




Dev Sanghvi

Scribing.pdf

-  My Files
-  My Files
-  Adani University

Document Details

Submission ID

trn:oid:::20705:110053626

Submission Date

Aug 29, 2025, 1:11 AM CDT

Download Date

Aug 29, 2025, 1:18 AM CDT

File Name

Scribing.pdf

File Size

245.2 KB

14 Pages**7,944 Words****35,621 Characters**

*% detected as AI

AI detection includes the possibility of false positives. Although some text in this submission is likely AI generated, scores below the 20% threshold are not surfaced because they have a higher likelihood of false positives.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Disclaimer

Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (i.e., our AI models may produce either false positive results or false negative results), so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?

The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?

Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.



COMP 480/580 — Probabilistic Algorithms and Data Structures August 29, 2025**Lecture 2***Lecturer: Anshumali Shrivastava**Scribe By: Dev Sanghvi*

Pseudorandom Numbers, Universal Hashing, and Collision Resolution

1 Pseudorandom Number Generators

1.1 True Randomness vs. Pseudorandomness

Random numbers are fundamental to probabilistic algorithms. A truly random sequence may be generated by physical phenomena (flipping coins, electronic noise, quantum effects, etc.). However, such “true” randomness is often impractical to obtain in software at high speed. Instead, we use *pseudorandom number generators* (PRNGs), which produce a deterministic sequence of numbers that “looks” random. A PRNG is initialized with a *seed* value, and thereafter it produces a sequence of outputs by a fixed algorithm. Because the process is deterministic, the same seed will always yield the same sequence. The challenge is to design PRNGs that produce sequences with statistical properties indistinguishable from true randomness for algorithmic purposes.

Desirable Properties: A good PRNG should have a long period (the sequence length before it repeats), and the output should be *uniformly distributed* in the expected range. Ideally, it should pass various statistical tests for randomness. For cryptographic applications, even stronger unpredictability properties are required (a cryptographically secure PRNG must withstand attempts to distinguish its output from a truly random sequence without knowledge of the seed). In our context of probabilistic algorithms, we primarily require that the PRNG’s outputs are uniform and independent enough to simulate random choices.

1.2 The Middle-Square Method

One of the earliest PRNGs, suggested by John von Neumann in 1946, is the **middle-square method**. The algorithm is simple: start with an n -digit seed. To generate the next number, square the current value (producing up to $2n$ digits), then extract the middle n digits of the result as the next output and also as the new seed. For example, with a 4-digit seed 1111, squaring yields $1111^2 = 1234321$, written as 8 digits 01234321. Taking the middle 4 digits gives 2343 as the next pseudorandom number. Repeating the process would then square 2343 to get the next number, and so on.

While simple, the middle-square method has serious flaws. Many sequences generated this way tend to degenerate quickly. For instance, some seeds enter short cycles or even reach 0, after which they will output only zeros forever (e.g. starting from seed 0000 obviously yields 0000 every time). In practice, most seeds eventually fall into a loop or repeat within a relatively small number of steps. Von Neumann was aware of these issues but used the method as a stopgap given the limited computing resources of the time. Today, the middle-square method is mostly of historical interest, having been supplanted by more robust generators.

Exercise: Try the middle-square method with a few different 4-digit seeds (for example, 1000, 3792, 6250). Observe how quickly the sequence repeats or reaches a trivial cycle. Can you find a seed (other than 0000) that causes an immediate repetition in one step?

Interestingly, a recent revival of the middle-square idea combines it with an additional sequence to improve its quality. This brings us to the concept of **Weyl sequences**.

1.3 Weyl Sequences and Linear Congruential Generators

A *Weyl sequence* is a sequence defined by repeated addition of a constant modulo some number. Specifically, choose an integer increment k and modulus m ; start from an initial value X_0 and generate $X_{n+1} = (X_n + k) \bmod m$. If k and m are relatively prime (i.e. $\gcd(k, m) = 1$), then this sequence cycles through all m possible residues before repeating, and the values are *equidistributed* modulo m . In other words, over a full cycle, each value in $0, 1, \dots, m-1$ appears exactly once, so the sequence “fills” the residue space uniformly. This is a direct consequence of number theory: adding a fixed increment k (with $\gcd(k, m) = 1$) permutes the residues mod m .

The Weyl sequence alone produces a simple linear progression modulo m , which is not very random (it’s just an arithmetic progression). However, it can be used to improve other generators. For example, one proposal called the **Middle Square Weyl Sequence** (MSWS) generator combines the middle-square method with a Weyl sequence increment to avoid the pitfalls of the middle-square alone. In MSWS, each iteration squares the state (like the middle-square) and also adds a constant (like a Weyl sequence) before taking the middle bits. The added constant helps ensure the state continues to evolve even if the squared part becomes 0 or stagnant. The C code snippet from the slides:

```
d += 362437;
```

illustrates adding a constant increment each time (here 362437). In that example, 362437 is chosen as an odd integer seed/increment. The reason it must be odd is to ensure it is coprime with 2^{32} (assuming $m = 2^{32}$ for a 32-bit generator). If it were even, then the sequence $d \leftarrow d + 362437 \pmod{2^{32}}$ would only ever produce numbers of the same parity as the seed and would actually cycle through only half of the 2^{32} values (skipping all odd outputs) – not fully utilizing the 32-bit range. With 362437 (odd), the additive sequence has a full period of 2^{32} , achieving equidistribution of outputs in the 32-bit space.

The most widely used class of PRNGs in practice (for non-cryptographic purposes) is the **linear congruential generator (LCG)**. An LCG produces a sequence according to the recurrence:

$$X_{n+1} = (a \cdot X_n + c) \bmod m,$$

with multiplier a , increment c , modulus m , and initial seed X_0 . The Weyl sequence is essentially a special case of an LCG with $a = 1$ (often called an *additive congruential* generator). More generally, by choosing appropriate a, c, m , one can obtain a much longer period and better statistical properties. In fact, a well-chosen LCG can have a period equal to m , meaning it cycles through all residues before repeating. For an LCG to have full period m , it must satisfy the standard Hull-Dobell conditions:

- c is coprime with m (ensuring the additive part can reach all residues).
- $a - 1$ is divisible by all prime factors of m .
- If m is a multiple of 4, then $a - 1$ is divisible by 4.

We won't derive these conditions here, but intuitively the second and third conditions ensure the multiplier a has certain residue cycle properties relative to m . For example, if m is prime, having a be a generator of the multiplicative group mod m (i.e. a has order $m - 1$) along with $c \neq 0$ will achieve full period.

Example: A classic example is the LCG used in the IBM PC library: $m = 2^{32}$, $a = 1664525$, $c = 1013904223$. This generator has a full period 2^{32} and was used in many systems (known as `rand()` in C implementations). However, not all LCGs are good. A notorious bad example was `RANDU` from the 1960s, which used $m = 2^{31}$, $a = 65539$, $c = 0$. Its choice of a caused the sequence to fall into planes in 3D space, failing spectral tests (it turned out $a - 1$ was divisible by 4 but other issues caused severe correlation).

Even with a full-period, LCG outputs can have subtle statistical shortcomings. For instance, using an increment c ensures full coverage of residues (Weyl sequence property), but the sequence still has a linear structure. For this reason, modern general-purpose PRNGs often use more complex algorithms (like **Mersenne Twister**, which has a period of $2^{19937} - 1$ and excellent equidistribution properties in high dimensions, or cryptographic generators using ciphers). Nonetheless, LCGs remain popular for their simplicity and speed.

Exercise: (i) Prove that if $\gcd(k, m) = 1$, the sequence $0, k, 2k, 3k, \dots \pmod{m}$ is uniformly distributed over $\{0, \dots, m - 1\}$. (Hint: show that the first m terms are a permutation of all residues mod m .) (ii) Using the above, explain why adding an odd constant in a modulus 2^n PRNG guarantees hitting all values, whereas adding an even constant would not.

2 Hash Functions and Universal Hashing

2.1 Hashing Basics

Hashing is a technique to map a potentially large universe of keys (e.g. strings, integers, etc.) into a fixed range of indices (slot numbers for a table). A **hash function** $h : U \rightarrow \{0, 1, \dots, m - 1\}$ takes an input key and computes an index in a table of size m . We store the key (and its associated data) at that index. Ideally, h should scatter keys uniformly across the m slots. However, since $|U|$ (universe size) is typically much larger than m , *collisions* are inevitable: two distinct keys $x \neq y$ may have $h(x) = h(y)$. Designing good hash functions and strategies to handle collisions is critical for efficient hashing.

A simple and common choice is the **division method**: pick a table size m and let

$$h(k) = k \bmod m.$$

This is easy to compute (one modulo operation). For example, if $m = 10$ and the key universe is integers, $h(k) = k \bmod 10$ just gives the last decimal digit of k . In choosing m , one should avoid certain values that could lead to non-uniform distributions. Powers of 2 are often poor choices for m because $k \bmod 2^p$ is just the lowest p bits of k . If, say, many keys share the same lower bits (e.g. all keys are even, or all are multiples of 100), they would all hash to a small subset of slots. A prime m not close to a power of 2 is usually recommended. As an example, if keys are character strings interpreted in base 128, a prime m like 10,007 is preferable to $m = 8192$ (which is 2^{13}) to avoid only hashing on a few bits of the sum or value. Another method, the **multiplication method**, avoids picking bad m by multiplying k by a constant $A \in (0, 1)$ and extracting the fractional part times m ; this can distribute keys evenly for any m (Knuth recommends $A \approx (\sqrt{5} - 1)/2$), but it involves floating-point multiplication which can be slower.

Computing a modulo can be slow on some hardware, especially if m is not a power of 2. An optimization (often a discussion point in class) is that if m is a power of 2, say $m = 2^p$, then $h(k) = k \bmod 2^p$ can be computed by taking the p lowest-order bits of k (essentially a bitmask). This eliminates the expensive division operation. *However*, as noted above, using $m = 2^p$ can harm the uniformity of h if the key values exhibit patterns in their lower bits. ****Class Exercise (Mod operation is slow):**** Consider scenarios where using $m = 2^p$ might be acceptable or beneficial. For example, if we know our keys are uniformly distributed 32-bit integers, $m = 2^{10} = 1024$ might work fine and allow using a bitmask. On the other hand, if keys are, say, all multiples of 1024 (such as addresses aligned to 1024 bytes), then $k \bmod 1024$ is always 0, causing a disastrous hash distribution. This exercise highlights the trade-off between computational efficiency and hash quality.

In summary, a good hash function should balance *speed* and *uniformity*. In practice, for general keys (like strings), one might combine operations (bit shifts, multiplications, etc.) to mix the bits of the key thoroughly, then take mod m . For example, a simple string hash might do:

$$h(s) = \left(\sum_i c_i \cdot 128^i \right) \bmod m,$$

where c_i are character codes. This treats the string as a number in base 128. A potential issue here is if m divides a power of 128 minus 1, leading to lots of collisions for anagrams. Using a prime m that is not near a power of 128 mitigates that.

A Simple Interview Question: “How would you design a hash function for telephone numbers?” One answer: take a telephone number (which is essentially a large integer) and mod by a suitable prime. A better answer might note patterns in phone numbers (like area codes) and suggest multiplying parts or using a base representation to ensure all digits influence the hash. The goal is to demonstrate understanding of making a hash function that distributes real-world data uniformly.

2.2 Universal Hashing

One problem with choosing a fixed hash function h is that a clever adversary could deliberately supply many keys that collide (e.g., all keys mapping to the same slot), causing worst-case performance to degrade. **Universal hashing** is a strategy, introduced by Carter and Wegman (1977), to thwart such adversaries by using randomness in the choice of h . Instead of a single static hash function, we design a *family* \mathcal{H} of hash functions with a nice property, and then select one h from \mathcal{H} at random at the start of the algorithm. This way, an adversary doesn’t know exactly which hash function is in use, and for a well-designed family, the chance of heavy collisions can be made low.

Formally, a family of functions $\mathcal{H} = \{h : U \rightarrow [m]\}$ is **universal** if for any two distinct keys $x \neq y$ in U , the probability (over a random choice of h from \mathcal{H}) that x and y collide is at most $1/m$. Equivalently:

$$\forall x \neq y, \quad \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}.$$

In a perfectly random function, the collision probability would be exactly $1/m$, so a universal family achieves (up to constant factors) the ideal collision chance. Sometimes the term **2-universal** (or **pairwise independent**) is used to mean that for any two distinct keys x, y , not only is $\Pr[h(x) = h(y)] = 1/m$, but the two hash values $h(x)$ and $h(y)$ are independent random

variables. In many contexts, “universal” as defined above is sufficient, and it’s the definition we will use here.

The main benefit of universal hashing is a provable guarantee on expected performance. Suppose we build a hash table of size m and choose $h \in \mathcal{H}$ uniformly at random from a universal family. Insert n keys into the table (we’ll discuss collision resolution soon). For any fixed key x , the expected number of other keys that collide with it is at most $(n - 1)/m$. This is because each of the $n - 1$ other keys y has $\Pr(h(y) = h(x)) \leq 1/m$, and by linearity of expectation:

$$E[\text{collisions on } x] = \sum_{y \neq x} \Pr[h(y) = h(x)] \leq \frac{n - 1}{m}.$$

So the expected *chain length* or cluster size for x (including x itself, in case x is in the table) is at most $1 + \frac{n-1}{m}$. If n is on the order of m (load factor $\alpha = n/m$ is a constant), this is $O(1)$. We emphasize this is an expectation taken over the random choice of h (which in turn randomizes the placement of keys). No matter how adversarial the key set is, a random $h \in \mathcal{H}$ will, on average, scatter them enough to keep collisions low.

Conclusion: With universal hashing, the expected time for a search (or insert or delete) in a hash table is $\Theta(1)$, assuming the load factor $n/m = O(1)$. This expected time is *input distribution independent*, meaning it holds even in the worst-case scenario for the keys, as long as our hash function was chosen at random from a universal family.

Designing a universal family \mathcal{H} is an interesting task. A classic and widely used construction for hashing integers is as follows: - Fix a prime number p larger than the maximum possible key value (or at least larger than $|U|$, the universe size). For example, if keys are 32-bit, one might choose a 32-bit prime like $p = 2^{31} - 1 = 2147483647$. - Define $\mathcal{H} = \{h_{a,b} : h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m\}$, where a and b are integers. We let a range over $\{1, 2, \dots, p - 1\}$ (i.e. $a \not\equiv 0 \pmod{p}$) and b range over $\{0, 1, \dots, p - 1\}$. This gives $|\mathcal{H}| = (p - 1) \cdot p$ possible hash functions. - To pick a random h from \mathcal{H} , we just choose a and b uniformly at random from those ranges and use $h_{a,b}$.

It can be shown that this family is universal (in fact, strongly universal/pairwise independent). The intuition: for any two distinct keys $x \neq y$, consider the random variables $h_{a,b}(x)$ and $h_{a,b}(y)$. We have

$$\begin{aligned} h_{a,b}(x) &= (ax + b) \bmod p \bmod m, \\ h_{a,b}(y) &= (ay + b) \bmod p \bmod m. \end{aligned}$$

Because arithmetic is mod p (a prime), the pair $(h(x), h(y))$ for a random choice of a, b ends up uniformly distributed over $[m] \times [m]$. In particular, $\Pr[h(x) = i \wedge h(y) = j] = 1/m^2$ for any i, j , which implies $\Pr[h(x) = h(y)] = 1/m$. We omit the full proof, but it uses the fact that $a(x - y) \bmod p$ is uniform in $\{0, \dots, p - 1\}$ when a is random and not 0, since $x - y \not\equiv 0 \pmod{p}$.

Other universal families exist for different key types (e.g., for strings, one can interpret the string as a polynomial and choose a random mod p evaluation). The important takeaway is that by randomizing the hash function choice, we obtain guarantees on collision probability. Many modern hash table implementations use a fixed hash but rely on randomness in keys or assume average-case distributions. However, truly adversary-resistant implementations (e.g., some hash tables in cryptographic contexts or in programming languages for security) do employ random hashing under the hood.

Exercise: Prove that the family $\mathcal{H} = \{h_{a,b}\}$ defined above is universal, by showing for any fixed $x \neq y$ and any $i, j \in [m]$, $\Pr[h_{a,b}(x) = i \wedge h_{a,b}(y) = j] = 1/m^2$. You will need to use the primeness of p in the argument.

3 Collision Resolution in Hash Tables

No matter how cleverly we design h , collisions will occur whenever we insert multiple keys into a finite table. We need strategies to handle these collisions while maintaining efficient operations. There are two primary classes of collision resolution techniques:

1. **Separate Chaining**, and
2. **Open Addressing**, which includes linear probing, quadratic probing, double hashing, etc.

Before diving into each, let's illustrate the basic problem. Suppose our hash table has size $m = 10$ and $h(k) = k \bmod 10$. We insert keys: 7, 18, 41, 94. These four keys happen to land in distinct slots:

$$7 \bmod 10 = 7, \quad 18 \bmod 10 = 8, \quad 41 \bmod 10 = 1, \quad 94 \bmod 10 = 4.$$

So after these insertions, the table slots 1,4,7,8 are occupied (each containing one of the keys) and there have been no collisions. Now we insert a fifth key, 17. We compute $h(17) = 17 \bmod 10 = 7$. Slot 7 is already occupied (by key 7), so we have a *collision*. We must store 17 somewhere else or in some structure that still allows us to find it quickly. Different strategies handle this scenario differently, as we discuss next.

3.1 Separate Chaining

Index	Chain (separate chaining example)
0	
1	41
2	
3	
4	94
5	
6	
7	7 → 17
8	18
9	

Figure 1: Chaining state for $m = 10$, $h(k) = k \bmod 10$ after inserting 7, 41, 18, 17.

In separate chaining, each slot of the hash table is the head of a linked list (or similar structure) of keys that hash to that slot. If multiple keys collide to the same index, they occupy the chain (linked list) at that index. For example, using the above scenario with chaining, when we insert key 17 and find that slot 7 is already occupied by 7, we simply add 17 to the linked list at slot 7. Figure 1 illustrates this: slot 7's chain will contain $[7 \rightarrow 17]$, slot 1 contains $[41]$, slot 4 contains $[94]$, slot 8 contains $[18]$ (assuming we inserted 18 before 41 and 94 for instance). The slide excerpt below shows an example state:

```

Index:    1    7    8    4    ...
Chain:   41 -> [7 -> 17] 18  94  ...

```


Here, index 7's chain has 7 and 17, index 1's chain has 41, etc..

Searching in a chained hash table is straightforward: to find key x , compute $i = h(x)$, then search the linked list at slot i . If the hash function distributes keys well, these lists will be short on average. In particular, as we derived with universal hashing, the expected length of the chain containing any particular key (or the expected length of an empty-chain search) is at most $1 + \alpha$, where $\alpha = n/m$ is the load factor. When α is low (say $\alpha \approx 1$ or less), chain lengths are $O(1)$ on average, so search and insert remain $O(1)$. In the worst-case (pathological input or bad h), one chain could contain all n keys (making search $O(n)$), but randomness in h prevents this on expectation.

The worst-case time for insert or search in chaining is $O(n)$ (if all keys went to one slot). But if we assume a good hash and/or use universal hashing, the probability of such heavy clustering is extremely low. One can show that if n is on the order of m , the probability of any chain length exceeding, say, $O(\log n)$ is tiny. So with high probability, operations are still quite fast. Thus, chaining gives a kind of stochastic guarantee.

One downside of chaining is the additional memory overhead for pointers (if using linked lists) and potentially poor cache performance. Each lookup may involve chasing a pointer to a linked list node that could be anywhere in memory. In modern architectures, this can incur cache misses. Another issue is that deletion is straightforward in chaining (just remove the item from the list), but the linked list could create memory fragmentation. Still, chaining is simple to implement and works well especially if the keys themselves are stored outside (like storing key-value pairs on the heap and just linking them).

Expected performance: With n keys and table size m , under simple uniform hashing or a universal hash assumption, the expected search time for a key (successful or not) is $O(1 + \alpha)$. The extra 1 term is the time to compute $h(x)$, and the α term is the expected chain length. Insert is $O(1)$ (just add to front of list, assuming we can compute $h(x)$ in $O(1)$). Delete is $O(1)$ given direct access to the list node or $O(\text{chain length})$ if we have to search the chain first to find the item.

If α grows (meaning the table is overloaded), performance degrades linearly with α . So typically, if α becomes too high (say > 2 or > 5 , depending on tolerance), one would increase the table size and rehash (possibly picking a new h) to bring α back down. This is analogous to dynamic array resizing for arrays.

****Exercise:**** Suppose we have $m = 10$ and we insert keys 7, 41, 18, 17 (in that order) into a hash table with separate chaining using $h(k) = k \bmod 10$. Draw the state of the hash table after these insertions. Then, calculate the average chain length and the longest chain length in your result. (Verify with the example above: the longest chain was of length 2 at slot 7, and average chain length was $5/10 = 0.5$ since 5 keys total including the new one, spread over 4 non-empty chains.)

Chaining works very well when n is not too much larger than m . However, if $n \gg m$ (load factor very high), the chains become long and performance degrades (approaching $O(n)$ in the extreme). Also, if memory is at a premium or if we want to avoid pointers (e.g., in memory-constrained or real-time systems), we might prefer to keep all data in the table array itself. This leads us to open addressing.

3.2 Open Addressing

In **open addressing**, all keys are stored in the hash table array itself (no extra chains). When a collision occurs, the key is placed in another slot in the table (the "next available" slot as determined by some probe sequence). Each slot holds at most one key. On insertion, if $h(k)$ is

occupied, we try some other index, and if that's occupied, another, and so on, until we find an empty slot to place the key. Likewise, search probes the sequence of slots until either the key is found or an empty slot is encountered (which indicates the key is not in the table).

Open addressing requires a *probe sequence* for each key. Typically, we have a family of functions h_0, h_1, h_2, \dots such that:

$$h_0(k) = h(k)$$

$$h_i(k) = (h(k) + f(i)) \bmod m, \quad i = 1, 2, 3, \dots$$

where $f(i)$ is some increment as a function of the probe count i . The initial probe is $h_0 = h(k)$, the home bucket. If that is full and k isn't there, we try $h_1(k)$, then $h_2(k)$, and so on. This continues until we either find the key or find an empty slot (which means the key is not present, since if it were, it would have been placed in the first empty slot in its probe sequence during insertion). In an open-addressed table, the load factor $\alpha = n/m$ must always be < 1 (we can't insert more keys than slots, otherwise by pigeonhole principle some slot would hold 2 keys which is not allowed). As α approaches 1, performance deteriorates sharply, so in practice we maintain α below some threshold (like 0.75).

The probe sequence is crucial in determining performance and clustering behavior. We will discuss three common strategies:

1. **Linear Probing:** $f(i) = i$.
2. **Quadratic Probing:** $f(i) = c_1 i + c_2 i^2$ for some constants c_1, c_2 (often simplified as i^2 with certain coefficients).
3. **Double Hashing:** $f(i) = i \cdot g(k)$ for a second hash function g .

Before detailing each, note an important requirement: the probe sequence for a given key k should be a permutation of $\{0, \dots, m-1\}$ (i.e., it should be able to visit every slot in the table) or at least cover all empty slots, otherwise an insertion might fail even if the table isn't full. For example, if m is prime and we use double hashing with $g(k)$ producing a number coprime with m , then $h_i(k) = (h(k) + i \cdot g(k)) \bmod m$ will hit every slot in some order $0 \dots m-1$. If the probe sequence covers only a subset of slots, we could end up in a situation where we loop through that subset and never find an empty slot even though one exists elsewhere.

Linear Probing: Here $h_i(k) = (h(k) + i) \bmod m$. We just check consecutive slots: if $h(k)$ is taken, try $h(k) + 1$, then $h(k) + 2$, etc., wrapping around at the end of the table (because of $\bmod m$). This method is extremely simple and utilizes cache well, since it tends to check contiguous locations in memory. However, a well-known issue is **primary clustering**: once a cluster of filled slots forms, it tends to grow and cause more keys to join it. For example, suppose slots 7,8,9 are filled. If a new key hashes to 7, it will probe to 10 (which wraps to 0) or if 0 also filled, onward; basically it will end up appending to this cluster of occupied slots, making it length 4. Another key hashing anywhere into this cluster (slots 7-0, if we view it circularly) will also extend the cluster. As clusters grow, linear probing search times degrade because any key whose home falls anywhere in the cluster has to scan through it. The phrase "once the primary cluster forms, the bigger it gets, the faster it grows" succinctly describes a positive feedback that leads to large blocks of occupied cells. In fact, for linear probing with high load factor α , the expected search time grows roughly as $O\left(\frac{1}{(1-\alpha)^2}\right)$ in the limit, meaning performance drops sharply as α approaches 1.

That said, at low to moderate load factors (say $\alpha < 0.7$), linear probing can be extremely fast in practice due to good locality (scanning an array is cache-friendly). In expectation (under a simple uniform hashing assumption), one can analyze linear probing's cost: it turns out the expected number of probes for an unsuccessful search is $\frac{1}{1-\alpha}$, and for a successful search it's about $-\frac{1}{\alpha} \ln(1-\alpha)$ (which is $\approx \frac{1}{1-\alpha}$ for moderate α as well) in the uniform hashing model. When α is small, these are close to 1. But as $\alpha \rightarrow 1$, they blow up (e.g., if $\alpha = 0.9$, $1/(1-\alpha) = 10$ expected probes for miss).

Example (Linear Probing): Let's insert keys into an empty table of size $m = 10$ using linear probing. Take the sequence: 38, 19, 8, 109, 10 (from the slides). Using $h(k) = k \bmod 10$:

- Insert 38: $h(38) = 8$. Slot 8 is empty, place 38 at index 8.
- Insert 19: $h(19) = 9$. Slot 9 is empty, place 19 at index 9.
- Insert 8: $h(8) = 8$. Slot 8 is occupied (38 is there). We probe next: index 9, but that's occupied by 19. Probe next: index 0 (since $9 + 1 \bmod 10$ wraps to 0). Slot 0 is free, so place 8 at index 0.
- Insert 109: $h(109) = 109 \bmod 10 = 9$. Slot 9 is occupied (19). Probe index 0: occupied (8). Probe index 1: free, place 109 at index 1.
- Insert 10: $h(10) = 0$. Slot 0 occupied (8). Probe 1: occupied (109). Probe 2: free, place 10 at index 2.

Now the table's occupied slots are [0:8], [1:109], [2:10], [8:38], [9:19]. Notice they ended up clustering: indices 8,9,0,1,2 are all full, forming one large "run" of 5 occupied slots (it wraps around the end to the beginning). A search for any key that hashes into this range (e.g. $h(x) \in \{8, 9, 0, 1, 2\}$) will have to scan through the cluster until it either finds the key or hits the end of the cluster.

If we remove keys, we have to be careful in linear probing: a naive deletion leaving a hole can break the probe chain for keys that were inserted later. The usual solution is to mark deleted slots with a special placeholder (often called a "tombstone") indicating that the slot is empty *for insertion* but not for termination of a search. When searching, tombstones are treated as occupied (so the search doesn't stop), but when inserting, a tombstone can be reused. Alternatively, one can rehash the cluster after a deletion by moving some following elements up, but the tombstone method is simpler. Over time, too many tombstones degrade performance, so periodically one might rebuild the table to remove them.

Quadratic Probing: This strategy avoids the primary clustering issue by using a quadratic function for $f(i)$. A typical scheme is $f(i) = i^2$ (or sometimes $f(i) = i^2 + i$ to ensure distinct values for each i). So the probe sequence is:

$$h_0 = h(k),$$

$$h_1 = (h(k) + 1^2) \bmod m,$$

$$h_2 = (h(k) + 2^2) \bmod m,$$

$$h_3 = (h(k) + 3^2) \bmod m,$$

and so on. For example, if $h(k) = x$, we check $x, x + 1, x + 4, x + 9, x + 16, \dots \bmod m$. These are not contiguous, they start to "jump" further out as i grows, which helps to disperse clusters.

If two keys have different initial $h(k)$, their probe sequences will be different and less likely to stay in lock-step. Thus primary clustering is mitigated. However, **secondary clustering** can still occur: if two keys have the same initial hash $h(k)$, they will follow the exact same probe sequence (since $f(i)$ doesn't depend on the key). They will thus cluster with each other, though their cluster won't necessarily attract keys with different $h(k)$. This is less severe than linear probing's clustering, but it's still a form of clustering around each hash location.

One caution with quadratic probing is that it might not explore all slots. In fact, with $f(i) = i^2$ and certain table sizes, it could cycle through a subset. A common choice (as in CLRS) is to use $f(i) = i^2$ but require m to be prime and also limit the load factor to < 0.5 . Under those conditions, it can be shown that the first $\lceil m/2 \rceil$ probes are all distinct and will find an empty slot if one exists in that range. Intuitively, if m is prime, the values $i^2 \bmod m$ for $i = 1, \dots, \lceil m/2 \rceil$ are all distinct and non-zero (because if $i^2 \equiv j^2 \pmod{m}$ with $i \leq m/2$, $j \leq m/2$, then $(i-j)(i+j) \equiv 0 \pmod{m}$. Since m is prime, either $i = j$ or $i+j = m$. The latter can't happen if both $i, j < m/2$. So $i = j$. Thus no repeats). This means the probe sequence will not hit the same slot twice in the first half of the table. So if the table is at most half filled, an empty slot is guaranteed to be found in that many probes. If the table is more than half full, quadratic probing might fail to find a slot even if one exists, because after $\sim m/2$ probes it potentially repeats positions. In practice, one ensures $n < m/2$ or if not, just resizes the table when it gets that half full.

Using a different quadratic formula can also help; some implementations use $f(i) = i^2$ but also $-i^2$ (i.e., $\pm i^2$) to get more distinct probes. Another strategy is to choose two constants c_1, c_2 and do $h_i = (h + c_1i + c_2i^2) \bmod m$; one must carefully choose c_1, c_2 and m to ensure full coverage. A common simple choice is $h_i = (h + i + i^2) \bmod m$ which gives sequence $h, h+1, h+2, h+4, h+7, h+11, \dots$ (the differences form an arithmetic progression of odd numbers, as seen: 1,2,4,3,4,... wait let's derive: $(i+i^2) - ((i-1) + (i-1)^2) = i+i^2 - (i-1 + (i-1)^2) = i+i^2 - i + 1 - (i^2 - 2i + 1) = 2i - \dots$ Actually, easier: if $f(i) = i^2$, then $f(i+1) - f(i) = 2i+1$, which yields differences 1,3,5,7,... If $f(i) = i + i^2$, then difference is $(i+1 + (i+1)^2) - (i + i^2) = 2i+1$ again! So it's the same differences). So indeed the difference sequence is odd numbers, which in a prime m setting will cycle through about half the residues).

****Example (Quadratic Probing):**** Let's retry the earlier insert sequence (38, 19, 8, 109, 10) with quadratic probing $f(i) = i^2$ and $m = 10$.

- 38: $h(38) = 8$. Place at 8.
- 19: $h(19) = 9$. Place at 9.
- 8: $h(8) = 8$. Slot 8 occupied (38). Now $i = 1$: $(8 + 1^2) \bmod 10 = 9$, slot 9 occupied. $i = 2$: $(8 + 4) \bmod 10 = 2$, slot 2 is free. Place 8 at index 2.
- 109: $h(109) = 9$. Occupied (19). $i = 1$: $(9 + 1) \bmod 10 = 0$, free (notice we jumped to 0 instead of going to $9+1=0$ anyway similar to linear first step). Place 109 at 0.
- 10: $h(10) = 0$. Occupied (109). $i = 1$: $(0 + 1) = 1$, slot 1 free, place 10 at 1.

The final occupancy: index 0:109, 1:10, 2:8, 8:38, 9:19. Interestingly, this is exactly the same set of occupied slots as linear probing gave (0,1,2,8,9). In this particular case the pattern of occupancy is the same, though the order in which elements were placed differs. Did we avoid clustering? We have one cluster in memory (0-2 and 8-9 are contiguous blocks if we consider wraparound). But crucially, note that keys with different home slots diverged sooner. For instance, 109 and 8 both collided with others, but 109 had home 9 and eventually went to 0,

whereas 8 had home 8 and went to 2. Under linear probing, they ended up contiguous (8 went to 0, 109 went to 1, joining the cluster). Under quadratic, 8 and 109 did not follow the exact same trail: 8's second probe was 2, while 109's first probe was 0. Yet ultimately they still ended up adjacent in this case. To see the benefit of quadratic, imagine many keys hashing to 8 and many hashing to 9; with linear, all those keys would form one big cluster. With quadratic, the ones from 8 will probe 8,9,2,7,6,...; the ones from 9 will probe 9,0,3,8,... – these sequences intersect at some points, but not in lockstep for every probe, so they won't all cluster into one contiguous block.

One must still resize the table when it gets too full. Typically, once α exceeds, say, 0.5 or 0.75, it's time to increase m and rehash, because quadratic probing (and linear and double hashing) performance worsens as α grows. High α also risks failing to insert in quadratic if m is not prime or load ≥ 0.5 as explained.

Summary of clustering: Linear probing suffers from *primary clustering* – long runs of occupied slots form, and any key that hashes into the middle of such a run will lengthen it. Quadratic probing avoids primary clustering because the probe sequence is not just a simple progression, so two clusters don't easily join into one large cluster. However, keys that hash to the same index still follow the same pattern, causing *secondary clustering* (keys that start at the same $h(k)$ stay in the same sequence together). Secondary clustering is less severe, since it only affects keys with identical $h(k)$, not those that hash to nearby slots.

Double Hashing: This method uses a second hash function to determine the probe step size. Specifically, one common formula:

$$h_i(k) = (h(k) + i \cdot g(k)) \bmod m,$$

where $g(k)$ is another hash function on the key, with the constraint that $g(k)$ is never 0 (mod m). Typically $g(k)$ is designed to output values in $\{1, \dots, m-1\}$. A simple choice is $g(k) = 1 + (k \bmod (m-1))$. If m is prime, then $g(k)$ will be in $\{1, \dots, m-1\}$ and not all keys will share a common factor with m (in fact any $g(k)$ in that range is automatically coprime with m if m is prime). Double hashing in effect says: the first time you collide, jump $g(k)$ slots ahead, then $2g(k)$, etc., wrapping around mod m . The step size $g(k)$ is fixed for a given key k but is typically different for different keys, so the probe sequences for different keys diverge quickly even if their initial position was the same or nearby.

Double hashing is often considered the best form of open addressing in terms of clustering avoidance, because it eliminates both primary and secondary clustering: even if two keys x and y have $h(x) = h(y)$, they will likely have $g(x) \neq g(y)$, so their probe sequences will be different (they coincide on the first slot, then x goes in steps of $g(x)$, y in steps of $g(y)$, which are distinct mod m). The result is that the table behaves closer to the “simple uniform hashing” ideal. In fact, one can show that if h and g are truly random, double hashing yields expected search/insert times on the order of $1/(1-\alpha)$ similar to random probing, without the extra quadratic factor that linear probing suffers at high α due to clustering.

Continuing our example with double hashing: **Example (Double Hashing):** Use $h(k) = k \bmod 10$ as primary and $g(k) = 1 + (k \bmod 9)$ as secondary. Table size $m = 10$. Insert 38, 19, 8, 109, 10:

- 38: $h(38) = 8$. Index 8 free, place 38.
- 19: $h(19) = 9$. Place at 9.

- 8: $h(8) = 8$ (collision at 8 with 38). Now $g(8) = 1 + (8 \bmod 9) = 1 + 8 = 9$. We probe: $8 + 1 \cdot 9 = 17 \bmod 10 = 7$. Slot 7 is free, place 8 at index 7.
- 109: $h(109) = 9$ (collision with 19). $g(109) = 1 + (109 \bmod 9) = 1 + (109 \bmod 9) = 1 + (1) = 2$ (since $109 = 9 \cdot 12 + 1$). Probe: $9 + 1 \cdot 2 = 11 \bmod 10 = 1$, free, place 109 at 1.
- 10: $h(10) = 0$, slot 0 free, place 10.

Final locations: 0:10, 1:109, 7:8, 8:38, 9:19. Notice now the cluster structure: we have small clusters: [7,8] is filled, [9] by itself, [0,1] is two adjacent slots but they aren't part of a linear probe chain (10 and 109 had different homes). The keys that collided (8 and 38 had same home 8) ended up far apart (8 went to index 7, 38 stayed at 8). So no long run formed. This illustrates double hashing's effectiveness in spreading out keys.

To ensure double hashing covers all slots, we need $g(k)$ to be coprime with m . A common approach: if m is prime, any $g(k)$ in $1 \dots m-1$ works. If m is a power of 2, one could choose $g(k)$ to be an odd number (ensuring it's coprime with 2^p). The formula $g(k) = 1 + (k \bmod (m-1))$ automatically ensures $1 \leq g(k) \leq m-1$. If m is prime, this covers all $m-1$ possible step sizes which are all coprime with m . If m is not prime, some step sizes might not be coprime, so the sequence might cycle through a subset of slots. It's safest to use prime m for double hashing (or ensure $g(k)$ always outputs a number coprime with m by some other means).

One subtlety: what if $g(k) = 1$ for many keys? That effectively reduces to linear probing for those keys. The chance of that depends on the key distribution. If g is well-designed, it shouldn't output the same small number for too many keys. In our chosen $g(k) = 1 + k \bmod (m-1)$, $g(k) = 1$ happens iff $k \bmod 9 = 0$ in the example, so keys like 9,18,27,... would all step by 1 (not great). In practice, a more robust choice of g or making m prime minus a small constant can avoid such coincidences. In a real implementation, one might pick a random secondary hash as well.

Deletion in Open Addressing: Removing a key from an open-addressed table is tricky because simply marking its slot empty may break the search chain for some other key. For example, if we removed 38 from index 8 in the linear probing example, searching for key 8 (which is at slot 0) would go: $h(8) = 8$, finds slot 8 empty, and conclude 8 is not present – which would be wrong. Thus, instead of truly emptying the slot, we often mark it with a special *deleted* marker (tombstone). Tombstones are treated as filled for search (so search doesn't stop) but as empty for insertion (new keys can reclaim that slot). Over time, too many tombstones can cause long probe sequences, so rebuilding or periodic cleanup may be needed.

Open addressing has the advantage that everything is stored in one array (better locality than pointers) and no extra memory overhead per key. It's very efficient when the table is not too full. However, as α approaches 1, performance degrades superlinearly (especially for linear probing). Typically, resizing and rehashing at, say, 70-80

****Exercise:**** (i) Insert the keys [12, 44, 13, 88, 23, 94, 11, 39, 20] into an initially empty table of size $m = 11$ using linear probing. How many probes does each insertion take? What does the table look like at the end? (ii) Repeat the insertion for quadratic probing with $f(i) = i^2$ (and $m = 11$, which is prime). (iii) Repeat for double hashing using $h(k) = k \bmod 11$ and $g(k) = 1 + (k \bmod 10)$. Compare the results and identify any clustering in each case.

3.3 Summary and Further Notes

We have covered the classical collision resolution techniques: - Chaining: easy and effective when space for pointers is not a concern; performance $O(1 + \alpha)$ on average. Worst-case $O(n)$

if hash fails badly or keys fall in one slot. - Linear Probing: very fast for low α due to cache locality, but susceptible to clustering; performance degrades as α increases, worse-case $O(n)$ and expected $\approx \frac{1}{(1-\alpha)^2}$ probes in heavy load scenarios. - Quadratic Probing: avoids primary clustering, only secondary clustering; needs careful table size management to ensure all slots can be reached. Usually good up to moderate load factors. - Double Hashing: best distribution of probes if h and g are good; avoids both primary and secondary clustering by making probe sequences key-dependent. Often gives performance close to uniform hashing theory, expected $\approx \frac{1}{1-\alpha}$ probes for lookups under simple model. Harder to implement deletion correctly; g must be chosen carefully.

There are other advanced methods as well. For example, **Cuckoo Hashing** uses two hash functions and allows relocating keys on collisions, guaranteeing $O(1)$ worst-case lookup by maintaining at most one key per slot with a possibility of rehashing on failure. **2-choice hashing** (or balanced allocations) chooses two h functions and inserts the key into the less loaded of two candidate slots, greatly reducing max chain length (this is a different paradigm requiring some coordination but is interesting theoretically). There are also perfect hashing schemes for static sets that achieve $O(1)$ worst-case search with no collisions by carefully choosing h after seeing the keys (at the cost of space).

However, those are beyond the scope of this lecture. In practice, for general hash tables:

- Use a decent hash function (possibly a universal or at least one with good mixing).
- Use separate chaining or open addressing depending on use case (chaining for easier unpredictability handling and simpler deletes, open addressing for memory efficiency and cache performance).
- Maintain load factor below a threshold by resizing the table (e.g., double the size when α exceeds 0.75).
- If adversarial keys are a concern, use a random seed in the hash (universal hashing) or more sophisticated schemes.

Finally, let's tie back to the theoretical guarantees: by using universal hashing (randomized h) and any collision resolution (chaining or open addressing with double hashing, say), we can achieve *expected* $O(1)$ time per operation for a hash table. This is why hashing is such a powerful technique in randomized algorithms and data structures: with high probability, operations are very fast, beating structures like balanced BSTs which are $O(\log n)$. The randomness in hashing protects against worst-case sequences of operations, making performance reliably good on average.

References

- [1] Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. (Chapter 11: Hash Tables – includes discussions on universal hashing and open addressing).
- [2] Motwani, R., Raghavan, P. (1995). *Randomized Algorithms*. (Section 5: Hashing – covers universal hashing and its use in randomized data structures).
- [3] Knuth, D. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. (Chapter 6.4: Hashing – classical analysis of hashing methods; also Volume 2 covers random number generation).
- [4] Carter, J., Wegman, M. (1977). “Universal Classes of Hash Functions.” *Journal of Computer and System Sciences*, 18(2): 143–154.

- [5] Pagh, R. et al. (2012). “On the cell probe complexity of dynamic membership.” (*Discusses clustering in hash tables and lower bounds*).
- [6] Matsumoto, M., Nishimura, T. (1998). “Mersenne Twister: A 623-dimensionally equidistributed uniform PRNG.” *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30.