CS 106X, Lecture 19
Trees

reading:

Programming Abstractions in C++, Chapters 16.1-16.4

Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

小结：

1. tree = ptr + recursion综合
2. 所有tree的问题：都与traverse order有关（思考的角度方向）
3. 不同节点的分类cases：nullptr（base case）、leaf（2边都没children）、“single” internal vertex（只有1边有child，出现最大最小值）、internal vertex（2边都有child）
4. tree branch arrow：可看作是一个指针
5. remove节点操作：分类：single vertex（调整arrow指向的终点）；internal vertex：replace it with right_min / left_max
第2点的补充：可以参考browser2的inrange函数如何从树种取出从小到大的元素，inorder traverse
6. 在heap中创建数据结构初始化的时候，即使是empty了记得也要加上nullptr防止dangling ptr；删除清空的时候也是如此，删完的指针设置成nullptr
7. Trie: 每条指针代表一个字母，节点的位置决定一个单词（参考leetcode一道模版题）
8. 平衡树的构建：hw6的displaying text：
https://www.baeldung.com/cs/balanced-bst-from-sorted-list

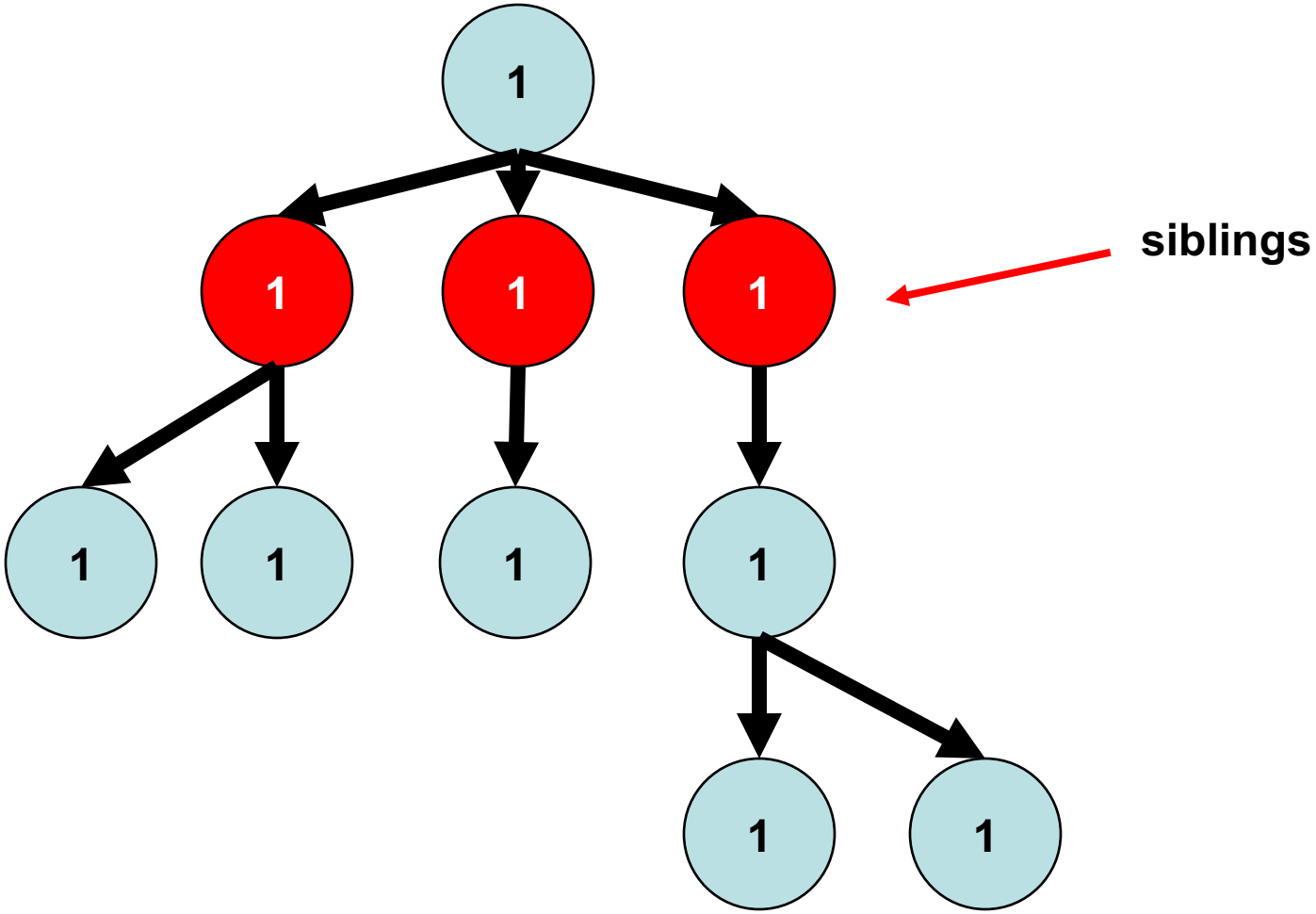
Learning Goals

- Understand how trees use pointers to represent data in useful ways
- Understand the structure of binary search trees and how to traverse/search/update them

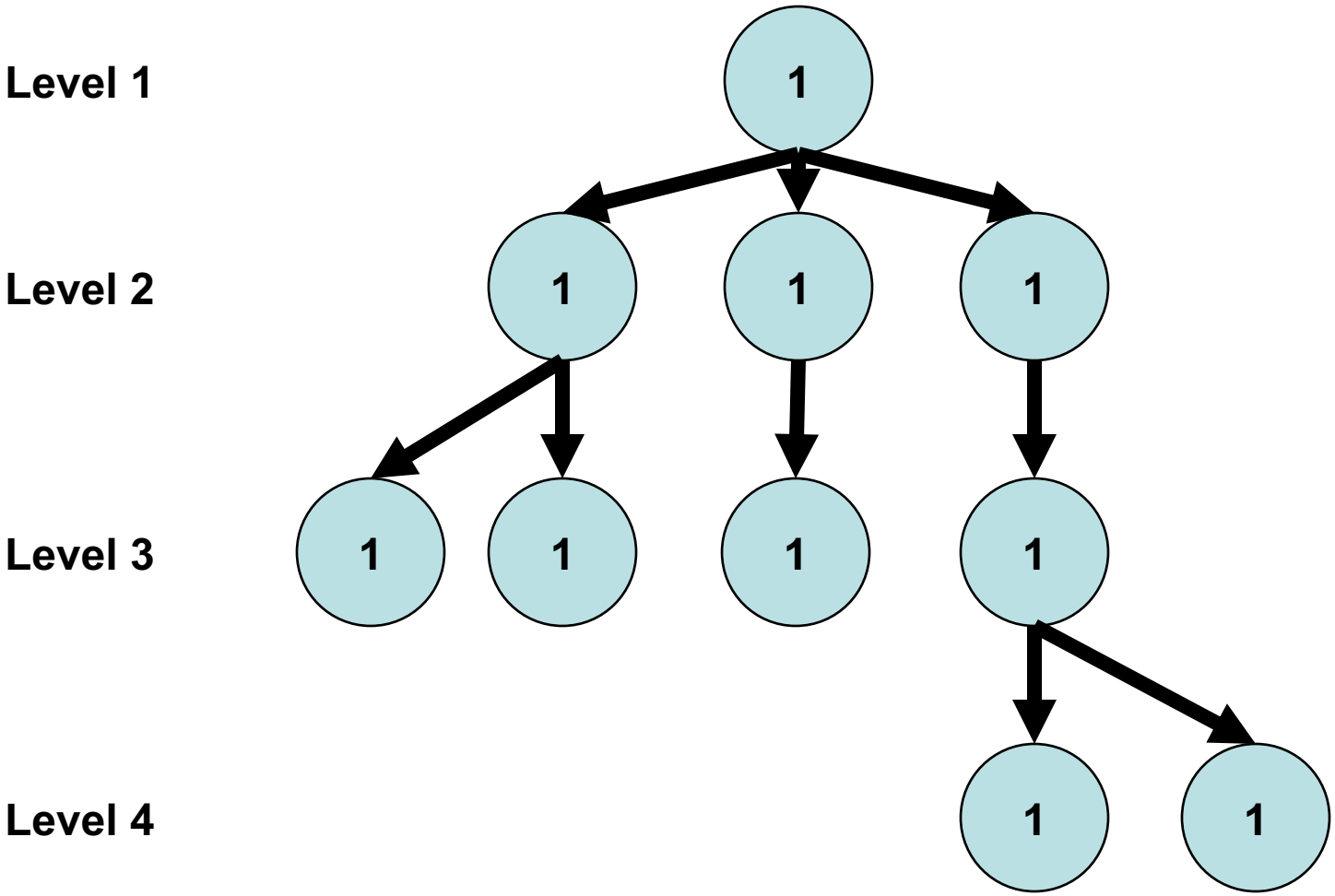
Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

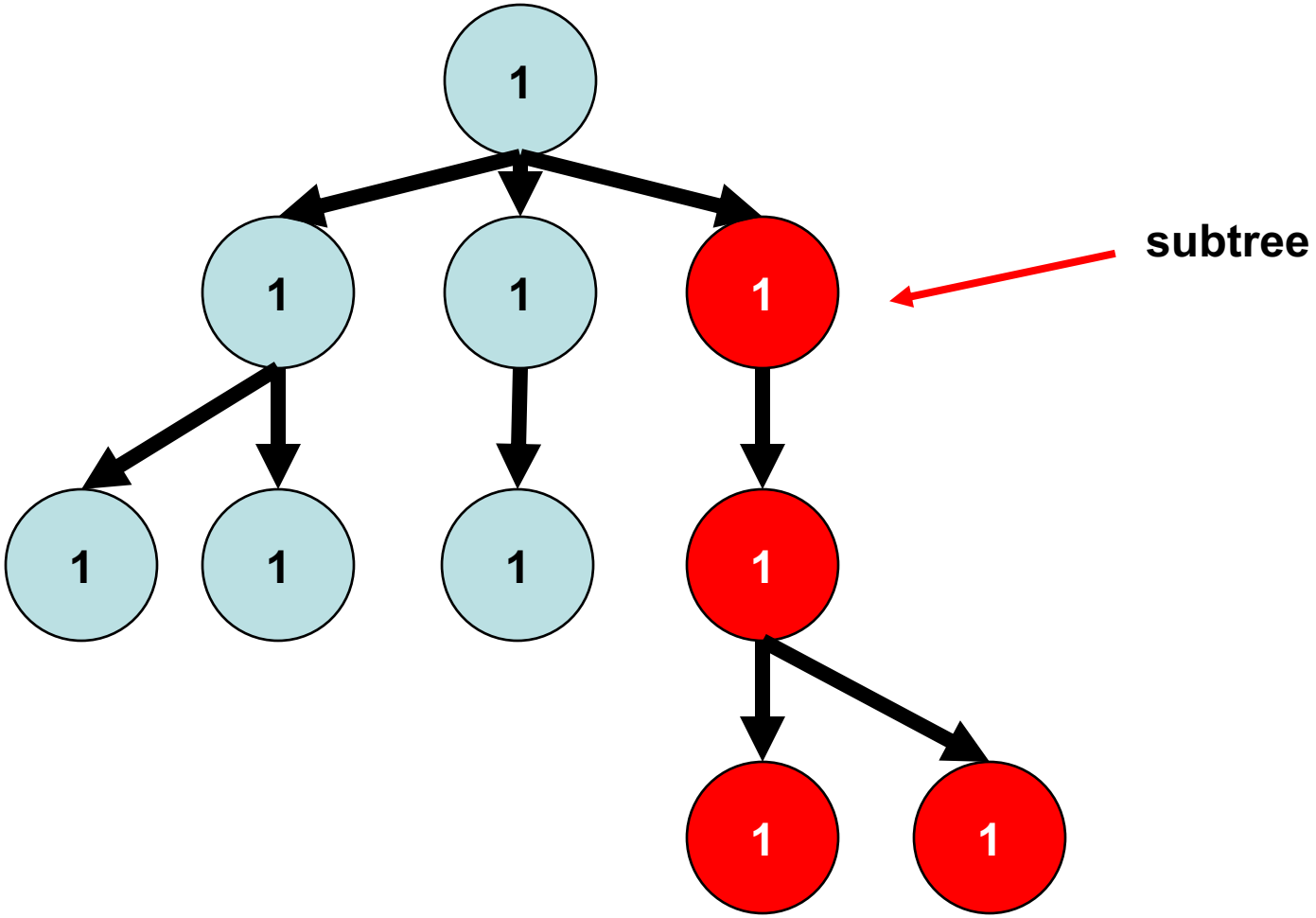
Trees



Trees

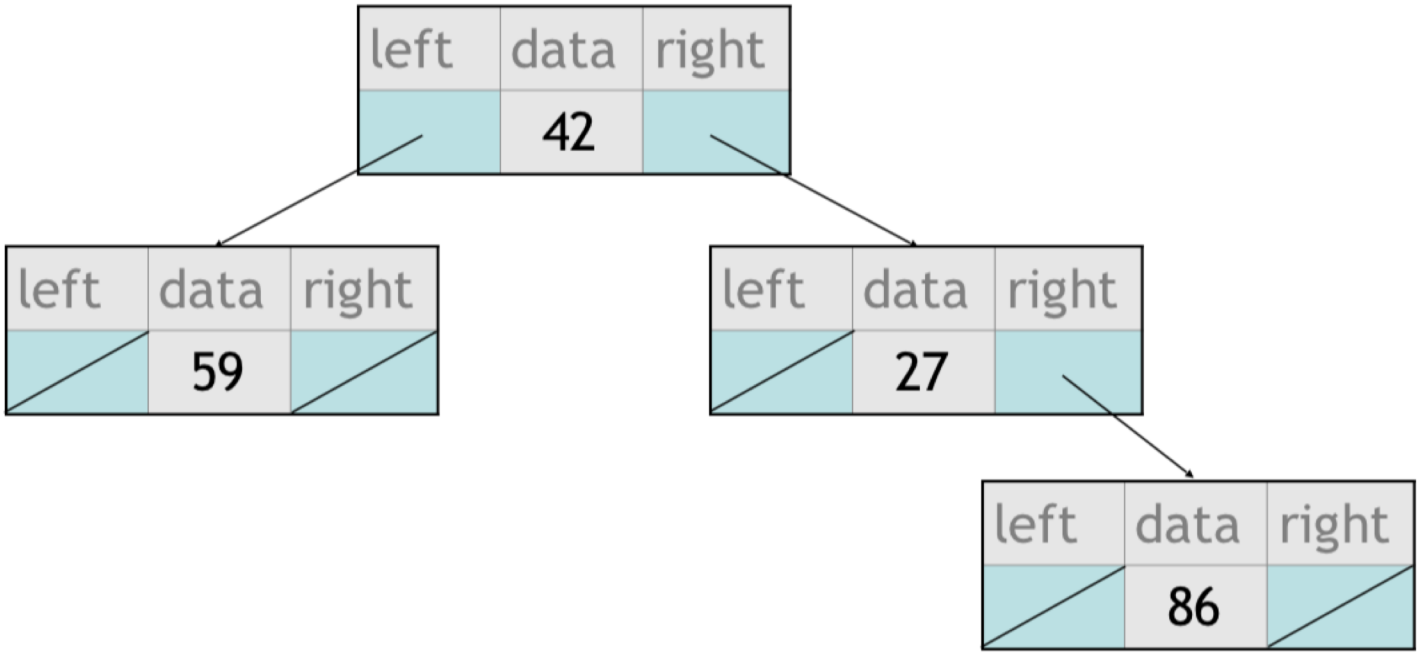


Trees



TreeNode

- A basic **tree node object** stores data and pointers to left/ right
  - Multiple nodes can be linked together into a larger tree



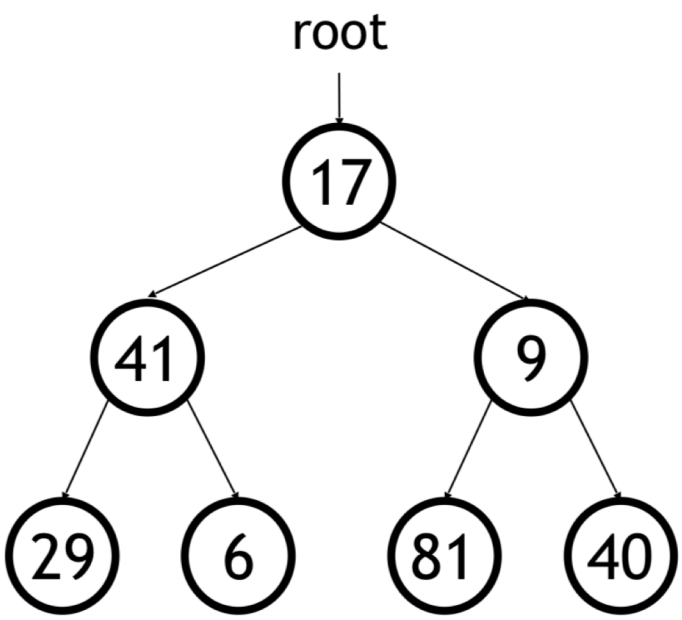
Common Tree Operations

- print
- height
- size
- contains
- deleteTree

print

- Write a function named **`print`** that accepts a tree node pointer as its parameter and prints the elements of that tree, one per line.
  - A node's left subtree should be printed before it, and its right subtree should be printed after it.
  - Example: `print(root);`

```
29
41
6
17
81
9
40
```



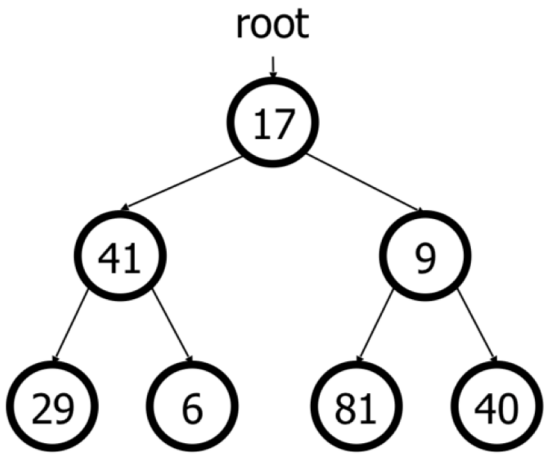
print

```
void print(TreeNode* node) {
    // (base case is implicitly to do nothing on NULL)
    if (node != nullptr) {
        // recursive case: print left, center, right
        print(node->left);
        cout << node->data << endl;
        print(node->right);
    }
}
```

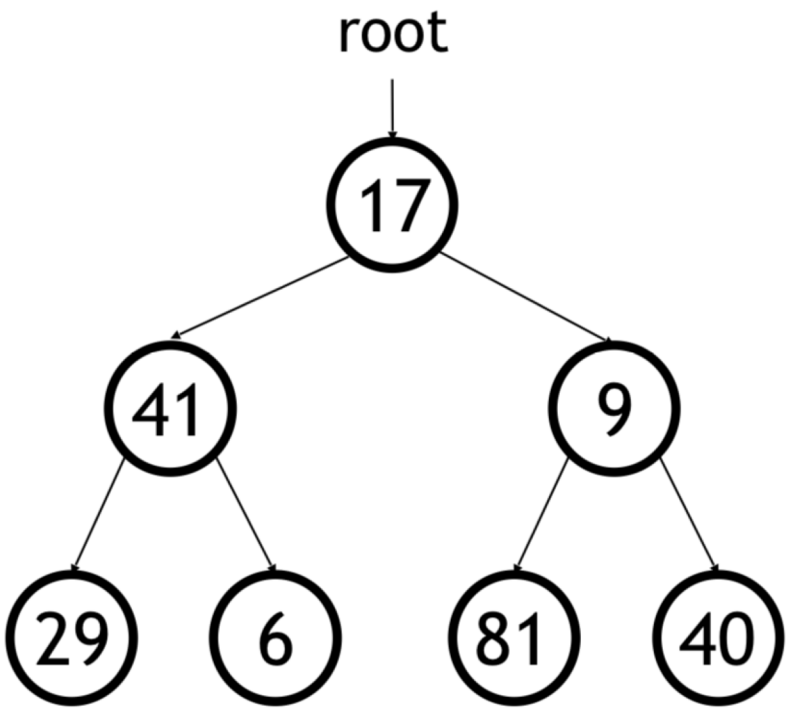
Traversal

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node



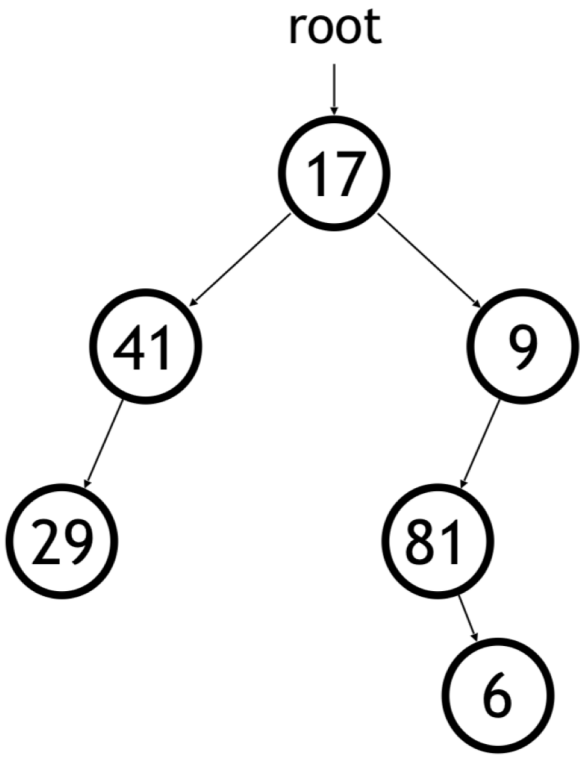
Traversal



- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

size

- Write a function named **`size`** that accepts a tree node pointer as its parameter and returns the number of elements of that tree.
  - An empty/null tree is defined as having a size of 0.
  - Example: `size(root)` returns `6`

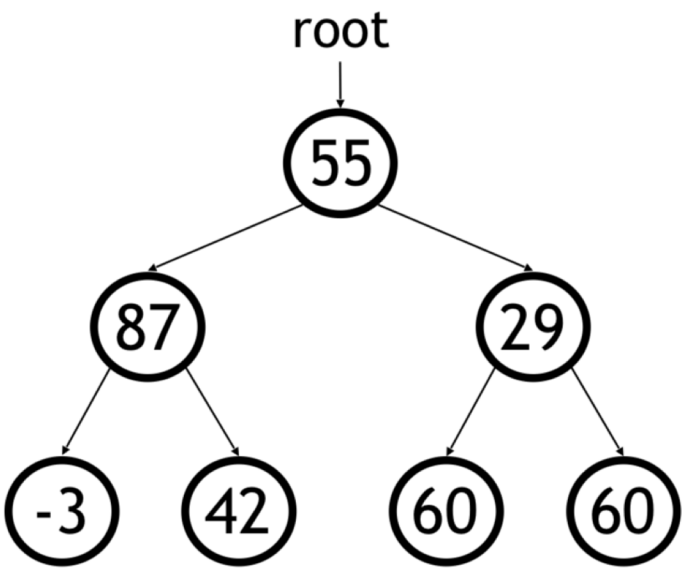


size

```
int size(TreeNode* node) {
    if (node == nullptr) {
        // base case: empty tree
        return 0;
    } else {
        // recursive case: non-NULL node
        // with possible children
        return 1 + size(node->left) + size(node->right);
    }
}
```

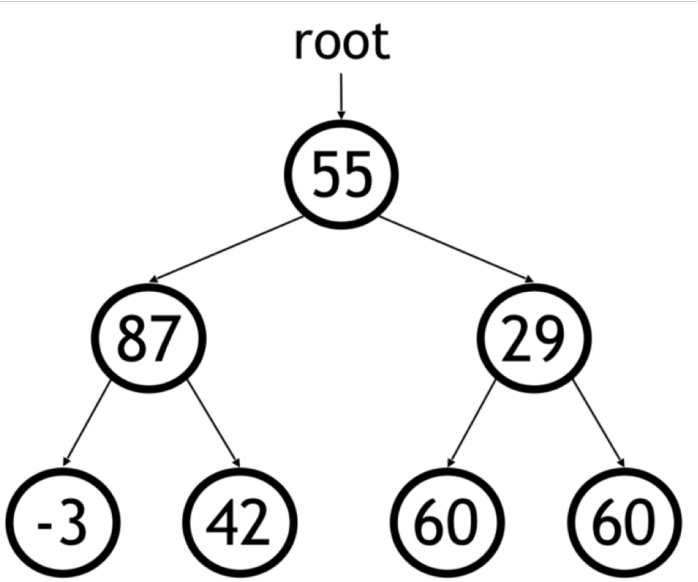
contains

- 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`



height

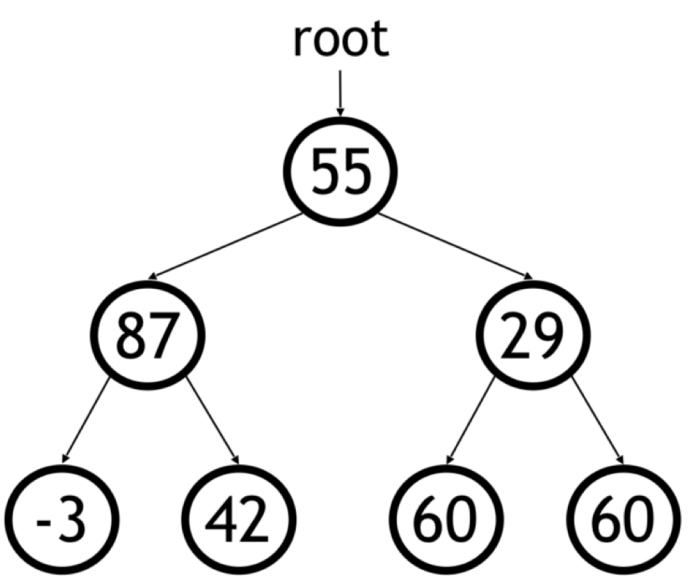
- Write a function **height** that accepts a tree node pointer as its parameter and returns the height of the tree.
- Height is defined as the longest path from the root to any node, in # nodes.



E.g. the height of this tree is **3**.

deleteTree

- Write a function **deleteTree** that accepts a tree node pointer as its parameter and frees all memory associated with the given tree.

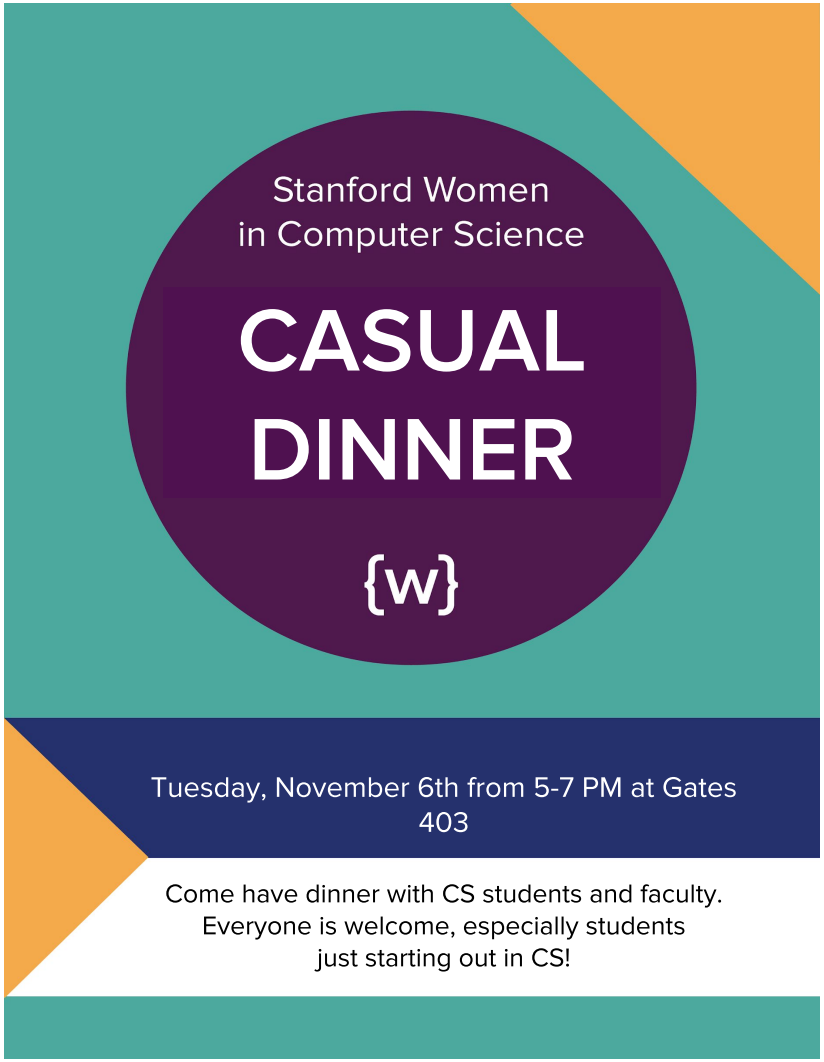


Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

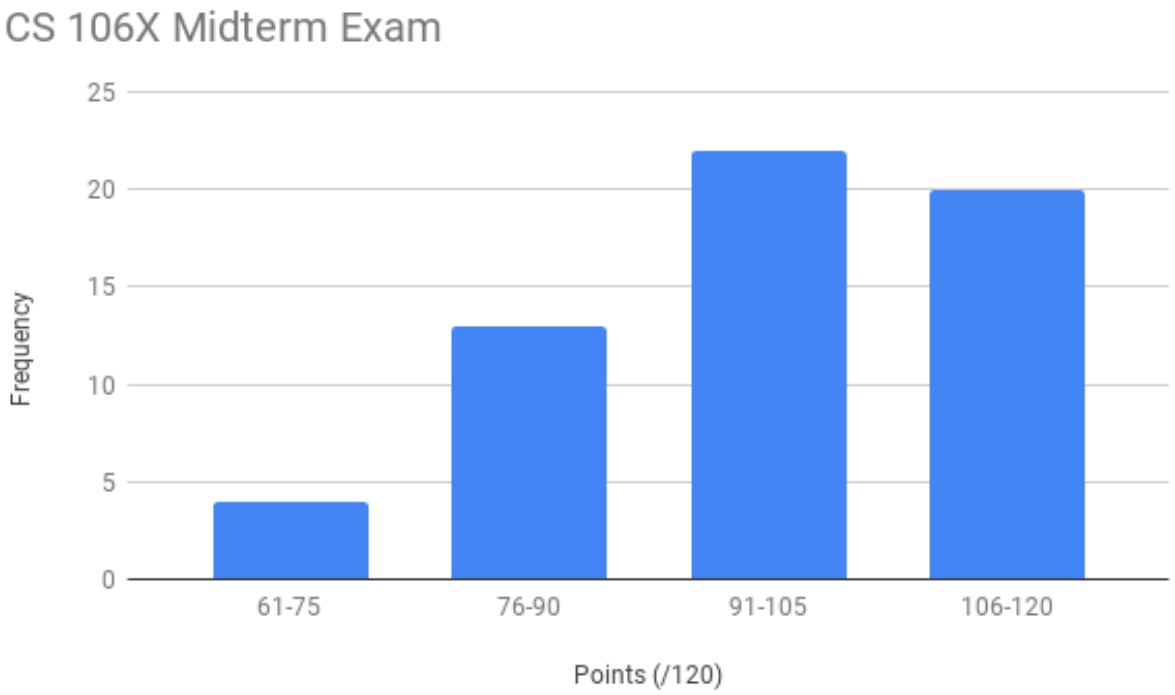
Announcements

- WiCS Casual Dinner **Tues. 11/6 5-7 in Gates 403**! Join me!



Announcements

- Midterms have been graded, and will be returned after class
- Visit gradescope.com (you should receive an email) to view your exam and score
- Regrades accepted until next Monday at 1:30PM

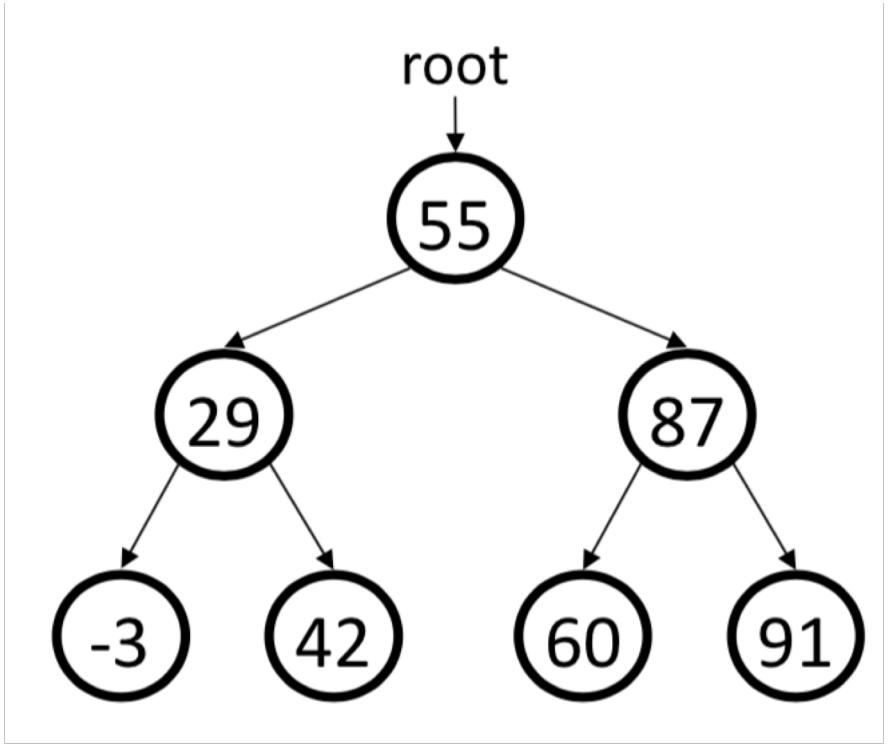


Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

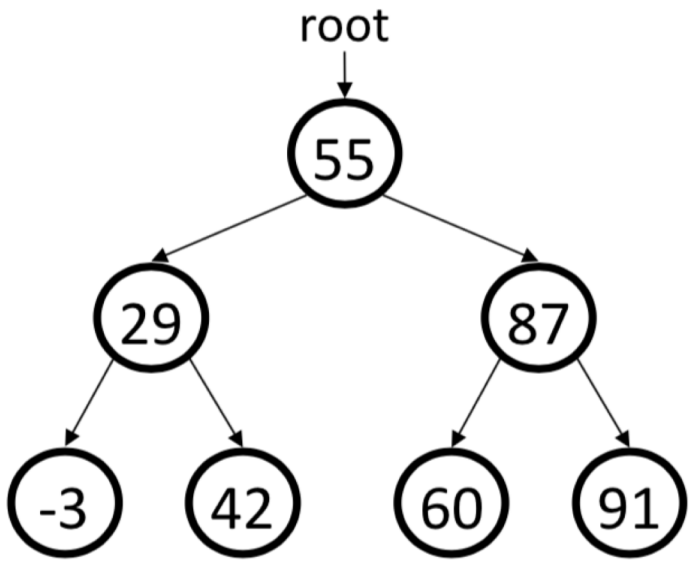
Binary Search Trees

- BSTs store their elements in **sorted order**, which is helpful for searching/sorting tasks.
- Used to implement the Stanford Set



Binary Search Trees

- **binary search tree** ("BST"): a binary tree where each non-empty node *R* has the following properties:
  - every element of *R*'s left subtree contains data less than *R*'s data,
  - every element of *R*'s right subtree contains data greater than *R*'s,
  - *R*'s left and right subtrees are also binary search trees.



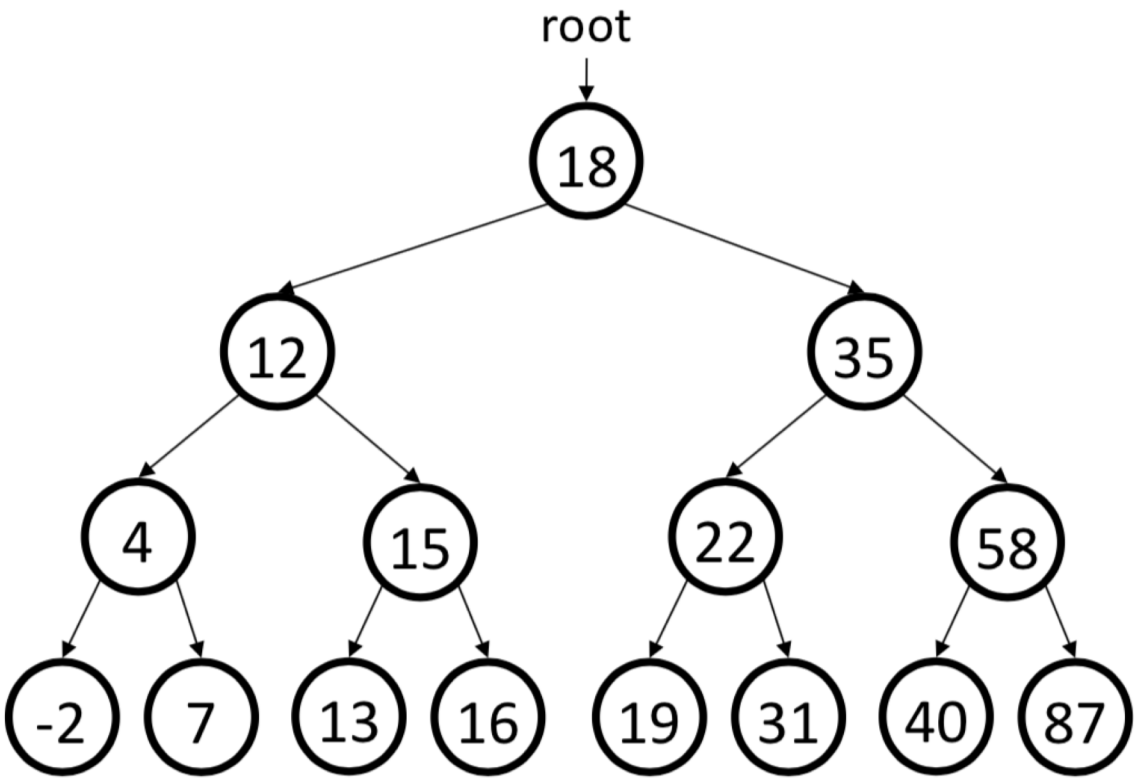
Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

Searching a BST

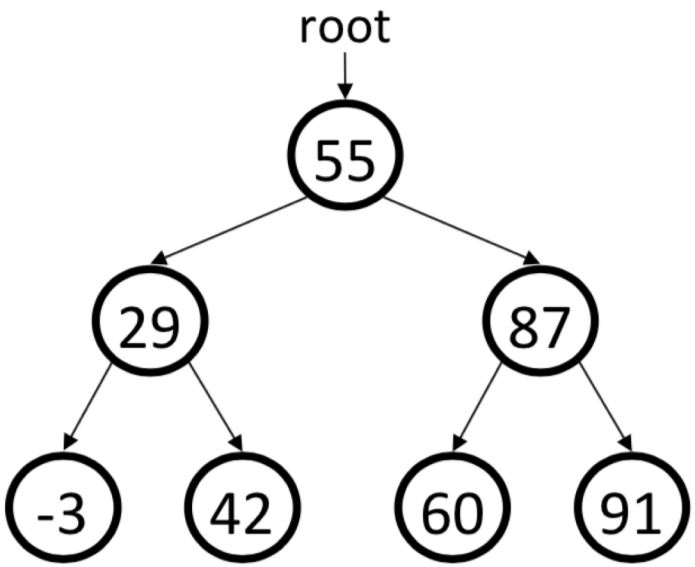
- Describe an algorithm for searching a binary search tree.
  - Try searching for the value 31, then 6.

- What is the maximum number of nodes you would need to examine to perform any search?



Exercise: contains

- Modify our **`contains`** function to take advantage of the BST's ordering and structure.
  - `contains(root, 29)` → `true`
  - `contains(root, 55)` → `true`
  - `contains(root, 63)` → `false`
  - `contains(root, 35)` → `false`



Exercise: contains

```
/*
 * Returns true if the given tree contains the given value.
 */
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 {
        // recursive case: search left/right subtrees
        return contains(node->left, value) ||
               contains(node->right, value);
    }
}
```

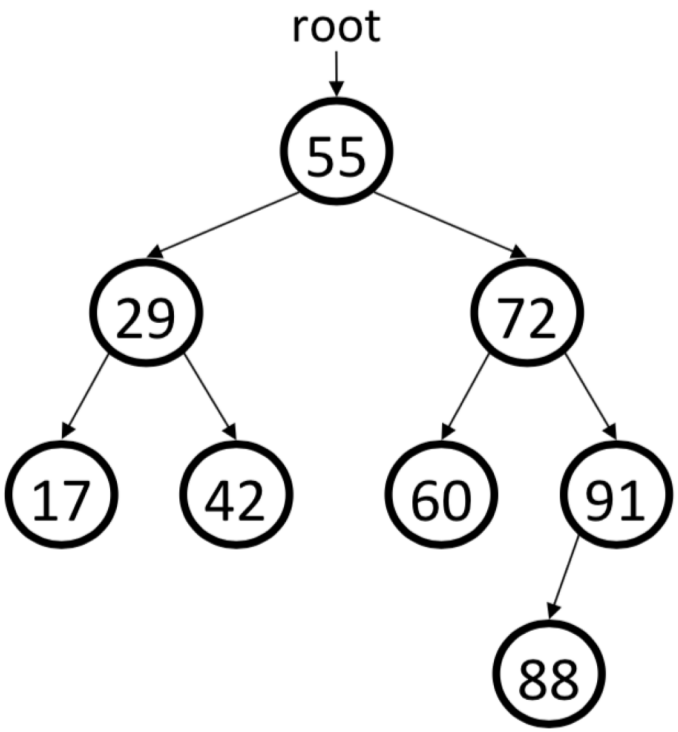
Exercise: contains

```
/*
 * Returns true if the given tree contains the given value.
 */
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 {
        return contains(node->right, value);
    }
}
```

Exercise: getMin/getMax

- Write functions **`getMin`** and **`getMax`** that accept a node pointer and return the minimum and maximum integer value from the tree. Assume that the tree is a valid non-empty binary search tree.

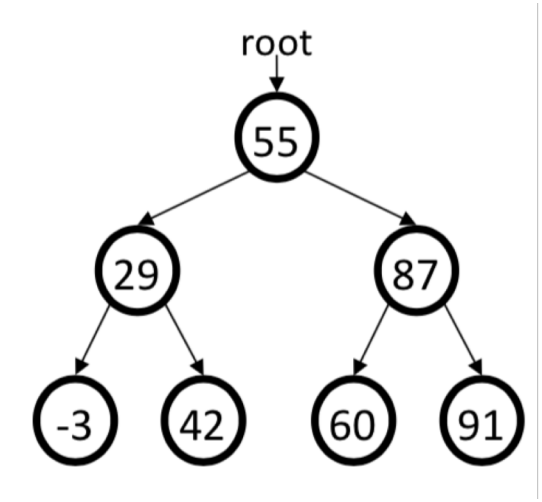
```
int min = getMin(root);   // 17
int max = getMax(root);   // 91
```



Exercise: getMin/getMax

```
// 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;
    }
}

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

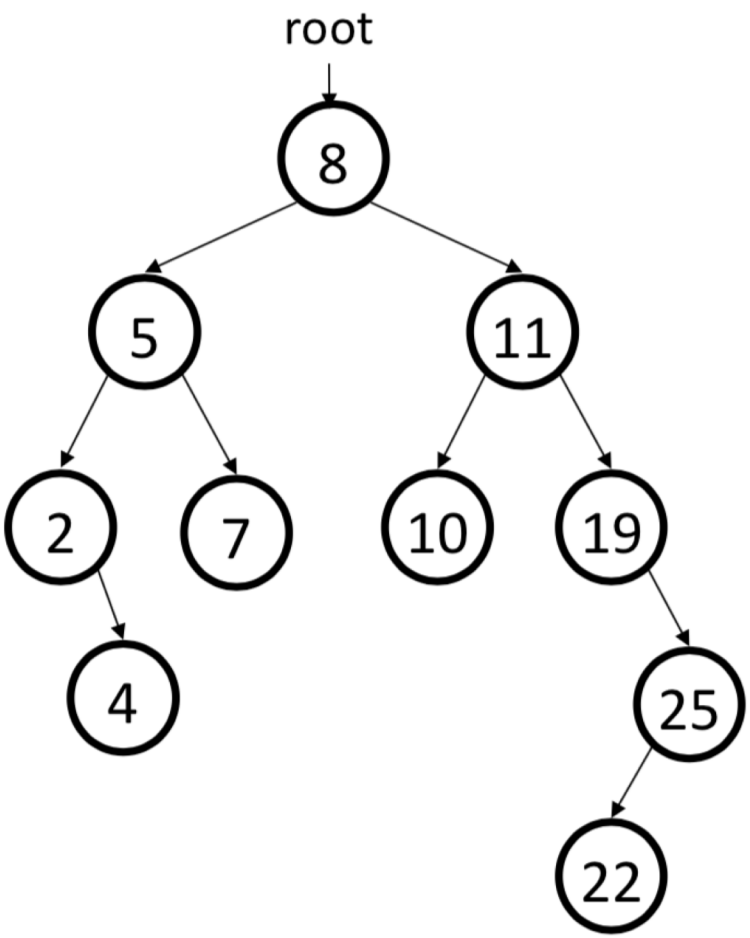


Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

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?
- What is the general algorithm?



Adding to a BST

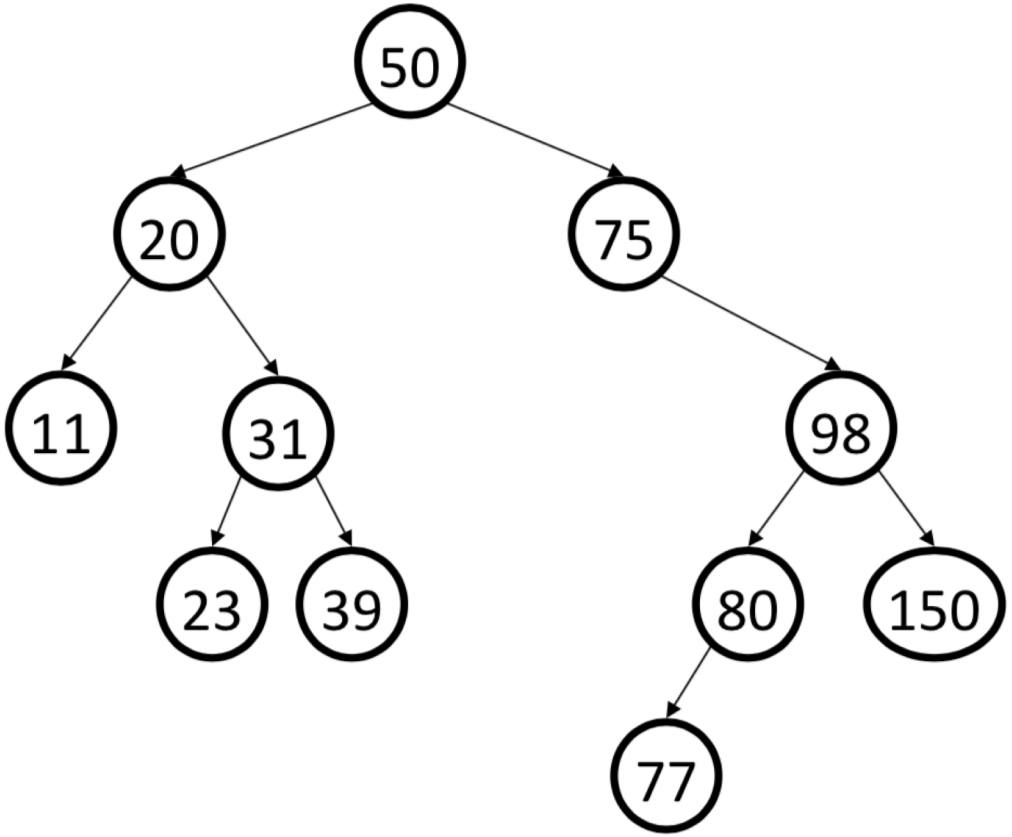
- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77

Adding to a BST

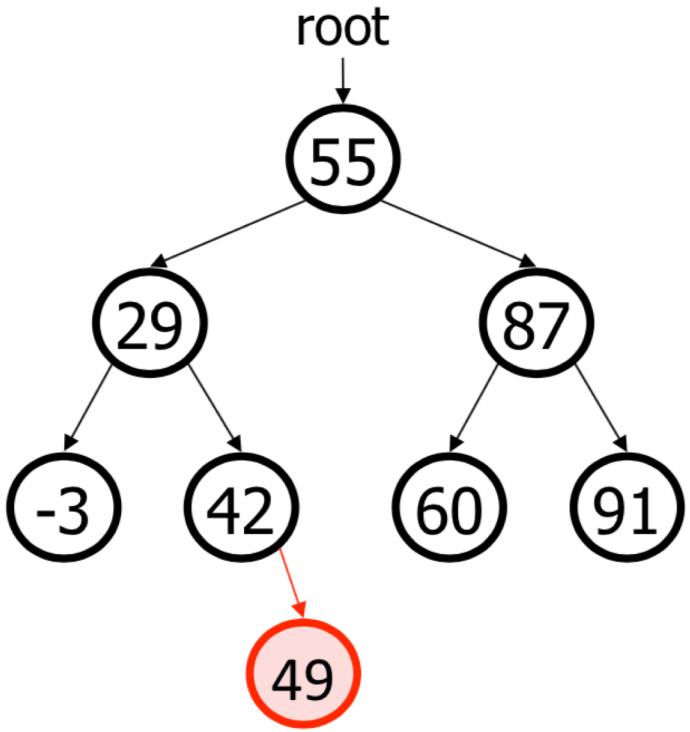
- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77



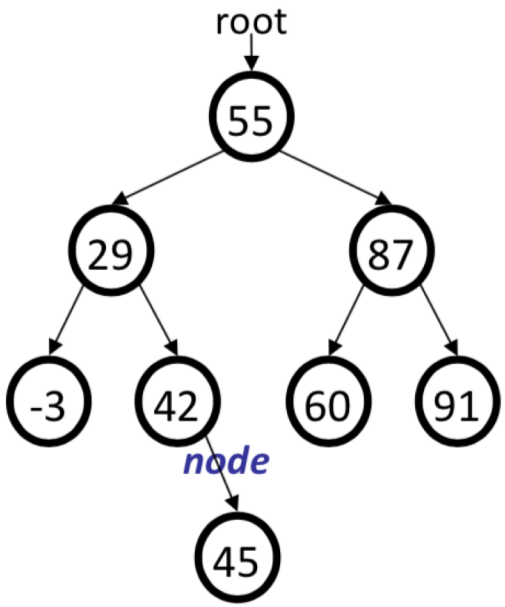
Exercise: add

- 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);`



Exercise: add

```
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);
    }
}
```



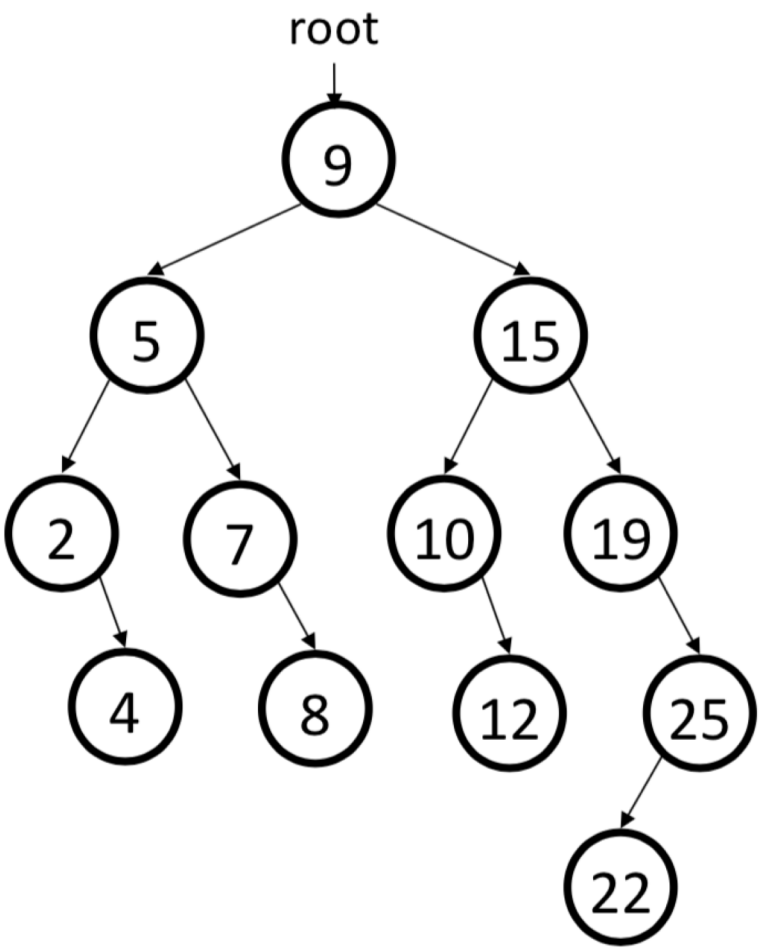
- Must pass the current node *by reference* for changes to be seen.

Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

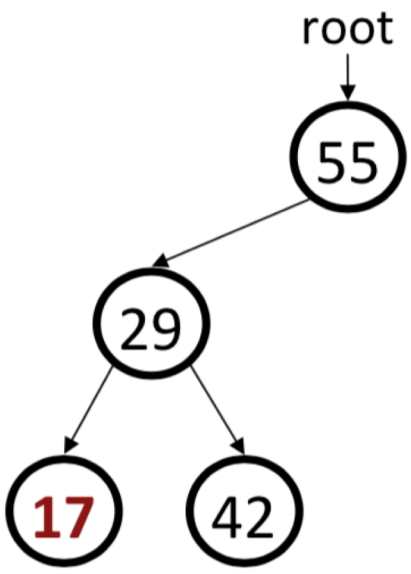
Removing from a BST

- 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?
- What is the general algorithm?

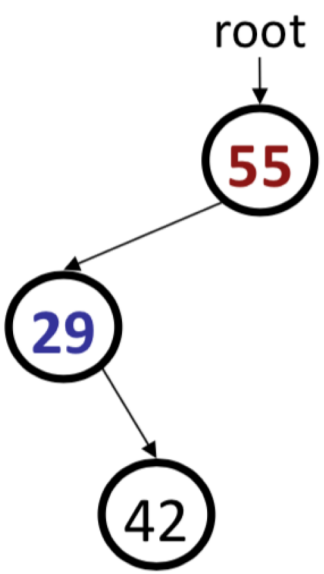


Removing from a BST

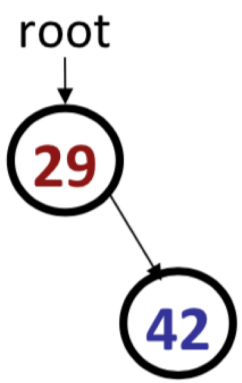
1. a **leaf**:
2. a node with a **left child only**:
3. a node with a **right child only**:



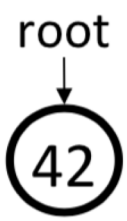
`remove(root, 17);`



`remove(root, 55);`

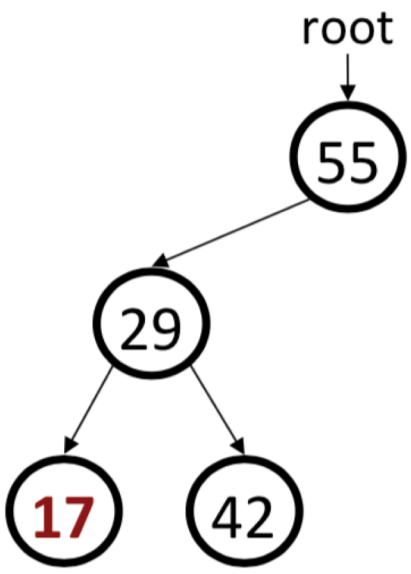


`remove(root, 29);`

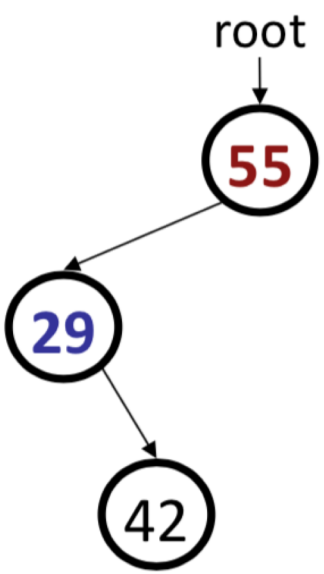


Removing from a BST

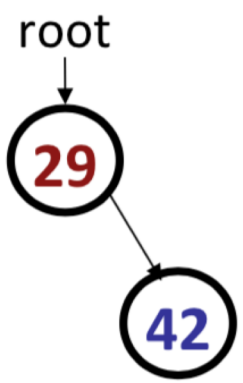
1. a **leaf**: Replace with `nullptr`
2. a node with a **left child only**: Replace with left child
3. a node with a **right child only**: Replace with right child



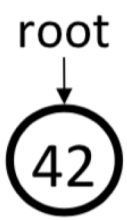
`remove(root, 17);`



`remove(root, 55);`

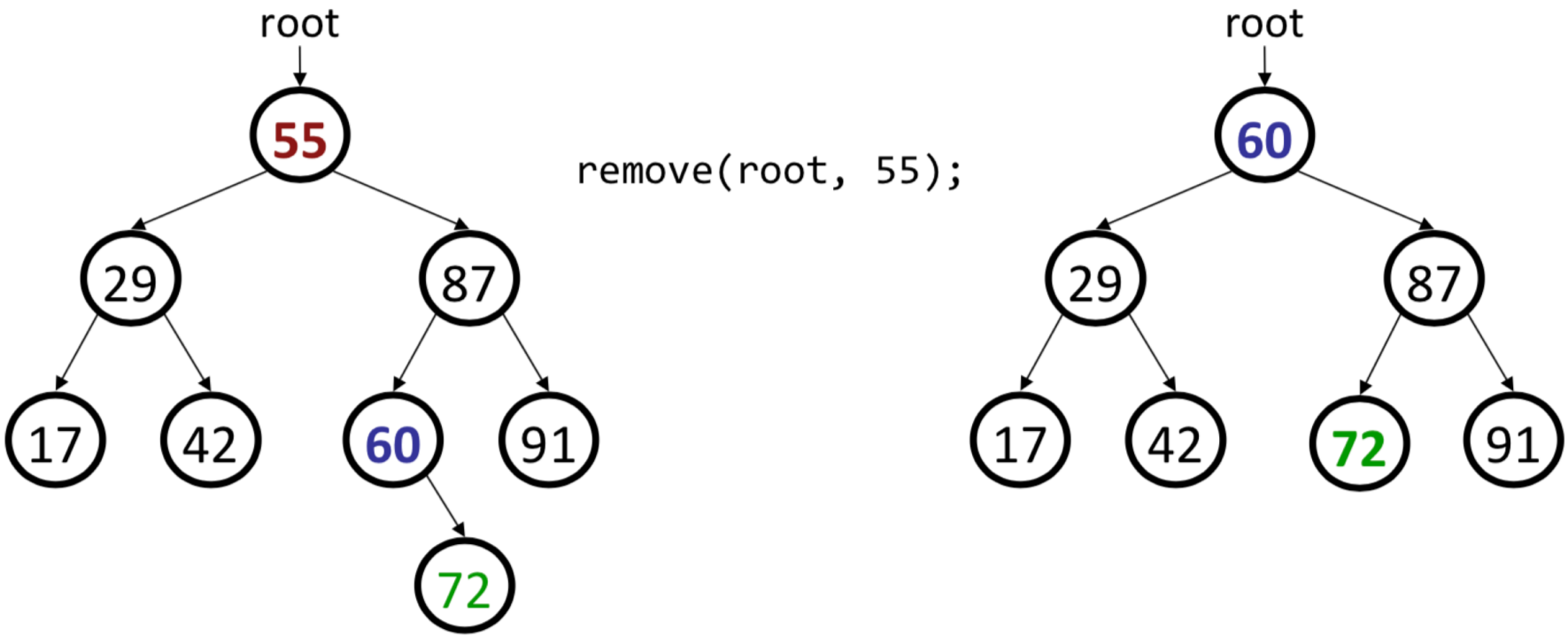


`remove(root, 29);`



Removing from a BST

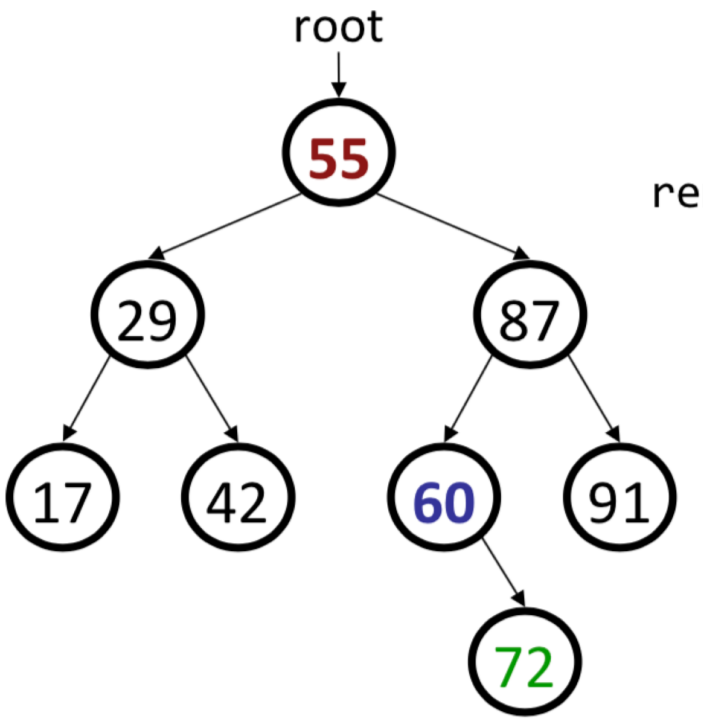
4. a node with **both** children:



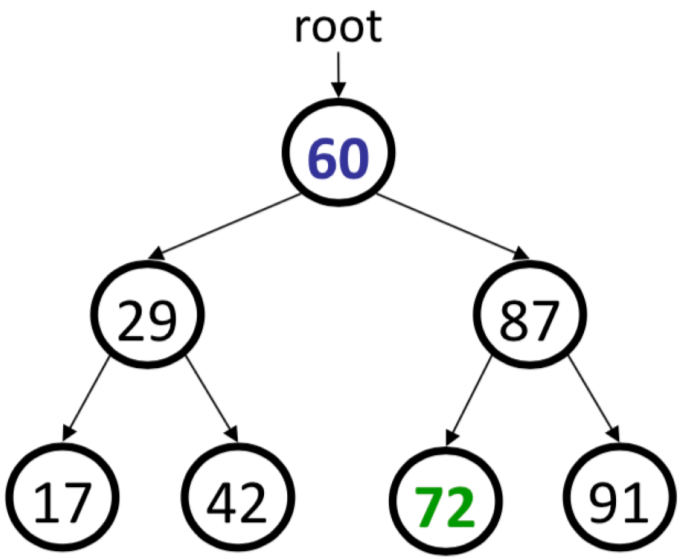
Removing from a BST

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

replace with **min from right**
(replacing with **max from left** would also work)

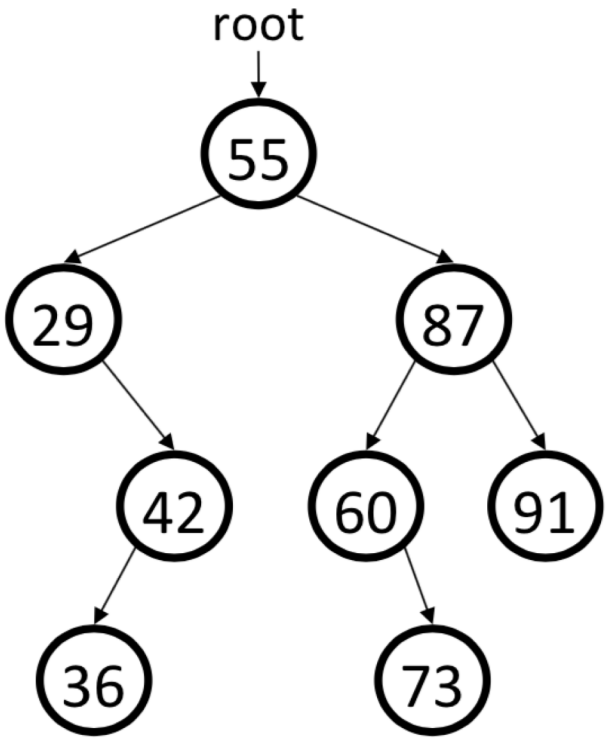



```
remove(root, 55);
```



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);`



Recap

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

Next time: Balanced Binary Trees and other advanced trees