Data Structures

Exam 3 Review

CS 225

Brad Solomon

March 13, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Have a great spring break!

(Enjoy the snow?)

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

See a short proof on min heap properties (and tradeoffs)

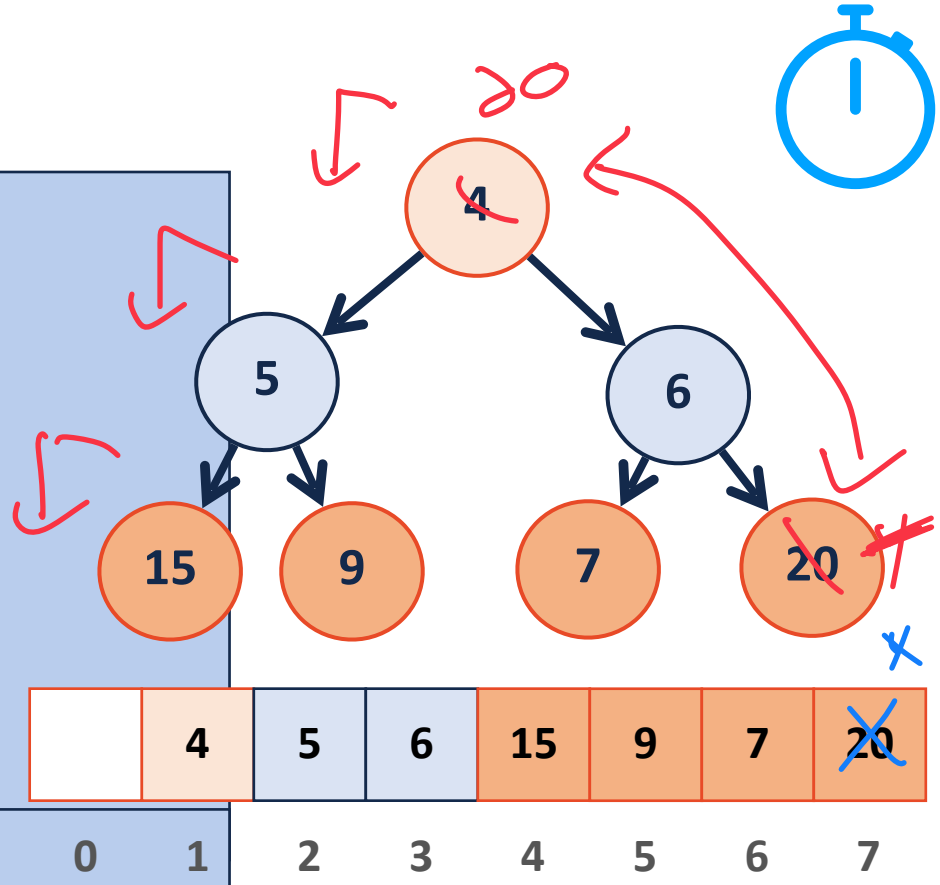
Review concepts on exam 3

Enjoy your spring break!

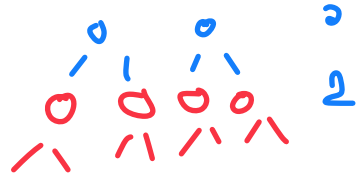
removeMin - heapifyDown

```
1  template <class T>
2  T Heap<T>::_removeMin() {
3    // Swap with the last value
4    T minValue = item_[1];
5    item_[1] = item_[size_ - 1];
6    size--;
7
8    // Restore the heap property
9    _heapifyDown(1);
10
11   // Return the minimum value
12   return minValue;
13 }
```

```
1  template <class T>
2  void Heap<T>::_heapifyDown(size_t index) {
3    if ( !_isLeaf(index) ) {
4      size_t minChildIndex = _minChild(index);
5
6      if ( item_[index] > item_[minChildIndex] ) {
7         std::swap( item_[index], item_[minChildIndex] );
8
9         _heapifyDown( minChildIndex );
10     }
11   }
12 }
```

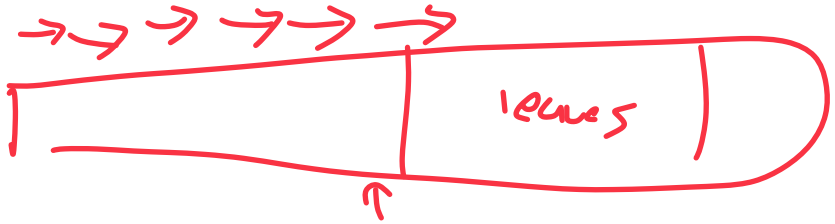


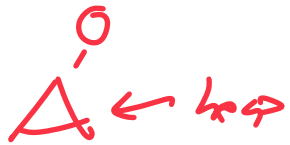
Proving buildHeap Running Time



Theorem: The running time of buildHeap on array of size **n** is: O(n)

Strategy:





1) We call heapifyDown() on every non-leaf node

2) HeapifyDown() has a runtime based on the height of the node

By (1) and (2), our runtime is simply the sum of all heights

Proving buildHeap Running Time

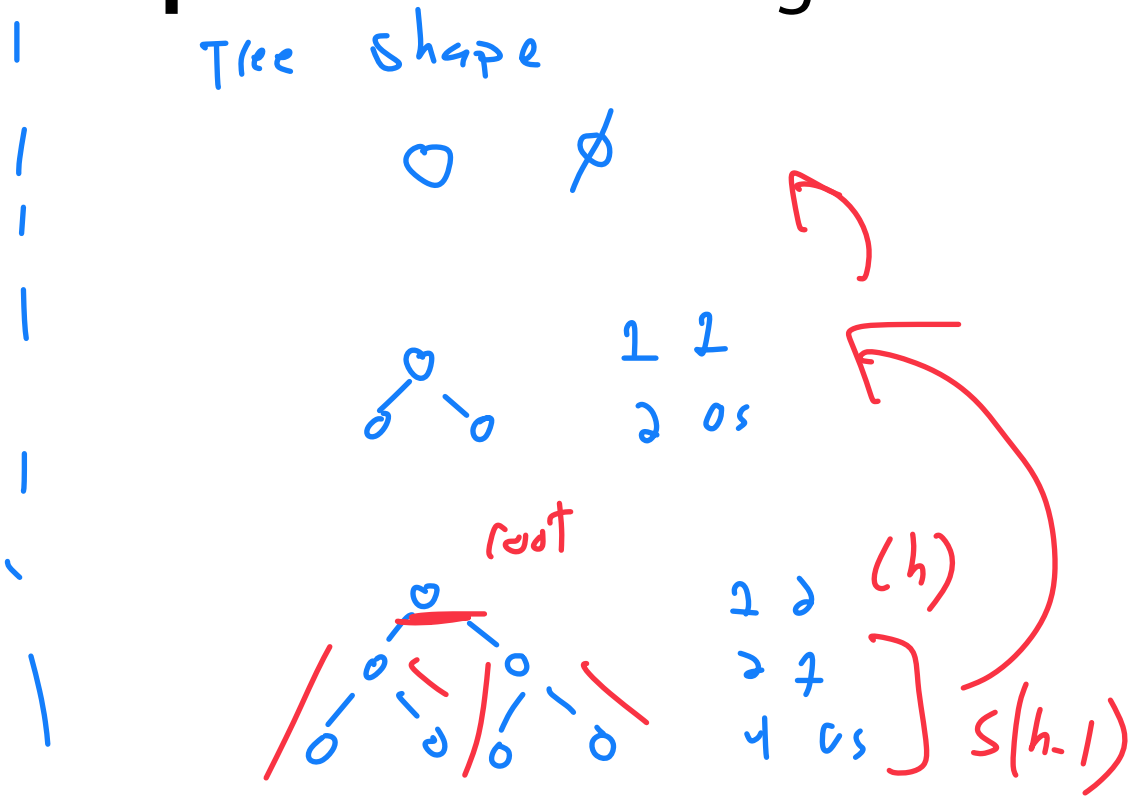
S(h): Sum of the heights of all nodes in a **perfect** tree of height **h**.

S(0) = 0

S(1) = 1

S(2) = 1 * 2 + 2 * 1 = 4

S(h) = h + 2S(h-1)



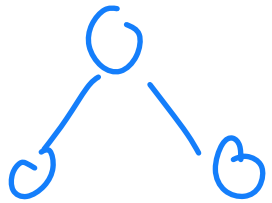
Proving buildHeap Running Time

unrolled ↓ last slides equation

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Base Case:

$h = 0$  $2^{0+1} - 2 - 0 = 0$ ✓

$h = 1$  $2^{1+1} - 2 - 1$

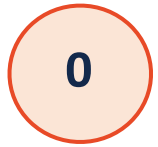
$4 - 3 = 1$ ✓

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Base Case:

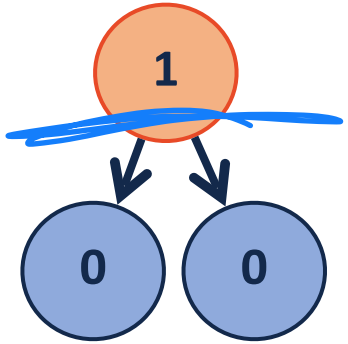
h = 0



$2^{0+1} - 2 - 0 = 0$

vs

h = 1



$2^{1+1} - 2 - 1 = 1$

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Induction Step: $S(i)$ is true for all $i < h$

Show true for $i = h$

$2^{i+1} - 2 - i \quad (i = h-1)$

$S(h) = h + 2S(h-1)$

Proving buildHeap Running Time

Claim: Sum of heights of all nodes in a perfect tree: $S(h) = 2^{h+1} - 2 - h$

Induction Step: $S(i) = i + 2\ S(i-1)$ is true for all values $i < h$

$$S(h-1) = 2^{h-1+1} - 2 - (h-1) = 2^h - h - 1 \quad \text{(By IH)}$$

$$S(h) = h + 2\ S(h-1) = h + \left(2\ (2^h - h - 1)\right) \quad \text{(Plug in)}$$

$h-1$

$$S(h) = 2^{h+1} - 2 - h \quad \text{(Simplify)}$$

$2 \cdot 2^h - 2h - 2$

$2^{h+1} - 2 - h$

Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is O(n)

$$S(h) = 2^{h+1} - 2 - h$$

How can we relate **h** and **n**? $h \leq O(\log n)$

How can we estimate running time?

Proving buildHeap Running Time

Theorem: The running time of buildHeap on array of size **n** is O(n)

$$S(h) = 2^{h+1} - 2 - h$$

How can we relate **h** and **n**? $h \leq log\ n$

How can we estimate running time?

$2^{log\ n+1} - 2 - log\ n$ (Plug in)

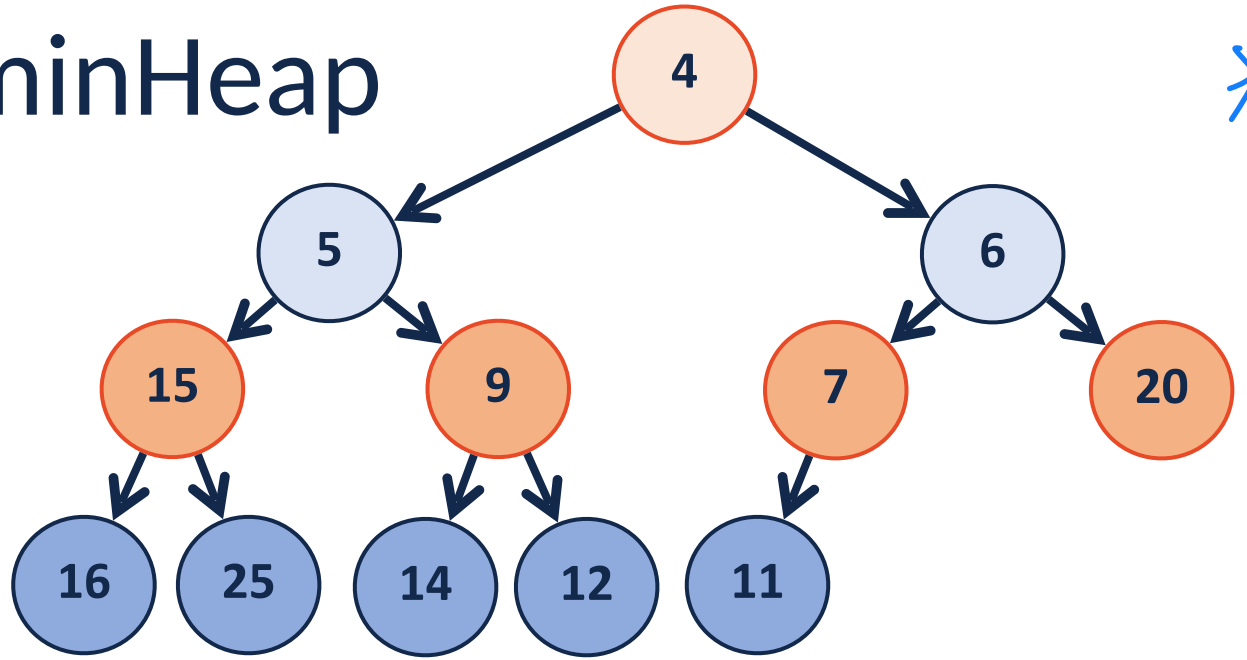
$2 * 2^{log_2\ n} - 2 - log\ n$ (Simplify)

$2\ n - log\ n - 2 \approx O(n)$ (Rearrange)

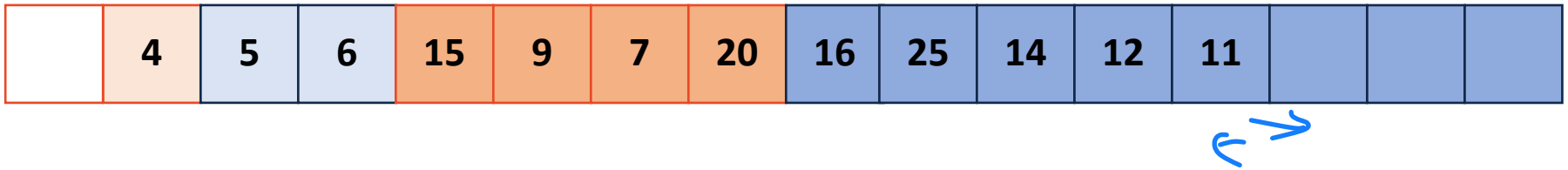
what we have proven

Sum of decreasing $d(\log n) \to 1$ (heap) is $O(n)$

minHeap



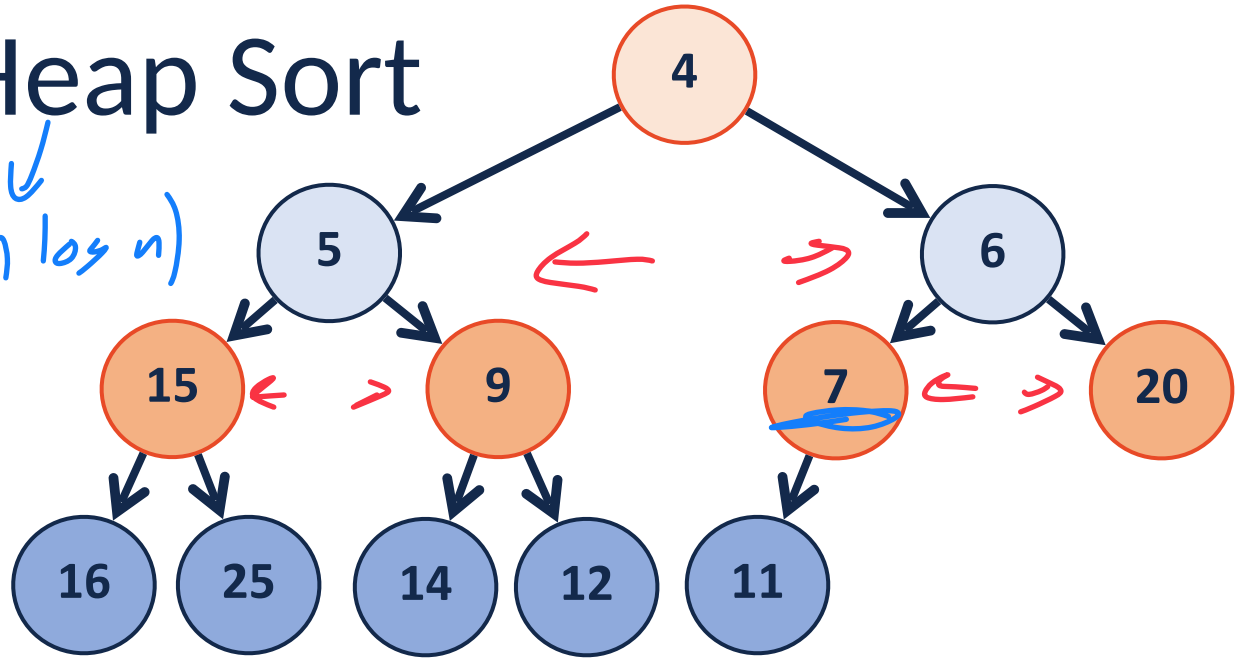
1. Construction
   - O(1)
2. Insert
   - O(log n)
3. RemoveMin
   - O(log n)



minHeap is a good example of tradeoffs:

- Array efficient insert/removeMark()
- Array swaps
- Improved construction time!

Heap Sort



1. Build Heap $O(n)$
2. Call removeMin() n times
   - insert back on array
3. Reverse the array as needed

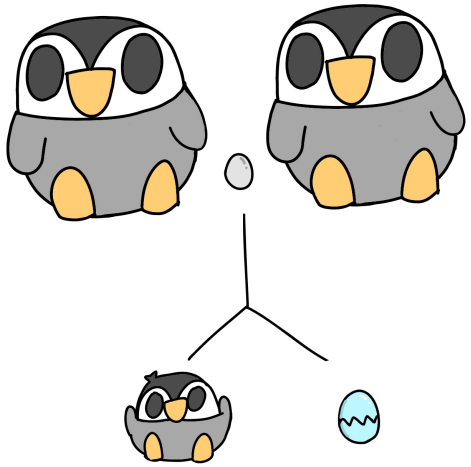
$O(n \log n)$

$O(n)$

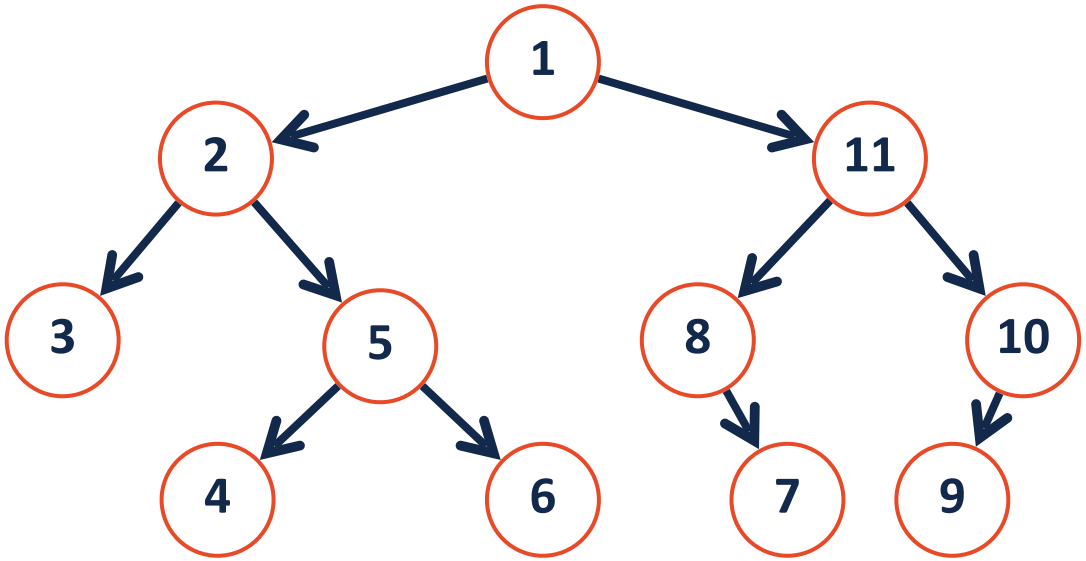
Running time? $O(n \log n)$

Heaps is not on exam. Any questions?

Trees



Tree Traversals

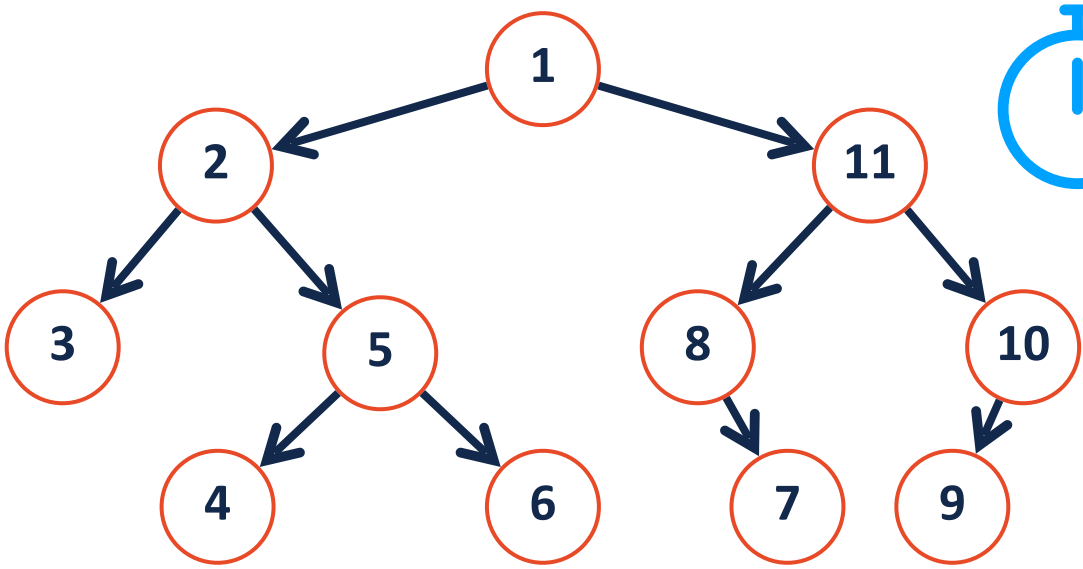


Pre-order:

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

(Assume left → right)

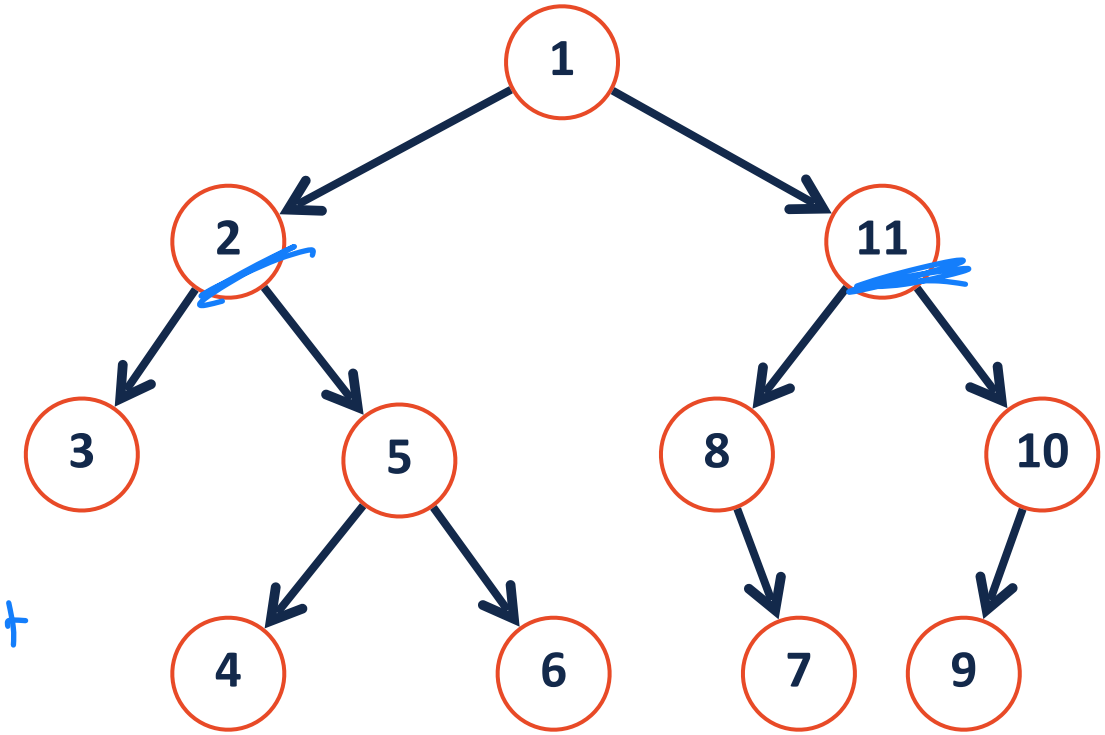
Explore as far along one path as possible before backtracking

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

important recursion concept



Stack: ~~1~~ 11 2

Print: 1

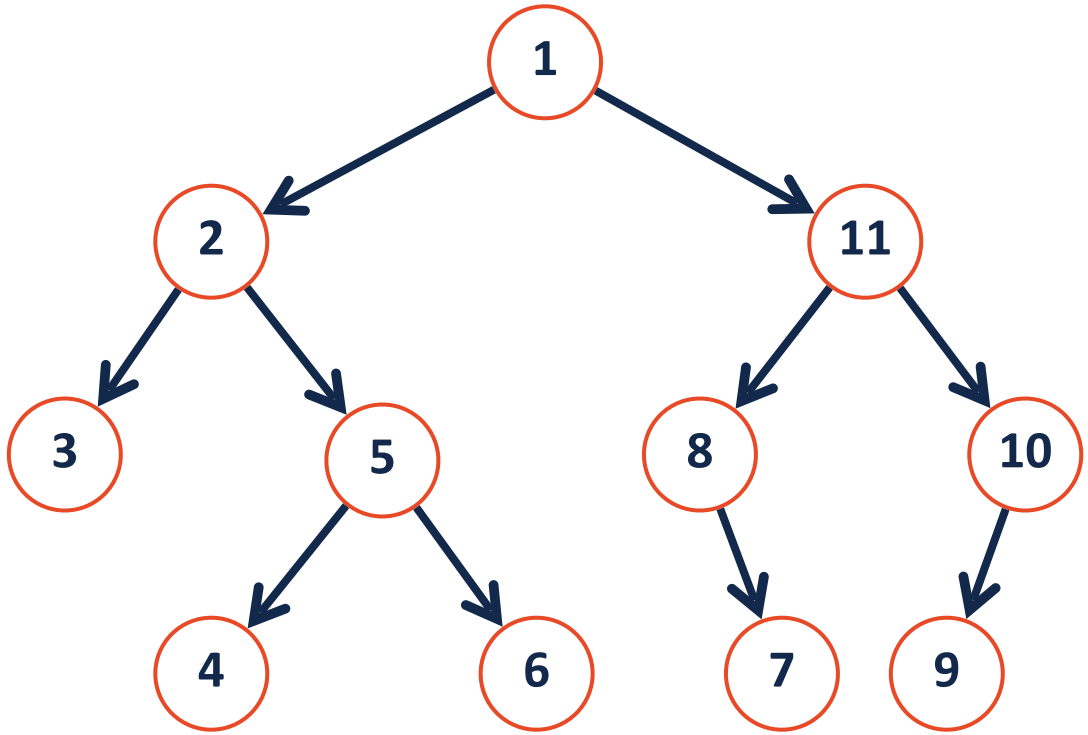
Depth First Search

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: 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

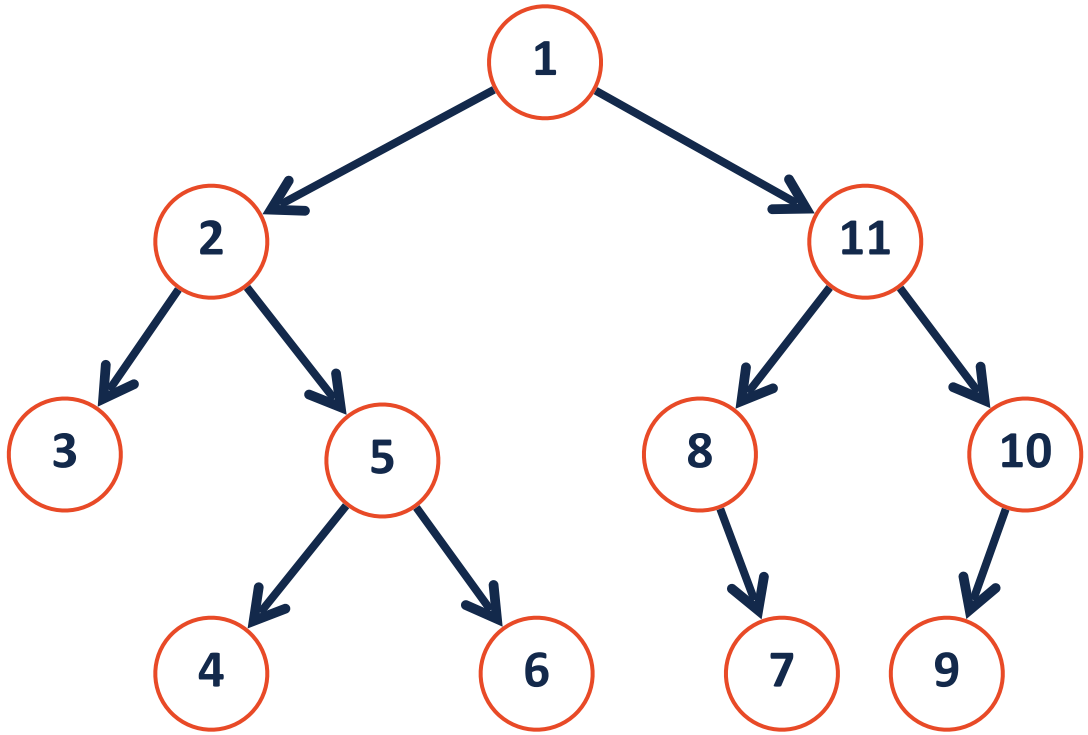
While queue isn't empty:

  Dequeue front element (as `tmp`)

  Print `tmp`

  Enqueue `tmp->left`

  Enqueue `tmp->right`



Queue:

Print:

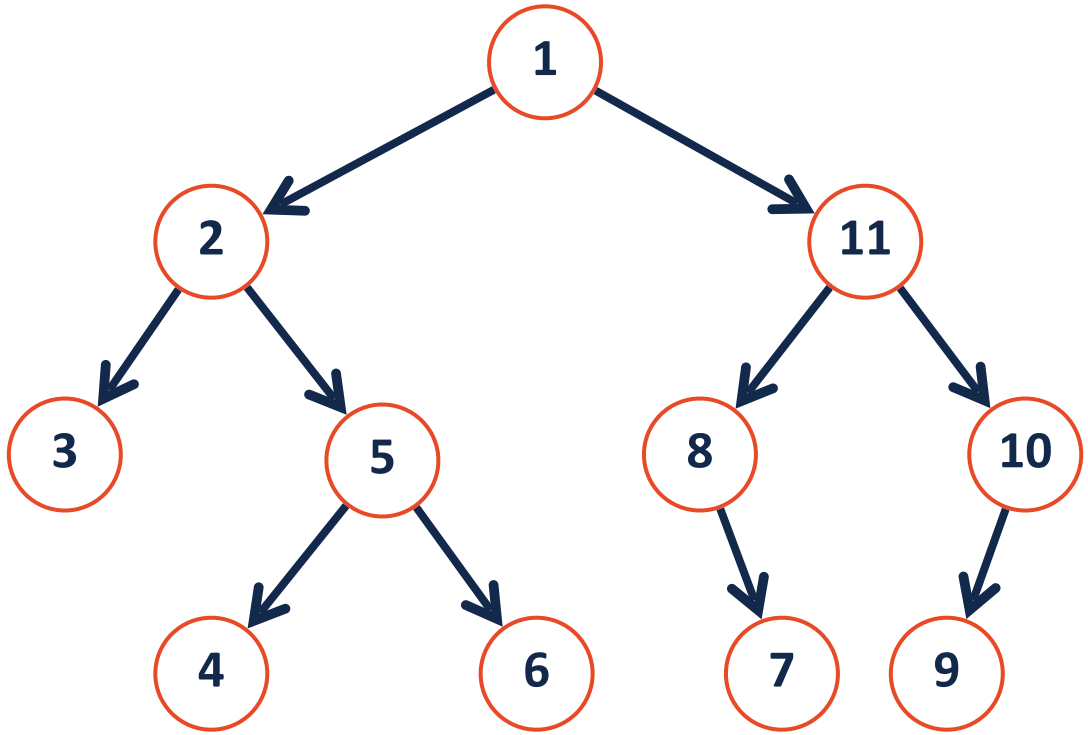
Breadth First Search

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`

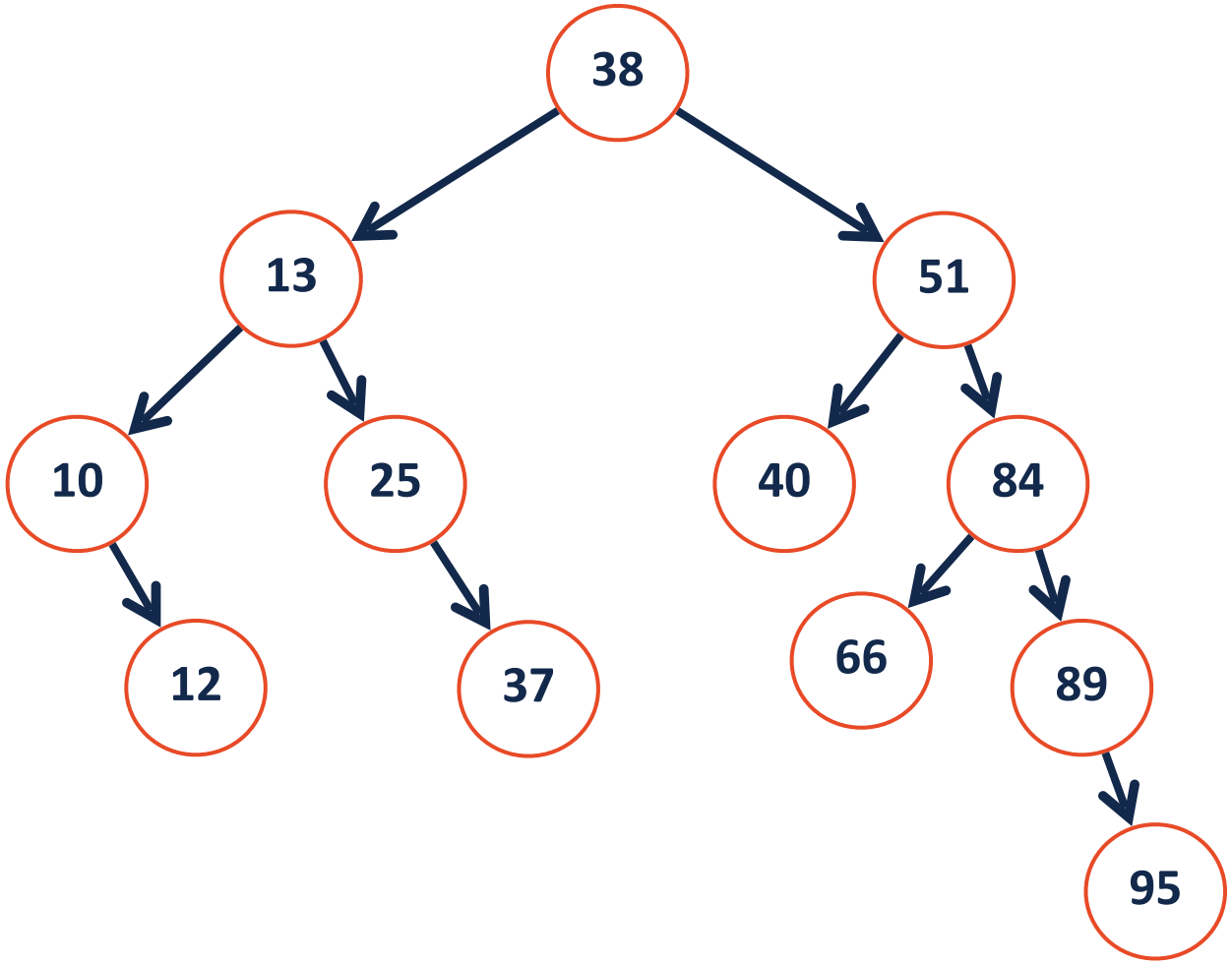


Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

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

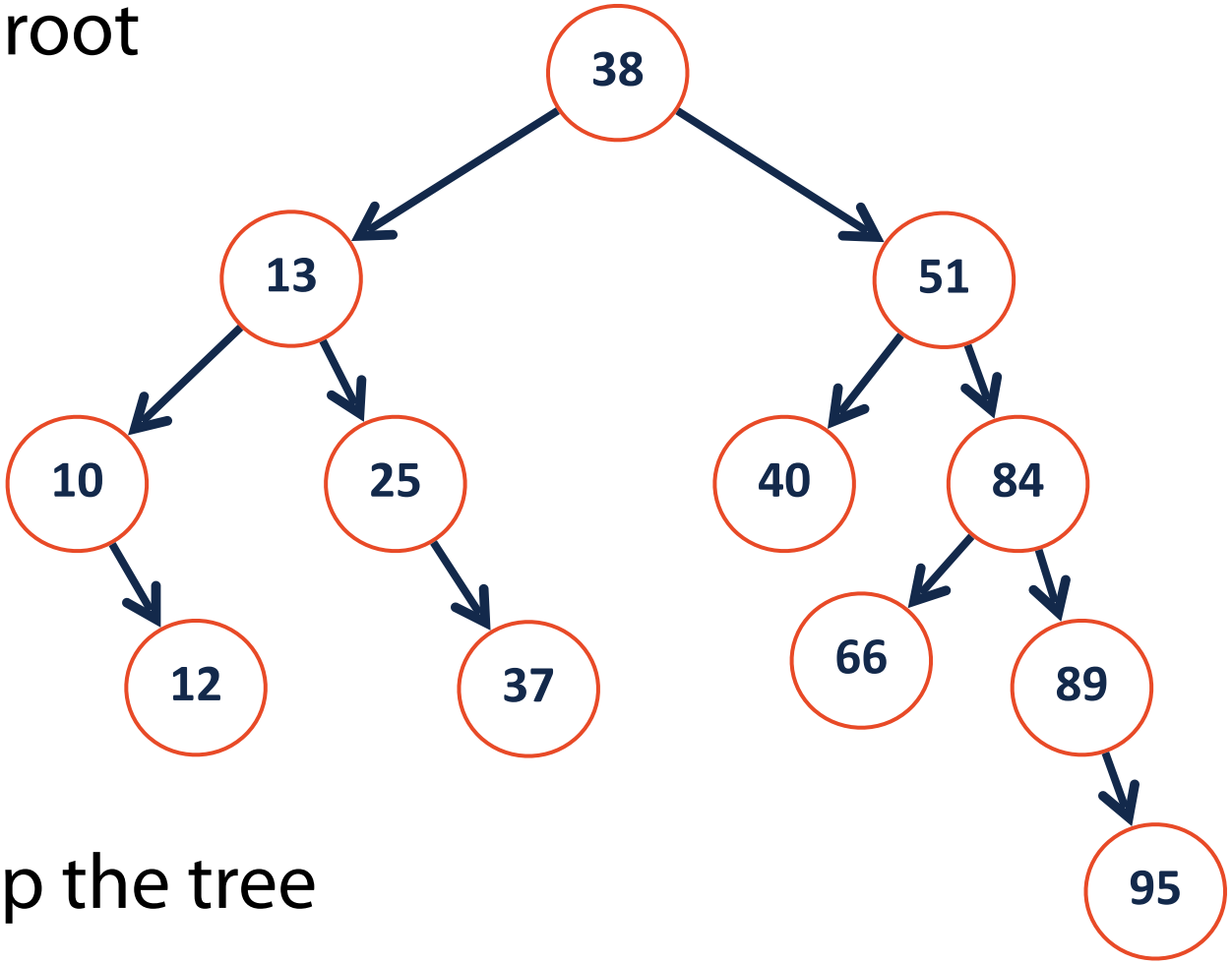
Recursion:

`tmp < query`, recurse right

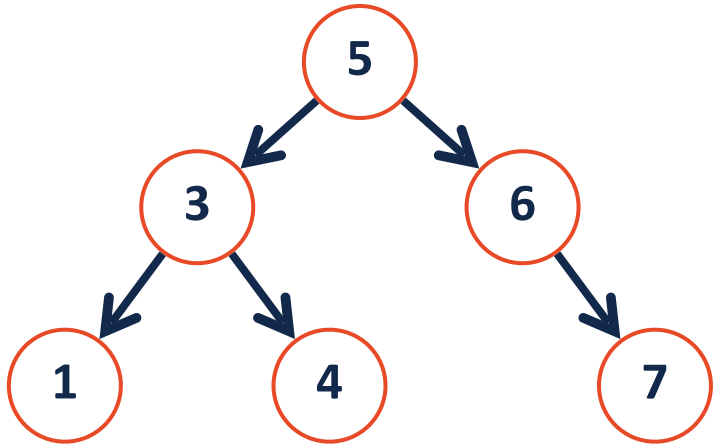
`tmp > query`, recurse left

Combining:

Return the recursive value back up the tree

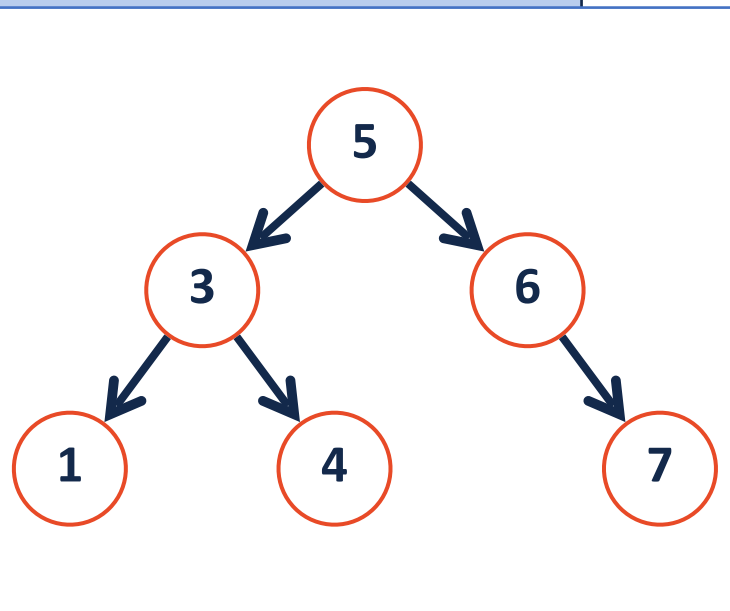


```
1  template<typename K, typename V>
2
3        TreeNode *&          _find(TreeNode *& root, const K & key) {
4
5
6  // Base Case
7  if(root == nullptr || root->key == key){
8      return root;
9  }
10
11 // Recursive Step ("Combining step" is 'return')
12 if (root->key > key){
13     return _find(root->left,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



```
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
```



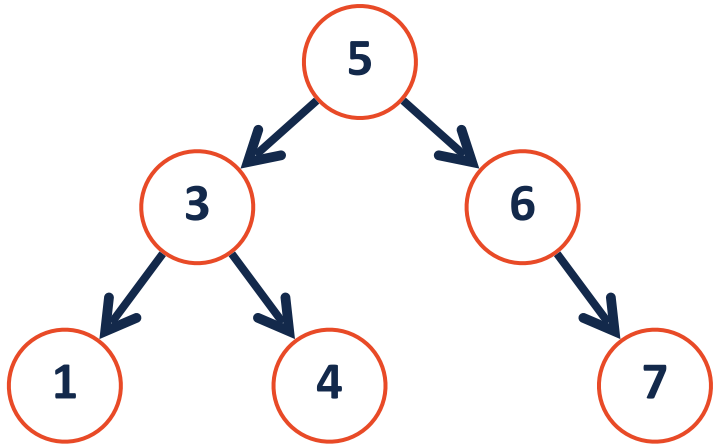
```
1  template<typename K, typename V>
2
3  void _remove(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }
```

0 child

1 child

2 child

↳ IOP / IOS



BST Analysis – Running Time

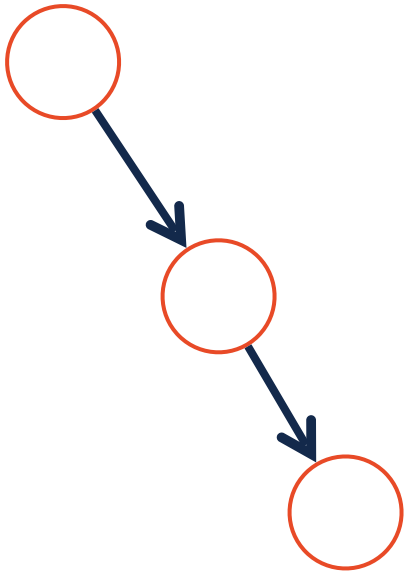
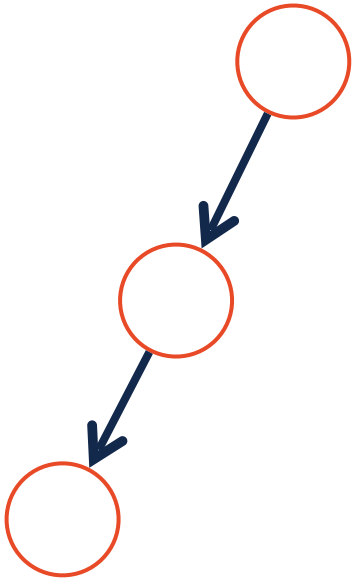
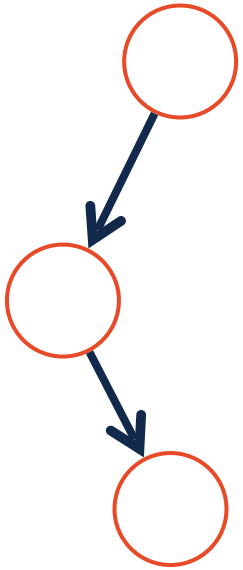
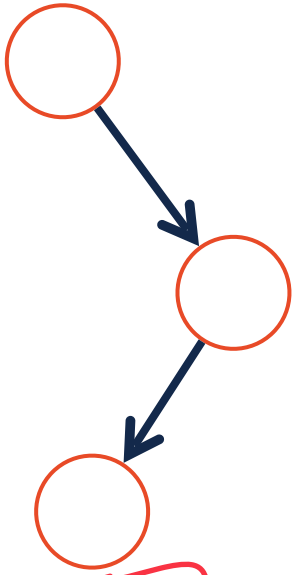
Operation	BST Worst Case
find	
insert	
remove	
traverse	

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)

search trees generally imply this

AVL Rotations

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

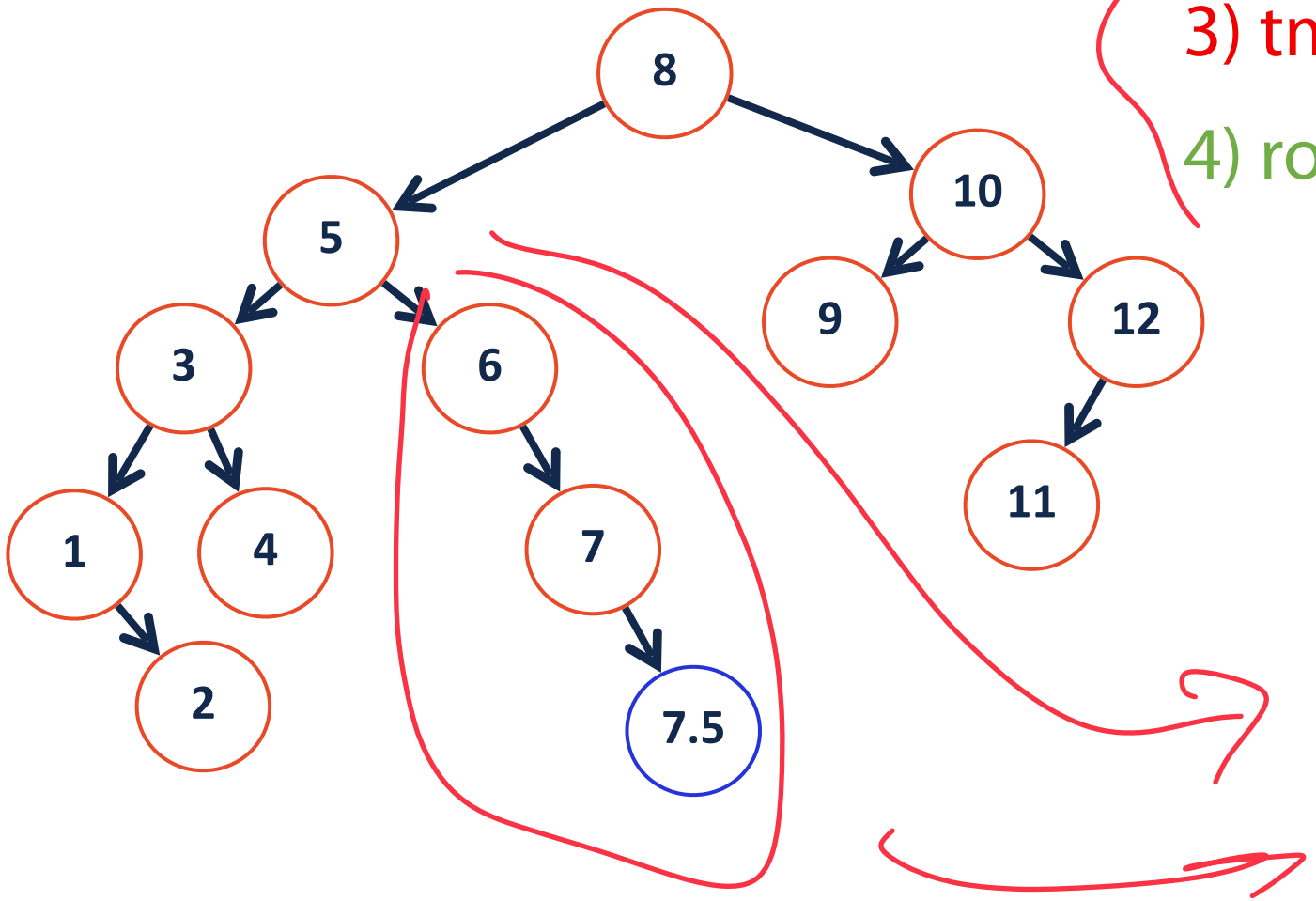
Left Rotation

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

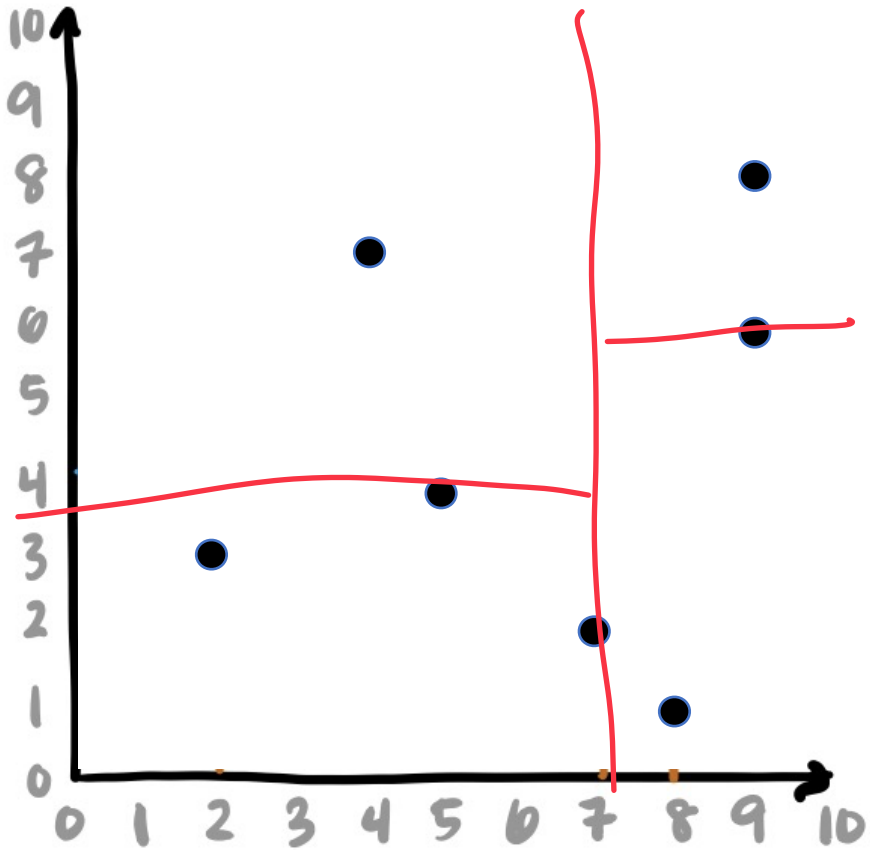
Goal: Maintain AVL Tree balance

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

Nearest Neighbor: k-d tree

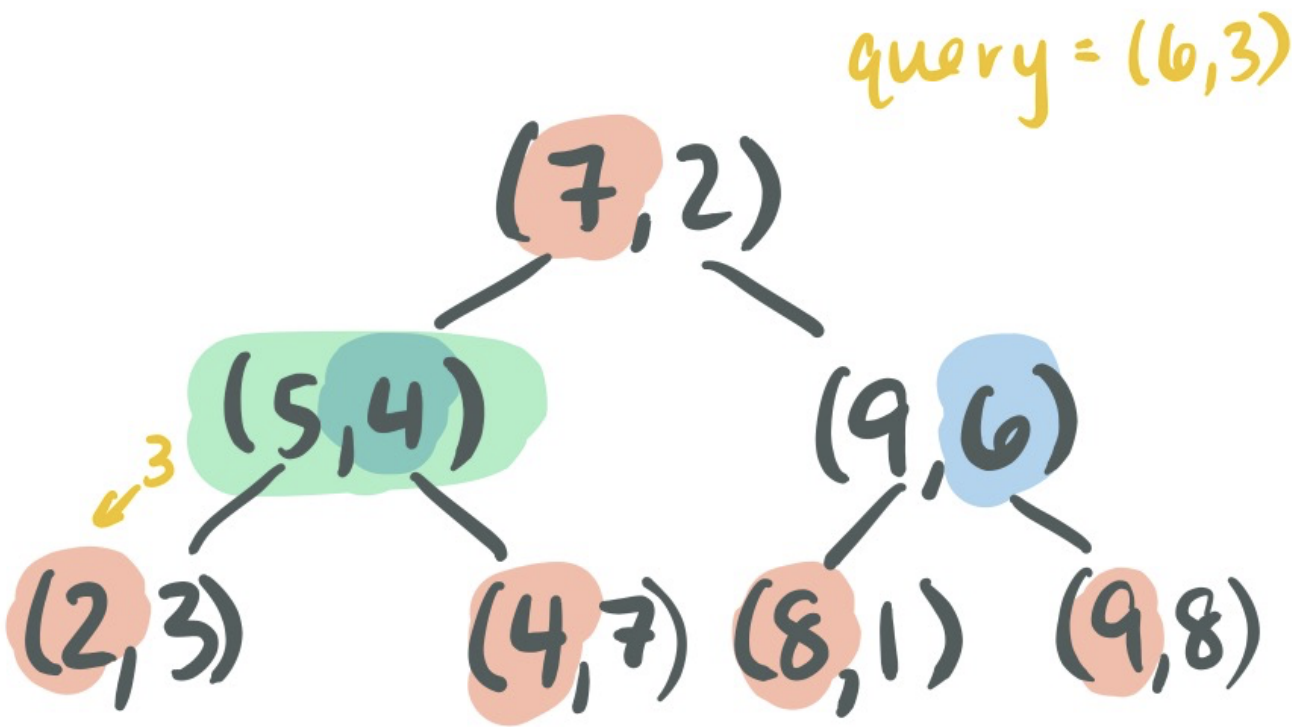
A **k-d tree** splits a multi-dimensional dataset on points:

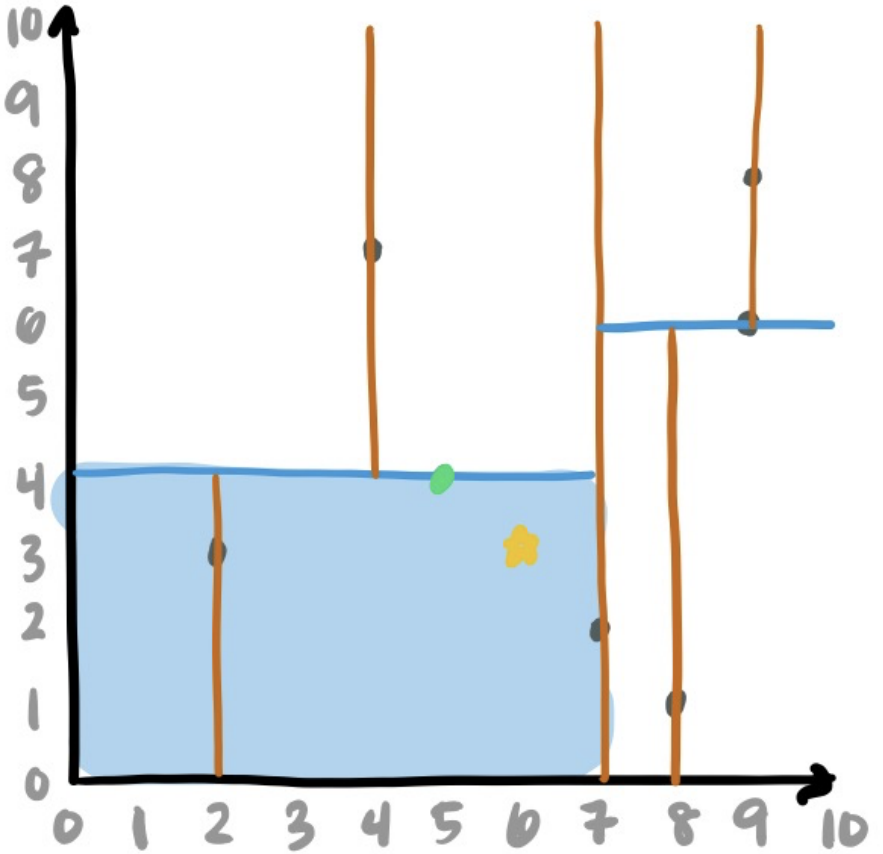
(7,2), (5,4), (9,6), (4,7), (2,3), (8,1), (9,8)



Nearest Neighbor: k-d tree

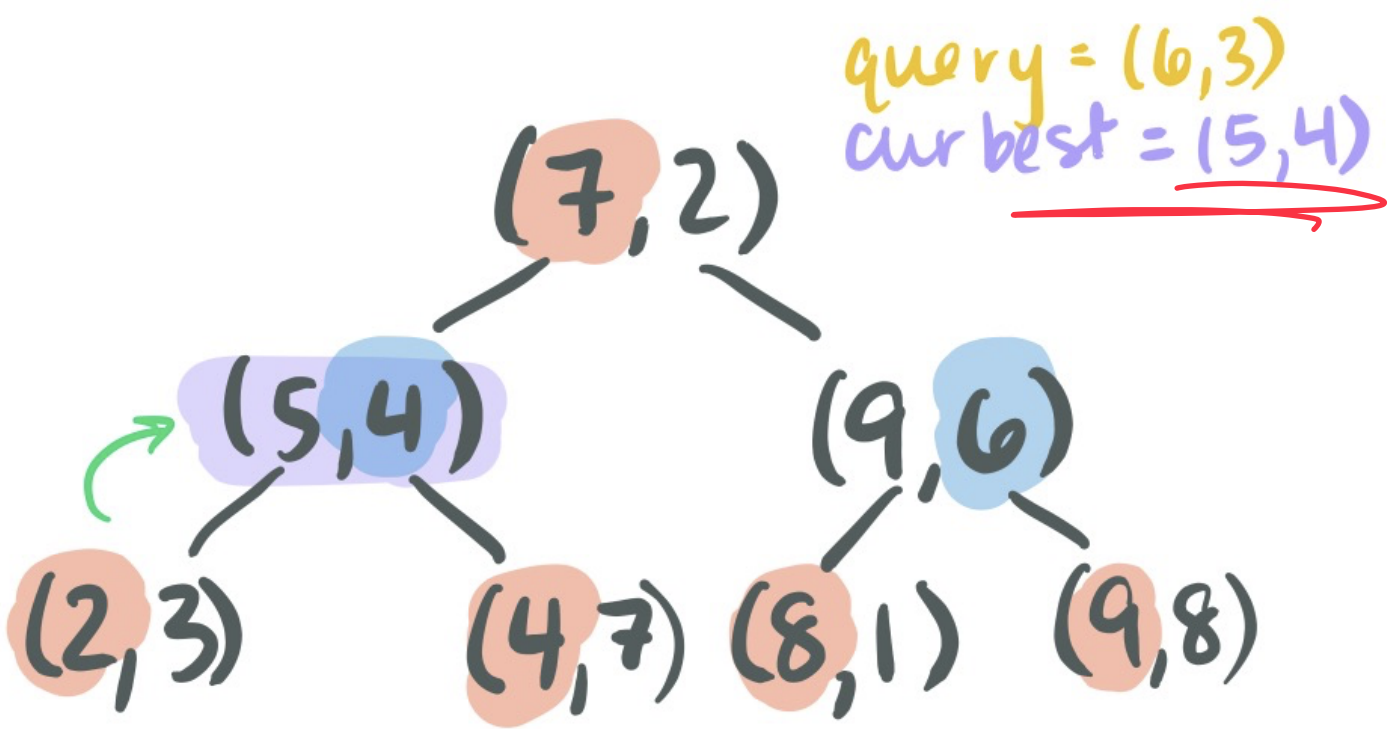
Search by comparing query and node in single **alternating** dimension

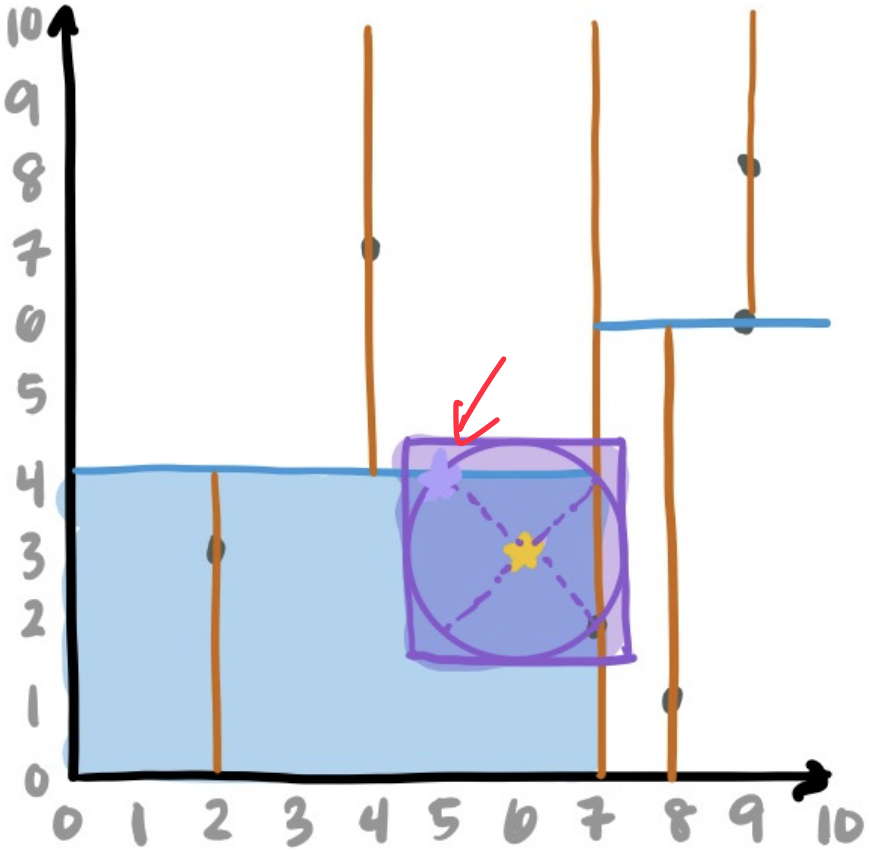




Nearest Neighbor: k-d tree

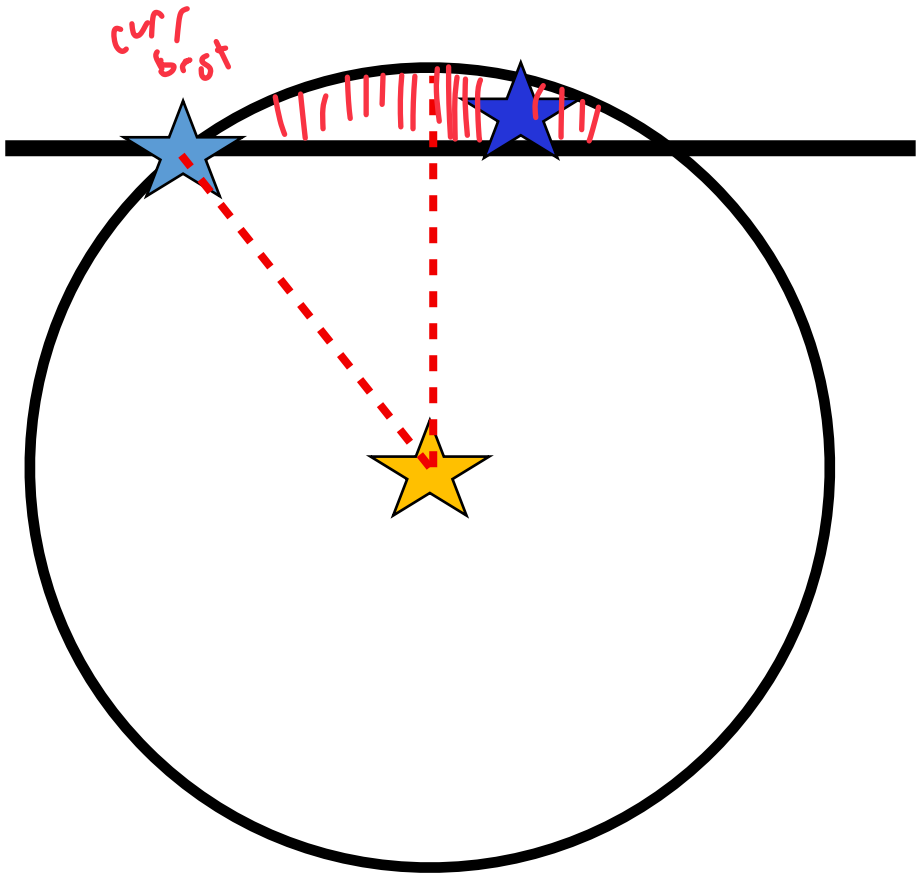
Backtracking: start recursing backwards -- store "best" possibility as you trace back

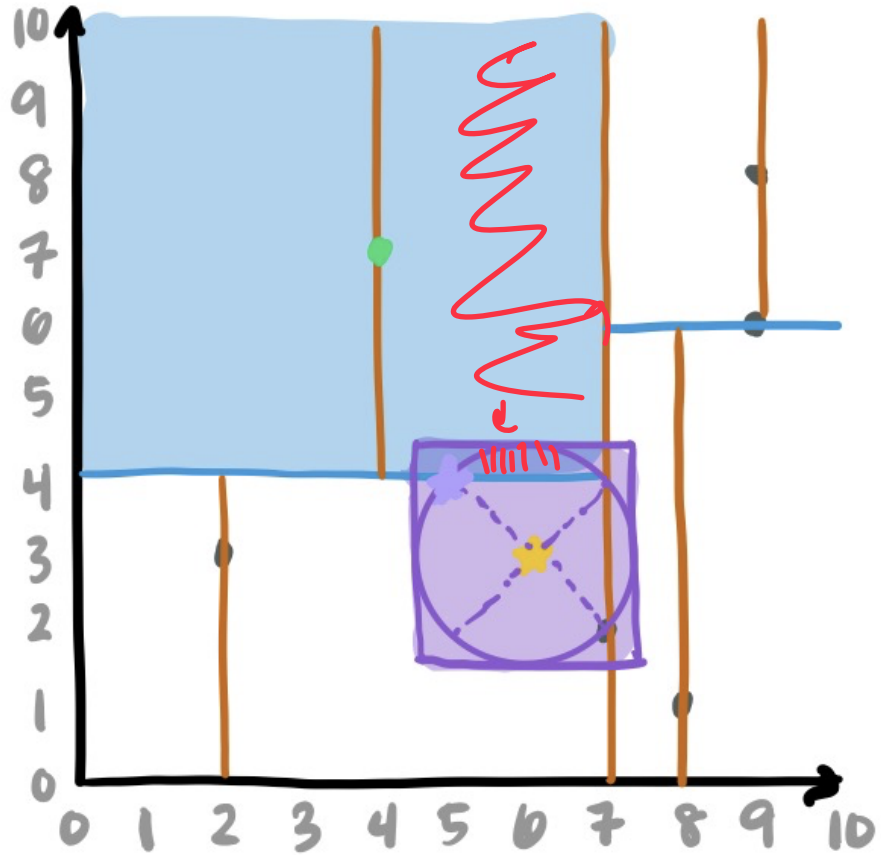




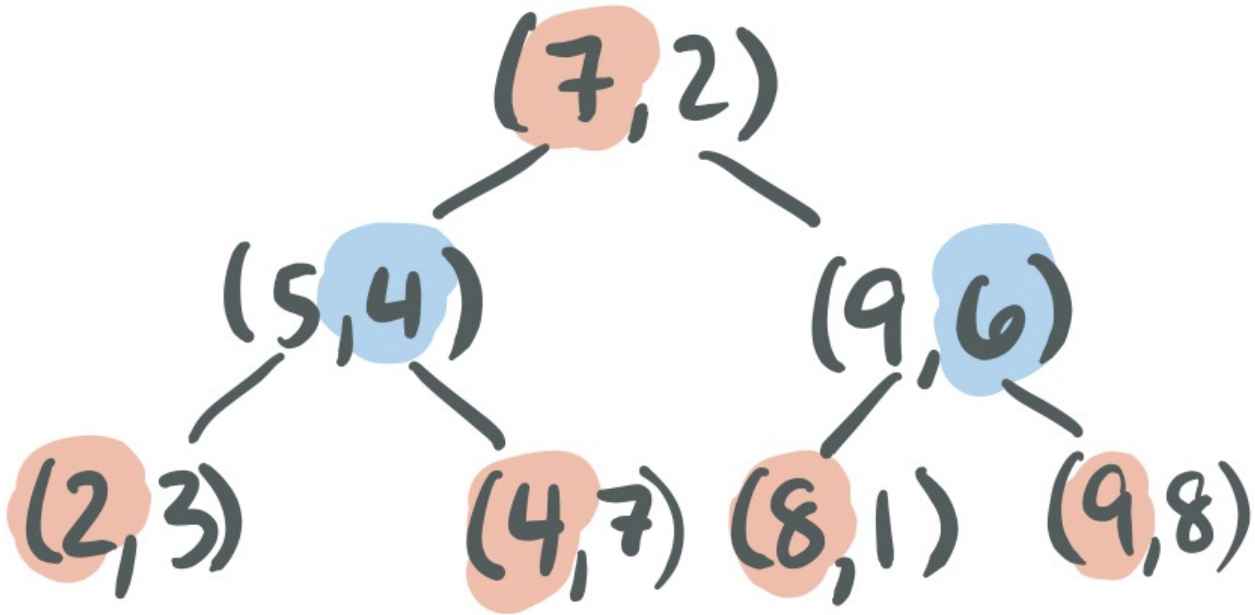
Nearest Neighbor: k-d tree

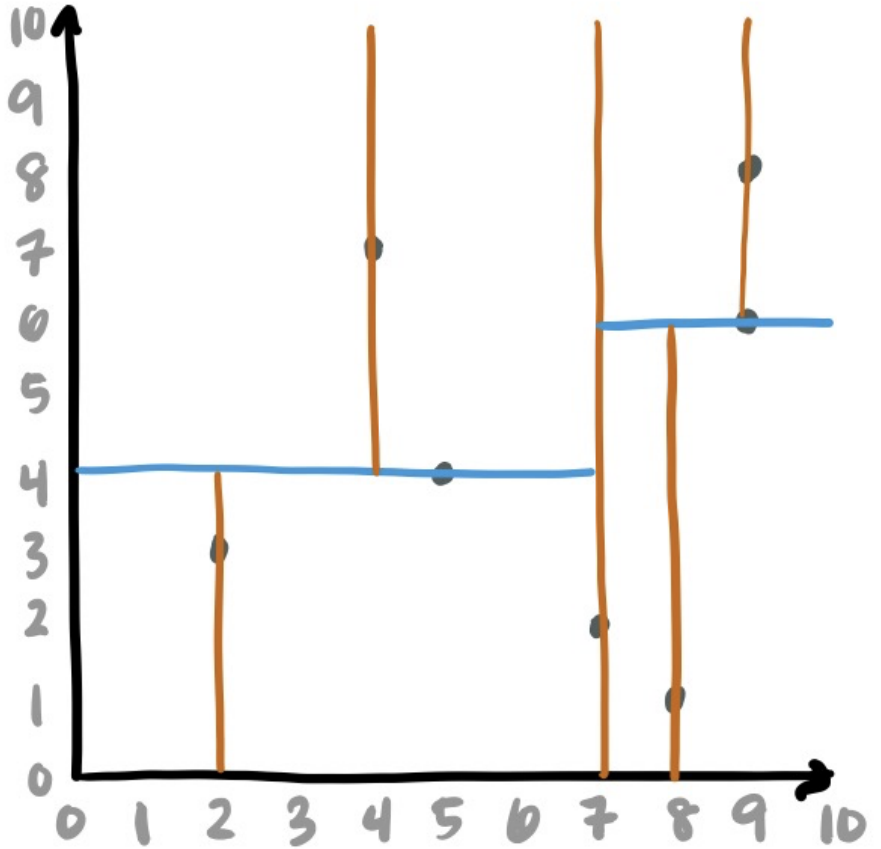
May have to recursively check other branches of tree — **why?**





Nearest Neighbor: k-d tree





BTree Properties

(Engineering focused optimization)

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 $[2, M)$ children.

All non-root, internal nodes have $\left[\lceil \frac{m}{2} \rceil, m\right)$ children.

insert guides this

All leaves in the tree are at the same level.

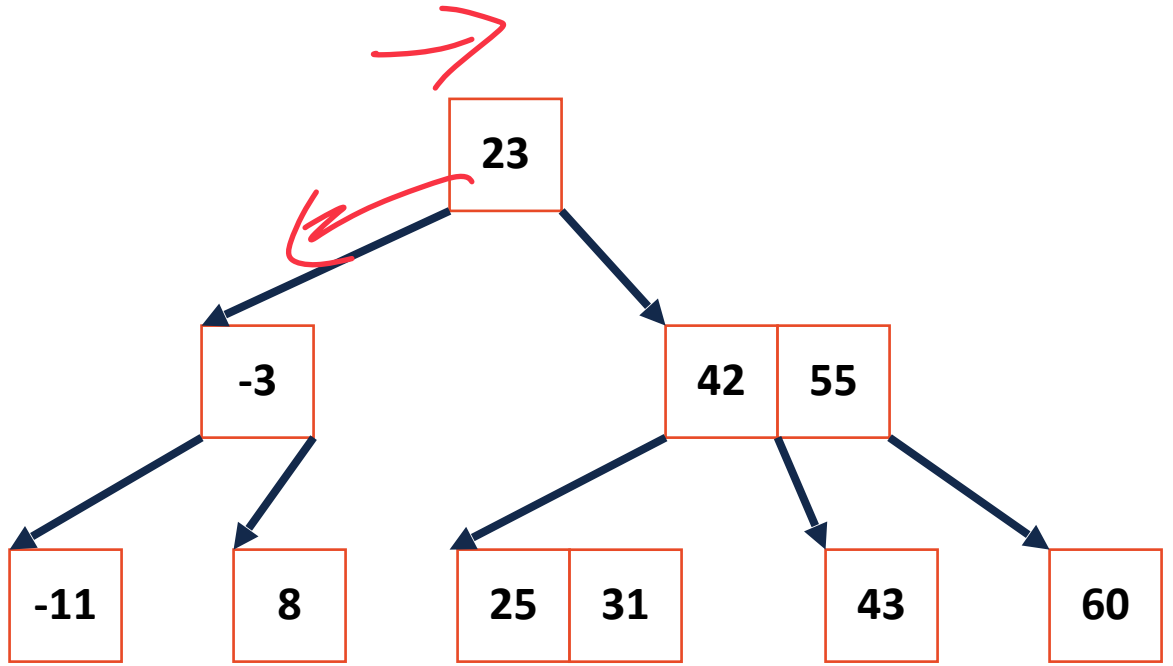
BTree Find

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

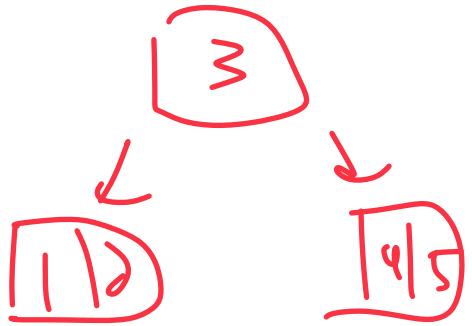
Tip: Index of first greater value is index of child we want to visit!

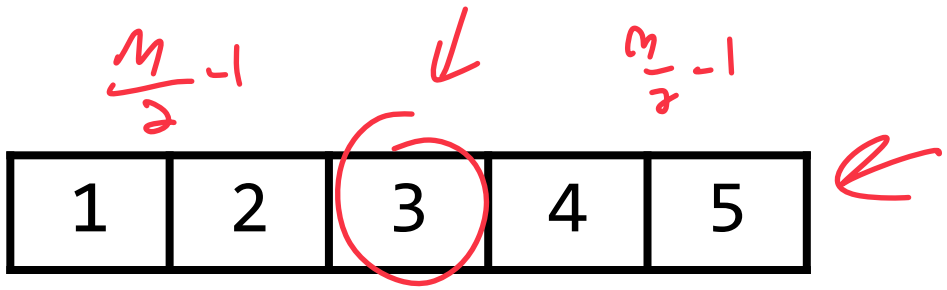
BTree Insertion

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)
```

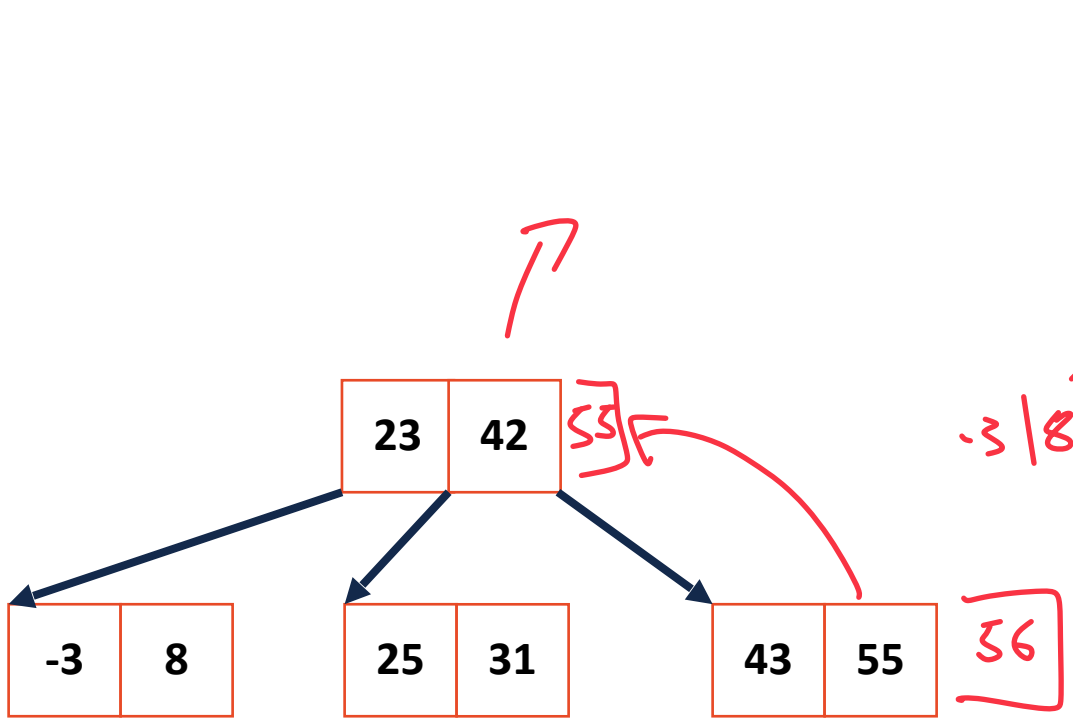


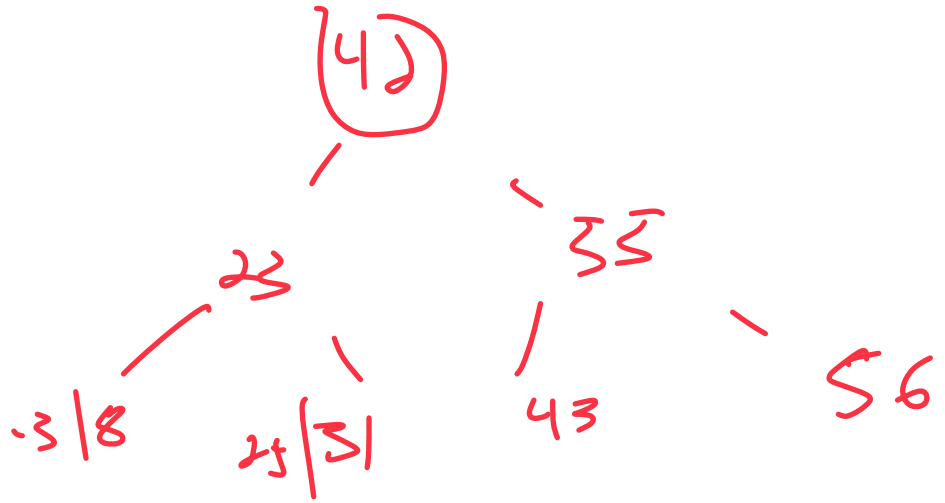


BTree Recursive Insert

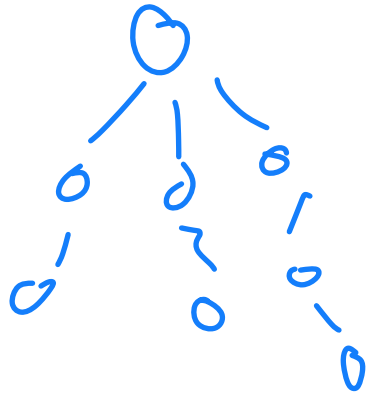
`Insert(56), M = 3`

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?

↳ Building a tree of unusual shape w/ a twist

↳ Recursion

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?

Final Thoughts on FRQs

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