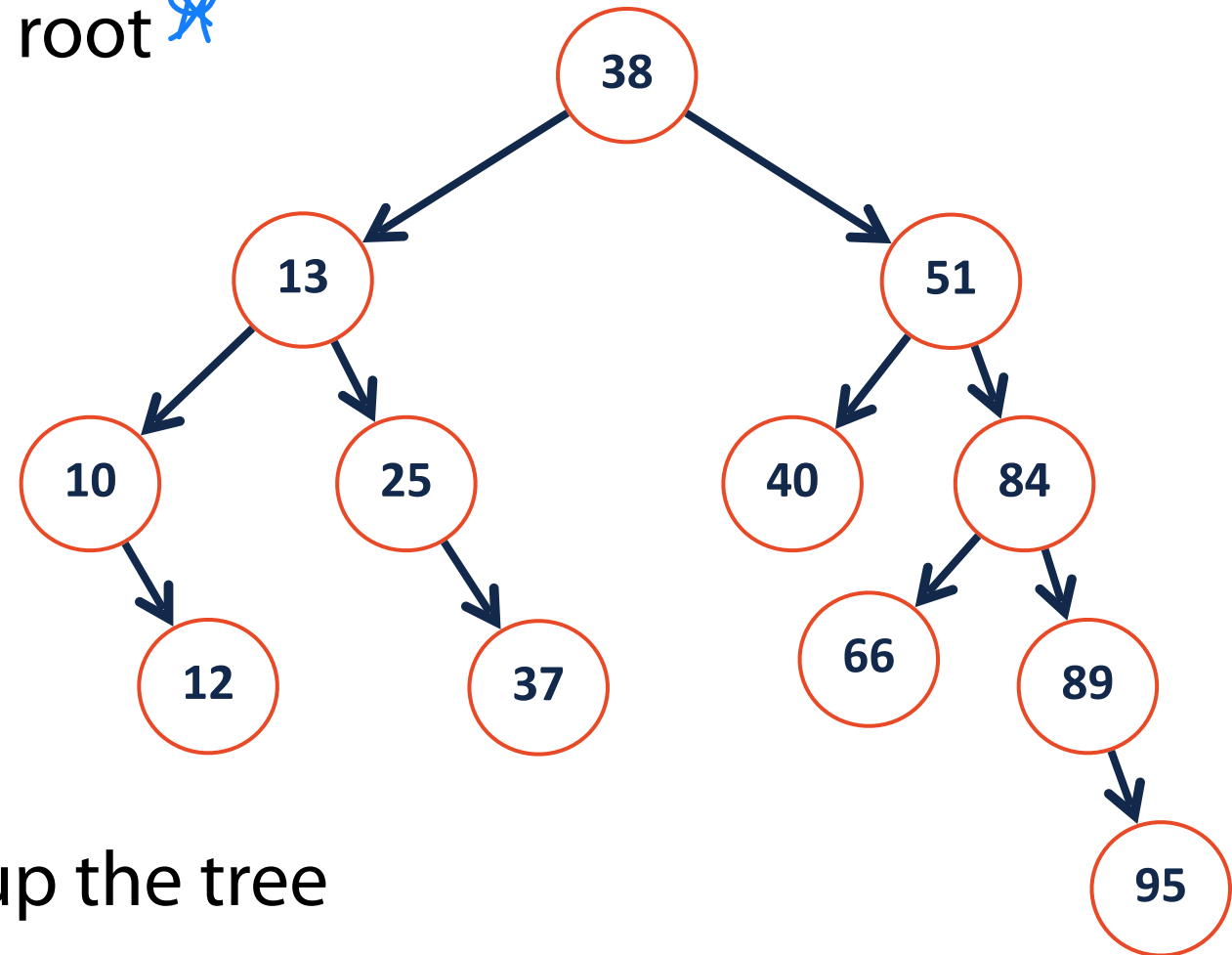
Recursion: ✂

tmp < query, recurse right

tmp > query, recurse left

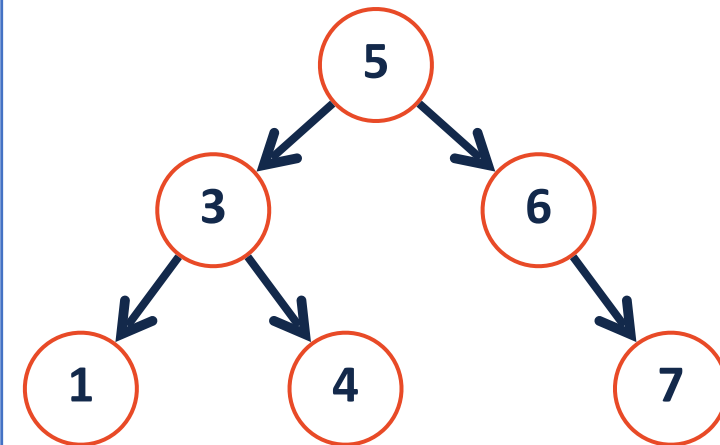
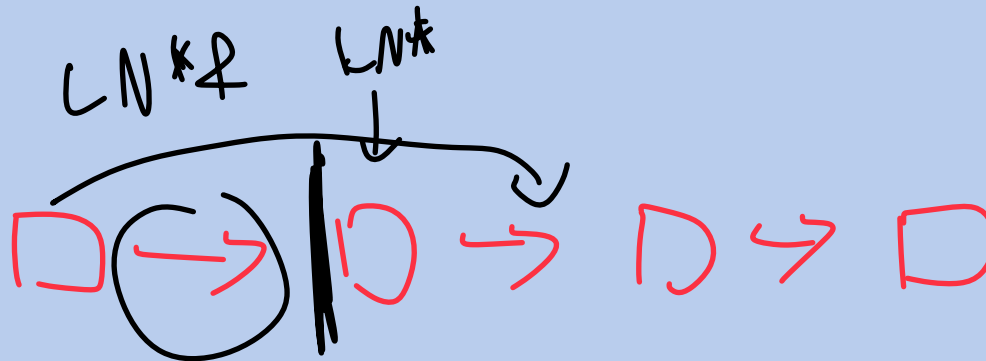
Combining: ✂

Return the recursive value back up the tree





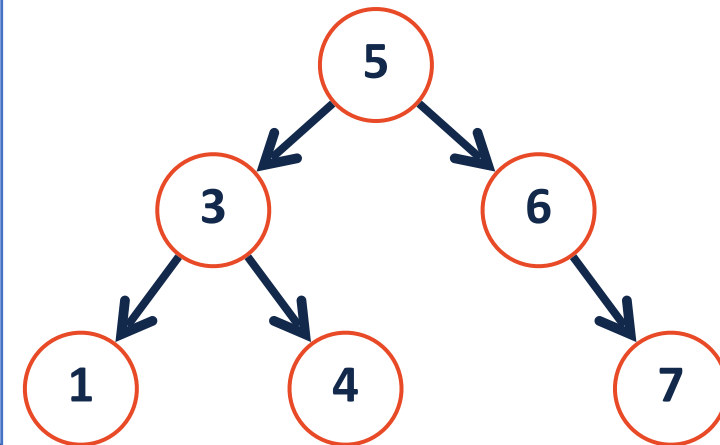
```
1 template<typename K, typename V>
2
3 TreeNode *& __find(TreeNode *& root, const K & key) {
4
5
6 // Base Case
7 if(root == nullptr || root->key == key){
8     return root;
9 }
10
11 // Recursive Step ("Combining step" is 'return')
12 if (root->key > key){
13     return __find(root->left, key);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



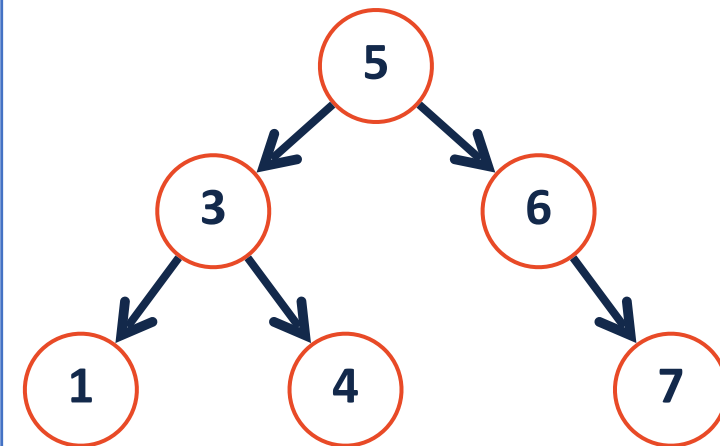


```
1 template<typename K, typename V>
2
3 void _insert(const K & key, const V & val) {
4
5     return _insert(root, key, val);
6 }
7
```

```
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     tmp = new treeNode(key, val);
8
9
10
11
12
13 }
14
15
16
```



```
1 template<typename K, typename V>
2
3 void _remove(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 }
```



BST Analysis – Running Time

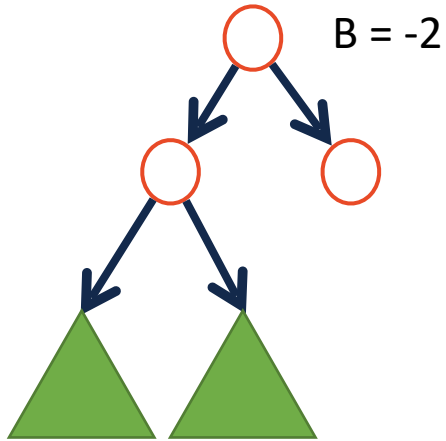


Operation	BST Worst Case
find	$O(h)$
insert	$O(h) \sim O(n)$
remove	$O(h) \sim$
traverse	$O(n)$

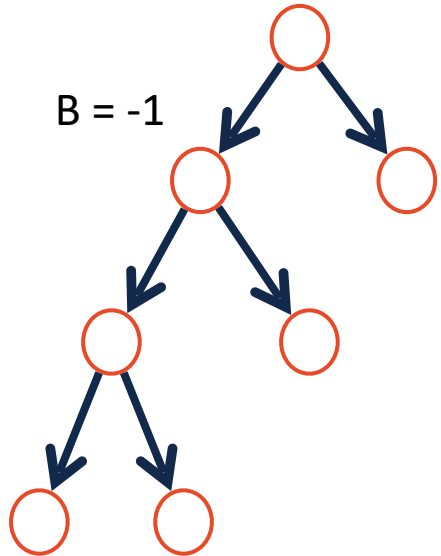
BST Analysis – Running Time

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

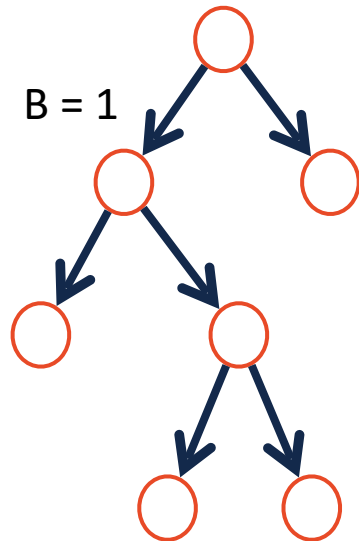
Left Imbalance



Check left subtree:

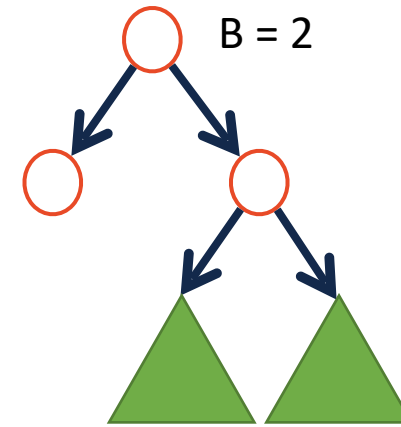


Right

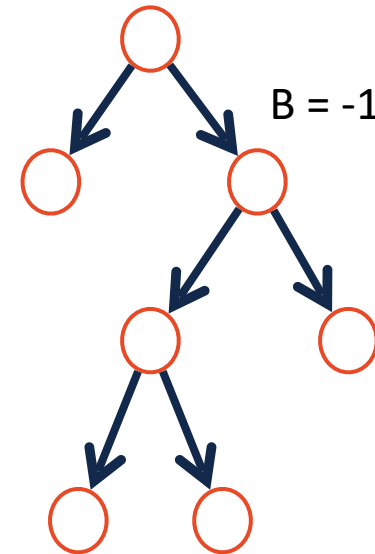


LeftRight

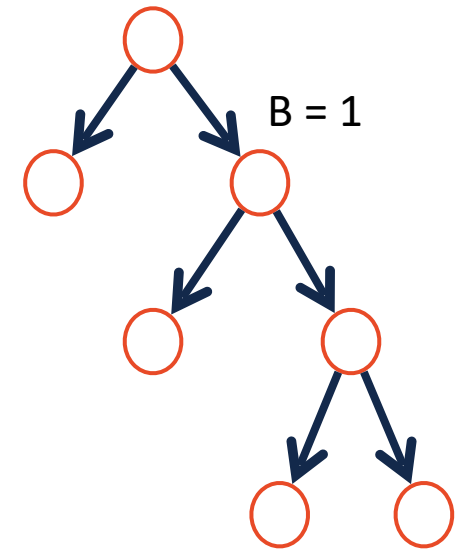
Right Imbalance



Check right subtree:



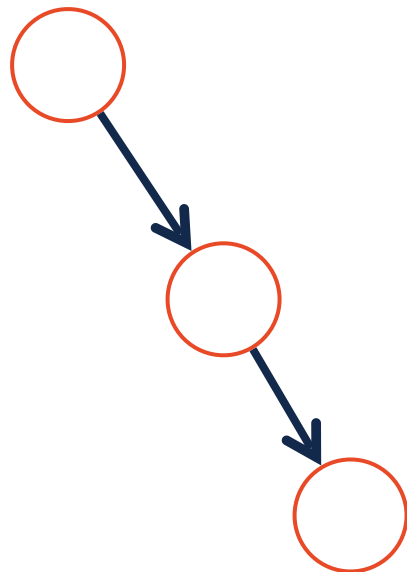
RightLeft



Left

AVL Rotations

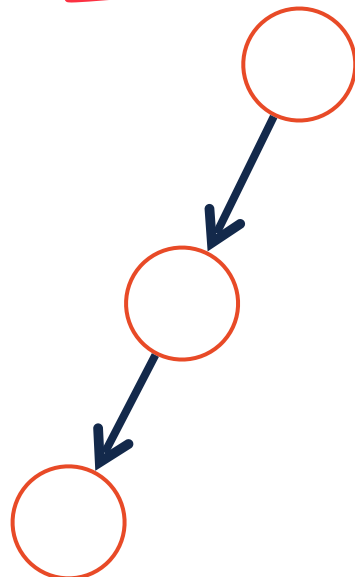
Left



Root Balance: 2

Child Balance: 1

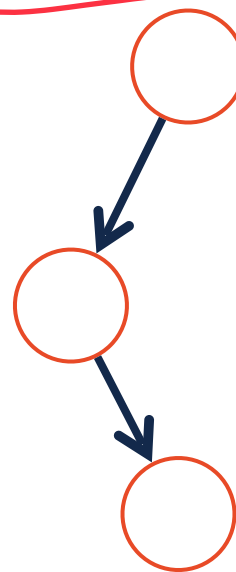
Right



-2

-1

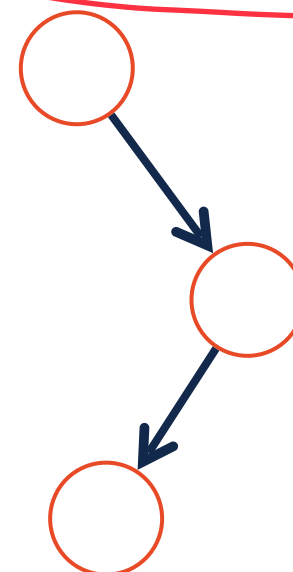
LeftRight



-2

1

RightLeft



2

-1



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

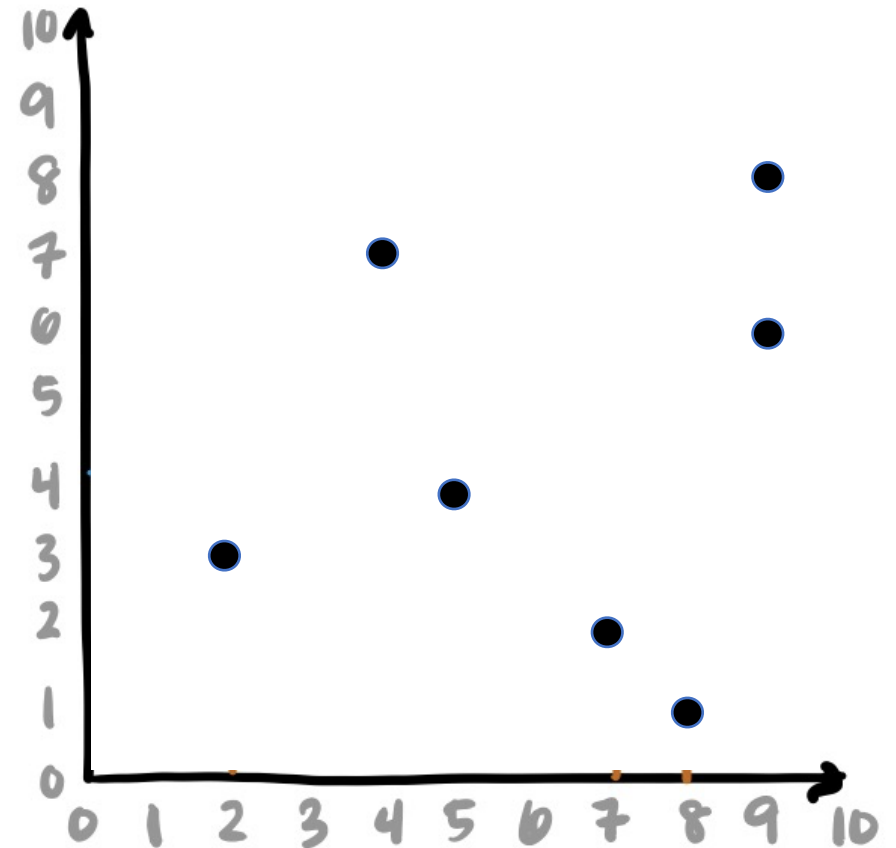
Goal: Maintain AVL Tree balance

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

Nearest Neighbor: k-d tree

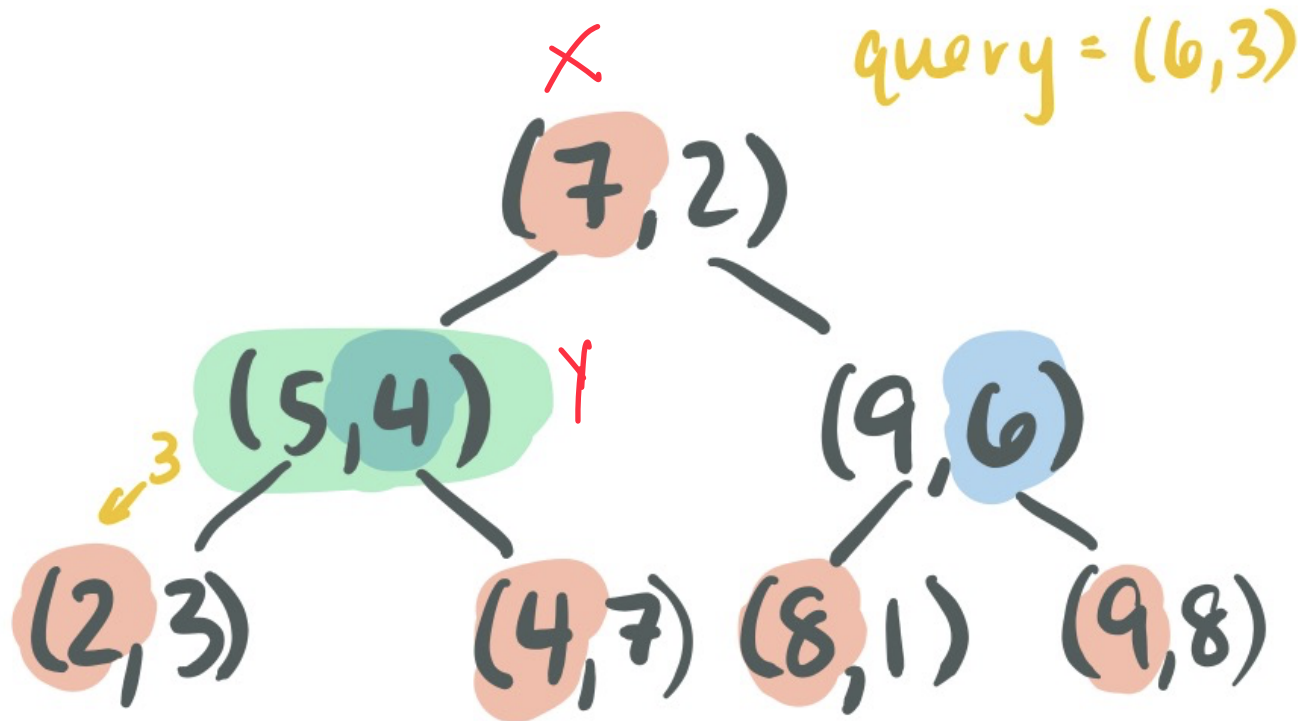
A **k-d tree** splits a multi-dimensional dataset on points:

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

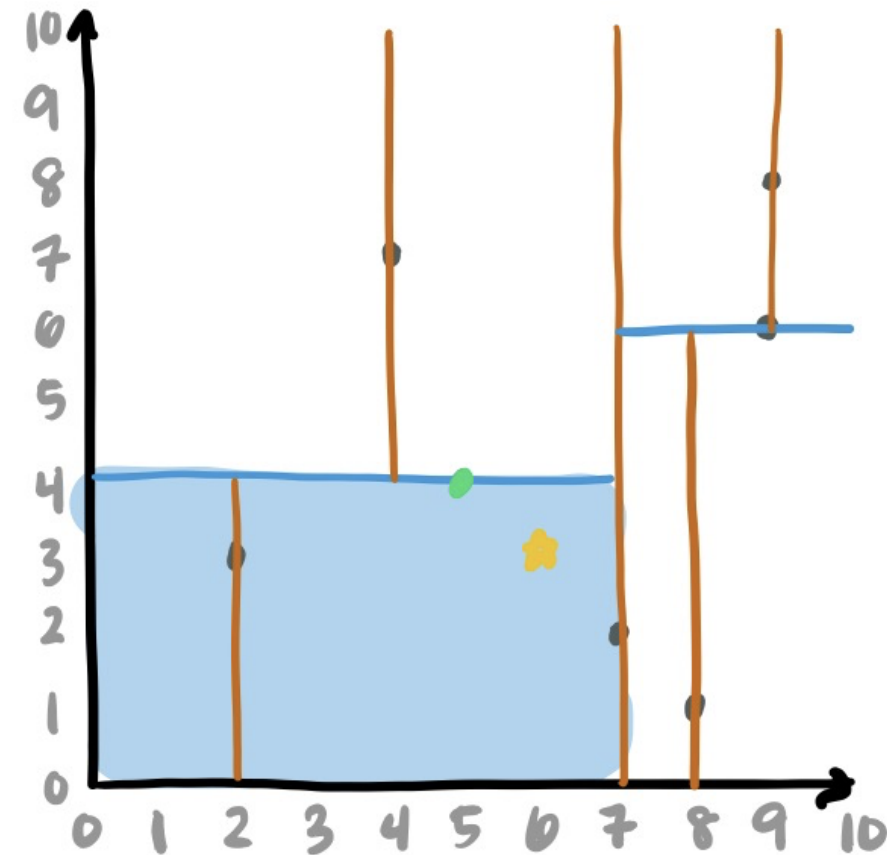


Nearest Neighbor: k-d tree

Search by comparing query and node in single alternating dimension

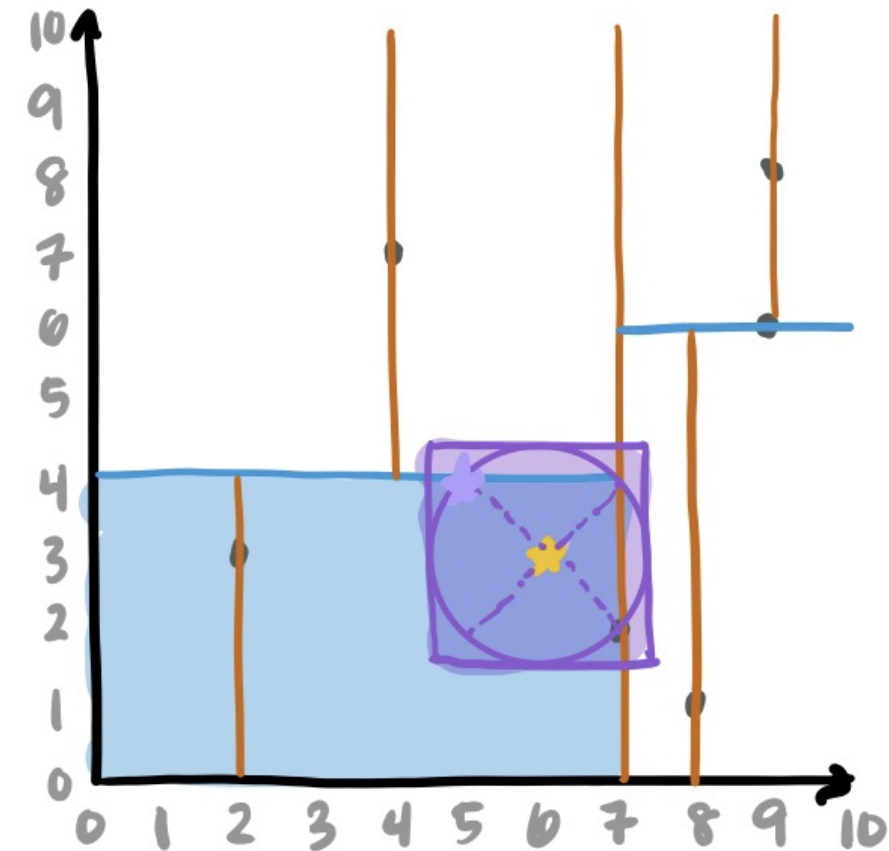
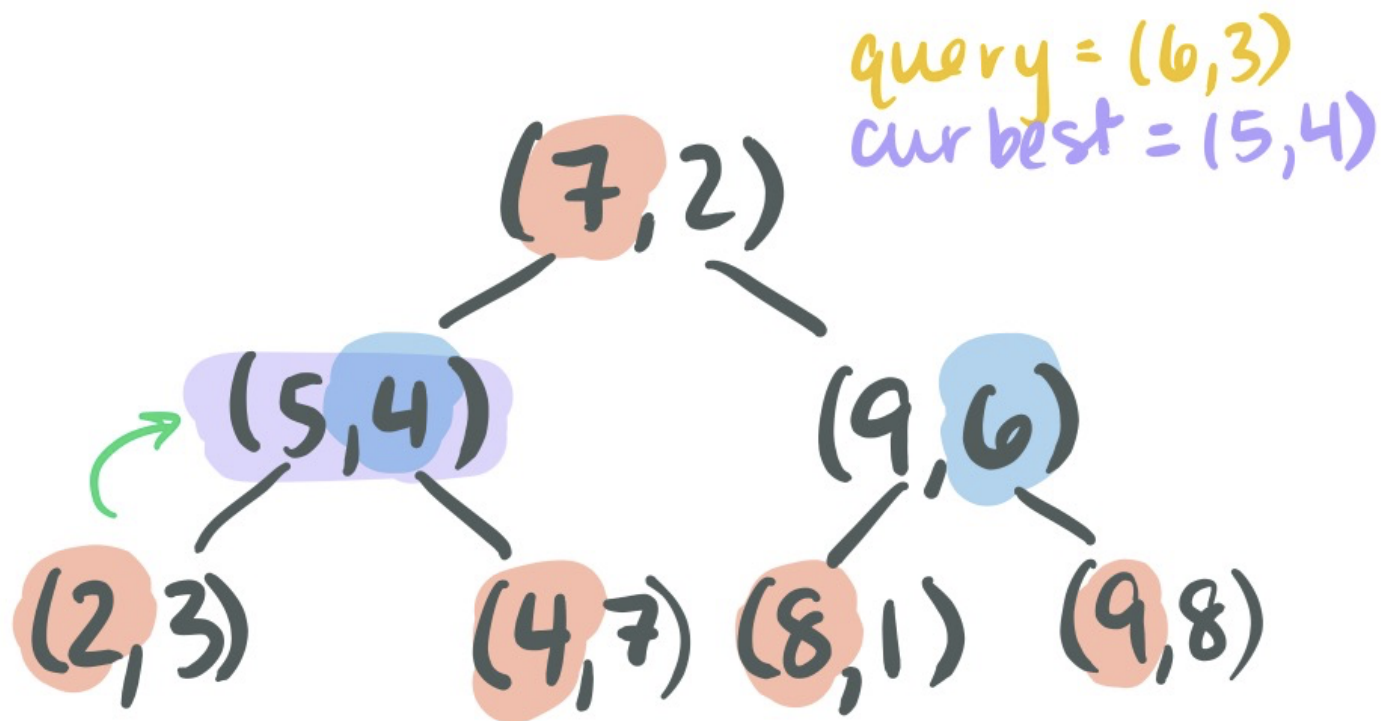


z



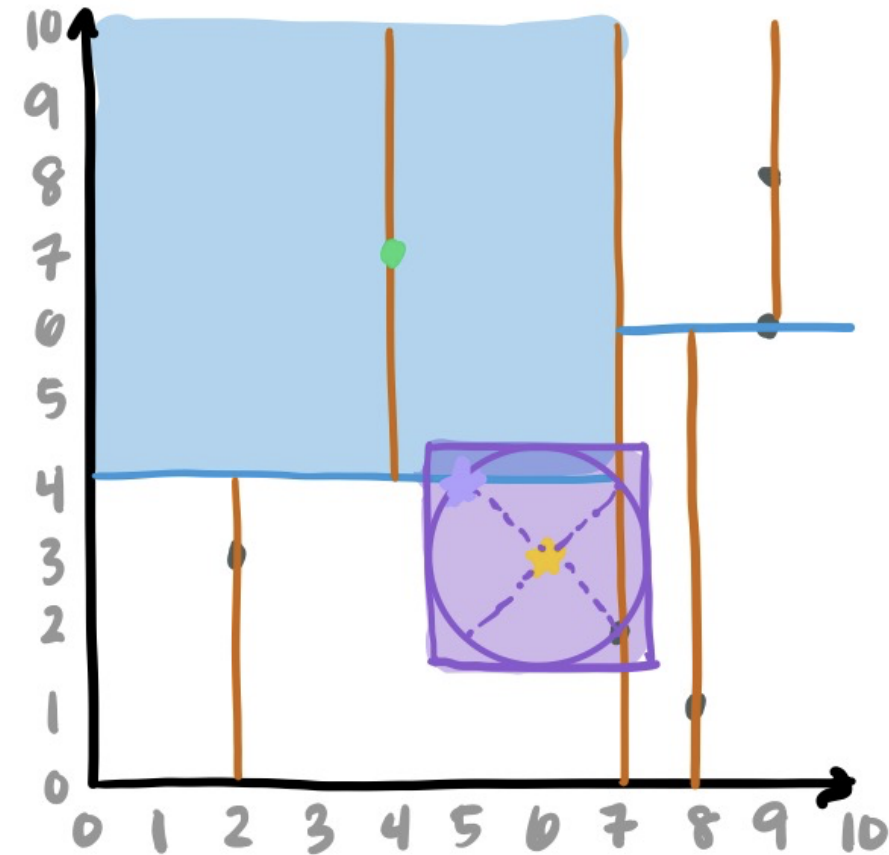
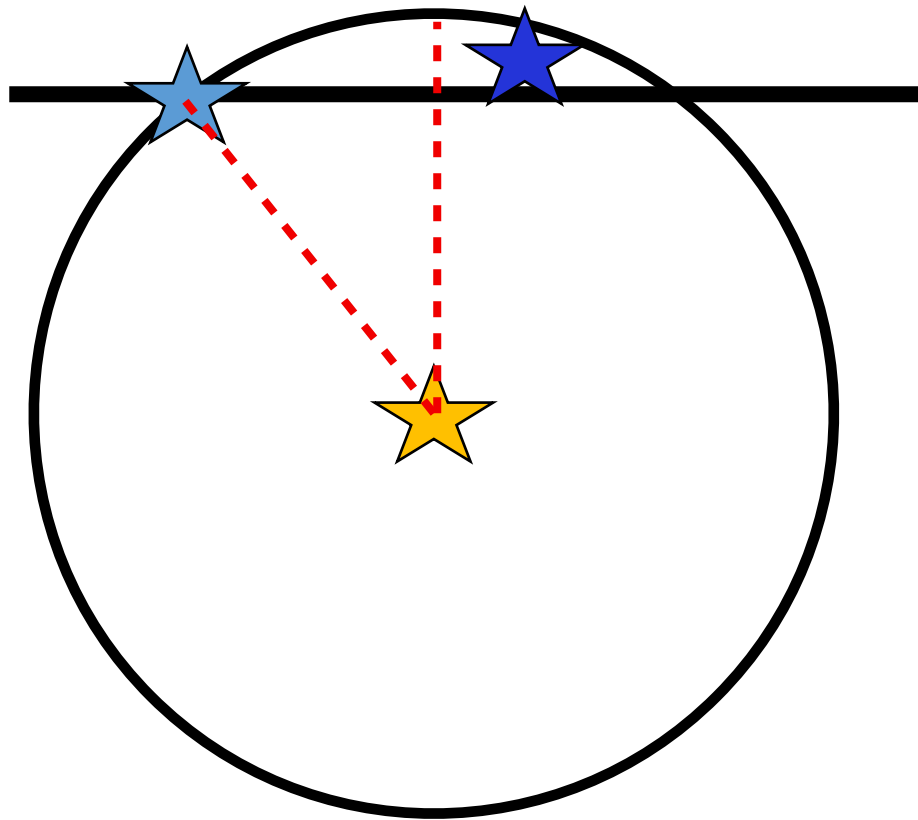
Nearest Neighbor: k-d tree

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

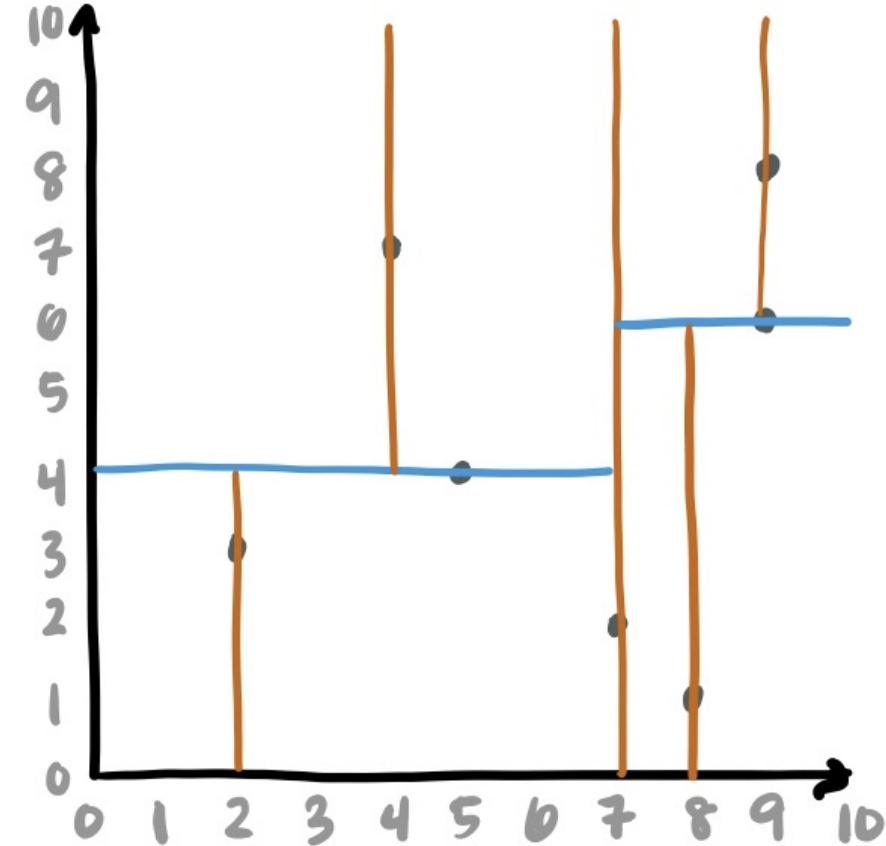
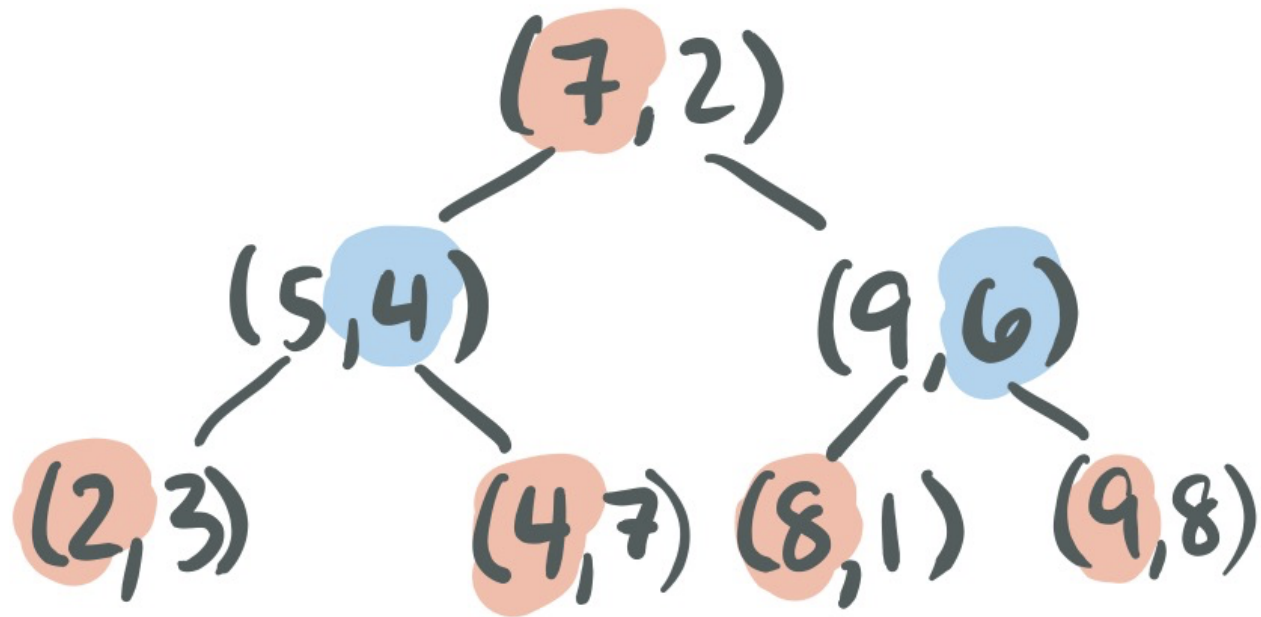


Nearest Neighbor: k-d tree

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



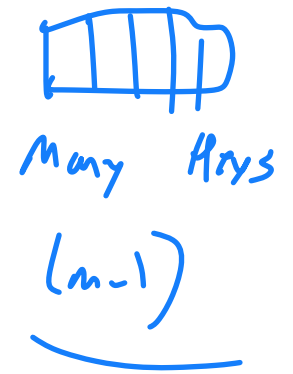
Nearest Neighbor: k-d tree



BTree Properties

A **BTree** 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 **[2, m]** children.

All non-root, internal nodes have **[ceil(m/2), m]** children.

All leaves in the tree are at the same level.

BTree Find

Find(31)

Base Cases:

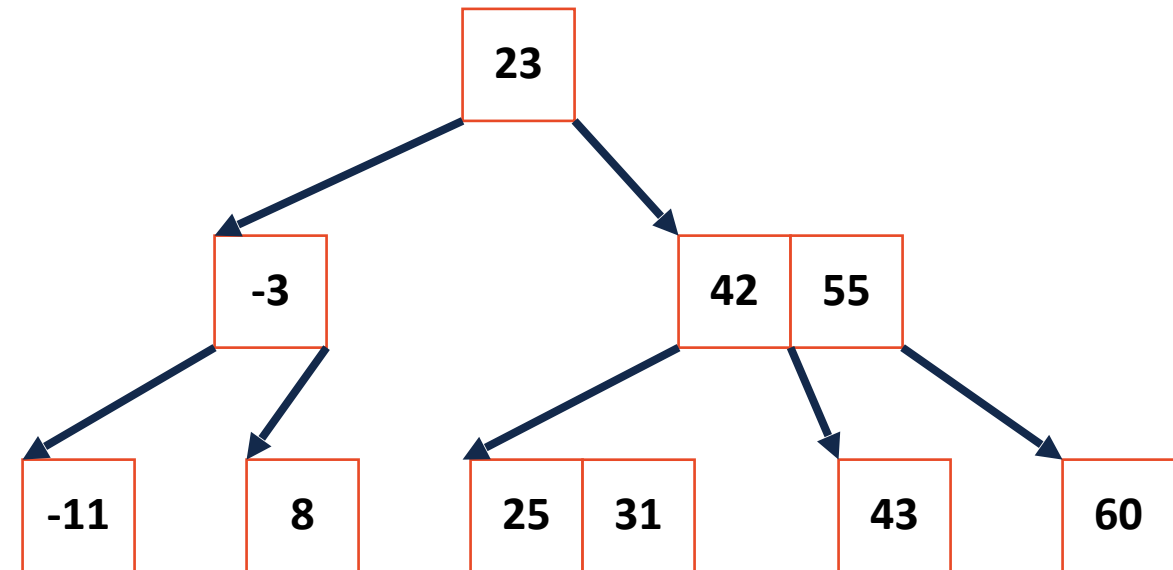
If root is empty, return

If leaf, do array find() and return

Recursive Step:

Array find() for match (then return) or first greater value (and recurse)

Recurse on appropriate child (index of first greater value or last child)



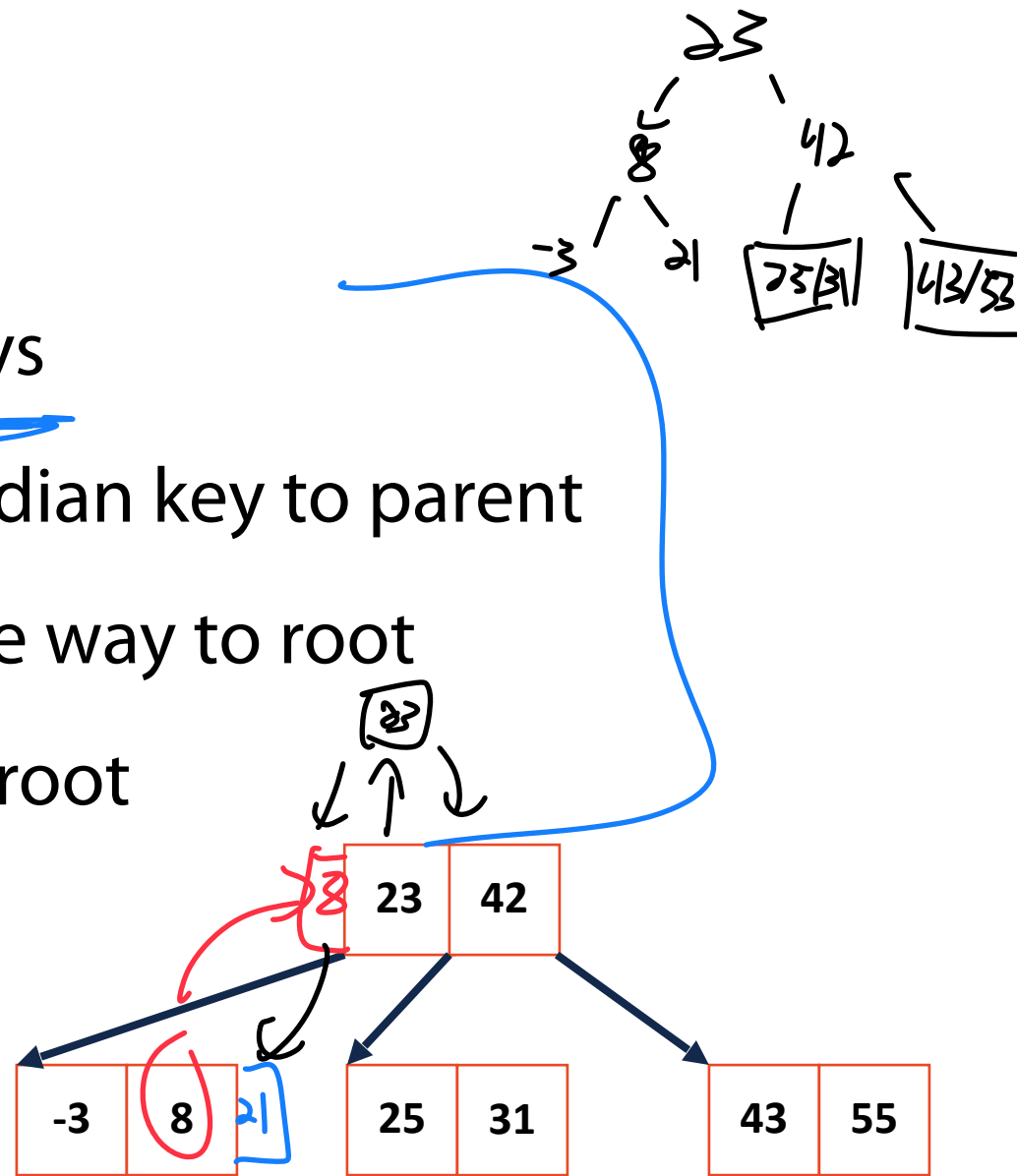
BTree Insertion

Insert (21) , M = 3

- 1) Use BTree **find** to get correct array
- 2) **Insert** item into array in sorted order
- 3) **Check** that array has at most $M - 1$ keys
- 4) If M keys, **split** node and **push up** median key to parent

Check (and push up as needed) all the way to root

If root splits, median becomes a new root



Final Thoughts on Trees

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

There are four coding questions on the practice exam:

Build Perfect Tree

Build Bracket Tree

Build Mirror Tree

Build Ternary Tree

What concepts are these question reinforcing?

Revision is central to trees → Build or find

Build!

A twist on a known tree!

Final Thoughts on Trees

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

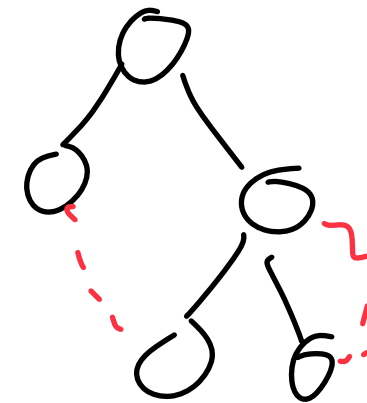
There are four coding questions on the practice exam:

Build Perfect Tree

Build Bracket Tree

Build Mirror Tree

Build Ternary Tree



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

↳ A bidirectional tree

children ↔ parent

→ How does this improve ability to do X?

↳ A tree where every node points to top / #0s

Final Thoughts on FRQs

why

The FRQs for exam 3 / 4 are 'Propose a solution' style problems

Keep in mind the general rubric for these new types of FRQs:

- Optimal Data Structure
- Optimal Data Structure Justified Correctly
- Big O Variables Defined
- Optimal Big O
- Data Structure Interface Described Correctly
- Correct Interface Big O

FRQ Example Rubric

Points	Rubric Item	Description
1	<u>Optimal Data Structure</u>	The answer has the best correct Big O for the overall problem
1	Optimal Big O	The Big Os given are both correct for the data structure chosen and optimal for the problem.
2	Optimal Structure Justified Correctly	There is a logically consistent and correct justification for why the chosen data structure is optimal.
1	Big O Variable(s) Defined	The variable(s) used make sense in the context of the problem and are used appropriately to give Big Os
2	Data Structure Correct Interface	The relevant functions of the interface are described correctly. No implementation details are needed , just a clear idea of what the functions are doing.
2	Correct Interface Usage	It is clear how the interface is being used to solve the problem
1	Correct Interface Big O	Every function described has the correct Big O for the data structure chosen (even if the data structure itself is not optimal).

even if
 $O(1)$ is
 continue

Not
 implementation

Whoops! Pointed out to me after lecture that the example I ran through I didn't actually define the Big O variable despite saying do this!

[It's a common mistake].

For the 3D matrix problem, my Big O would probably be:

"Let's say we have Q spaces in a warehouse and we expect to get N items added to the warehouse and M items removed over the runtime"

My Big O would be $O(N+M)$ as every add or remove is $O(1)$.

My runtime is not affected by Q

Range Search over Shifting numeric values



10%

20%

10%

30%

	Unsorted Array	Sorted Array	Unsorted Linked List	Stack	Queue	AVL	B Tree
Find	$O(n)$	<u>$O(\log n)$</u>	$O(n)$	—	—	$O(h) = O(\log n)$	$O(\log n)$
Insert	$O(1)^* / O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(h) = O(\log n)$	$O(\log n)$
Remove	$O(1)^* / O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(h) = O(\log n)$	$O(\log n)$
Traverse	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$





Open Office Hours!