Huffman Trees & Applications

One greedy idea, two payoffs: the cheapest order to merge unequal runs, and the compression algorithm inside every ZIP file, JPEG, and MP3.

Dr. Manu Shrivastava

Course Instructor · Consultation Fri 2–5 PM, LHC 308F

40 minutes

Session outcome: apply Huffman coding and tree construction methods

Today, minute by minute

The debt: why merge order suddenly matters 00–04

Replacement selection gave unequal runs. Lecture 4's lone "15" gets its answer.

Merge trees and WEPL — try both plans yourself 04–10

Interactive: two merge orders for runs 2, 4, 5, 15 — cost 43 vs 52.

Animation: Huffman's greedy algorithm, step by step 10–17

Weights 4, 6, 8, 9, 15, 28 → the optimal tree, with a live cost ledger.

Back to external sorting: closing Unit I 17–21

Apply it to our own runs (4, 8, 1); why L4's balanced pairing was fine for equal runs.

Animation: Huffman coding — compressing text 21–30

Fixed vs variable-length codes, prefix property, the CLRS example: 300 → 224 bits.

Why greedy is optimal + complexity 30–35

The exchange argument, sketched for undergraduates. O(n log n) with a heap.

Applications, recap, homework 35–40

ZIP, JPEG, MP3 — and the end of Unit I.

Two lectures ago we planted a question. Time to answer it.

THE EVIDENCE SO FAR

Lecture 4, the 13-record trace: the lone run `15` was read and rewritten in *every single pass* while bigger runs merged around it. You asked (we made you ask): could a smarter schedule avoid that waste?

Lecture 5, replacement selection: runs now come out *unequal by design* — ours were lengths `4, 8, 1`. Unequal runs are no longer an accident; they are the normal output of a good Phase 1. So "in what order do we merge unequal runs?" is now a question we face on every single sort.

THE KEY PHYSICAL FACT

When two runs of lengths a and b are merged, **every record of both is read and written once** — cost proportional to **a + b**. And the output run of length a + b may be merged *again* later, moving those same records *again*.

So a record's total movement = **how many merges its run participates in**. Records in a run merged early and often move many times; records in a run merged once move once. Intuition forming: **small runs can afford many merges; big runs should merge as few times as possible.** Let's make that precise — and provable.

Everyday version: you must combine several stacks of graded answer sheets into one sorted pile. Combining stacks means re-handling every sheet in both. Would you repeatedly re-handle the 500-sheet stack, or keep it aside and let the small stacks consolidate first? You already know the answer in your hands — today we learn its name: **the Huffman tree**.

A merge schedule is a tree; its cost is WEPL

DEFINITIONS

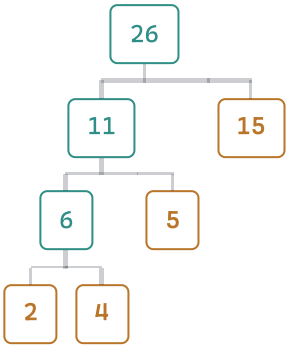
- A **merge tree** records a merge schedule: **square external nodes** = the initial runs (with their lengths as weights); **circular internal nodes** = merges, weight = sum of children.
- A record moves once for every internal node above its run — i.e. its run's **depth**.
- Total merge cost = Σ (run length × its depth) = **Weighted External Path Length**:

$$\text{WEPL}(T) = \Sigma\, w_i \cdot \text{depth}(i) \quad = \Sigma \text{ (internal node weights)}$$

The second equality is the practical one: **just add up the merge sums**. Same number, two readings.

TRY BOTH PLANS — 4 RUNS OF LENGTHS 2, 4, 5, 15

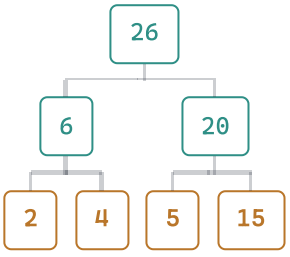
PLAN A — SMALLEST FIRST



AMBER SQUARES = RUNS · TEAL = MERGES · PLAN A: (2+4) → +5 → +15

WEPL = 2·3 + 4·3 + 5·2 + 15·1 = 6 + 12 + 10 + 15 = **43**. Or add the merge (teal) nodes: 6 + 11 + 26 = **43** ✓ — the run of 15 moved only **once**.

PLAN B — BALANCED PAIRS



PLAN B: (2+4) AND (5+15) IN ONE PASS, THEN MERGE THE RESULTS — LECTURE 4'S STYLE

WEPL = 2·2 + 4·2 + 5·2 + 15·2 = 4 + 8 + 10 + 30 = **52**. Merge nodes: 6 + 20 + 26 = **52** ✓ — the run of 15 moved **twice**, and that alone costs 30 of the 52.

The "natural" balanced pairing — exactly what Lecture 4 did — **loses by 9 records of movement** here, because it drags the run of 15 through two merges. Plan A buries

the tiny runs deep (they're cheap to re-move) and keeps 15 at depth 1. Generalize that instinct and you have Huffman's algorithm.

The greedy rule: always merge the two lightest trees

THE ALGORITHM (D. HUFFMAN, 1952)

```
put all n weights in a min-heap, each a 1-node tree
repeat n − 1 times:
    a ← pop smallest tree
    b ← pop next smallest
    c ← new node, children a and b,
        weight = a + b          // one merge
    push c
// the single remaining tree is optimal
```

HOW TO READ THE ANIMATION

The **forest** starts as n single runs. Each step, the **two lightest** trees fuse into a **new tree** whose weight is their sum — and that sum is exactly the **cost of that merge**, posted to the ledger. When one tree remains, the ledger total **is** the WEPL. Weights: `4, 6, 8, 9, 15, 28` (a favourite exam configuration).

INTERACTIVE — BUILD THE OPTIMAL MERGE TREE

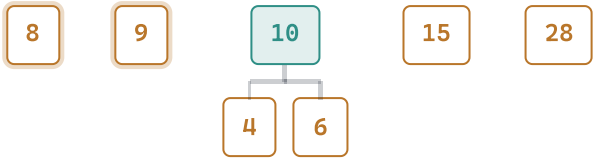
STEP 1 OF 6

FOREST — WATCH THE TREES GROW (AMBER RING = THE TWO ABOUT TO FUSE, TEAL FILL = JUST FUSED)

Start: six single-run trees in a min-heap. The greedy rule: pop the two lightest — **4 and 6** — and fuse them under a new parent.

STEP 2 OF 6

FOREST — WATCH THE TREES GROW (AMBER RING = THE TWO ABOUT TO FUSE, TEAL FILL = JUST FUSED)



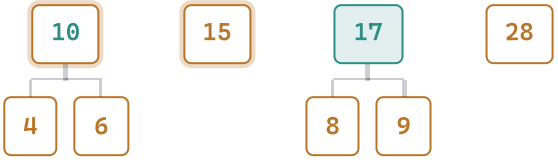
`merge cost +10`

`Total so far = 10`

There it is: 4 and 6 now hang under a new parent of weight **10** — your first real tree. That merge moves 10 records → ledger +10. Next two lightest: **8 and 9**.

STEP 3 OF 6

FOREST — WATCH THE TREES GROW (AMBER RING = THE TWO ABOUT TO FUSE, TEAL FILL = JUST FUSED)



`merge cost +10`

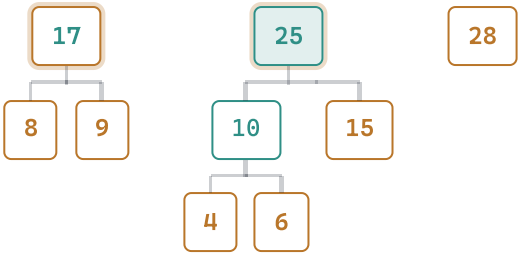
`merge cost +17`

`Total so far = 27`

Fused 8 + 9 → the tree **17**, ledger +17. Now look carefully: the two lightest are the **whole tree 10** and the leaf **15** — grown trees compete exactly like single runs.

STEP 4 OF 6

FOREST — WATCH THE TREES GROW (AMBER RING = THE TWO ABOUT TO FUSE, TEAL FILL = JUST FUSED)

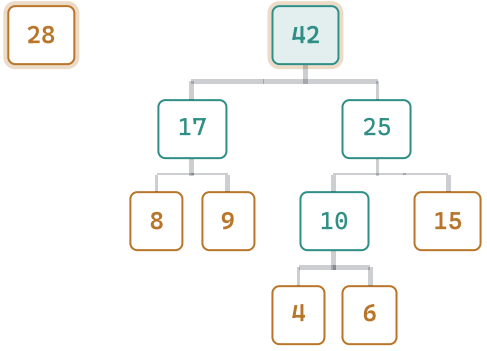



```
merge cost +10
merge cost +17
merge cost +25
Total so far = 52
```

Fused the 10-tree with 15 → tree **25**, ledger +25. See 4 and 6 sinking: they are now at depth 2 of their tree, and every future fusion pushes them deeper — they can afford it. Next: **17 and 25** fuse.

STEP 5 OF 6

FOREST — WATCH THE TREES GROW (AMBER RING = THE TWO ABOUT TO FUSE, TEAL FILL = JUST FUSED)

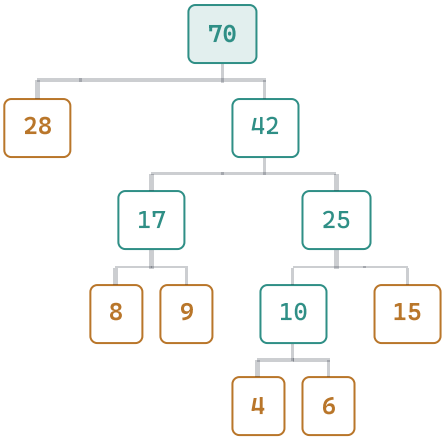



```
merge cost +10
merge cost +17
merge cost +25
merge cost +42
Total so far = 94
```

Fused 17 + 25 → tree **42**, ledger +42. Two trees left. The heavyweight **28 has not moved once** — it kept winning the "not lightest" contest every round.

STEP 6 OF 6

THE FINISHED HUFFMAN TREE



merge cost **+10**

merge cost **+17**

merge cost **+25**

merge cost **+42**

merge cost **+70**

Total so far = 164 = WEPL of the finished tree

Done: final fuse 28 + 42 → **70**. Read the design principle straight off the drawing: 28 sits at depth 1 (moves once); 8, 9, 15 at depth 3; the featherweights 4 and 6 at depth 4. Ledger: 10+17+25+42+70 = **164 = WEPL**. Cross-check by depths: 4·4 + 6·4 + 8·3 + 9·3 + 15·3 + 28·1 = 164 ✓

Apply it to our own runs — and close Unit I's story

OUR REPLACEMENT-SELECTION RUNS: LENGTHS 4, 8, 1

Huffman order: merge the two smallest first — 1 + 4 = 5 (cost 5), then 5 + 8 = 13 (cost 13). Total movement: **18 records**.
WEPL check: 1·2 + 4·2 + 8·1 = 2 + 8 + 8 = 18 ✓

The other order: 4 + 8 = 12 (cost 12), then 12 + 1 = 13 (cost 13). Total: **25 records**.
WEPL: 4·2 + 8·2 + 1·1 = 25. The big run of 8 got dragged through two merges — **39% more I/O**, for choosing the wrong order on just three runs.

TWO LOOSE ENDS, TIED

- **Lecture 4's lone "15":** carrying it down pass after pass was a bad merge tree — it sat at maximum depth. Huffman would merge it into the smallest partner immediately: minimum total movement, question answered.
- **Was Lecture 4 wrong to pair evenly?** No — when all runs are *equal*, Huffman itself produces the balanced, complete tree (all weights tie, pairing is level by level). Balanced pairing is the special case; Huffman is the general law. (Homework asks you to argue this.)

For k-way merging the same idea generalizes: a k-ary Huffman tree (pad with dummy zero-length runs so the arity works out) — mentioned for completeness, not examined in depth.

Unit I is now complete: amortized analysis (the toolkit) → external sorting (the problem) → tournament trees, buffers, replacement selection (the engine) → Huffman merging (the scheduler). Every piece exists because disks punish wasted movement.

The same tree, compressing every file you own

THE COMPRESSION PROBLEM

A 100-character file uses 6 letters with these counts: `a:45 · b:13 · c:12 · d:16 · e:9 · f:5`. A **fixed-length code** needs $\lceil \log_2 6 \rceil = 3$ bits per character → **300 bits**.

Idea: give **frequent letters short codes** and rare letters long ones — like Morse code gave 'E' a single dot. Danger: with variable lengths, where does one code end and the next begin? Solution: a **prefix code** — no codeword is a prefix of another. Then decoding is unambiguous: read bits, walk the tree, hit a leaf, emit, restart.

And here is the punchline: **expected code length = Σ freq × depth = WEPL again.** Minimizing bits is the *same problem* as minimizing merge cost — frequencies play the role of run lengths. Same greedy, same tree.

READING THE CODE OFF THE TREE

- Build the Huffman tree on frequencies (animation →).
- Label every left edge **0**, every right edge **1**.
- A character's code = the 0/1 path from root to its leaf. Leaves only ⟹ prefix property is automatic.
- Frequent characters end up shallow (short codes); rare ones deep (long codes) — Huffman guarantees the optimum.

Decode demo for later: with the tree built, `1100·0·100·1101` reads f–a–c–e: **"face"** in 12 bits instead of 4 × 3 = 12… try `"dead"` and `"cab"` in the homework — that's where the savings show.

INTERACTIVE — BUILD THE CODE FOR A:45 B:13 C:12 D:16 E:9 F:5

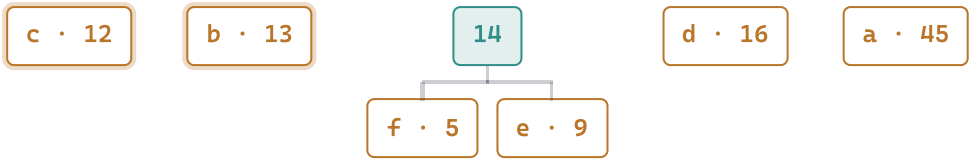
STEP 1 OF 7

FOREST OF FREQUENCY TREES (AMBER RING = TWO RAREST, TEAL FILL = JUST FUSED)

Start: six characters as single-leaf trees, weight = frequency per 100 characters. Fixed-length coding costs 3 bits each → **300 bits**. Greedy begins with the two rarest: **f (5)** and **e (9)**.

STEP 2 OF 7

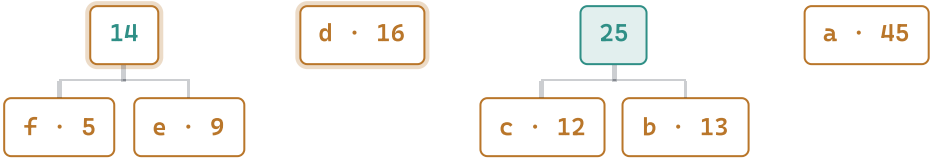
FOREST OF FREQUENCY TREES (AMBER RING = TWO RAREST, TEAL FILL = JUST FUSED)



f and e now hang under parent **14** — one level deeper, so their eventual codes are one bit longer. Fine: they barely occur. Next rarest: **c (12)** and **b (13)**.

STEP 3 OF 7

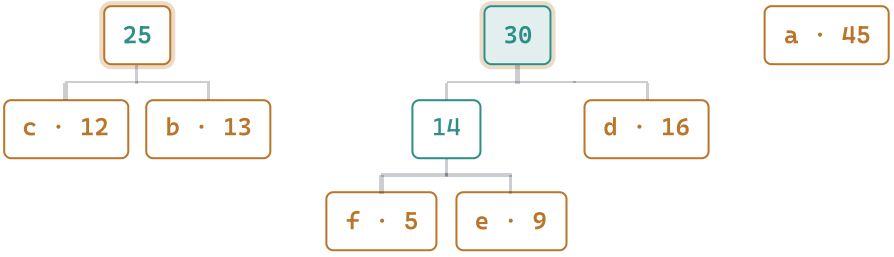
FOREST OF FREQUENCY TREES (AMBER RING = TWO RAREST, TEAL FILL = JUST FUSED)



c + b fused under **25**. Now the lightest two are the **whole 14-tree** and the leaf **d (16)** — trees and leaves compete on equal terms.

STEP 4 OF 7

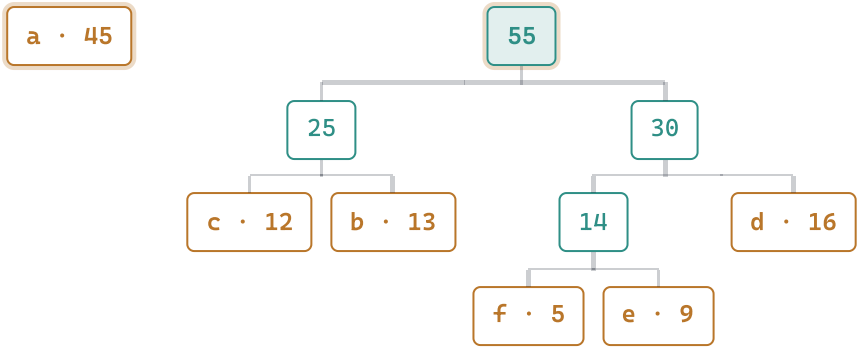
FOREST OF FREQUENCY TREES (AMBER RING = TWO RAREST, TEAL FILL = JUST FUSED)



14-tree + d fused under **30** — f and e are now two levels down. Meanwhile **a (45)**, nearly half the file, still stands alone at depth 0: greedy is hoarding shallowness for the frequent letter. Next: the **25**- and **30**-trees.

STEP 5 OF 7

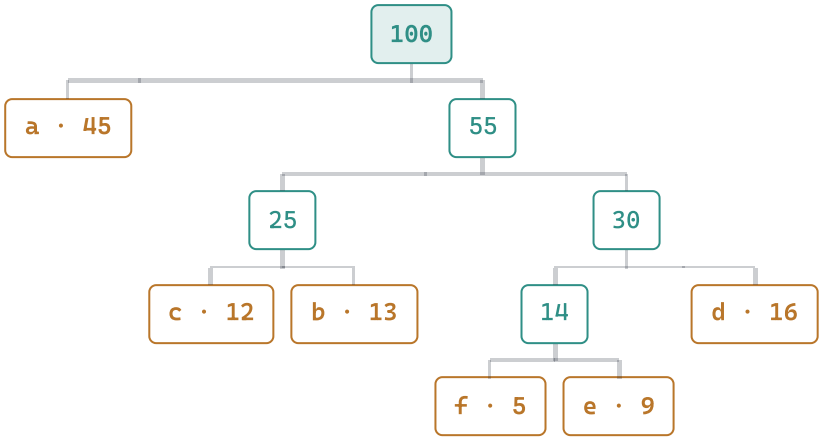
FOREST OF FREQUENCY TREES (AMBER RING = TWO RAREST, TEAL FILL = JUST FUSED)



25 + 30 fused under **55** — everything except *a* is in one tree. Final fusion: **a (45)** with the 55-tree.

STEP 6 OF 7

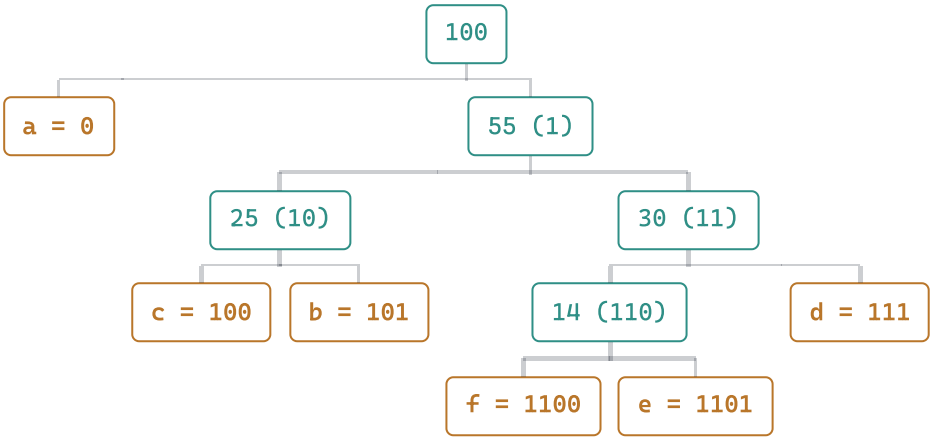
THE FINISHED TREE (WEIGHT 100 = THE WHOLE FILE)



Tree complete. One glance shows the outcome: *a* hangs directly off the root; f and e are four levels down. Now label every left edge **0** and right edge **1**, and read the codes →

STEP 7 OF 7

CODES = ROOT-TO-LEAF PATHS (LEFT EDGE 0, RIGHT EDGE 1)



THE BILL

a: 45×1 = 45	c: 12×3 = 36	b: 13×3 = 39	d: 16×3 = 48	f: 5×4 = 20	e: 9×4 = 36	Σ = 224 bits

The payoff: 224 bits vs 300 fixed — a **25% saving**, provably the best any prefix code can achieve for these frequencies. The shape says it all: a's code is one bit because a is half the file; f and e pay 4 bits because they barely occur. Σ freq × depth strikes again — WEPL, wearing a compression hat.

The exchange argument — grounded in the tree you already built

L6·03's animation showed greedy *working* on weights 4, 6, 8, 9, 15, 28 (final WEPL 164). This slide proves it must always work — using that exact same tree as the running example, not a fresh abstract one.

OBS 1, MADE CONCRETE — A BAD TREE, THEN THE FIX

Suppose someone built a *different* tree on just {4, 6, 28}: merge 28 with 4 first (32), then merge with 6. Result: 6 sits at depth 1, while heavy 28 sits deeper, at depth 2.

$$\text{WEPL} = 6{\cdot}1 + 28{\cdot}2 + 4{\cdot}2 = 6 + 56 + 8 = 70$$

Swap 6 and 28's positions (heavy rises to depth 1, light sinks to depth 2) — nothing else about the tree changes:

$$\text{WEPL} = 28{\cdot}1 + 4{\cdot}2 + 6{\cdot}2 = 28 + 8 + 12 = 48$$

Cost dropped from 70 to 48 just by moving the heavier weight shallower. **Any tree with a heavy weight sitting deeper than a light one can always be improved this way** — so it was never optimal to begin with.

OBS 1, THE GENERAL RULE BEHIND THAT SWAP

Swapping a heavy weight (at deep level d_{deep}) with a light weight (at shallow level $d_{shallow}$) changes the cost by exactly:

$$\Delta\text{WEPL} = -(w_{heavy} - w_{light}) \times (d_{deep} - d_{shallow})$$

Check it against the example: $-(28-6)\times(2-1) = -22$ — exactly the 70→48 drop.

Both factors are always positive whenever heavy-sits-deeper happens (bigger weight, bigger depth-gap) — so ΔWEPL is **always negative**. Conclusion: **in any optimal tree, no heavier weight can sit deeper than a lighter one** — lighter weights always sink to the bottom.

OBS 2, MADE CONCRETE — LOOK AT THE BOTTOM OF YOUR OWN TREE

Go back to the real 6-weight tree from L6·03. Its deepest level (depth 4) holds exactly **one sibling pair: 4 and 6** — the two lightest weights of all six — sitting under their shared parent node "10". This is not a coincidence: every internal node has exactly two children, so the deepest occupied level always comes in sibling pairs, and by Obs 1, the lightest available weights are exactly the ones Obs 1 will always push into those bottom slots. **So some optimal tree always begins with the two lightest as siblings** — precisely Huffman's first move.

…THEN INDUCTION FINISHES IT, ONE FAMILIAR STEP AT A TIME

Merge 4+6→10 (cost 10) and the 6-weight problem becomes a 5-weight problem: {10, 8, 9, 15, 28} — literally the forest after L6·03's animation step 1. The same two arguments now apply to this smaller forest: its own two lightest (8, 9) must again be optimal siblings at its own deepest level.

Repeat: 6 weights → 5 → 4 → 3 → 2 → 1, each step justified the same way. **Complexity:** heap init O(n); n−1 pops/pushes at O(log n) each → **O(n log n)** total — seconds by hand with the forest-and-ledger method.

SANITY CHECK — HUFFMAN VS THE BEST "BALANCED" TREE

Weights 2, 3, 5, 7, 9, 13: the Huffman tree's WEPL is **93**; the best *complete* binary tree on the same weights scores **95**. Balanced is close but not optimal the moment weights are unequal — the two-point gap is precisely the algorithm's edge, and a favourite exam comparison.

One tree, three takeaways — and Unit I closes

- **A merge schedule is a tree, and its cost is WEPL** = Σ weight × depth = Σ internal-node weights. Order matters the moment runs are unequal.
- **Huffman's greedy — always fuse the two lightest — is provably optimal**, by the exchange argument; O(n log n) with a min-heap.
- **The identical tree compresses data:** frequencies for weights, code length for depth, prefix property for free — 300 bits → 224 in the classic example. Inside ZIP/gzip (DEFLATE), JPEG, MP3, PNG. Morse code was the hand-made ancestor.

LECTURE 7 — UNIT II BEGINS

Binary Search Trees Review & AVL Trees

Back to in-memory structures: the degenerate-BST problem from Lecture 2, and the first self-balancing answer. Bring your rotations.

HOMEWORK — BRING TO LECTURE 7

- Build the Huffman tree for weights 2, 3, 5, 7, 9, 13 (forest + ledger method); verify WEPL = 93. Draw the best complete binary tree and verify 95.
- Your L5 homework's replacement-selection runs: compute the optimal merge order and its WEPL; compare against merging in arrival order.
- Using the a–f code built today: encode **"deadface"**; decode **0·101·0·111**. How many bits does "deadface" save vs 3-bit fixed?
- Argue (3–4 sentences): if all n runs have equal length, Huffman's construction yields a complete binary tree — so Lecture 4's balanced pairing was already optimal for equal runs.
- Letter frequencies of **MANIPAL UNIVERSITY JAIPUR** (count them, ignore spaces): build the Huffman code and compute the compression ratio vs 5-bit fixed-length.
- Reading: CLRS §16.3 (Huffman codes); Weiss §10.1.2.

Questions?

Dr. Manu Shrivastava — LHC 308F — Friday 2:00–5:00 PM

> *Next: Lecture 7 — Binary Search Trees Review and AVL Trees. Unit II begins.*