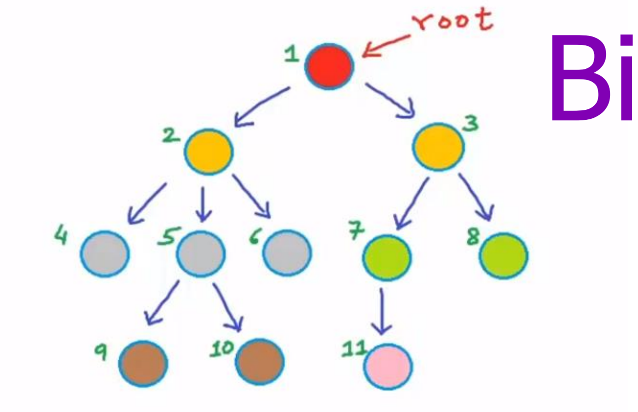
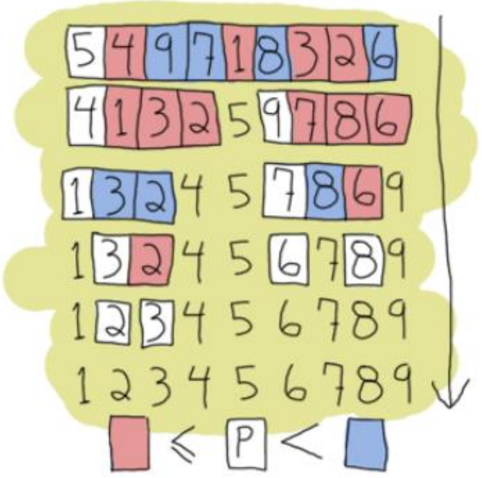
ECE 250 Data Structures & Algorithms



Binary Search Trees



Ziqiang Patrick Huang

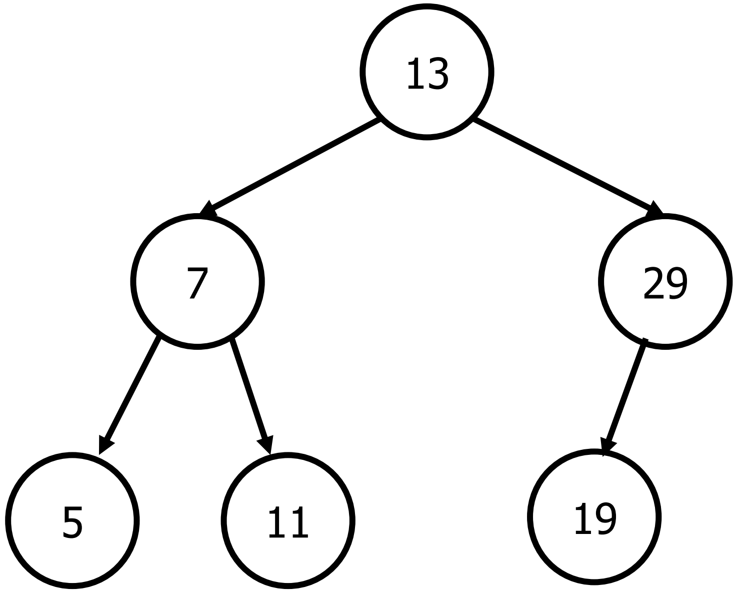
Electrical and Computer Engineering

University of Waterloo

WATERLOO | ENGINEERING

Last Time: Binary Search Tree

- Pointer based dynamic data structure (like Linked list)
  - But with up to two children (left + right) instead of one next
- Order Invariant
  - Everything to the left is smaller
  - Everything to the right is greater
- Offer O(log(n)) search
  - With a caveat …



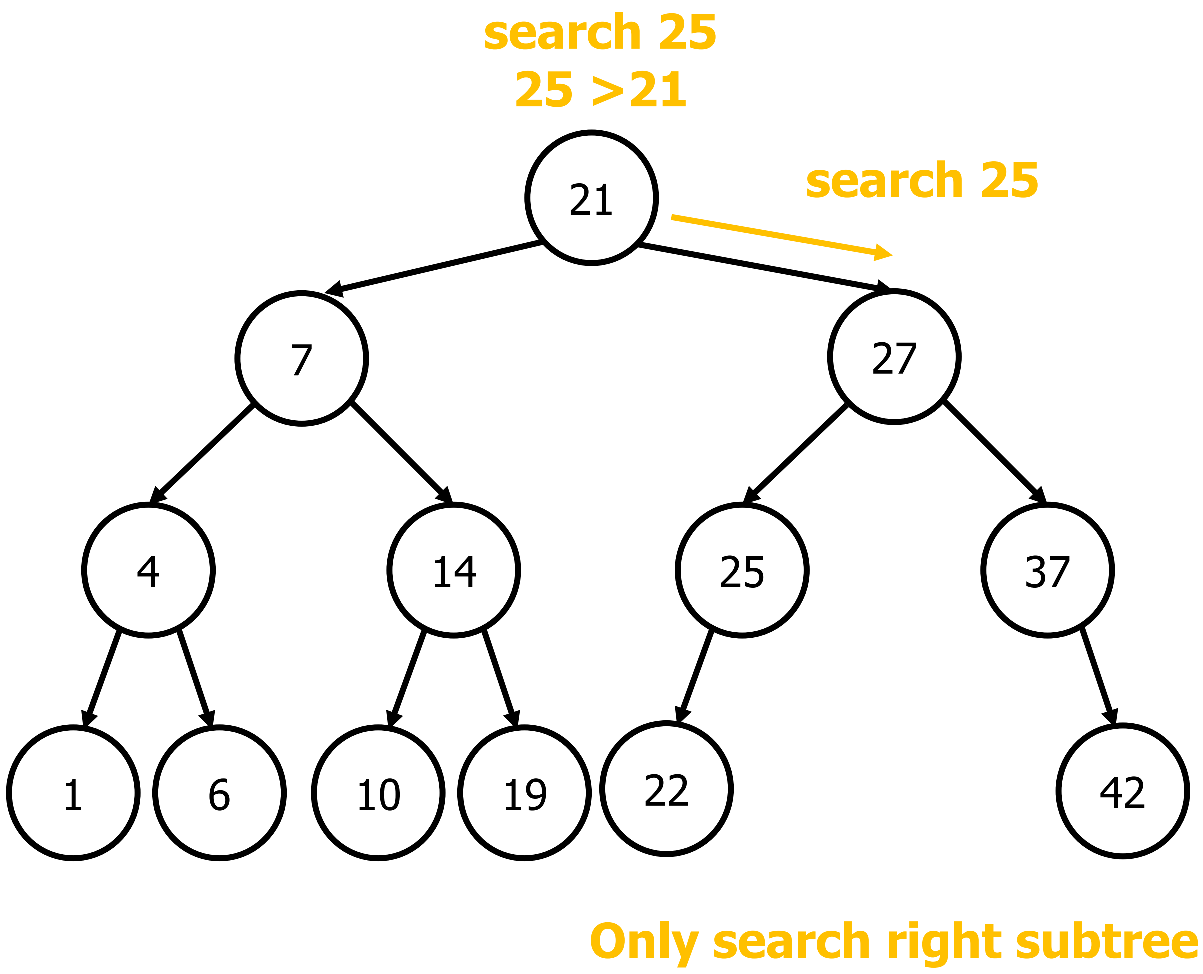
WATERLOO | ENGINEERING

BST Applications

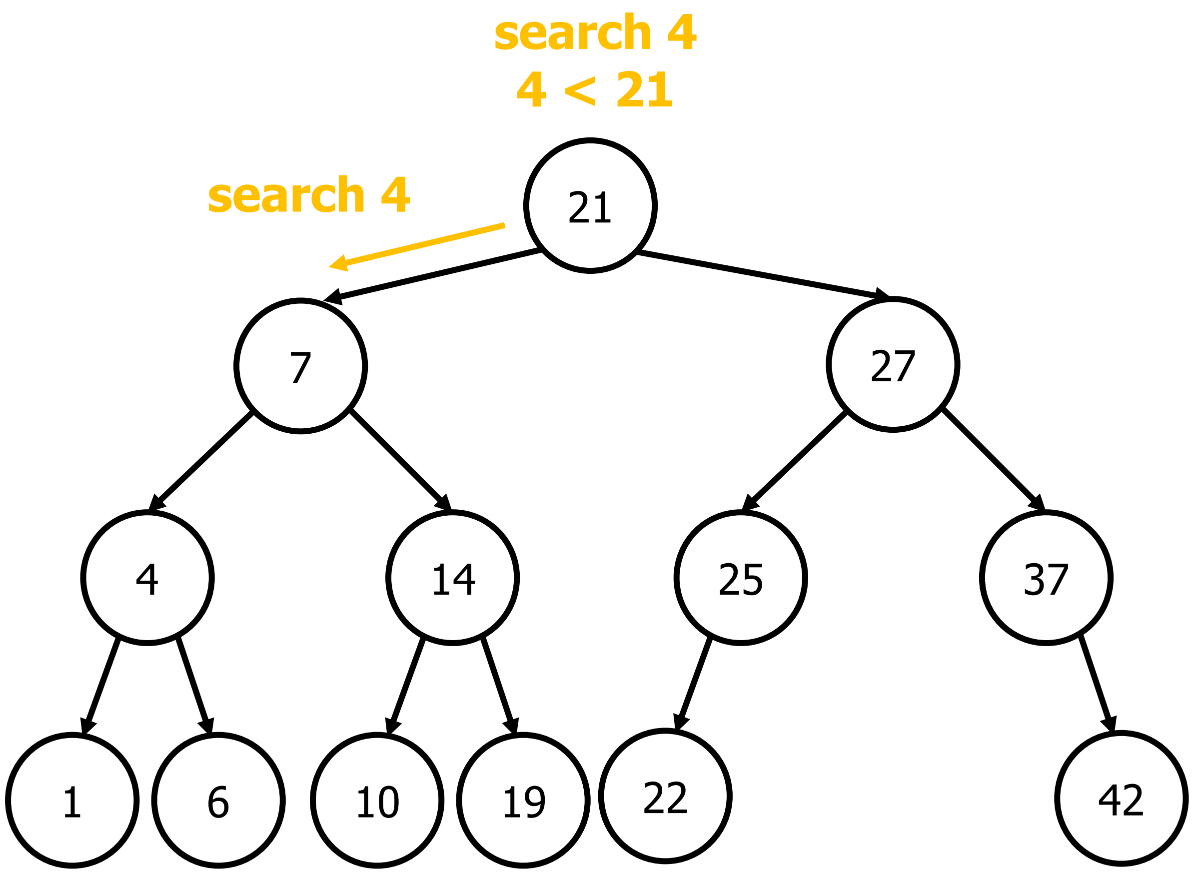
- Can be used to implement map and set ADTs
  - O(log(n)) addition, search, and removal
  - Requires keys to be a totally ordered type
    - i.e., can compare a and b and conclude either a < b, a = b, or a > b
    - Only restriction → can use BST when we cannot use others (e.g., hash tables)
  - Each BST nodes will hold the data for one entry:
    - Both key and the value for maps
    - Just the item for sets

- Other operations typically not part of a map or set
  - Find all keys within a given range (e.g., between 5000 and 30,000)
  - Find the smallest key greater than or equal to a particular value

Binary Search with BST

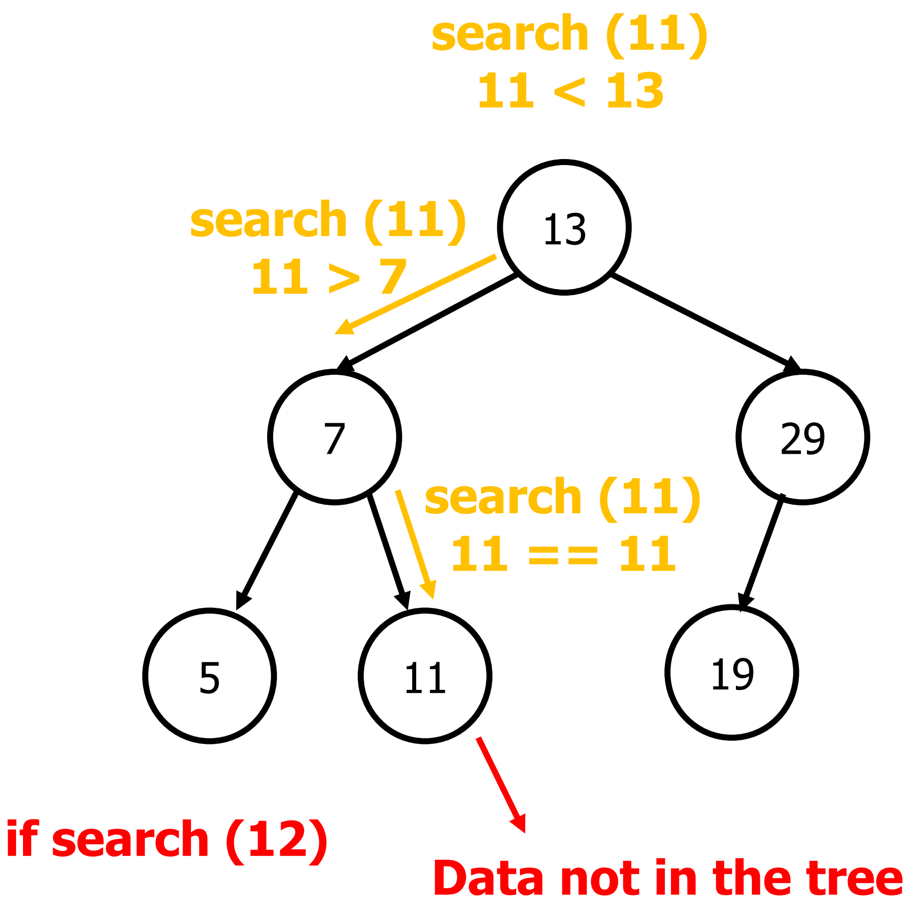


Binary Search with BST



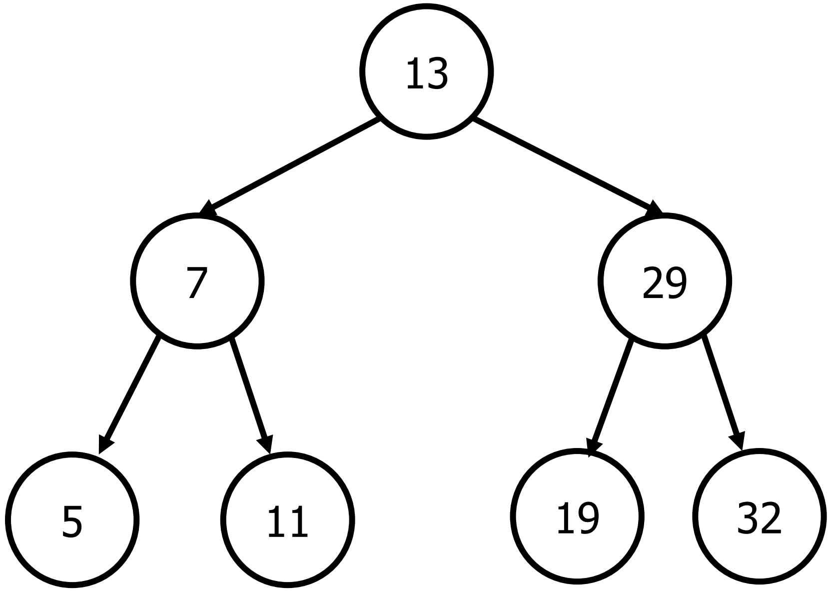
Only search left subtree

Binary Search with BST

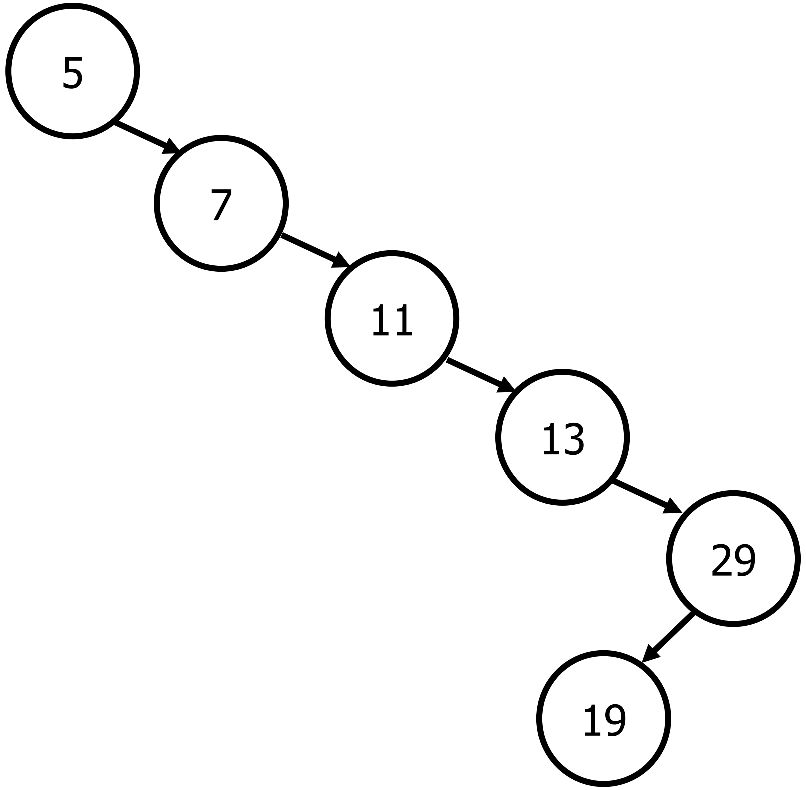


BST Search Caveat

Caveat: BST needs to remain balanced: for every node in the tree, the height of its children differ by at most 1.



Average: O(log(n))



Worst: O(n)

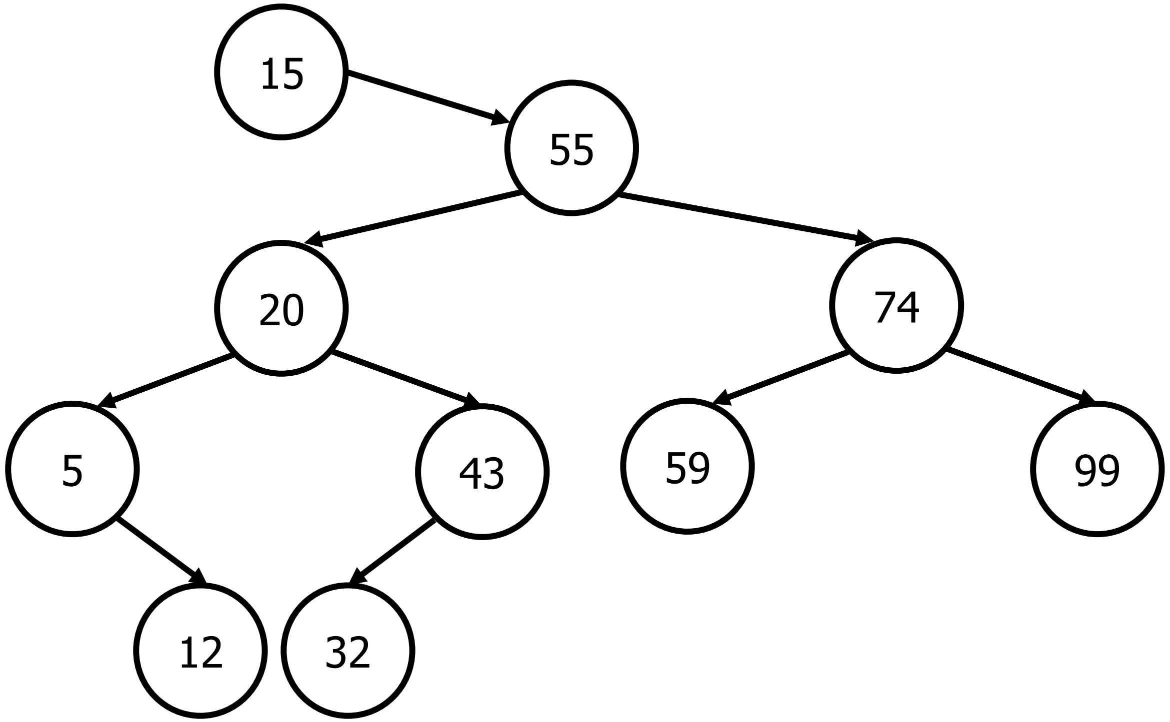
Adding to a Binary Search Tree

Invalid BST

Move out-of-place nodes?

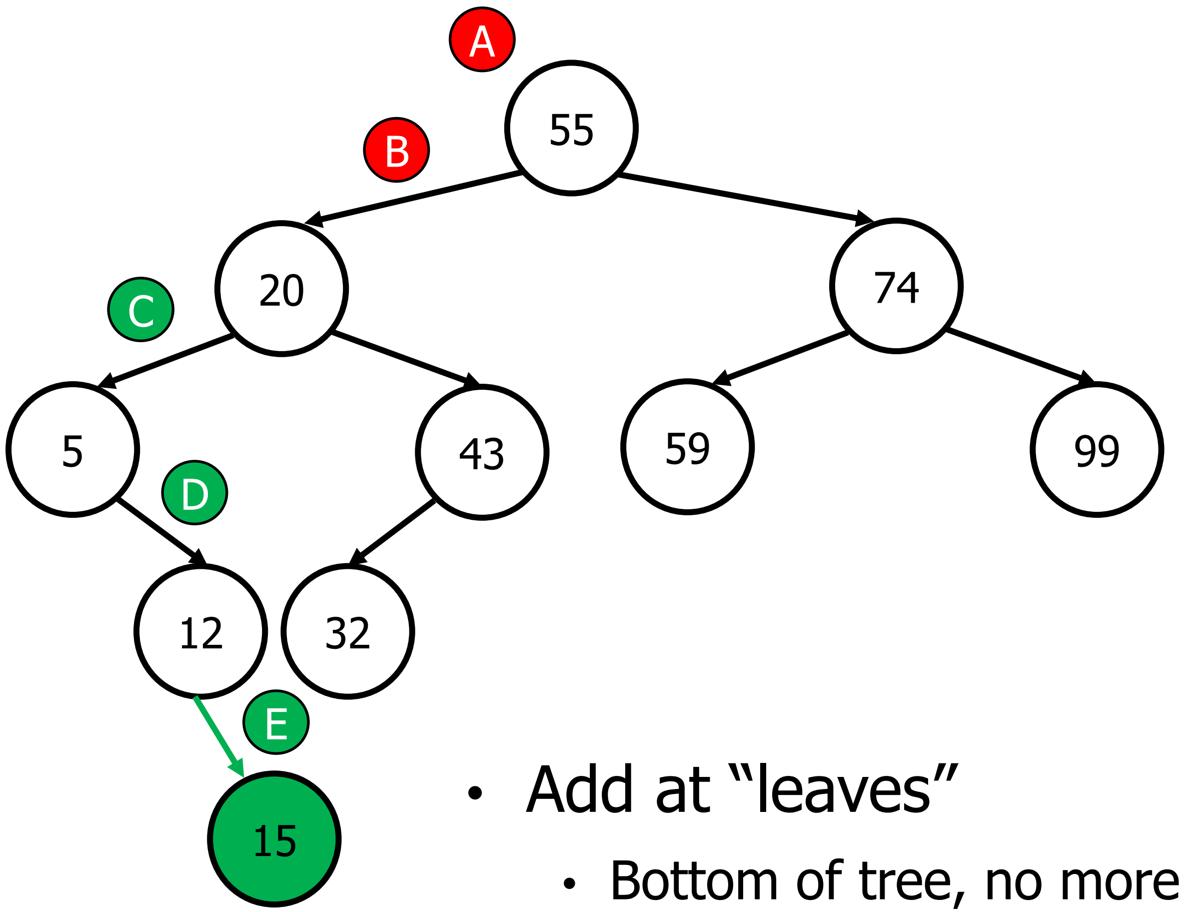
But to where?

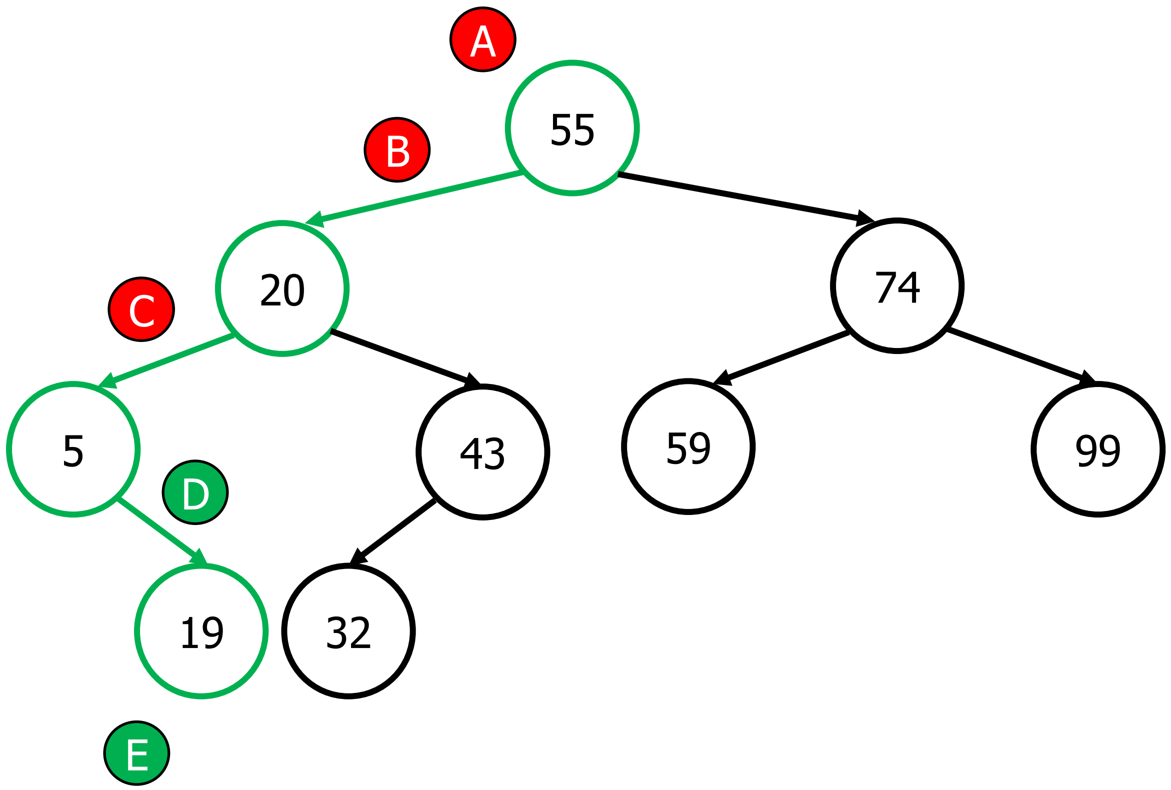
Back to the same problem!



- Where to add a node to a binary search tree?
  - Suppose we want to add 15
  - Aways add as root?

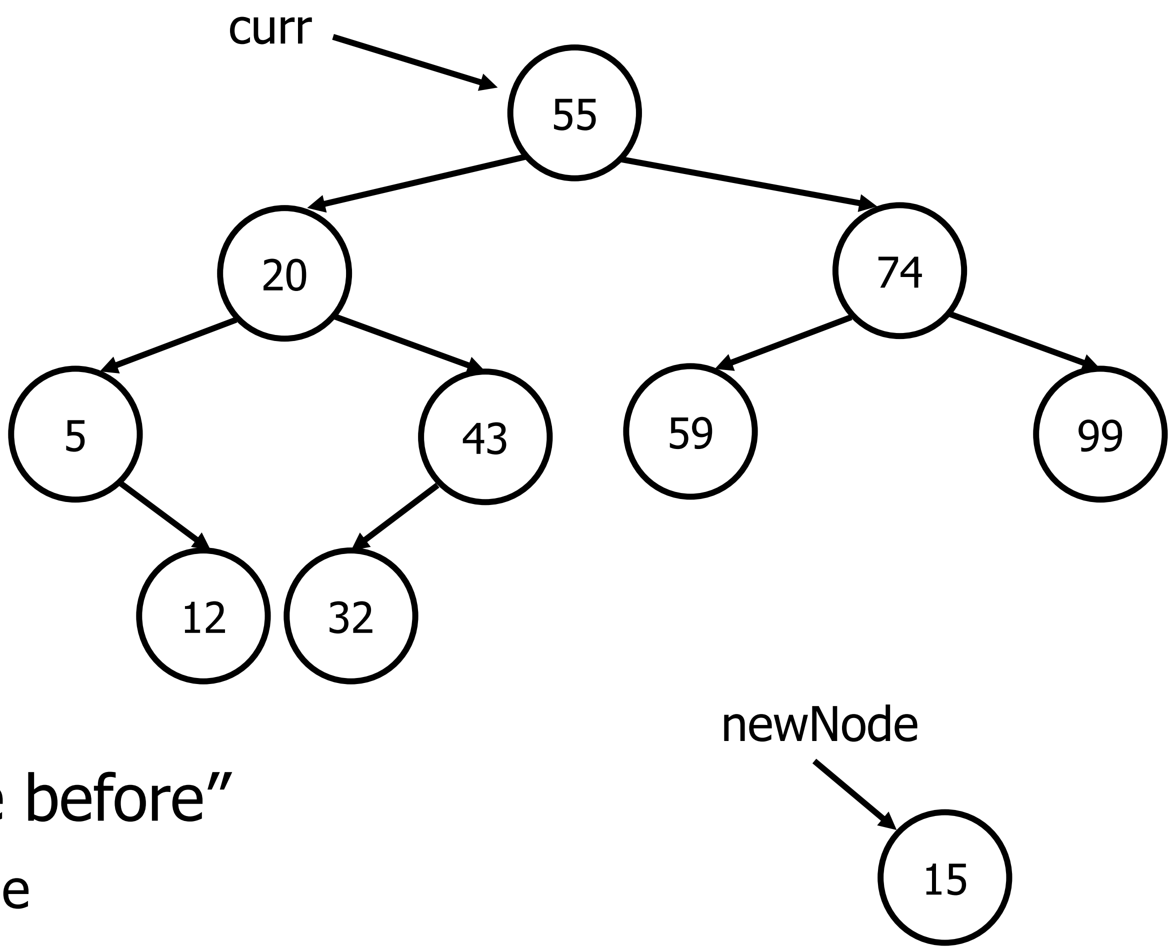
Adding to a Binary Search Tree





- Add at "leaves"
  - Bottom of tree, no more children
  - Find place by checking direction at each node

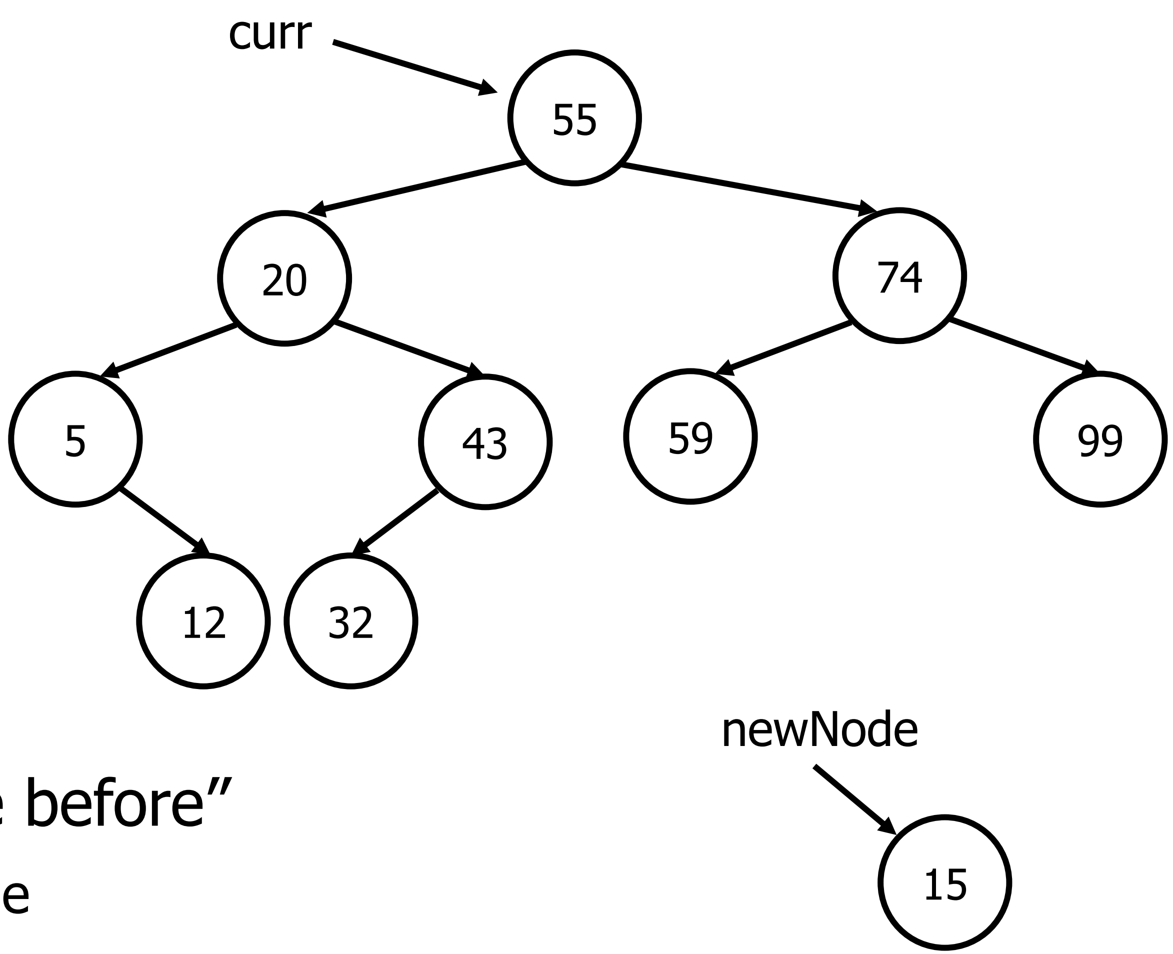
Find the Parent of the Node to Add



- Pointer to “node before”
  - Create new node

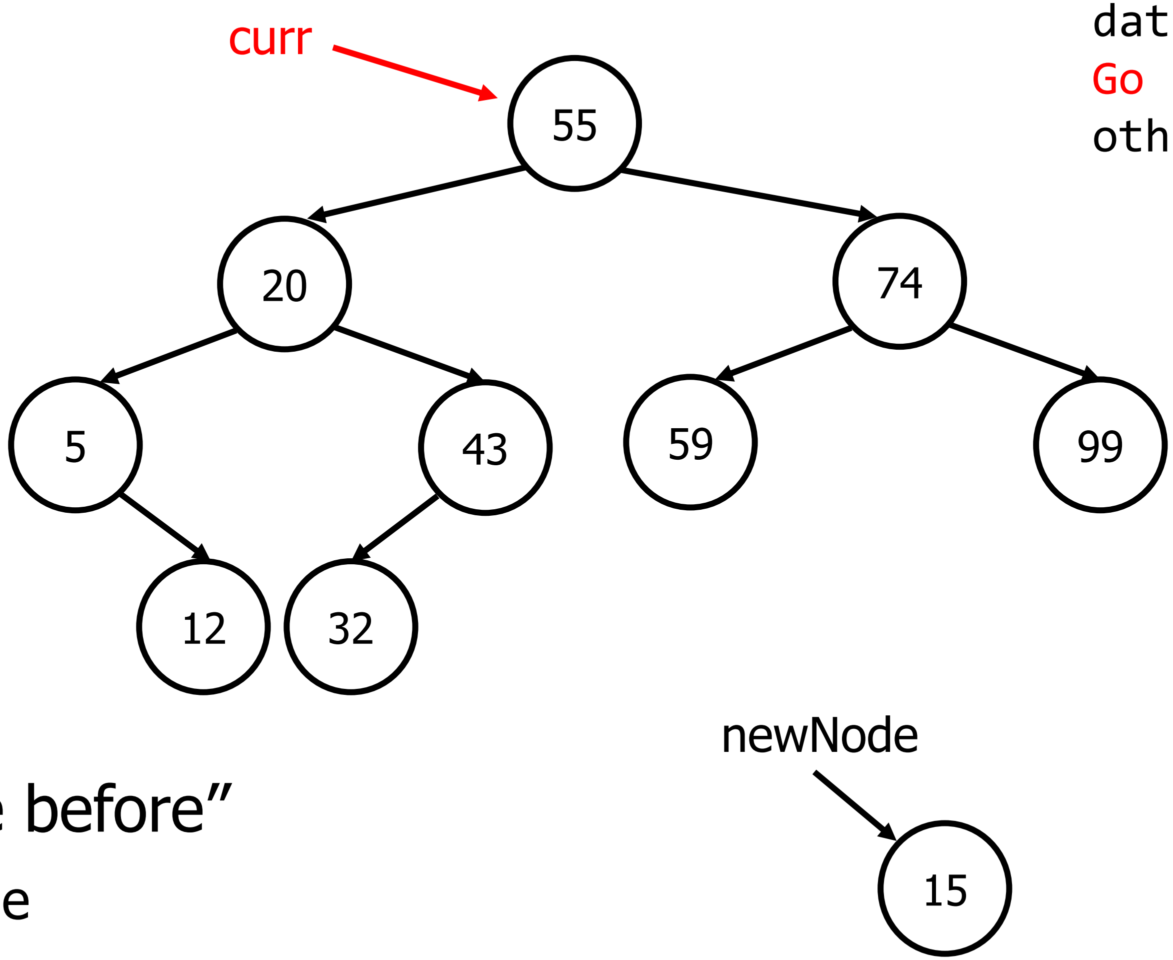
WATERLOO | ENGINEERING

Find the Parent of the Node to Add



- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)

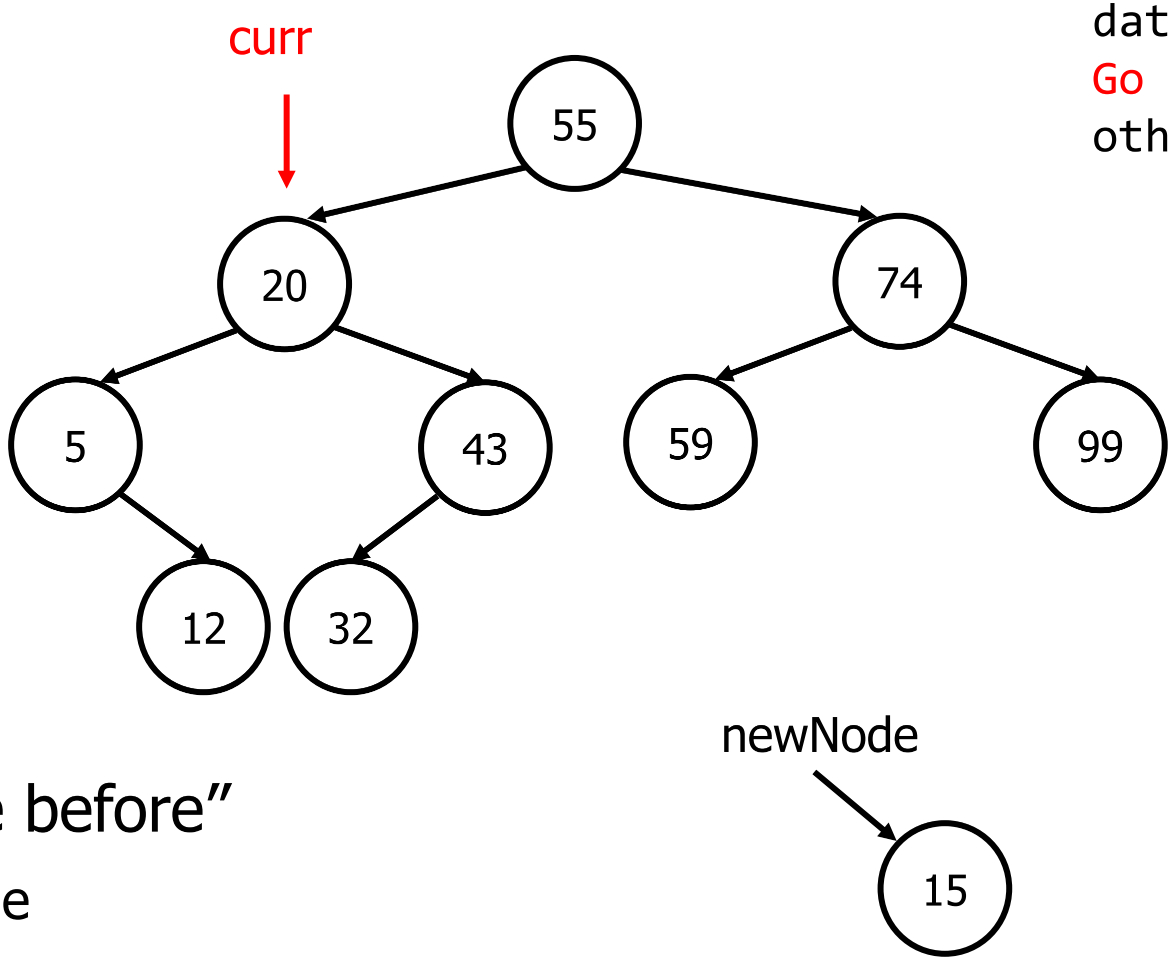
Find the Parent of the Node to Add




```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

- Pointer to "node before"
  - Create new node
  - Search for "node before" (Parent)

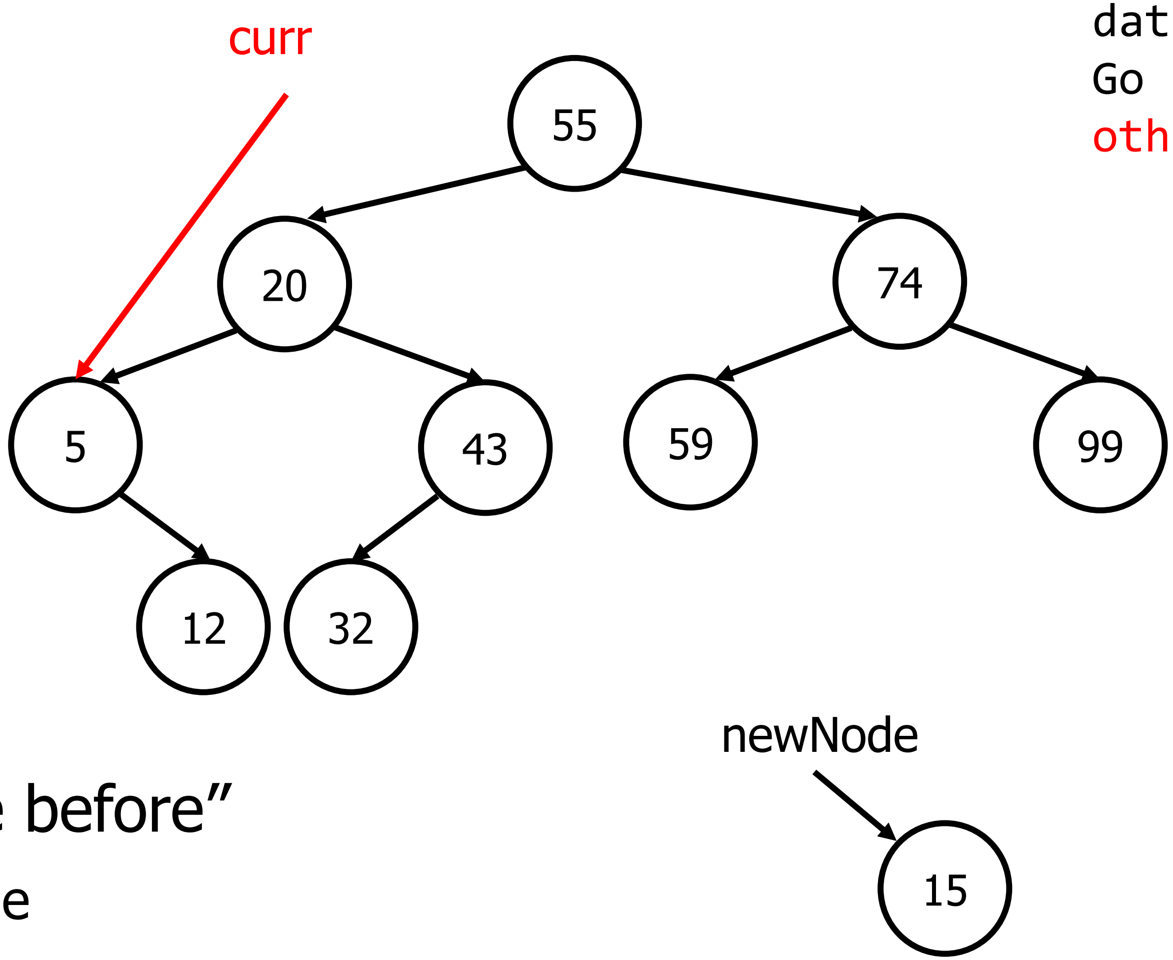
Find the Parent of the Node to Add




```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)

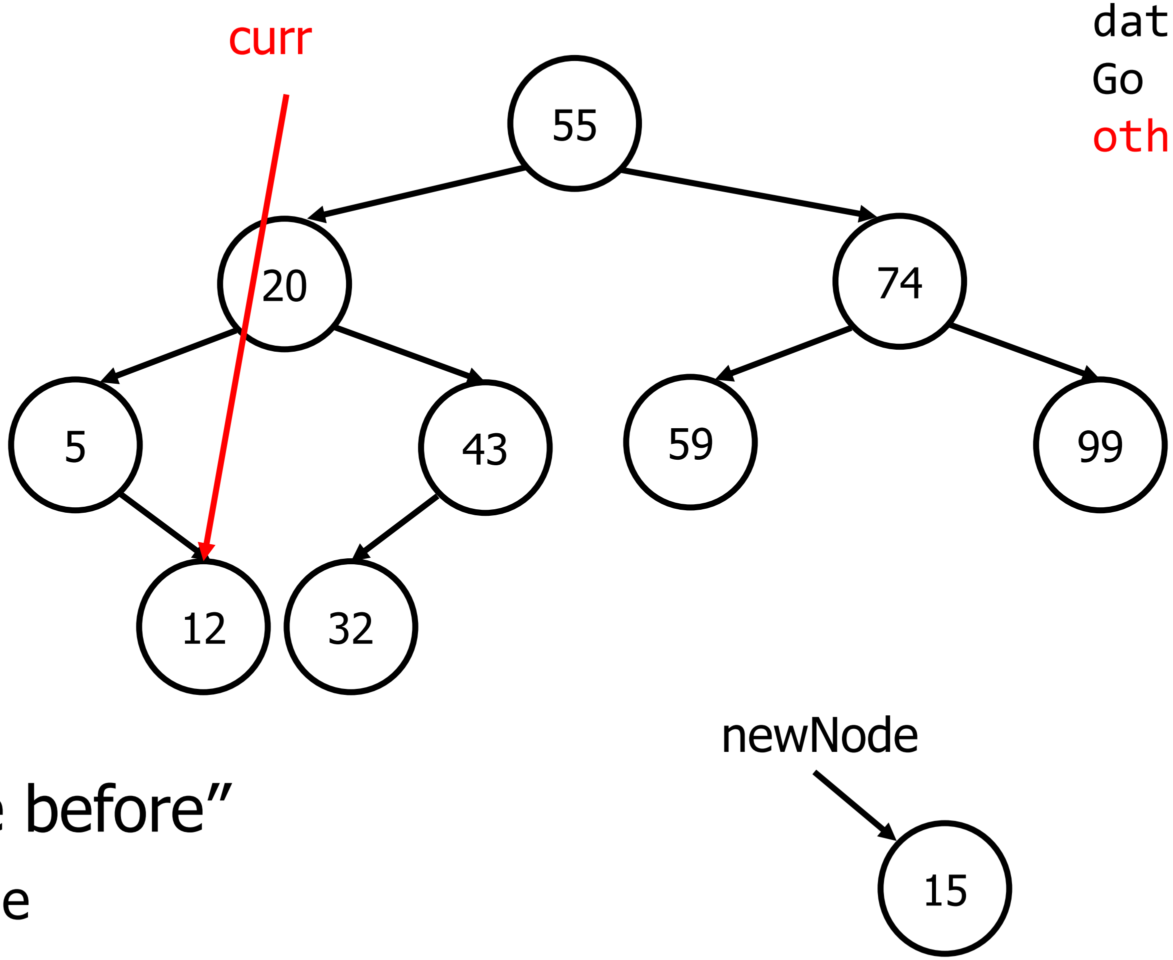
Find the Parent of the Node to Add




```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)

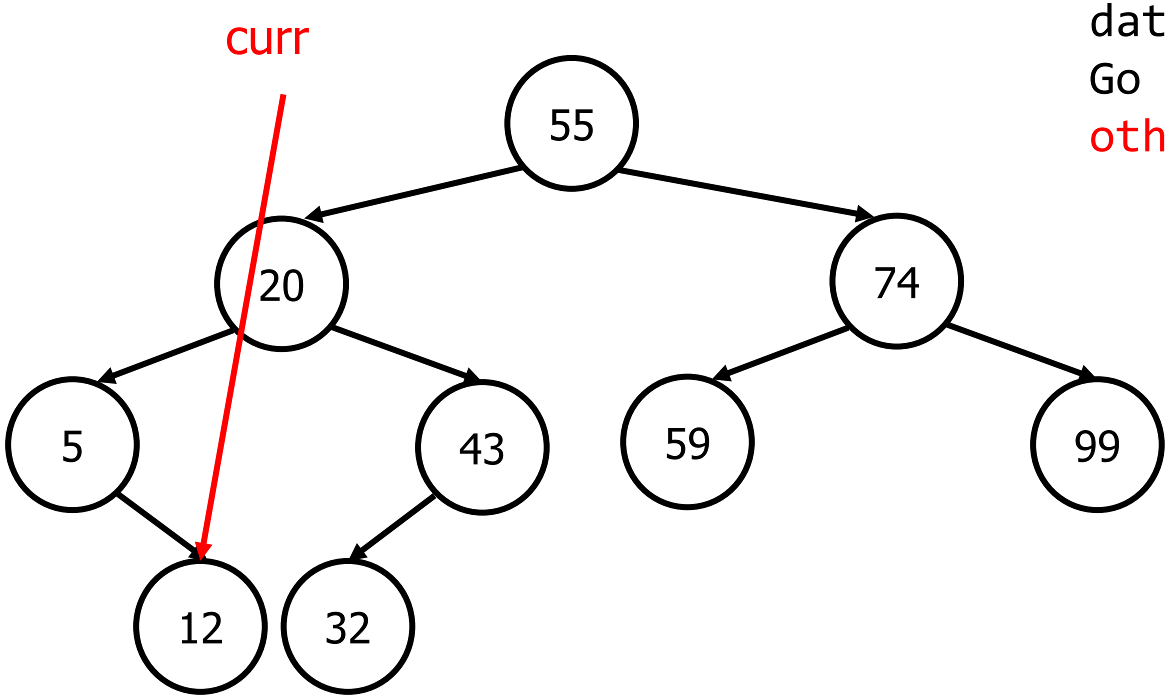
Find the Parent of the Node to Add




```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

- Pointer to "node before"
  - Create new node
  - Search for "node before" (Parent)

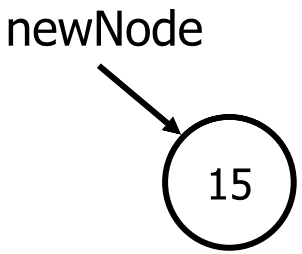
Find the Parent of the Node to Add




```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

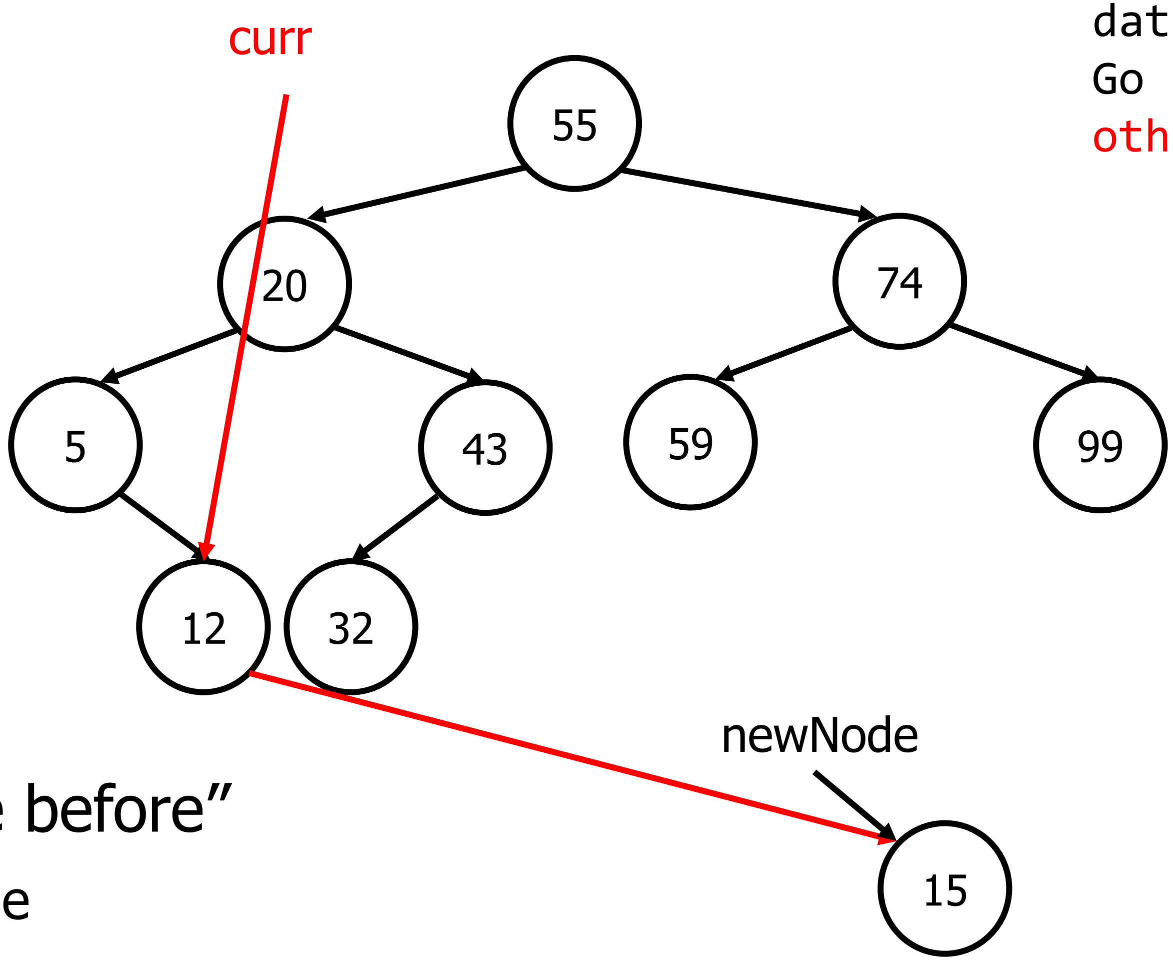
```
also need to check
curr->left/right
against NULL
```

```
Once found, set
curr->left/right to
the new node
```



- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)

Find the Parent of the Node to Add



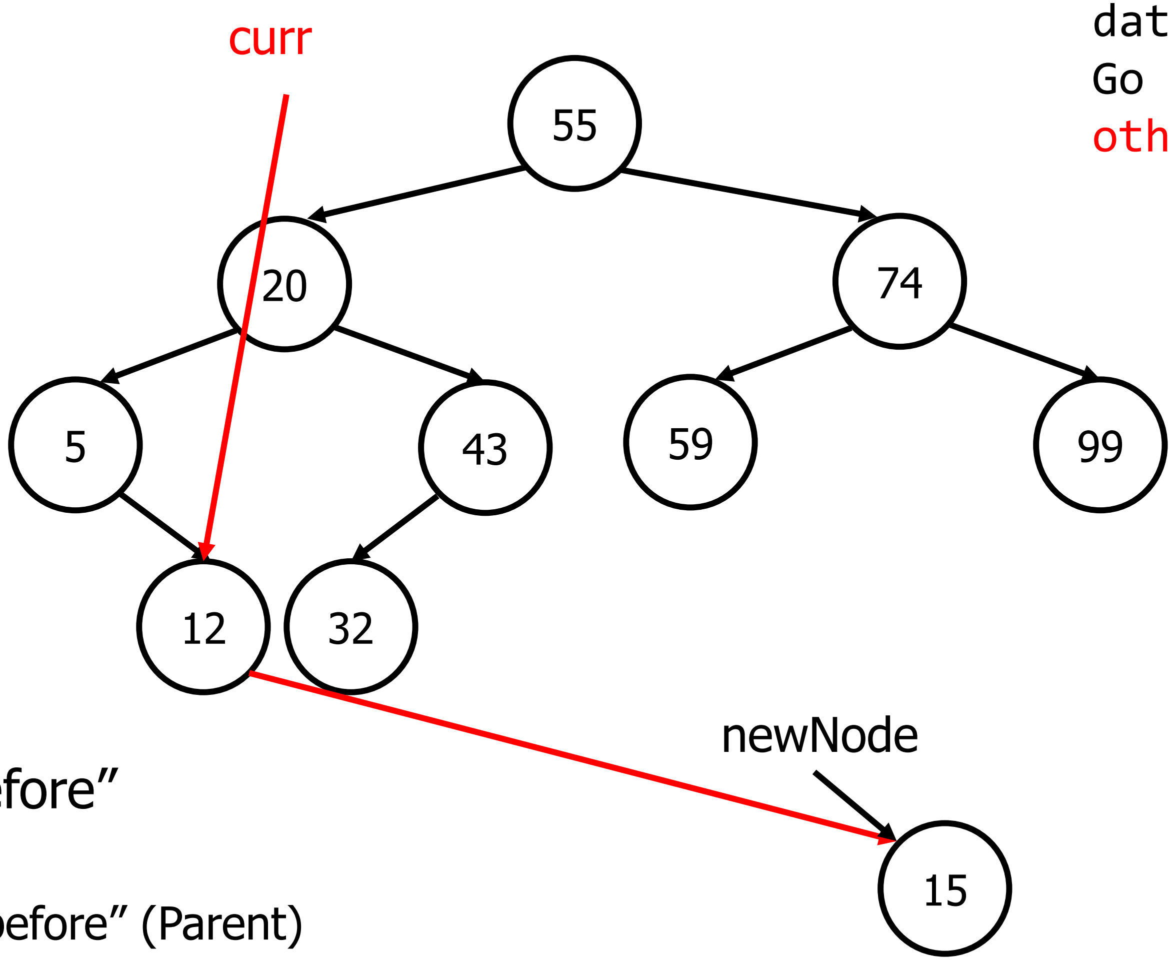

```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

```
also need to check
curr->left/right
against NULL
```

```
Once found, set
curr->left/right to
the new node
```

- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)

Find the Parent of the Node to Add



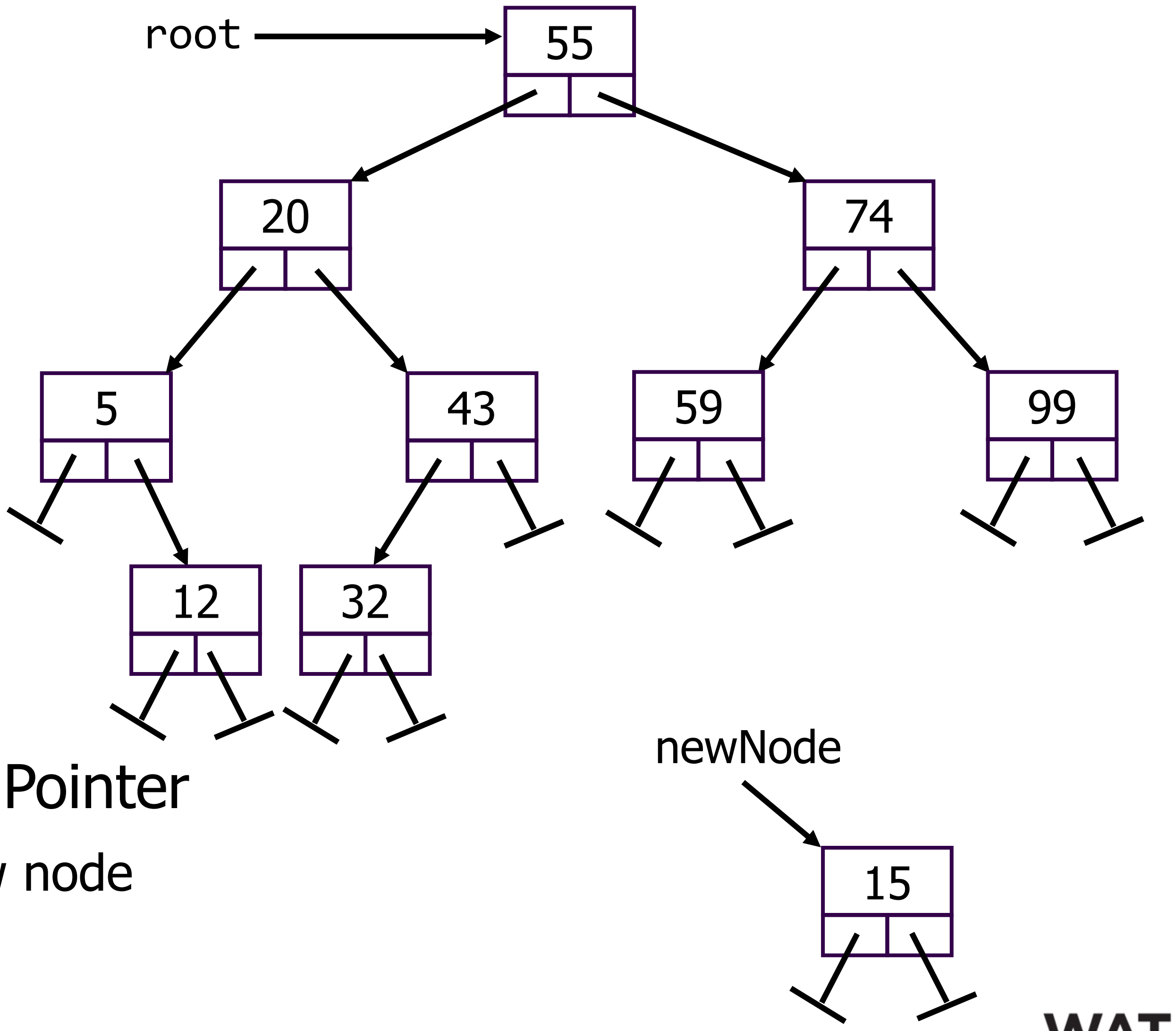

```
data < curr->data ?
Go left (curr = curr->left)
otherwise go right
```

```
also need to check
curr->left/right
against NULL
```

```
Once found, set
curr->left/right to
the new node
```

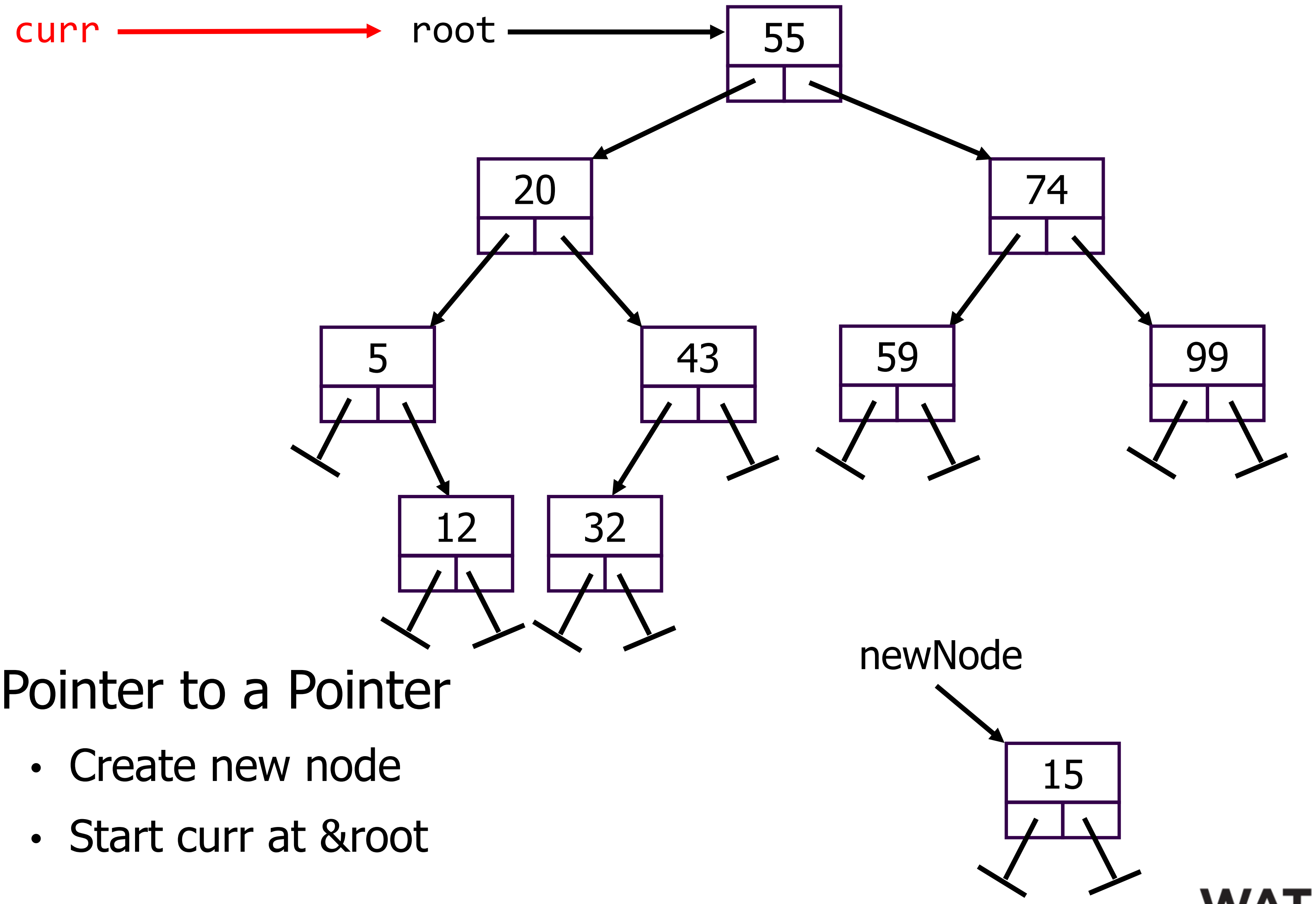
- Pointer to “node before”
  - Create new node
  - Search for “node before” (Parent)
  - Special case: empty tree → add as root

Pointer to a Pointer Approach



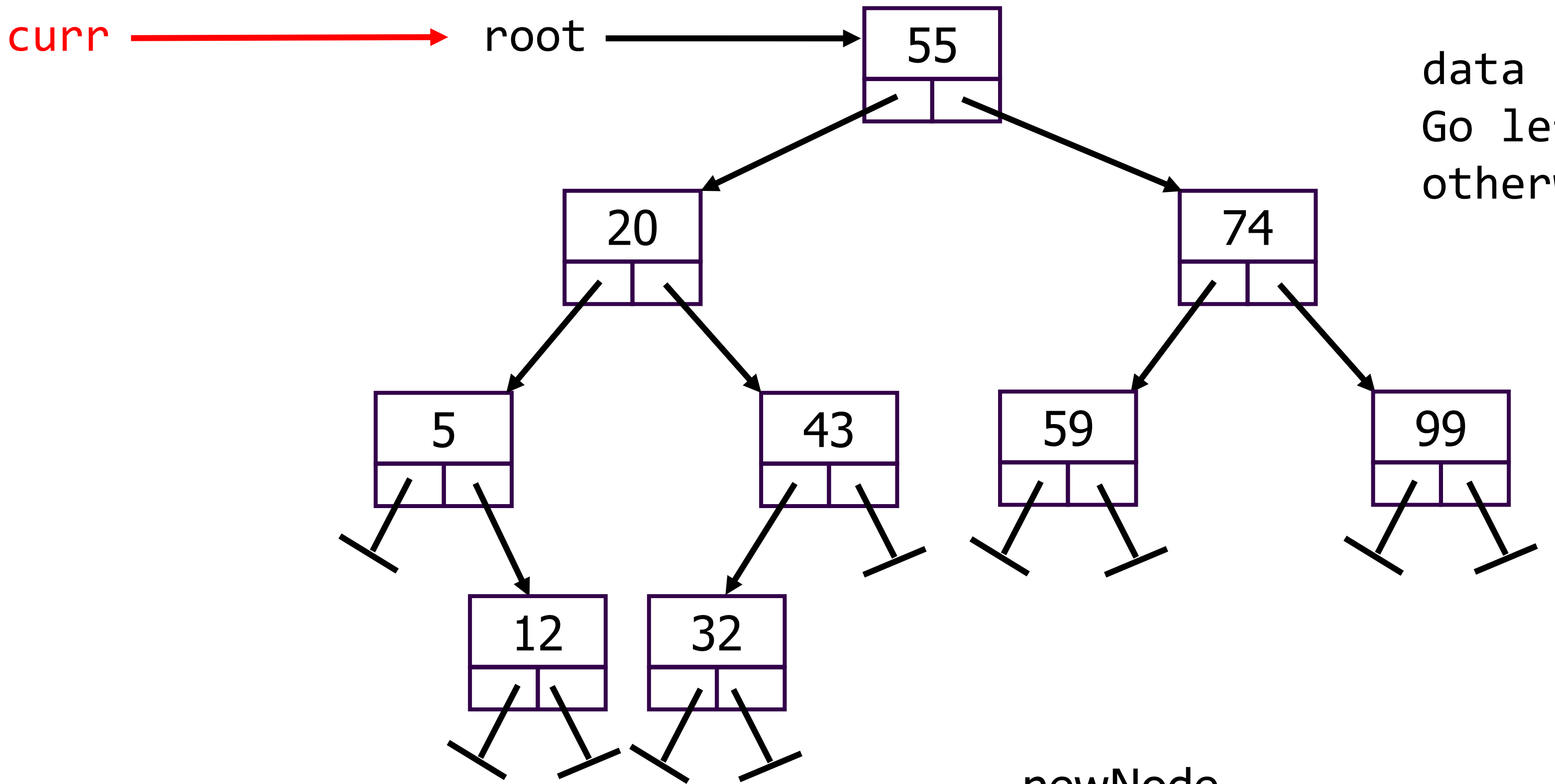
- Pointer to a Pointer
  - Create new node

Pointer to a Pointer Approach



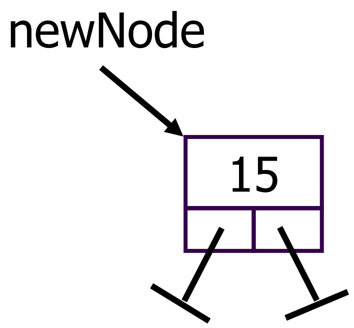
- Pointer to a Pointer
  - Create new node
  - Start curr at &root

Pointer to a Pointer Approach

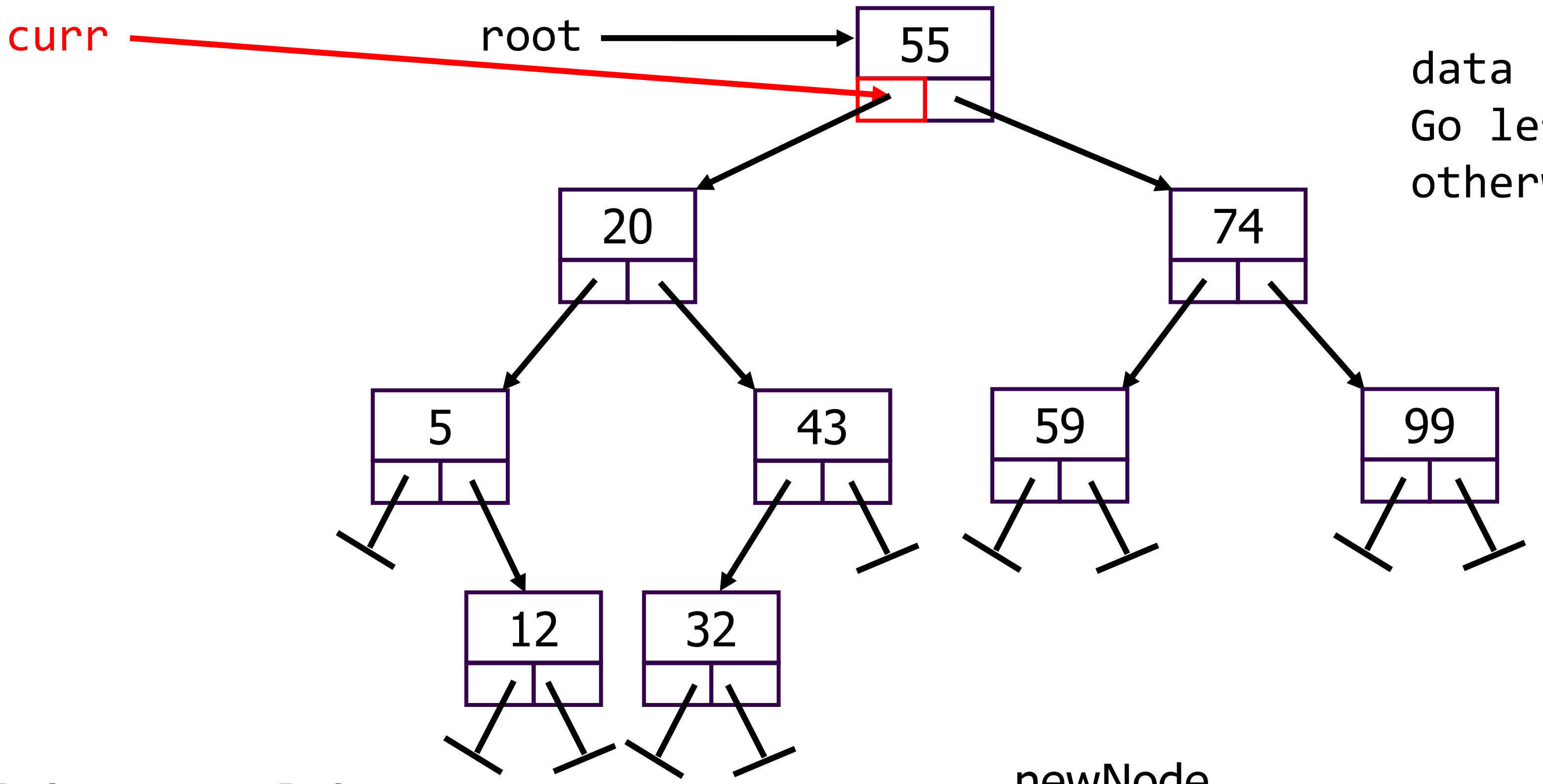



```
data < (*curr)->data ?
Go left; curr = &((*curr)->left)
otherwise go right
```

- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL

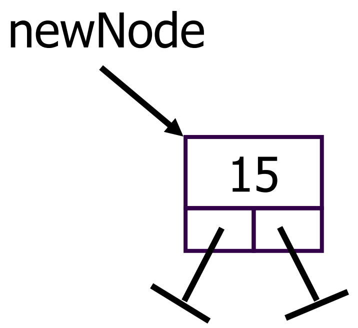


Pointer to a Pointer Approach

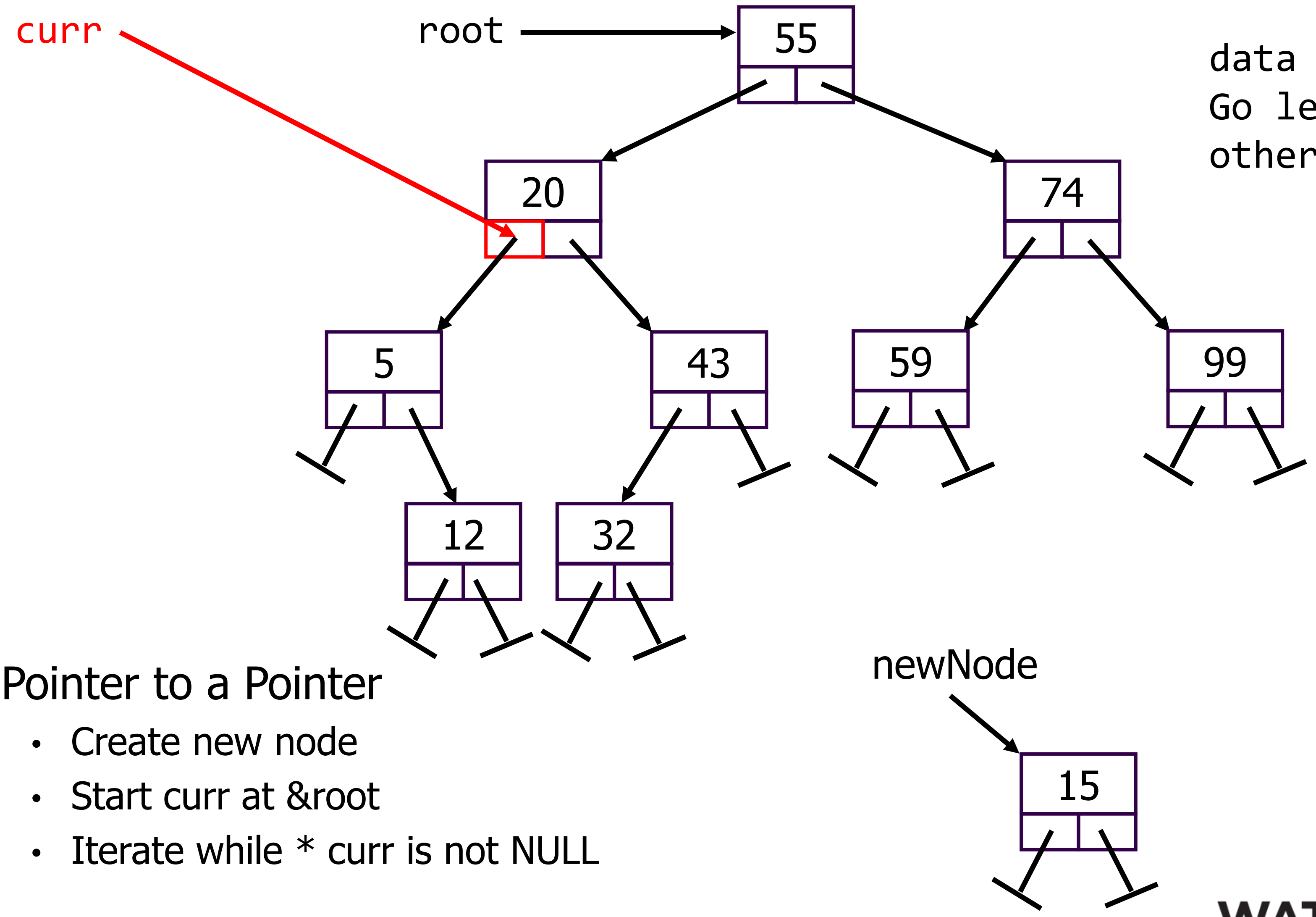



```
data < (*curr)->data ?
Go left; curr = &((*curr)->left)
otherwise go right
```

- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL



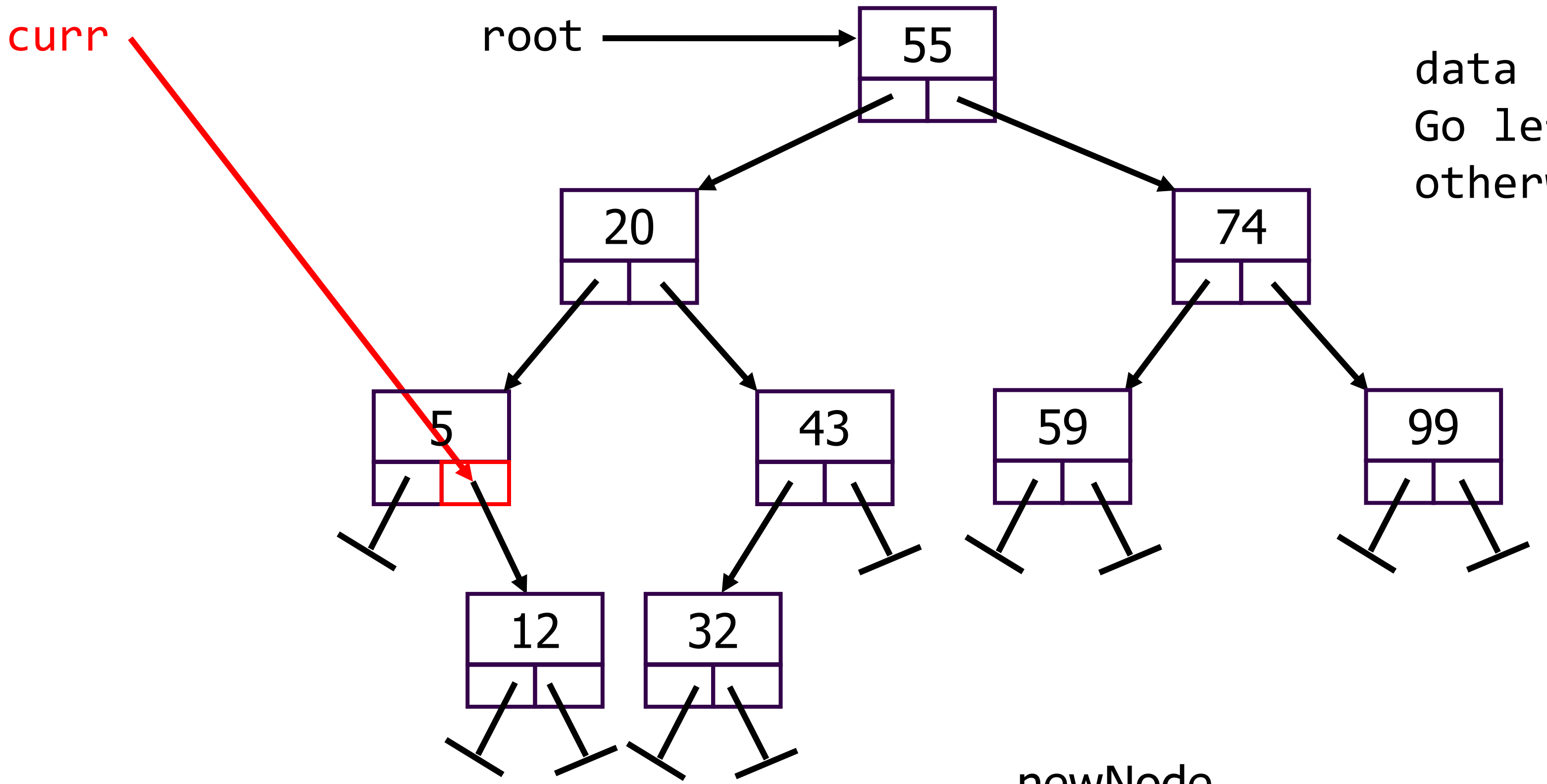
Pointer to a Pointer Approach




```
data < (*curr)->data ?
Go left; curr = &((*curr)->left)
otherwise go right
```

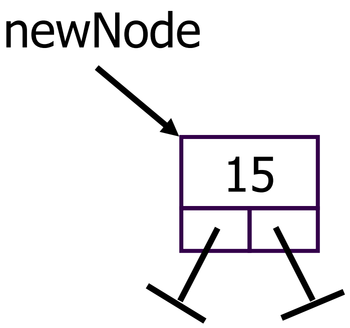
- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL

Pointer to a Pointer Approach

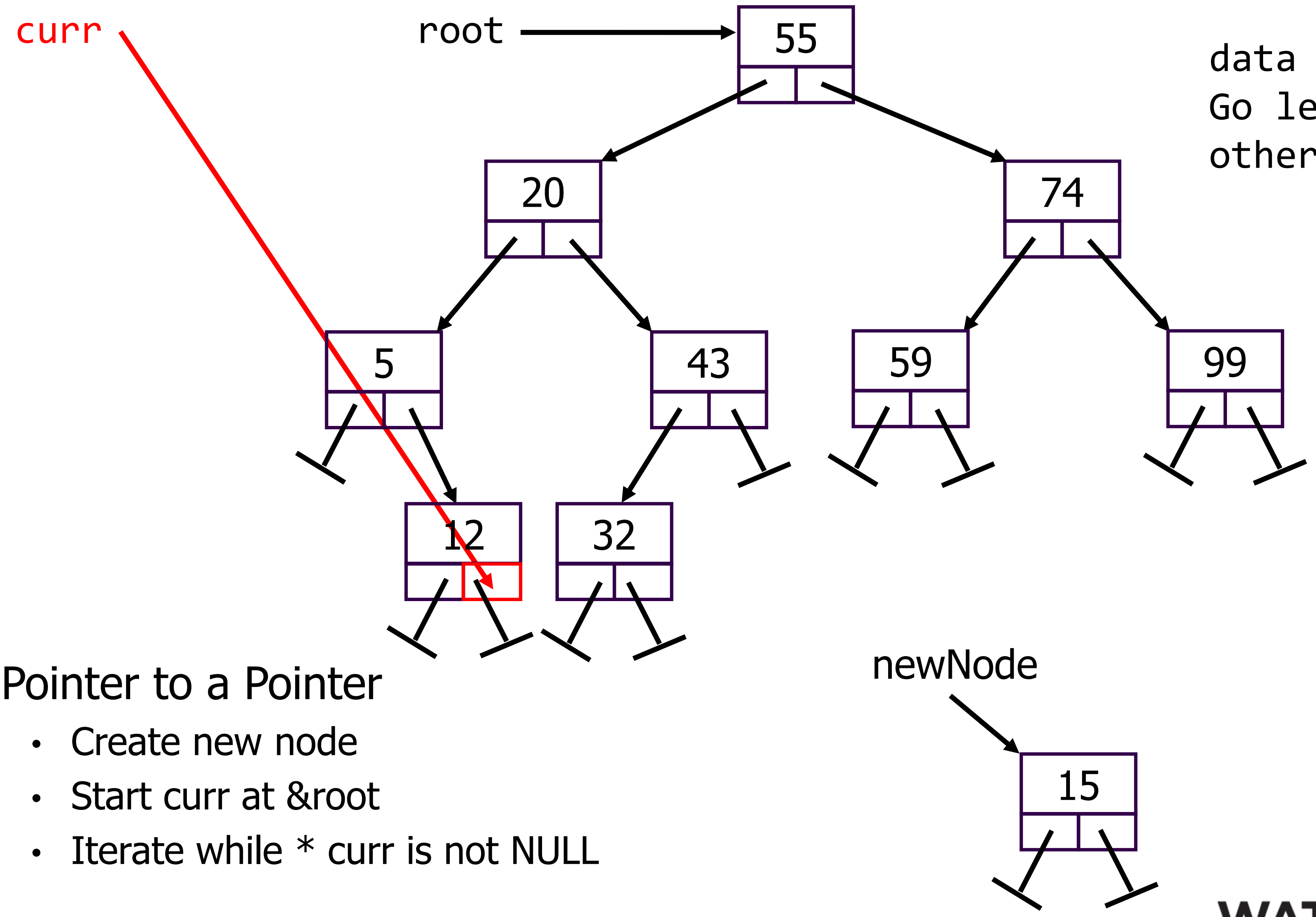



```
data < (*curr)->data ?
Go left; curr = &((*curr)->left)
otherwise go right
```

- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL



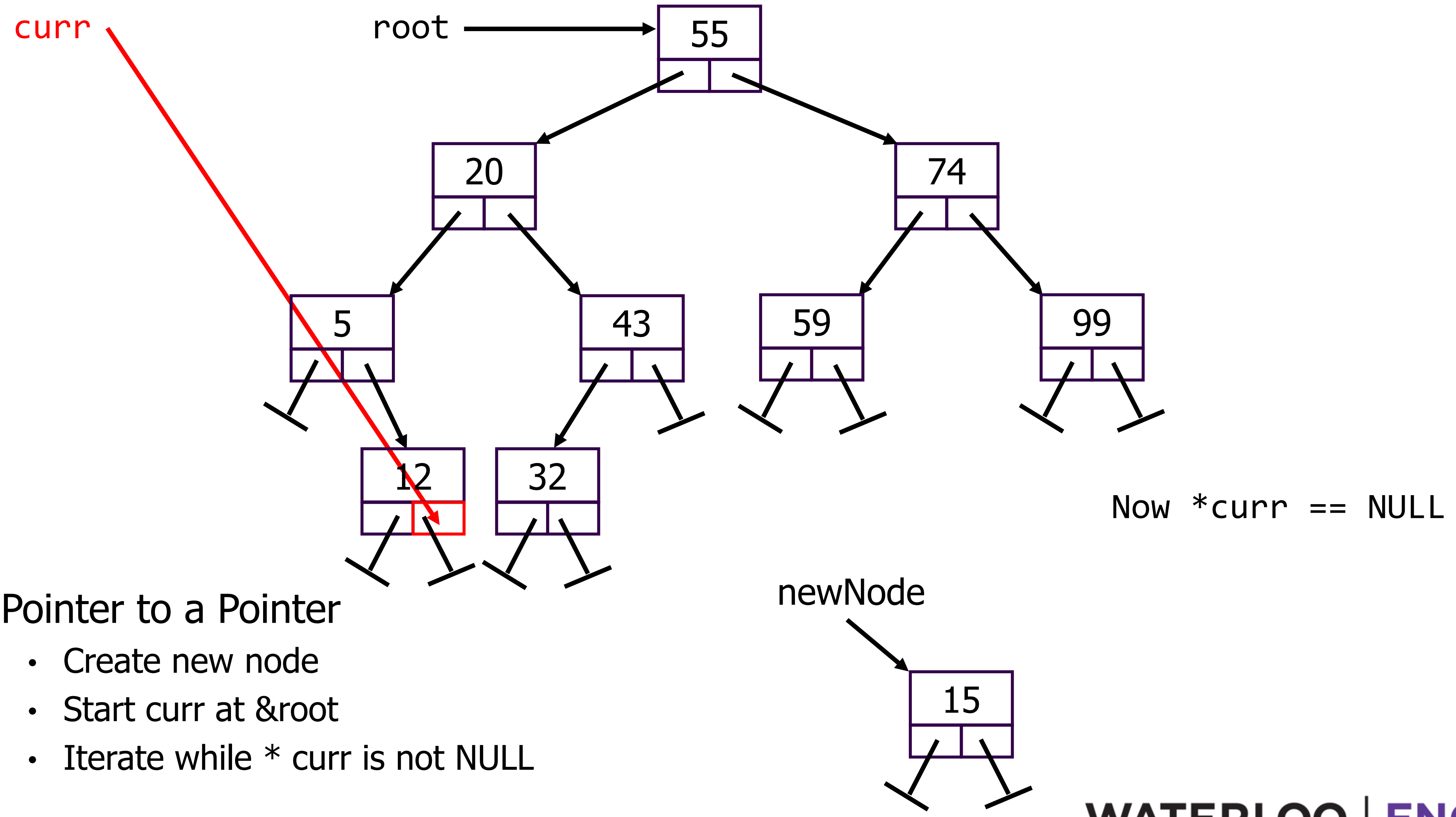
Pointer to a Pointer Approach




```
data < (*curr)->data ?
Go left; curr = &((*curr)->left)
otherwise go right
```

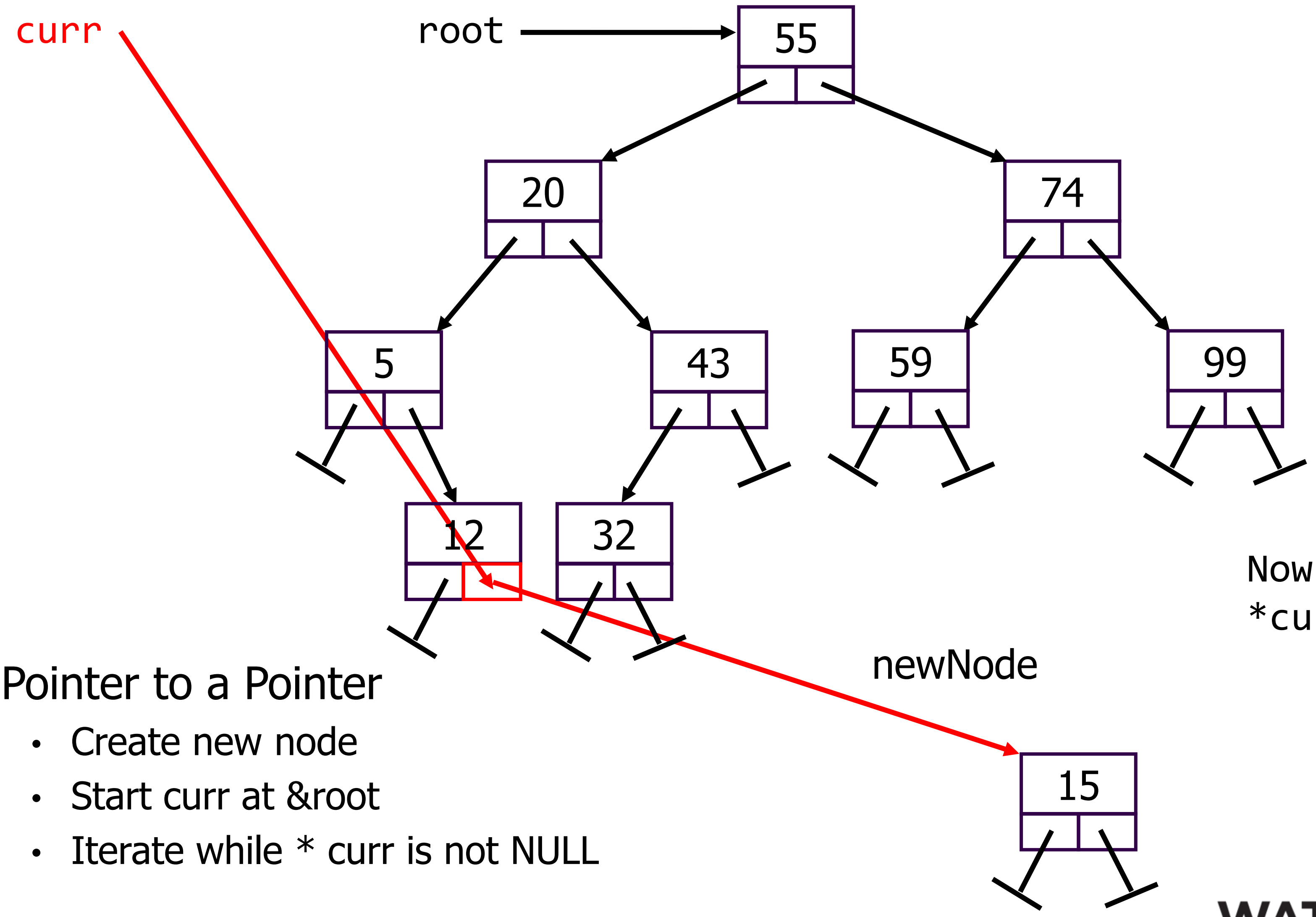
- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL

Pointer to a Pointer Approach



- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL

Pointer to a Pointer Approach



Now *curr == NULL
*curr = newNode

- Pointer to a Pointer
  - Create new node
  - Start curr at &root
  - Iterate while * curr is not NULL

Adding to BST with Recursion

- BSTs are naturally recursive data structures
  - If you have a BST and you go left, you have a BST
  - Same if you go right

- Can do add recursively too
  - Base case? Empty tree
    - What is the simplest tree to add?
  - Recursive case?
    - Compare the current node's data to the data we want to add
    - Recursively add to the appropriate sub-tree
    - Set the current node's left/right to the updated subtree (returned by recursion)

      Not strictly needed for every case

Adding with Recursion Algorithm

Check if current is NULL
- If so:
  - Make a new node(call it ans) with the data to add(call it toAdd)
  - My answer is ans
- If not:
  - Compare toAdd to current's data
    - If toAdd is less:
      - Add toAdd to current's left subtree (call the result newLeft)
      - Set current's left to newLeft
    - Otherwise:
      - Add toAdd to current's right subtree (call the result newRight)
      - Set current's right to newRight
    - My Answer is current

You should try this out before proceeding

Translating Algorithm to Code

```
void add (int toAdd) {
    root = add(root, toAdd);
}

Node * add (Node * current, int toAdd) {  // This should be private
```

Exercise for you

```
}
```

Removing from a Binary Search Tree

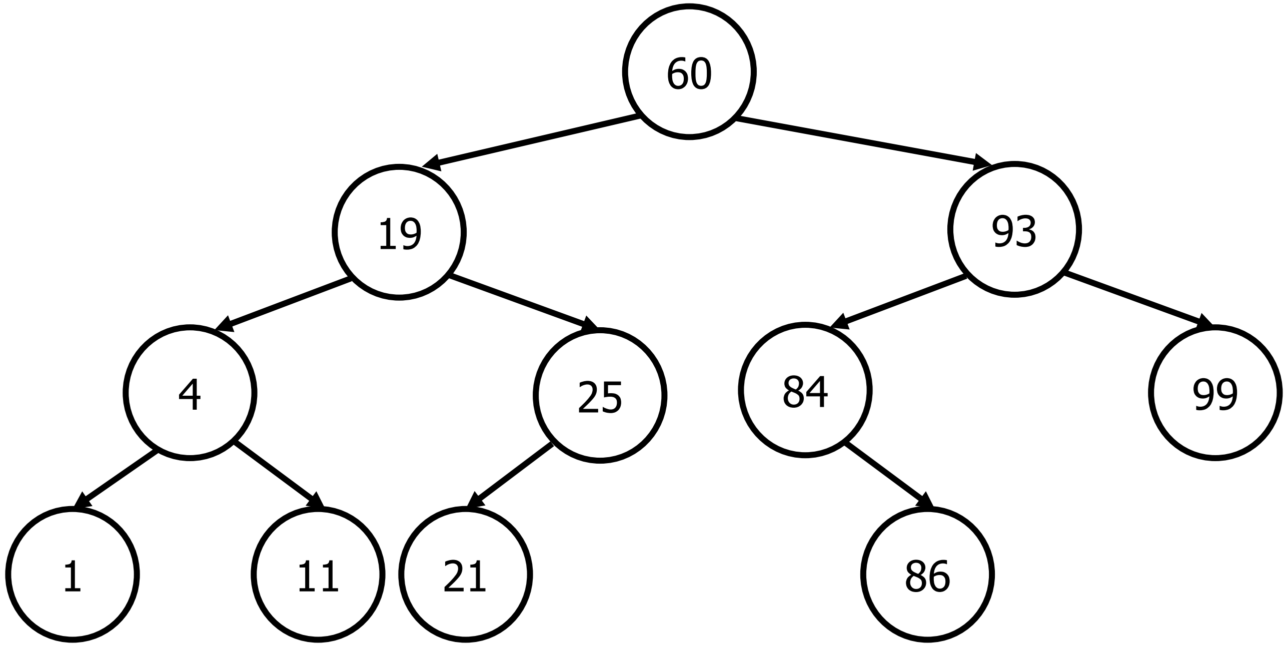
- Step 1: find the node
  - Using binary search & pointer to node before

- Step 2: pointer manipulation & delete
  - Case 1: leaf node
  - Case 2: has one child
  - Case 3: has two children

Easy! Pretty much like linked list removal

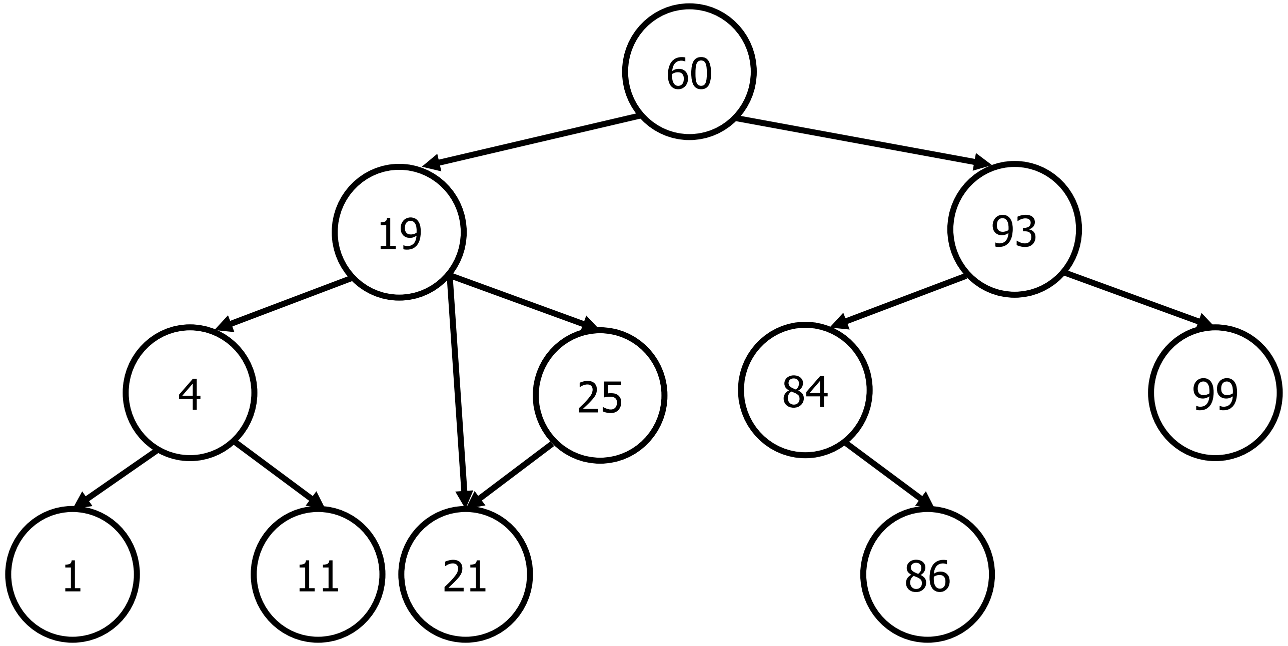
A bit more complicated …

Removing a Leaf Node



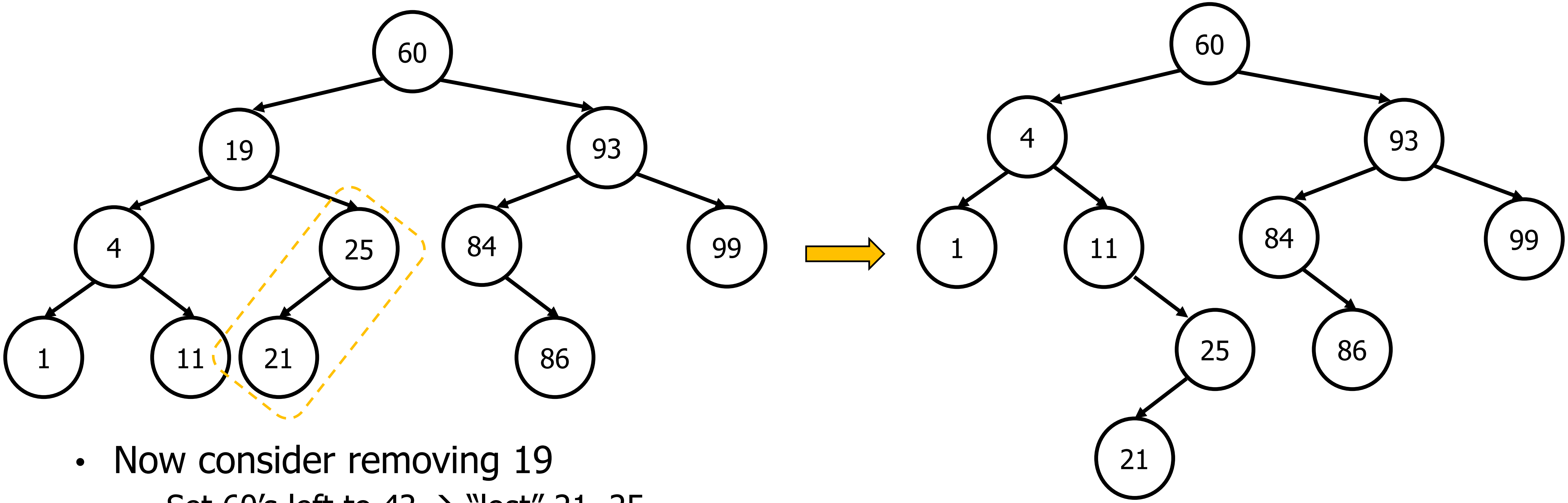
- Start with the easiest case: remove 1
  - Find the parent of the node to remove (4)
  - Set 4's left to NULL
  - Delete 1

Removing a Node with Single Child



- Now remove 25
  - Find the parent of the node to remove (19)
  - Set 19's right to 25's child (21)
  - Delete 25

Removing a Node with 2 Children



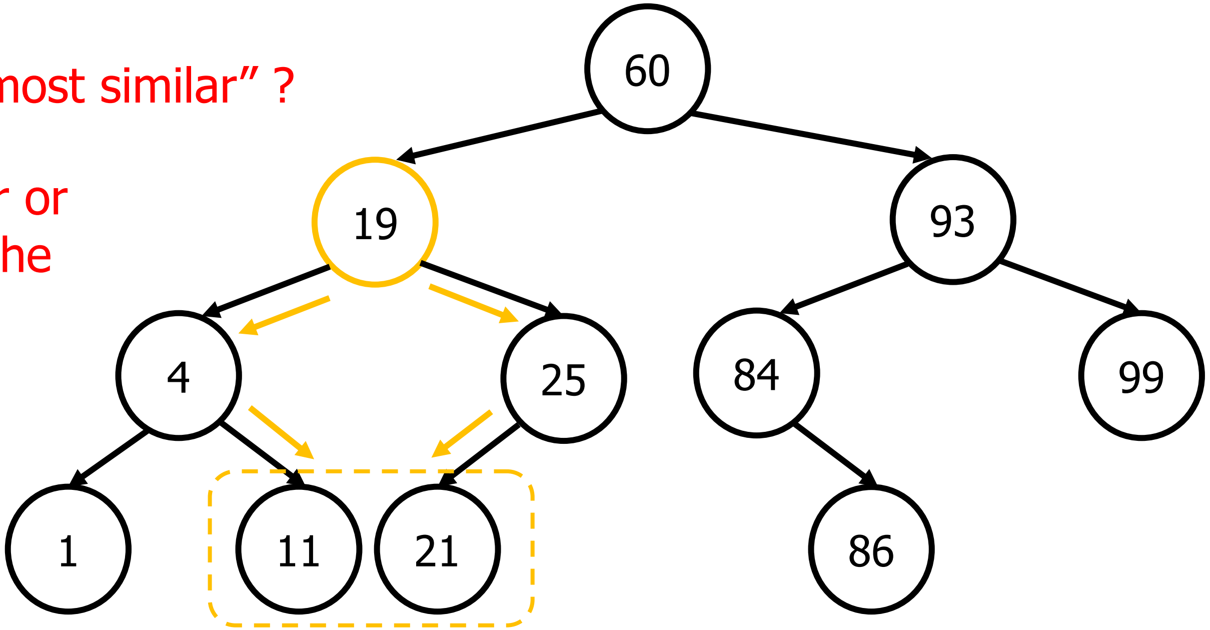
- Now consider removing 19
  - Set 60's left to 4? → "lost" 21, 25
    - Can reattaching the lost sub-tree (e.g., set 11's right to 25)
  - Works but result in imbalanced tree

Removing a Node with 2 Children

What do we mean by "most similar" ?

The immediately smaller or immediately greater in the ordering of the tree

e.g., 19's most similar node → 11 or 21



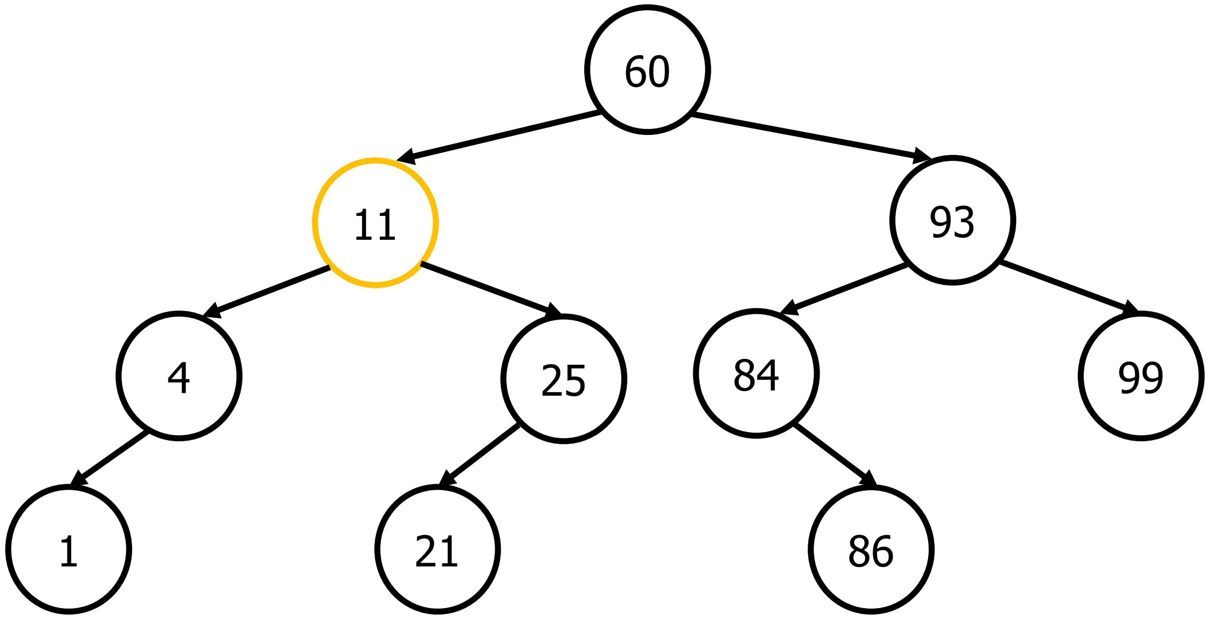
How do we find it?

Immediately smaller → left once, all the way right

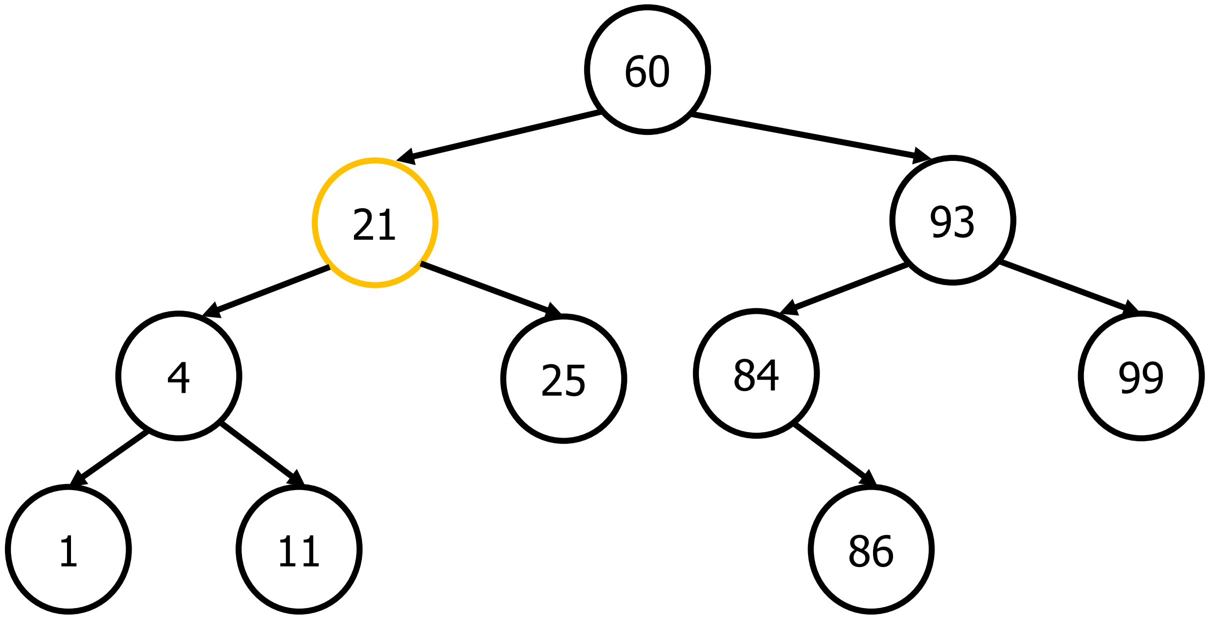
Immediately greater → right once, all the way left

- A better approach: replace with another node in the tree
  - Find most similar node in the tree with 0 or 1 child
  - Put its data into the node to remove
  - Remove that node instead

Removing a Node with 2 Children



Removing 19 by replacing it with 11



Removing 19 by replacing it with 21

Removing from BST with Recursion

- Base case(s)?
  - Empty tree
    - Either we start with an empty tree
    - After a few recursive calls, we end up with an empty tree
    - Either case, what we try to remove is not in the tree, do nothing
  - Found what we are looking for
    - Remove according to the 3 cases we discussed earlier Apparently this is too high-level

- Recursive case? Almost identical to adding to BST
  - Compare the current node's data to the data we want to remove
  - Recursively remove from the appropriate sub-tree
  - Set the current node's left/right to the updated subtree (returned by recursion)

Not strictly needed for every case

Translating Algorithm to Code

```
void remove (int toRemove) {
    root = add(root, toRemove);
}

Node * remove (Node * curr, int toRemove) {  // This should be private
    if (curr == NULL) {return NULL;} // Base case #1: empty tree
    if (curr->data == toRemove) {// Base case #2: found the node to remove
        if(curr->left == NULL) {// Exercise for you}
        if(curr->right == NULL) {// Exercise for you}
        //helper function to remove a node with 2 children
        curr->left = twoChildRm(curr->left, curr);
        return curr;
    }
    else if (toRemove < curr->data) {//Recursive cases
        curr->left = remove(curr->left, toRemove);
    } else {
        curr->right = remove(curr->right, toRemove);
    }
    return curr;
}
```

Missing the case with 0 children?

First one already covers that

Think about what we do in the first case?
Delete, then return right subtree

Also works for leaf node, right subtree is just NULL, which is what we want to return

Takeaway: recognize similarities, reduce cases

Translating Algorithm to Code

```
Node * twoChildRm (Node * curr, Node * replace) {
      //can’t go right anymore, found the immediate smaller node
      if (curr->right == NULL) {
            replace->data = curr->data;
            Node * temp = curr->left;
            delete curr;
            return temp;
      }
      //recurse right, look for the immediate smaller node
      curr->right = twoChildRm (curr->right, replace);
      return curr;
}
```

Same as removing a node with 0 or 1 child

Performance Summary

	Array	LinkedList	Sorted Array	BST
add	O(n)	O(1)	O(n)	O(log(n)) [*]
remove	O(n)	O(n)	O(log(n))	O(log(n)) [*]
search	O(n)	O(n)	O(n)	O(log(n)) [*]

- BSTs:
  - Doing pretty good on performance: O(log(n)) is very fast
  - … but the [*] there is because its not exactly true
    - Need to keep the tree balanced
    - Will learn how in the next few lectures

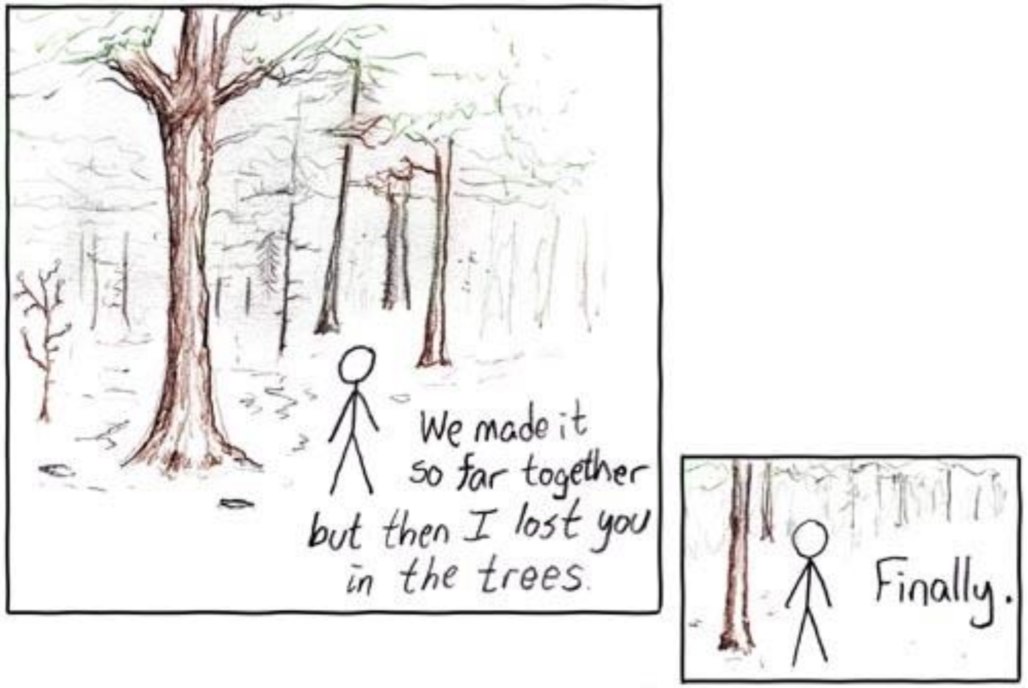
Wrap Up

- In this lecture we talked about
  - Application of BSTs
  - Binary search with BST
  - Implementation of key operations
    - Add
    - Remove

- Next up
  - Balanced BSTs: AVLs & Red-Black Trees

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 4.1 – 4.3



source

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)
  - Mary Hudachek-Buswell (Gatech)