MTH 4300/4299

Midterm 2 Results; Trees

Jaime Abbariao

Midterm 2 Results

Statistics

- Average: 30
- Standard Deviation: 14.79
- Median: 35
- Maximum: 50

Final Exam

For those who want a headstart for studying for the final, the following topics will be heavily covered:

- Hashing
- Binary Trees
- Graphs

Introduction to Trees

What is a tree?

A **tree** is a hierarchical data structure consisting of nodes connected by edges, where:

- There is exactly one **root** node (no parent)
- Each node can have zero or more **children**
- Each node (except root) has exactly one **parent**
- No cycles exist in the structure

Key terminology:

- **Root**: The topmost node
- **Leaf**: A node with no children
- **Height**: Maximum distance from root to any leaf
- **Depth**: Distance from root to a specific node
- **Subtree**: A tree formed by a node and all its descendants

How can we represent a tree in code?

Binary Tree Node Structure:

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

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

General Tree Node (multiple children):

```
struct Node {
    int data;
    std::vector<Node*> children;
```

How can we represent a tree in code? (ii)

```
    Node(int value) : data(value) {}
};
```

Creating a simple binary tree:

```
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);
```

Different operations with trees

Common tree operations include:

- **Insertion**: Adding a new node to the tree
- **Deletion**: Removing a node from the tree
- **Search**: Finding a specific value in the tree
- **Traversal**: Visiting all nodes in a specific order
- **Height calculation**: Finding the maximum depth
- **Size calculation**: Counting total number of nodes

Tree traversal is particularly important - there are four main types:

1. **In-order** (Left → Root → Right)
2. **Pre-order** (Root → Left → Right)
3. **Post-order** (Left → Right → Root)
4. **Level-order** (Breadth-first, level by level)

In-order traversal

In-order traversal visits nodes in this order: Left subtree → Root → Right subtree

For binary search trees, this gives values in sorted order!

```
void in_order_traversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    in_order_traversal(node->left);   // Visit left subtree
    std::cout << node->data << " ";   // Process current node
    in_order_traversal(node->right);  // Visit right subtree
}
```

Example: Tree with values [4, 2, 5, 1, 3]

- In-order output: `1 2 3 4 5`

Pre-order traversal

Pre-order traversal visits nodes in this order: Root → Left subtree → Right subtree

Useful for copying trees or creating prefix expressions!

```
void pre_order_traversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    std::cout << node->data << " ";    // Process current node
    pre_order_traversal(node->left);   // Visit left subtree
    pre_order_traversal(node->right);  // Visit right subtree
}
```

Example: Tree with values [4, 2, 5, 1, 3]

- Pre-order output: `4 2 1 3 5`

Post-order traversal

Post-order traversal visits nodes in this order: Left subtree → Right subtree → Root

Useful for deleting trees or evaluating postfix expressions!

```
void post_order_traversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }

    post_order_traversal(node->left);   // Visit left subtree
    post_order_traversal(node->right);  // Visit right subtree
    std::cout << node->data << " ";     // Process current node
}
```

Example: Tree with values [4, 2, 5, 1, 3]

- Post-order output: `1 3 2 5 4`

Level-order traversal

Level-order traversal visits nodes level by level, from left to right

Uses a queue data structure - breadth-first search!

```
#include <queue>

void level_order_traversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    std::queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
```

Level-order traversal (ii)

```
        std::cout << current->data << " ";  // Process current
node

        if (current->left != nullptr) {
            q.push(current->left);
        }
        if (current->right != nullptr) {
            q.push(current->right);
        }
    }
}
```

Example: Tree with values [4, 2, 5, 1, 3]

- Level-order output: `4 2 5 1 3`

Tree Traversal Visual Summary

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

Listing 1: Example Binary Tree

Traversal Methods and Results:

Method	**Order**	**Output**
In-order	Left ⟶ Root ⟶ Right	`1 2 3 4 5`
Pre-order	Root ⟶ Left ⟶ Right	`4 2 1 3 5`

Tree Traversal Visual Summary (ii)

Post-order	Left → Right → Root	`1 3 2 5 4`
Level-order	Level by level	`4 2 5 1 3`