CS 106B, Lecture 20
Binary Search Trees

Plan for Today

- How to implement a Set
  - The importance of choosing a good data structure
- Trees, a new kind of data structure
- ”Reading" trees today – modifying trees will be tomorrow

Review

- What were the pros of a linked list?

- What were the cons of a linked list?

Review

- What were the pros of a linked list?
  - Easy to insert/remove

- What were the cons of a linked list?
  - Difficult to search through

Designing a Set

- We've seen how to implement:
  - Stack (array or linked list)
  - Vector (array)
  - Queue (linked list)
- How would we implement Set?
  - Add
  - Contains
  - Remove

First Try

- Store all the elements in an **unsorted** array or linked list
  - What is the Big-Oh of contains?
  - What is the Big-Oh of adding an element?
  - What is the Big-Oh of removing an element?

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
3	8	9	7	5	12	4	8	1	6	75

Another attempt

- What if we **sorted** the array?
  - What is the Big-Oh of contains?
  - What is the Big-Oh of adding an element?
  - What is the Big-Oh of removing an element?

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search

- Fast way to search for elements in a **sorted array**
- Looping through elements one by one is slow [O(N)]
- Idea:

Jump to the middle element:

if the middle is what we're looking for, we're done. Hooray!

if the middle is too small – we rule out the entire **left side** of elements smaller than the middle element

if the middle is too big – we rule out the entire **right side** of elements bigger than the middle element

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search in Action

- Search for 8:

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search in Action

- Search for 8:

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search in Action

- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search in Action

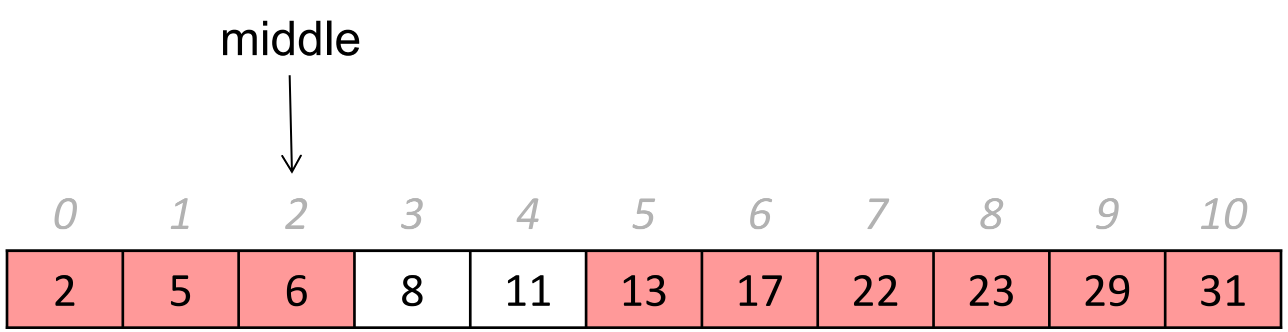
- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Pick the new middle of the remaining elements
- Look at 6:

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

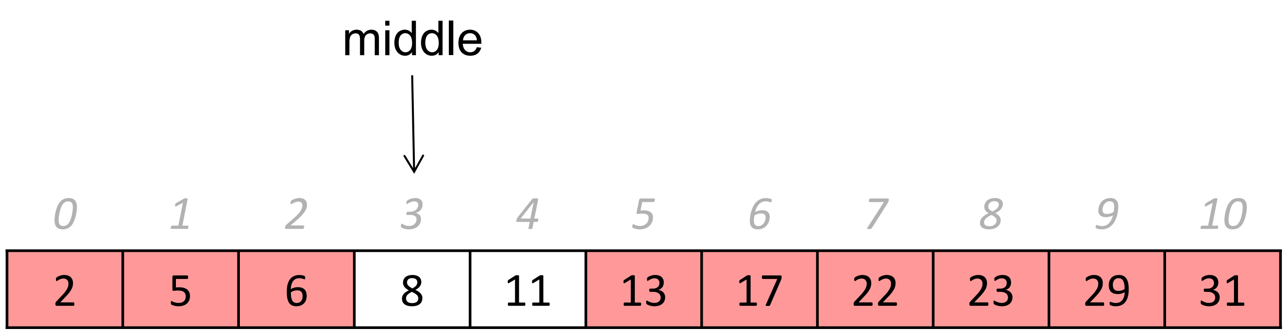
Binary Search in Action

- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Pick the new middle of the remaining elements
- Look at 6:
  - it's too small, so we rule out indices 0-3



Binary Search in Action

- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Pick the new middle of the remaining elements
- Look at 6:
  - it's too small, so we rule out indices 0-3
- Look at 8:
  - it's just right! We return true



Binary Search in Action

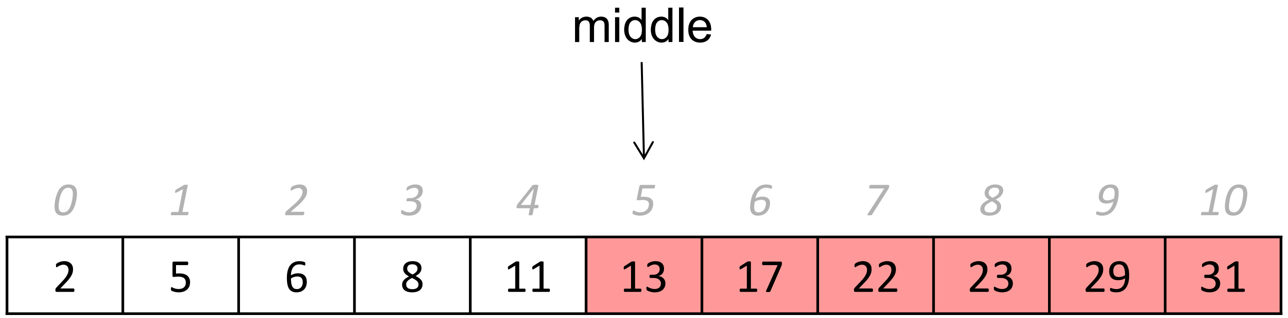
- Search for 7:

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

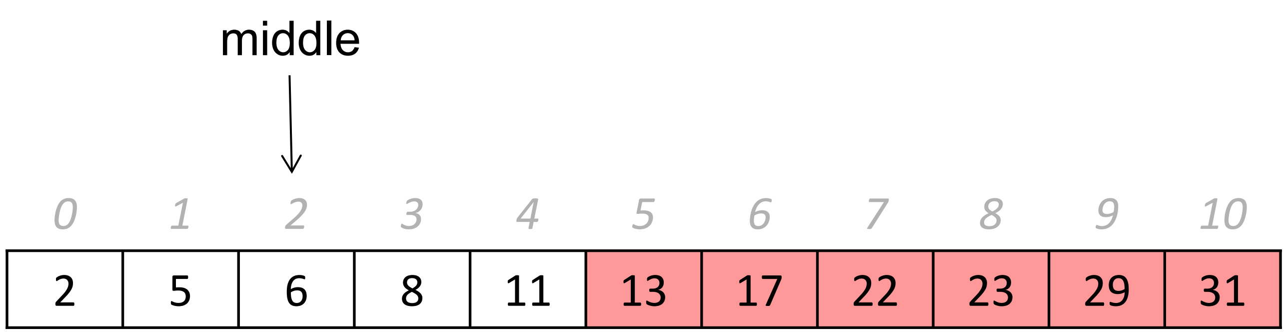
Binary Search in Action

- Search for 7:
- Look at 13
  - it's too big, so we rule out indices 5-10



Binary Search in Action

- Search for 7:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Pick the new middle of the remaining elements
- Look at 6:



Binary Search in Action

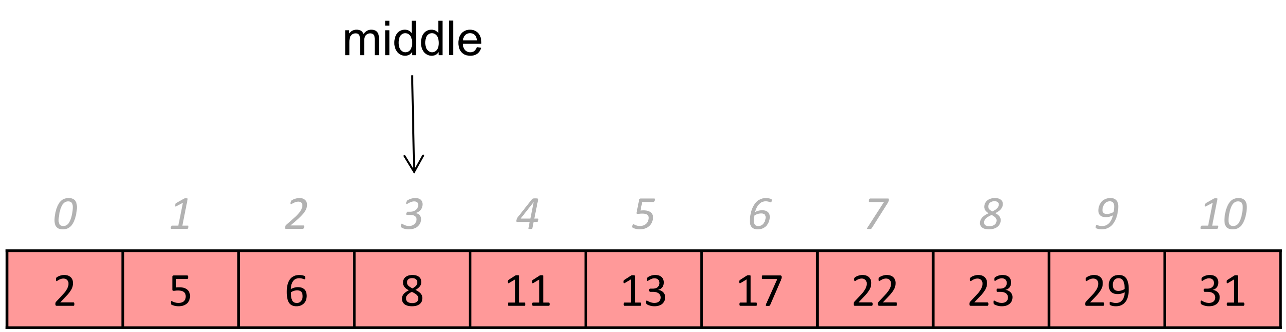
- Search for 7:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Pick the new middle of the remaining elements
- Look at 6:
  - it's too small, so we rule out indices 0-3

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Binary Search in Action

- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Look at 6:
  - it's too small, so we rule out indices 0-3
- Look at 8:
  - it's too big! We rule out elements 3-4



Binary Search in Action

- Search for 8:
- Look at 13
  - it's too big, so we rule out indices 5-10
- Look at 6:
  - it's too small, so we rule out indices 0-3
- Look at 8:
  - it's too big! We rule out elements 3-4
- No elements left to search – we return false

middle

0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Sorted Array

- What if we **sorted** the array?
  - What is the Big-Oh of contains?
    - O(log N)
  - What is the Big-Oh of adding an element?
    - O(N)
  - What is the Big-Oh of removing an element?
    - O(N)

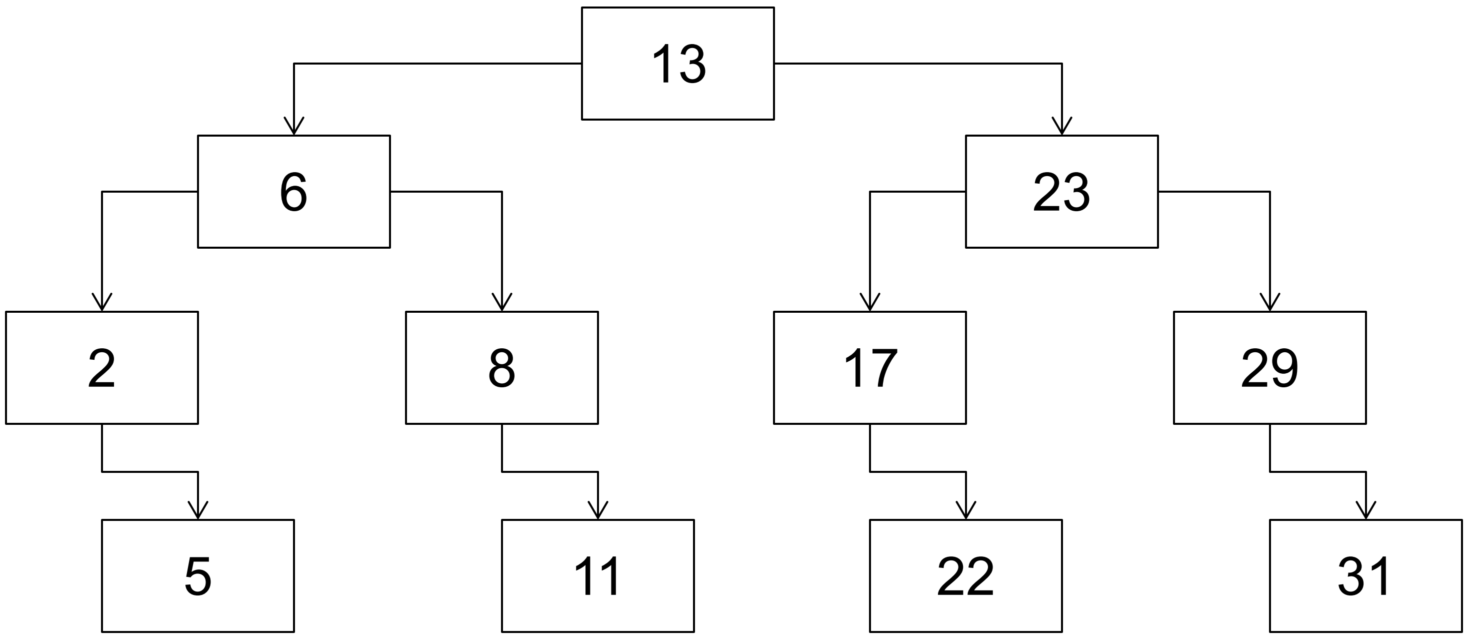
0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

A Modification

- Problem: an array is slow to insert into or remove from
- Our solution was a **linked list** – have each element connected to one other element
  - Easy to add/remove elements
  - Can't skip elements – need to go in order
- Maybe we can find some way to implement the jumps necessary for binary search...

A Modification

- What are all the possible paths binary search could take on this array?



0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

A Modification

- We always jump to one of two elements in binary search (depending on if the element we're looking at is too big or too small)
- What if we had a Linked List where we stored two pointers, allowing us to make those jumps quickly?

Binary Search Tree

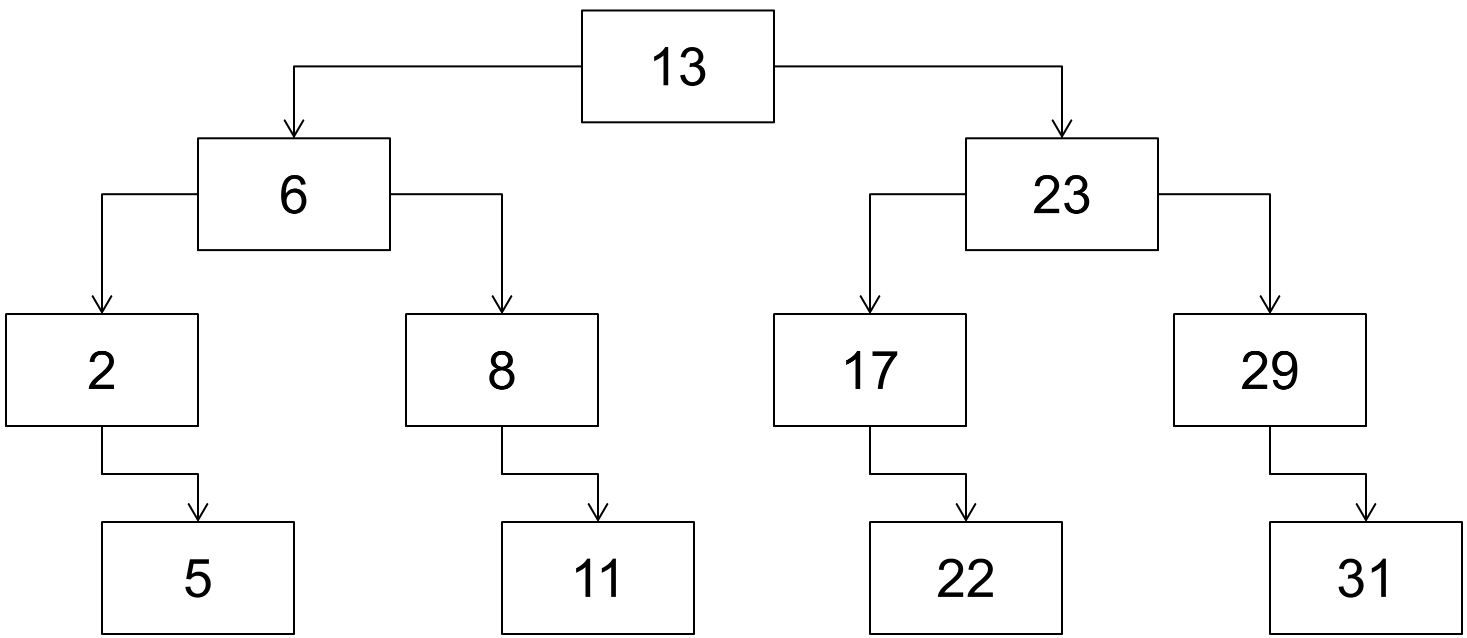
- A **tree** is a data structure where each element (**parent**) stores two or more pointers to other elements (its **children**)
  - A doubly-linked list doesn't count because, just like outside of computer science, a child can not be its own ancestor
- Each node in a **binary tree** has two pointers
  - Some of these pointers may be `nullptr` (just like in a linked list)
  - We'll see examples of non-binary trees in future lectures
- A **binary search tree** is a binary tree with special ordering properties that make it easy to do binary search
- Similar to a Linked List:
  - Each element in its own block of memory
  - Have to travel through pointers (can't skip "generations")

(Binary) TreeNode

```
struct TreeNode {
    int data; // assume that the tree stores ints
    TreeNode *left;
    TreeNode *right;
};
```

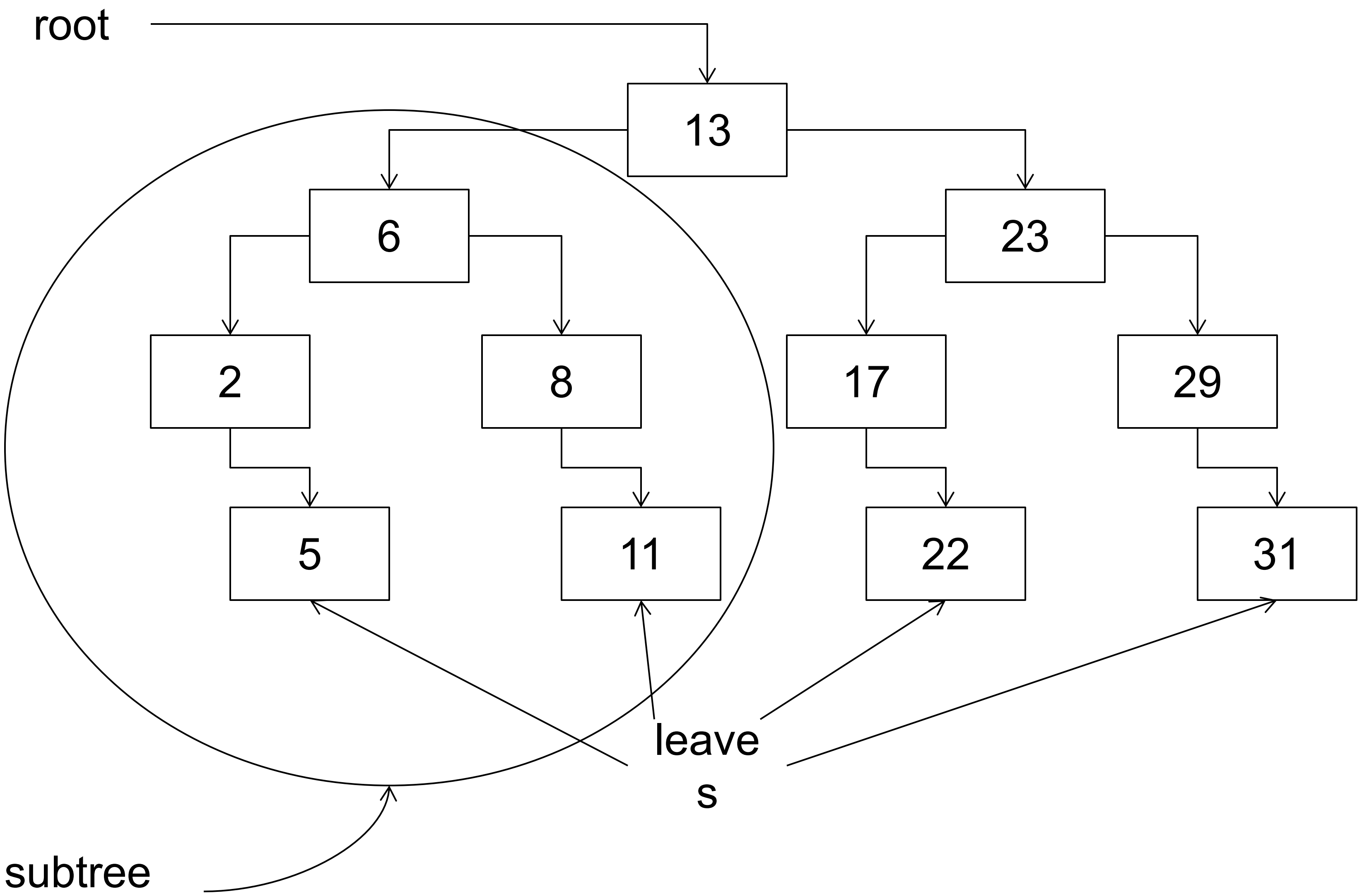
Binary Search Trees

- A binary search tree has the following property:
  - All elements to the left of an element are smaller than that element
  - All elements to the right of an element are bigger than that element
  - Just like our sorted array!



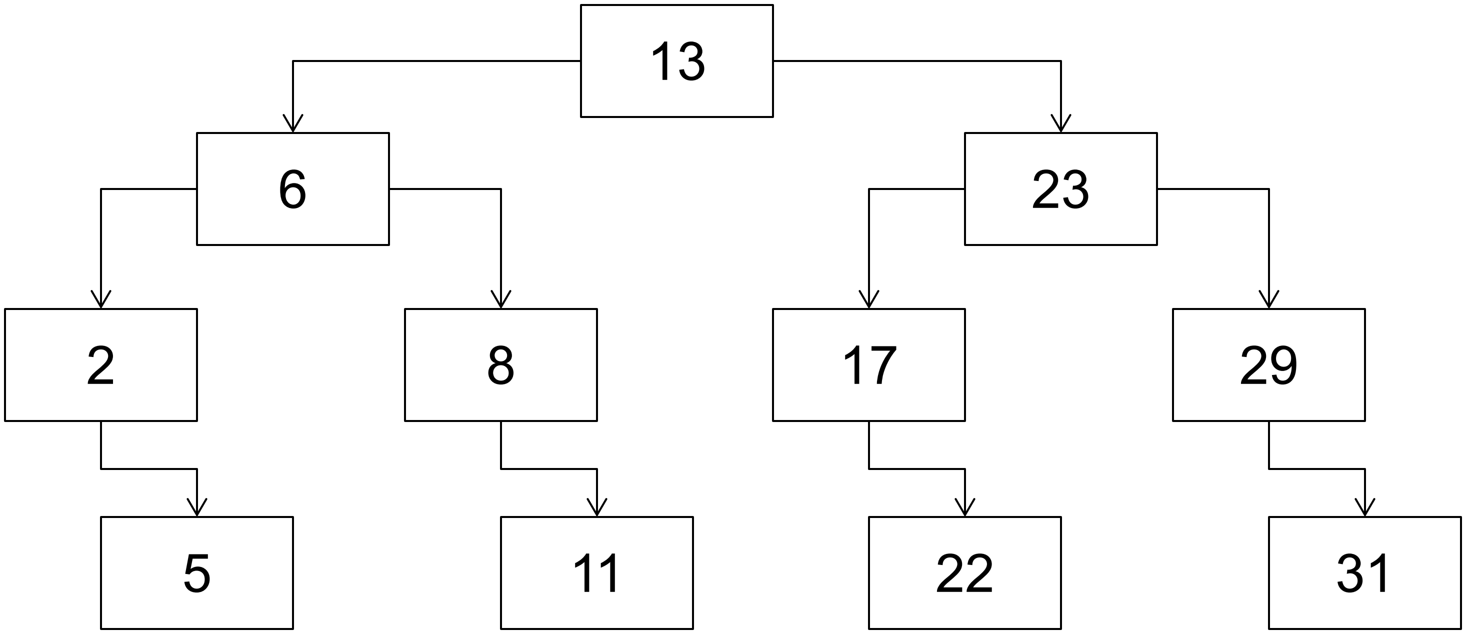
0	1	2	3	4	5	6	7	8	9	10
2	5	6	8	11	13	17	22	23	29	31

Tree anatomy



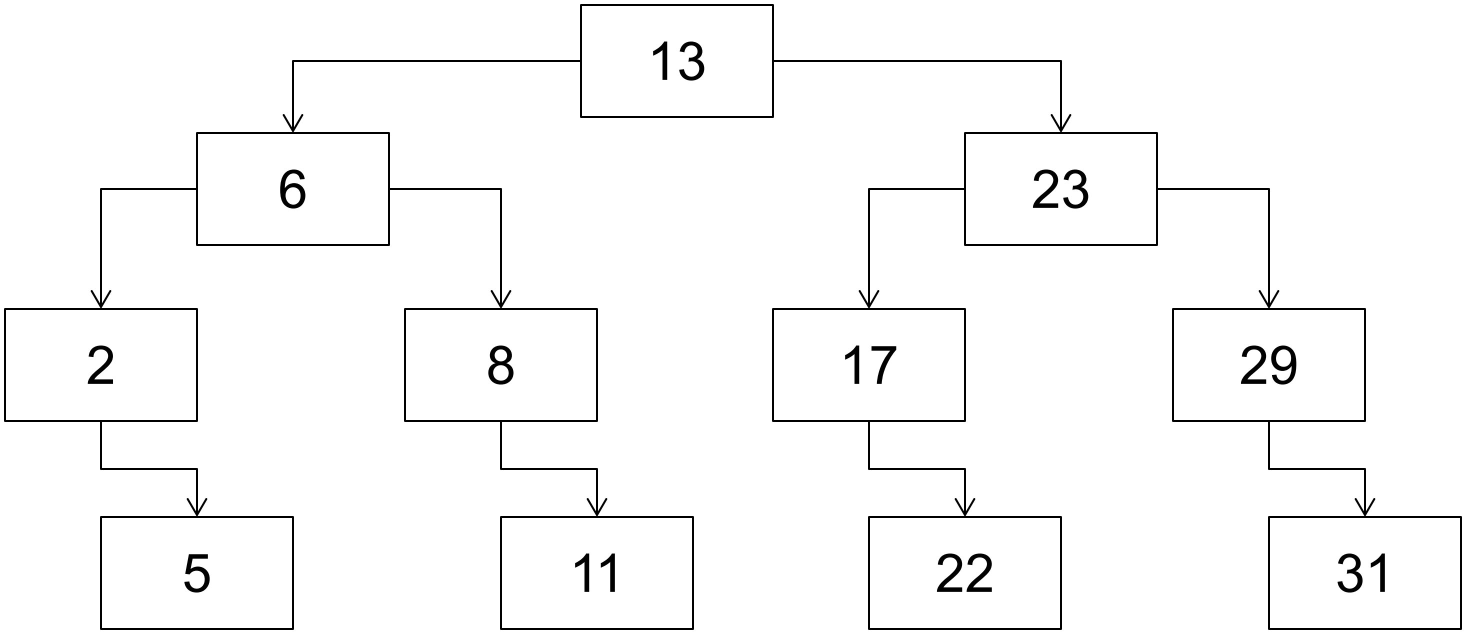
BST Contains

- How would you search a BST for an element?



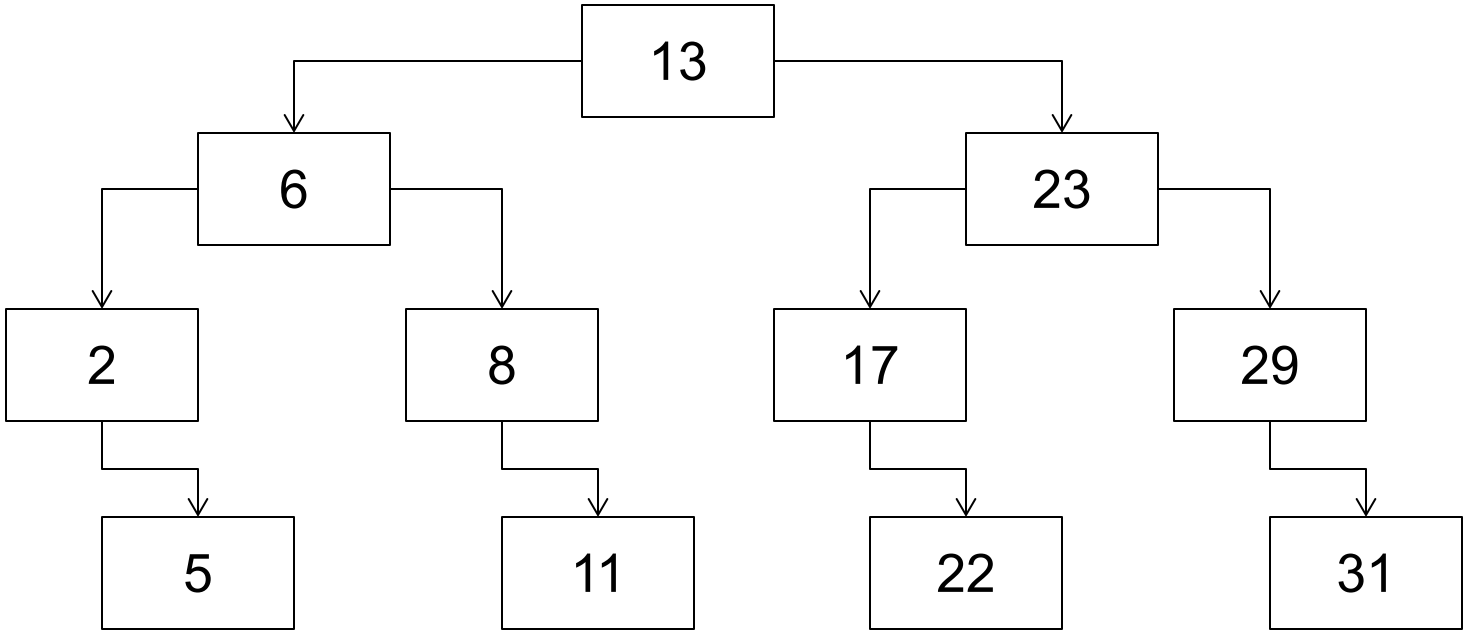
BST Contains

- How would you search a BST for an element?
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



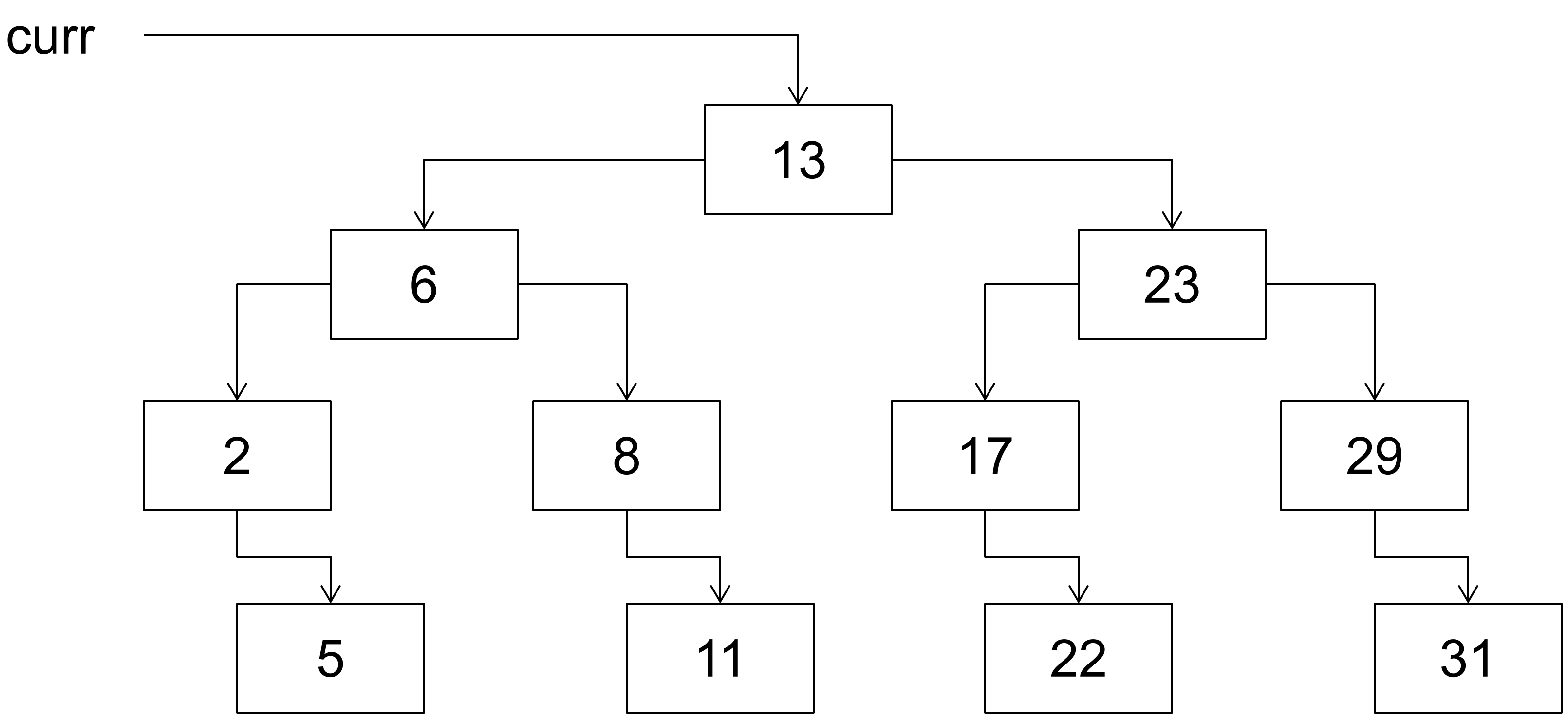
Trees and Recursion

- Trees are fundamentally **recursive (**subtrees are smaller trees)
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



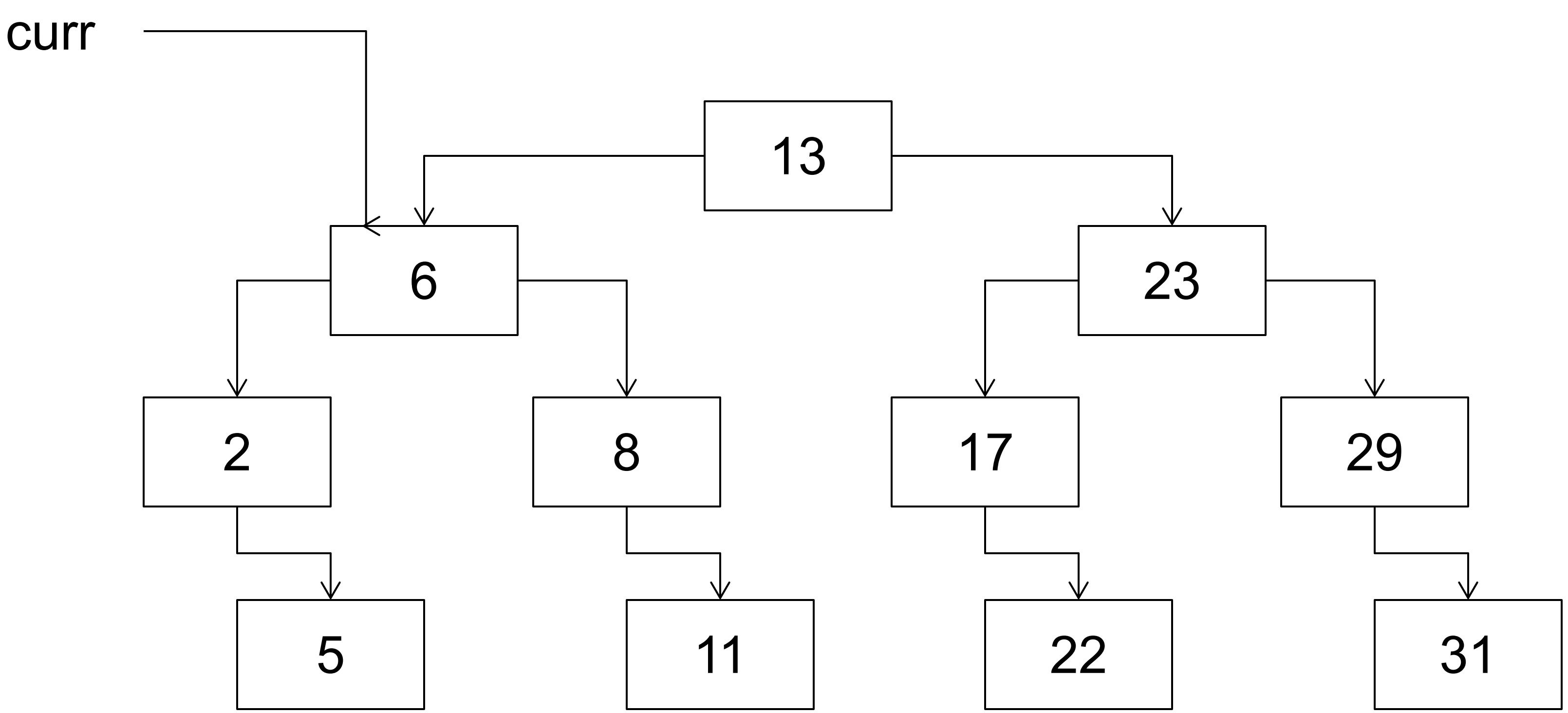
Trees and Contains

- Search for 5
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



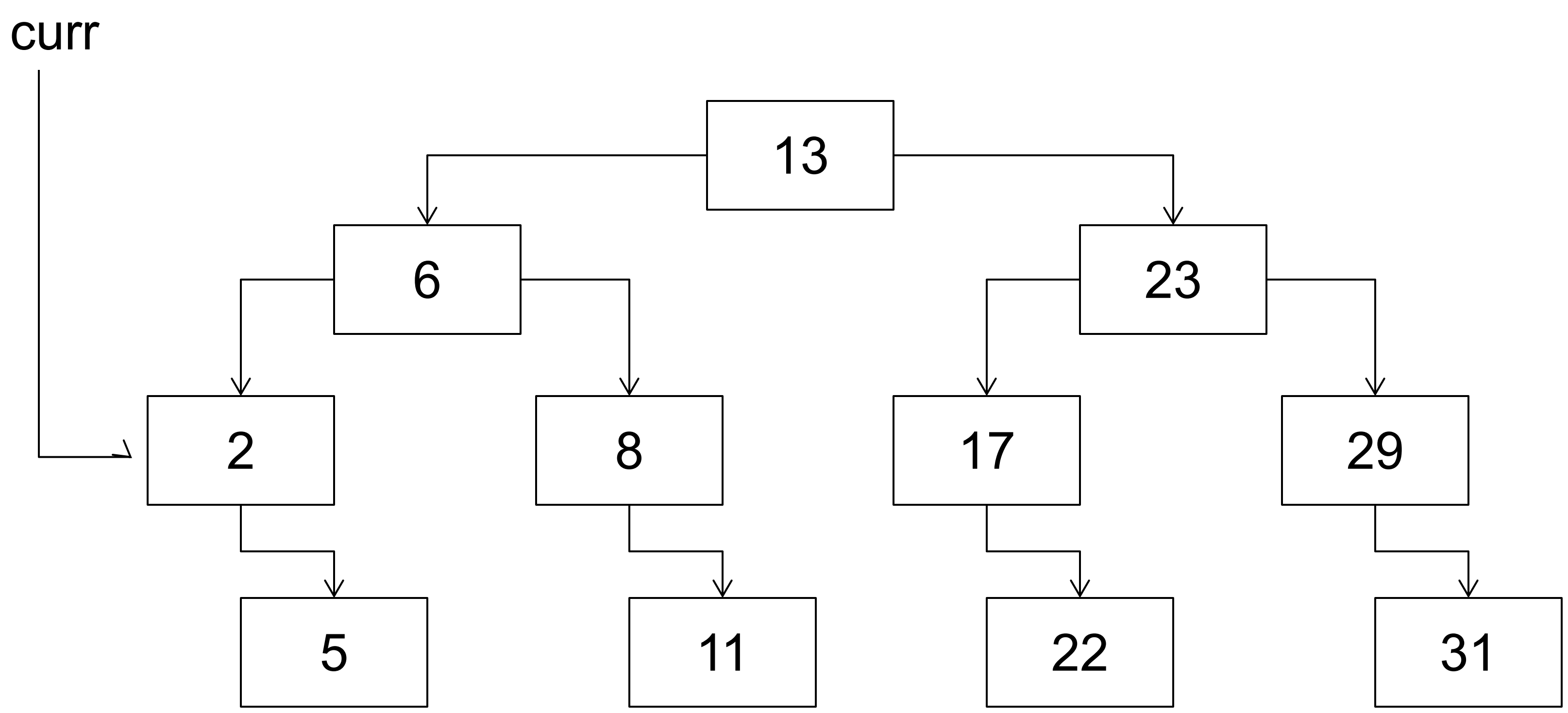
Trees and Contains

- Search for 5
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



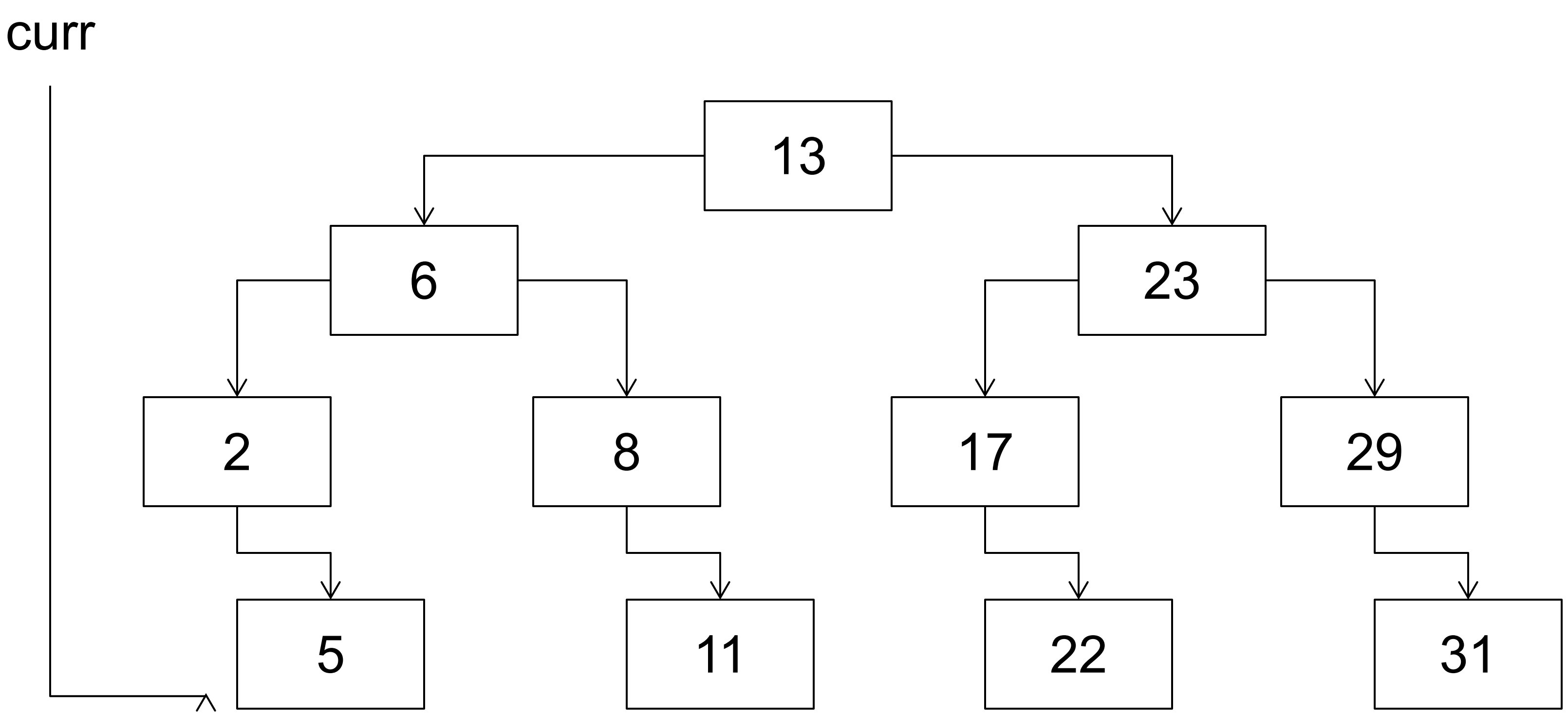
Trees and Contains

- Search for 5
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



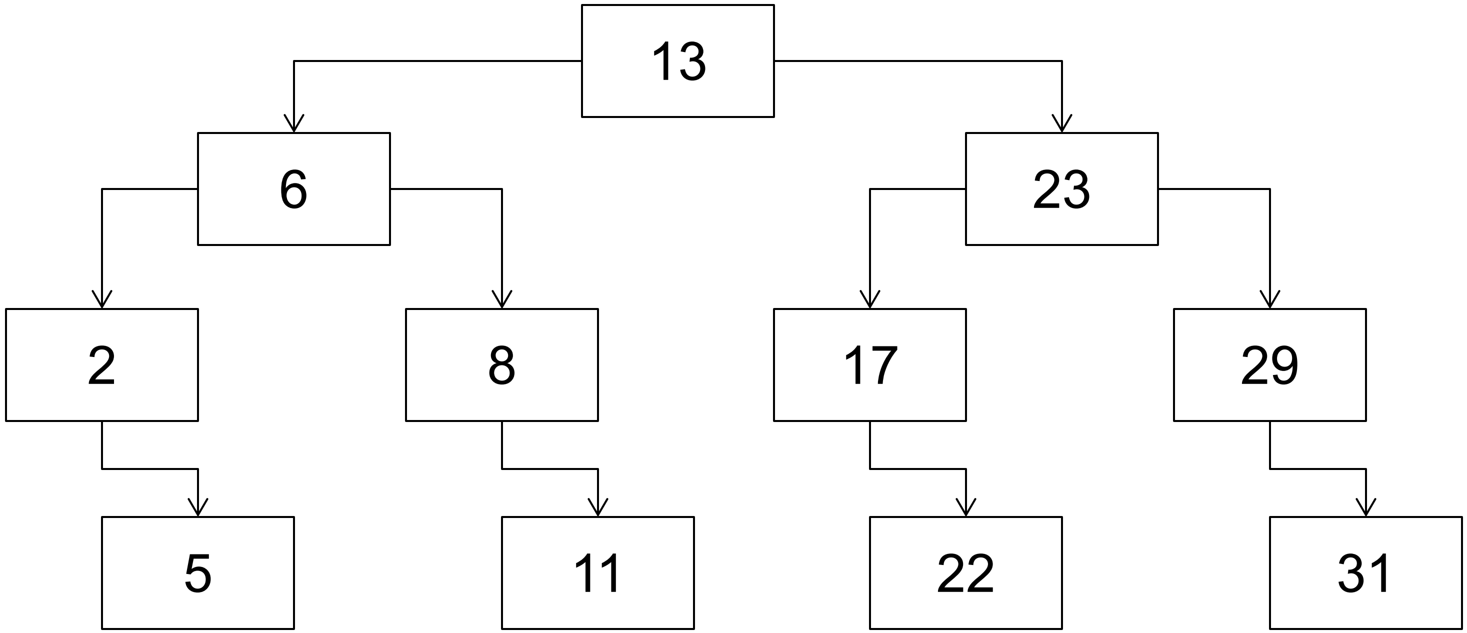
Trees and Contains

- Search for 5
- Start at root:
  - If root is too big, go left (entire right subtree is too big)
  - If root is too small, go right (entire left subtree is too small)



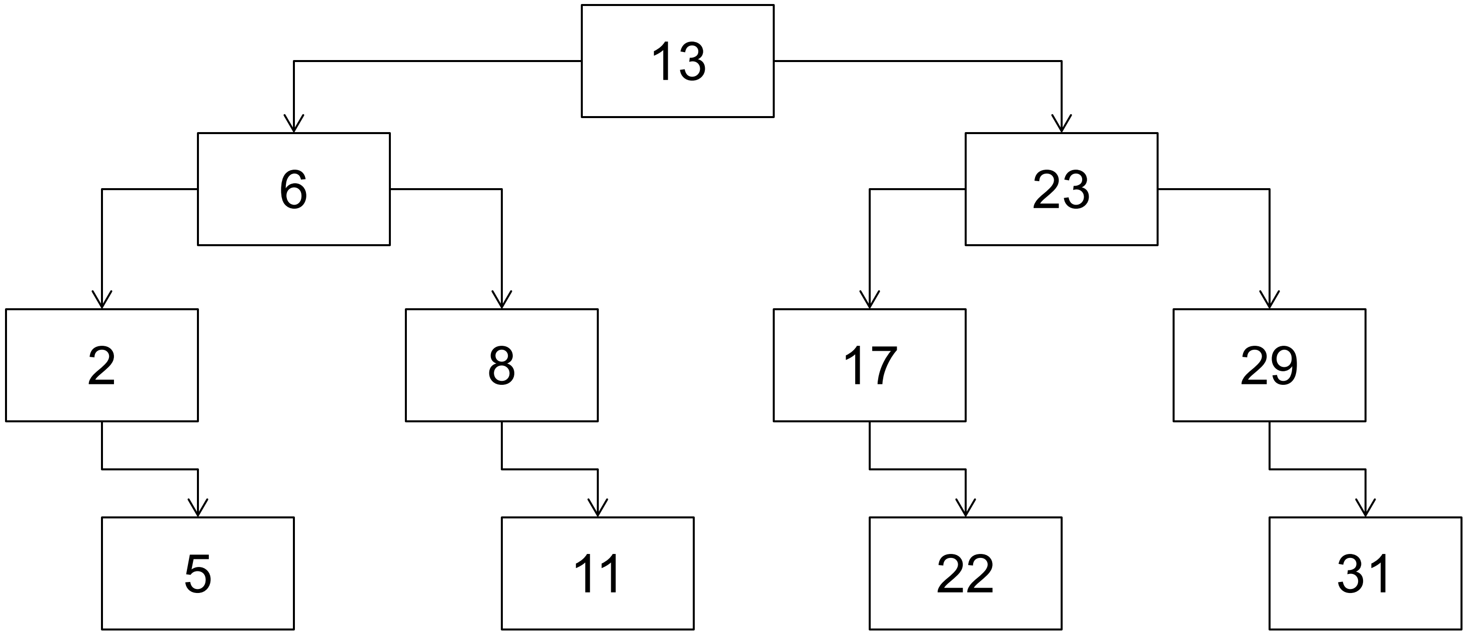
Printing Trees

- We need to be able to print our Set
- How would we print a tree?



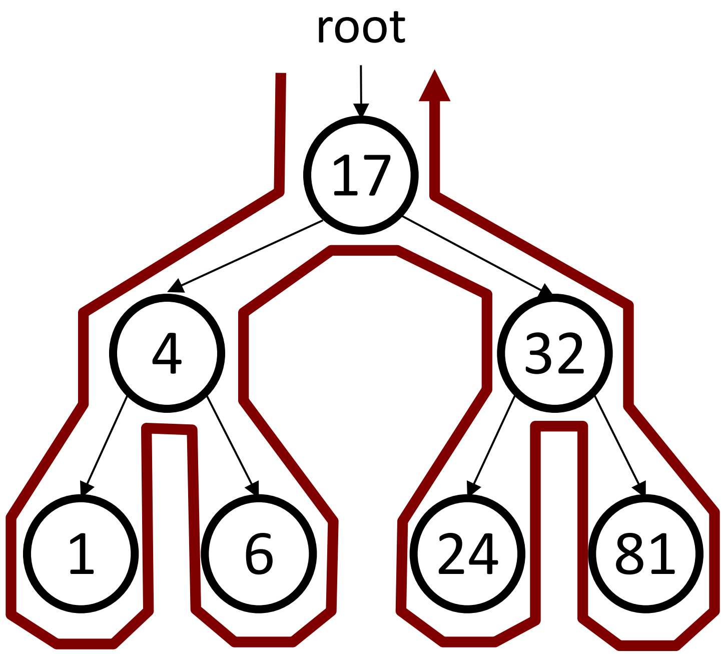
Printing Trees

- How would we print a tree?
  - Need to recurse both left and right
  - **Traverse the tree!**
    - Most tree problems involve traversing the tree



Traversal trick

- To quickly generate a traversal:
  - Trace a path counterclockwise.
  - As you pass a node on the proper side, process it.
    - pre-order: left side
    - in-order: bottom
    - post-order: right side
- What kind of traversal does a for-each loop in a Set do?



• pre-order:	17	4	1	6	32	24	81
• in-order:	1	4	6	17	24	32	81
• post-order:	1	6	4	24	81	32	17

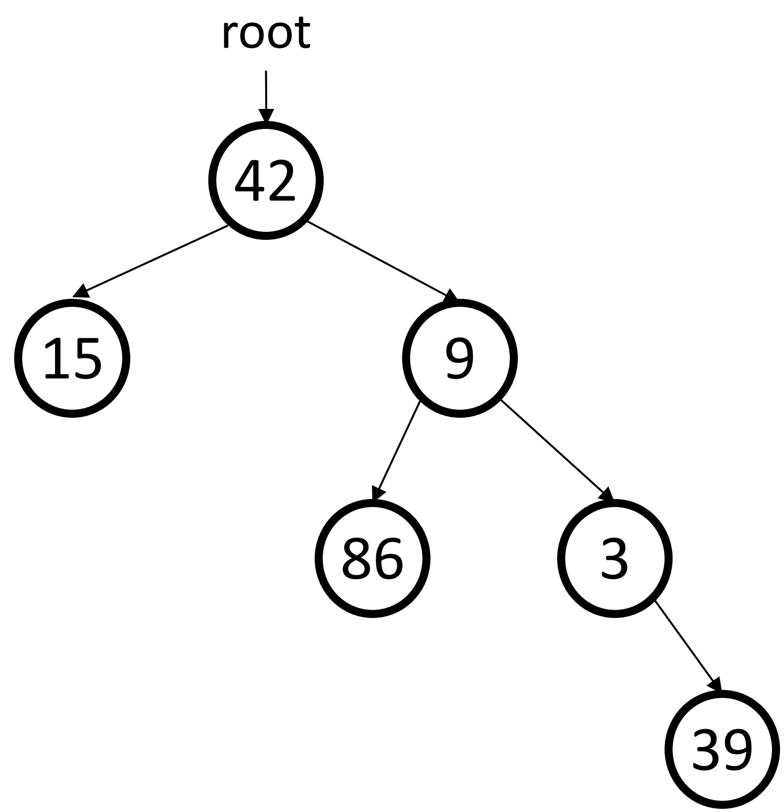
Announcements

- Assignment 5 is due **tomorrow**
- Assignment 6 will be released tomorrow

- Midterm Regrade Requests are open until Monday at 5PM

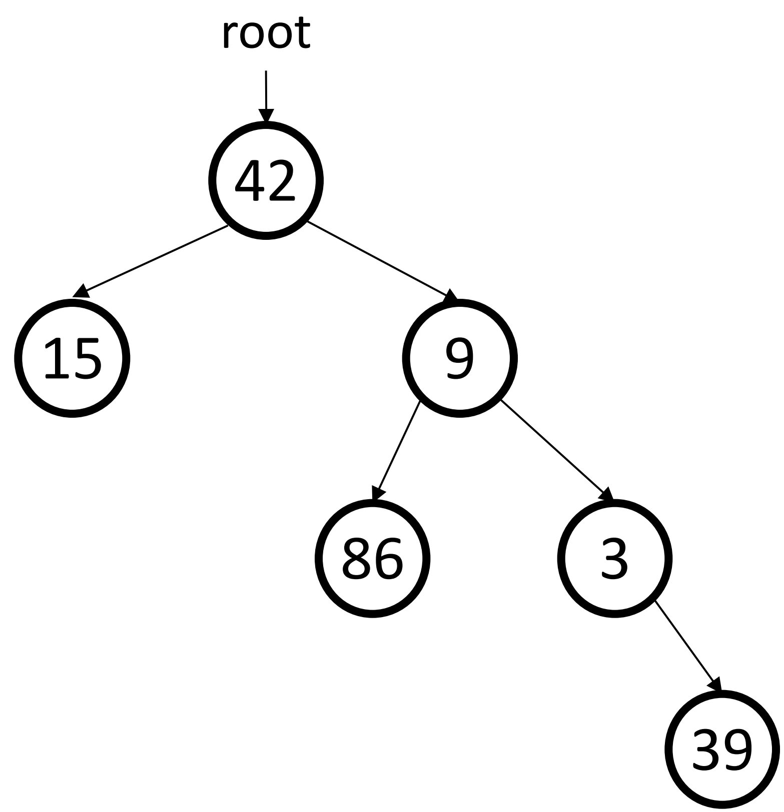
Traversal exercise

- Give pre-, in-, and post-order traversals for the following tree:



Traversal exercise

- Give pre-, in-, and post-order traversals for the following tree:



  - pre: 42 15 9 86 3 39
  - in: 15 42 86 9 3 39
  - post: 15 86 39 3 9 42

print as traversal

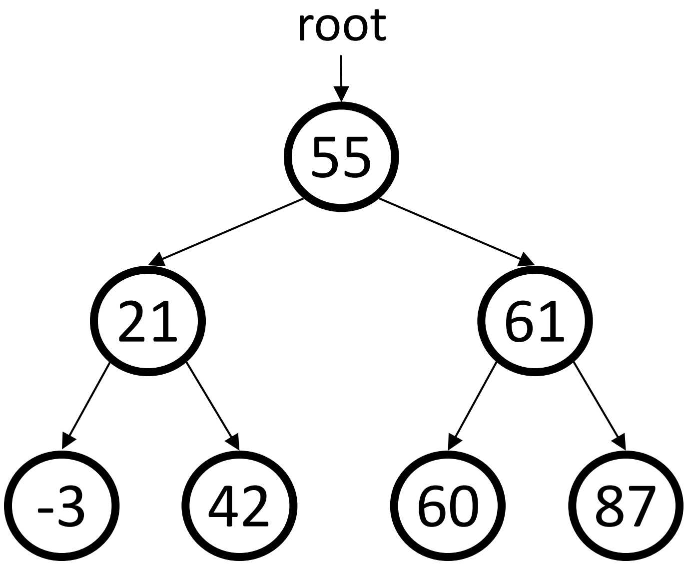
- What happens if I put the following line of code at each of the following locations?

```
cout << node->data << endl;
```

```
void print(TreeNode* node) {
    // (A)
    if (node != nullptr) {
        // (B)
        print(node->left);
        // (C)
        print(node->right);
        // (D)
    }
    // (E)
}
```

Exercise: contains

- Write a function **`contains`** that accepts a tree node pointer as its parameter and searches the tree for a given integer, returning `true` if found and `false` if not.
  - `contains(root, 87)` → `true`
  - `contains(root, 60)` → `true`
  - `contains(root, 63)` → `false`
  - `contains(root, 44)` → `false`

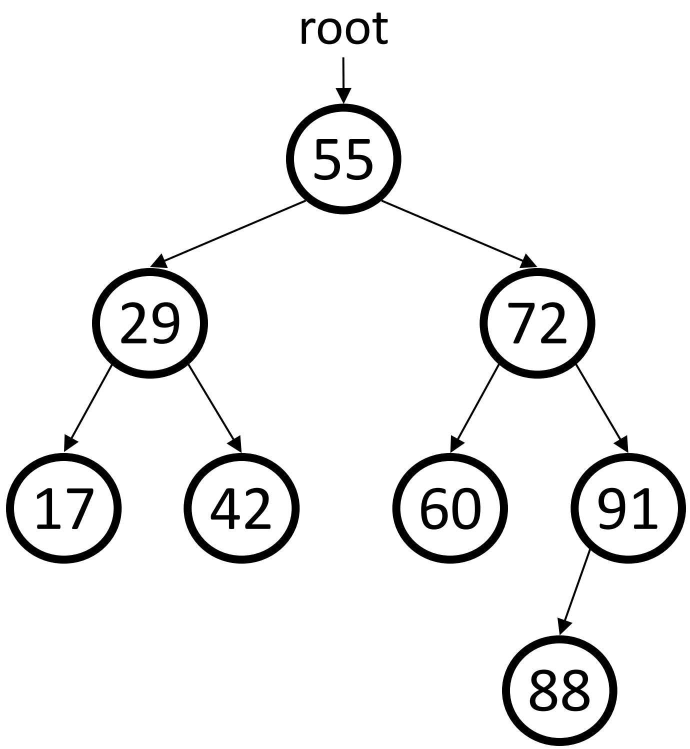


contains solution

```
// Returns whether this BST contains the given integer.
// Assumes that the given tree is in valid BST order.
bool contains(TreeNode* node, int value) {
    if (node == nullptr) {
        return false;   // base case: not found here
    } else if (node->data == value) {
        return true;    // base case: found here
    } else if (node->data > value) {
        return contains(node->left, value);
    } else {            // root->data < value
        return contains(node->right, value);
    }
}
```

getMin/getMax

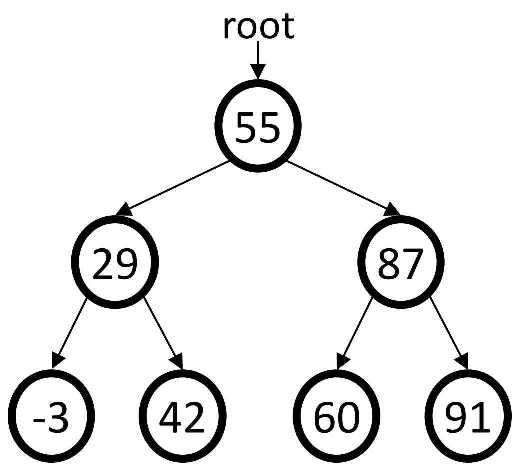
- Sorted arrays can find the smallest or largest element in O(1) time (how?)
- How could we get the same values in a binary search tree?



getMin/Max solution

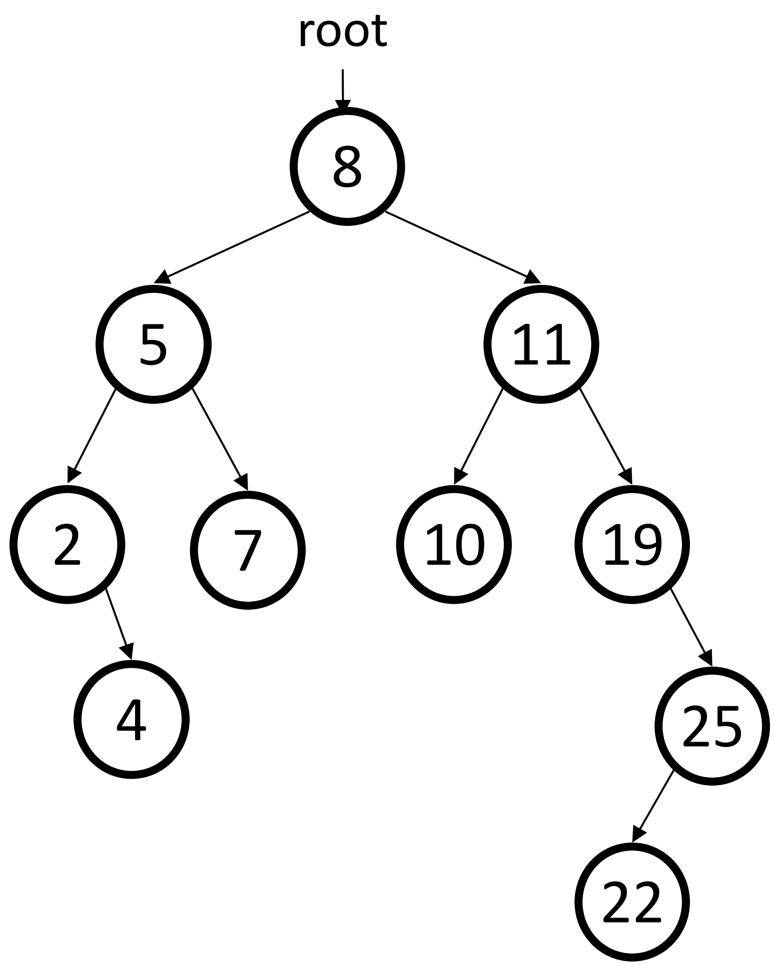
```
// Returns the minimum/maximum value from this BST.
// Assumes that the tree is a nonempty valid BST.
int getMin(TreeNode* root) {
    if (root->left == nullptr) {
        return root->data;
    } else {
        return getMin(root->left);
    }
}

int getMax(TreeNode* root) {
    if (root->left == nullptr) {
        return root->data;
    } else {
        return getMax(root->left);
    }
}
```



Adding to a BST

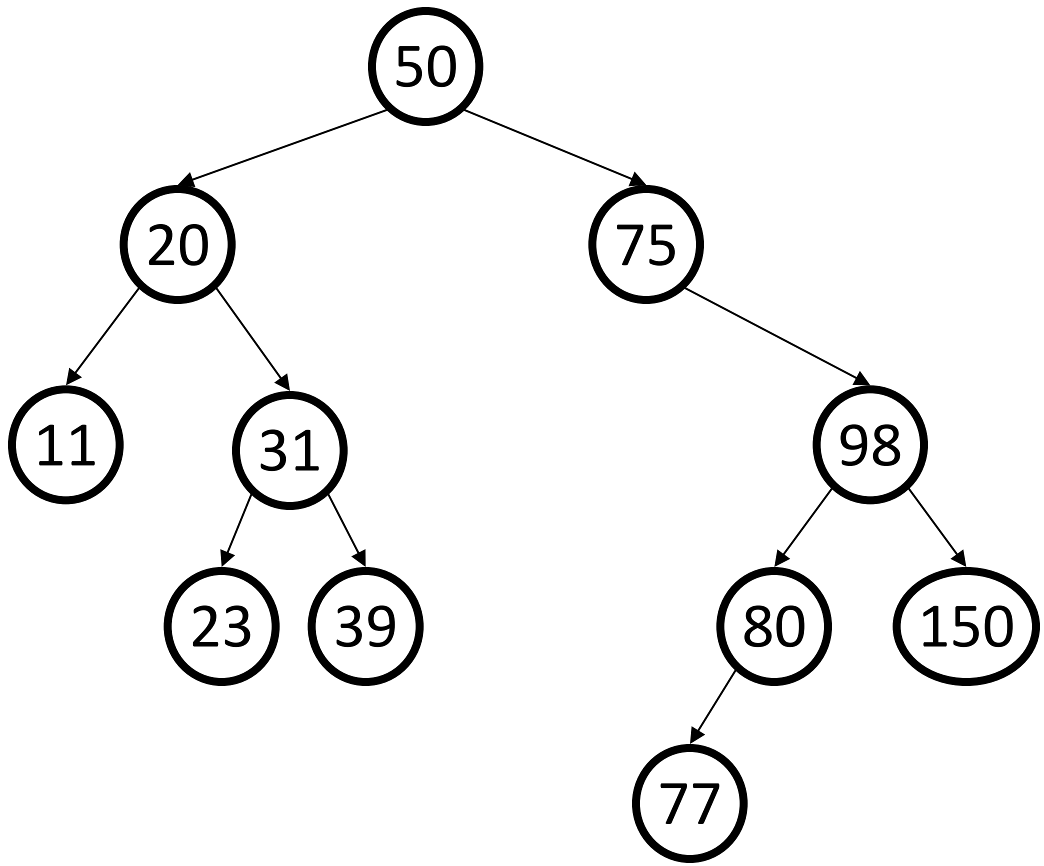
- Suppose we want to add new values to the BST below.
  - Where should the value 14 be added?
  - Where should 3 be added? 7?
  - If the tree is empty, where should a new value be added?



Adding exercise

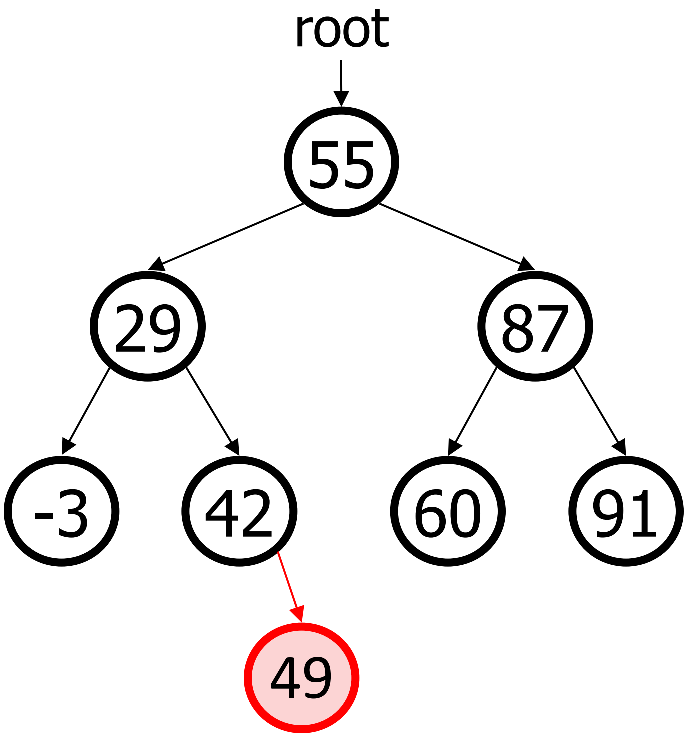
- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77



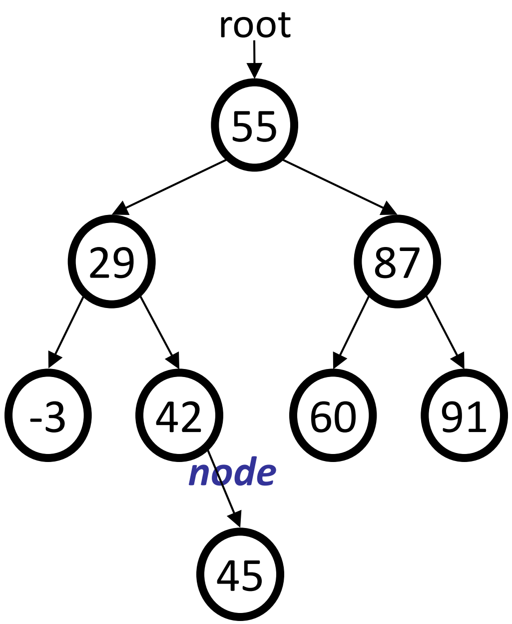
Exercise: add

- Write a function **add** that adds a given integer value to the BST.
  - Add the new value in the proper place to maintain BST ordering.
    - `tree.add(root, 49);`



Add Solution

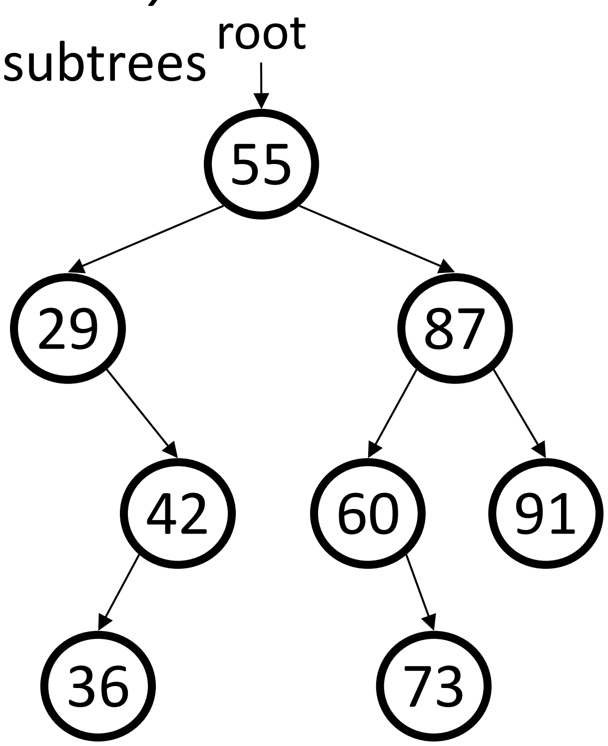
```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
        node = new TreeNode(value);
    } else if (node->data > value) {
        add(node->left, value);
    } else if (node->data < value) {
        add(node->right, value);
    }
}
```



- Must pass the current node *by reference* for changes to be seen.

Free Tree

- To avoid leaking memory when discarding a tree, we must free the memory for every node.
  - Like most tree problems, often written *recursively*
  - must free the node itself, and its left/right subtrees
  - this is another *traversal* of the tree
    - should it be pre-, in-, or post-order?

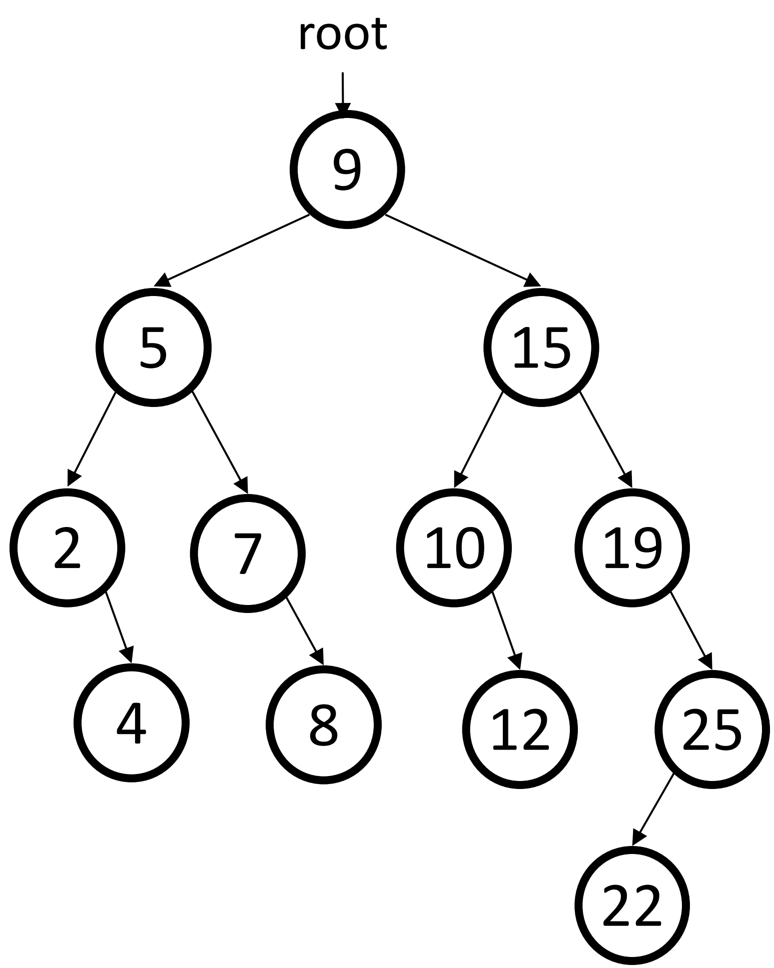


Free tree solution

```
void freeTree(TreeNode*& node) {
    if (node == nullptr) {
        return;
    }
    freeTree(node->left);
    freeTree(node->right);
    delete node;
}
```

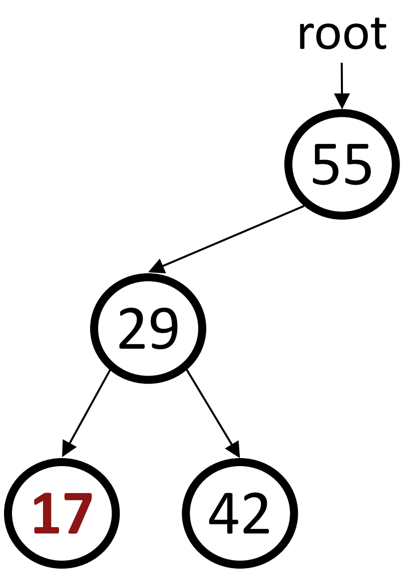
Removing from a BST

- Suppose we want to **remove** values from the BST below.
  - Removing a leaf like 4 or 22 is easy.
  - What about removing 2? 19?
  - How can you remove a node with two large subtrees under it, such as 15 or 9?

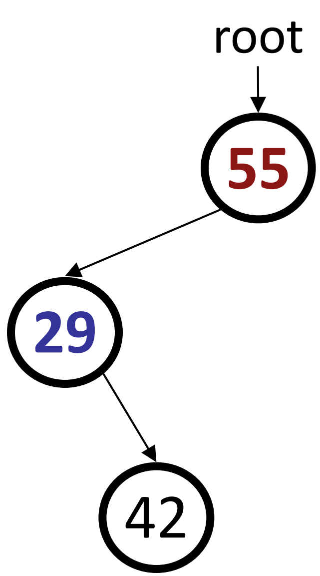


Cases for removal

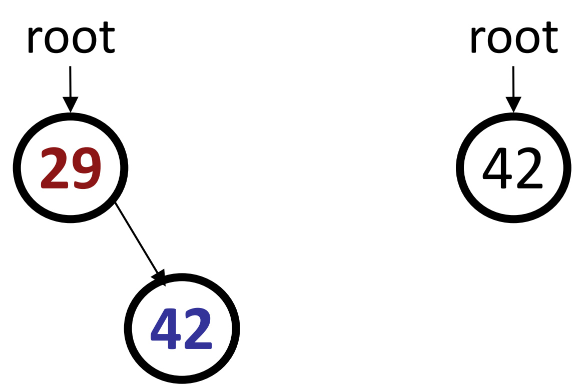
1. a **leaf**: Replace with `nullptr`
2. a node with a **left child only**: Replace with left child
3. a node with a **right child only**: Replace with right child



`remove(root, 17);`



`remove(root, 55);`



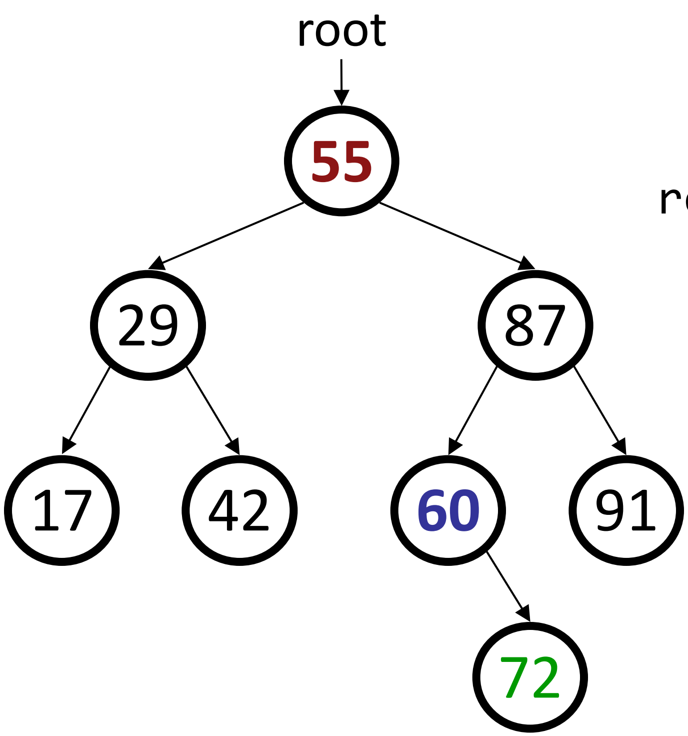
`remove(root, 29);`

Cases for removal

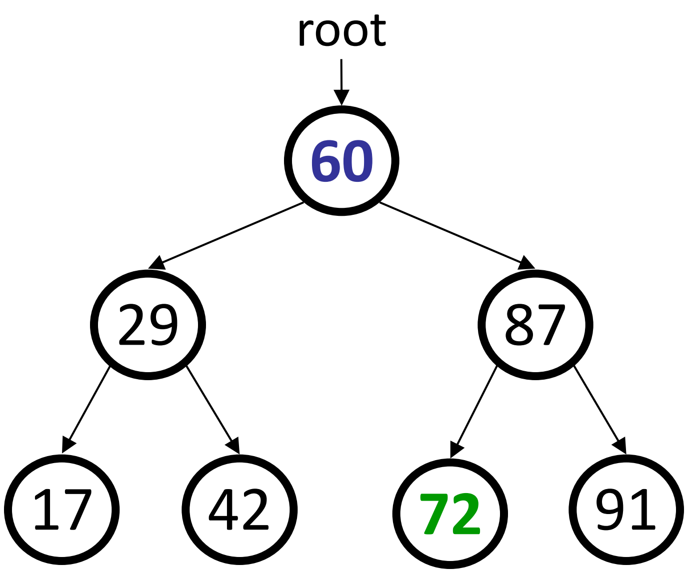
4. a node with **both** children:

   replace with **min from right**

   (replacing with **max from left** would also work)

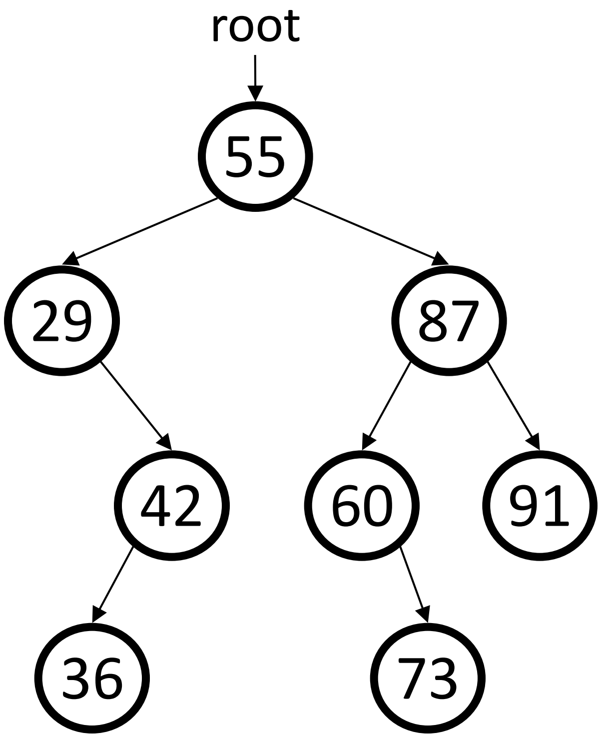



```
remove(root, 55);
```



Exercise: remove

- Add a function **`remove`** that accepts a root pointer and removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.
  - `remove(root, 73);`
  - `remove(root, 29);`
  - `remove(root, 87);`
  - `remove(root, 55);`



remove solution

```
// Removes the given value from this BST, if it exists.
// Assumes that the given tree is in valid BST order.
void remove(TreeNode*& node, int value) {
    if (node == nullptr) {
        return;
    } else if (value < node->data) {
        remove(node->left, value);    // too small; go left
    } else if (value > node->data) {
        remove(node->right, value);   // too big; go right
    } else {
        // value == node->data; remove this node!
        // (continued on next slide)
        ...
```

remove solution

```
        // value == node->data; remove this node!
        if (node->right == nullptr) {
            // case 1 or 2: no R child; replace w/ left
            TreeNode* trash = node;
            node = node->left;
            delete trash;
        } else if (node->left == nullptr) {
            // case 3: no L child; replace w/ right
            TreeNode* trash = node;
            node = node->right;
            delete trash;
        } else {
            // case 4: L+R both; replace w/ min from right
            int min = getMin(node->right);
            remove(node->right, min);
            node->data = min;
        }
    }
}
```

Overflow

- We saw how to add to a binary search tree. Does it matter what order we add in?
  - Try adding: 50, 20, 75, 98, 80, 31, 150
  - Now add the same numbers but in sorted order: 20, 31, 50, 75, 80, 98, 150