MTH 4300/4299: Programming and Computer Science II

Lecture 21: Queues and Level Order Traversal; BSTs; Tries

Jaime Abbariao

▒▒ Where We Are

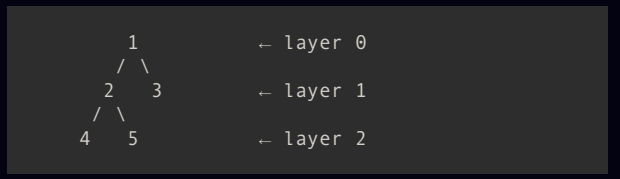
Last lecture we met three traversals:

Order	When we visit the node
Preorder	before children
Inorder	between children
Postorder	after children

All three are depth-first — recursion dives all the way down one side before coming back.

But sometimes we want to visit nodes layer by layer, top to bottom. That's a fourth traversal: level-order — and recursion alone won't give it to us.

▒▒ Why Level Order?



Level-order output on this tree: `1 2 3 4 5`.

Why we care:

- "Find the shallowest node matching X" → stop at the first match; layers come out closest-first.
- Printing a tree top-to-bottom (debugging, visualizing).
- Shortest path in an unweighted graph — same idea, called BFS (breadth-first search).

Recursion naturally does DFS. For BFS we need a different tool.

▓▓ The Tool: std::queue

A queue is FIFO — first in, first out. We saw it in Lecture 18.

Three operations we need:

```
#include <queue>

std::queue<TreeNode*> q;

q.push(root);        // put a node at the back
TreeNode* n = q.front();  // peek at the front
q.pop();             // remove the front
bool empty = q.empty();
```

The idea: push the root. Repeatedly pop the front, visit it, then push its children. Because children go to the back, the whole current layer is processed before any of the next layer.

level_order — Code

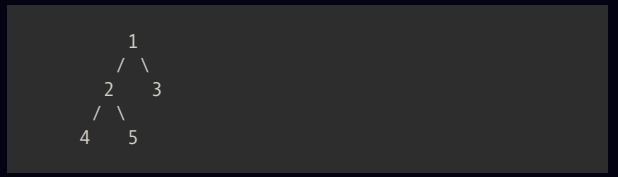
```cpp
#include <queue>

void level_order(TreeNode* root) {
  if (root == nullptr) return;
  std::queue<TreeNode*> q;
  q.push(root);
  while (!q.empty()) {
    TreeNode* node = q.front();
    q.pop();
    std::cout << node->value << " ";
    if (node->left  != nullptr) q.push(node->left);
    if (node->right != nullptr) q.push(node->right);
  }
}
```

No recursion. The queue is doing the bookkeeping the call stack would normally do.

level_order — Traced

Tree:



Step-by-step — queue contents on the left, output on the right:

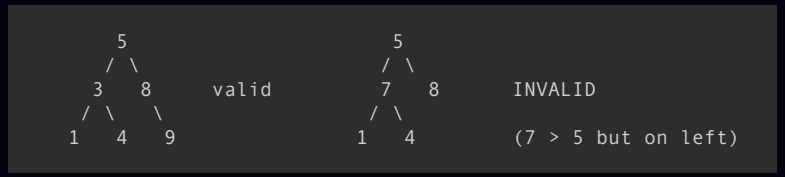
```
Queue: [1]           →  pop 1, print 1, push 2,3
Queue: [2, 3]        →  pop 2, print 2, push 4,5
Queue: [3, 4, 5]     →  pop 3, print 3, (no children)
Queue: [4, 5]        →  pop 4, print 4, (no children)
Queue: [5]           →  pop 5, print 5, (no children)
Queue: []            →  done.
```

Output: `1 2 3 4 5`. Exactly layer-by-layer.

▒▒ From Binary Tree to BST

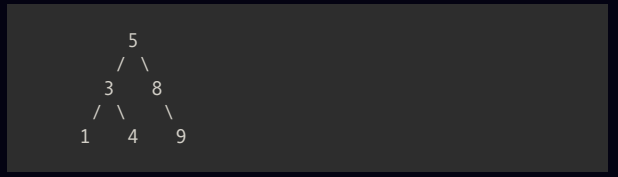
Last lecture: `contains` was `O(n)` — we had to search everywhere because the tree had no ordering.

Binary Search Tree (BST) property: for every node, every value in the left subtree is `<` node, every value in the right subtree is `>` node.



With this rule, at every node we can eliminate half the tree: smaller goes left, larger goes right. Lookup drops from `O(n)` to `O(h)`.

Inorder on a BST is Sorted



Inorder output: `1 3 4 5 8 9` — sorted.

Why: at every node, inorder visits everything smaller than it (the entire left subtree), then the node itself, then everything larger than it (the right subtree). That's literally the definition of sorted order.

Free sorting algorithm: insert all values into a BST, then inorder traversal. (We'll see a faster way to sort later; this one is `O(n log n)` only if the tree stays balanced.)

▒▒ BST contains — O(h)

Last lecture's `contains` checked both children. Now we only check one:

```
bool contains(TreeNode* node, int target) {
  if (node == nullptr) return false;          // base: not found
  if (target == node->value) return true;     // found
  if (target < node->value)
    return contains(node->left, target);      // go left only
  else
    return contains(node->right, target);     // go right only
}
```

At each step we eliminate half the tree.

What's `h`? The height. If the tree is balanced, `h ≈ log n`. If it's skewed (we'll see this in a minute), `h = n` and we're back to `O(n)`.

BST insert — The Idiom

```
TreeNode* insert(TreeNode* node, int value) {
  if (node == nullptr) return new TreeNode(value);  // base: attach here
  if (value < node->value)
    node->left  = insert(node->left,  value);       // recurse left
  else
    node->right = insert(node->right, value);       // recurse right
  return node;                                      // return THIS node unchanged
}
```

Caller uses it like this:

```
root = insert(root, 7);
```

Why return `TreeNode*`? When recursion hits `nullptr`, we create a new node and return its pointer. The parent call catches that pointer and assigns it to `node->left` or `node->right`. That's how the new node gets connected to the tree. If we didn't return, the parent would have no way to rewire.

▒▒▒ insert — Traced

Start with empty tree. Insert `5`:

```
root = insert(nullptr, 5)
  node == nullptr → return new TreeNode(5)
root ← pointer to [5]
```

Now insert `3`:

```
root = insert([5], 3)
  3 < 5 → [5].left = insert(nullptr, 3)
                       node == nullptr → return new TreeNode(3)
          [5].left ← pointer to [3]
  return [5]
root ← [5]
```

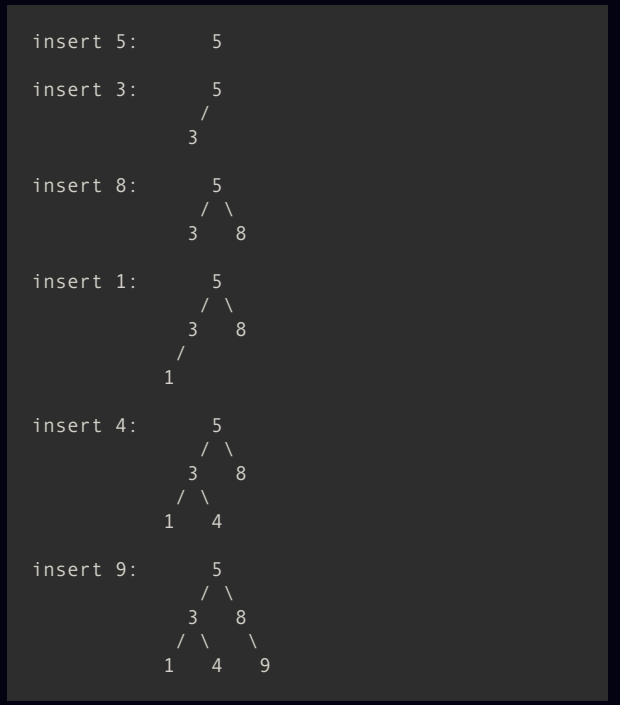
Tree now:

```
    5
   /
  3
```

The recursive call returns the newly created `[3]` pointer; the parent call catches it into `[5].left`. That's the magic of returning `TreeNode*`.

Building a BST: insert 5, 3, 8, 1, 4, 9

After each insert:



Inorder of the final tree: `1 3 4 5 8 9`. Sorted — as promised.

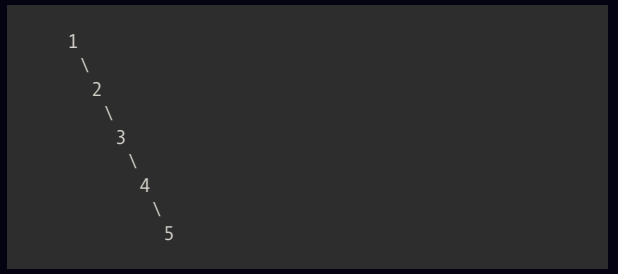
Min, Max, and Balance

Once you see the BST property, min and max are free:

- Minimum = leftmost node. Keep going left until `left == nullptr`.
- Maximum = rightmost node. Keep going right until `right == nullptr`.

Both `O(h)`. In our tree above: min = 1 (far left), max = 9 (far right).

The balance problem. What if we insert `1, 2, 3, 4, 5` in that order?



A linked list. `h = n`. Every operation is `O(n)` — we lost everything.

Fix (beyond this course): self-balancing BSTs — AVL trees, red-black trees (what `std::map` and `std::set` use under the hood). They rotate nodes on insert to keep `h ≈ log n`. Name-dropped for your awareness; we won't implement them.

▒▒ Tries — Motivation

How does your phone suggest `"cat"`, `"car"`, `"cab"` as you type `"ca"`?

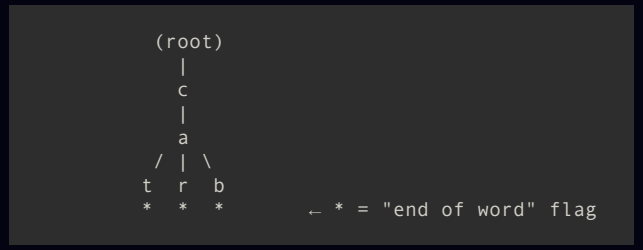
A hash set of words gives you `O(1)` exact lookup — but zero support for prefixes. "Does any word start with `ca`?" forces you to scan the whole set.

A trie (rhymes with "try", from retrieval) is a tree designed for strings where the path from the root spells out the word.

Use cases: autocomplete, spell-check, IP routing tables, dictionary lookup.

▒▒ Trie — Picture

Inserting `"cat"`, `"car"`, `"cab"`:



Walk `c → a → t` and you've spelled `"cat"`. Walk `c → a` and you're at a prefix, not a word (no `*` there). That's why each node needs an `is_end_of_word` flag — it distinguishes "this is a real word that ends here" from "this is just along the path to longer words."

Example: if we also inserted `"car"` and `"cars"`, the `r` node gets a `*` (for `"car"`) AND continues to an `s` node that also has a `*` (for `"cars"`).

```
struct TrieNode {
  TrieNode* children[26];  // one slot per letter a..z
  bool is_end_of_word;

  TrieNode() : is_end_of_word(false) {
    for (int i = 0; i < 26; i++) children[i] = nullptr;
  }
};

void insert(TrieNode* root, const std::string& word) {
  TrieNode* node = root;
  for (char c : word) {
    int i = c - 'a';
    if (node->children[i] == nullptr)
      node->children[i] = new TrieNode();
    node = node->children[i];
  }
  node->is_end_of_word = true;
}
```

Walk down the tree letter by letter, creating missing nodes. Mark the final node as end-of-word.

▒▒ Trie — Lookup vs Prefix

Two operations, almost identical code:

- `contains(word)` — walk the letters. If any child is `nullptr`, return `false`. At the end, return `node->is_end_of_word`.
- `starts_with(prefix)` — walk the letters the same way. If you finish the walk, return `true`. You don't care about `is_end_of_word`.

	Hash set of strings	Trie
Exact lookup	`O(1)` avg	`O(L)` (L = word length)
Prefix queries	no	`O(L)` — free
Memory	compact	larger (26 pointers per node)

Trie trades memory for prefix power. That's the tradeoff.

Summary

- Level-order traversal visits a tree layer by layer using a `std::queue`. Recursion can't easily do BFS; the queue tracks the frontier for us.
- BSTs add one rule: left subtree `<` node `<` right subtree. That rule buys us `O(h)` lookup, `O(h)` insert, free sorted output via inorder, and free min/max.
- Balance matters. Skewed trees degrade to `O(n)`. Real-world `std::map` / `std::set` keep themselves balanced for you.
- Tries are trees over strings. Paths spell words; an `is_end_of_word` flag marks real words. They make prefix queries cheap — hash sets can't.

Next lecture: sorting algorithms.

▒▒ Lab: BST — Setup

Implement `insert` and `contains` for a BST. The `inorder` function is provided so you can verify your `insert` is correct — if your BST is built correctly, inorder should print values in sorted order. That's your self-check.

```cpp
#include <iostream>

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

TreeNode* insert(TreeNode* node, int value) {
  // TODO: base case returns a new TreeNode; otherwise recurse left or right
  //       and return node at the end.
}

bool contains(TreeNode* node, int target) {
  // TODO: base case empty → false; compare target to node->value and
  //       go left OR right (not both).
}

int count_in_range(TreeNode* node, int lo, int hi) {
  // TODO: return how many values in the BST fall in [lo, hi] (inclusive).
  //       Use the BST property to PRUNE — don't visit subtrees that
  //       can't contain anything in range.
}

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 = nullptr;
  int values[] = {5, 3, 8, 1, 4, 9};
  for (int v : values) root = insert(root, v);

  std::cout << "inorder: ";
  inorder(root);
  std::cout << std::endl;

  std::cout << "contains 4: " << contains(root, 4) << std::endl;
  std::cout << "contains 7: " << contains(root, 7) << std::endl;
  std::cout << "contains 9: " << contains(root, 9) << std::endl;

  std::cout << "count in [3, 8]: "  << count_in_range(root, 3, 8)   << std::endl;
  std::cout << "count in [4, 100]: " << count_in_range(root, 4, 100) << std::endl;
  std::cout << "count in [0, 2]: "  << count_in_range(root, 0, 2)   << std::endl;

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

Expected output:

```
inorder: 1 3 4 5 8 9
contains 4: 1
contains 7: 0
contains 9: 1
count in [3, 8]: 4
count in [4, 100]: 4
count in [0, 2]: 1
```