Trees

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why trees matter **2**

2 Learning goals **2**

3 Prerequisites **2**

4 Tree models, terminology, and shape **2**

5 Representations and invariants **2**

6 Traversal from recursion **3**

7 Serialization and expression trees **3**

8 Tries **4**

9 Binary search trees **4**

10 Worked trace: BST deletion **4**

11 Rotations and balanced search trees **5**

12 B-trees and external memory **5**

13 Survey extension: spatial trees **5**

14 Augmenting trees **6**

15 Disjoint sets (union-find) **6**

16 Complexity assumptions and trade-offs **7**

17 Connections **7**

18 Common mistakes **7**

19 Self-check **7**

20 Revision summary **8**

21 Implementations and hands-on exploration **8**

21.1 Small experiments 8

22 Sources and further study **8**

References **8**

1 Why trees matter

A tree turns hierarchy into paths and recursive subproblems. File systems, syntax, search dictionaries, database indexes, spatial partitions, priority queues, and connectivity algorithms all use tree-shaped representations. Their performance depends less on the word "tree" than on shape, ordering invariants, branching, and the operations required.

2 Learning goals

After studying this note, you should be able to:

- use rooted-tree terminology and distinguish important binary-tree shapes;
- choose pointer, child-list, or implicit array representations;
- derive depth-first and breadth-first traversals and their space costs;
- search, insert, and delete in a binary search tree (BST);
- explain how rotations preserve search order and how balancing controls height;
- state B-tree occupancy invariants under an external-memory model;
- identify suitable spatial-tree queries and their assumptions;
- design a locally maintainable tree augmentation; and
- implement union-find with union by rank and path compression.

3 Prerequisites

You should understand records, references or array indices, recursion, stacks and queues, sorting order, and asymptotic and amortised analysis. No previous balanced-tree implementation is assumed.

4 Tree models, terminology, and shape

An undirected **tree** is a connected acyclic graph. A tree with n nodes has exactly $n - 1$ edges and a unique simple path between every pair of nodes. Choosing one node as the **root** orients the hierarchy: every other node has one parent, and nodes may have children, siblings, ancestors, and descendants.

A leaf has no children. The **depth** of a node is the number of edges from the root. Its **height** is the maximum number of edges on a downward path to a leaf; the tree height is the root height. The degree of a rooted node is its number of children.

In an **ordered tree**, sibling order matters. A binary tree distinguishes a left and right child even when only one exists. Several shape terms must not be confused:

- a **full** binary tree has either zero or two children at every node;
- a **perfect** binary tree has all leaves at the same depth and every internal node has two children; and
- a **complete** binary tree fills every level except possibly the last, which is filled from left to right.

A perfect binary tree of height h contains

$$1 + 2 + \cdots + 2^h = 2^{h+1} - 1$$

nodes. A complete tree supports compact array storage. With zero-based indexing, node i has potential children $2i + 1$ and $2i + 2$, and a non-root node has parent $\lfloor (i - 1)/2 \rfloor$. This layout becomes the basis of binary heaps.

5 Representations and invariants

A fixed-small-degree node can store one field per child. A general tree can store a dynamic child array or list. The **first-child/next-sibling** representation uses two links per node to encode arbitrary branching. Parent pointers make upward navigation cheap at an additional space and update cost.

An array representation removes explicit links only when shape determines positions, as in a complete binary tree. Sparse arbitrary trees would waste array slots.

Static ordered trees can also store shape **succinctly**. A depth-first traversal that emits bit `1` as an opening parenthesis on entry and bit `0` as its closing parenthesis on exit uses exactly $2n$ bits for an n-node shape before labels and navigation indexes.

Here **rank** counts opening bits up to a position, **select** locates the position of a given opening bit, and matching-parenthesis support finds the closing bit paired with an opening. Indexes for these operations can recover parent, child, and subtree relations without pointers. The price is more involved construction and updates, and the indexes must be included in any space claim.

The core representation invariant is reachability: every non-root node is reached from its parent exactly once, no link creates a cycle, and all recorded metadata agrees with the represented subtree. Ordered structures add constraints on child order or keys.

6 Traversal from recursion

A general depth-first traversal has an event before and after every subtree:

```
dfs(v):
    if v == null: return
    enter(v)                    // preorder event
    for child in children(v):
        dfs(child)
    leave(v)                    // postorder event
```

For a binary tree, an **inorder** event occurs between the left and right recursive calls. Preorder is useful for copying or prefix notation; postorder processes children before a parent and is useful for deletion or evaluation; inorder exposes BST keys in sorted order.

Breadth-first traversal replaces recursion with a queue:

```
bfs(root):
    if root == null: return
    Q = empty queue
    enqueue(Q, root)
    while Q is not empty:
        v = dequeue(Q)
        visit(v)
        for child in children(v):
            enqueue(Q, child)
```

By induction on subtrees, DFS visits every node reachable from its argument exactly once. BFS enqueues a node exactly when its unique parent is processed, so it visits nodes in nondecreasing depth. Both take $\Theta(n)$ time. DFS uses $O(h)$ call-stack space; BFS uses $O(w)$ queue space where w is the maximum width. Either can need $\Theta(n)$ space on an unfavourable shape.

7 Serialization and expression trees

A traversal order alone may not identify tree shape. A binary-tree serialization can emit a value in preorder and a marker for every null child:

```
serialize(v):
    if v == null: print "#"; return
    print v.value
    serialize(v.left)
    serialize(v.right)
```

Null markers make reconstruction unambiguous. Balanced parentheses give another representation: print an opening parenthesis at `enter`, the label, each child serialization, and a closing parenthesis at `leave`.

In an expression tree, leaves are operands and internal nodes are operators. Postorder recursively obtains child values before applying the parent operator, exactly matching postfix evaluation. Correctness follows structurally: if both child subexpressions are evaluated correctly, applying the stored operator evaluates their parent expression.

8 Tries

A **trie** stores strings by prefixes. An edge carries a character; a node represents the prefix along its root path; a terminal flag says that the prefix itself is a stored key. The flag is necessary when one key is a prefix of another, such as `he` and `hers`.

To search a pattern of length m, follow one outgoing edge per character and then test the terminal flag. Time is $O(m)$ if child lookup is $O(1)$. A full alphabet-sized child array gives fast lookup but wastes space at sparse nodes; maps or sorted edge lists use less space with a different lookup cost. Path compression merges nonbranching paths and leads toward radix and suffix-tree structures.

9 Binary search trees

Fix a duplicate policy. One rotation-stable choice orders records by `(key, unique tie-breaker)`; another stores all records with an equal key together in one node. In the resulting total record order, every record in a node's left subtree is smaller and every record in its right subtree is greater. This **BST invariant** implies that inorder traversal lists user-visible keys in nondecreasing order.

```
bst_search(root, k):
    v = root
    while v != null and v.key != k:
        if k < v.key: v = v.left
        else:         v = v.right
    return v
```

At each step, the invariant proves that the discarded subtree cannot contain `k`. Search takes $O(h)$ time and $O(1)$ iterative space. Insertion follows the same path until a null child is found, then attaches the new record there. Minimum follows left links; maximum follows right links.

The successor of a node is the minimum of its right subtree when that subtree exists. Otherwise, move upward until first arriving from a left child. Both cases identify the next record in inorder order; its user-visible key can be equal when duplicate records use tie-breakers.

Deletion has three structural cases:

1. A leaf is detached.
2. A node with one child is replaced by that child.
3. A node with two children uses its inorder successor, which has no left child, and then removes the successor from its old location.

When records have associated values, move or transplant the complete record, not only its key. Copying a key alone can pair it with the wrong value. Parent links, subtree metadata, and the root reference must also be updated.

10 Worked trace: BST deletion

Insert keys `8,3,10,1,6,14,4,7,13`. Searching for 7 follows `8 -> 3 -> 6 -> 7`, so its cost is proportional to that path.

Deleting 3 uses the two-child case. Its successor is 4, the minimum in the right subtree rooted at 6. Move the key-value record for 4 into 3's structural position, then detach the old 4 node, which is a leaf. The left subtree still contains only keys below 4, and the remaining right subtree contains `6,7`, both above 4. Thus the BST invariant is restored. The operation touches $O(h)$ nodes.

An ascending insertion sequence can produce a chain of height $n-1$, making all operations linear. Balance is therefore a structural guarantee, not a consequence of the BST order alone.

11 Rotations and balanced search trees

A rotation changes a parent-child relationship while preserving inorder order. In a right rotation around `y`, its left child `x` moves up, `x`'s right subtree becomes `y`'s left subtree, and `y` becomes `x`'s right child. The inorder sequence remains `A, x, B, y, C` before and after. Parent pointers and all augmentation metadata must be recomputed.

An **AVL tree** stores heights or balance factors and requires the two child heights of every node to differ by at most one. After insertion, repairing the lowest unbalanced ancestor with one or two rotations restores the required height along the remaining path. After deletion, height loss can propagate, so additional ancestors may also need repair.

Why does this local rule control the whole height? The minimum number of nodes at height h follows a Fibonacci-like recurrence, which implies $h = O(\log n)$ and therefore logarithmic search and update time.

A **red-black tree** stores one colour bit and maintains:

- the root and null leaves are black;
- a red node has black children; and
- every path from a node to a descendant null leaf contains the same number of black nodes.

Every root-to-null path has the same black-node count, and red nodes cannot be adjacent. Therefore any such path has at most twice as many internal nodes as its black-node count, giving height at most $2\log_2(n+1)$. Insertions and deletions recolour and rotate to restore the properties. AVL trees enforce tighter height; red-black trees usually permit fewer structural update repairs.

A splay tree moves an accessed node toward the root and gives logarithmic amortised, not per-operation worst-case, bounds. A treap stores a BST key and an independently randomized priority, maintaining BST order on keys and heap order on priorities; it has expected logarithmic height.

12 B-trees and external memory

When one storage-block access is far more expensive than comparisons in memory, a wide shallow tree is preferable. A B-tree of minimum degree $t \geq 2$ maintains:

- sorted keys within each node;
- one more child than keys at internal nodes;
- between $t-1$ and $2t-1$ keys at every non-root node;
- between 1 and $2t-1$ keys at a nonempty root, with at least two children when that root is internal;
- key ranges separated by the corresponding child pointers; and
- all leaves at the same depth.

Search chooses one child after locating a key interval inside a node. Insertion never descends into a full child without first splitting it: promote the median key to the parent and leave $t-1$ keys in each new child. Splitting the root creates a new root and increases height by one.

Deletion must also preserve minimum occupancy. For an internal target, use a predecessor or successor from a child with spare capacity, or merge two minimal children with their separating key. Before descending into a minimal child, a top-down implementation similarly borrows through the parent from a sibling or merges with one.

These repair rules preserve the invariants; the performance gain comes from matching a node to a storage block.

If a node is sized to one block and holds $\Theta(B)$ keys and child pointers, the height and number of block transfers are $O(\log_B n)$. CPU work inside each node depends on whether its keys are scanned, binary-searched, or indexed. "B" here describes block-scale branching, not an unexplained universal constant.

13 Survey extension: spatial trees

One-dimensional key order does not directly answer geometric queries. Spatial trees instead attach a region or bounding object to each subtree, then try to prove that a query can ignore whole regions:

- A **k-d tree** recursively splits points by coordinates. Orthogonal range search prunes cells disjoint from the query; nearest-neighbour search visits the nearer side first and prunes a farther cell only when its bounding region cannot improve the current answer.
- A **quadtree** divides a two-dimensional cell into four equal cells; an **octree** uses eight in three dimensions. Their depth depends on coordinate precision and point distribution.
- An **R-tree** groups objects by bounding rectangles and is designed for block storage. Overlapping rectangles may require exploring multiple branches.
- A **random-projection tree** partitions along chosen directions and can support approximate similarity search in higher-dimensional data.

These structures do not guarantee logarithmic queries for every geometry. Degenerate distributions, overlapping bounds, and the curse of dimensionality can force extensive search. State the metric, distribution, exact/approximate guarantee, and output size.

14 Augmenting trees

Augmentation stores metadata computable from a node and its children. If `size(v) = 1 + size(v.left) + size(v.right)`, a balanced BST can select rank k:

```
select(v, k):                          // zero-based rank
    left_size = size(v.left)
    if k < left_size: return select(v.left, k)
    if k == left_size: return v
    return select(v.right, k - left_size - 1)
```

The BST invariant fixes relative rank, and subtree sizes choose the only possible branch. Time is $O(h)$. Updates recompute size along changed paths and after rotations.

For intervals ordered by left endpoint, storing the maximum right endpoint in each subtree allows overlap search to prune a left subtree whose maximum endpoint lies before the query. Correctness requires proving that every pruned subtree cannot contain an answer. The general recipe is: define metadata locally, prove the pruning rule, and update every structural change.

15 Disjoint sets (union-find)

Here the trees are an internal representation, not the abstract value exposed to clients. The ADT represents a partition; changing parent links is valid whenever it preserves which elements share a representative.

Union-find maintains a partition under merging:

- `make_set(x)` creates the singleton set containing `x`;
- `find(x)` returns a representative of `x`'s current set; and
- `union(x,y)` merges the sets containing them.

A parent-pointer forest represents each set by one rooted tree. Roots represent sets.

```
make_set(x):
    parent[x] = x
    rank[x] = 0

find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])
    return parent[x]

union(x, y):
    rx = find(x); ry = find(y)
    if rx == ry: return
    if rank[rx] < rank[ry]: swap(rx, ry)
    parent[ry] = rx
    if rank[rx] == rank[ry]: rank[rx] = rank[rx] + 1
```

Union links only roots, so it cannot create a cycle between nodes already in one tree. Path compression changes parent links to the same representative, so it does not change set membership. Union by rank ensures that a rank-r root represents at least 2^r elements, hence rank is at most $\lfloor \log_2 n \rfloor$.

After `union(1,2)`, `union(3,4)`, and `union(1,3)`, a possible path is `4 -> 3 -> 1`. Calling `find(4)` returns 1 and rewrites it to `4 -> 1`.

For a sequence of $m \geq n$ operations on at most n elements, including the singleton creations, union by rank plus path compression takes $O(m\alpha(n))$ total time. The inverse Ackermann function α grows so slowly that the amortised cost is effectively constant for practical sizes, though it is not literally a constant worst-case bound for each call.

16 Complexity assumptions and trade-offs

Structure	Principal bound	Assumption or strength
Unbalanced BST	$O(h)$, worst $\Theta(n)$	simple dynamic ordered set
AVL/red-black tree	$O(\log n)$	guaranteed height under updates
B-tree	$O(\log_B n)$ block transfers	block-scale branching
Trie	$O(m)$ for key length m	constant-time child lookup
k-d/R-tree family	data- and query-dependent	geometric pruning
Augmented balanced BST	base operation plus local metadata	specialised rank/interval queries
Union-find	amortised $O(\alpha(n))$	unions and finds, no set splitting

17 Connections

- Complete binary trees and array layouts lead directly to binary and d-ary heaps.
- Skip lists from the preceding note offer randomized ordered dictionaries without rotations.
- *Succinct Data Structures* develops bit-vector rank/select and balanced-parentheses navigation in detail.
- Tries reappear in exact pattern matching, automata, and full-text indexing.
- Union-find supplies Kruskal's minimum-spanning-tree connectivity test in the graph note.
- B-trees motivate the distinction between RAM operations and storage-block transfers.

18 Common mistakes

- Confusing full, perfect, complete, balanced, binary, and binary-search trees.
- Counting traversal time but omitting recursion-stack or queue space.
- Assuming inorder traversal is sorted for every binary tree rather than only a BST.
- Copying only a successor key during deletion and leaving its associated value behind.
- Rotating without updating parent links, heights, colours, or augmentation metadata.
- Quoting B-tree or spatial-tree bounds without the block or data-distribution assumptions.
- Treating union-find amortised bounds as per-operation worst-case bounds.
- Using union-find for deletions, set splitting, or fully dynamic connectivity.

19 Self-check

1. Distinguish full, perfect, and complete binary trees with counterexamples.
2. Why does a tree with n nodes have $n - 1$ edges?
3. Compare DFS and BFS auxiliary space on a path and on a wide shallow tree.
4. Why are null markers needed to reconstruct an arbitrary binary tree from preorder?
5. Prove that inorder traversal of a BST is nondecreasing.
6. Explain why a rotation preserves the inorder sequence.

7. What B-tree invariant allows a node split to preserve occupancy?
8. Which metadata supports order-statistic selection, and where must it be updated?
9. Why does path compression preserve the represented partition?
10. What query or input assumptions would you require before claiming a spatial tree is efficient?

20 Revision summary

- Trees have unique paths; rooted order and shape add the structure used by algorithms.
- Full, perfect, and complete describe different binary-tree properties.
- DFS and BFS both take linear time but use height- and width-dependent space.
- BST operation cost follows height; rotations plus balance invariants guarantee logarithmic height.
- B-trees exchange binary branching for block-efficient shallow search.
- Spatial trees obtain speed from geometric pruning under stated assumptions.
- Augmentation is safe when metadata is local and every structural update maintains it.
- Union-find uses a forest, rank, and path compression for near-constant amortised merging and lookup.

21 Implementations and hands-on exploration

- NetworkX `UnionFind` source is a compact Python implementation using path compression and weighted linking. Looking up an unknown object creates a singleton, which is an API choice rather than a universal union-find rule.
- SciPy `KDTree` implements exact and approximate nearest-neighbour queries in Python. Its documentation warns that around 20 dimensions can already erase the advantage over brute force, and mutating uncopied input data can invalidate results.
- Rust `BTreeMap` provides an ordered map, range queries, and ordered iteration backed by a B-tree. Its public API deliberately hides node degree and layout, so textbook page-transfer constants cannot be inferred from the type alone.
- Java `TreeMap` is a red-black `NavigableMap` with guaranteed logarithmic basic operations. It is not synchronized, and its ordering should be consistent with `equals` when used as a `Map`.
- Linux kernel rbtree exposes an intrusive red-black tree and augmented-tree callbacks in C. Clients provide comparison, ownership, and locking, so the library does not by itself establish a complete dictionary contract.
- SDSL v3 is a header-only C++ library with compressed bit vectors, rank/select and balanced-parentheses support, wavelet trees, and compressed indexes. Succinct structures are often static or construction-heavy; practical speed depends on word operations, caches, and support structures.

21.1 Small experiments

1. Insert distinct keys into a simple BST in sorted order and in 100 shuffled orders. After every insertion, assert the BST bounds and that inorder traversal equals the sorted inserted keys; record height and average successful-search depth for the $O(h)$ cost.
2. Encode an ordered rooted tree with one bit on entry and one on exit. Assert that every prefix has at least as many openings as closings and that the final balance is zero; verify the $2n$ length, reconstruct with a stack, compare the ordered shape, and report raw bits separately from any navigation index.

22 Sources and further study

- Trees lecture deck, Jaak Vilo.
- (Cormen et al. 2022), Chapters 12-13, 18, and 19.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.