MTH 4300/4299: Programming and Computer Science II

Lecture 20: Binary Trees

Jaime Abbariao

▒▒ Quiz

Fill in the function below. It should return `true` if `a` and `b` are anagrams (same letters, same counts), `false` otherwise. Use `std::unordered_map<char, int>`.

Example: `is_anagram("listen", "silent")` → `true`, `is_anagram("foo", "bar")` → `false`.

```
#include <string>
#include <unordered_map>

bool is_anagram(const std::string &a, const std::string &b) {
  // TODO: use std::unordered_map<char, int>
}
```

Quiz Solution

```cpp
bool is_anagram(const std::string &a, const std::string &b) {
  if (a.size() != b.size()) return false;
  std::unordered_map<char, int> counts;
  for (int i = 0; i < a.size(); i++) counts[a[i]]++;
  for (int i = 0; i < b.size(); i++) counts[b[i]]--;
  for (auto it = counts.begin(); it != counts.end(); ++it) {
    if (it->second != 0) return false;
  }
  return true;
}
```

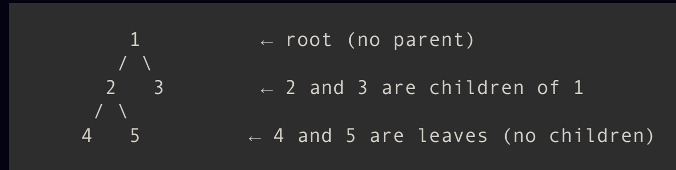
Add for `a`, subtract for `b`. If anything is left over, they're not anagrams. O(n) — one pass each.

Why Trees?

Last time we got O(1) average lookup with `unordered_map`. So why a new structure?

Need	Hash table	Tree
Lookup by key	O(1) avg	O(log n) — but ordered
Keys in sorted order	no	yes
Range queries	no	yes
Hierarchical data (file system, DOM)	awkward	natural

Trees are how we represent hierarchy and how we get lookup that's both fast and ordered.



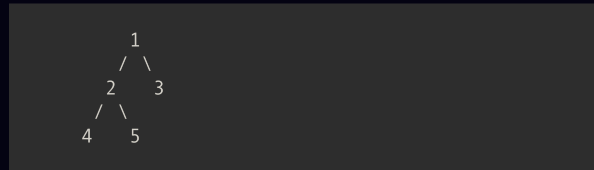
- Root: the top node.
- Parent / child: 1 is the parent of 2; 2 is a child of 1.
- Leaf: a node with no children.
- Edge: the line connecting parent and child.
- Depth of a node: edges from the root. Depth(4) = 2.
- Height of the tree: longest path from root to any leaf. Height = 2 here.
- Subtree: any node + everything beneath it.

▒▒▒ Binary Tree

A binary tree is a tree where every node has at most two children, called left and right.

That's it. The order matters — left and right are distinct.

We'll use this exact tree for the rest of the lecture:



Five nodes. Root 1. Leaves 3, 4, 5.

▒▒ Recursion Refresher

Trees are recursive: every subtree is itself a tree. So our code will be recursive too.

Recipe for any recursive function:

1. Base case — the smallest input you can answer directly.
2. Recursive case — break the problem into smaller pieces, solve those, combine.

Quick example: sum of integers 1 to n.

```
int sum(int n) {
  if (n == 0) return 0;          // base case
  return n + sum(n - 1);         // recursive case
}
```

For trees, the base case will almost always be `node == nullptr`.

▓▓ Building the TreeNode Struct (1/3)

Start simple — a node holds a value:

```
struct TreeNode {
  int value;
};
```

Picture:

```
[ 5 ]
```

A box with a number. No connections yet.

▒▒ Building the TreeNode Struct (2/3)

Now give it ONE child pointer, like a linked list:

```
struct TreeNode {
  int value;
  TreeNode* next;
};
```

Picture:

```
[ 5 | • ] → [ 9 | nullptr ]
```

This is just a linked list. Each node points to one other node.

▒▒ Building the TreeNode Struct (3/3)

For a binary tree, we want two child pointers — left and right:

```
struct TreeNode {
  int value;
  TreeNode* left;
  TreeNode* right;

  TreeNode(int v) : value(v), left(nullptr), right(nullptr) {}
};
```

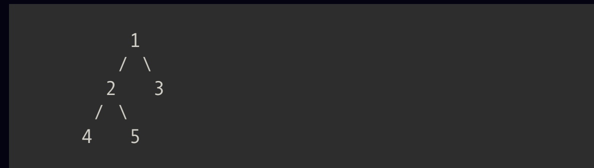
Picture:

```
        [ 5 | • | • ]
             /     \
        [ 2 | n | n ]   [ 9 | n | n ]
```

The constructor sets `left` and `right` to `nullptr` so we never have garbage pointers. Forgetting this is the #1 bug source today.

▓▓ Building Our Tree by Hand

Let's actually build:




```
TreeNode* root = new TreeNode(1);              //         1
root->left  = new TreeNode(2);             //        / \
root->right = new TreeNode(3);             //       2   3
root->left->left  = new TreeNode(4);       //      / \
root->left->right = new TreeNode(5);       //     4   5
```

Each `new` returns a pointer to a fresh node. We hook them together by assigning `left` and `right`.

▒▒▒ Traversals — The Three Orders

A traversal visits every node once. For binary trees, the three classic orders only differ in when we visit the current node:

Order	When we visit the node
Preorder	before recursing into children
Inorder	between the two children
Postorder	after both children

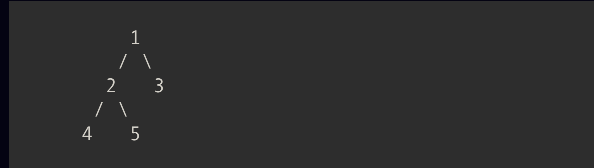
Same code shape, one line moves. We'll trace all three on our 5-node tree.

▓▓ Preorder — "Top-Down Copy"

Use case: copying or serializing a tree (you need the parent before its children).

```
void preorder(TreeNode* node) {
  if (node == nullptr) return;     // base case
  std::cout << node->value << " "; // visit FIRST
  preorder(node->left);
  preorder(node->right);
}
```

On our tree:



Output: `1 2 4 5 3`

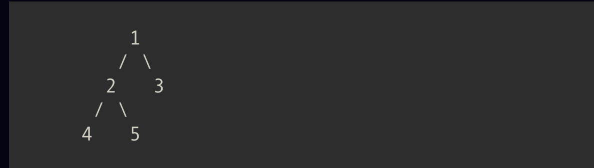
Visit the node, then recurse left, then recurse right.

▒▒ Inorder — "Sorted (for BSTs)"

Use case: for a binary search tree, this prints values in sorted order. (Big tease for next lecture.)

```
void inorder(TreeNode* node) {
  if (node == nullptr) return;
  inorder(node->left);
  std::cout << node->value << " "; // visit BETWEEN
  inorder(node->right);
}
```

On our tree:



Output: `4 2 5 1 3`

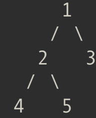
Left subtree first, then the node, then right subtree.

▒▒ Postorder — "Children Before Parent"

Use case: deleting the tree, or evaluating an expression tree (kids must be done first).

```cpp
void postorder(TreeNode* node) {
  if (node == nullptr) return;
  postorder(node->left);
  postorder(node->right);
  std::cout << node->value << " "; // visit LAST
}
```

On our tree:



```
      1
     / \
    2   3
   / \
  4   5
```

Output: `4 5 2 3 1`

Notice 1 (the root) comes out last — every parent waits for its children.

▒▒ The Recursive Recipe for Trees

Every function we write today has the same shape:

```
ReturnType f(TreeNode* node) {
  if (node == nullptr) return /* base */;
  auto L = f(node->left);
  auto R = f(node->right);
  return /* combine node, L, R */;
}
```

Three things change between functions:

1. What you return for `nullptr` (the base value).
2. How you combine `L` and `R`.
3. Whether the node's own value contributes.

Once you see this template, the next three slides are tiny variations.

▒▒ count_nodes

How many nodes are in the tree?

- Base: empty tree has 0 nodes.
- Combine: 1 (this node) + left count + right count.

```
int count_nodes(TreeNode* node) {
  if (node == nullptr) return 0;
  return 1 + count_nodes(node->left) + count_nodes(node->right);
}
```

On our tree: `count_nodes(root)` → `5`.

▒▒ height

How tall is the tree (longest root-to-leaf path in edges)?

- Base: empty tree has height -1 (so a single node has height 0).
- Combine: 1 + max of the two child heights.

```
int height(TreeNode* node) {
  if (node == nullptr) return -1;
  int hl = height(node->left);
  int hr = height(node->right);
  return 1 + std::max(hl, hr);
}
```

On our tree: `height(root)` → `2`.

▒▒ contains

Does the tree contain the value `target`? (Plain structural search — not BST yet.)

- Base: empty tree contains nothing → `false`.
- Combine: this node matches, OR the left subtree has it, OR the right does.

```
bool contains(TreeNode* node, int target) {
  if (node == nullptr) return false;
  if (node->value == target) return true;
  return contains(node->left, target)
      || contains(node->right, target);
}
```

Worst case O(n) — we may have to look everywhere. Next lecture we'll make this O(h).

▓▓ Cleaning Up Memory

We used new for every node. We must delete every node. In what order?

Wrong: delete the root first. Now we've lost the only pointers to its children → memory leak.

Right: delete the children first, then the parent. That's postorder.

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

(With Rule of Five, a class owning its tree could automate this. For now we do it by hand.)

```cpp
int main() {
  TreeNode* root = new TreeNode(1);
  root->left  = new TreeNode(2);
  root->right = new TreeNode(3);
  root->left->left  = new TreeNode(4);
  root->left->right = new TreeNode(5);

  preorder(root);  std::cout << std::endl;  // 1 2 4 5 3
  inorder(root);   std::cout << std::endl;  // 4 2 5 1 3
  postorder(root); std::cout << std::endl;  // 4 5 2 3 1

  std::cout << count_nodes(root) << std::endl;  // 5
  std::cout << height(root)      << std::endl;  // 2
  std::cout << contains(root, 4) << std::endl;  // 1
  std::cout << contains(root, 9) << std::endl;  // 0

  delete_tree(root);
  return 0;
}
```

▓▓ Tease: Binary Search Tree

What if we enforce an ordering rule on the tree?

BST property: for every node, all values in the left subtree are `<` node, all values in the right subtree are `>` node.

Inorder traversal: `1 3 4 5 8 9` — sorted! And lookup becomes O(h) — ignore half the tree at every step.

That's next lecture.

Summary

- A binary tree is a node with `left` and `right` child pointers — define it recursively, code it recursively.
- Three traversals (preorder, inorder, postorder) differ only in when the node is visited.
- The recursive recipe: base case `nullptr`, recurse left, recurse right, combine.
- `count_nodes`, `height`, `contains`, `delete_tree` are all the same shape.
- Always pair `new` with `delete`, and always delete in postorder so children aren't orphaned.
- (There's also level-order with `std::queue` — we'll see it later.)

Next lecture: binary search trees — order the tree, get O(h) lookup.

▒▒▒ Lab: Tree Helpers — Setup

Complete `count_nodes`, `height`, and `sum_values` for the tree below. `delete_tree` is provided so you don't have to worry about leaks.

```
#include <iostream>
#include <algorithm>

struct TreeNode {
  int value;
  TreeNode* left;
  TreeNode* right;
  TreeNode(int v) : value(v), left(nullptr), right(nullptr) {}
};

int count_nodes(TreeNode* node) {
  // TODO
}

int height(TreeNode* node) {
  // TODO: empty tree has height -1
}

int sum_values(TreeNode* node) {
  // TODO
}

void inorder(TreeNode* node) {
  if (node == nullptr) return;
  inorder(node->left);
  std::cout << node->value << " ";
  inorder(node->right);
}

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

▒▒▒ Lab: main and Expected Output

```
int main() {
  TreeNode* root = new TreeNode(10);
  root->left  = new TreeNode(5);
  root->right = new TreeNode(20);
  root->left->left  = new TreeNode(3);
  root->left->right = new TreeNode(7);
  root->right->right = new TreeNode(25);

  std::cout << "count: "  << count_nodes(root) << std::endl;
  std::cout << "height: " << height(root)      << std::endl;
  std::cout << "sum: "    << sum_values(root)  << std::endl;
  std::cout << "inorder: ";
  inorder(root);
  std::cout << std::endl;

  delete_tree(root);
  return 0;
}
```

Expected output:

```
count: 6
height: 2
sum: 70
inorder: 3 5 7 10 20 25
```

Bonus: write `int max_value(TreeNode* node)` that returns the largest value in the tree (assume non-empty). Same recursive recipe — base case is a single node, combine with `std::max`.

Hints:

- All three required functions are 2-4 lines and follow the recipe from the lecture.
- For `height`, return `-1` on `nullptr` so a leaf comes out as `0`.
- The inorder output is not sorted in general — only sorted for BSTs (next lecture).