DAA Theory Notes — Unit V

Complexity Theory and Hashing

PART A — COMPLEXITY THEORY

---

1. Computational Complexity — Basic Concepts

1. Computational complexity is the study of the **resources (time and space)** required to solve a problem, and it classifies problems based on how difficult they are to solve algorithmically.
2. Problems are divided into two broad groups — **polynomial time problems** whose solution time is bounded by a polynomial in input size n (such as O(n), O(n log n), $O(n^2)$), and **non-polynomial problems** whose best known algorithms take exponential or factorial time.
3. Examples of polynomial-time problems: Linear Search O(n), Binary Search O(log n), Heap Sort O(n log n), Matrix Multiplication $O(n^3)$.
4. Examples of non-polynomial problems: Travelling Salesman Problem $O(n^2 \times 2^n)$, 0/1 Knapsack $O(2^n)$, Hamiltonian Cycle O(n!).
5. The goal of complexity theory is to determine whether non-polynomial problems can ever be solved in polynomial time, and to **classify problems by their inherent difficulty**.

---

2. Deterministic Algorithms

1. A **deterministic algorithm** is one where the result of every operation is **uniquely defined** — given the same input, it always follows exactly the same sequence of steps and produces the same output.
2. There is no ambiguity or randomness — at every step, only one possible next action exists.
3. All conventional programs running on real computers are deterministic — the machine always knows exactly what to do next.
4. **Time complexity** of a deterministic algorithm is measured by the number of basic operations performed as a function of input size n.
5. Examples: Merge Sort, Binary Search, Dijkstra's Algorithm — all produce the same output for the same input every single time they are run.

---

3. Non-Deterministic Algorithms

1. A **non-deterministic algorithm** is one where the result of certain operations is **not**

uniquely defined — the algorithm can "guess" or "choose" from a set of possibilities at certain steps.

2. Three special functions are used in non-deterministic algorithms:
   - **Choice(S)** — arbitrarily chooses one element from set S and returns it. Time complexity O(1).
   - **Failure()** — signals that the current sequence of choices has led to an unsuccessful termination.
   - **Success()** — signals that the current sequence of choices has led to a successful termination.
3. A non-deterministic algorithm is said to **succeed** if there exists at least one sequence of choices that leads to a Success() — the algorithm always makes the right guess when a successful path exists.
4. A non-deterministic algorithm **fails** only if no sequence of choices leads to Success() — meaning the problem truly has no solution for that input.
5. Non-deterministic machines do not exist in practice — they are a **theoretical model** used to reason about problem difficulty and complexity classes.

---

4. Non-Deterministic Searching Algorithm

1. The non-deterministic search algorithm searches for element x in an unordered array A[1:n] by **guessing the correct index** in O(1) time.
2. It calls Choice(1, n) to guess a position j, then checks if A[j] equals x — if yes it calls Success(), otherwise calls Failure().
3. The non-deterministic complexity of search is **O(1)** because the guess is always correct when a solution exists.
4. In contrast, the best deterministic search on an unordered array requires **Ω(n)** comparisons in the worst case.
5. This illustrates the power of non-determinism — it can solve in constant time what takes linear time deterministically.

```
Algorithm NDSearch(A, n, x)
{
    j = Choice(1, n);        // Guess the index non-deterministically
    if (A[j] == x)
    {
        write(j);
        Success();
    }
    write(0);
    Failure();
}
```

5. Non-Deterministic Sorting Algorithm

1. The non-deterministic sorting algorithm sorts n positive integers by **guessing the correct sorted permutation** in O(n) non-deterministic time.
2. It uses Choice(1, n) to assign each element A[i] to a guessed position B[j] in array B — if any position j is already occupied, it calls Failure() immediately.
3. After all elements are placed, it **verifies** that B is sorted by scanning once — if any B[i] > B[i+1], it calls Failure().
4. If the entire array passes the sorted check, it writes B[1:n] and calls **Success()**.
5. Non-deterministic complexity is **O(n)** — the best deterministic sorting algorithms require O(n log n), showing again how non-determinism provides theoretical speedup.

```
Algorithm NDSort(A, n)
{
    for i = 1 to n do B[i] = 0;        // Initialize B to 0
    for i = 1 to n do
    {
        j = Choice(1, n);              // Guess position for A[i]
        if (B[j] != 0) Failure();      // Position already taken
        B[j] = A[i];
    }
    for i = 1 to n-1 do
        if (B[i] > B[i+1]) Failure();  // Not sorted
    write(B[1:n]);
    Success();
}
```

---

6. Complexity Class P

1. **Class P** (Polynomial time) is the set of all **decision problems** that can be solved by a **deterministic algorithm in polynomial time** — i.e., in $O(n^k)$ for some constant k.
2. Problems in P are considered **tractable** or **efficiently solvable** — they can be practically solved even for large inputs.
3. Examples of problems in P: Searching O(log n), Sorting O(n log n), Shortest Path (Dijkstra's $O(n^2)$), MST (Prim's $O(n^2)$), Matrix Multiplication $O(n^3)$.
4. The key property of P is that **both solving and verifying** the solution can be done in polynomial time on a deterministic machine.
5. P is a subset of NP — every problem that can be solved in polynomial time can certainly be verified in polynomial time.

7. Complexity Class NP

1. **Class NP** (Non-deterministic Polynomial time) is the set of all **decision problems** that can be **solved by a non-deterministic algorithm in polynomial time** — equivalently, problems whose solutions can be **verified in polynomial time** by a deterministic algorithm.
2. The key insight is the **two-phase nature of NP** — a non-deterministic "guess" phase that generates a candidate solution, and a deterministic "verify" phase that checks whether the candidate is correct in polynomial time.
3. A problem is in NP if given a **certificate** (a proposed solution), you can verify its correctness in polynomial time — even if finding that solution from scratch takes much longer.
4. Examples: 0/1 Knapsack (verify: check sum of weights $\leqslant M$ and sum of profits $\geqslant$ target), Hamiltonian Cycle (verify: check if given path visits all vertices exactly once and returns), Clique (verify: check all pairs in proposed subset are connected).
5. It is currently **unknown whether P = NP** — this is one of the greatest open problems in computer science. If P = NP, every problem whose solution can be verified quickly can also be solved quickly.

---

8. NP-Hard Problems

1. A problem H is called **NP-Hard** if **every problem in NP can be polynomial-time reduced to H** — meaning H is at least as hard as the hardest problems in NP.
2. An NP-Hard problem does **not need to be in NP itself** — it may not even be a decision problem or have a verifiable solution in polynomial time.
3. Informally, NP-Hard means "at least as hard as NP-Complete" — solving an NP-Hard problem in polynomial time would imply P = NP.
4. Examples of NP-Hard problems: Halting Problem, Travelling Salesman Problem (optimization version), 0/1 Knapsack (optimization version).
5. The concept of NP-Hardness is established through **polynomial-time reduction** — if a known NP-Hard problem L reduces to H, then H is also NP-Hard.

---

9. NP-Complete Problems

1. A problem Q is **NP-Complete** if it satisfies **two conditions simultaneously** — it is in NP (solution verifiable in polynomial time) AND it is NP-Hard (every NP problem reduces to it in polynomial time).
2. NP-Complete problems are the **hardest problems within NP** — if any NP-Complete problem can be solved in polynomial time, then ALL problems in NP can be solved in polynomial time (i.e., P = NP).
3. The **first NP-Complete problem** was the **Satisfiability Problem (SAT)**, proved by Cook in 1971 using Cook's Theorem — it was proved NP-Complete without needing to reduce from

another NP-Complete problem.

4. All other NP-Complete problems are proved by **reducing from a known NP-Complete problem** (like SAT, 3-SAT, or Clique) — if the known NPC problem reduces to Q, then Q is NP-Hard, and if Q is also in NP, then Q is NP-Complete.
5. Examples of NP-Complete problems: SAT, 3-SAT, Clique, Vertex Cover, Hamiltonian Cycle, Graph Coloring, Subset Sum.

---

10. Relationship between P, NP, NP-Hard, and NP-Complete

Class	Definition	Example Problems
P	Solved deterministically in polynomial time	Sorting, Searching, Shortest Path
NP	Solution verifiable in polynomial time	Knapsack, Hamiltonian Cycle, Clique
NP-Hard	Every NP problem reduces to it; may not be in NP	TSP (optimization), Halting Problem
NP-Complete	Both in NP AND NP-Hard — hardest problems in NP	SAT, Clique, Vertex Cover

1. **P** $\subseteq$ **NP** — every problem solvable in polynomial time is also verifiable in polynomial time.
2. **NP-Complete** $\subseteq$ **NP** — all NP-Complete problems are inside NP by definition.
3. **NP-Complete** $\subseteq$ **NP-Hard** — every NP-Complete problem is also NP-Hard.
4. If **P = NP**, then P = NP = NP-Complete — all these classes collapse into one.
5. If **P** $\neq$ **NP** (widely believed to be true), NP-Complete problems have no polynomial-time solution.

---

11. Polynomial-Time Reducibility

1. A problem L is **polynomial-time reducible** to problem M (written $L \leqslant_p M$) if there exists a function f computable in polynomial time that transforms every instance x of L into an instance f(x) of M such that x is a YES-instance of L if and only if f(x) is a YES-instance of M.
2. The key property is that **the answer is preserved** — the transformation converts the problem correctly without changing the yes/no answer.
3. If $L \leqslant_p M$ and M can be solved in polynomial time, then L can also be solved in polynomial time — M is "at least as hard as" L.

4. Reductions are used to **prove NP-Hardness** — to show Q is NP-Hard, take a known NP-Complete problem R and show $R \leqslant_p Q$ (R reduces to Q, meaning Q is at least as hard as R).
5. The three main techniques for reduction are **restriction** (showing the known NPC problem is a special case of Q), **local replacement** (converting basic units of one problem to another), and **component design** (building components that enforce structural properties).

---

12. Cook's Theorem — SAT is NP-Complete

1. **Cook's Theorem (1971)** states that the **Boolean Satisfiability Problem (SAT) is NP-Complete** — this was the first problem ever proved to be NP-Complete.
2. **SAT problem:** Given a Boolean formula with n variables and m clauses in Conjunctive Normal Form (CNF), determine whether there exists an assignment of TRUE/FALSE to the variables that makes the entire formula evaluate to TRUE.
3. SAT is in **NP** because given a proposed assignment of variables, we can verify in O(nm) polynomial time whether the formula is satisfied.
4. SAT is **NP-Hard** because Cook proved directly (using Turing Machine simulation) that every NP problem can be polynomially reduced to SAT — without needing another NP-Complete problem to reduce from.
5. **3-SAT** is a restricted version of SAT where every clause has exactly 3 literals — 3-SAT is also NP-Complete and is commonly used as the starting point for proving other problems NP-Complete.

---

13. Steps to Prove a Problem Q is NP-Complete

1. **Step 1 — Show Q is in NP:** Describe a **certificate** (a compact proposed solution) and show that it can be **verified in polynomial time** by a deterministic algorithm.
2. **Step 2 — Choose a known NP-Complete problem R:** Pick an already-proven NP-Complete problem such as SAT, 3-SAT, Clique, or Vertex Cover to reduce from.
3. **Step 3 — Construct a polynomial-time reduction from R to Q:** Show how to transform every instance of R into an instance of Q in polynomial time such that R has a YES answer $\Leftrightarrow$ Q has a YES answer.
4. **Step 4 — Prove correctness of the reduction:** Prove both directions — if R is YES then Q is YES (forward direction), and if Q is YES then R is YES (backward direction).
5. **Step 5 — Conclude:** Since $R \leqslant_p Q$ and R is NP-Complete, Q is NP-Hard. Combined with $Q \in$ NP from Step 1, Q is **NP-Complete**.

---

14. Clique Problem is NP-Complete

1. A **clique** in an undirected graph G = (V, E) is a subset $V' \subseteq V$ such that **every pair of vertices in V' is connected by an edge** — a complete subgraph. The CLIQUE problem asks:

does G contain a clique of size k?

2. **CLIQUE ∈ NP:** Certificate is the proposed set V' of k vertices. Verifier checks for every pair (u, v) in V' whether edge (u,v) ∈ E — this takes $O(k^2)$ time which is polynomial.
3. **CLIQUE is NP-Hard:** Proved by reducing **3-CNF-SAT to CLIQUE**. Given a 3-CNF formula φ with k clauses, construct graph G as follows — for each clause $C_r = (l_{1r} \vee l_{2r} \vee l_{3r})$, create three vertices v_{1r}, v_{2r}, v_{3r}. Add an edge between v_{ir} and v_{js} if and only if $r \neq s$ (different clauses) AND the literals l_{ir} and l_{js} are **consistent** (l_{ir} is not the negation of l_{js}).
4. **Correctness:** φ is satisfiable ⇔ G has a clique of size k. (⇒) A satisfying assignment makes at least one literal true per clause — pick one true literal per clause, the corresponding k vertices form a clique because they are in different triples and their literals are consistent. (⇐) A k-clique selects exactly one vertex per triple — assign those literals to TRUE, satisfying all k clauses.
5. The reduction runs in **polynomial time** ($O(k^2)$ vertices and edges), so CLIQUE is NP-Complete.

---

15. Vertex Cover Problem is NP-Complete

1. A **vertex cover** of an undirected graph G = (V, E) is a subset V' ⊆ V such that **every edge (u, v) ∈ E has at least one endpoint in V'** — every edge is "covered" by at least one vertex in V'. The VERTEX-COVER problem asks: does G have a vertex cover of size ⩽ k?
2. **VERTEX-COVER ∈ NP:** Certificate is the proposed set V' of k vertices. Verifier checks for every edge (u, v) whether u ∈ V' or v ∈ V' — this takes $O(|E|)$ time which is polynomial.
3. **VERTEX-COVER is NP-Hard:** Proved by reducing **CLIQUE to VERTEX-COVER**. Given an instance ⟨G, k⟩ of CLIQUE, compute the **complement graph $\bar{G}$** (same vertex set V, but $\bar{G}$ contains exactly those edges NOT in G). Output $\langle \bar{G}, |V| - k \rangle$ as the VERTEX-COVER instance.
4. **Correctness:** G has a clique of size k ⇔ $\bar{G}$ has a vertex cover of size $|V| - k$. (⇒) If V' is a k-clique in G, then in $\bar{G}$ no edge exists within V' (since all pairs in V' were edges in G, not in $\bar{G}$) — so every edge of $\bar{G}$ must have at least one endpoint in $V \setminus V'$, making $V \setminus V'$ a vertex cover of size $|V| - k$. (⇐) If $\bar{G}$ has vertex cover C of size $|V| - k$, then $V \setminus C$ has size k — for any two vertices $u, v \in V \setminus C$, edge (u,v) is not covered by C, so (u,v) cannot be in $\bar{G}$, meaning (u,v) must be in G — so $V \setminus C$ is a k-clique in G.
5. The reduction runs in **polynomial time** (computing complement graph takes $O(|V|^2)$), so VERTEX-COVER is NP-Complete.

---

16. Differentiate: Deterministic vs Non-Deterministic Algorithms

Feature	Deterministic Algorithm	Non-Deterministic Algorithm
Operation result	Every operation has a uniquely defined result	Operations can "choose" from a set of possible results
Execution path	Always follows the same path for the same input	Can follow different paths; always picks the successful one
Machine model	Runs on a real conventional computer	Runs on a theoretical non-deterministic machine (does not exist in practice)
Complexity	Time measured by actual operations performed	Time measured assuming the best possible guess is always made
Purpose	Solves problems on real hardware	Theoretical model to classify problem difficulty
Examples	Merge Sort, Binary Search, Dijkstra's	NDSearch, NDSort, NDKnapsack

---

PART B — HASHING

---

17. Introduction to Hashing — Why Hashing?

1. A **sequential search** takes O(n) comparisons which is too slow for large databases, and a **binary search** takes O(log n) but requires data to be sorted beforehand.
2. Hashing provides **O(1) average-case access time** for search, insert, and delete operations — without needing the data to be sorted.
3. In hashing, the **address (index) of a record is computed directly from its key** using an arithmetic function called the hash function — no sequential scanning is needed.
4. The function f(x) = h(key) is called the **hash function** and the table used to store records is called the **hash table** — the computed address is called the **home address** of the record.
5. Best case time complexity of hashing is **O(1)** (no collision) and worst case is **O(n)** (all keys collide at same address and must be searched linearly).

---

18. Hashing — Key Terminology

1. **Hash Table** — an array of fixed size MAX, indexed from 0 to MAX-1, where records are stored based on their computed hash values.

2. **Hash Function** — a function h(key) that transforms a key into an integer index within the range [0, MAX-1] — it maps the key to a home address in the table.
3. **Bucket** — an index position in the hash table that can store one or more records — when multiple keys hash to the same address, all are stored in the same bucket.
4. **Collision** — the situation when two different keys produce the same hash value, meaning both are mapped to the same home address in the table.
5. **Synonym** — two or more keys that hash to the same address are called synonyms of each other.
6. **Probe** — each individual action of computing a hash address and checking whether it is the target or is empty is called one probe.
7. **Overflow** — occurs when a new key hashes to a bucket that is already completely full — when bucket size is 1, collision and overflow happen simultaneously.
8. **Open Hashing (External Hashing)** — a scheme where records can be stored in potentially unlimited space outside the table (e.g., using linked lists) — no fixed space limit.
9. **Closed Hashing (Internal Hashing)** — a scheme where all records must be stored within the fixed-size hash table itself — limits the total number of records.
10. **Load Factor** — the ratio of the number of records currently stored to the total capacity of the hash table, expressed as a percentage — a high load factor increases collision probability.
11. **Load Density** — the maximum storage capacity of the hash table — the total number of records it can accommodate.
12. **Full Table** — a hash table where all positions are occupied — further insertions require overflow handling or rehashing.
13. **Perfect Hash Function** — a hash function that maps each key to a unique address with no collisions at all — only achievable when all keys are known in advance.
14. **Rehashing** — the process of creating a new, larger hash table and re-inserting all existing records using a new or modified hash function — done when load factor becomes too high.

---

19. Properties of a Good Hash Function

1. The hash function must produce **uniformly distributed addresses** — all slots in the table should have an equal probability of being mapped to, minimizing clustering.
2. **Small changes in the key** must produce large changes in the hash value — similar keys should hash to very different addresses to avoid clustering.
3. The function must be **fast to compute** — O(1) time to calculate h(key) for any key.
4. The function must **minimize collisions** — the fewer synonyms generated, the better the performance.
5. The function must be **deterministic** — the same key must always produce the same hash value every time it is computed.

---

20. Hash Functions — Division Method

1. The division method computes: **h(key) = key mod M**, where M is the size of the hash table.
2. It is the **simplest and most commonly used** hash function — easy to implement and fast to compute.
3. The choice of M matters — M should be a **prime number** not close to a power of 2 or 10, as this gives the best distribution.
4. **Example:** key = 31, M = 10 → h(31) = 31 mod 10 = 1. Key = 42 → h(42) = 2. Key = 35 → h(35) = 5.
5. The **disadvantage** is that if many keys share a common factor with M, they all hash to the same few addresses — choosing M as a prime minimizes this problem.

---

21. Hash Functions — Multiplication Method

1. The multiplication method computes: **h(k) = ⌊ M × (k × A mod 1) ⌋**, where A is a constant with 0 < A < 1 and "mod 1" means taking only the fractional part.
2. Donald Knuth suggested using **A ≈ 0.6180339887** (the golden ratio conjugate) as the optimal constant.
3. The **advantage** is that the value of M is not critical — the distribution is good for any M, unlike the division method.
4. The **disadvantage** is that it is **slower than division** due to the floating-point multiplication and fractional extraction.
5. **Example:** k = 107, M = 50, A = 0.618 → 107 × 0.618 = 66.126 → fractional part = 0.126 → 50 × 0.126 = 6.3 → h(107) = ⌊6.3⌋ = **6**.

---

22. Hash Functions — Mid-Square, Extraction, Folding

Mid-Square Method:

1. Square the key and **extract the middle digits** of the result as the hash address.
2. Works best when the key is small (⩽ 4 digits) — for large keys, the square becomes too large to store.
3. **Example:** key = 312 → 312^2 = 97344 → extract middle 3 digits → h = 734.
4. If the key is large, use only a **selected portion** of the key for squaring instead of the whole key.

Extraction Method:

1. **Select and extract specific digits** from the key directly and use them as the hash address — no arithmetic needed.
2. The digits are selected based on which positions provide the most uniform distribution.

3. **Example:** key = 345678 → extract positions 1, 3, 5 → h = 357.

Folding Method:

1. Divide the key into equal-sized **subparts** and add them together to get the hash address.
2. **Fold Shift:** Divide key into parts, add all parts directly. Example: key = 987654321 → 987 + 654 + 321 = 1962 → discard leading digit → h = 962.
3. **Fold Boundary:** The left and right parts are **reversed (folded)** before adding to the centre. Example: key = 987654321 → reversed left 789 + centre 654 + reversed right 123 = 1566 → discard → h = 566.
4. Folding is useful when the key is long and we want all parts to contribute to the address.

---

23. Universal Hashing

1. **Universal hashing** selects the hash function **randomly at runtime** from a carefully designed family (set) of hash functions — instead of using one fixed function.
2. Because the function is chosen randomly, the algorithm can behave differently on each execution even for the same input — making it harder for any specific input to cause worst-case performance.
3. The key property is that for any two distinct keys k_1 and k_2, the probability that $h(k_1) = h(k_2)$ (collision) is at most **1/M** when a function is chosen uniformly at random from the family.
4. This approach **guarantees good average-case performance** regardless of the input distribution — no adversary can craft a bad input that always causes collisions.
5. Universal hashing is used in practice for security-sensitive applications where an attacker might try to force worst-case behavior by choosing keys that all collide.

---

24. Collision Resolution — Open Addressing

1. **Open addressing** is a collision resolution strategy where all records are stored **within the hash table itself** — when a collision occurs, another empty slot in the table is found by probing.
2. When h(key) is occupied, a **probe sequence** is computed to find the next candidate slot — the table is probed until either an empty slot is found or the entire table has been searched.
3. Open addressing is also called **closed hashing** because all records are stored inside the fixed-size table — no external linked lists.
4. Three main probing strategies exist: **Linear Probing, Quadratic Probing, and Double Hashing** — they differ in how they compute the next probe address.
5. The main challenge of open addressing is **clustering** — when many occupied slots are adjacent, new insertions require many probes to find an empty slot.

---

25. Linear Probing

1. In **linear probing**, when a collision occurs at position h(key), the next positions h(key)+1, h(key)+2, h(key)+3, … (mod MAX) are probed sequentially until an empty slot is found.
2. **Probe formula:** (h(key) + i) mod MAX, where i = 1, 2, 3, … increases by 1 each time.
3. The major problem with linear probing is **primary clustering** — a long run (cluster) of occupied consecutive slots forms, making future insertions progressively slower as they must probe through the entire cluster.
4. **Linear Probing Without Replacement:** When a collision occurs, the new key always moves to the next available empty slot regardless of which key currently occupies the home address.
5. **Linear Probing With Replacement:** When a collision occurs and the occupying key is NOT at its own home address, the new key takes the home address slot and the displaced key is moved to the next available empty slot — reduces clustering and improves search performance.

```
Algorithm Insert_Linear_Probe(hashtable[], key)
{
    loc = key mod MAX;
    if (hashtable[loc] == -1)
        hashtable[loc] = key;   // Empty slot, insert directly
    else
    {
        i = (loc + 1) mod MAX;
        while (i != loc)
        {
            if (hashtable[i] == -1)
            {
                hashtable[i] = key;
                return;
            }
            i = (i + 1) mod MAX;
        }
        print "Hash table is full";
    }
}
```

---

26. Quadratic Probing

1. In **quadratic probing**, when a collision occurs at position h(key), the next positions are computed as h(key) + 1^2, h(key) + 2^2, h(key) + 3^2, … (mod MAX) — the offset increases quadratically.

2. **Probe formula:** $(h(key) + i^2) \bmod MAX$, where $i = 1, 2, 3, \ldots$
3. Quadratic probing **eliminates primary clustering** because probes spread out across the table rather than scanning consecutive slots.
4. However, quadratic probing can suffer from **secondary clustering** — keys with the same home address follow the same probe sequence and cluster together.
5. Quadratic probing requires that the table size MAX is **prime** and the load factor is below 0.5 to guarantee that an empty slot will always be found within (MAX-1)/2 probes.

---

27. Double Hashing

1. **Double hashing** uses **two separate hash functions** — the first determines the home address, and the second determines the step size for probing when a collision occurs.
2. **Formulas:** $h1(key) = key \bmod MAX$ (home address), $h2(key) = R - (key \bmod R)$ where R is a **prime number smaller than MAX** (step size).
3. **Probe sequence:** $(h1(key) + i \times h2(key)) \bmod MAX$, for $i = 0, 1, 2, \ldots$
4. Double hashing **eliminates both primary and secondary clustering** — different keys with the same home address use different step sizes, so they follow different probe sequences.
5. It provides the **best distribution** among all open addressing methods, approaching the theoretical ideal of random probing, but is slightly slower due to computing two hash functions.

---

28. Separate Chaining (Open Hashing)

1. **Separate chaining** handles collisions by maintaining a **linked list at each bucket** — all synonyms (keys hashing to the same address) are stored in the same linked list.
2. Each slot in the hash table acts as the **head of a linked list** — when a key collides, a new node is appended to the chain at that index.
3. Search requires computing h(key) to find the correct chain, then **sequentially traversing** the linked list to find the target key.
4. Separate chaining can handle an **unlimited number of synonyms** — there is no constraint on how many keys can hash to the same address.
5. The **disadvantage** is extra memory needed for pointers in each node, and sequential scanning of chains increases search time when chains become long.

```
Algorithm Insert_Chain(hashtable[], key)
{
    loc = key mod MAX;
    q = new node;
    q->data = key;
    q->next = NULL;
    if (hashtable[loc]->next == NULL)
        hashtable[loc]->next = q;    // First element in chain
    else
    {
        p = hashtable[loc];
        while (p->next != NULL)
            p = p->next;
        p->next = q;                 // Append to end of chain
    }
}
```

---

29. Open Addressing vs Separate Chaining

Feature	Open Addressing	Separate Chaining
Storage location	All records stored inside the hash table	Records stored in linked lists outside the table
Memory	No extra pointer memory needed	Requires extra memory for pointers in each node
Overflow handling	Probing within the table (linear/quadratic/double)	Linked list at each bucket grows as needed
Maximum records	Limited by table size — table can become full	Unlimited — chains can grow as long as needed
Search performance	Faster when load factor is low (few probes)	Slower when chains become long — sequential scan
Deletion	Complicated — may break probe chain	Simple — just remove node from linked list
Best suited for	When memory is limited and load factor is kept low	When number of records is unpredictable or very large

---

30. Hash Table Overflow and Rehashing

1. **Overflow** occurs when a new key hashes to a **completely full bucket** — it cannot be inserted at its home address or any probed address within the current scheme.
2. For open addressing, overflow is handled by **linear probing, quadratic probing, or double hashing** to find the next available slot in the table.
3. For chaining, overflow is handled by **extending the linked list** at the overflowing bucket — the chain simply grows longer.
4. **Rehashing** is the process of creating a **new, larger hash table** (typically double the original size, rounded to the next prime) and re-inserting all existing records using the new table size.
5. Rehashing is triggered when the **load factor exceeds approximately 70–80%** — at this point, collision rates become high enough to significantly degrade performance.
6. **Time complexity of rehashing: O(N)** where N is the current number of records — it is expensive but done rarely, so amortized cost per insertion remains O(1).

---

31. Summary — Hashing Complexity and Comparison

Operation	Best Case	Average Case	Worst Case
Insert (no collision)	O(1)	O(1)	O(n)
Search (no collision)	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)
Rehashing	—	O(n)	O(n)

Hash Function	Advantage	Disadvantage
Division (key mod M)	Simple, fast	Sensitive to choice of M
Multiplication	M not critical	Slower than division
Mid-square	All digits participate	Fails for large keys
Folding	Good for long keys	More computation
Extraction	Very fast	May not distribute well
Universal	Guaranteed average performance	More complex to implement

---

End of Unit V — Complexity Theory and Hashing Theory Notes All the best!