UNIVERSAL SYLLABUS · DATA STRUCTURES

Data Structures

Hash tables and collisions

The notes for the lesson “Hash tables and collisions”: chapters 6.7 to 6.8, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 6.7 to 6.8

About these notes

These are the notes for the University free n easy video lesson **Hash tables and collisions** on Data Structures: chapters 6.7 to 6.8, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.7 Hashing: hash tables and hash functions

Searching so far has cost O(n) or O(log n) because the key's position had to be discovered by comparing. Hashing computes the position instead: a hash function turns the key into an index of an array called the hash table, so insert, search and delete take O(1) on average. The price is that two keys can land on the same index (a collision, chapter 6.8), and that the table must be kept from filling up (the load factor and rehashing). Hash tables are the dictionary behind compilers' symbol tables, Python's `dict`, and the lookup of a UPI ID to a bank account.

You need: arrays (an earlier lesson); the dictionary ADT (an earlier lesson); modular arithmetic.

From direct addressing to hashing

If keys are small integers 0..m−1, store the record for key k at a[k]: **direct addressing**, O(1) with no search. Real keys are roll numbers, PAN numbers, words, with a universe U far larger than the number of keys n you actually hold. A **hash table** keeps an array T[0..m−1] of m **slots** (also called buckets) with m close to n, and a **hash function** h: U → {0, 1, ..., m−1} that says where key k lives: T[h(k)]. Search computes h(k) and looks in one slot. The **load factor** α = n/m is the average number of keys per slot and governs every cost formula in the next chapter.

A good hash function is (1) deterministic: the same key always gives the same index; (2) fast: O(1) arithmetic on the key; (3) uniform: it spreads the expected keys evenly over the m slots, so that clustering of real keys (consecutive roll numbers, words sharing a prefix) does not pile them into a few slots.

Hash functions for integer keys

- **Division method**: $h(k) = k \bmod m$. Choose m a prime not close to a power of 2 or 10; with $m = 2^p$ the hash is just the low p bits of k, and keys that differ only in higher bits collide. Fastest and the usual exam choice.
- **Multiplication method**: $h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$, where $0 < A < 1$ and $kA \bmod 1$ is the fractional part of kA. The value of m does not matter, so $m = 2^p$ is fine. Knuth suggests $A = (\sqrt{5} - 1)/2 \approx 0.6180339887$ (the golden ratio conjugate), which spreads consecutive keys well.
- **Mid-square**: square the key and take the middle digits (or bits): $72^2 = 5184$, middle digits 18, then 18 mod m. The middle of the square depends on every digit of the key.
- **Folding**: split the key into pieces and add them: 9876543210 → 987 + 654 + 321 + 0 = 1962, then mod m. Used for long numeric keys like Aadhaar or phone numbers.

String keys become integers first. The common choice is a **polynomial hash**: for a string $s_0 s_1 \ldots s_{L-1}$ with character codes c_i, $h = (c_0 \cdot b^{L-1} + c_1 \cdot b^{L-2} + \ldots + c_{L-1}) \bmod m$ with a small base such as $b = 31$, computed left to right as $h = (h \cdot b + c_i) \bmod m$ so no large number ever forms. Simply adding the codes is a poor function: "act", "cat" and "tac" would all collide.

Load factor and rehashing

As keys are inserted, α rises and every operation slows: chains get longer, probes get longer. Implementations fix a threshold (0.75 is common with chaining, 0.5 with open addressing) and when α crosses it they **rehash**: allocate a new table about twice as large (the next prime above 2m for the division method), and insert every existing key again with the new hash function, because $h(k) = k \bmod m$ changes when m changes. One rehash costs O(n), but it happens after about n insertions, so the average (amortized) cost per insertion stays O(1). **Applications**: symbol tables in compilers (identifier → type and address), dictionaries and sets in every language's library, caches (URL → page), database indexes for equality lookups, deduplication, and the `HashMap` behind an Android app's state.

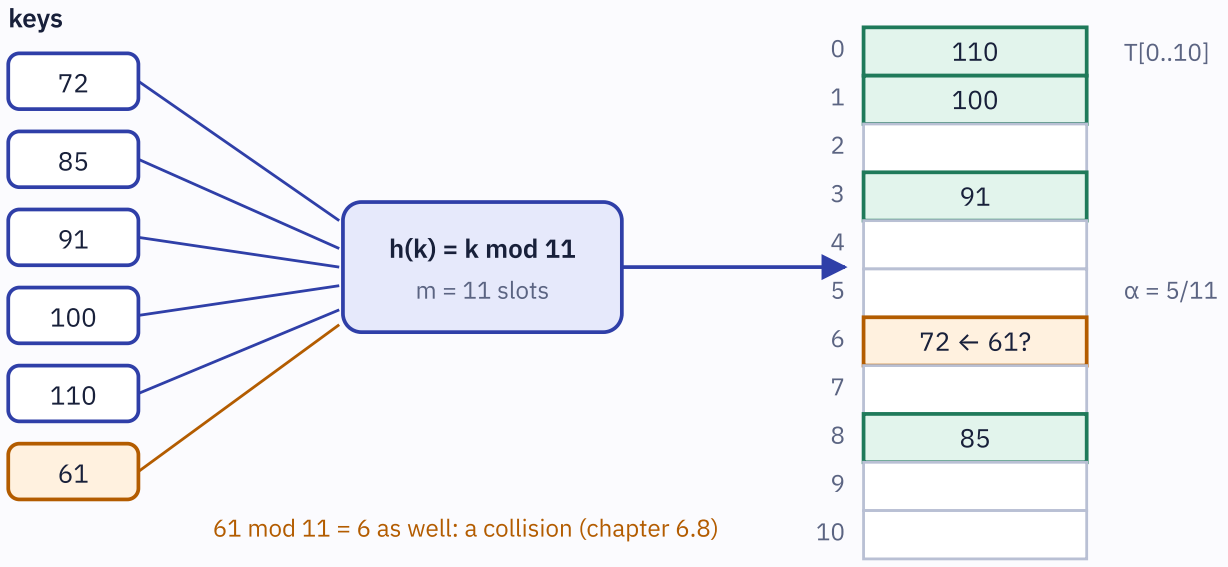

Figure 6.7.1 Hashing five keys with h(k) = k mod 11. Each key goes straight to its slot with no comparisons; search for 91 computes 91 mod 11 = 3 and looks only at T[3]. The key 61 also maps to slot 6, already holding 72: the collision that chapter 6.8 resolves. Load factor α = 5/11 ≈ 0.45.

WORKED EXAMPLE

Question. Compute the hash values of the keys 72, 85, 91, 100, 110 (a) by the division method with m = 11, (b) by the multiplication method with m = 8 and A = 0.618034. (c) Hash the string "cat" (ASCII c = 99, a = 97, t = 116) with base 31 and m = 101. (d) After inserting 8 keys into the 11-slot table, state the load factor and the new table size if rehashing happens at α > 0.7.

Solution (a). h(k) = k mod 11: 72 = 6 × 11 + 6 → **6**; 85 = 7 × 11 + 8 → **8**; 91 = 8 × 11 + 3 → **3**; 100 = 9 × 11 + 1 → **1**; 110 = 10 × 11 + 0 → **0**. No collisions among the five.

Solution (b). h(k) = ⌊8 × frac(kA)⌋:

K	KA	FRAC(KA)	8 × FRAC	H(K)
72	44.4984	0.4984	3.99	3
85	52.5329	0.5329	4.26	4
91	56.2411	0.2411	1.93	1
100	61.8034	0.8034	6.43	6
110	67.9837	0.9837	7.87	7

Again five different slots, this time in a table of only 8.

Solution (c). h = (99 × 31² + 97 × 31 + 116) mod 101 = (99 × 961 + 3007 + 116) mod 101 = (95139 + 3123) mod 101 = 98262 mod 101. 101 × 972 = 98172, remainder 90, so h("cat") = **90**. Computed incrementally: h = 99; h = (99 × 31 + 97) mod 101 = 3166 mod 101 = 35; h = (35 × 31 + 116) mod 101 = 1201 mod 101 = 90. Same answer, and no number ever exceeded 3166. (For "act" the same formula gives 48, so the anagram does not collide.)

Solution (d). α = 8/11 = 0.727 > 0.7, so rehash. New m = the first prime above 2 × 11 = 22, which is **23**; all 8 keys are re-inserted with h(k) = k mod 23 (72 → 3, 85 → 16, 91 → 22, 100 → 8, 110 → 18, …), and α falls to 8/23 = 0.35.

EXAM TRAPS

- Average case O(1) is for a good hash function and bounded α; the worst case of any hash table is O(n), when all keys collide. Say both.
- In the division method m should be prime and not near a power of 2; "m = 16 because it is a power of 2" is the examiner's trap, except in the multiplication method where m = 2^p is fine.
- Load factor is n/m (keys per slot), not m/n, and it can exceed 1 with chaining but never with open addressing.
- After rehashing you must recompute every key's hash with the new m; copying slots across does not work.
- A hash function must be deterministic; using the record's memory address or the time as a hash is a common wrong answer.

CHECK YOURSELF

– REMEMBER **Define hash table, hash function and load factor.** 3 marks

Hash table: an array of m slots holding records by key. Hash function: a map h from keys to slot indices 0..m−1. Load factor α = n/m, the number of stored keys per slot.

– REMEMBER **Write the formulas of the division and multiplication methods.** 2 marks

Division: h(k) = k mod m, m prime. Multiplication: h(k) = ⌊m (kA mod 1)⌋, 0 < A < 1, A ≈ 0.618.

– UNDERSTAND **Why is m = 2^p a poor choice for the division method?** 3 marks

k mod 2^p is just the lowest p bits of k. Keys that share those bits (for example, all multiples of 1024, or addresses aligned to 16 bytes) collide, and the higher bits never influence the slot.

– UNDERSTAND **Why is rehashing O(1) per insertion on average although one rehash is O(n)?** 3 marks

Doubling the table means the next rehash is about n insertions away, so the O(n) cost is spread over n insertions: O(1) each, an amortized bound.

– APPLY **Hash the keys 2024, 1331, 4096, 700 with h(k) = k mod 13.** 3 marks

2024 = 155 × 13 + 9 → 9; 1331 = 102 × 13 + 5 → 5; 4096 = 315 × 13 + 1 → 1; 700 = 53 × 13 + 11 → 11. No collision.

– APPLY **Hash "ab" (a = 97, b = 98) with base 31 and m = 50 by the incremental method.** 2 marks

h = 97 mod 50 = 47; h = (47 × 31 + 98) mod 50 = 1555 mod 50 = 5.

– ANALYZE **An exam office stores 12,000 roll numbers of the form 2024CS001..2024CS999, 2023CS..., in a table with h = (numeric part) mod 1000. What goes wrong and how do you fix it?** 4 marks

Each numeric part is a 4-digit year followed by a 3-digit number; mod 1000 keeps only the last three digits, so 2024CS001 and 2023CS001 collide, and only 1000 slots are ever used for 12,000 keys (α = 12). Fix: hash the whole key (year, branch and number) with a polynomial string hash, choose a prime m such as 17,011 (α ≈ 0.71), and keep α below 0.75.

BEYOND THE SYLLABUS

Not for the exam. Modern hash functions for tables (MurmurHash3, xxHash, SipHash, wyhash) mix every bit of the key in a few CPU cycles, and SipHash is used by Python, Rust and Perl because an attacker who can predict a weak hash can send keys that all collide and turn an O(1) table into an O(n) denial-of-service. Cryptographic hashes (SHA-256, used in UPI transaction signing and blockchains) are a different tool: collision resistance matters, speed does not. Universal and perfect hashing give provable guarantees; cuckoo and Robin Hood hashing change how collisions are handled. Search for "SipHash hash flooding", "universal hashing" and "Swiss table".

6.8 Collision resolution: separate chaining and open addressing

A collision is two keys with the same hash value, and with any hash function and more keys than slots it is a certainty. There are two families of cures. Separate chaining lets each slot hold a linked list of all keys that hash there. Open addressing keeps every key inside the table array and, on a collision, probes other slots in a fixed sequence (linear, quadratic or double hashing) until an empty one turns up. Exams ask you to insert a list of keys under each scheme, draw the table, and quote the expected number of probes as a function of the load factor α.

You need: hash functions and load factor (chapter 6.7); singly linked lists (an earlier lesson).

Separate chaining

T[i] is the head of a linked list of all keys with h(k) = i. Insert: compute i, add the node at the head of the list, O(1). Search: walk the list at T[i]. With a uniform hash function, each list has $\alpha = n/m$ keys on average, so an **unsuccessful search** examines about α nodes and a **successful search** about $1 + \alpha/2$ (the key is on average halfway along its list). α may exceed 1; performance degrades gently, and deletion is just a list deletion. Costs: one pointer per key plus the array, and poor cache behaviour because the nodes are scattered in memory. This is what Java's `HashMap` does (with the chains turning into balanced trees when they grow past 8).

Open addressing

All keys live in T itself, so $\alpha \le 1$ and m must exceed n. On a collision the table is **probed** at positions $h_0(k)$, $h_1(k)$, $h_2(k)$, ... until an empty slot is found (insert) or the key or an empty slot is found (search).

1. **Linear probing**: $h_i(k) = (h(k) + i) \bmod m$. Try the next slot, then the next, wrapping around. Simple and cache-friendly, but runs of occupied slots grow and merge: **primary clustering**. Any key hashing into a cluster must walk to its end, and the cluster grows further.
2. **Quadratic probing**: $h_i(k) = (h(k) + i^2) \bmod m$. Probes jump 1, 4, 9, ... slots ahead, breaking up primary clusters. Keys with the same h(k) still follow the same path (**secondary clustering**). With m prime and $\alpha \le 0.5$, the first m/2 probes are all distinct, so an insertion always succeeds; beyond that it may fail even with empty slots.
3. **Double hashing**: $h_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod m$ with a second function such as $h_2(k) = R - (k \bmod R)$, R a prime smaller than m. $h_2(k)$ is never 0 and, with m prime, the probe sequence visits every slot. Different keys get different step sizes, so no clustering of either kind; this is the closest practical scheme to ideal uniform hashing.

Deletion cannot simply empty a slot: a later search for a key that probed past it would stop early and miss. Mark the slot **deleted** (a tombstone): search passes through it, insert may reuse it. Tombstones accumulate and count towards α, so tables are periodically rehashed.

Expected probes (Knuth's analysis, uniform hash)

SCHEME	SUCCESSFUL SEARCH	UNSUCCESSFUL SEARCH / INSERT
Separate chaining	$1 + \alpha/2$	$1 + \alpha$ (one hash plus α nodes)
Linear probing	$\frac{1}{2}(1 + 1/(1 - \alpha))$	$\frac{1}{2}(1 + 1/(1 - \alpha)^2)$
Double hashing (≈ uniform)	$(1/\alpha) \ln(1/(1 - \alpha))$	$1/(1 - \alpha)$

At α = 0.5 these give 1.5 and 2.5 probes for linear probing, 1.39 and 2 for double hashing; at α = 0.9 linear probing's unsuccessful search needs 50.5 probes while double hashing needs 10. That is why open-addressing tables rehash at α around 0.5 to 0.7, and chaining tables at about 0.75 to 1.

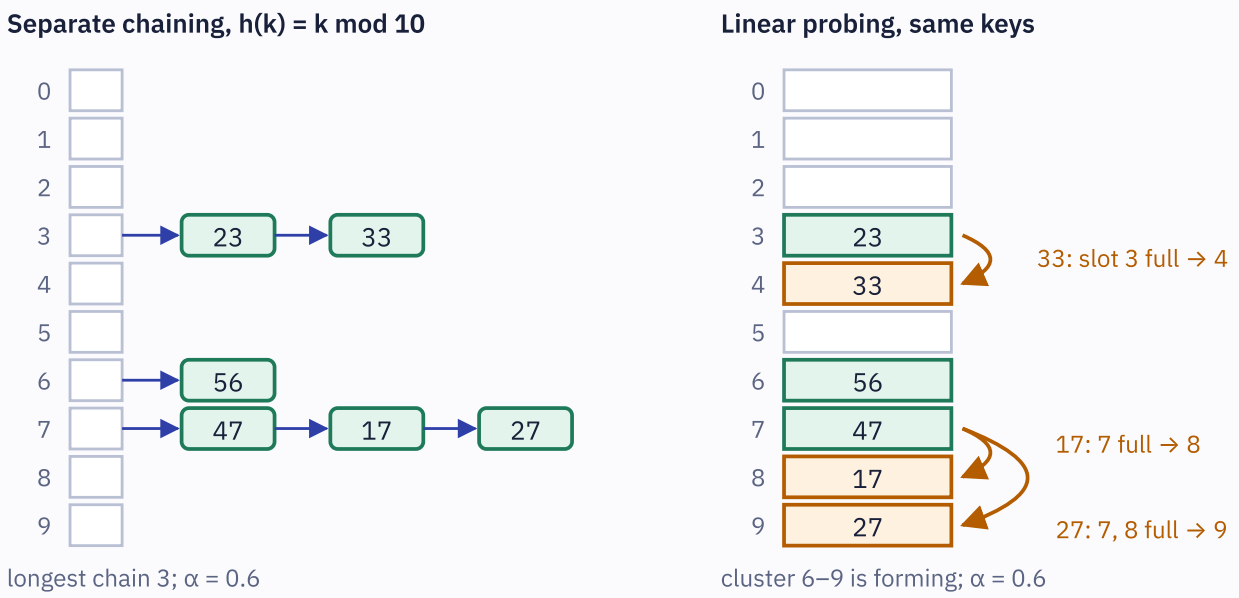

Figure 6.8.1 Six keys, h(k) = k mod 10, under the two schemes. Chaining puts 47, 17 and 27 on one list at slot 7. Linear probing pushes 17 to slot 8 and 27 to slot 9, next to 56 at 6: a cluster of four occupied slots that any key hashing to 6, 7, 8 or 9 must now walk through.

WORKED EXAMPLE

Question. Insert the keys 47, 23, 17, 33, 56, 27 in that order into a hash table of size m = 10 with h(k) = k mod 10 using (a) linear probing, (b) quadratic probing, (c) double hashing with $h_2(k) = 7 - (k \bmod 7)$, (d) separate chaining. Show the final tables, the number of probes for each insertion, and compare the average successful-search cost under linear probing with Knuth's formula.

Solution. Home slots: 47 → 7, 23 → 3, 17 → 7, 33 → 3, 56 → 6, 27 → 7.

(a) Linear probing.

KEY	PROBE SEQUENCE	PLACED AT	PROBES
47	7 free	7	1
23	3 free	3	1
17	7 full, 8 free	8	2
33	3 full, 4 free	4	2
56	6 free	6	1
27	7 full, 8 full, 9 free	9	3

Final table: T[3] = 23, T[4] = 33, T[6] = 56, T[7] = 47, T[8] = 17, T[9] = 27; slots 0, 1, 2, 5 empty. Average successful search = (1 + 1 + 2 + 2 + 1 + 3)/6 = 10/6 = **1.67 probes**. Knuth's formula with α = 6/10 = 0.6: ½ (1 + 1/(1 − 0.6)) = ½ (1 + 2.5) = **1.75**, close to the measured value.

(b) Quadratic probing, $h_i = (h + i^2) \bmod 10$: 47 → 7; 23 → 3; 17 → 7 full, 7 + 1 = 8 free → 8; 33 → 3 full, 4 free → 4; 56 → 6; 27 → 7 full, 8 full, 7 + 4 = 11 mod 10 = 1 free → **1**. Table: T[1] = 27, T[3] = 23, T[4] = 33, T[6] = 56, T[7] = 47, T[8] = 17. The jump of 4 took 27 out of the cluster.

(c) Double hashing: 17: $h_1 = 7$ full; $h_2 = 7 - (17 \bmod 7) = 7 - 3 = 4$; 7 + 4 = 11 mod 10 = 1 free → **1**. 33: $h_1 = 3$ full; $h_2 = 7 - 5 = 2$; 3 + 2 = 5 free → **5**. 27: $h_1 = 7$ full; $h_2 = 7 - 6 = 1$; 8 free → **8**. Table: T[1] = 17, T[3] = 23, T[5] = 33, T[6] = 56, T[7] = 47, T[8] = 27; every colliding key needed exactly 2 probes.

(d) Separate chaining (Figure 6.8.1, new keys appended at the tail): slot 3: 23 → 33; slot 6: 56; slot 7: 47 → 17 → 27. Inserting at the head instead would reverse each list. Longest chain 3; an unsuccessful search for 97 compares against 3 keys in slot 7, for 10 against none.

EXAM TRAPS

- Probe count includes the first look at the home slot. A key placed in its home slot took 1 probe, not 0.
- In double hashing, $h_2(k)$ must never be 0 (the sequence would stay put) and should be coprime to m; R − (k mod R) with R prime below m guarantees the first, and making m prime guarantees the second (with m = 10, h_2 = 2 or 5 would skip slots).
- Quadratic probing can fail to find an empty slot when α > 0.5 or m is not prime; say so when the question asks "always succeeds?".
- Deleting in open addressing by clearing the slot breaks later searches; use a tombstone. Chaining has no such problem.
- Write the probe sequence formulas with mod m and the wrap-around; students who stop at slot 9 instead of wrapping to 0 lose the marks.

CHECK YOURSELF

– REMEMBER **Write the probe sequences for linear, quadratic and double hashing.** 3 marks

Linear (h(k) + i) mod m; quadratic (h(k) + i^2) mod m; double ($h_1(k)$ + i·$h_2(k)$) mod m, i = 0, 1, 2, ...

– REMEMBER **What are primary and secondary clustering?** 2 marks

Primary: in linear probing, runs of occupied slots grow and merge so any key landing in a run walks to its end. Secondary: in quadratic probing, keys with the same home slot follow the same probe path.

– UNDERSTAND **Why can an unsuccessful search in a chained table stop after α comparisons but in linear probing may need far more at the same α?** 3 marks

In chaining only the keys that hash to the same slot are examined. In linear probing the search also walks through keys that hash to neighbouring slots and were pushed along, so clusters make the probe count grow like $1/(1 - \alpha)^2$.

– APPLY **Insert 12, 44, 13, 88, 23 into m = 11 with h = k mod 11 and linear probing. Show the table.** 4 marks

12 → 1; 44 → 0; 13 → 2; 88 → 0 full, 1 full, 2 full, 3 free → 3; 23 → 1 full, 2, 3 full, 4 free → 4. Table: T[0] = 44, T[1] = 12, T[2] = 13, T[3] = 88, T[4] = 23. Probes 1, 1, 1, 4, 4.

– APPLY **Expected probes for an unsuccessful search at α = 0.8 under linear probing and double hashing?** 3 marks

Linear: ½ (1 + 1/0.2^2) = ½ (1 + 25) = 13. Double: 1/0.2 = 5.

– ANALYZE **A real-time system stores 10,000 sensor IDs and must answer every lookup within a fixed time. Chaining or open addressing, and at what load factor?** 4 marks

Open addressing with double hashing (or linear probing for cache speed) at $\alpha \le 0.5$, in one contiguous array: at most about 2 expected probes, no pointer chasing, no memory allocation during operation. Chaining's lists allocate nodes and scatter memory, which hurts worst-case latency. For a hard guarantee, pair the table with a bound on the longest probe sequence and rehash if it is exceeded.

BEYOND THE SYLLABUS

Not for the exam. Linear probing, dismissed in textbooks for clustering, is the fastest scheme on modern CPUs because a cluster sits in one cache line; Google's Swiss tables (now in C++ `absl::flat_hash_map`) and Rust's `HashMap` probe 16 slots at a time with SIMD instructions. Robin Hood hashing steals slots from "rich" keys to even out probe lengths; cuckoo hashing guarantees at most two probes for a lookup; hopscotch hashing bounds the distance from the home slot. Search for "Swiss table SIMD", "Robin Hood hashing" and "cuckoo hashing".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Division method	$h(k) = k \bmod m$, m prime not near 2^p	6.7
Multiplication method	$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$, $A \approx 0.618$	6.7
Load factor	$\alpha = n/m$; rehash to about 2m (next prime) when α crosses the threshold	6.7
Separate chaining	Linked list per slot; unsuccessful $\approx 1 + \alpha$, successful $\approx 1 + \alpha/2$	6.8
Linear probing	$(h + i) \bmod m$; successful $\frac{1}{2}(1 + 1/(1-\alpha))$, unsuccessful $\frac{1}{2}(1 + 1/(1-\alpha)^2)$	6.8
Double hashing	$(h_1 + i \cdot h_2) \bmod m$, $h_2 = R - (k \bmod R)$; unsuccessful $\approx 1/(1-\alpha)$	6.8

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- **4. Define hash table, hash function, collision and load factor. Describe the division, multiplication, mid-square and folding methods with one example each. (8 marks)**

  Outline: definitions; properties of a good hash function (deterministic, fast, uniform); k mod m with prime m; ⌊m(kA mod 1)⌋ with A ≈ 0.618; $72^2 = 5184 \rightarrow 18$; 987 + 654 + 321; α = n/m and why it matters.

Apply

- **9. Insert 19, 27, 36, 10, 64, 52, 28 into a hash table of size 9 with h(k) = k mod 9 using (a) linear probing, (b) quadratic probing, (c) separate chaining. Draw each final table and count the probes. (12 marks)**

  Outline: home slots 1, 0, 0, 1, 1, 7, 1; (a) linear: 19→1, 27→0, 36→0,1,2→2, 10→1,2,3→3, 64→1..4→4, 52→7, 28→1..5→5; (b) quadratic: 36→0+1=1 full, 0+4=4→4; 10→1+1=2; 64→1,2,5→5; 28→1,2,5,1+9=10 mod 9=1 full... continue to 1+16=17 mod 9=8; (c) chains at 0: 36→27, 1: 28→64→10→19, 7: 52; probe counts for each.

Analyze and evaluate

- **14. "A hash table is always faster than a balanced BST." Evaluate this claim with respect to average and worst case, ordered operations, memory and the choice of hash function. (10 marks, spans chapters 6.7, 6.8 and an earlier lesson)**

  Outline: hash O(1) average vs BST O(log n); hash worst O(n) with bad function or adversarial keys vs AVL/red-black guaranteed O(log n); range queries, min/max, in-order listing need the BST; memory and rehashing pauses; conclusion: faster for exact lookup with a good hash and bounded α, not "always".

- **15. Compare linear probing, quadratic probing and double hashing at α = 0.5 and α = 0.9 using the expected-probe formulas, and recommend a rehash threshold for a system that must answer in a bounded time. (10 marks)**

  Outline: compute ½(1 + 1/(1−α)²): 2.5 and 50.5 for linear; 1/(1−α): 2 and 10 for double hashing; clustering explanation; quadratic may fail above 0.5 unless m is prime; recommend rehash at about 0.5 to 0.7 and double hashing or linear probing in one cache line.

- **16. A search feature must support exact match, prefix autocomplete and "find this substring in a document". Match each need to binary search, hashing, a trie, KMP or Boyer-Moore and justify the costs. (10 marks, spans chapters 6.1, 6.7 and 6.9)**

  Outline: exact match → hash table O(1) (or binary search O(log n) when memory matters); prefix → trie O(L) or binary search on a sorted array for the first match; substring in a document → KMP Θ(n + m) guaranteed, Boyer-Moore faster on natural text; explain why a hash cannot do prefixes and why a trie cannot do substrings without a suffix structure.

Glossary (terms used in this lesson)

Bucket	A slot of a hash table, or one of the b groups in a radix-sort pass.
Collision	Two different keys hashing to the same slot.
Division method	h(k) = k mod m with prime m.
Double hashing	Open addressing with probe step $h_2(k)$, different for different keys.
Linear probing	Open addressing that tries slots h, h+1, h+2, ... mod m.
Load factor	α = n/m, stored keys per slot.
Multiplication method	h(k) = ⌊m (kA mod 1)⌋ with 0 < A < 1.
Open addressing	Collision resolution that keeps all keys inside the table and probes for free slots.
Primary clustering	Growing runs of occupied slots under linear probing.
Rehashing	Moving every key to a larger table with a new hash function when α gets high.
Run	A sorted block of records written to disk in the first phase of external sorting.
Separate chaining	Collision resolution with a linked list per slot.
Tombstone	A "deleted" marker in open addressing that searches pass over and inserts may reuse.

Sources and further reading

- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition: the chapters on hashing and sorting are prescribed reading in many syllabi.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*: heapsort, quicksort, lower bounds and linear-time sorts, hash tables, and string matching (KMP).
- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*: sorting (including external sorting) and hashing.
- Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th edition: shellsort, hash tables, tries and substring search, with the free companion site.
- Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*: the source of the probe-count formulas for linear probing and of the multiplication-method constant.
- Wikipedia, "Linear probing": en.wikipedia.org/wiki/Linear_probing, for the expected-cost bounds in terms of the load factor (opened Nov 2026).
- Wikipedia, "Knuth–Morris–Pratt algorithm": en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm, for the 1977 date and the Θ(m) + Θ(n) bounds (opened Nov 2026).
- Wikipedia, "Shellsort": en.wikipedia.org/wiki/Shellsort, for Shell's 1959 paper, the gap sequences and their worst-case bounds (opened Nov 2026).
- Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.