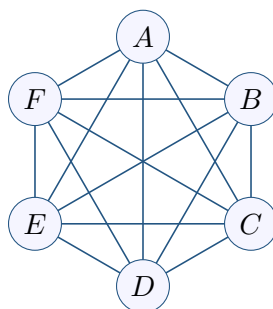


Lecture 15: Trees and Their Applications

Lecturer and scribe: Karthik Gajulapalli

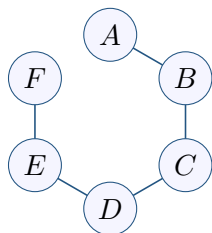
1 Connecting Six Routers

Suppose we have six routers. A direct cable could connect any pair, so all possible connections form the **complete graph** K_6 . There are $\binom{6}{2} = 15$ possible cables. Crossings in the drawing are not extra routers; only marked vertices count.

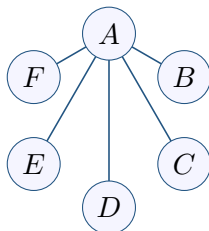


Do we need all fifteen cables? Our requirement is that every router can communicate with every other router, possibly by forwarding a message through intermediate routers. In graph language, the chosen network must be **connected**.

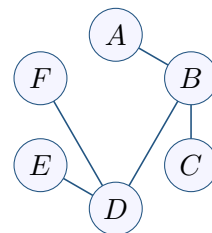
1.1 Several Configurations Work



A path



A star



A branching tree

Each configuration uses all six routers and only five cables. Each is connected, and each has no cycle. These are **spanning trees** of K_6 . They have different shapes but the same number of edges.

Connectivity alone does not force a tree: K_6 itself is connected. What distinguishes these examples is that every selected cable is necessary. If a connected network contains a cycle, we can remove an edge of that cycle and still travel between its endpoints along the rest of the cycle. Thus a network with no redundant edges should have no cycles.

2 Trees and Spanning Trees

2.1 Definitions and Edge Counts

Throughout these notes, graphs are finite, undirected, and simple. A **path** is a sequence of distinct vertices joined by edges. A graph is **connected** if a path joins every pair of vertices. A **cycle** is a closed path with at least three edges and no repeated vertices except its starting and ending vertex.

Definitions. A **tree** is a nonempty connected graph with no cycles. A **forest** is a graph with no cycles; each of its connected components is a tree. A **spanning tree** of $G = (V, E)$ is a subgraph $T = (V, F)$, with $F \subseteq E$, that is a tree. It must contain *every* vertex of G .

A single vertex with no edges is a tree. A disconnected collection of trees is a forest but is not a tree. In a tree with at least two vertices, a vertex of degree one is called a **leaf**.

Lemma 1. *Every tree with at least two vertices has at least two leaves.*

Proof. Choose a longest path, and consider one endpoint v . It already has one neighbor on the path. An additional neighbor on the path would create a cycle; an additional neighbor outside the path would let us extend the path. Both are impossible. Thus v has degree one. The same argument applies to the other endpoint. $\square$

Theorem 1. *A tree with n vertices has exactly $n - 1$ edges.*

Proof. We induct on n . When $n = 1$, there are no edges, as required. For $n \geq 2$, choose a leaf and remove it together with its one incident edge. The remaining graph has no cycles. It is still connected: a path between two remaining vertices cannot use a leaf as an intermediate vertex. Hence it is a tree on $n - 1$ vertices.

By induction, the smaller tree has $n - 2$ edges. Restoring the leaf restores exactly one edge, so the original tree has $n - 1$ edges. $\square$

Be careful about the converse. Having $n - 1$ edges alone does not make a graph a tree. A triangle plus an isolated vertex has four vertices and three edges, but is disconnected. The missing condition is important.

2.2 Unique Paths

Theorem 2. *For any two distinct vertices a, b of a tree, there is exactly one path from a to b .*

Proof. There is at least one path because the tree is connected. Suppose there were two different paths. Follow their common beginning from a until they first diverge. Follow one branch until it first meets the other path again; this must eventually happen because both end at b . The two portions between divergence and reunion form a cycle, contradicting the definition of a tree. $\square$

Conversely, if exactly one path joins every pair of distinct vertices, the graph is connected and cannot contain a cycle: the two directions around a cycle would give different paths between two of its vertices. Thus unique paths also characterize trees.

Two consequences will be useful later:

- Removing any edge of a tree disconnects it. Otherwise, another path between the edge's endpoints would remain.
- Adding a new edge to a tree creates exactly one cycle: the new edge together with the unique old path between its endpoints.

2.3 Growing a Spanning Tree

Given a connected graph, choose a starting vertex r . Let $S = \{r\}$ be the vertices reached so far and let $F = \emptyset$.

While S does not contain every vertex:

 Choose an edge $\{u, v\}$ with u in S and v outside S .

 Add $\{u, v\}$ to F , and add v to S .

Return the graph (V, F) .

Why can we always continue? If $S \neq V$, connectedness supplies a path from r to a vertex outside S . The first edge on that path that leaves S is an available choice.

Why is the answer a spanning tree? Initially the chosen graph is a one-vertex tree. Each step attaches one new vertex by one edge, preserving connectedness and creating no cycle. After $|V| - 1$ additions, it contains every vertex.

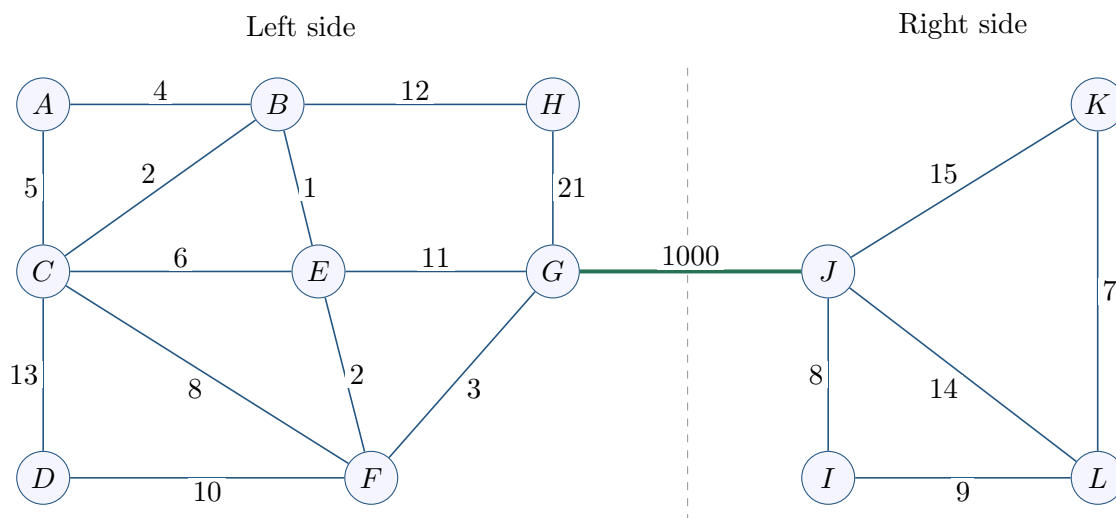
Breadth-first search or depth-first search implements this idea: keep the edge used to discover each new vertex. With adjacency lists, either search constructs a spanning tree in $O(|V| + |E|)$ time.

3 Minimum Spanning Trees

3.1 The Florida Hurricane Problem

Imagine that a hurricane has destroyed all the roads in a network of Florida communities. Surveyors estimate the cost of rebuilding each road. Money is tight, so we want to restore travel between every pair of communities while spending as little as possible.

Model a community by a vertex and a candidate road by an edge. Give edge e a positive weight $w(e)$ equal to its reconstruction cost. The drawing below shows the classroom road network. Vertex labels are added for reference; the positions are schematic.



A least-cost connected solution contains no cycle. If it did, removing one road from that cycle would preserve connectivity and, since costs are positive, strictly reduce the total cost. Therefore a least-cost solution is a spanning tree.

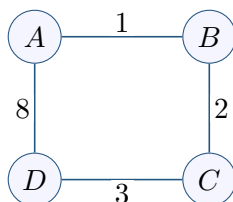
Minimum spanning tree. For a connected weighted graph $G = (V, E, w)$, a **minimum spanning tree** (MST) is a spanning tree T minimizing

$$\text{cost}(T) = \sum_{e \in E(T)} w(e).$$

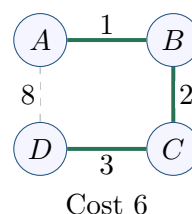
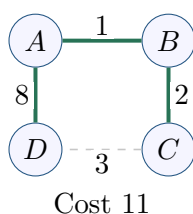
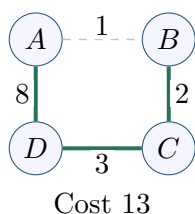
All spanning trees have $|V|-1$ edges, but their costs can be very different. An MST minimizes the *sum of edge weights*; it need not give a shortest route between each pair of vertices.

3.2 A Spanning Tree Need Not Be Minimum

Consider four communities arranged in a square. There are four available roads, with the indicated costs.



Every spanning tree of this cycle is obtained by deleting one edge. Here are three choices; solid edges are rebuilt and dashed edges are omitted.



Omitted edge	Rebuilt edges	Total cost
AB	BC, CD, DA	$2 + 3 + 8 = 13$
CD	AB, BC, DA	$1 + 2 + 8 = 11$
DA	AB, BC, CD	$1 + 2 + 3 = 6$

The first two networks are spanning trees, but they are not minimum. Deleting BC instead would give cost $1 + 3 + 8 = 12$. Thus the MST has cost 6, obtained by omitting the most expensive edge of the cycle.

This small example suggests two useful questions. When can we certify that an edge is safe to *include*? When can we certify that an edge should be *excluded*? Cuts answer the first question, and cycles answer the second.

3.3 The Cut Property and the Cycle Property

A **cut** is a partition $(S, V \setminus S)$ of the vertices into two nonempty sets. An edge **crosses** the cut if its endpoints are on opposite sides. Every spanning tree must contain a crossing edge, or its two sides would not be connected.

Theorem 3 (Cut property). *A minimum-weight edge crossing a cut belongs to some MST. If it is the unique minimum-weight crossing edge, it belongs to every MST.*

Proof. Let e be a cheapest crossing edge and let T be an MST. If $e \in T$, there is nothing to prove. Otherwise, add e to T . This creates a cycle. The remainder of that cycle joins opposite sides of the cut, so it contains another crossing edge f .

Replace f by e . The result $T' = T - f + e$ is a spanning tree and

$$\text{cost}(T') = \text{cost}(T) - w(f) + w(e) \leq \text{cost}(T).$$

Thus T' is also minimum and contains e . If e is uniquely cheapest, then $w(e) < w(f)$, contradicting the minimality of any MST that omitted e . Hence every MST must contain it. $\square$

Theorem 4 (Cycle property). *A unique maximum-weight edge on a cycle belongs to no MST. More generally, a maximum-weight edge on a cycle can be omitted from some MST.*

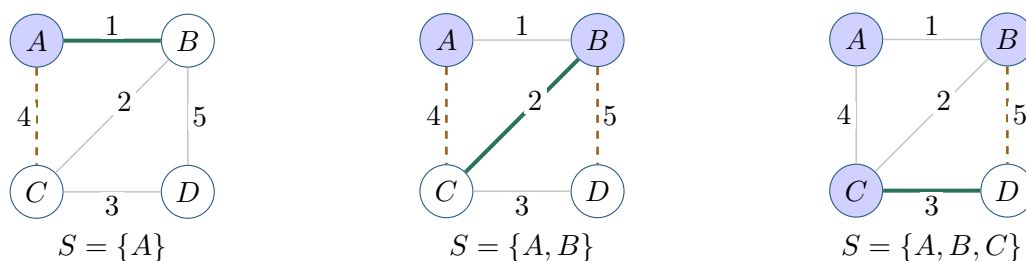
Proof. Let e be a maximum-weight edge on a cycle. If an MST T already omits e , the general claim holds. Otherwise, deleting e splits T into two components. The rest of the original cycle supplies a route between these components, so it contains another crossing edge f .

The graph $T - e + f$ is a spanning tree and has cost at most that of T , because $w(f) \leq w(e)$. Thus an MST omitting e exists. If e is uniquely heaviest, then $w(f) < w(e)$, so any tree containing e can be improved. In that case no MST contains e . $\square$

Both proofs use an **exchange argument**: make one local replacement, verify that a spanning tree remains, and compare the two costs. The words “unique minimum” and “unique maximum” matter when weights tie.

3.4 Three Cuts That Determine an MST

Use the following five-edge graph. It is a separate example from the four-road square on the preceding pages. In each drawing, shaded vertices form S , dashed edges cross the cut, and the thick solid edge is the unique cheapest crossing edge.



Cut	Crossing edges and weights	Forced edge
$\{A\} \mid \{B, C, D\}$	$AB : 1, AC : 4$	AB
$\{A, B\} \mid \{C, D\}$	$AC : 4, BC : 2, BD : 5$	BC
$\{A, B, C\} \mid \{D\}$	$BD : 5, CD : 3$	CD

Every MST must contain AB , BC , and CD . These three edges already form a spanning tree, so the MST is

$$T = \{AB, BC, CD\}, \quad \text{cost}(T) = 1 + 2 + 3 = 6.$$

An edge need not be cheapest in the whole graph to be forced: CD is only required to be cheapest across its particular cut.

3.4.1 See the Exchange Directly

The tree $T_0 = \{AC, BC, CD\}$ has cost $4 + 2 + 3 = 9$. For the first cut, AB is the cheapest crossing edge. Adding it creates cycle $A - B - C - A$. Remove AC , the other crossing edge on that cycle. The new tree costs $9 - 4 + 1 = 6$.

3.4.2 What the Property Does Not Say

For the cut $\{A, C\} \mid \{B, D\}$, the crossing edges have weights $AB : 1$, $BC : 2$, and $CD : 3$. Although AB is cheapest, the MST contains *all three*. The cut property does not forbid the other crossing edges; a tree may cross a cut several times.

If we change only $w(AC)$ from 4 to 2, the second cut has a tie. Both $\{AB, AC, CD\}$ and $\{AB, BC, CD\}$ are MSTs of cost 6. Neither tied edge is forced by that cut.

3.5 Kruskal's Greedy Algorithm

Kruskal builds a forest, considering roads in increasing order of cost. It takes an edge exactly when the edge connects two different components. The choice is **greedy**: among the remaining edges, try the cheapest one first.

```
Kruskal(G):
  F = empty set
  Sort the edges by nondecreasing weight.
  For each edge {u,v}, in this order:
    If u and v are in different components of (V,F):
      Add {u,v} to F.
  If F contains |V| - 1 edges:
    Break out of the loop.
  Return (V,F).
```

The input graph is assumed connected. Adding an edge between different components preserves acyclicity and reduces the number of components by one. An edge whose endpoints are already connected would close a cycle, so we skip it. Ties may be processed in any order.

3.5.1 A Trace on the Four-Vertex Example

Edge	Decision	Components afterward
Start	No selected edges	$\{A\}, \{B\}, \{C\}, \{D\}$
$AB : 1$	Add	$\{A, B\}, \{C\}, \{D\}$
$BC : 2$	Add	$\{A, B, C\}, \{D\}$
$CD : 3$	Add; stop	$\{A, B, C, D\}$

We stop after selecting $|V| - 1 = 3$ edges. If we continued scanning, $AC : 4$ and $BD : 5$ would both be skipped because their endpoints are already connected. The former closes cycle $A - B - C - A$; the latter closes cycle $B - C - D - B$.

3.5.2 A Trace with a Rejected Edge Before Completion

On vertices A, B, C, D , instead use edges $AB : 1$, $BC : 2$, $AC : 3$, $CD : 4$, and $AD : 5$. Kruskal adds AB , adds BC , rejects AC because it closes a triangle, and then adds CD . The resulting MST has cost $1 + 2 + 4 = 7$.

The cycle test is a connectivity test: “Are these endpoints already in the same component?” We do not need to list or enumerate cycles.

3.6 Why Kruskal Works

Showing that Kruskal produces a spanning tree is not enough; we must show that its cost is minimum. The key invariant is

Some MST contains all edges selected so far.

Initially no edges have been selected, so the invariant is true. Suppose Kruskal next accepts an edge $e = \{u, v\}$, and let S be the current component containing u . The edge e crosses the cut $(S, V \setminus S)$, and no previously selected edge crosses this cut.

The edge e is a cheapest crossing edge. If a strictly cheaper crossing edge existed, it would have been processed earlier. Its endpoints are in different components now, so they were also in different components then: components only merge. Kruskal would have accepted it, contradicting that its endpoints are still separated by the current cut.

By the invariant, choose an MST T containing all previously selected edges. If $e \in T$, the invariant persists. Otherwise, apply the cut exchange proof: add e and remove another crossing edge f from the resulting cycle. Since no previously selected edge crosses the cut, f is not one of them. The new tree is still minimum and contains *all* our selected edges, including e .

At termination the selected edges form a spanning tree. They are contained in an MST, and both trees have $|V| - 1$ edges, so they are the same tree. Therefore Kruskal returns an MST.

3.6.1 A Consequence: Distinct Weights Give a Unique MST

Theorem 5. *If all edge weights are distinct, a connected graph has exactly one MST.*

Proof. Let T be an MST and $e \in T$. Removing e splits T into two components. For any other edge f crossing this cut, $T - e + f$ is a spanning tree. Minimality gives $w(e) \leq w(f)$, and distinctness gives $w(e) < w(f)$. Thus e is uniquely cheapest across this cut and, by the cut property, belongs to every MST. This holds for every edge of T , so every MST contains all of T and hence equals T . $\square$

Distinct weights are sufficient, not necessary. A triangle with weights 1, 1, 2 also has a unique MST: it uses the two edges of weight 1.

3.7 Union–Find and Running Time (Advanced extra)

Let $n = |V|$ and $m = |E|$. Kruskal needs to keep track of a partition of the vertices into connected components. A **union–find**, or **disjoint-set**, data structure supports two operations:

- $\text{Find}(v)$ returns a representative of the set containing v . Two vertices are connected exactly when their representatives agree.
- $\text{Union}(u, v)$ merges the two sets containing u and v .

Initially each vertex is in its own set. For each edge $\{u, v\}$, compare $\text{Find}(u)$ with $\text{Find}(v)$. If they are different, accept the edge and merge the sets.

3.7.1 Why These Operations Can Be Fast

Represent each set by a rooted tree of parent pointers. The root is its representative. **Union by size** attaches the smaller tree under the larger tree. Without any other optimization, this keeps parent chains of length $O(\log n)$: each time a vertex gains an extra ancestor, its set at least doubles in size.

Path compression makes future searches faster. During a find, redirect the visited vertices to point directly to the root. Together with union by size or rank, it gives an amortized $O(\log^* n)$ bound per operation. “Amortized” bounds the total work of a sequence; it does not promise that each individual operation is equally cheap.

Here $\log^* n$ is the number of times we apply $\log_2$ until the value is at most 1. For example,

$$16 \mapsto 4 \mapsto 2 \mapsto 1, \quad \log^* 16 = 3.$$

It grows extremely slowly. A sharper bound uses the inverse Ackermann function $\alpha(n)$, but its definition is not needed here.

3.7.2 Separate Sorting from Connectivity Checks

Part of Kruskal	Running time
Sort arbitrary edge weights	$O(m \log m) = O(m \log n)$
Scan with union by size alone	$O(m \log n)$
Scan with size/rank and path compression	$O(m \log^* n)$

For a connected simple graph, $m \leq n(n-1)/2$, so $\log m = O(\log n)$. The full comparison-based algorithm therefore remains $O(m \log n)$: sorting dominates. **If the edges are already sorted**, the total work is $O(m \log^* n)$ (more sharply, $O(m\alpha(n))$). This is the precise sense in which union–find improves the bound.

4 Huffman Coding

Suppose a message contains ten A's, five B's, two C's, and three D's: 20 characters in total. One straightforward binary code is

$$A = 00, \quad B = 01, \quad C = 10, \quad D = 11.$$

This uses two bits per character, for a total of $20 \cdot 2 = 40$ bits. Since A occurs more often than the other symbols, perhaps we should give A a shorter codeword.

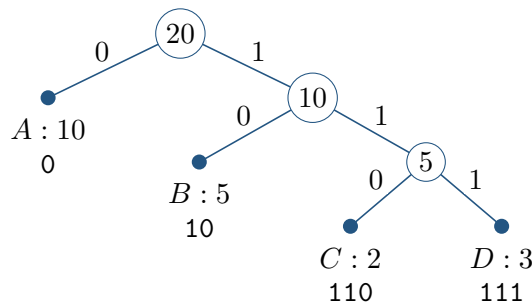
A **rooted tree** has a distinguished vertex, its root. A vertex's children are its neighbors farther from the root; a leaf has no children. A **binary coding tree** puts symbols at leaves and labels the two edges leaving each internal vertex 0 and 1. A symbol's codeword is the sequence of labels on its root-to-leaf path. Its length is the leaf's depth. We use 0 for a left edge and 1 for a right edge.

4.1 Constructing the Four-Symbol Code

Start with leaves of weights $A : 10$, $B : 5$, $C : 2$, and $D : 3$. Merge the two smallest current weights, replacing them by their sum:

$$C : 2 + D : 3 \longrightarrow 5, \quad B : 5 + (CD) : 5 \longrightarrow 10, \quad A : 10 + (BCD) : 10 \longrightarrow 20.$$

Each merge creates a new parent with the two chosen trees as children.



The code is $A = 0$, $B = 10$, $C = 110$, and $D = 111$. Its total length is

$$10(1) + 5(2) + 2(3) + 3(3) = \boxed{35 \text{ bits}}.$$

A random character from this message has expected codeword length

$$L = \frac{10}{20}(1) + \frac{5}{20}(2) + \frac{2}{20}(3) + \frac{3}{20}(3) = \boxed{1.75 \text{ bits per character}}.$$

The payload is five bits shorter than the fixed-length encoding. Both comparisons assume the decoder knows the codebook; transmitting the codebook adds an overhead of its own.

4.2 The Huffman Algorithm

For a finite alphabet with at least two symbols, give each symbol a positive weight $f(s)$, either a count or a probability. Scaling all weights by the same positive constant does not change the choices.

```
Huffman(symbols and weights):
  Make one weighted leaf for each symbol.
  While more than one tree remains:
    Remove two trees with the smallest root weights.
    Make a new parent with these trees as children.
    Give it the sum of their weights.
    Insert the new tree among the remaining trees.
  Label left edges 0 and right edges 1.
  Read each symbol's codeword from its root-to-leaf path.
```

A priority queue can keep track of the current smallest weights. A merged tree goes back into consideration immediately. Ties can be broken arbitrarily; swapping children changes bits but not lengths. For k symbols there are $k - 1$ merges. A binary heap implements the construction in $O(k \log k)$ time.

4.3 Encoding and Decoding the First Example

To encode a message, concatenate its symbols' codewords. For example,

$$\text{CABAD} \mapsto 110 \mid 0 \mid 10 \mid 0 \mid 111 = 1100100111.$$

The vertical bars explain the boundaries; they are not transmitted.

To decode, start at the root. Read one bit and follow the corresponding edge. At a leaf, output that leaf's symbol and return to the root. Continue until all bits have been read.

Bit positions	Bits followed from root	Leaf	Output so far
1–3	110	C	C
4	0	A	CA
5–6	10	B	CAB
7	0	A	CABA
8–10	111	D	CABAD

A leaf supplies the stopping rule for one symbol. We do not need a separator, a fixed block size, or advance knowledge of the next symbol.

4.4 A Six-Symbol Example

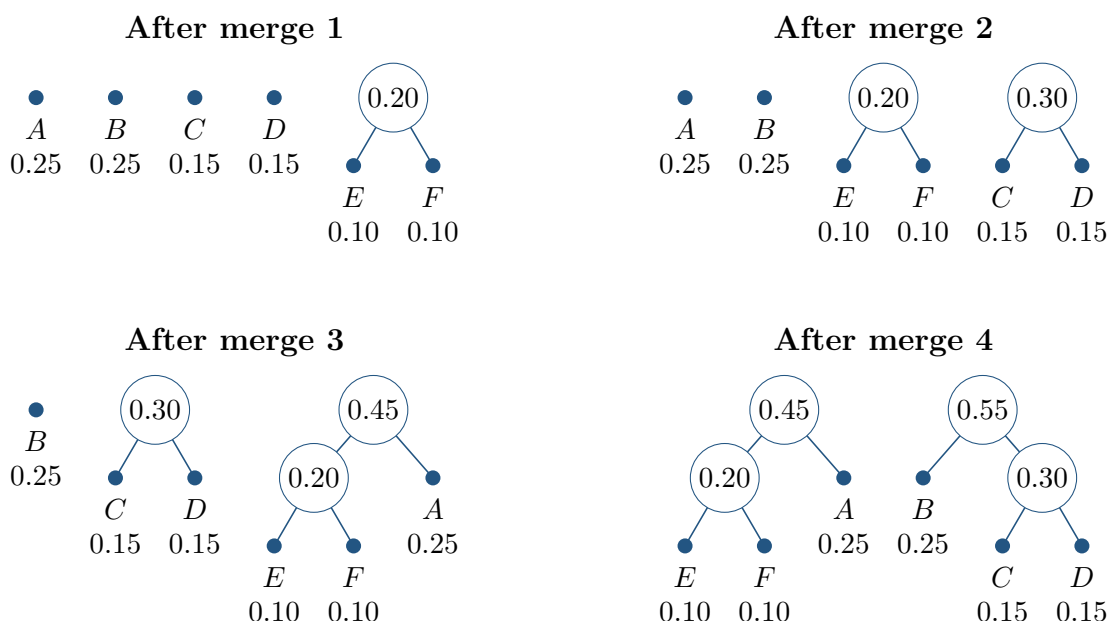
Now suppose the symbol probabilities are

s	A	B	C	D	E	F
$p(s)$	0.25	0.25	0.15	0.15	0.10	0.10

A fixed-length binary code needs three bits per symbol, because two bits offer only four distinct codewords. Huffman builds a variable-length code. Break the tie between A and B by choosing A at merge 3.

Merge	Chosen trees	New parent weight
1	E and F	$0.10 + 0.10 = 0.20$
2	C and D	$0.15 + 0.15 = 0.30$
3	(EF) and A	$0.20 + 0.25 = 0.45$
4	B and (CD)	$0.25 + 0.30 = 0.55$
5	(EFA) and (BCD)	$0.45 + 0.55 = 1.00$

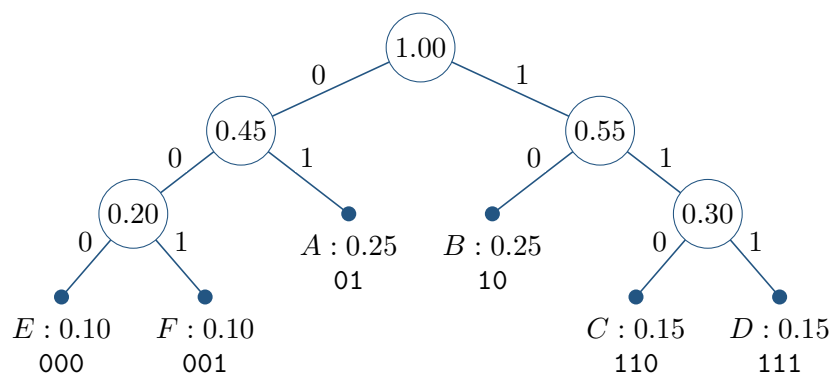
The pictures show the *whole forest* after each of the first four merges. Each parent is labeled by the sum of all leaf probabilities below it. The new trees remain eligible for the next merge.



At merge 3, the available root weights are 0.20, 0.25, 0.25, 0.30. We choose 0.20 and 0.25, not the two smallest *original* probabilities. At merge 4, the remaining weights are 0.25, 0.30, 0.45.

4.4.1 The Final Code and a Bit-by-Bit Decoding

The fifth merge joins the last two trees into the root. Label left edges 0 and right edges 1.



s	A	B	C	D	E	F
$c(s)$	01	10	110	111	000	001
$\ell(s)$	2	2	3	3	3	3

Thus the expected number of bits per symbol is

$$L = 0.25(2) + 0.25(2) + 0.15(3) + 0.15(3) + 0.10(3) + 0.10(3) = \boxed{2.5}.$$

For example, $\text{FACE} \mapsto 001 \mid 01 \mid 110 \mid 000 = 00101110000$.

Position	Bit	Path since last return to root	Action / output
1	0	0	Internal vertex; continue
2	0	00	Internal vertex; continue
3	1	001	Emit F; return to root
4	0	0	Internal vertex; continue
5	1	01	Emit A; return to root
6	1	1	Internal vertex; continue
7	1	11	Internal vertex; continue
8	0	110	Emit C; return to root
9	0	0	Internal vertex; continue
10	0	00	Internal vertex; continue
11	0	000	Emit E; return to root

The decoded message is **FACE**. There are no leftover bits.

4.5 Correctness and Unique Decoding (Optional)

An arbitrary variable-length code need not decode uniquely. For example, $A = 0$, $B = 1$, $C = 01$ makes 01 ambiguous: it could represent AB or C .

A code is **prefix-free** if no codeword is a prefix of a different codeword. The tree construction guarantees this property.

Lemma 2. *Huffman's algorithm produces a binary tree with exactly one leaf for each symbol. Its codewords are distinct, nonempty, and prefix-free.*

Proof. Initially the trees are disjoint one-symbol leaves. Each merge joins two disjoint trees under a new root, preserving every existing symbol leaf and creating no cycle. Every new internal vertex has two children. After $k - 1$ merges there is one tree containing all k symbol leaves.

The edge labels determine a unique route from the root, so distinct leaves have distinct codewords. Since $k \geq 2$, no symbol is at the root, and all codewords are nonempty. If one codeword were a prefix of another, its leaf would lie on the route to a different leaf. A leaf has no children, so this is impossible. $\square$

Theorem 6 (Correctness of the decoder). *For every finite message M , decoding its encoding recovers M : $\text{Dec}(\text{Enc}(M)) = M$.*

Proof. The empty message encodes to the empty bitstring and decodes to the empty message. For a nonempty message $M = s_1 s_2 \cdots s_r$, its encoding begins with the path labels for s_1 . Reading those bits follows that path. No earlier vertex on the path is a leaf, so the decoder emits exactly s_1 when it reaches the leaf. It then returns to the root. The remaining input encodes $s_2 \cdots s_r$. Induction on r completes the proof. $\square$

Theorem 7 (Unique decoding). *A bitstring produced by this code represents exactly one message.*

Proof. Existence follows from decoder correctness. For uniqueness, suppose two messages encode to the same bitstring. If either message is empty, nonempty codewords force the other to be empty as well. Otherwise, their first codewords both begin that string, so the shorter would be a prefix of the longer. Prefix-freeness forces them to be equal, hence their first symbols agree. Remove this common codeword and repeat. Neither message can have extra symbols after the other ends, because codewords are nonempty. Thus the two messages are identical. $\square$