Lecture 5 – Hashing and Trees: Hash Functions, Collision Resolution, Balanced BSTs, and Tries

NURSULTAN KHAIMULDIN

N.KHAIMULDIN@ASTANAIT.EDU.KZ

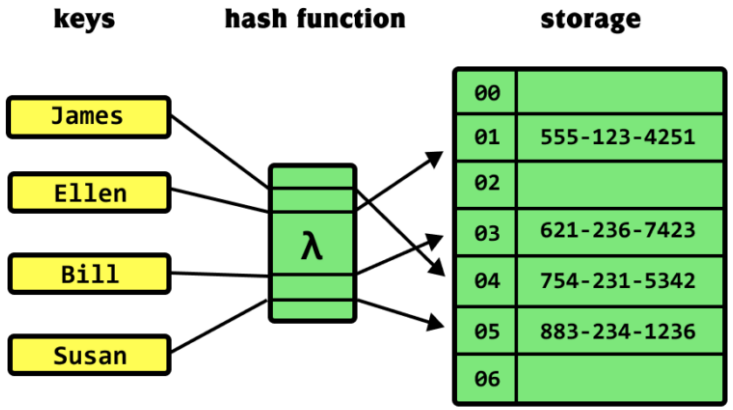
ASTANA IT UNIVERSITY

Content

1. Hash Function Design
2. Table Size & Resizing
3. Collision Resolution: Chaining
4. Collision Resolution: Open Addressing
5. Back-of-the-Envelope Math
6. Balanced BSTs
7. B-Trees & Treaps
8. Tries

ASTANA IT UNIVERSITY

Hash Function Design



A hash function is a fast, deterministic mapping $h:U \to \{0,\ldots,2^w-1\}$ that transforms keys into machine-word integers so that, for realistic inputs, the resulting bucket indices are indistinguishable from uniform random. A good hash diffuses small input changes across many output bits (avalanche), exhibits low correlation for nearby keys, and remains consistent with equality (same key → same hash), while being cheap enough that its cost doesn't dominate table operations

Mix → Mask, not mask raw hashCode.

Keep α low (resize early).

Normalize text; avoid mutable keys.

Seed for untrusted inputs.

ASTANA IT UNIVERSITY

Table Size & Resizing

The table size m is the number of buckets. We choose m and a resize policy to keep the load factor $\alpha = n/m$ near a target so operations stay fast.

Size choice:

- Power-of-two (2^k): index = mix(h) & (m−1)
  - very fast; cheap ×2 growth; good cache locality
    - must mix before masking (raw low bits are biased)
- Prime modulus (p): index = h % p
  - more forgiving for weak hashes
    - division is slower; awkward growth sequence

Load factor targets:

- Chaining: $\alpha \approx 0.75–2.0$ (small-vector buckets help locality)
- Linear/Quadratic probing: $\alpha \leq 0.6–0.7$
- Robin Hood: $\alpha \leq 0.85–0.9$
- Cuckoo: $\alpha \approx 0.5–0.6$

Growth & amortization:

- Trigger when $n > \alpha_{hi} \cdot m$; grow by ×2 (2^k) or ×1.5.
- Optional shrink with hysteresis when $n < \alpha_{lo} \cdot m$.
- Rehash cost is $O(m)$ per resize, but amortized $O(1)$ per insertion across many operations.
- For latency-sensitive systems: incremental rehash (move a few buckets per operation).

Implementation:

- Pre-size: $m \approx [n_{expected}/\alpha_{target}]$.
- Open addressing: track tombstones; rebuild if tombstone ratio grows.
- Double hashing: ensure step is coprime with m (odd step for 2^k tables).

Collision Resolution: Chaining

Store each bucket as a container of entries, so multiple keys that hash to the same index live in the same bucket.

Bucket layout options

- Linked list: simple O(1) insert at head; pointer-heavy, poor locality
- Small vector (array) per bucket: better cache locality; scan is branch-light
- Hybrid: small vector (4–8 slots) then spill to list if full

Complexity

- Expected lookup/insert: O(1)
- Successful probe cost ≈ 1 + α/2; unsuccessful ≈ α
- Worst case: O(n) if many collide (defend via better hashing / lower α)

Deletions

- O(1) unlink/remove; no tombstones or backward-shifts

Load factor

- Can operate with α > 1 (chains get longer); keep α modest for low p95 latency

Engineering tips

- Pre-store the hash in each entry to avoid recomputation and speed equality checks
- Prefer contiguous storage (small vectors) to reduce cache misses
- Consider resizing earlier or treeifying long buckets in adversarial settings
- Keep bucket scans branch-light (compare hash first, then equals)

Collision Resolution: Open Addressing

ASTANA IT UNIVERSITY

Keep all entries in the table itself. On collision, probe alternative slots until an empty one is found.

Core techniques

- Linear probing (LP): i, i+1, i+2, …
  - cache-friendly, very fast at low α
    - primary clustering; degrades quickly as α→1
- Quadratic probing (QP): i+1^2, i+2^2, …
  - breaks primary clustering
    - secondary clustering; needs care to guarantee full coverage
- Double hashing (DH): i + k·step(h2) (mod m)
  - better dispersion; avoids both clustering types
    - extra hash cost; step must be coprime with m (odd for 2^k)

Robin Hood hashing

- Idea: equalize probe distances (swap if newcomer is “older”).
- Effect: tighter probe-length tail (better p95/p99), supports higher α.
- Note: deletions still use tombstones; bookkeeping of displacement.

Cuckoo hashing (2 choices)

- Each key has 2 possible slots; lookup probes ≤2.
- Inserts may evict and “relocate” a chain; rehash on long cycles.
- Sweet spot: predictable reads, α ≈ 0.5–0.6 (2-way), optional stash for spikes.

Rules of thumb

- Target load factor: LP/QP ≤ 0.6–0.7, Robin Hood ≤ 0.85–0.9, Cuckoo ≈ 0.5–0.6.
- Deletions: use tombstones; rebuild when tombstones accumulate.
- With 2^k tables: mix → mask, and for DH use an odd step.

Back-of-the-Envelope Math

Simple *uniform hashing*; “probe count” ≈ number of bucket inspections. Using rough, quick calculations to estimate system capacity, storage needs, or performance requirements for hashing-related components like hash tables or distributed hash tables

How it works:

- **Define Parameters:** Start with key requirements like the number of users, the size of data per user, or the expected query rate.
- **Use Rules of Thumb:** Leverage well-known approximations, such as powers of two (10 doublings = ~1 million, 20 doublings = ~1 billion) or common latency figures.
- **Perform Simple Arithmetic:** Apply basic multiplication, division, and unit conversions to arrive at an estimate.
- **Simplify Assumptions:** Use simplified assumptions, such as multiples of 10 for easier computation.
- **Iterate and Compare:** Compare estimates for different architectural choices to find the most suitable design.

Example:

If you need to estimate the total storage for 200 million user records, each containing a small 5MB hash key and associated data:

Calculation:

200 million users * 5 MB/user = 1 billion MB = 1,000 GB = 1 TB.

Balanced BSTs

Self-balancing binary search trees keep height O(log n) by enforcing invariants and applying rotations after updates.

Invariants & Height Bounds

- AVL: balance factor $\in \{-1,0,+1\}$ at every node $\rightarrow$ height $\lesssim 1.44 \cdot \log_2 n$.
- Red-Black (RB): no red–red parent/child; equal black-height on all root-leaf paths $\rightarrow$ height $\leq 2 \cdot \log_2 n$.

Operations & Costs (worst-case)

- Search / Insert / Delete: O(log n)
- Rotations: AVL tends to rotate more (tighter balance); RB rotates less (looser balance).

Best usage in practice:

- Order / range queries, sorted iteration.
- k-th smallest / rank via subtree sizes.
- Good default map/set with predictable worst-case latency.

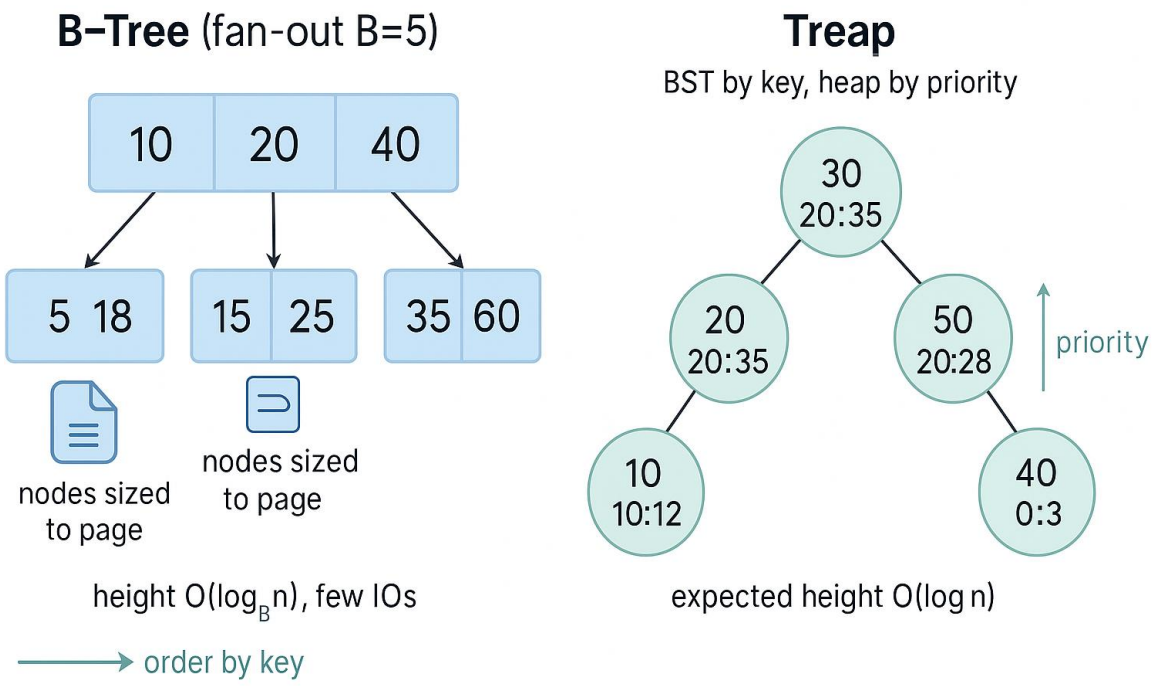
AVL vs RB:

- AVL: read-heavy workloads; best lookup times due to stricter balance.
- RB: write-heavy / mixed workloads; fewer rebalances, simpler constant factors.

Common augmentations

- Order-statistics tree: store subtree_size $\rightarrow$ select(k), rank(x) in O(log n).
- Interval tree: store max(high) in each node $\rightarrow$ interval stabbing queries in O(log n + k).
- Also: range sums/min/max with subtree aggregates; 2D variants (range trees).

B-Trees & Treaps



B-Trees

- High fan-out (B children per node) ⇒ height O(log_B n).
- Nodes sized to page size (e.g., 4–16 KiB) → few I/Os, great range scans.
- Standard in databases/filesystems; split/merge on updates; bulk-load friendly.

Treaps (simple randomized balancing)

- BST by key + heap by random priority.
- Expected O(log n), tiny code; easy split/merge/concat.
- Great for in-memory sets with order/range and order-stats via subtree_size.
- No strict worst-case bound (probabilistic balance).

B-Trees maximize storage locality/I-O efficiency;

Treaps give lightweight probabilistic balance for in-memory maps/sets.

Tries

A trie, or prefix tree, is a specialized tree-like data structure used to store and retrieve a dynamic set of strings, where the keys are sequences of symbols (e.g., characters or bits). Each node in a trie represents a common prefix of one or more keys, and the path from the root to a node defines that prefix.

Prefix tree keyed by symbols (characters or bits). A key is the path from root; operations depend on key length L, not on set size n.

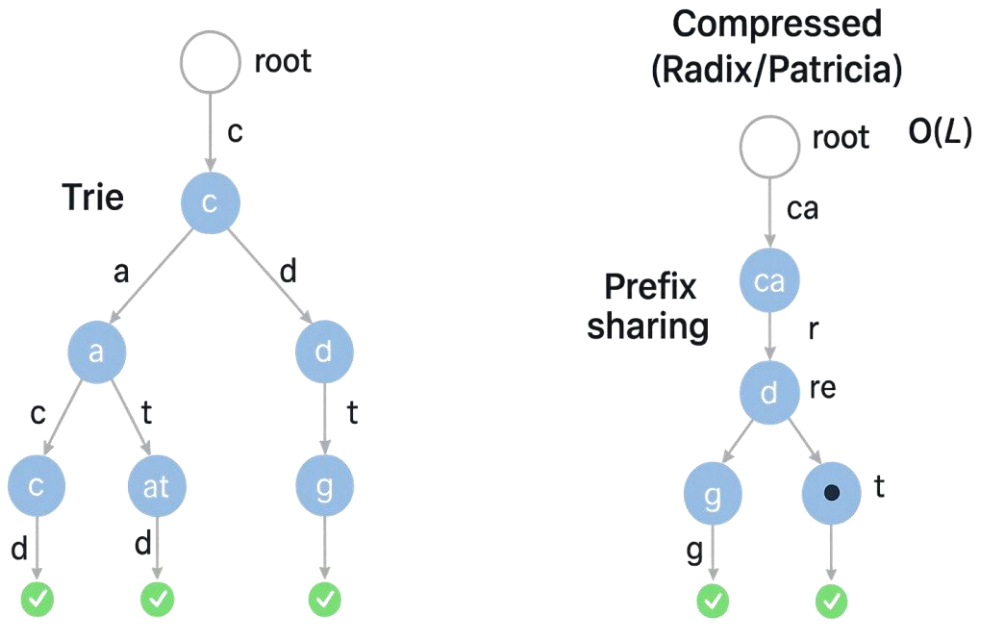
How tries are structured:

Root node: The trie starts with an empty root node that does not store a character. It acts as the starting point for all search and insertion operations.

Edges: Each edge connecting a parent node to a child node represents a single character or symbol.

Paths: The concatenation of characters along a path from the root to any node forms a unique prefix. A complete key is stored by traversing a path from the root, and the node at the end of the path is often marked as "terminal" to indicate a full word or key.

Shared prefixes: Keys that share a common prefix will also share the same sequence of nodes from the root. This "path sharing" is what makes tries memory-efficient for datasets with many similar keys.



Thank you