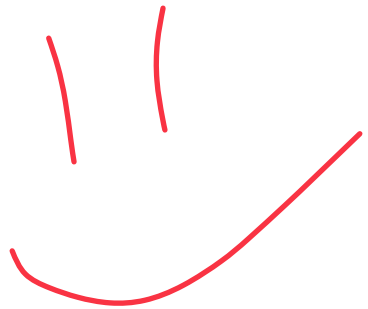
Data Structures

Binary Search Trees

CS 225

Brad Solomon

September 20, 2023



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Apologies for async lecture!

Ideally Monday lecture will be in person!

If you have questions, please post on Piazza or Discord!

Learning Objectives

Explore implementations of DFS and BFS on binary trees

Extend binary trees into binary *search* trees

Build conceptual and coding understanding of BST

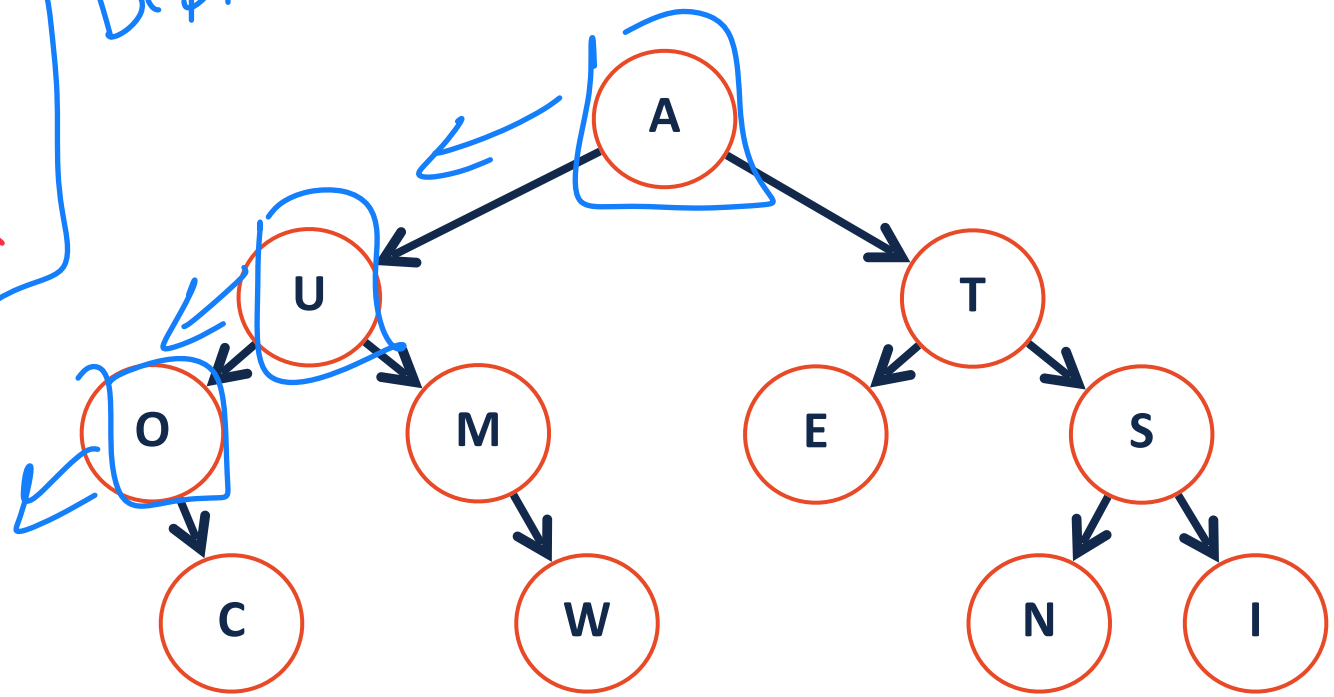
Tree Search

sol'n to search was traversal!

There are two main approaches to searching a binary tree:

Pre Order
In Order
Post order

Depth first search



Depth First Search

Explore as far along one path as possible before backtracking

Make a stack, initialize to root

While stack not empty

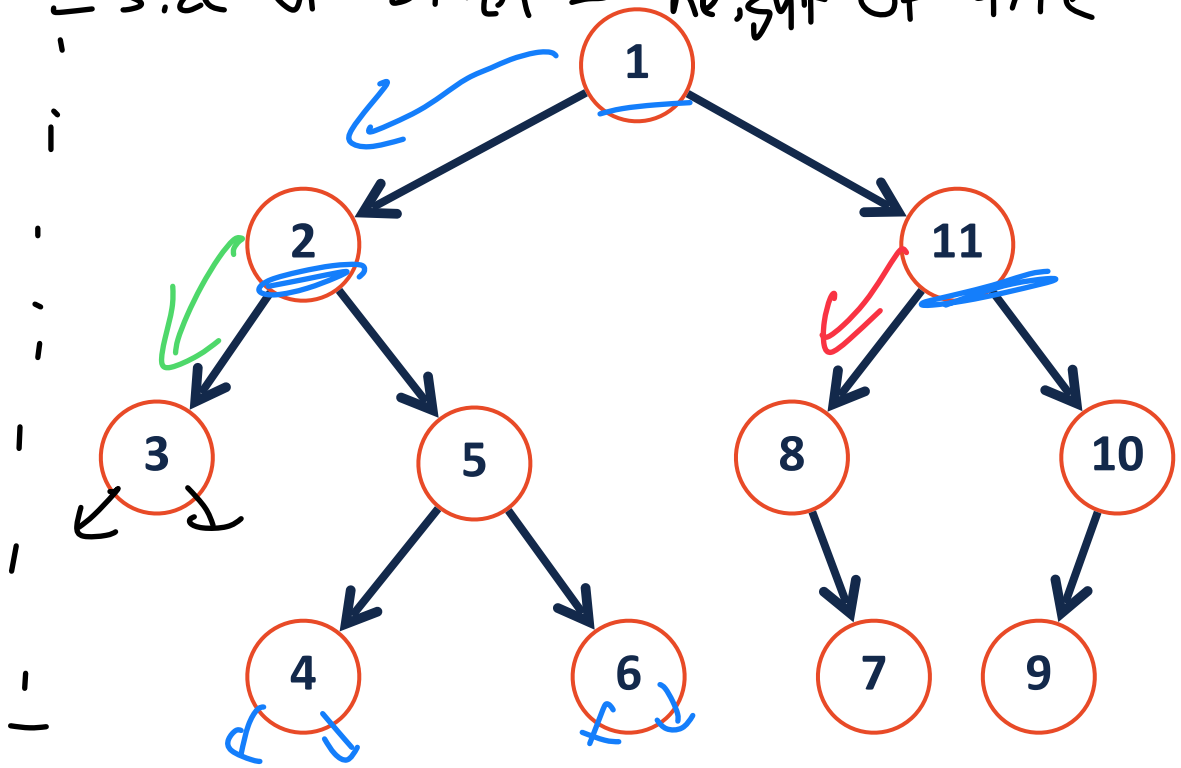
  Pop the top element (as tmp)

  Print tmp

  Push tmp -> right

  Push tmp -> left

size of stack ≃ height of tree



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

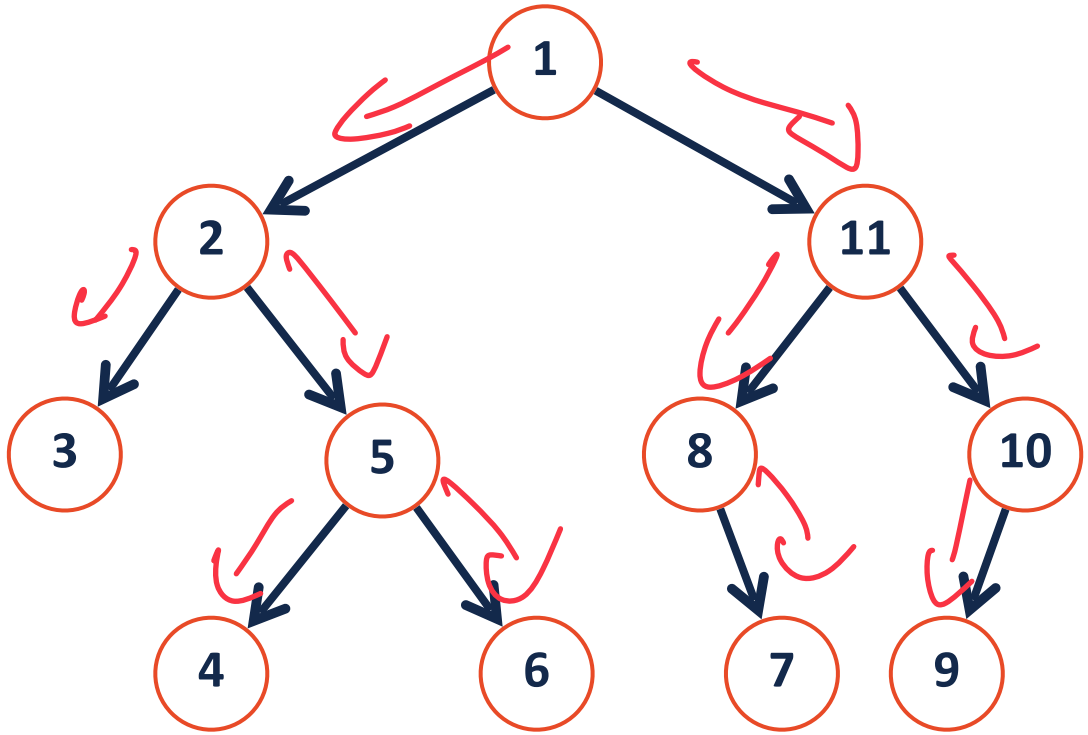
Explore as far along one path as possible before backtracking

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

LIFO



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

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

Preorder!

Breadth First Search

Size of queue ≈ width of tree

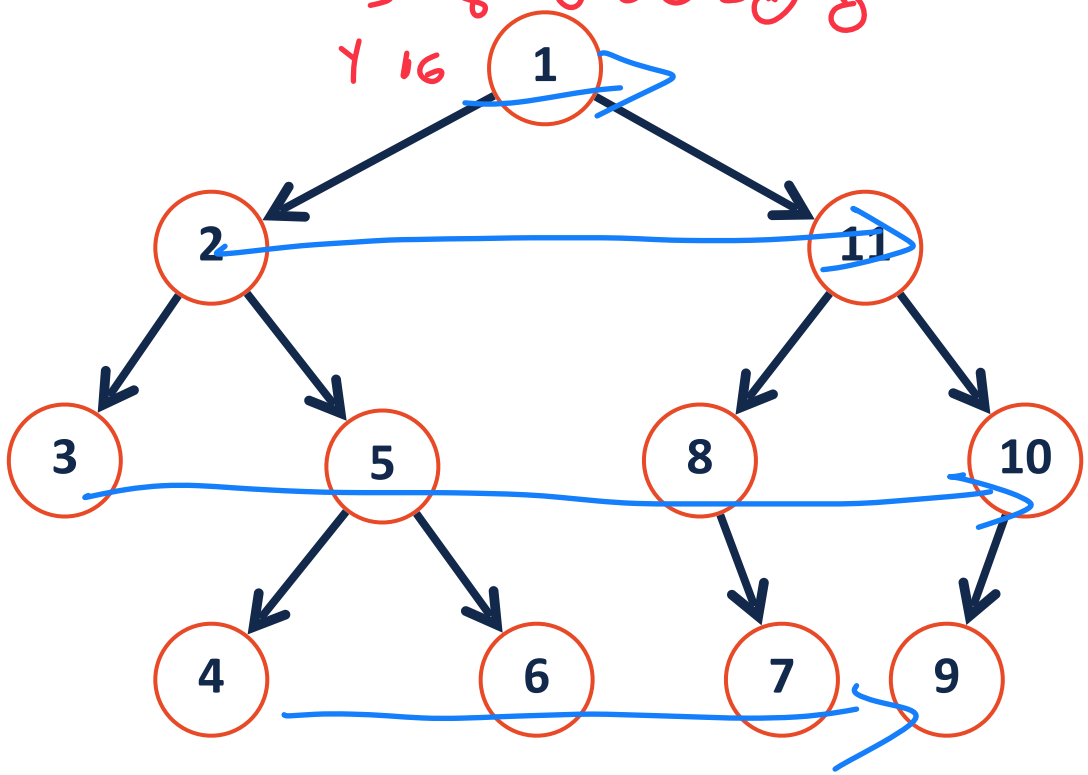
Fully explore depth i before exploring depth i+1

Make a queue initialized with root

While queue isn't empty:

- Dequeue front element (as `tmp`)
- Print `tmp`
- Enqueue `tmp->left`
- Enqueue `tmp->right`

FIFO



equal to width

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

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

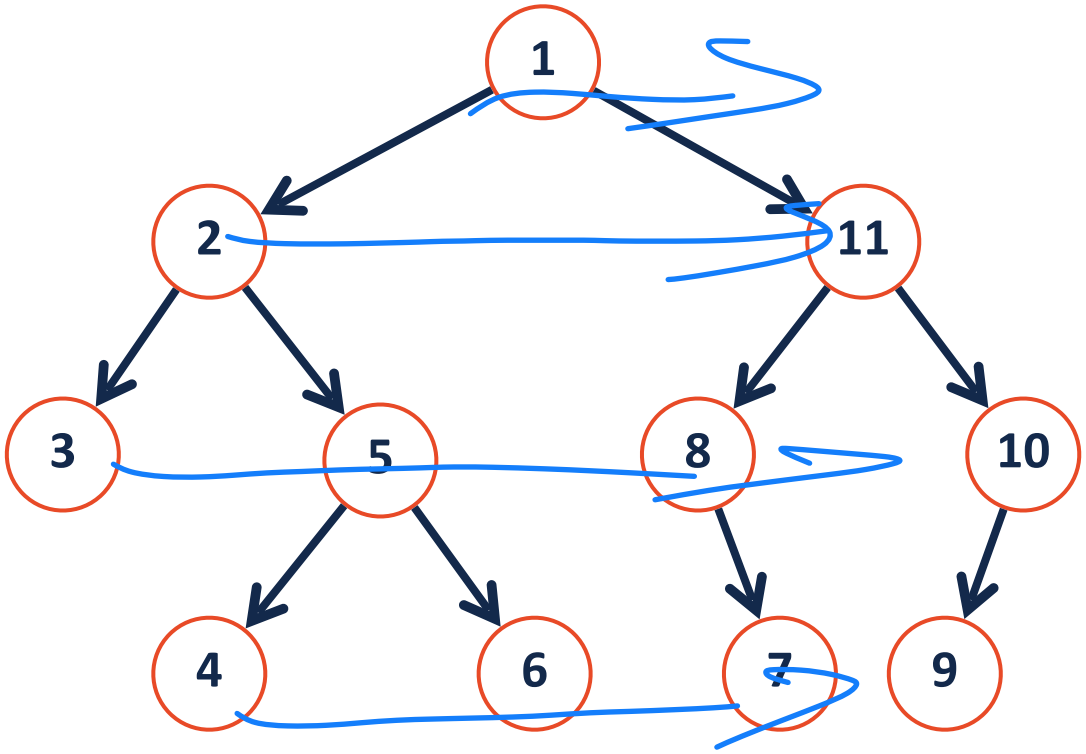
Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue initialized with root

While queue isn't empty:

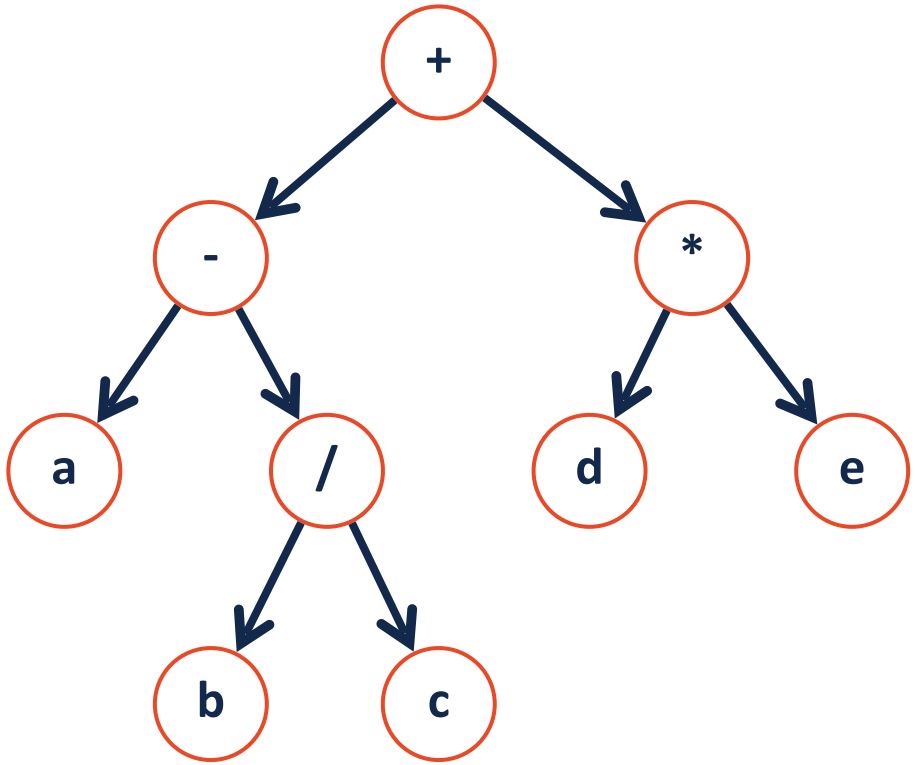
- Dequeue front element (as `tmp`)
- Print `tmp`
- Enqueue `tmp->left`
- Enqueue `tmp->right`



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

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

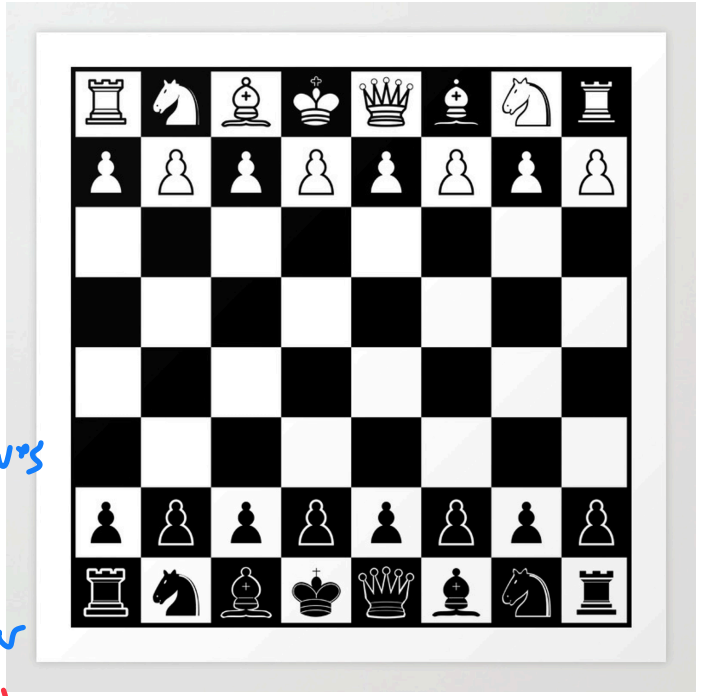
Level-Order Traversal




```
1  template<class T>
2  void BinaryTree<T>::lOrder(TreeNode * root)
3  {
4
5      Queue<TreeNode*> q;
6      q.enqueue(root);
7
8      while( q.empty() == False){
9
10         TreeNode* temp = q.head();
11         process(temp);
12
13         q.dequeue();
14
15         q.enqueue(temp->left);
16         q.enqueue(temp->right);
17
18     }
19 }
```

What search algorithm is best?

The average 'branch factor' for a game of chess is ~31. If you were searching a decision tree for chess, which search algorithm would you use?



DFS

- Pick one path and follow until eof game
- (+) Looking ahead is better to know if move was good
- (+) Better space complexity
- (−) If I'm on a bad path, I lose!

Length of chess game
30 — 70 moves

BFS

- Look at all possible moves
- (+) Will find "best" move?
  - If we let run forever
- (−) Branch factor is bad / 31
  - Queue size is width 31^2 31^3 ...

solves both problems!

Iterative Deepening DFS

1) Do DFS to depth limit (X)
2) If soln not found increase X

Tree Search → O(n) b/c traversal O(n)

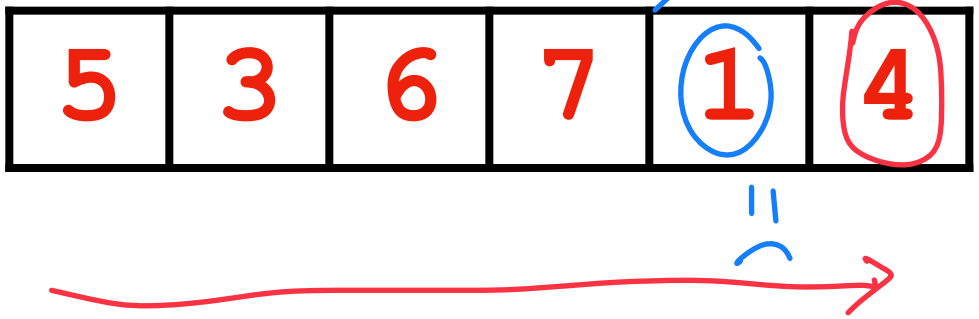
How can we improve our ability to search a binary tree?

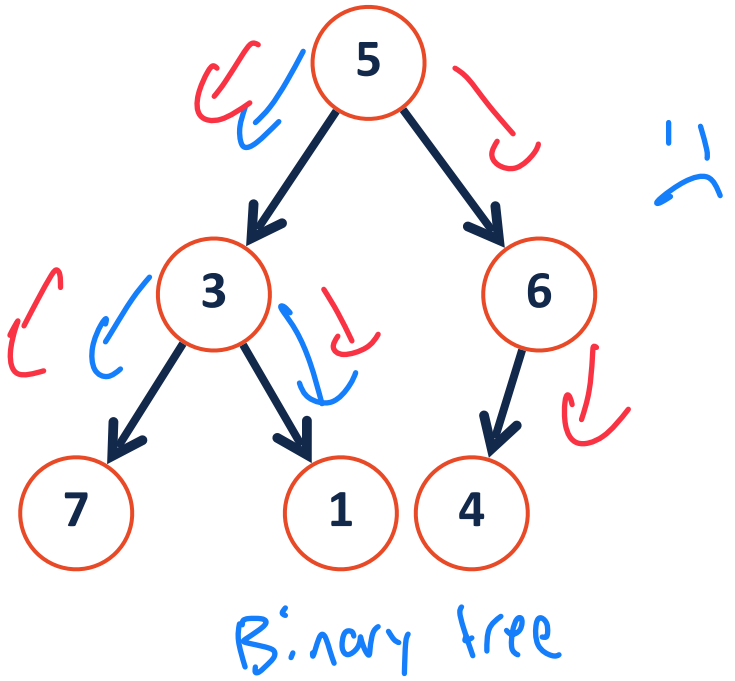
↳ can we add an order to our tree?

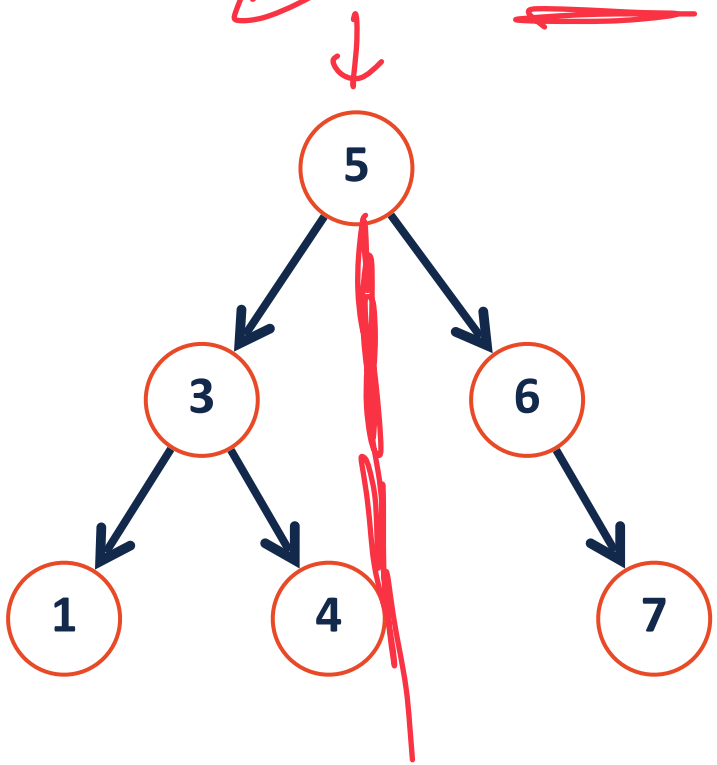
What do we trade in order to do so? tradeoff!

↳ enforcing order changes insert / remove / find.

Improved search on a binary tree







Dictionary ADT

Modify tree to be a dictionary

Data is often organized into key/value pairs:

Key → value

Word → Definition

Course Number → Lecture/Lab Schedule

Node → Incident Edges

Flight Number → Arrival Information

URL → HTML Page

…

Binary Search Tree (BST)

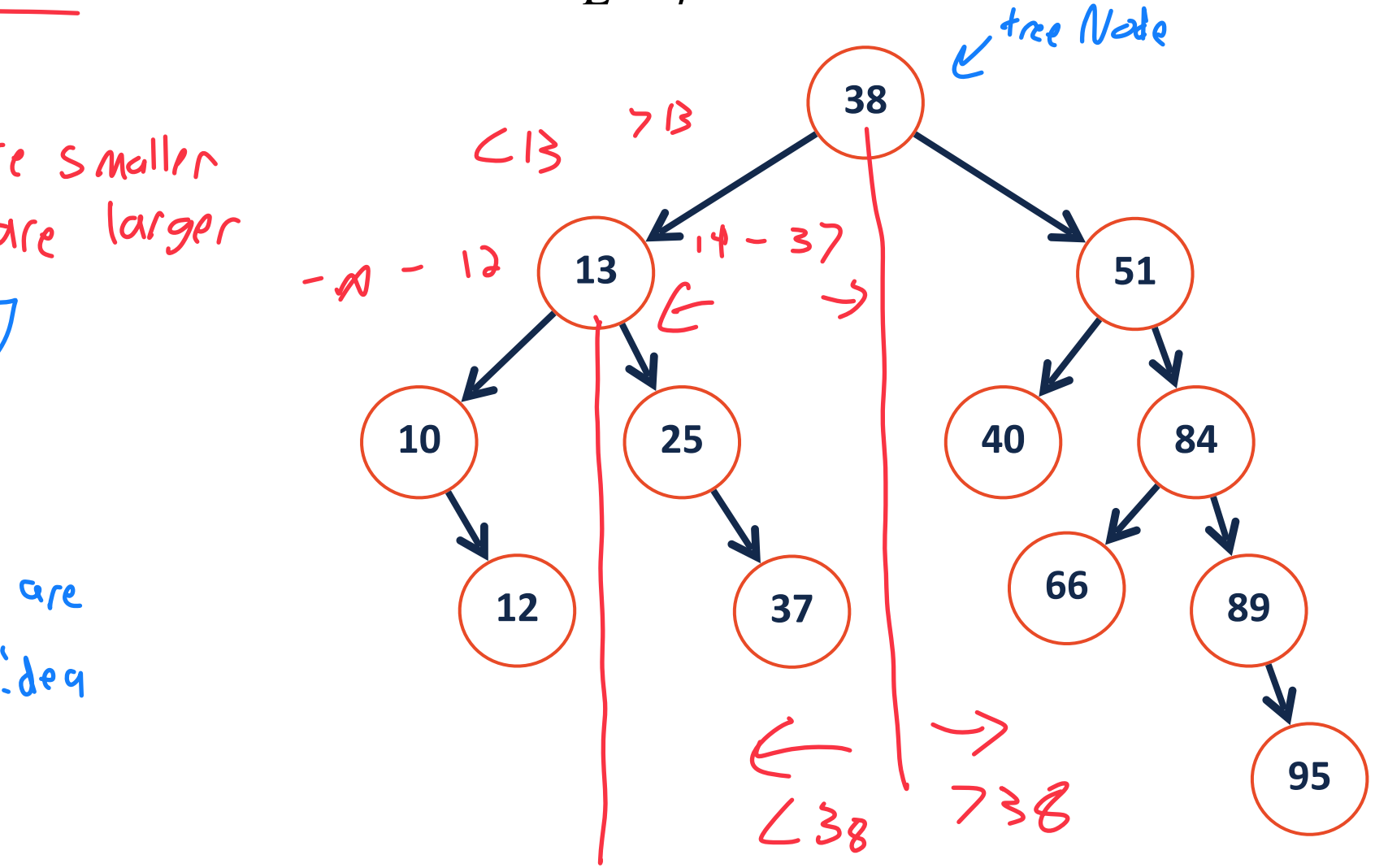
A **BST** is a binary tree $T = TreeNode(val, T_L, T_r)$ such that:

For each node
all nodes to left are smaller
all nodes to right are larger

Assume no duplicates!

{38, 38}
"38_1" "38_2"

These sorts of ideas are distractions from main idea



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
```

functions enforce BST property

Dictionary

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

Our ADT doesn't change but how we implement does

* will change

Others won't

BST Find

root → nullptr

root → 66

find(66)

Start @ root

Recursive Problem!

Base case:

↳ If tree empty, return root

↳ If root is query, return root

Recursive step:

Compare root→key w/ query

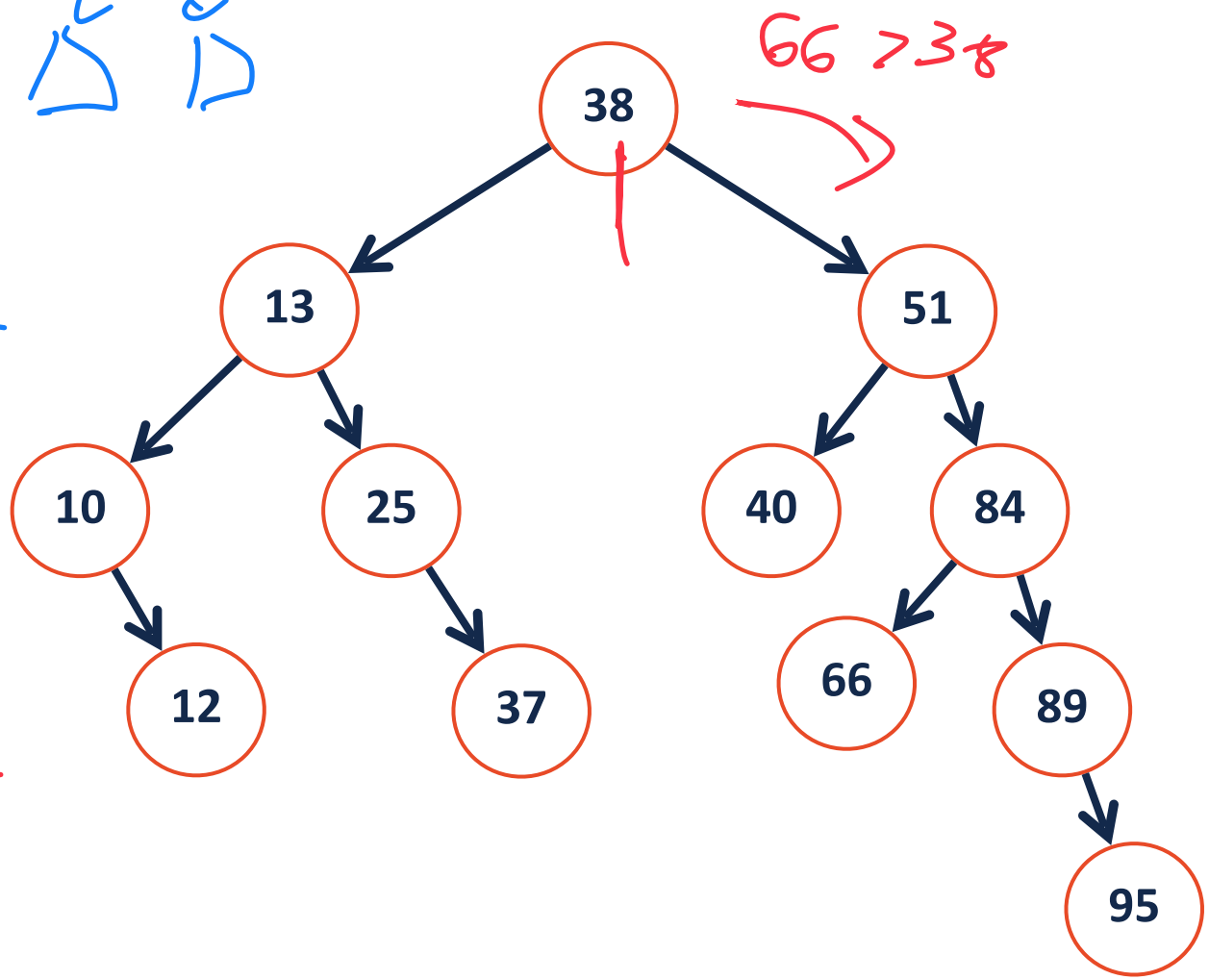
tmp

if

tmp > query, recurse right

tmp < query, recurse left

==



BST Find

A recursive function based around value of root:

Base Case: If root is `null`, return root

Let `tmp = root->key()`

`tmp == query`, return root

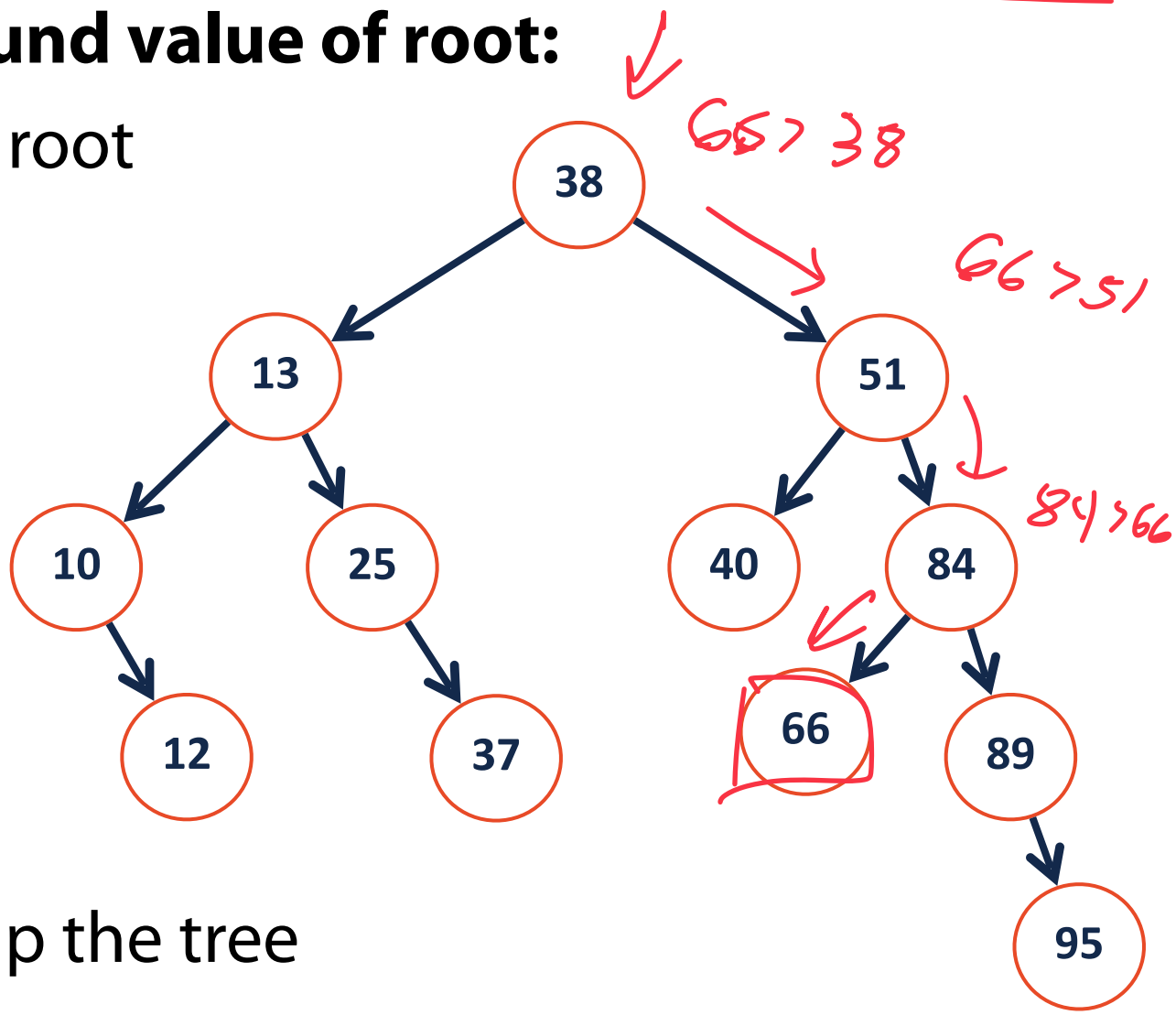
Recursion:

`tmp < query`, recurse right

`tmp > query`, recurse left

Combining:

Return the recursive value back up the tree



BST Find

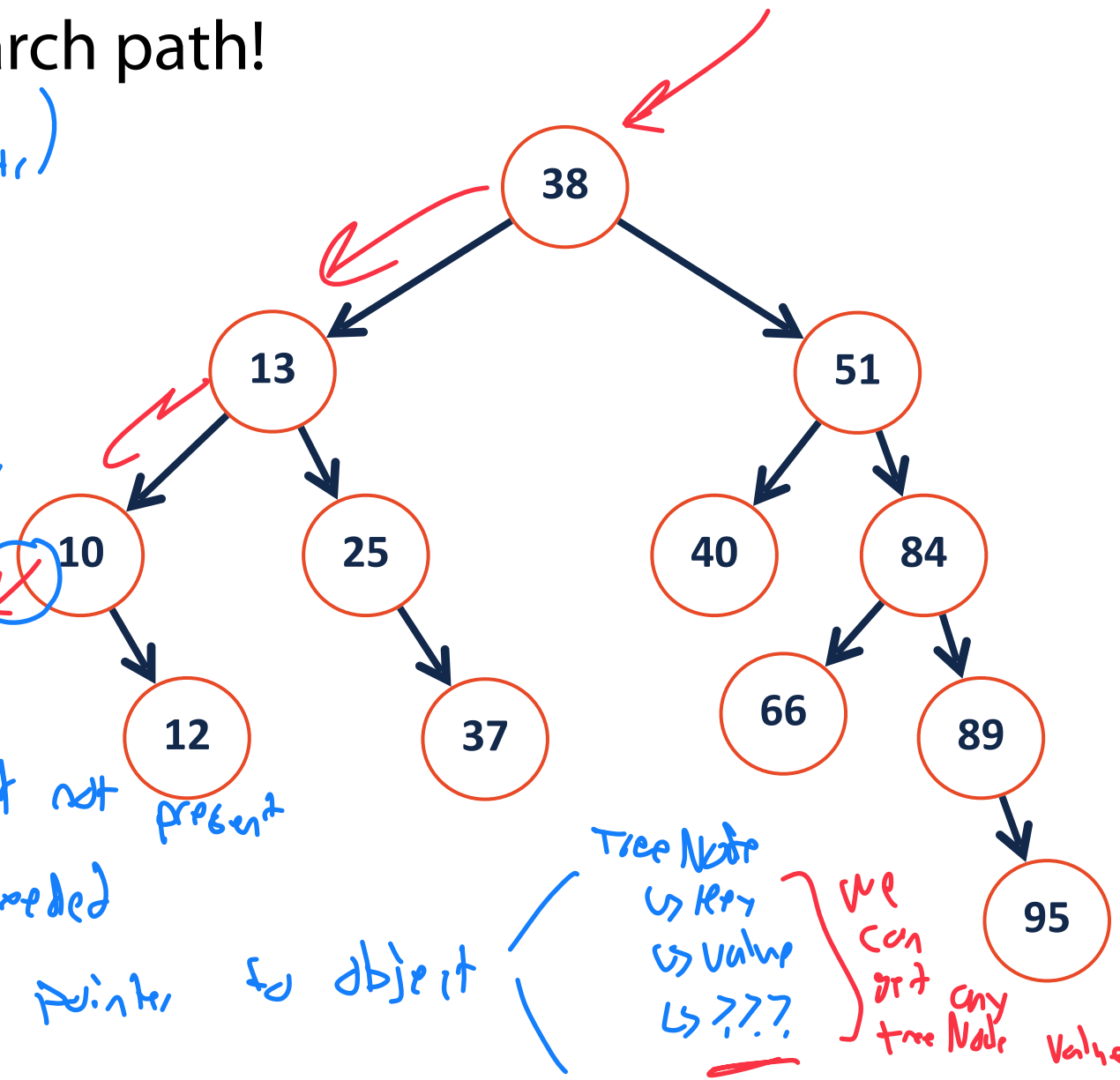
find(9)

Make sure you can follow the search path!

returning 10 → left (has value nullptr)

(List Node *&)
Tree Node *&

↳ If we return this by ref to pointer

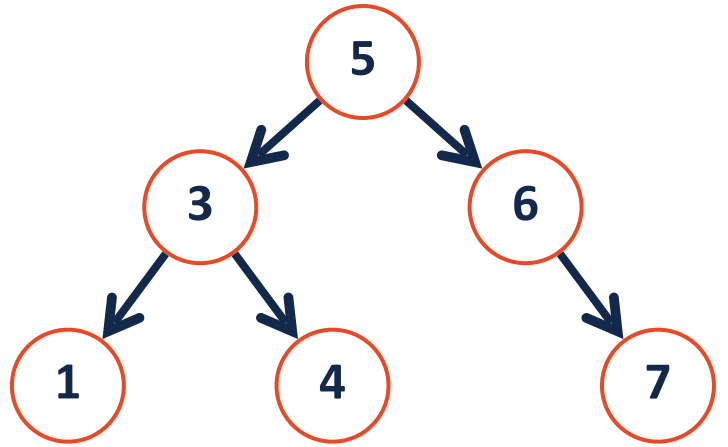


- We get value nullptr if object not present
- We can modify pointer if needed
- If object exists we have pointer to object

Tree Node
↳ key
↳ value
↳ ???

we can get any tree Node value

```
1  template<typename K, typename V>
2
3  ____TreeNode * &_____ _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->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
```

query

No const here

No const here

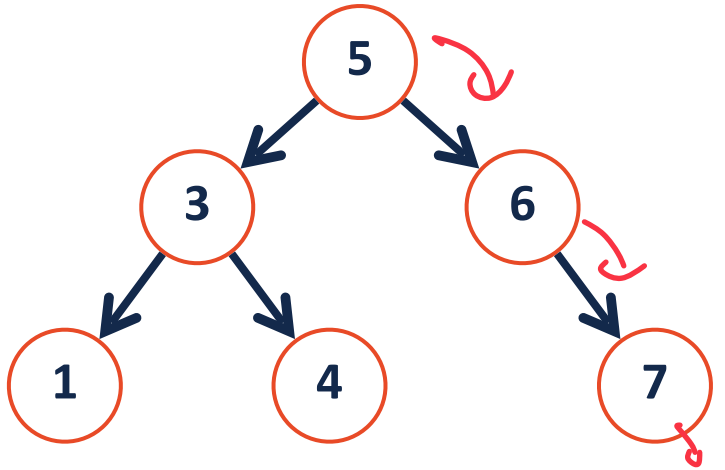
Find(8) returns pointer by ref (7->right)

Not nullptr!

Smaller go left

"else"

larger go right



BST Insert

`insert(33, v)`

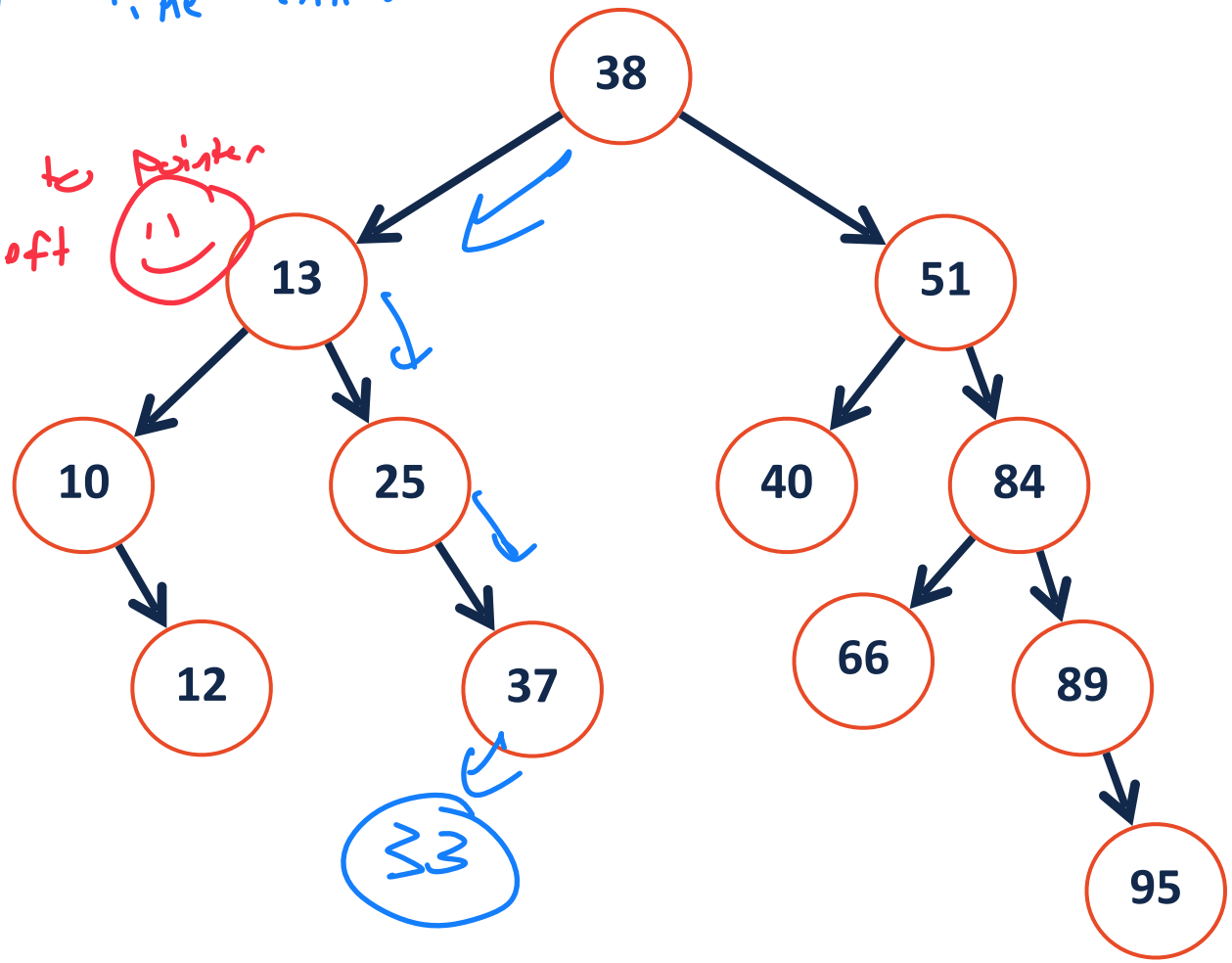
Need to find insert location

Need to do actual insert

A lot like linked list!

1) find(33) returns ref to pointer to 37 -> left

2) Insert 33 as 37 -> left

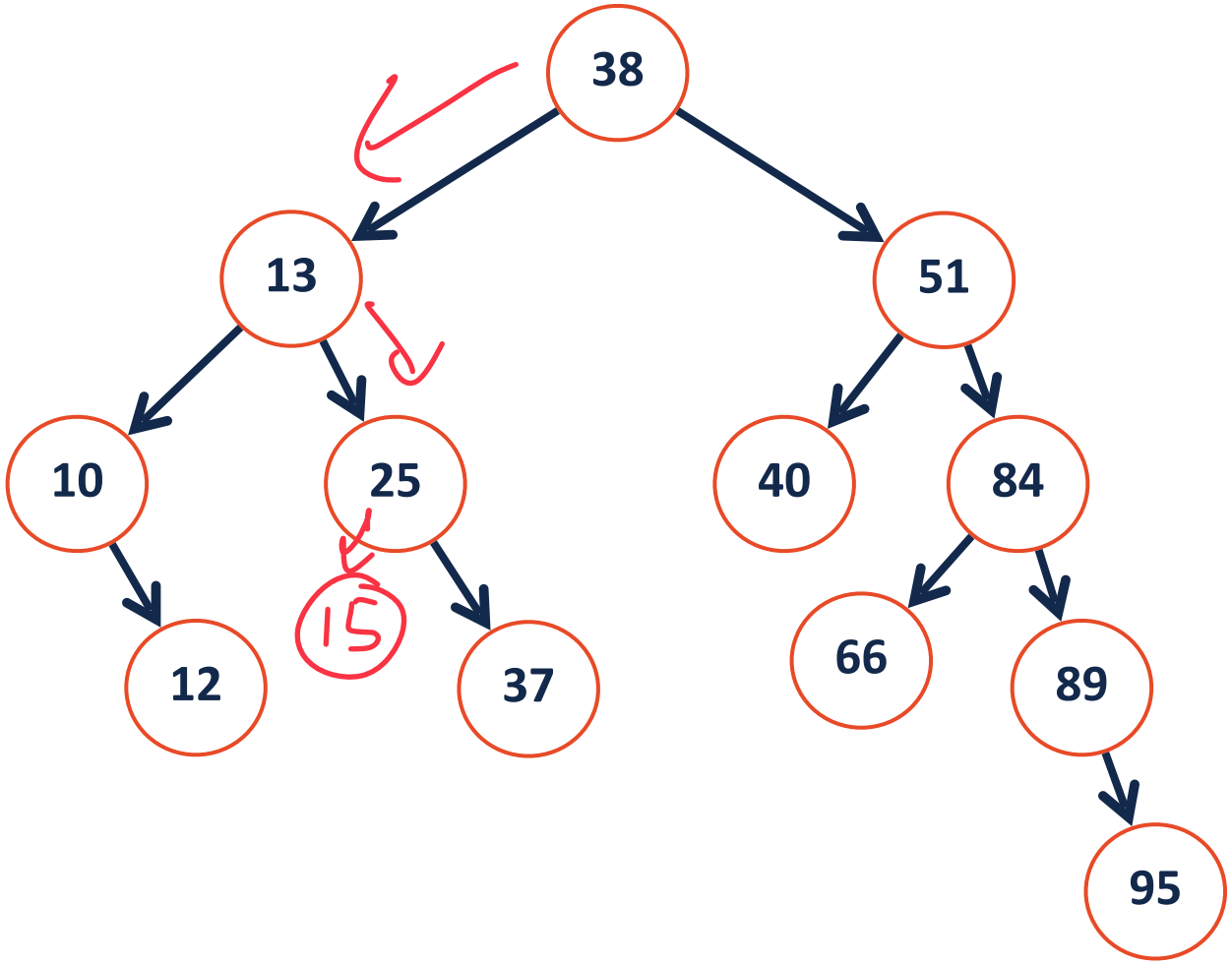


BST Insert

`insert(15, v)`

Find the insert location using `_find()`

Trivially insert at location

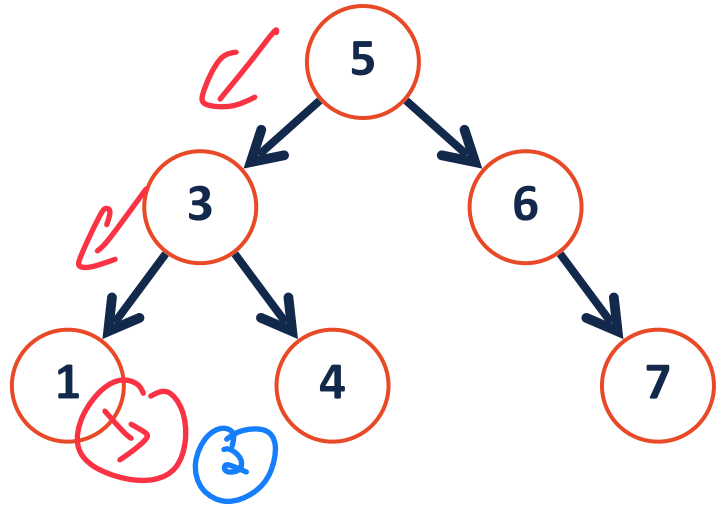


```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

recursive (in find)

```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5      TreeNode *t edge = _find(root, key);
6
7      edge = new TreeNode(key, value);
8
9
10
11
12
13
14     Note we did not need helper function!
15         Could have done this in top function
16 }
```

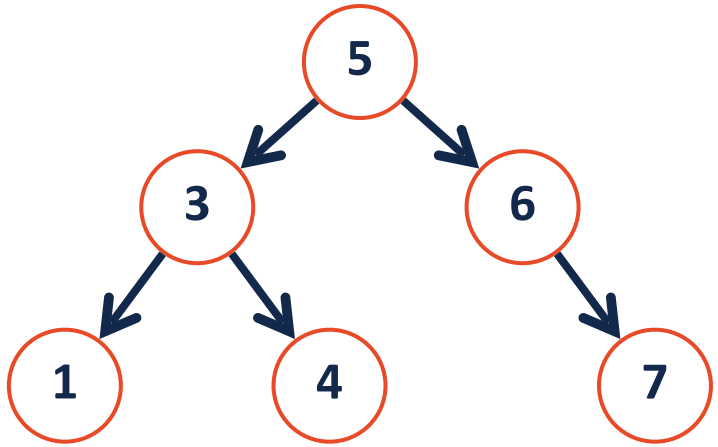
insert(2)



```
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
```

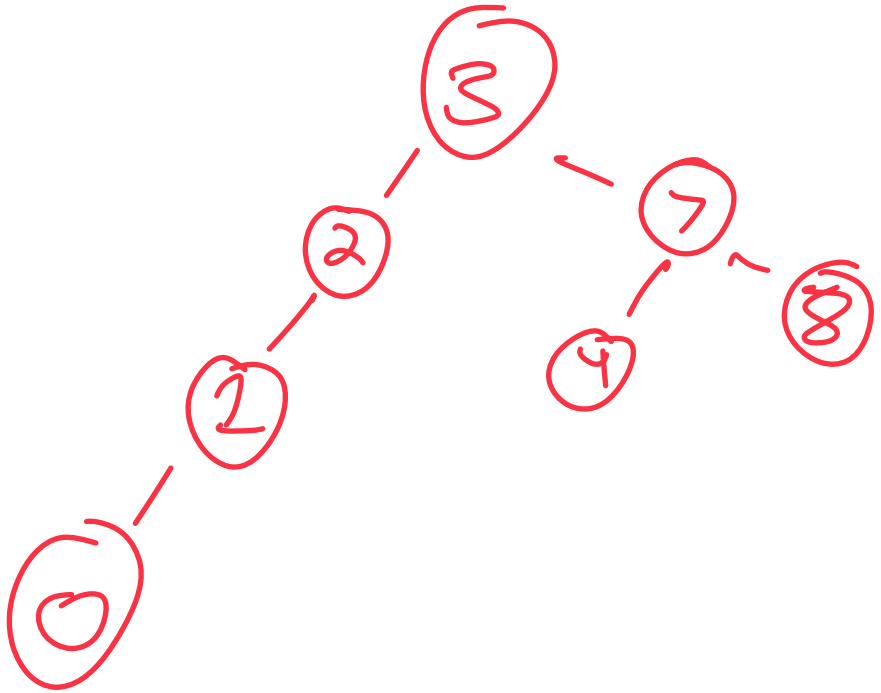
```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5  TreeNode *& tmp = _find(root, key);
6
7
8  tmp = new treeNode(key, val);
9
10
11
12
13 }
14
15
16
```

find is key!



BST Insert

What binary tree would be formed by inserting the following sequence of integers: `[3, 7, 2, 1, 4, 8, 0]`



randomly generate lists of numbers

See if tree you manually did matches computer generated

BST Remove

1) Find item being removed
2) Remove it! ← trickier than it seems

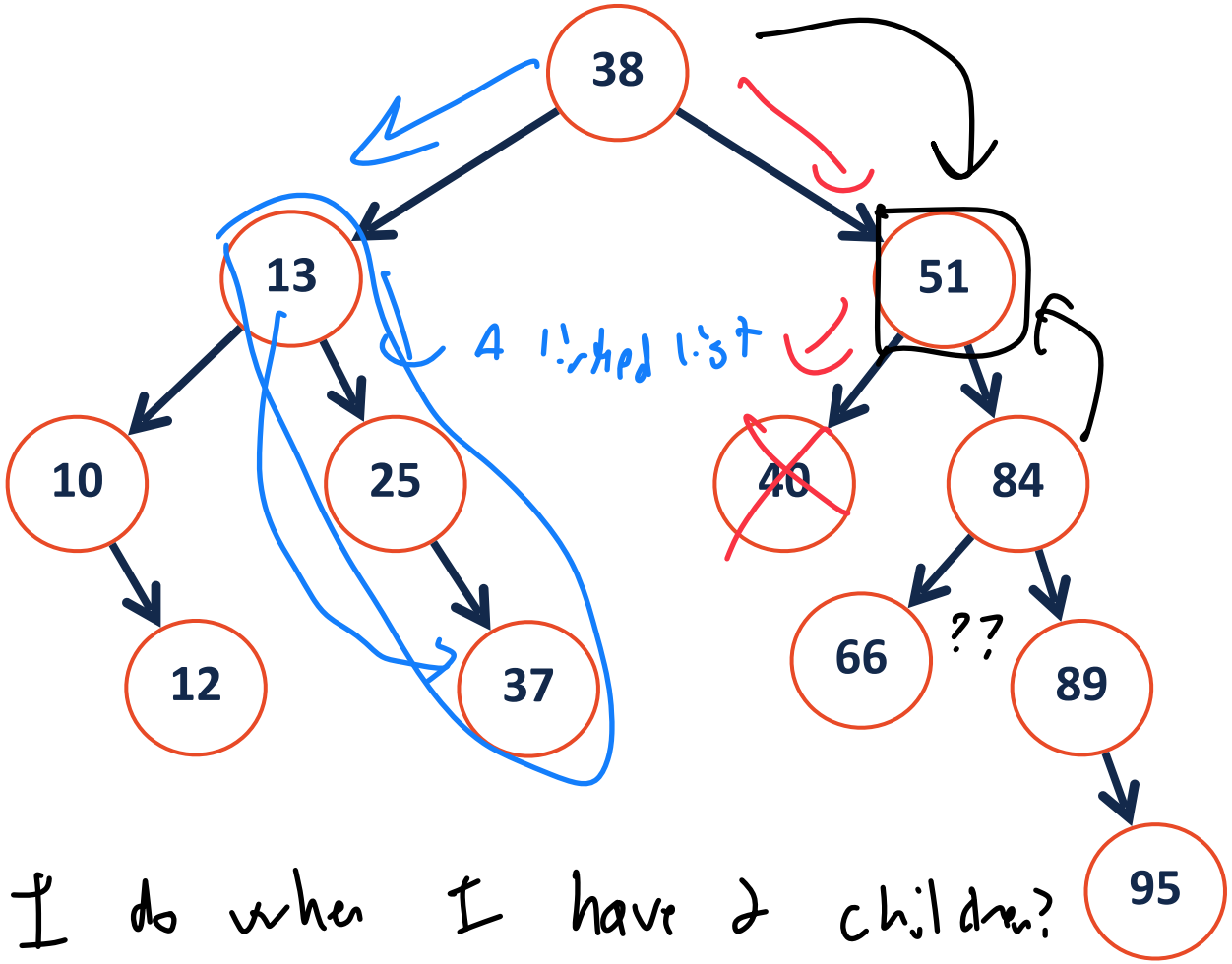
What should our tree look like after the following…

remove(40)

↳ easy!

remove(25)

↳ LL remove



remove(51) → what do I do when I have 2 children?