LECTURE 24

Tree Applications

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

LECTURE 24

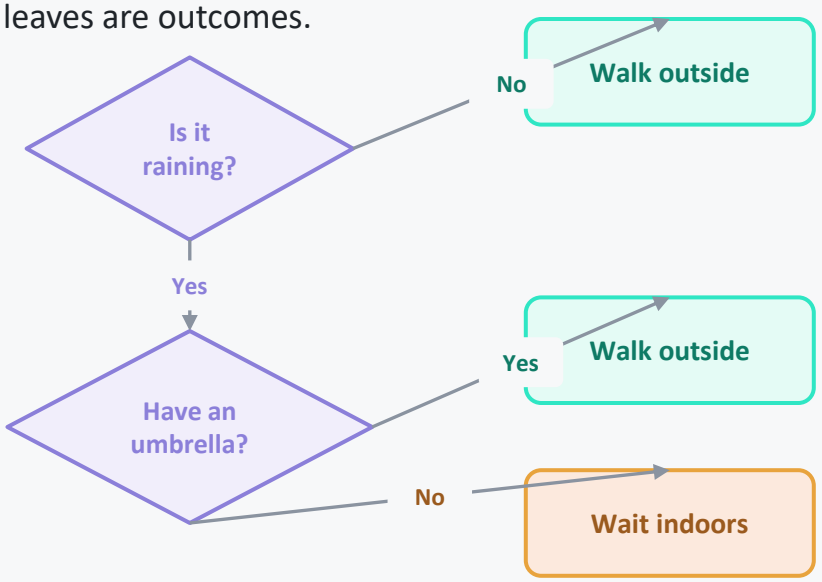
Agenda / Learning Objectives

1. Decision trees, briefly, plus a compiled example that walks one
2. Expression trees: building from postfix, evaluating, reconstructing infix
3. Huffman coding: an optimal compression tree built with a min-heap
4. Huffman decoding: a real, verified encode-then-decode round trip
5. A survey of where binary trees show up in real systems
6. Common pitfalls when applying trees to real problems

APPLICATION 1

Decision Trees

A decision tree represents a sequence of yes/no decisions: internal nodes are questions, leaves are outcomes.



STRUCTURALLY IDENTICAL TO A BST

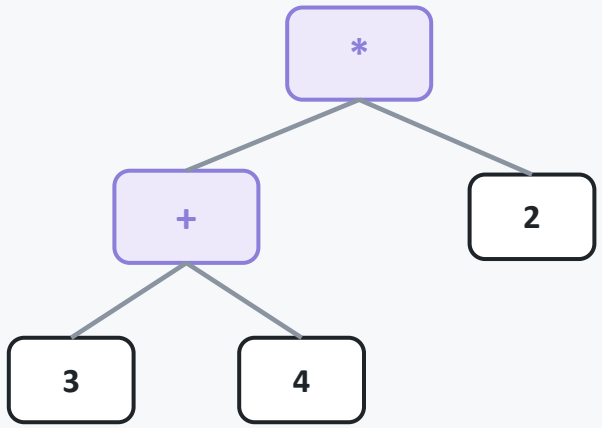
- Every internal node is a question with two (or more) outcomes.
- Every leaf is a final answer with no further branching.
- Same “smaller goes left, larger goes right” idea, except the branching condition is an arbitrary yes/no question, not a numeric comparison.

Verified: “not raining” reaches the leaf after only ONE question — a decision tree's worst case is bounded by its height, exactly like a BST search.

APPLICATION 2

Expression Trees

An expression tree represents arithmetic: every leaf is an operand, every internal node is an operator with two children as its operands.



represents (3 + 4) × 2

Evaluating = post-order traversal: evaluate both children first, then apply the operator at the node.

expression_tree.cpp — "3 4 + 2 *" → 14

```
ExprNode* buildFromPostfix(const string& pf) {
    stack<ExprNode*> nodes;
    for (each token) {
        ExprNode* node = new ExprNode(token[0]);
        if (isOperator(token[0])) {
            node->right = nodes.top(); nodes.pop();
            node->left  = nodes.top(); nodes.pop();
        }
        nodes.push(node);
    }
    return nodes.top();
}
```

ONE TREE, THREE MEANINGS

Traversal Order Changes Meaning

Traversal	Produces	Example (for (3+4)×2)
Pre-order	Prefix notation	* + 3 4 2
In-order	Infix notation	((3 + 4) * 2)
Post-order	Evaluates a number	14

expr_infix.cpp

```
string toInfix(ExprNode* node) {
    if (!isOperator(node->value)) return string(1, node->value);
    return "(" + toInfix(node->left) + " " + node->value + " " + toInfix(node->right) + ")";
}
```

Always fully parenthesized — unambiguous without reasoning about operator precedence at print time.

APPLICATION 3

Huffman Coding: Building the Tree

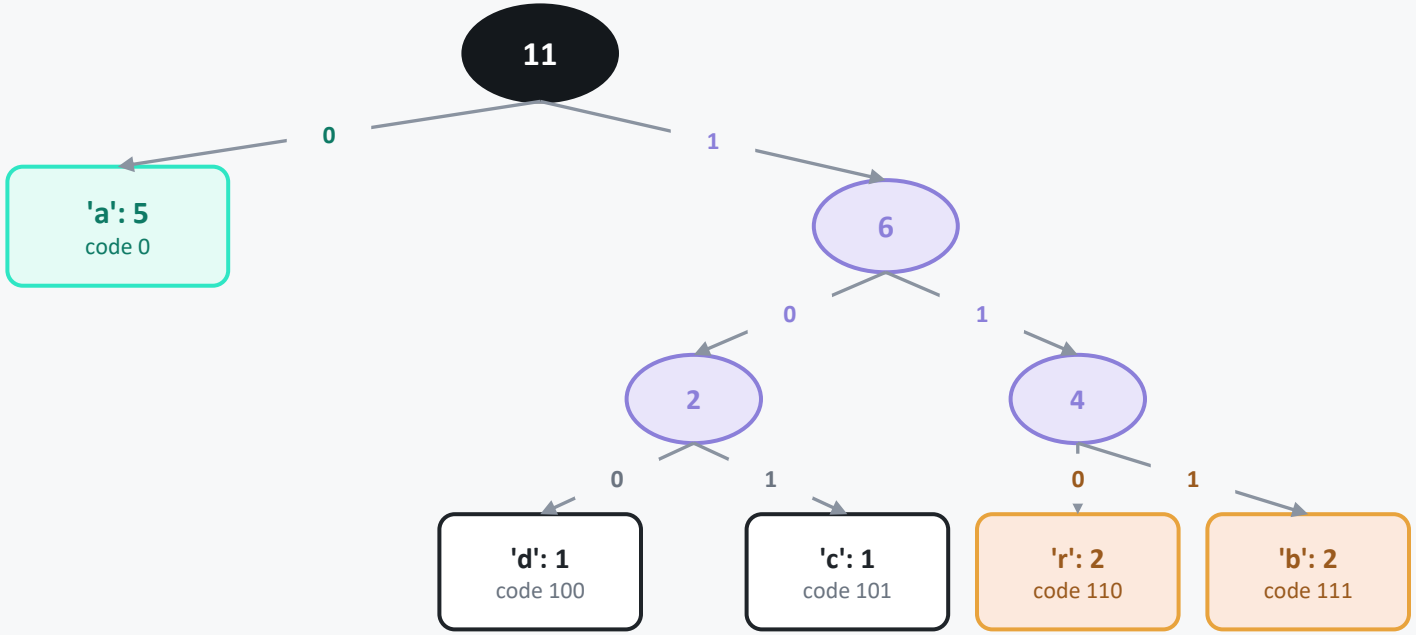
Huffman coding gives more-frequent characters shorter binary codes, built with a min-heap (Lecture 23):

1. Put every character in a min-heap, keyed by frequency.
2. Repeatedly extract the two smallest, merge into a new internal node (sum of frequencies), insert back.
3. Stop when one node remains — the root of the Huffman tree.
4. Walk the tree: left = 0, right = 1; each leaf's path is that character's code.

```
Example text: "abracadabra"
a: 5   b: 2   r: 2   c: 1   d: 1   — 'a' is by far the most frequent, so it should end up shallowest
in the tree.
```

VISUALIZED

The Huffman Tree for "abracadabra"



'a' sits at depth 1 (a 1-bit code) because it was the LAST node merged — the two rarest ('c', 'd') were merged first, then buried deeper as larger subtrees stacked on top.

VERIFIED

A Real Compression Result

88 bits
fixed-width, 8 bits/char

23 bits
Huffman coding

↓ 74%
reduction for this 11-char text

huffman_coding.cpp — output

```
'a': freq 5, code 0     'r': freq 2, code 110     'b': freq 2, code 111
'c': freq 1, code 101   'd': freq 1, code 100
```

Guaranteed, not printed-order-dependent: characters print in unordered_map's internal hash order, not frequency order — but the CODE LENGTHS are guaranteed: more frequent characters always get shorter or equal codes, never the reverse.

CLOSING THE LOOP

Huffman Decoding: a Verified Round Trip

huffman_roundtrip.cpp

```
string decode(const string& bits,
              HuffmanNode* root) {
    string result;
    HuffmanNode* current = root;
    for (char bit : bits) {
        current = (bit == '0')
            ? current->left : current->right;
        if (current->left == nullptr &&
            current->right == nullptr) {
            result += current->character;
            current = root;   // restart
        }
    }
    return result;
}
```

Original: **"abracadabra"**

Encoded: 01111100101010001111100

Decoded: **"abracadabra"**

Round trip matches? YES

Prefix-free by construction: no code is a prefix of another, since every code is a distinct leaf — that's what makes “restart at root after every leaf” correct.

23 bits decoded matches the total-bits figure from the previous slide — the same computation, now carried out on the full message.

SURVEY

Applications of Binary Trees

Application	Tree Concept
File compression (ZIP, JPEG)	Huffman trees
Compilers and calculators	Expression trees
Decision support / ML classifiers	Decision trees
Ordered sets and maps	Self-balancing BSTs (Lecture 22's AVL)
Task/event scheduling	Heaps as priority queues (Lecture 23)
Autocomplete and spell-check	Trie — same node/child ideas
Databases and file systems	B-trees — tuned for disk/SSD access
Parsing HTML, JSON, XML	A tree mirrors the nested document structure

WATCH OUT

Common Pitfalls When Applying Trees

⇆ Confusing which traversal a task needs — post-order for evaluation, in-order for infix; the wrong order gives nonsense.

✓ Assuming Huffman codes need an explicit prefix-free check — a tree-based code gets this for free (every code is a distinct leaf).

∅ Building a Huffman tree from a single-character input — the merge loop never runs; generateCodes must handle the empty-path case.

… Decoding a corrupted or truncated bitstring — the walk is left partway down the tree with no character emitted for trailing bits.

Key Takeaways

1. Decision trees model branching yes/no processes — the same idea as a BST's comparison-based branching, generalized to arbitrary questions.

2. Expression trees model arithmetic: post-order evaluates a number, in-order reconstructs infix, pre-order produces prefix — one tree, three meanings.

3. Huffman coding builds an optimal-length binary code by repeatedly merging the two least-frequent nodes with a min-heap.

4. Huffman codes are prefix-free by construction, which is exactly what makes unambiguous decoding possible.

5. This closes Unit 5. Unit 6 moves from trees to graphs, where a node can connect to any number of others in any pattern at all.