



Trees, Heaps and Huffman Coding

CEN207 Data Structures — Week 4

Asst. Prof. Dr. Uğur CORUH · Fall 2026–2027

Today's plan (3 hours)

Hour	Topic
1	Trees, vocabulary, shapes Anim 1–2 · traversals: pre/in/post/iterative/level Anim 3–7
2	Array repr. Anim 8 · binary heap: insert/extract/build/sort Anim 9–12
3	Priority queue Anim 13 · variants Anim 14–16 · Huffman Anim 17–18

Learning outcomes: LO.1 (explain fundamental data structures) · LO.2 (analyze complexity) · LO.7 (choose the right structure)

This week's concepts — where

Concept	Where
Tree vocabulary, binary tree shapes	Sections 1–2
Traversals (pre/in/post/iterative/level)	Section 3
Array representation of a tree	Section 4
Binary heap, heap sort	Section 5
Priority queue, variants, Huffman	Sections 6–8

How the code examples work

- Every idea has a complete **C** and **Java** program
- C: `gcc -std=c11 -Wall -Wextra -o /tmp/x file.c && /tmp/x`
- Java: `javac -d /tmp/j File.java && java -cp /tmp/j File`
- Sources: `code/week-04/c/` and `code/week-04/java/`
- Each program's expected output is in the week notes

Recap — Weeks 1–2: pointers and lists

- `malloc` / `free` : one box, one owner, `NULL` when empty
- A linked-list node: a value plus one `next` pointer
- A tree node: a value plus **two** pointers, `left` / `right`
- A list ends at `NULL` ; a tree **leaf** has both `NULL`

Recap — Week 3: stack, queue, recursion

- Recursion **is** a stack: a call pushes, a return pops
- Today: the *same* traversal, using our **own** stack
- The queue (FIFO) returns unchanged, for level order
- Week 3's structures now simply hold tree nodes

Map of the week — at a glance

Trees & traversals	Heaps & Huffman
Vocabulary, five shapes	Binary heap, heap sort
Five traversal orders	Priority queue, variants
Array representation	Huffman coding

1. Why Trees? Vocabulary and History

A question to start

A file manager: one home folder, folders inside folders, files inside those. One root, unlimited branching, and no folder is ever its own ancestor.

A short history

- **1857** — Cayley counts trees while counting molecules
- **1968** — Knuth fixes the vocabulary in *TAOCP* Vol. 1
- Root, leaf, degree, level, height — his terms, still used
- One of the oldest shapes in mathematics, borrowed by CS

Intuition — an upside-down tree

- The **root** is the trunk — drawn at the **top**
- Everything branches **downward** from the root
- A **leaf** has no children — the end of a branch
- Yes, upside down: every CS diagram draws it this way

Vocabulary (1/3)

Term	Meaning
Root	The one node with no parent
Parent / child	Edge from A down to B : A is parent, B child
Sibling	Two nodes sharing the same parent
Leaf	A node with no children — degree 0

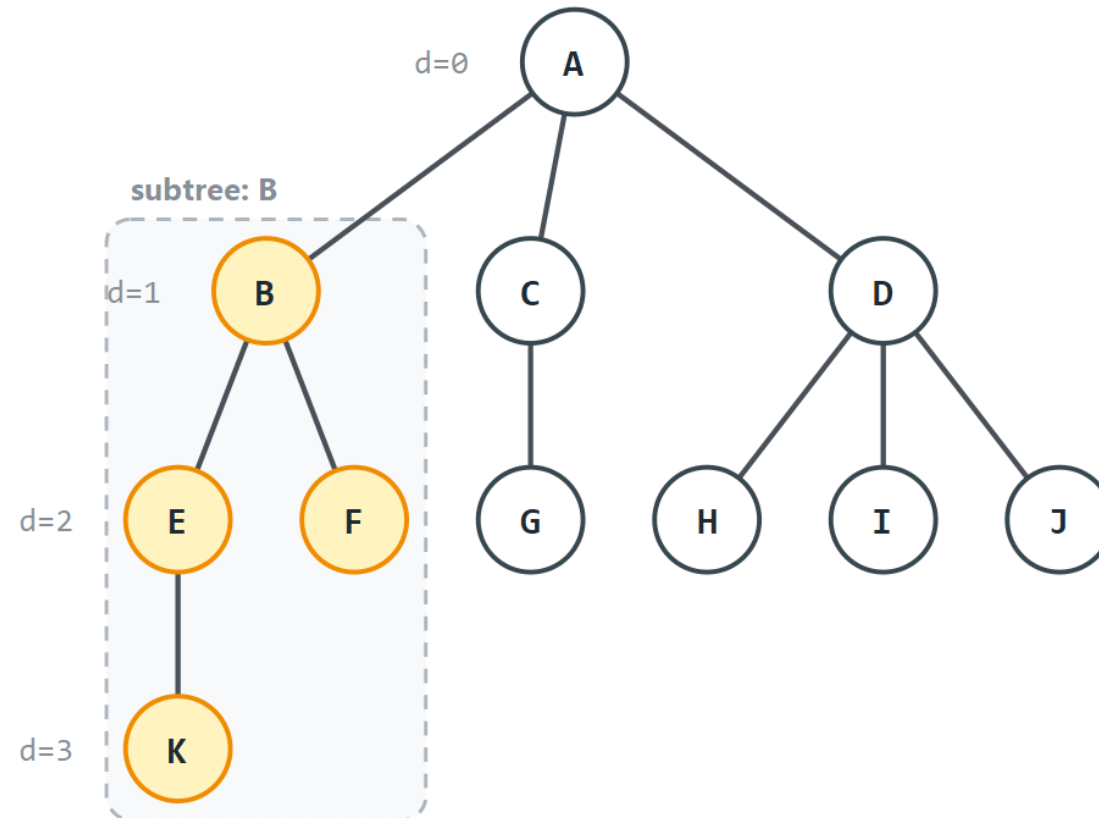
Vocabulary (2/3)

Term	Meaning
Internal node	A node with at least one child
Edge	A parent-child connection; n nodes, $n-1$ edges
Degree (of a node)	Its number of children
Depth (of a node)	Edges from the root down to it

Vocabulary (3/3)

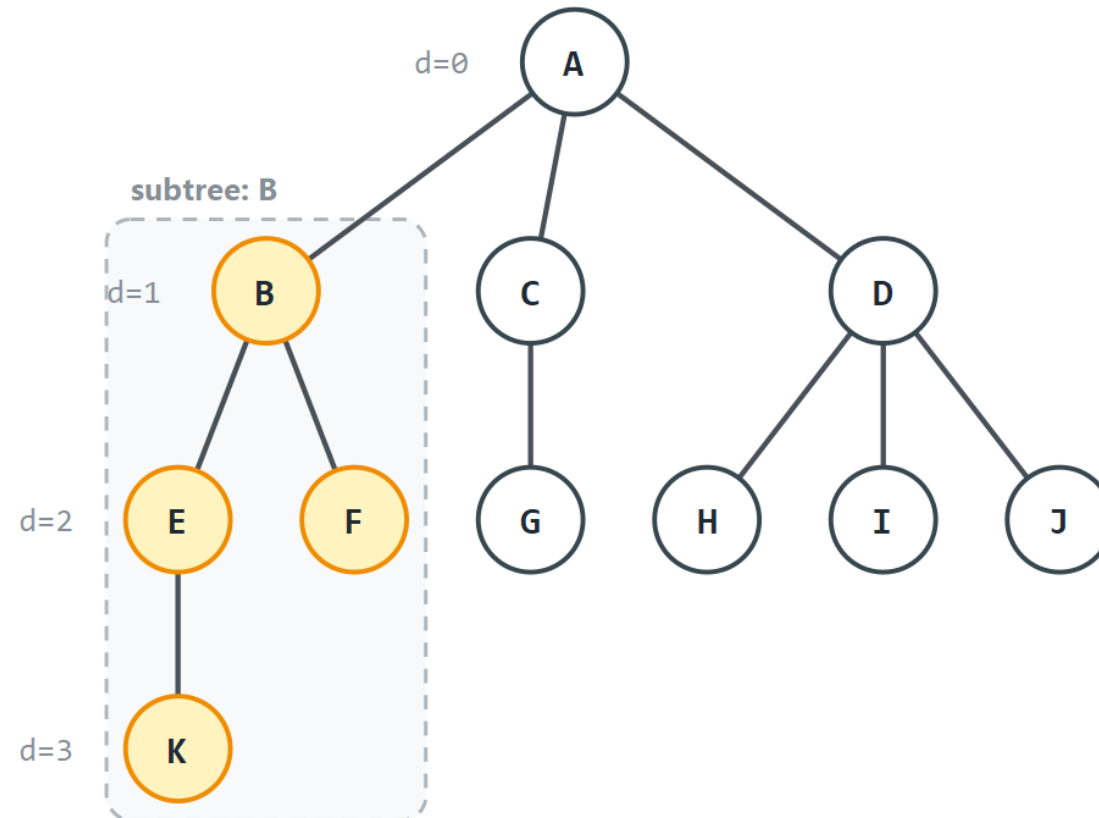
Term	Meaning
Height (of a node)	Edges on its longest path down to a leaf
Height (of the tree)	The root's height
Subtree	A node plus everything below it

Tree vocabulary, one term at a time



A SUBTREE: node B together with everything below it (B, E, K, F), framed in dashes.

Edge case — a star: the root has 10 children



A SUBTREE: node B together with everything below it (B, E, K, F), framed in dashes.

Code — the tree node

```
typedef struct Node {  
    char label[4];  
    struct Node *children[MAX_CHILDREN];  
    int child_count;    /* degree of this node */  
    int depth;  
    struct Node *parent;  
} Node;
```

Code — height() (bottom-up, recursive)

```
int height(Node *n) {  
    if (n->child_count == 0)  
        return 0;          /* a leaf: height 0 */  
    int best = -1;  
    for (int i = 0; i < n->child_count; i++) {  
        int h = height(n->children[i]);  
        if (h > best) best = h;  
    }  
    return best + 1;        /* 1 + tallest child */  
}
```

Code — compute_depths() (top-down, BFS)

```
void compute_depths(Node *root) {
    Node *queue[MAX_NODES];
    int front = 0, rear = 0;
    root->depth = 0;
    queue[rear++] = root;
    while (front < rear) {
        Node *cur = queue[front++];
        for (int i = 0; i < cur->child_count; i++) {
            Node *ch = cur->children[i];
            ch->depth = cur->depth + 1;
            queue[rear++] = ch;
        }
    }
}
```

Complexity

- `height` : every node visited once — $O(n)$
- `compute_depths` : every node visited once — $O(n)$
- Neither depends on the tree's **shape**
- A thin chain costs the same as a bushy tree, same n

Common mistakes

- Confusing **depth** (from the root down) with **height** (down to a leaf)
- Forgetting a leaf's height is 0, not 1
- Assuming every tree is **binary** — a general node can have any number of children

Mini-quiz

In the 18-node example, **Q1** has children **S1** and **S2**. What is **Q1**'s **degree**? What is **S1**'s **depth**, given **Q1**'s depth is 1?

Answer

Q1's degree is 2 (two children). S1 is a child of Q1, so its depth is Q1's depth + 1 = 2.

2. Binary Trees: Shapes and Node Counts

A question to start

Restrict every node to **at most two** children, left and right. Why would giving up "any number of children" ever be worth it?

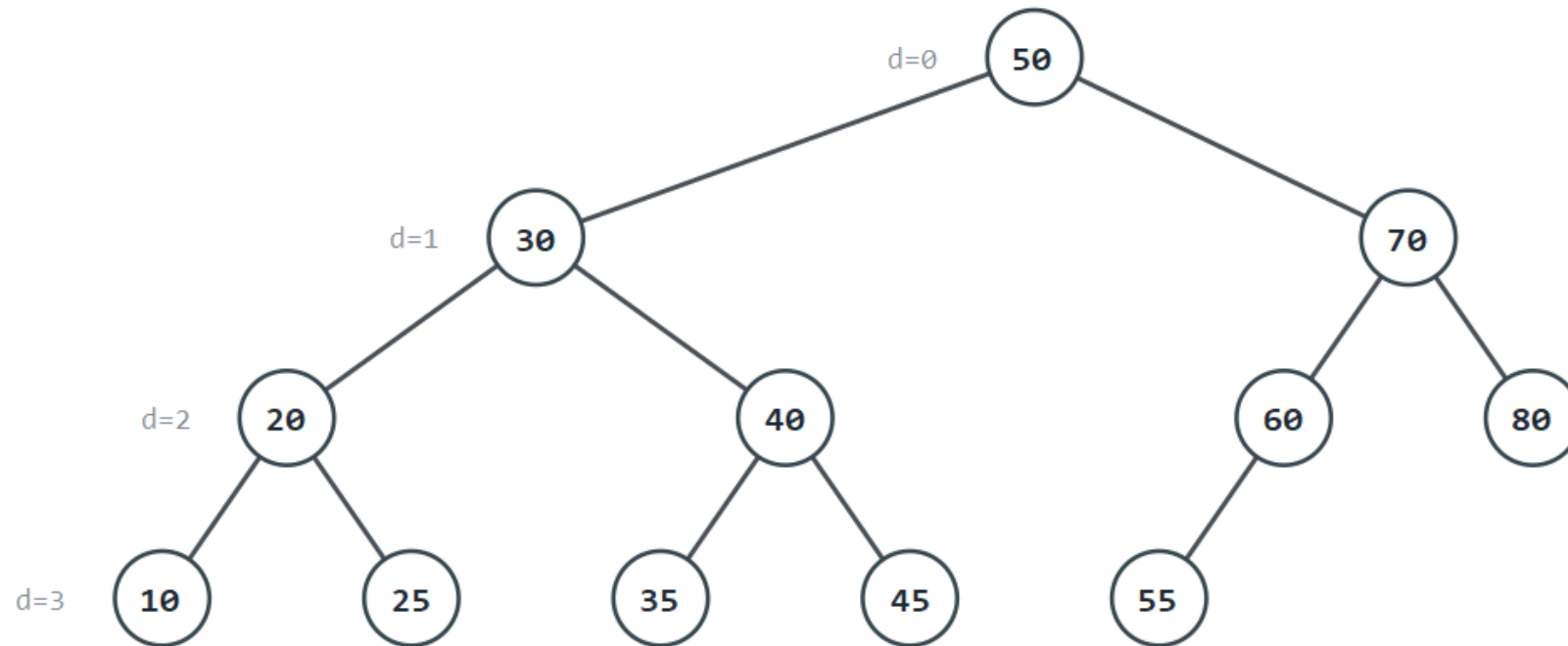
Code — the binary tree node

```
typedef struct Node {  
    int value;  
    struct Node *left, *right;  
} Node;
```

Five shapes, precisely defined

Shape	Definition
Full	Every node has 0 or 2 children, never 1
Complete	Every level full except the last, filled left-to-right
Perfect	Full and every leaf at the same depth
Degenerate	Every node has 0 or 1 children — a chain
Balanced	Left/right subtree heights differ by ≤ 1

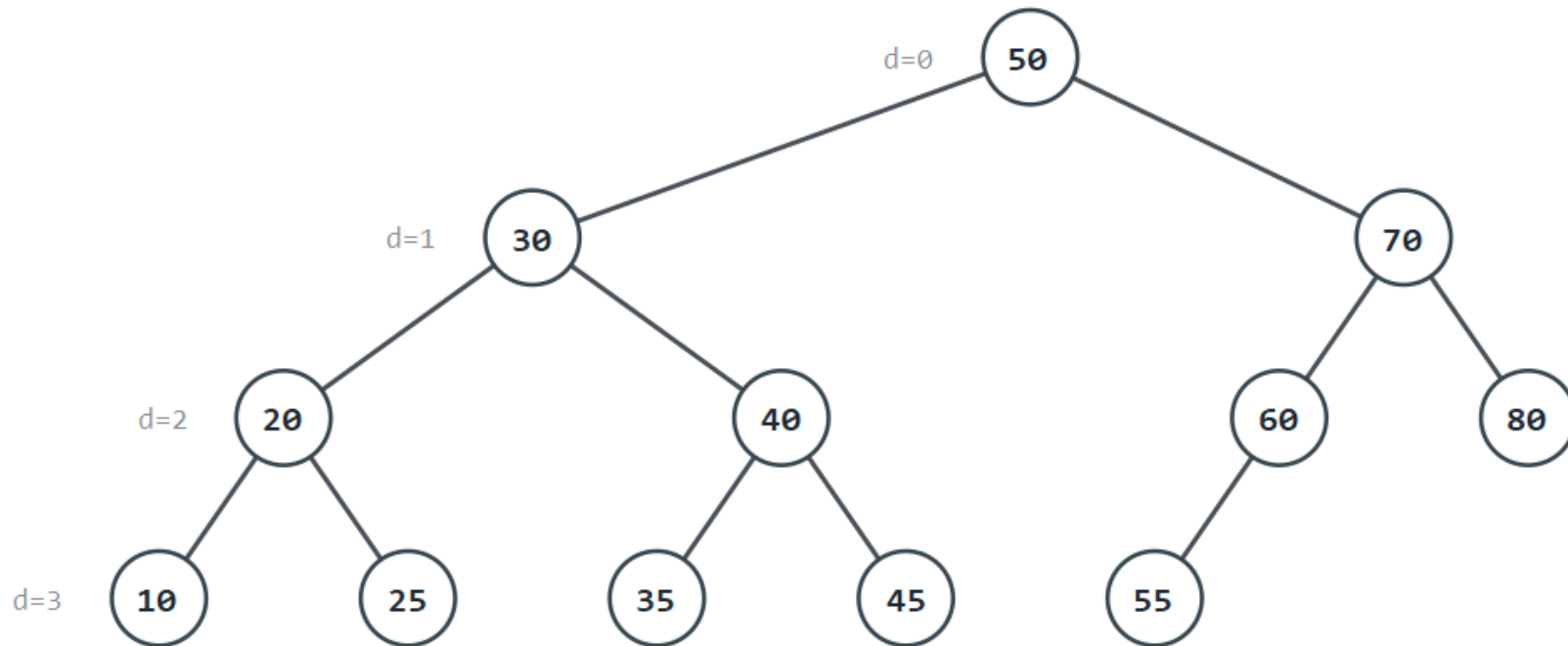
Binary tree shapes, one question at a time



**Summary — full: no, complete: yes, perfect: no.
degenerate: no, balanced: yes (12 nodes, height 3).**

Summary: 12 nodes, height 3. The five answers are shown above.

Edge case — a degenerate chain, 10 nodes



**Summary — full: no, complete: yes, perfect: no.
degenerate: no, balanced: yes (12 nodes, height 3).**

Summary: 12 nodes, height 3. The five answers are shown above.

How many nodes can fit?

- **Max at height h :** a perfect tree has $2^{h+1} - 1$ nodes
- Level k holds 2^k nodes; sum them through level h
- **Min height for n nodes:** $\lfloor \log_2 n \rfloor$, no arrangement beats it
- Each level holds at most twice the nodes of the one above

Why balance matters

- A **balanced** tree keeps height close to $\log_2 n$
- Root-to-leaf operations then cost **$O(\log n)$**
- A **degenerate** tree's height is $n - 1$, no better than a list
- Same node count, wildly different cost — shape decides

Code — is_complete (spotting gaps)

```
static bool is_complete(Node *root) {  
    Node *queue[MAX_NODES]; int front = 0, rear = 0;  
    queue[rear++] = root;  
    bool seen_gap = false;  
    while (front < rear) {  
        Node *n = queue[front++];  
        if (n == NULL) { seen_gap = true; continue; }  
        if (seen_gap) return false;  
        queue[rear++] = n->left;  
        queue[rear++] = n->right;  
    }  
    return true;  
}
```

Code — check_balance (one pass, not two)

```
static int check_balance(Node *n) {  
    if (n == NULL) return 0;  
    int hl = check_balance(n->left);  
    if (hl == -1) return -1;  
    int hr = check_balance(n->right);  
    if (hr == -1) return -1;  
    if (abs(hl - hr) > 1) return -1;  
    return 1 + (hl > hr ? hl : hr);  
}
```

Complexity

- All five shape checks are $O(n)$ — visit each node once
- `is_complete` needs $O(n)$ extra space, for its queue
- The other four need only $O(h)$ recursion-stack space

Common mistakes

- Thinking "complete" means every level is full
- Confusing **perfect** (complete + all leaves equal depth) with complete
- Calling `height()` twice per node — turns $O(n)$ into $O(n^2)$

Mini-quiz

A binary tree has height 3 and is **perfect**.
How many nodes does it have? How many
of them are leaves?

Answer

$2^4 - 1 = 15$ nodes total. The last level (level 3) holds $2^3 = 8$ leaves; the other 7 nodes are internal.

3. Traversals: Visiting Every Node Once

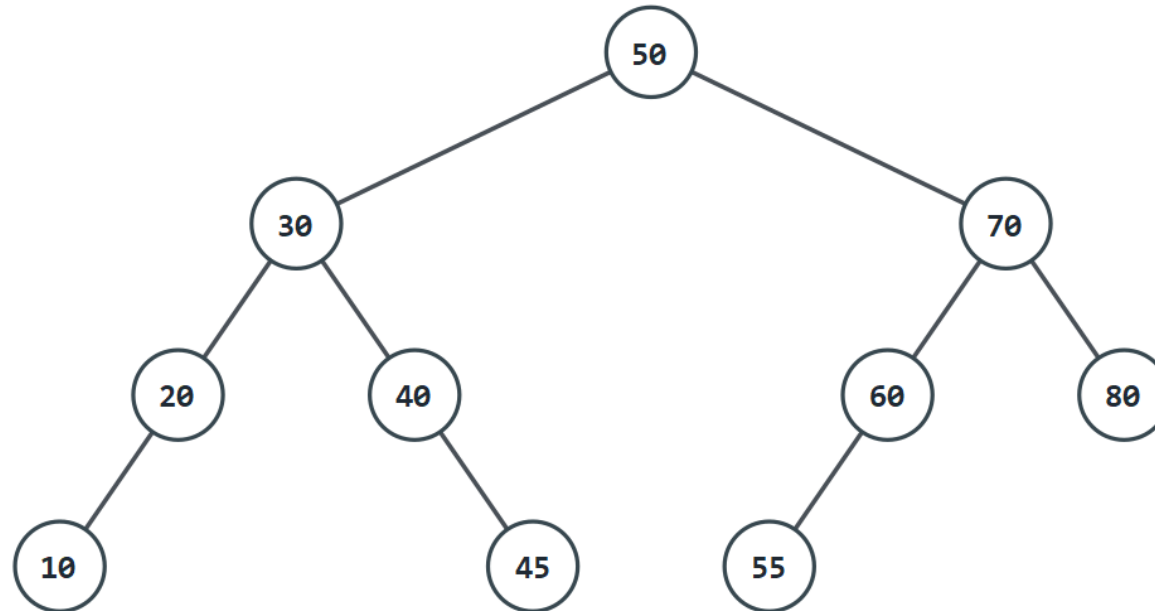
No single "natural" order

- A node has (up to) two children — a real choice exists
- Three recursive orders: **preorder**, **inorder**, **postorder**
- Two more without recursion: our own **stack**, and a **queue**
- Same tree, five orders, generally five different sequences

Preorder: visit, left, right

Visit the node **before** either child. The root is always visited **first** — exactly the order you need to **rebuild** a tree from scratch.

Preorder, step by step

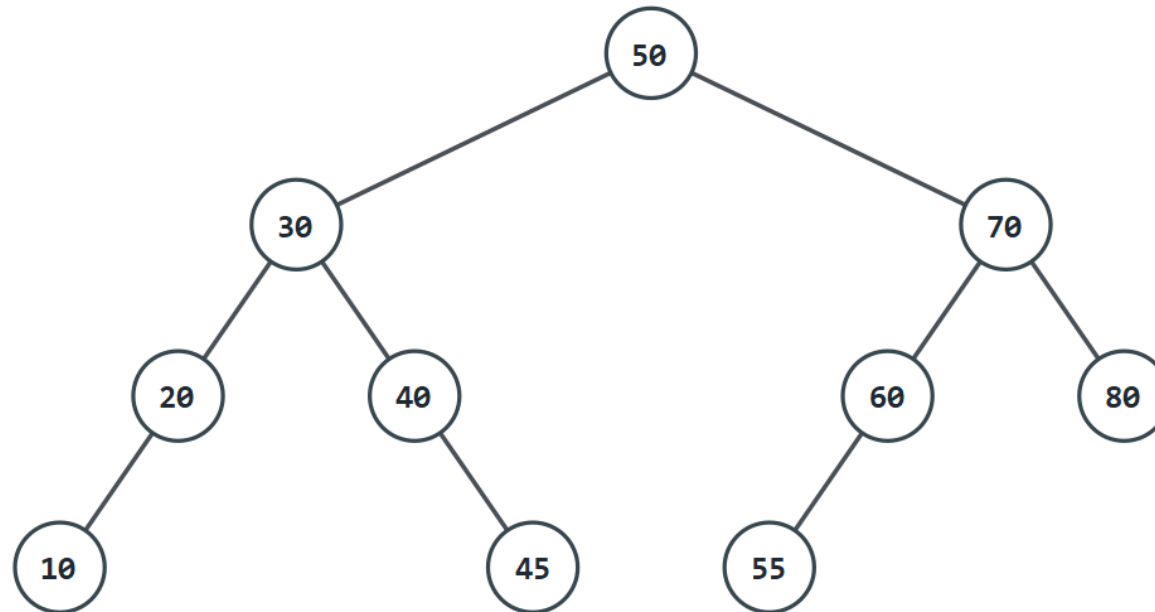


visit order (preorder):



Done. Full preorder sequence: 50, 30, 20, 10, 40, 45, 70, 60, 55, 80. The root is always visited FIRST -- preorder is ideal for rebuilding a copy of the tree (parent before children).

Edge case — a left-skewed chain



visit order (preorder):



Done. Full preorder sequence: 50, 30, 20, 10, 40, 45, 70, 60, 55, 80. The root is always visited FIRST -- preorder is ideal for rebuilding a copy of the tree (parent before children).

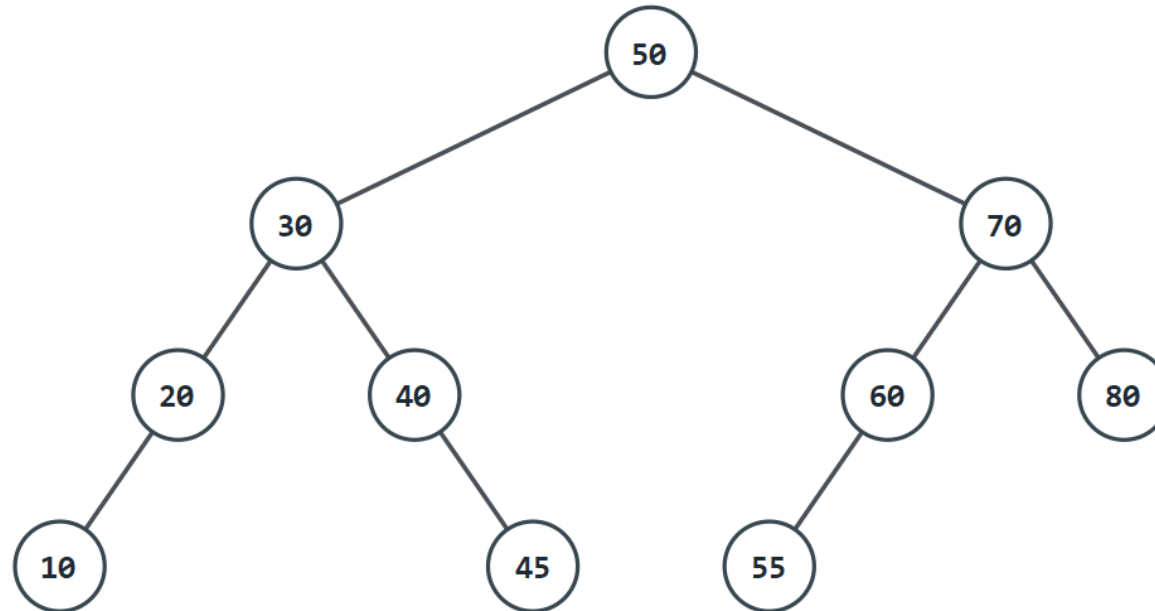
Code — preorder (recursive)

```
static void preorder(Node *node) {  
    if (node == NULL) return;  
    printf("visit %d\n", node->value);  
    visited[visited_count++] = node->value;  
    preorder(node->left);  
    preorder(node->right);  
}
```

Inorder: left, visit, right

Visit the node **between** its children. On a binary **search** tree, this visits every value in **sorted** order — previewed here, built later.

Inorder, step by step

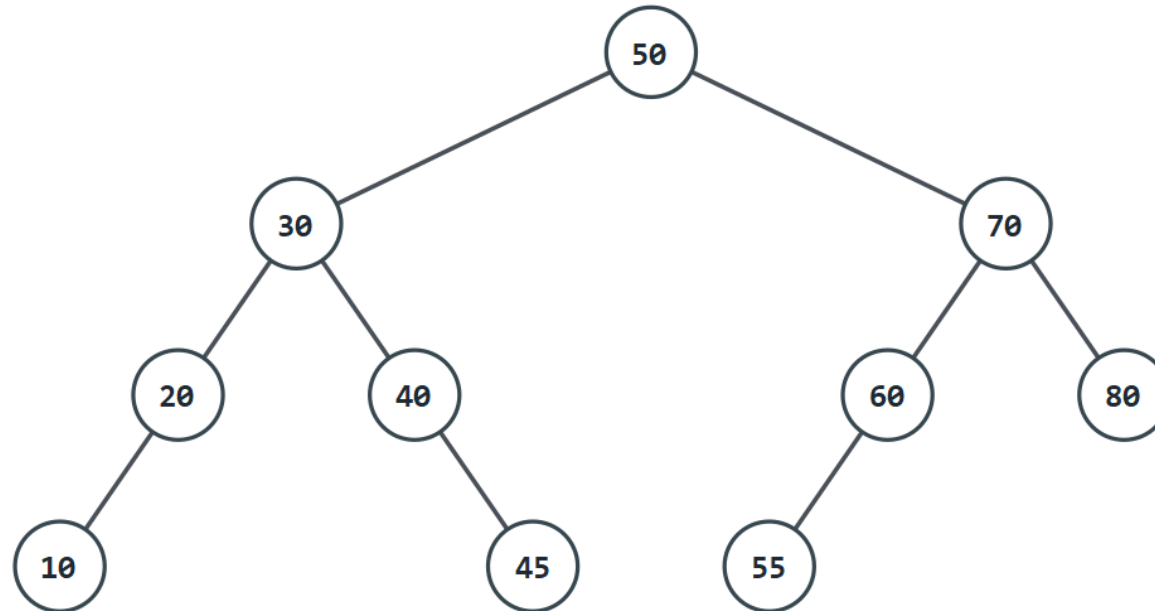


visit order (inorder):



Done. Full inorder sequence: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80. Every node of the tree is visited exactly once: $O(n)$.

Edge case — a right-skewed chain



visit order (inorder):



Done. Full inorder sequence: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80. Every node of the tree is visited exactly once: $O(n)$.

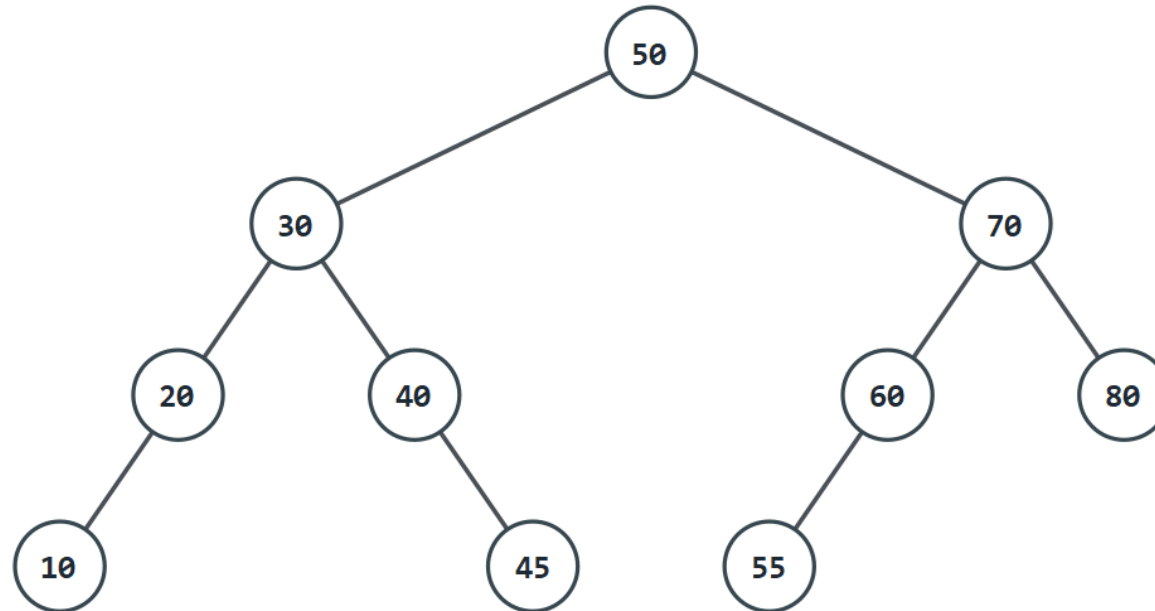
Code — inorder (recursive)

```
static void inorder(Node *node) {  
    if (node == NULL) return;  
    inorder(node->left);  
    printf("visit %d\n", node->value);  
    visited[visited_count++] = node->value;  
    inorder(node->right);  
}
```

Postorder: left, right, visit

Visit the node **after** both children. The root is always visited **last** — exactly the order you need to **delete** a tree safely.

Postorder, step by step

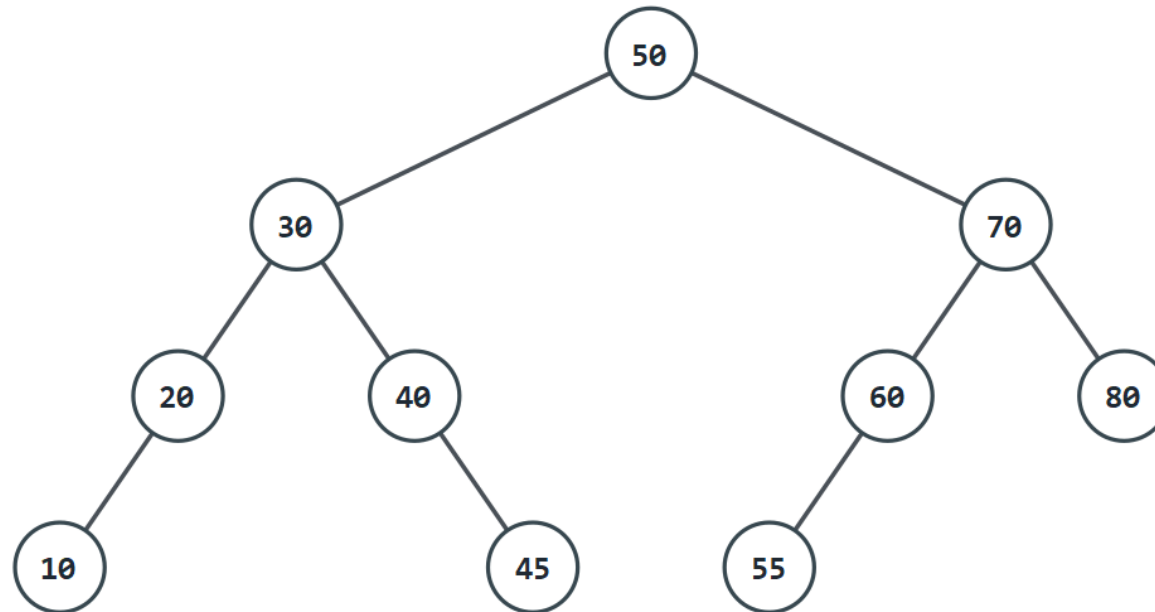


visit order (postorder):



Done. Full postorder sequence: 10, 20, 45, 40, 30, 55, 60, 80, 70, 50. The root is always visited LAST -- postorder is used to safely delete a tree (children before the parent).

Edge case — a right-skewed chain, reversed



visit order (postorder):



Done. Full postorder sequence: 10, 20, 45, 40, 30, 55, 60, 80, 70, 50. The root is always visited LAST -- postorder is used to safely delete a tree (children before the parent).

Code — postorder (recursive)

```
static void postorder(Node *node) {  
    if (node == NULL) return;  
    postorder(node->left);  
    postorder(node->right);  
    printf("visit %d\n", node->value);  
    visited[visited_count++] = node->value;  
}
```

Complexity — all three traversals

- $O(n)$ time: every node visited once, $O(1)$ work each
- $O(h)$ recursion-stack space — the tree's height
- $O(\log n)$ for a balanced tree, but $O(n)$ for a chain
- That worst case motivates section 3.4's explicit stack

Common mistakes

- Forgetting the base case — `node == NULL` must return
- Not resetting a shared visited-count between scenarios
- Assuming the three orders "mostly agree" — even one extra node makes them diverge
- Picking the wrong traversal: rebuild needs preorder, delete needs postorder

Mini-quiz

You must save a tree to a file so that reading the values back in and inserting each one in order rebuilds the exact same tree. Which order?

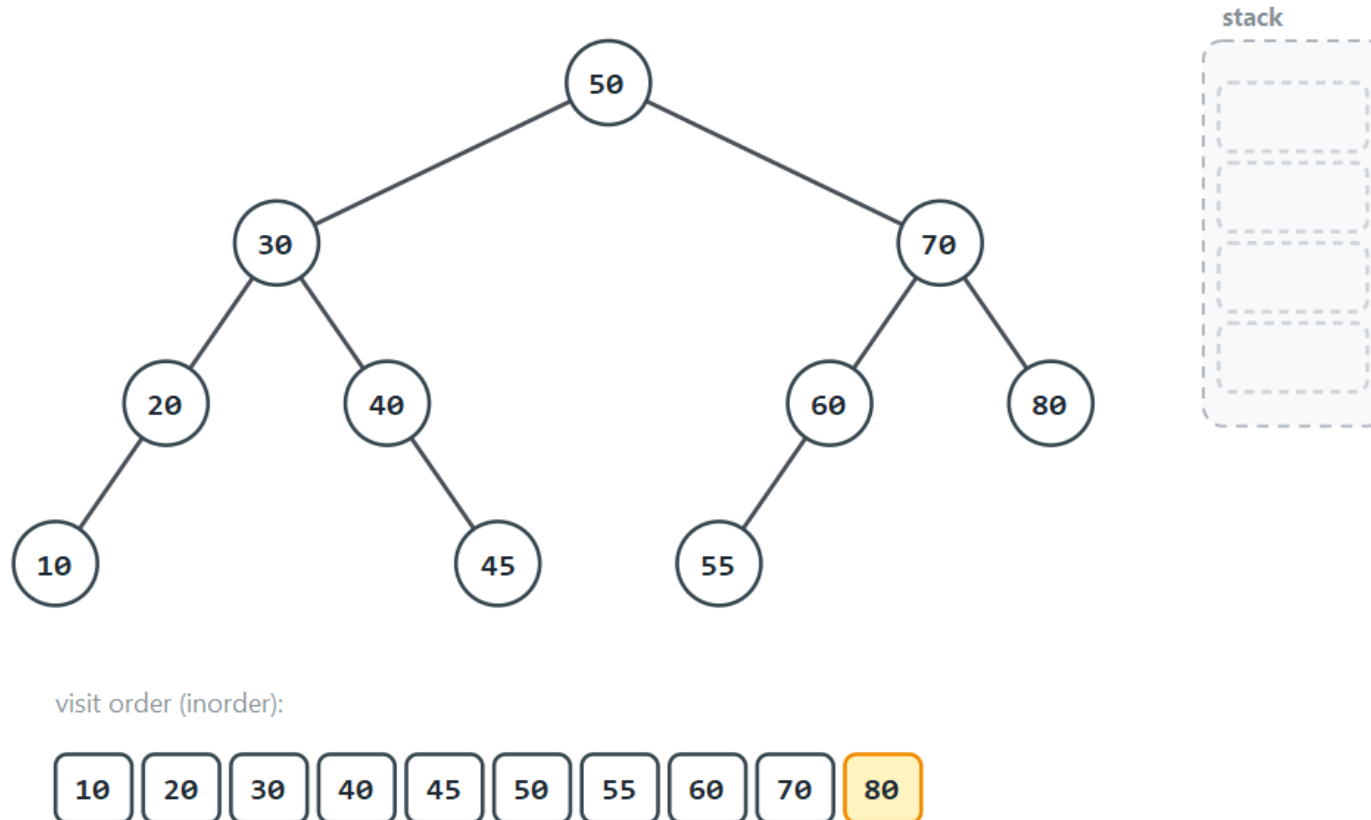
Answer

Preorder. The first value read back must be the root, and preorder is the only order that visits the root before either subtree.

Inorder without recursion

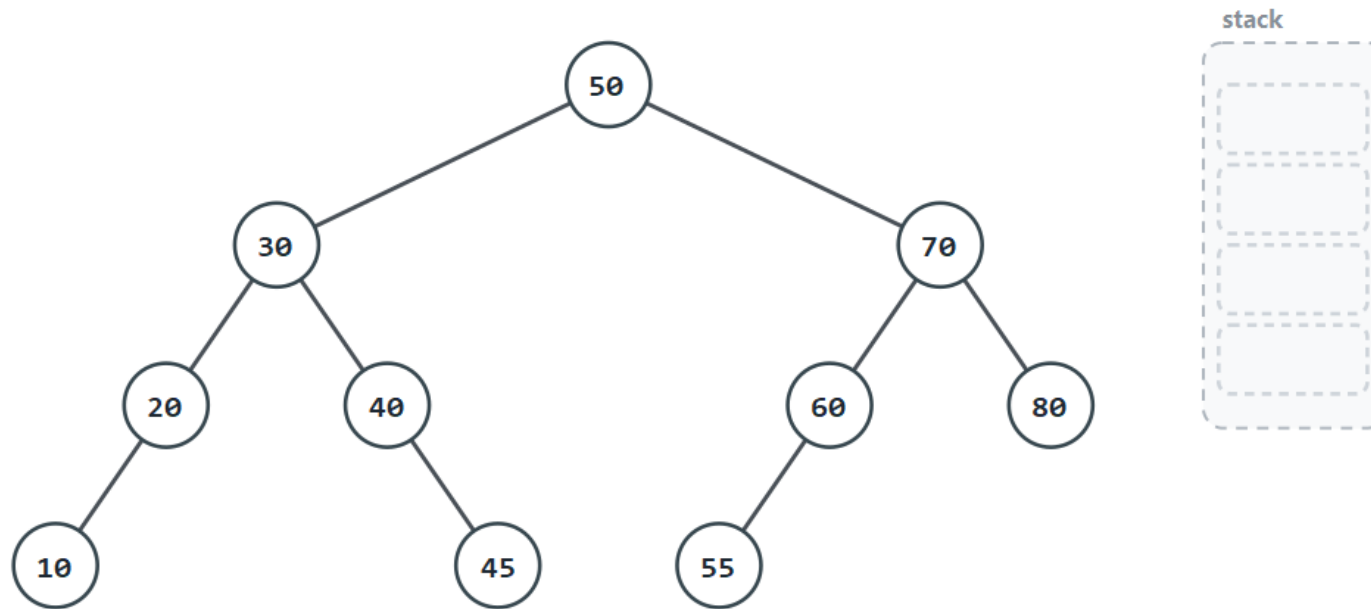
Every recursive traversal secretly uses the compiler's **call stack**. This time, no recursion at all — our **own** stack, from Week 3, holding `Node *` instead of `int`.

Iterative inorder, step by step



Done. Full sequence: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80 -- exactly the same as recursive inorder. The difference: we manage the stack by hand now, not the compiler.

Edge case — the stack reaches its deepest point



visit order (inorder):



Done. Full sequence: 10, 20, 30, 40, 45, 50, 55, 60, 70, 80 -- exactly the same as recursive inorder. The difference: we manage the stack by hand now, not the compiler.

Code — push / pop (our own stack)

```
static void push(Node *n) {  
    top = top + 1;  
    stack_data[top] = n;  
}  
static Node *pop(void) {  
    Node *n = stack_data[top];  
    top = top - 1;  
    return n;  
}
```

Code — the main loop

```
Node *cur = root;
while (cur != NULL || !is_empty()) {
    while (cur != NULL) {
        push(cur);
        cur = cur->left;
    }
    cur = pop();
    printf("visit %d\n", cur->value);
    cur = cur->right;
}
```

Complexity

- $O(n)$ time — every node pushed once, popped once
- $O(h)$ space — matches the recursive version exactly
- No memory saved; only the bookkeeping moved, to our own stack
- Benefit: an explicit stack can grow past a fixed recursion limit

Common mistakes

- Dropping either half of `cur != NULL || !is_empty()`
- Forgetting `cur = cur->right` after visiting a popped node
- Skip it, and every right subtree silently disappears

Mini-quiz

On a **perfect** tree of height h , what is the **maximum** number of nodes ever sitting on the stack at once during this traversal?

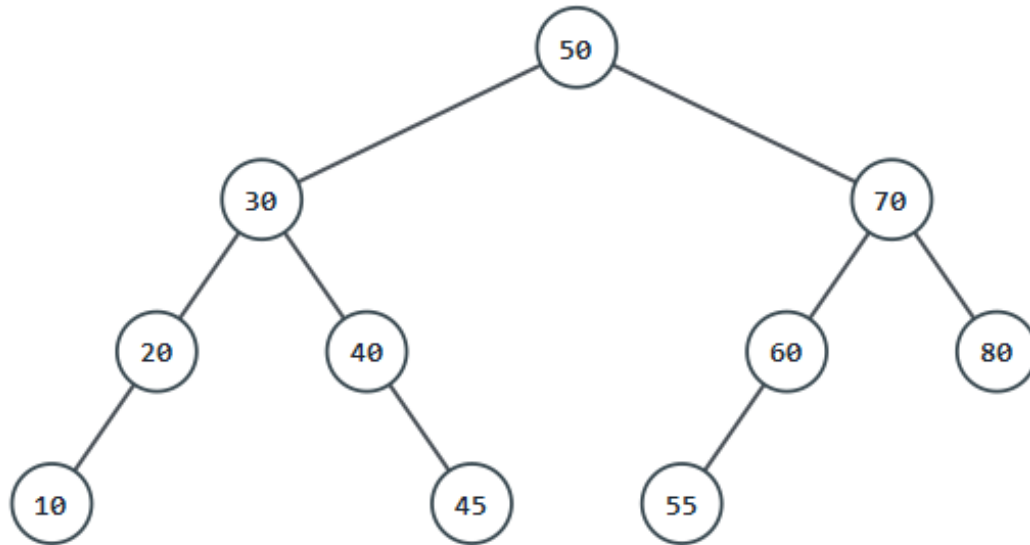
Answer

$h + 1$. The stack only ever holds the current left spine, and a perfect tree's longest left spine has $h + 1$ nodes.

Level order (breadth-first) with a queue

Every traversal so far goes **deep** before **wide**. Level order visits the root, then **every** node at depth 1, then depth 2 — needs a **queue**, not a stack.

Level order, step by step

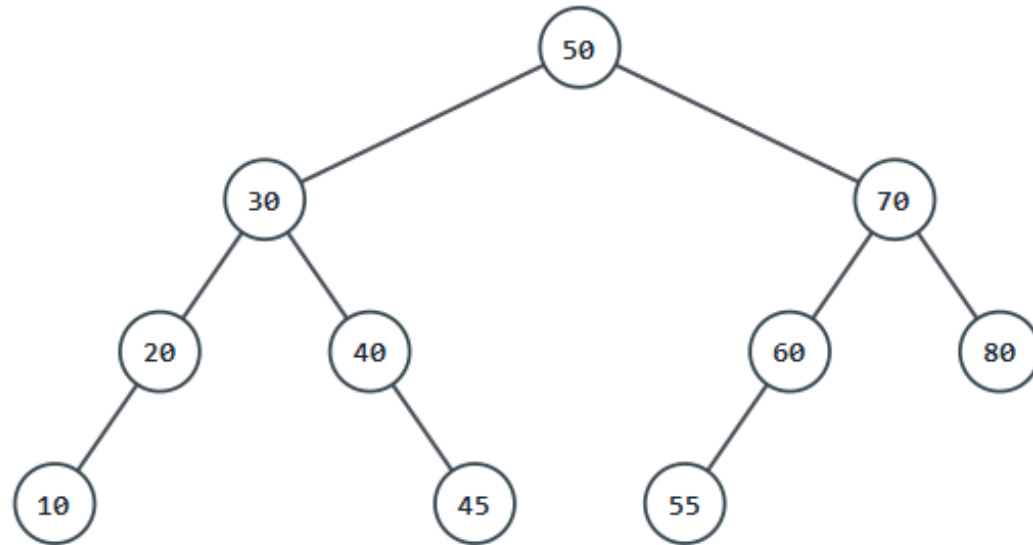


visit order (level-order):



The queue is empty, done. Full sequence: 50, 30, 70, 20, 40, 60, 80, 10, 45, 55 -- we finished each level left to right before moving one level down.

Edge case — the queue always holds one item



visit order (level-order):



The queue is empty, done. Full sequence: 50, 30, 70, 20, 40, 60, 80, 10, 45, 55 -- we finished each level left to right before moving one level down.

Code — enqueue / dequeue

```
static void enqueue(Node *n) {  
    rear = (rear + 1) % QUEUE_CAP;  
    queue_data[rear] = n;  
    count++;  
}  
static Node *dequeue(void) {  
    Node *n = queue_data[front];  
    front = (front + 1) % QUEUE_CAP;  
    count--;  
    return n;  
}
```

Code — the main loop

```
enqueue(root);  
while (!is_empty()) {  
    Node *cur = dequeue();  
    printf("visit %d\n", cur->value);  
    if (cur->left != NULL) enqueue(cur->left);  
    if (cur->right != NULL) enqueue(cur->right);  
}
```

Complexity

- $O(n)$ time — every node enqueued once, dequeued once
- $O(w)$ space, where w is the tree's maximum **width**
- w can be as large as $O(n)$ for a bushy, balanced tree
- Contrast with $O(h)$ space for every depth-first traversal

Common mistakes

- Swapping `dequeue` for `pop` "because it's basically the same" — that silently becomes depth-first
- Enqueuing `NULL` children by mistake — crashes on the next dequeue

Mini-quiz

Can two **different** binary trees share the exact same **level-order** sequence of values?
True or false?

Answer

True. A level-order sequence alone does not encode parent-child relationships once a level has any gaps.

4. A Complete Tree in an Array

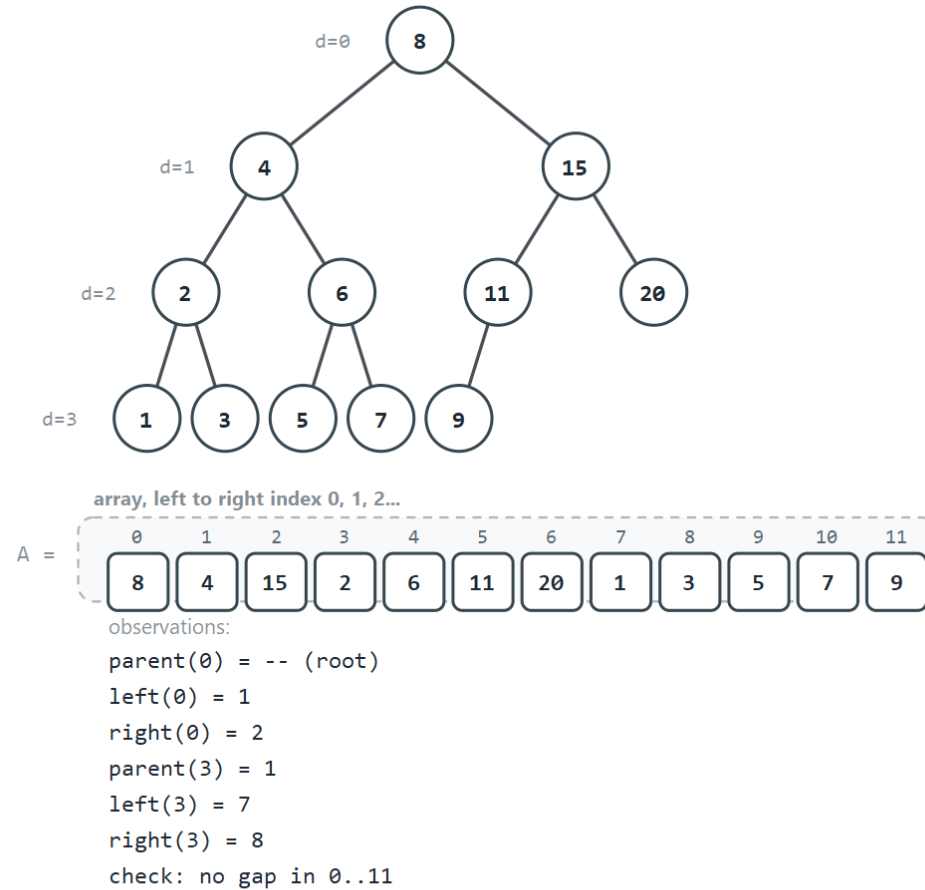
A question to start

Pointers cost memory, and finding a parent needs either a stored pointer or a search. For a **complete** tree, is there a cheaper way?

The index formulas

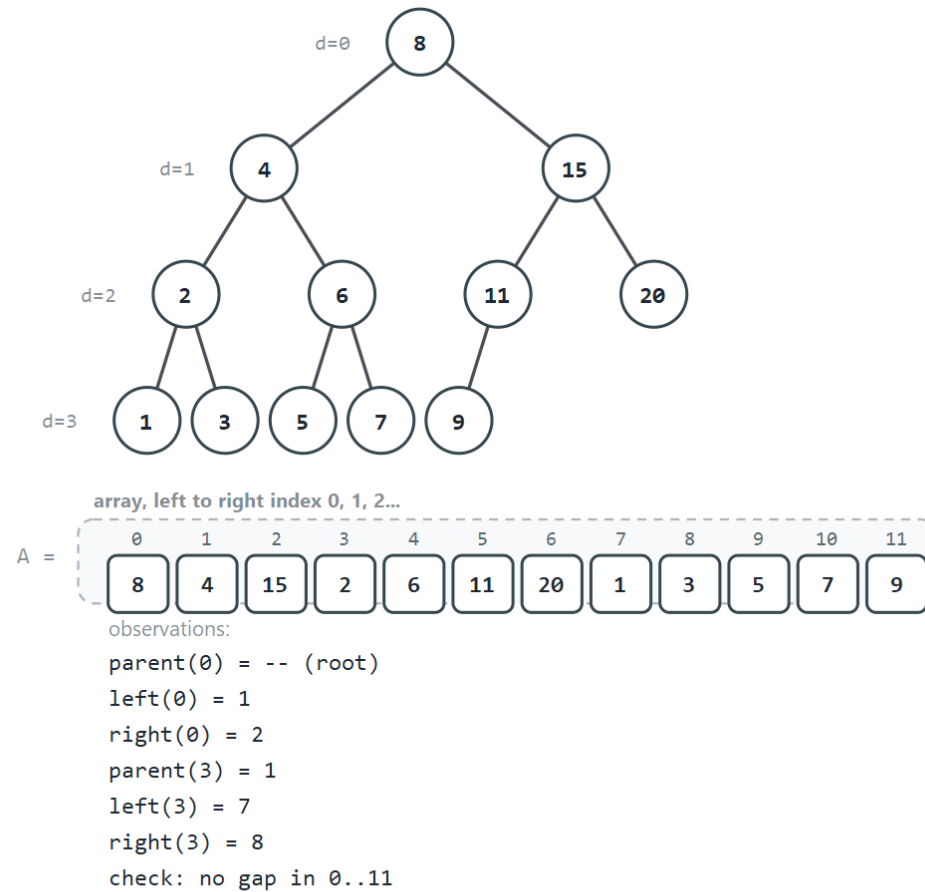
- Node at index i : **left child** at $2*i + 1$
- **Right child** at $2*i + 2$
- **Parent** at $(i - 1) / 2$ (integer division)
- No **left**, **right**, or **parent** field anywhere — just an array

The array representation, step by step



Result: 12 nodes, complete? YES. For a complete tree the array representation is perfect (no wasted cells); for a tree that is not complete, the array usually needs far more cells than the real node count.

Edge case — NOT complete: a gap



Result: 12 nodes, complete? YES. For a complete tree the array representation is perfect (no wasted cells); for a tree that is not complete, the array usually needs far more cells than the real node count.

Code — parent / left / right, is_complete

```
static int parent(int i) { return (i - 1) / 2; }
static int left(int i)   { return 2 * i + 1; }
static int right(int i)  { return 2 * i + 2; }

static bool is_complete(int arr[], int n, int last_real) {
    for (int i = 0; i <= last_real; i++)
        if (arr[i] == EMPTY) return false;
    return true;
}
```

Complexity

- `parent`, `left`, `right` : pure arithmetic — $O(1)$
- Cost is the same for 10 nodes or 10 million
- `is_complete` itself: $O(n)$, one scan of the array

Common mistakes

- Using these formulas on a tree that is **not** actually complete
- Trusting $(i - 1) / 2$ without checking your language's integer-division rules
- Forgetting the root ($i = 0$) has no parent — a special case

Mini-quiz

A complete binary tree, stored in an array.

The node at index 11 has two children.

At which indices? At which index is its parent?

Answer

Children at $2 \cdot 11 + 1 = 23$ and $2 \cdot 11 + 2 = 24$.

Parent at $(11 - 1) / 2 = 5$.

5. The Binary Heap

A question to start

An OS needs the most urgent of a hundred waiting processes, instantly, over and over. Sorting the whole list each time is wasteful. What is the cheapest structure that always keeps the single best item within reach?

A short history

- **1964** — J. W. J. Williams introduces the heap and **heap sort**, together
- **1964** — R. W. Floyd publishes an $O(n)$ way to **build** one
- The heap was invented specifically to make sorting fast

The heap property

- **Min-heap:** every parent $\leq$ both children — smallest at the root
- **Max-heap:** every parent $\geq$ both children — largest at the root
- **Not** a sorted structure — siblings have no required order
- Only every parent beats its own two children, nothing more

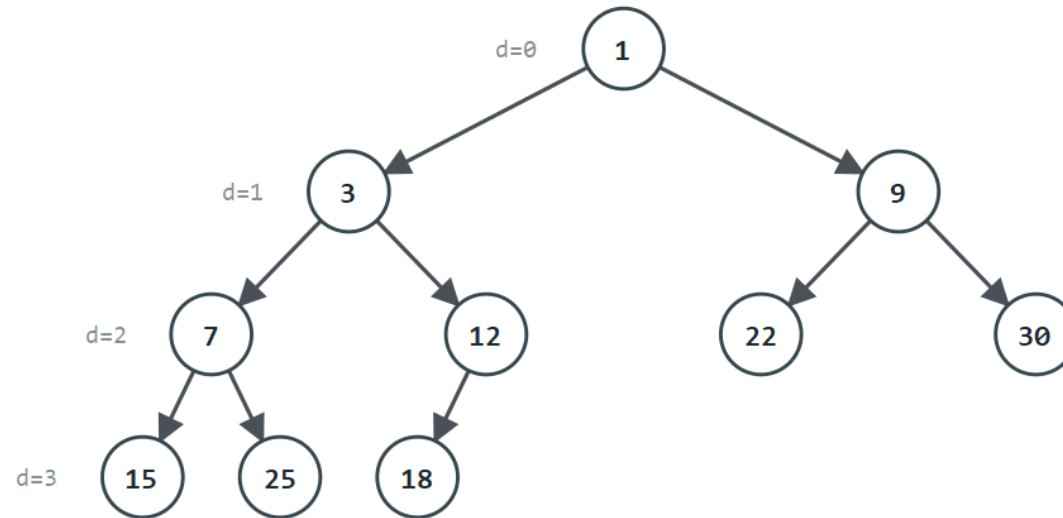
The heap ADT

Operation	What it does	Complexity
<code>peek()</code>	Returns the root without removing it	$O(1)$
<code>insert(x)</code>	Adds x , restores order by sift-up	$O(\log n)$
<code>extract()</code>	Removes the root, restores by sift-down	$O(\log n)$
<code>build_heap(arr)</code>	Turns any array into a heap, all at once	$O(n)$

Insertion: sift-up

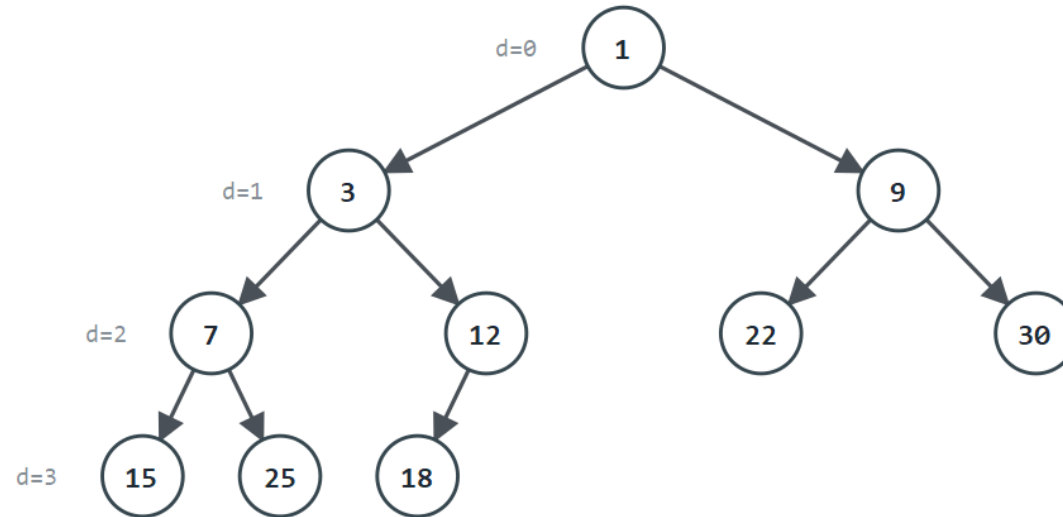
Place the new value in the next free slot — keeps the tree complete. Then repeatedly swap it with its **parent** while the heap property is violated. This is **sift-up**.

Insertion by sift-up, step by step



Done: final heap array [1, 3, 9, 7, 12, 22, 30, 15, 25, 18]. The root (index 0) always holds the best value: 1. Each insert does at most $O(\log n)$ comparisons, one per level of the tree.

Edge case — already in order, no sifting needed



heap[] =

0	1	2	3	4	5	6	7	8	9
1	3	9	7	12	22	30	15	25	18

Done: final heap array [1, 3, 9, 7, 12, 22, 30, 15, 25, 18]. The root (index 0) always holds the best value: 1. Each insert does at most $O(\log n)$ comparisons, one per level of the tree.

Code — insert() / sift-up

```
void insert(int value) {  
    heap[size] = value;  
    int i = size;  
    size++;  
    while (i > 0) {  
        int parent = (i - 1) / 2;  
        if (!better(heap[i], heap[parent]))  
            break;  
        int tmp = heap[parent];  
        heap[parent] = heap[i];  
        heap[i] = tmp;  
        i = parent;  
    }  
}
```

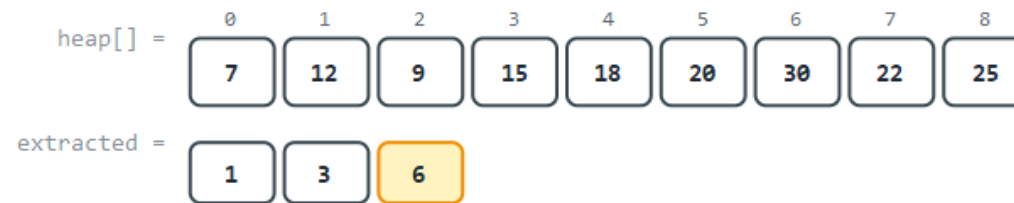
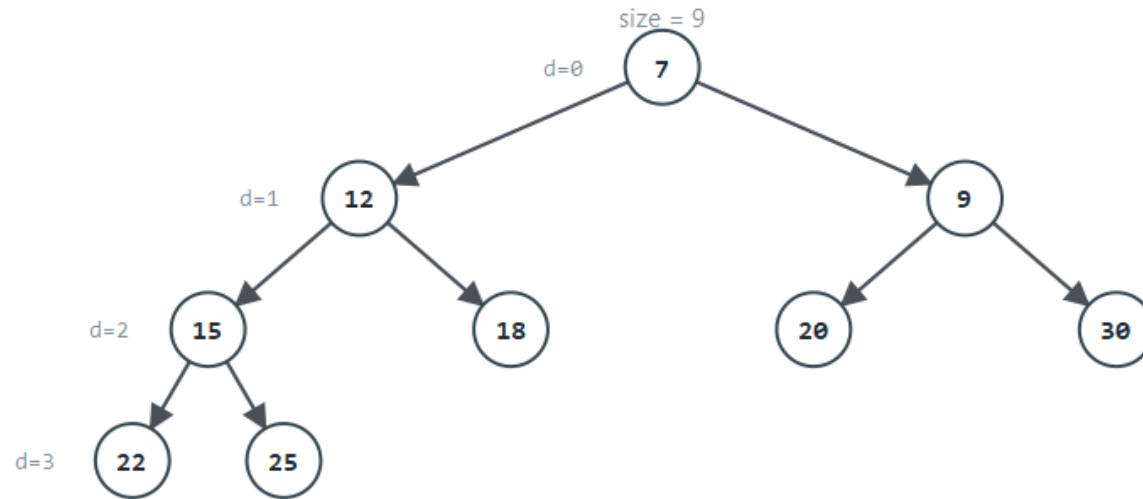
Complexity

- `insert` : at most one swap per **level** — **$O(\log n)$**
- $\log n$ is the height of a complete tree with n nodes
- `peek` (reading `heap[0]`): **$O(1)$**

Extraction: sift-down

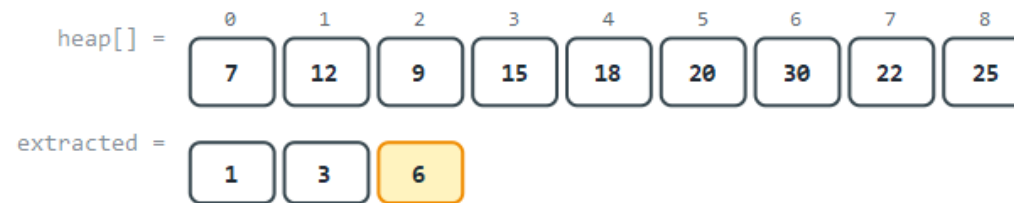
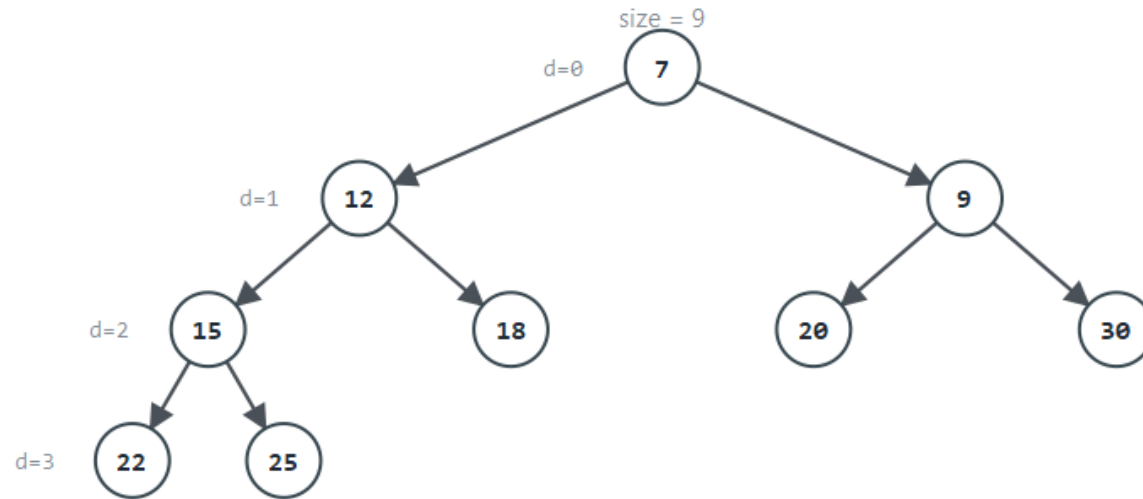
Save the root, move the **last** array element into its place — keeps the tree complete. Then repeatedly swap it with its **better child**. This is **sift-down**.

Extraction by sift-down, step by step



Done: extraction order [1, 3, 6]; 9 values remain: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Each extract costs $O(\log n)$.

Edge case — drain fully: all 10 values extracted



Done: extraction order [1, 3, 6]; 9 values remain: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Each extract costs $O(\log n)$.

Code — extract() / sift-down

```
int extract(void) {
    int best = heap[0];
    size--;
    heap[0] = heap[size];    /* ... */
    int i = 0;
    while (1) {              /* sift-down */
        /* ... target = better of left, right child ... */
        if (target == i)
            break;
        /* ... swap heap[i], heap[target]; i = target ... */
    }
    return best;
}
```

Complexity

- `extract` : at most one swap per **level** — $O(\log n)$
- The exact mirror image of sift-up's cost
- Same $O(\log n)$ bound, going the opposite direction

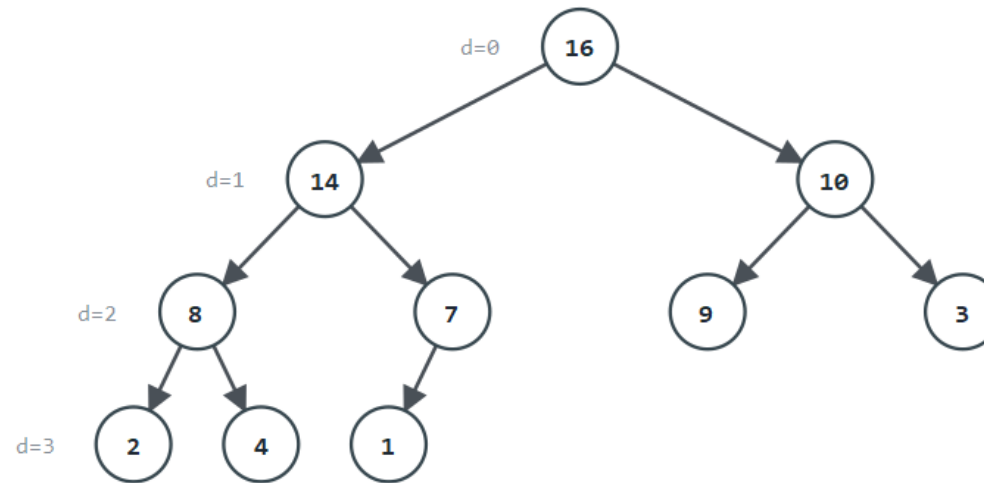
Common mistakes (insert and extract)

- Calling `extract` without checking `size > 0` first
- Comparing sift-down against only **one** child, not both
- Using `<=` instead of `<` in `better` — harmless, but changes tie behavior

Building a heap in $O(n)$

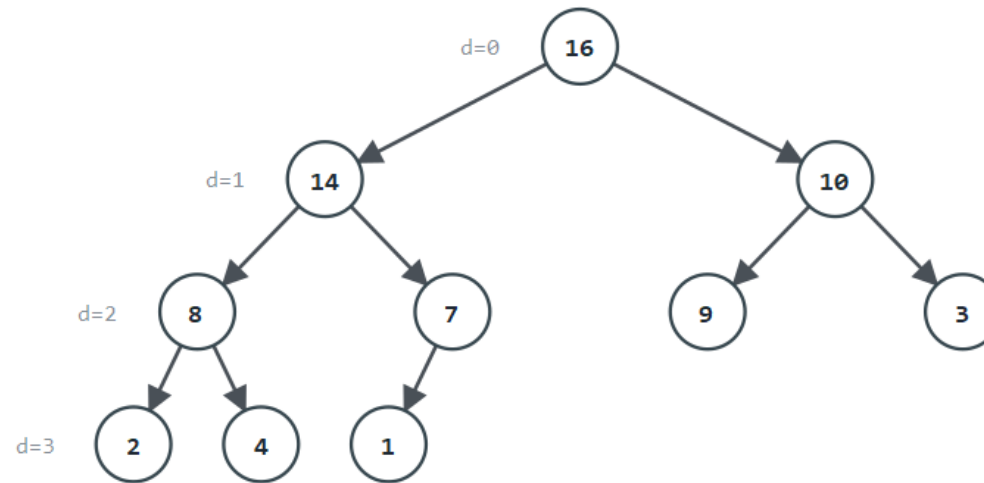
n single inserts cost $O(n \log n)$ total. **Floyd's algorithm** does better: every **leaf** is already a valid one-node heap; sift-down only the **internal** nodes, from the last one back to the root.

Build-heap, step by step



Done: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] is now a valid max-heap. Although each call costs differently, the TOTAL work is $O(n)$ (NOT $O(n \log n)$) -- the many nodes near the bottom need very little sifting.

Edge case — already a valid heap



Done: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1] is now a valid max-heap. Although each call costs differently, the TOTAL work is $O(n)$ (NOT $O(n \log n)$) -- the many nodes near the bottom need very little sifting.

Code — build_heap()

```
void build_heap(int arr[], int n) {  
    for (int i = n / 2 - 1; i >= 0; i--)  
        sift_down(arr, n, i);  
}
```

Why $O(n)$, not $O(n \log n)$?

- Roughly $n/2$ nodes are **leaves** — skipped entirely
- Roughly $n/4$ nodes cost at most 1 swap; $n/8$ cost at most 2
- "Count at height h times cost h ", summed, converges to **$O(n)$**
- Many cheap sifts near the bottom dominate the few costly ones near the top

Common mistakes

- Starting the loop at `i = 0` instead of `i = n/2 - 1` — sifts unfixed subtrees
- Assuming `build_heap` is "just n calls to insert" — valid, but not the same layout or cost

Mini-quiz

`build_heap` and `heap_sort`'s main loop both call `sift_down` repeatedly. Why is the first $O(n)$ overall, but the second $O(n \log n)$?

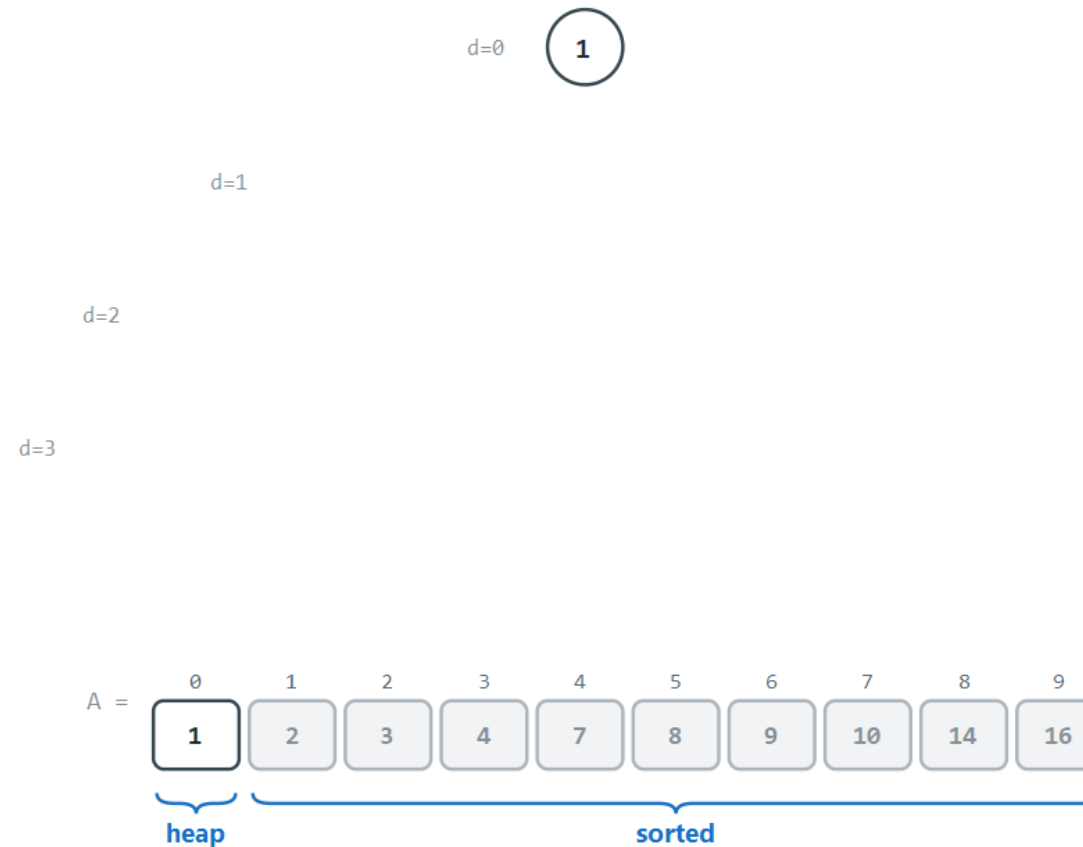
Answer

`build_heap` sifts mostly-cheap nodes near the bottom. `heap_sort` calls sift-down once per **extraction**, always starting from the **root** — no cheap majority to average against.

Heap sort — the idea

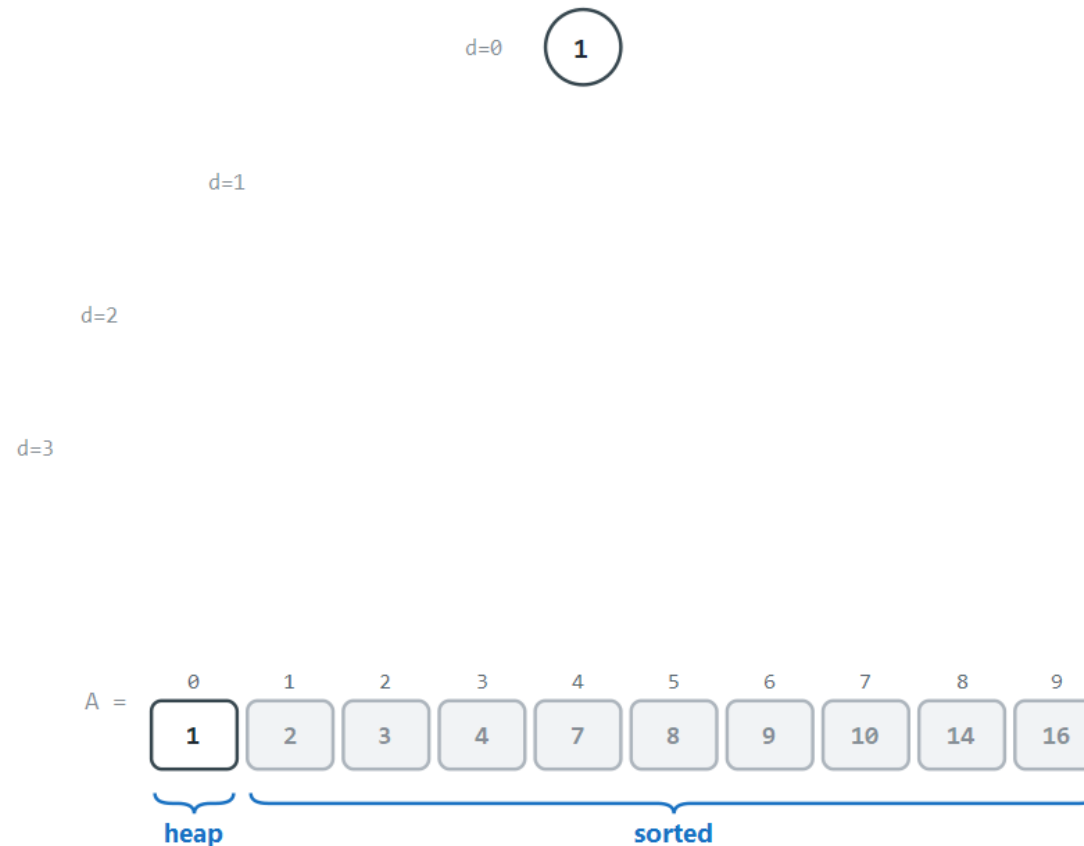
`build_heap`, then repeatedly move the root to the sorted tail, shrink, and sift-down.
A max-heap sorts **ascending**; a min-heap sorts **descending**.

Heap sort, step by step



Done: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16] -- fully sorted. Heap sort runs in place and takes $O(n \log n)$ time, but it is NOT stable.

Edge case — already ascending, same cost anyway



Done: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16] -- fully sorted. Heap sort runs in place and takes $O(n \log n)$ time, but it is NOT stable.

Code — heap_sort()

```
void heap_sort(int arr[], int n) {  
    for (int i = n / 2 - 1; i >= 0; i--)  
        sift_down(arr, n, i);  
    for (int heap_size = n; heap_size > 1; heap_size--) {  
        int tmp = arr[0];  
        arr[0] = arr[heap_size - 1];  
        arr[heap_size - 1] = tmp;  
        sift_down(arr, heap_size - 1, 0);  
    }  
}
```

Complexity

- `build_heap` : $O(n)$; the loop then runs $n - 1$ times
- Each round: one $O(1)$ swap plus one $O(\log n)$ sift-down
- Total: $O(n \log n)$ — sorts in place, no extra array

Common mistakes

- Believing heap sort is **stable** — it is not
- Sifting over the full array instead of the **shrunk** region — corrupts the already-sorted tail

6. Priority Queue: The ADT a Heap Serves

The ADT

A **priority queue** manages items, each with a **priority**. `insert` / `peek` / `extract` map directly onto the heap operations you just built.

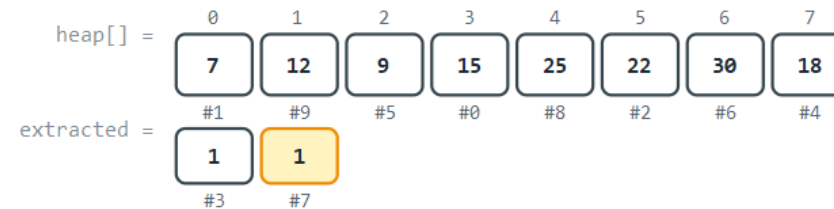
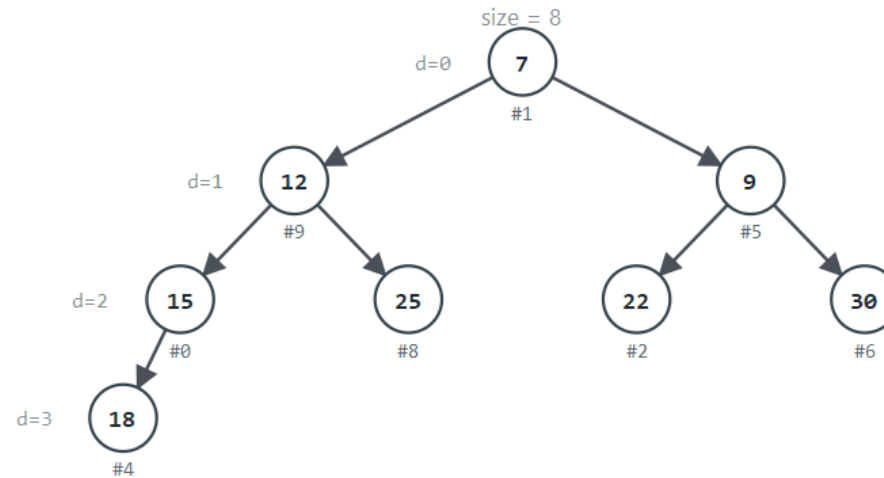
The priority queue ADT

Operation	What it does	Complexity
<code>insert(x)</code>	Adds <code>x</code> with its priority	$O(\log n)$
<code>peek()</code>	Returns the top item, keeps it	$O(1)$
<code>extract()</code>	Removes and returns the top item	$O(\log n)$
<code>update_key(id, p)</code>	Changes an item's priority	$O(n)$ naive

update_key — the new piece

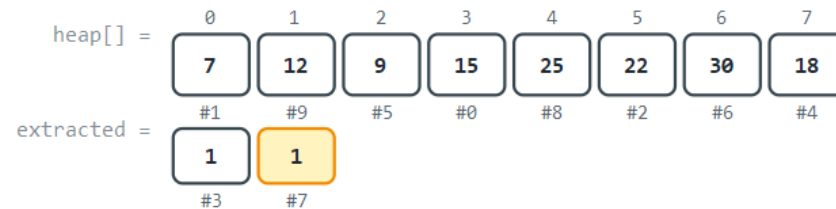
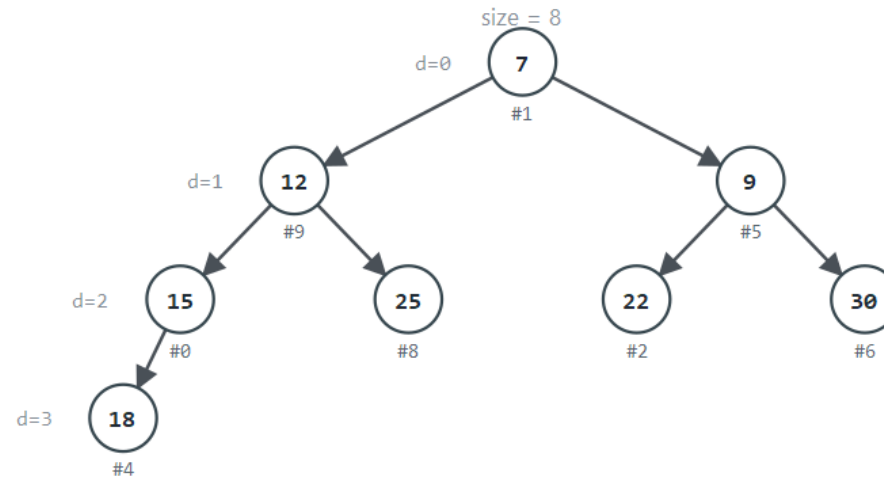
- Priority changes → item may need to move
- Better now: **decrease-key** — sift **up**
- Worse now: **increase-key** — sift **down**
- Real uses: Dijkstra's algorithm, OS schedulers

Priority queue operations, step by step



Done: 10 inserts, 2 extracts, 1 update-keys. 8 elements remain: [7, 9, 12, 15, 18, 22, 25, 30]. Every operation is $O(\log n)$ (peek is $O(1)$); the linear id scan inside `update_key` is $O(n)$ -- real implementations cut it to $O(\log n)$ with an id-to-index table.

Edge case — extract/peek while empty



Done: 10 inserts, 2 extracts, 1 update-keys. 8 elements remain: [7, 9, 12, 15, 18, 22, 25, 30]. Every operation is $O(\log n)$ (peek is $O(1)$); the linear id scan inside `update_key` is $O(n)$ -- real implementations cut it to $O(\log n)$ with an id-to-index table.

Code — Item, insert()

```
typedef struct {  
    int id;  
    int key;  
} Item;  
  
void insert(int id, int key) {  
    heap[size] = (Item){id, key};  
    sift_up(size);  
    size++;  
}
```

Code — update_key()

```
void update_key(int id, int new_key) {  
    int i = find_by_id(id);  
    if (i == -1) { /* ... print a message */ return; }  
    heap[i].key = new_key;  
    if (i > 0 && better(heap[i], heap[(i - 1) / 2]))  
        sift_up(i);  
    else  
        sift_down(i);  
}
```

Complexity

- `insert` , `extract` : $O(\log n)$, inherited from the heap
- `peek` : $O(1)$
- `update_key` : $O(n)$ with a linear scan for the id
- A hash table `id`→`index` cuts that to $O(\log n)$

Common mistakes

- Confusing an item's **id** (permanent) with its **array index** (not)
- Calling `peek` / `extract` without checking the queue is non-empty
- Sifting the wrong direction after `update_key`

Mini-quiz

A min-priority scheduler is keyed by "time to deadline". A running process is just interrupted by something far more urgent. Decrease-key or increase-key? Which sift?

Answer

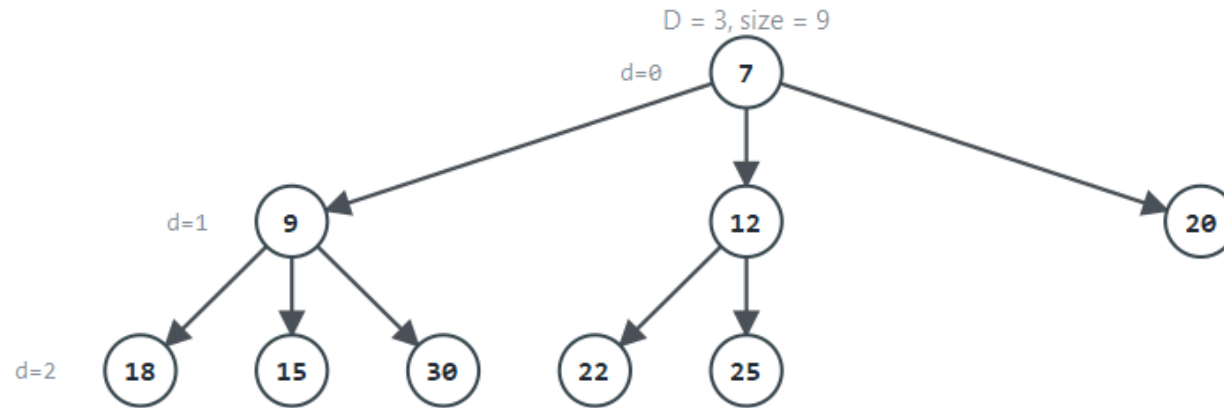
Decrease-key — more urgent means a **smaller** key. It triggers **sift-up**, floating toward the root, which holds the minimum.

7. Heap Variants: Trading One Property for Another

The d-ary heap

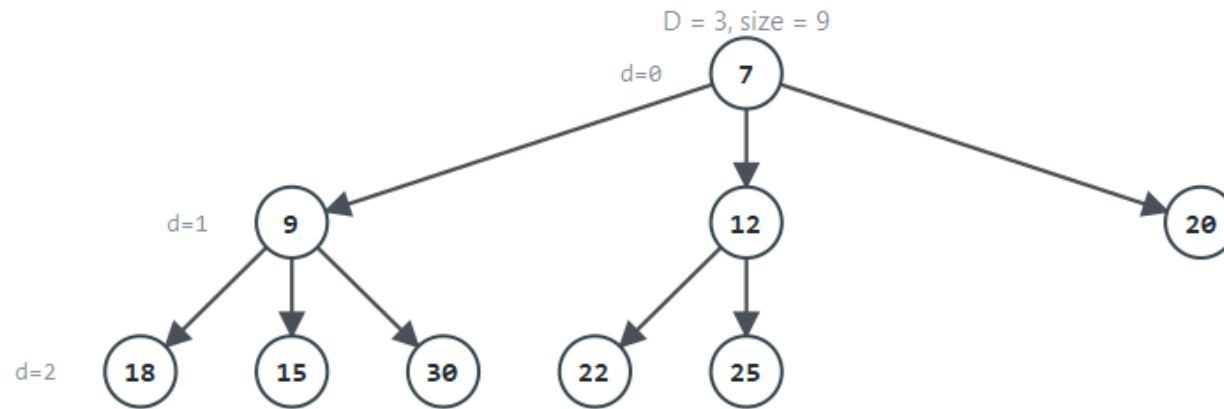
Same array-backed idea, but every node has up to **D** children, not 2. Child c of node i is at $D*i + 1 + c$; parent at $(i-1)/D$.
 $D = 2$ recovers the ordinary binary heap.

D-ary heap extraction, step by step



Done: extraction order [1, 3, 6]; 9 values remain: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Height is only $\log_D(n)$, so sift-down costs $O(\log_D n)$ -- a larger D means a shorter tree but more children to compare per step.

Edge case — $D=3$, drain fully



Done: extraction order [1, 3, 6]; 9 values remain: [7, 9, 12, 15, 18, 20, 22, 25, 30]. Height is only $\log_D(n)$, so sift-down costs $O(\log_D n)$ -- a larger D means a shorter tree but more children to compare per step.

Code — extract() with D children

```
int extract(void) {
    int best = heap[0];
    size--;
    heap[0] = heap[size];
    int i = 0;
    while (1) {
        int target = i, base = D * i + 1;
        /* ... check up to D children, base..base+D-1 ... */
        if (target == i) break;
        /* ... swap heap[i], heap[target]; i = target ... */
    }
    return best;
}
```

Complexity

- `extract` : $O(D \cdot \log_D n)$ — fewer levels, more per level
- `insert` : $O(\log_D n)$ — only ever compares one parent
- Larger D : cheaper inserts, potentially costlier extracts

Common mistakes

- Reusing the binary heap's $2*i+1$ / $2*i+2$ formulas unchanged
- Choosing a large D purely for a shorter tree, ignoring extract's cost

The binomial heap

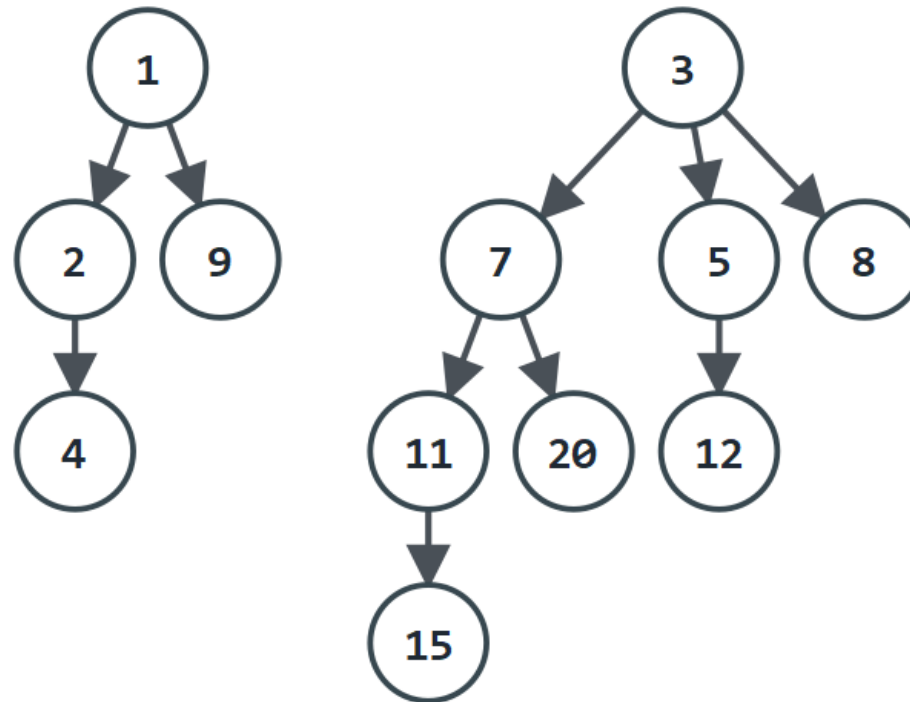
Introduced by **Jean Vuillemin**, 1978. A **forest** of binomial trees; order k has 2^k nodes. An n -element heap's orders are exactly the **set bits** of n in binary.

Union: binary addition with a carry

- Walk orders low to high, like adding two numbers
- One heap has a tree at this order → passes straight through
- **Both** do → they **link**, producing a "carry" one order up
- Up to **three** trees can meet at once: A, B, and an incoming carry

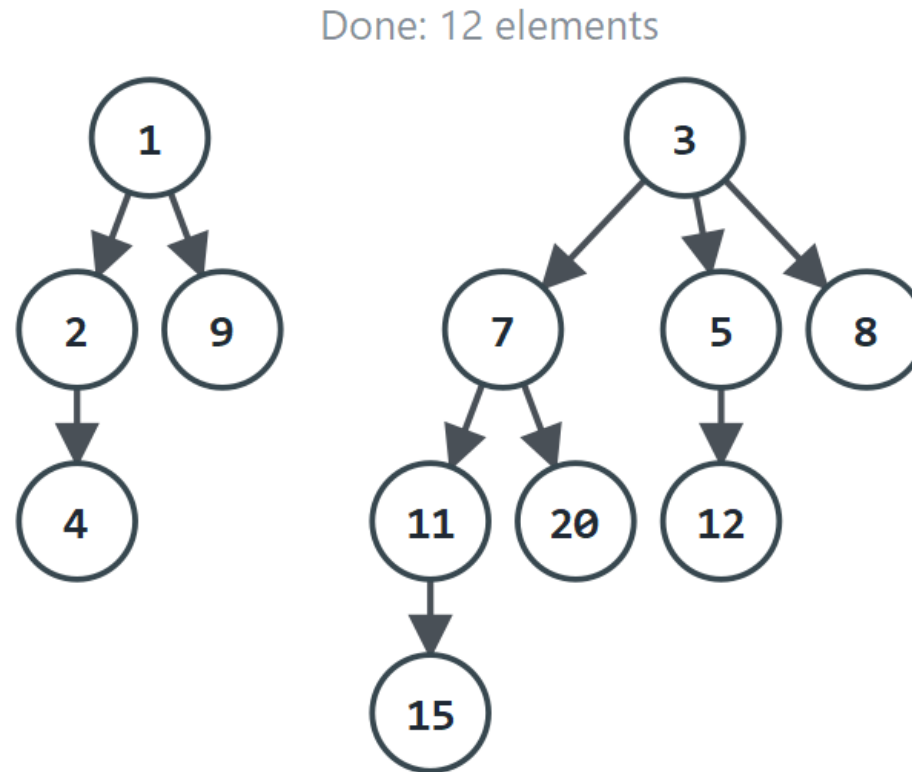
Binomial heap union, step by step

Done: 12 elements



Done: the unioned heap, orders [2, 3], 12 elements total. The best value (1) sits at some root -- every binomial tree's root is the best value in its own subtree. union costs $O(\log n)$: at most $\log n$ orders are visited.

Edge case — one long carry chain



Done: the unioned heap, orders [2, 3], 12 elements total. The best value (1) sits at some root -- every binomial tree's root is the best value in its own subtree. union costs $O(\log n)$: at most $\log n$ orders are visited.

Code — union_heaps()

```
void union_heaps(Node *a[], Node *b[], Node *result[]) {  
    Node *carry = NULL;  
    for (int order = 0; order < MAX_ORDER; order++) {  
        Node *group[3]; int g = 0;  
        if (a[order]) group[g++] = a[order];  
        if (b[order]) group[g++] = b[order];  
        if (carry) group[g++] = carry;  
        carry = NULL;  
        /* ... g==0: none; g==1: pass through ... */  
        /* ... g>=2: link() two trees, carry up ... */  
    }  
}
```

Complexity

- `link`: $O(1)$ — a few pointer reassignments
- `union`: visits at most $O(\log n)$ orders — $O(\log n)$
- `insert` is defined as `union` with one order-0 tree — also $O(\log n)$

Common mistakes

- Handling only **two** trees meeting at an order, forgetting the carry makes three
- Re-deriving `insert` from scratch instead of just calling `union`

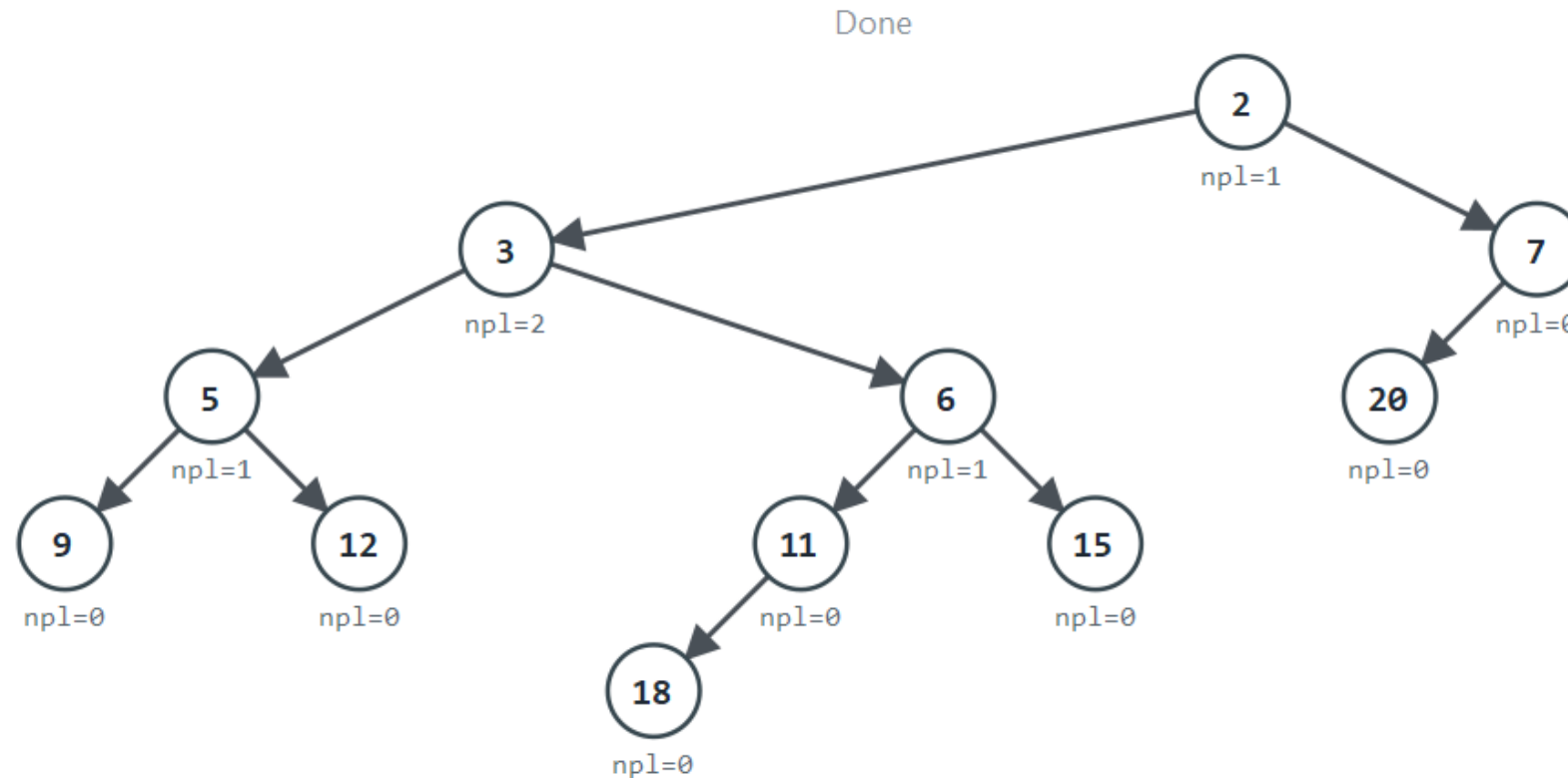
The leftist heap

Introduced by **C. A. Crane**, 1972. A pointer tree, generally **not** complete, built around one operation: **merge**. **insert** and **extract** both fall out of it as special cases.

The leftist property

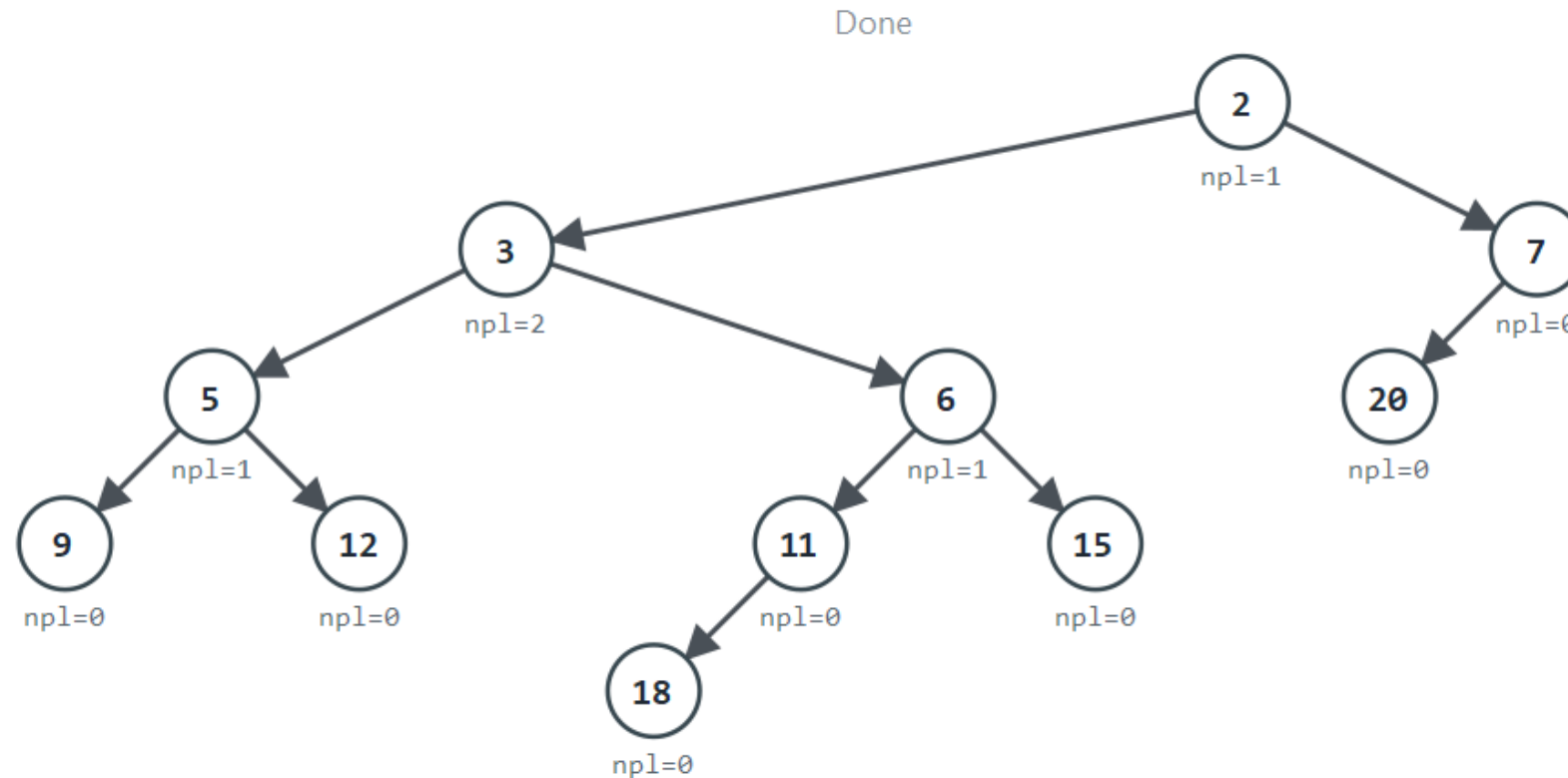
- Every node tracks its **null path length (npl)**
- **NULL** has npl -1 ; a leaf has npl 0
- **Leftist**: left child's npl never smaller than the right's
- Consequence: the **right spine** is always $O(\log n)$ long

Leftist heap merge, step by step



Done: the merged leftist heap, 11 elements, the best value is at the root: 2. merge costs $O(\log n)$, bounded only by the right-spine lengths -- and insert/extract are themselves just calls to merge.

Edge case — merging with an empty heap



Done: the merged leftist heap, 11 elements, the best value is at the root: 2. merge costs $O(\log n)$, bounded only by the right-spine lengths -- and insert/extract are themselves just calls to merge.

Code — merge() (recursive)

```
Node *merge(Node *t1, Node *t2) {  
    if (t1 == NULL) return t2;  
    if (t2 == NULL) return t1;  
    if (!better(t1->key, t2->key)) {  
        Node *tmp = t1; t1 = t2; t2 = tmp;  
    }  
    t1->right = merge(t1->right, t2);  
    if (npl(t1->left) < npl(t1->right)) {  
        /* ... swap t1->left and t1->right ... */  
    }  
    t1->npl = npl(t1->right) + 1;  
    return t1;  
}
```

Complexity

- `merge` : $O(\log n)$, bounded by both right spines
- `insert` , `extract` : both defined via `merge` — same $O(\log n)$
- Holds regardless of how unbalanced the rest of the tree is

Common mistakes

- Confusing **npl** (distance to nearest missing child) with **height**
- Skipping the child-swap after merge — silently breaks the leftist property

Mini-quiz

You need to merge two large heaps together,
far more often than inserting single items.
Which variant fits best?

Answer

The **leftist heap** (or the **binomial heap**).
Both are built so merge/union costs only $O(\log n)$ — a plain or d-ary heap has no fast way to merge two whole heaps at all.

8. Huffman Coding

A question to start

Plain ASCII spends 8 bits on every character, **E** or **Z** alike. What if common characters got **short** codes, and rare ones **long** — like Morse code already does?

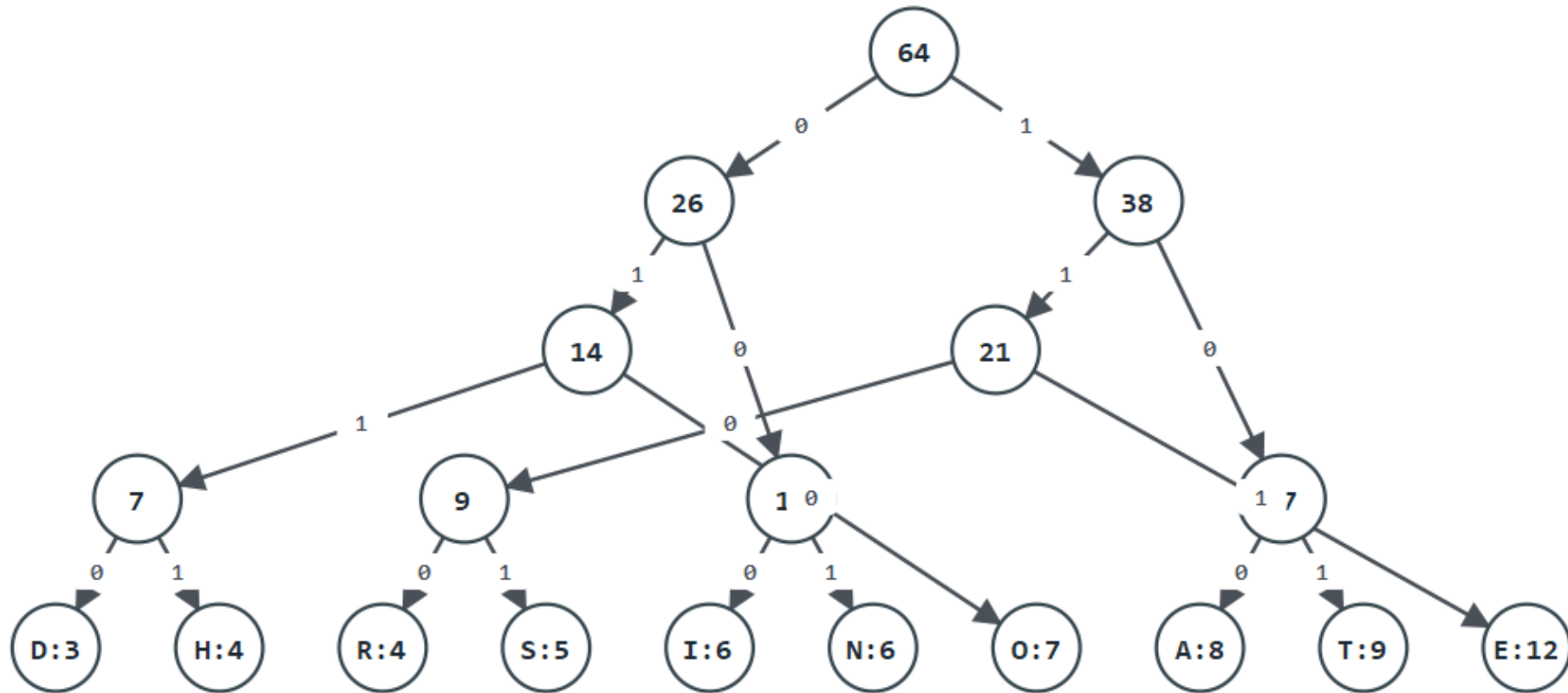
A short history

- **1952** — David Huffman, a term paper at MIT
- Professor Fano's own scheme was good, but not optimal
- Huffman's greedy tree-merging is **provably** optimal
- Still inside ZIP, JPEG, and MP3 today

Building the tree

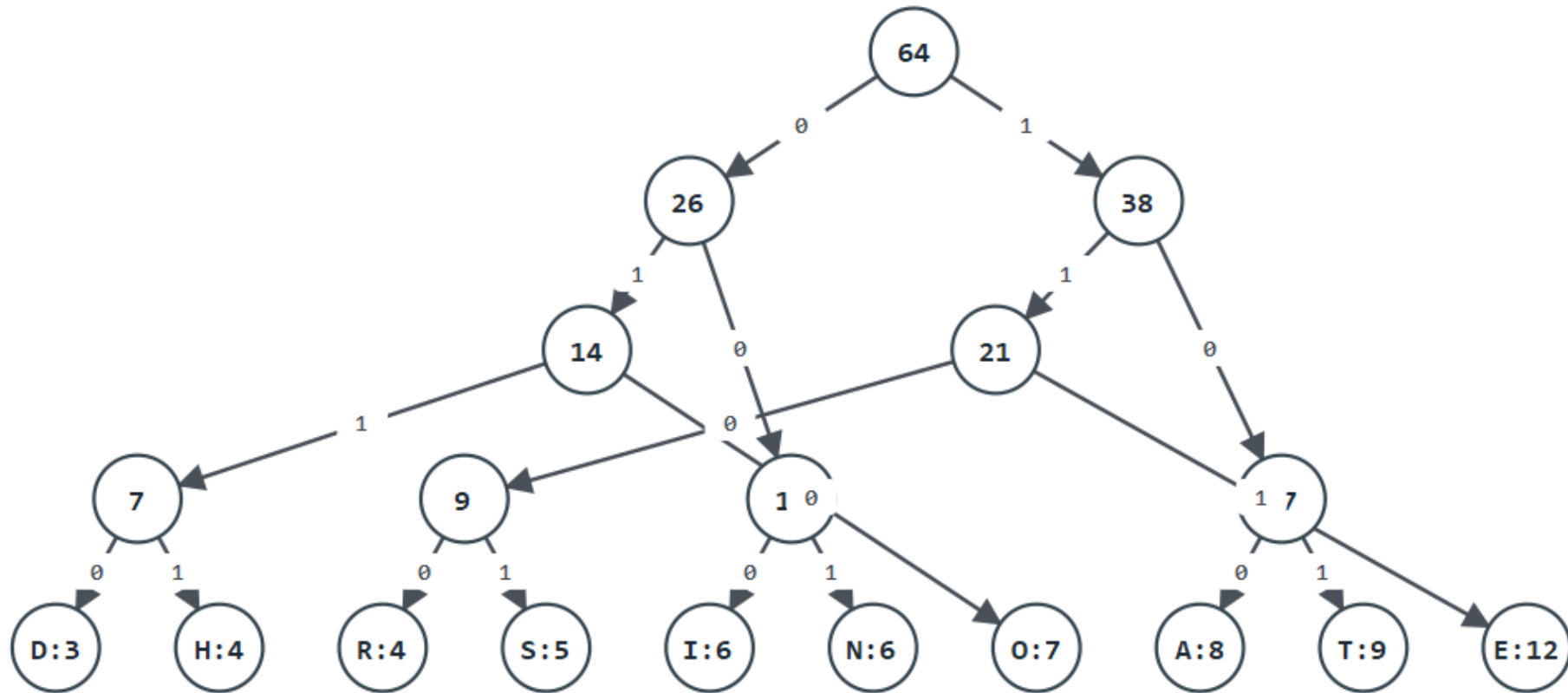
Put every symbol in a **min-heap**, keyed by frequency. Repeatedly: pop the two **smallest** roots, make a new internal node with those two as children, push it back. Repeat to one.

Building the Huffman tree, step by step



Done: one node remains, the root of the Huffman tree. Total merge cost (weighted path length) = 208. Every edge is 0 or 1 -- the path from the root to a leaf is that symbol's code.

Edge case — the smallest meaningful example



Done: one node remains, the root of the Huffman tree. Total merge cost (weighted path length) = 208. Every edge is 0 or 1 -- the path from the root to a leaf is that symbol's code.

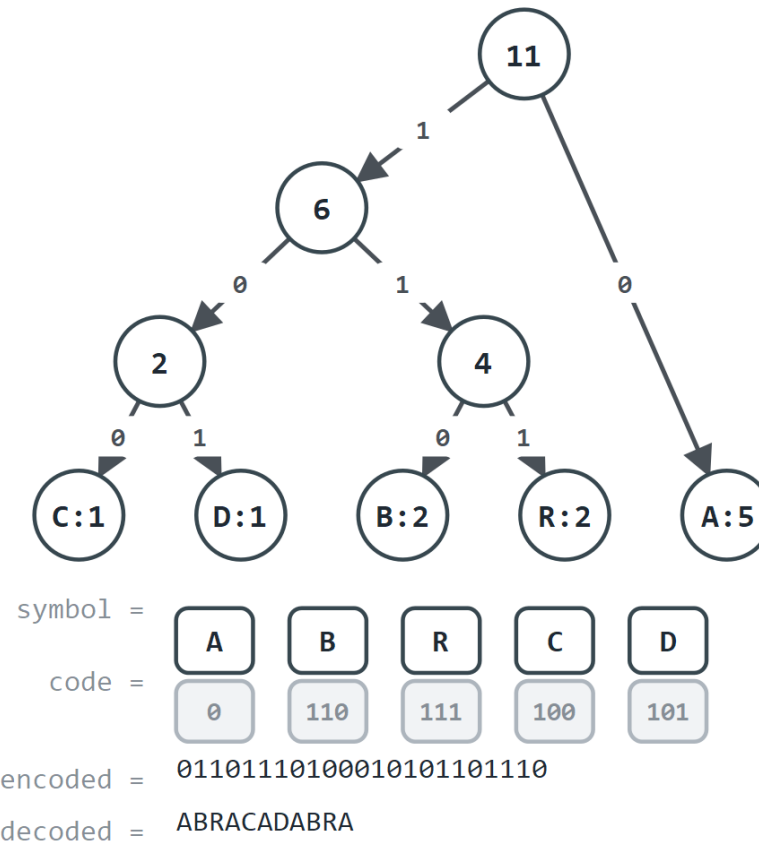
Code — the merge loop

```
int merge_id = 256;
while (heap_size > 1) {
    Node *a = heap_pop();
    Node *b = heap_pop();
    Node *parent = new_internal(a, b, merge_id++);
    heap_push(parent);
}
Node *root = heap_pop();
```

Encoding and decoding

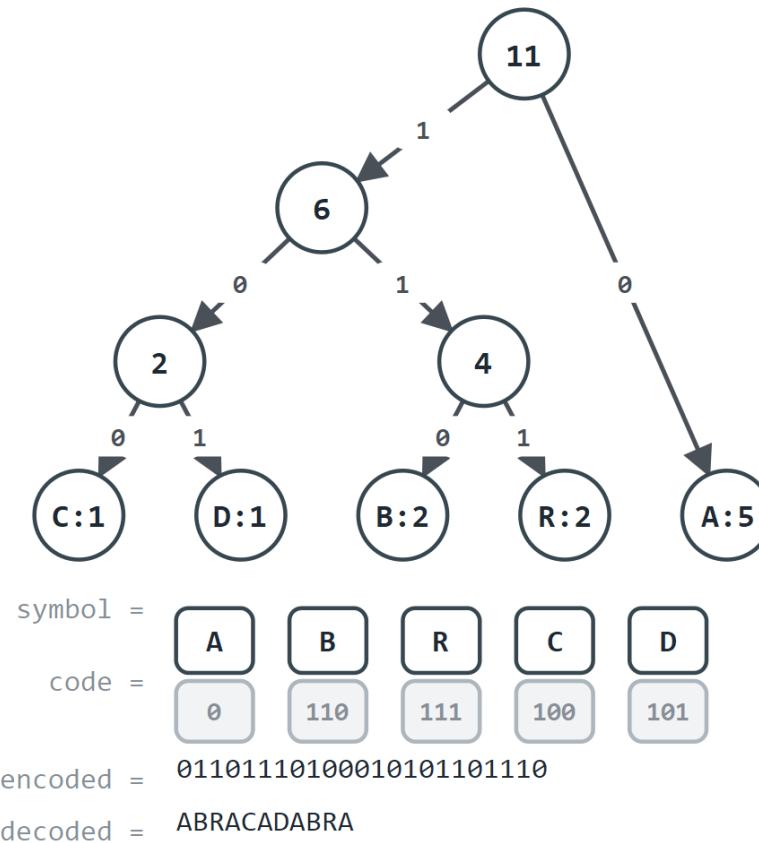
A symbol's **code** is its root-to-leaf path:
left = **0**, right = **1**. **Decoding** walks bit
by bit from the root; a leaf emits a character
and restarts. This works because the code is
prefix-free.

Encoding and decoding, step by step



Decoding done: 01101110100010101101110 → "ABRACADABRA". It matches the original text EXACTLY. A Huffman code is prefix-free -- no code is the start of another -- so it decodes in a single left-to-right pass, with no backtracking.

Edge case — very skewed: 9 A's, 1 B



Decoding done: 01101110100010101101110 → "ABRACADABRA". It matches the original text EXACTLY. A Huffman code is prefix-free -- no code is the start of another -- so it decodes in a single left-to-right pass, with no backtracking.

Code — assign_codes() (walk the tree)

```
static void assign_codes(Node *node, char *path,
                        int depth) {
    if (node->left == NULL && node->right == NULL) {
        path[depth] = '\\0';
        strcpy(codes[(unsigned char) node->ch], path);
        return;
    }
    path[depth] = '0';
    assign_codes(node->left, path, depth + 1);
    path[depth] = '1';
    assign_codes(node->right, path, depth + 1);
}
```

Code — decode() (walk bit by bit)

```
static char *decode(const char *bits, Node *root,
                    char *out) {
    int n = 0;
    Node *node = root;
    for (int i = 0; bits[i] != '\0'; i++) {
        node = bits[i] == '0' ? node->left : node->right;
        if (node->left == NULL && node->right == NULL) {
            out[n++] = node->ch;
            node = root;
        }
    }
    out[n] = '\0';
    return out;
}
```

Complexity

- Building: $n - 1$ merges, $O(\log n)$ each — **$O(n \log n)$**
- Encoding: **$O(L)$** — one lookup and append per character
- Decoding: **$O(B)$** — one tree step per encoded bit

Common mistakes

- No deterministic tie-breaker — equal frequencies can build two different, equally optimal trees
- Assuming the code table is fixed, like ASCII — it is built **per message**
- Testing `decode` alone, never the full `decode(encode(text)) == text` round trip

Mini-quiz

In a Huffman tree, can one symbol's code ever be a **prefix** of another symbol's code?

Answer

No. Every symbol is exactly one **leaf**,
and a leaf has no children — so no code can
ever be extended into a different, longer code.

Summary — trees, shapes, traversals, array

Idea	Key fact
Tree	One root, no cycles, $n-1$ edges for n nodes
Binary tree shapes	Full, complete, perfect, degenerate, balanced
Traversals	Pre/in/post (recursive), iterative, level order
Array representation	$2i+1$, $2i+2$, $(i-1)/2$ — all $O(1)$

Summary — heaps, priority queues, Huffman

Idea	Key fact
Binary heap	insert/extract $O(\log n)$; build $O(n)$
Heap sort	$O(n \log n)$, in place, not stable
Priority queue	<code>update_key</code> needs a stable id
Variants	d-ary (faster insert), binomial/leftist (fast merge)
Huffman coding	Min-heap of trees; prefix-free codes

The big picture

One structure, the **heap**, built from two moves — sift-up, sift-down — produces a sort, a priority queue, three variants, and an optimal compression code.

Self-check round

Four short questions. Think before the answer appears on the next slide. Full exercises and a ten-question quiz are in the week notes.

1. What is the maximum number of nodes in a binary tree of height 4?

$2^5 - 1 = 31$ nodes — a perfect tree of that height.

2. In a complete tree stored in an array, what is the parent index of node 9?

$(9 - 1) / 2 = 4$, using integer division.

3. Why is building a heap from n items $O(n)$, not the "obviously plausible" $O(n \log n)$?

Most nodes are near the bottom, where sift-down is cheap;
the sum converges to $O(n)$.

4. In a priority queue, why does every item need a permanent, stable id?

**An item's array position changes on nearly every operation;
only the id stays fixed.**

Next week

Week 5 — Graphs and Traversals

Drop "no cycles, one parent" and a tree becomes a **graph**. Level order generalizes to BFS; the priority queue you built today becomes the engine inside Dijkstra's algorithm.

References (1/2)

- Course syllabus, Week 4: `docs/syllabus/syllabus.en.md`
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 4th ed. MIT Press
- Sedgewick, Wayne. *Algorithms*, 4th ed. Addison-Wesley
- Knuth. *The Art of Computer Programming, Vol. 1*, 3rd ed.

References (2/2)

- Huffman (1952) · Williams (1964) · Floyd (1964)
- Vuillemin (1978) — the binomial heap
- Cayley (1857) — the first tree-counting paper
- [williamfiset/Algorithms](#) · Programiz DSA