

LECTURE 8 OF 12

Hash Tables

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 3.4 & 3.5

Duration: 4 hours (with breaks)

Learning Objectives

1. Understand the principles of hash function design and analyze the probability of collisions via the birthday paradox, enabling informed choices when implementing hash-based data structures.
 2. Implement a separate chaining hash table with linked-list buckets, including insert, search, and delete operations, and reason about expected chain length as a function of load factor.
 3. Implement a linear probing hash table using open addressing, handling deletions correctly through tombstone markers or Knuth's backward-shift algorithm, and analyze clustering behavior.
 4. Derive and apply the expected-cost formulas for search hits and misses under the uniform hashing assumption for both separate chaining and linear probing.
 5. Evaluate practical considerations such as the hashCode() contract, algorithmic complexity attacks, universal hashing families, and advanced schemes including cuckoo hashing and Robin Hood hashing.
 6. Compare hash tables against all previously studied symbol table implementations (sequential search, binary search, BST, red-black BST) across multiple dimensions and select the appropriate implementation for a given application.
-

Table of Contents

1. Hash Functions
 2. Separate Chaining
 3. Linear Probing
 4. Analysis of Hashing
 5. Practical Considerations
 6. Symbol Table Implementation Comparison
 7. Practice Problems
 8. Summary
 9. References
-

1. Hash Functions

Estimated time: 30 minutes

1.1 The Fundamental Idea

Every symbol table implementation we have studied so far relies on comparing keys. Sequential search compares keys one by one. Binary search compares keys to narrow a sorted range. Binary search trees and red-black BSTs compare keys to navigate a tree structure. All of these approaches are inherently limited by the information-theoretic lower bound of $\Omega(\log N)$ comparisons for search and insert in the comparison-based model.

Hashing takes a fundamentally different approach. Instead of comparing keys to each other, we compute an arithmetic function of the key that tells us directly where to look in an array. If we can compute this function in constant time, and if the function distributes keys uniformly across the array, we can achieve expected constant-time search and insert -- a dramatic improvement over any comparison-based method.

The core idea is simple: use the key itself to compute an array index. Given a key k and a table of size M , we compute a hash function $h(k)$ that returns an integer in the range $[0, M-1]$. We then store the key-value pair at (or near) position $h(k)$ in the array. To search for a key, we compute the same function and look at (or near) the indicated position.

The challenge, of course, is that distinct keys may hash to the same index. This event is called a **collision**, and the design of hash tables revolves around two intertwined concerns: (1) choosing hash functions that minimize collisions, and (2) designing collision resolution strategies that handle the inevitable collisions efficiently.

1.2 What Makes a Good Hash Function

DEFINITION: Hash Function. A hash function is a function $h : K \rightarrow \{0, 1, \dots, M-1\}$ that maps keys from a universe K to indices in a table of size M . A hash function is said to be **ideal** if it maps each key to a uniformly random index, independently of all other keys.

An ideal hash function would distribute keys perfectly uniformly across the table, with each slot equally likely to receive any given key. In practice, no deterministic hash function can achieve this for all possible inputs (since the input space is typically much larger than the table size), but we can design functions that behave well for the inputs we expect to encounter.

A good hash function must satisfy three properties:

1. **Deterministic:** The same key must always produce the same hash value. Without this property, we could never retrieve a stored key.
2. **Efficient:** The hash value must be computable in constant time (or at least time proportional to the key length for variable-length keys such as strings).
3. **Uniform:** The hash values should be distributed as uniformly as possible across $\{0, 1, \dots, M-1\}$. Ideally, each index should be equally likely for a "random" key, and there should be no exploitable patterns.

A fourth desirable property, relevant to certain applications, is the **avalanche effect**: a small change in the key should produce a large, unpredictable change in the hash value. This property is essential for cryptographic hash functions but is also beneficial for general-purpose hashing, as it helps avoid clustering.

1.3 Modular Hashing for Integers

The simplest and most widely used hash function for integer keys is modular hashing:

```
function hashInteger(key, M):
    return key mod M
```

If `key` can be negative, we must take care to produce a non-negative result:

```
function hashInteger(key, M):
    return ((key mod M) + M) mod M
```

The choice of `M` is critical. If `M` is a power of 2, say $M = 2^k$, then `key mod M` extracts only the lowest `k` bits of the key. If the keys have any pattern in their low-order bits (as is common with memory addresses, which are often multiples of 4 or 8), this leads to severe clustering. For this reason, Sedgewick and many other authors recommend choosing `M` to be a **prime number** not close to a power of 2.

PROPOSITION A. If `M` is prime and the keys are uniformly distributed integers, then modular hashing distributes keys uniformly across the `M` buckets.

Proof. If keys are drawn uniformly from a range much larger than `M`, and `M` is prime, then for any residue `r` in `{0, 1, ..., M-1}`, the fraction of integers in the range that are congruent to `r` modulo `M` is approximately $1/M$. Since the keys are uniformly distributed, each bucket receives approximately the same number of keys. The primality of `M` ensures that no common factor between the key distribution and `M` can create systematic bias, unlike the case when `M` shares factors with regularities in the key set. QED.

In practice, good choices for `M` include primes such as 97, 997, 9973, 99991, or primes near powers of 2 such as 127, 8191, 131071, and 524287 (these are Mersenne primes of the form $2^p - 1$).

1.4 Polynomial Rolling Hash for Strings

For string keys, we need a hash function that incorporates all characters and is sensitive to their positions. The standard approach is the **polynomial rolling hash**, which treats the string as the coefficients of a polynomial and evaluates it at a fixed base:

DEFINITION: Polynomial Hash. For a string `s` of length `n` with characters `s[0], s[1], ..., s[n-1]`, the polynomial hash with base `B` and modulus `M` is:

$$h(s) = (s[0] * B^{(n-1)} + s[1] * B^{(n-2)} + \dots + s[n-2] * B + s[n-1]) \bmod M$$

The base B is typically chosen to be a small prime. Sedgewick uses $B = 31$, which is the same value used in Java's `String.hashCode()`. Other common choices include 37, 41, and 53.

1.5 Horner's Method

Evaluating the polynomial hash naively would require computing powers of B , which is both slow and numerically unstable. Instead, we use **Horner's method** (also known as Horner's rule or Horner's scheme), which rewrites the polynomial in nested form:

$$h(s) = (\dots((s[0] * B + s[1]) * B + s[2]) * B + \dots + s[n-2]) * B + s[n-1]) \bmod M$$

This allows us to compute the hash incrementally, processing one character at a time, using only multiplications and additions -- no exponentiation required.

```
function polynomialHash(s, B, M):  
    h = 0  
    for i = 0 to length(s) - 1:  
        h = (h * B + characterCode(s[i])) mod M  
    return h
```

Trace Example. Consider hashing the string "HASH" with base $B = 31$ and table size $M = 97$.

Character codes (using ASCII): H=72, A=65, S=83, H=72.

```
Step 0:  h = 0  
Step 1:  h = (0 * 31 + 72) mod 97 = 72 mod 97 = 72  
Step 2:  h = (72 * 31 + 65) mod 97 = (2232 + 65) mod 97 = 2297 mod 97 = 67  
Step 3:  h = (67 * 31 + 83) mod 97 = (2077 + 83) mod 97 = 2160 mod 97 = 26  
Step 4:  h = (26 * 31 + 72) mod 97 = (806 + 72) mod 97 = 878 mod 97 = 8
```

So $h(\text{"HASH"}) = 8$ with $M = 97$.

This computation takes $O(n)$ time for a string of length n , which is optimal since we must examine every character at least once.

The polynomial hash has the attractive property that it is **position-sensitive**: rearranging the characters of a string changes the hash value. This is in contrast to naive approaches such as summing character codes, where anagrams would always collide (e.g., "stop" and "pots" would hash to the same value).

1.6 The Birthday Paradox and Collision Probability

A fundamental result in probability theory governs the likelihood of collisions in hash tables. The **birthday paradox** (or birthday problem) tells us that collisions occur far sooner than intuition might suggest.

PROPOSITION B (Birthday Paradox). If we insert N keys into a table of size M using a truly random hash function, the expected number of keys we must insert before the first collision occurs is approximately $\sqrt{\pi * M / 2}$.

Proof sketch. The probability that the first k keys all hash to distinct indices is:

$$\begin{aligned} P(\text{no collision among } k \text{ keys}) &= (M/M) * ((M-1)/M) * ((M-2)/M) * \dots * ((M-k+1)/M) \\ &= \text{Product}_{i=0}^{k-1} (1 - i/M) \end{aligned}$$

Using the approximation $1 - x \sim e^{-x}$ for small x :

$$\begin{aligned} P(\text{no collision}) &\sim \exp(-(0 + 1 + 2 + \dots + (k-1)) / M) \\ &= \exp(-k(k-1) / (2M)) \end{aligned}$$

Setting this equal to $1/2$ (the median point) and solving for k :

$$\begin{aligned} \exp(-k^2 / (2M)) &\sim 1/2 \\ k^2 / (2M) &\sim \ln(2) \\ k &\sim \sqrt{2M * \ln(2)} \sim \sqrt{\pi * M / 2} \sim 1.177 * \sqrt{M} \end{aligned}$$

QED.

Practical implication. In a hash table with $M = 1,000,000$ slots, we expect the first collision after inserting only about $\sqrt{\pi * 500000} \sim 1,253$ keys. This means that even with a million slots, collisions begin occurring after only about a thousand insertions. Collision resolution is therefore not an edge case -- it is the normal mode of operation for any hash table with a reasonable number of entries.

This result also explains the famous birthday paradox from probability: in a group of only 23 people, there is a greater than 50% chance that two share a birthday (here $M = 365$, and $\sqrt{\pi * 365 / 2} \sim 23.9$).

The birthday paradox underscores a critical point: **collisions are inevitable** in any practical hash table, and the quality of a hash table implementation depends primarily on how well it handles collisions.

2. Separate Chaining

Estimated time: 35 minutes

2.1 Concept and Structure

Separate chaining (also called **chaining** or **closed addressing**) is the most straightforward collision resolution strategy. The idea is simple: maintain an array of M linked lists (called **chains** or **buckets**). When a key hashes to index i , we store it in the linked list at position i . Multiple keys that hash to the same index coexist peacefully in the same chain.

DEFINITION: Separate Chaining Hash Table. A separate chaining hash table consists of an array `table[0..M-1]` of linked lists, together with a hash function $h : K \rightarrow \{0, 1, \dots, M-1\}$. A key-value pair (k, v) is stored in the linked list `table[h(k)]`.

The beauty of separate chaining is its simplicity and robustness. Each bucket is essentially an independent unordered symbol table (implemented as a linked list), and the hash function simply directs us to the correct bucket. The performance of the overall structure depends on how evenly the hash function distributes keys among the M buckets.

2.2 Implementation

```

class ChainingHashTable:
    M          // number of buckets (table size)
    N          // number of key-value pairs
    table[M]   // array of linked lists

    constructor(initialCapacity):
        M = initialCapacity
        N = 0
        for i = 0 to M - 1:
            table[i] = new LinkedList()

    function hash(key):
        return (polynomialHash(key) mod M + M) mod M

    function loadFactor():
        return N / M

    function insert(key, value):
        // Check if key already exists -- update value if so
        i = hash(key)
        for each node in table[i]:
            if node.key == key:
                node.value = value
                return
        // Key not found -- prepend to chain
        table[i].prepend(key, value)
        N = N + 1
        // Resize if load factor exceeds threshold
        if N > 8 * M:
            resize(2 * M)

    function search(key):
        i = hash(key)
        for each node in table[i]:
            if node.key == key:
                return node.value
        return null    // key not found

    function remove(key):

```

```

i = hash(key)
for each node in table[i]:
    if node.key == key:
        table[i].delete(node)
        N = N - 1
        if M > 4 and N < 2 * M:
            resize(M / 2)
        return true
return false    // key not found

function resize(newM):
    newTable = new ChainingHashTable(newM)
    for i = 0 to M - 1:
        for each node in table[i]:
            newTable.insert(node.key, node.value)
    M = newTable.M
    table = newTable.table

```

2.3 Insert Operation

Insertion proceeds in three logical steps:

1. **Compute the hash.** Apply the hash function to the key to determine which bucket it belongs in.
2. **Search the chain.** Traverse the linked list at that bucket to check if the key already exists. If it does, update its value (maintaining the symbol table invariant that each key appears at most once).
3. **Add the key.** If the key is not already present, prepend a new node to the chain. Prepending takes $O(1)$ time and is preferred over appending for this reason.

After insertion, we check the load factor and resize if necessary. The resizing policy -- doubling when $\alpha > 8$ -- ensures that chains remain short on average, keeping operations efficient.

2.4 Search Operation

To search for a key, we compute its hash to identify the bucket, then perform a sequential search through the chain at that bucket. If the chain contains k nodes, the search takes $O(k)$ time in the worst case. Under the

uniform hashing assumption, the expected chain length is $\alpha = N/M$, so the expected search time is $O(\alpha)$.

2.5 Delete Operation

Deletion in separate chaining is straightforward because linked lists support efficient deletion. We locate the node in the appropriate chain and remove it. After deletion, we check whether the load factor has dropped below the lower threshold and halve the table if so.

Note the asymmetry in the resizing thresholds: we double when $\alpha > 8$ but halve when $\alpha < 2$. This hysteresis prevents thrashing -- the pathological situation where the table repeatedly resizes up and down due to a sequence of alternating insertions and deletions near the threshold.

2.6 Load Factor and Expected Chain Length

DEFINITION: Load Factor. The load factor of a hash table is $\alpha = N/M$, where N is the number of stored key-value pairs and M is the table size. For separate chaining, α represents the average number of keys per chain.

The load factor is the single most important parameter governing hash table performance. Unlike open addressing methods (discussed in the next section), separate chaining can tolerate load factors greater than 1 -- chains can grow to arbitrary length. However, performance degrades linearly with α .

PROPOSITION C. Under the uniform hashing assumption, in a separate chaining hash table with load factor $\alpha = N/M$:

- The expected number of keys in any given chain is α .
- The expected cost of a **search miss** is α (we must traverse the entire chain).
- The expected cost of a **search hit** is $1 + \alpha/2$ (on average, we find the key halfway through the chain, plus the cost of examining the first node).

Proof. Under the uniform hashing assumption, each key is equally likely to hash to any of the M buckets, independently of all other keys. The number of keys in a given bucket follows a binomial distribution $\text{Bin}(N, 1/M)$, which has expected value $N/M = \alpha$.

For a search miss, we must traverse the entire chain at the hashed bucket, examining all keys. The expected length is α , so the expected cost is α .

For a search hit, suppose the target key is the j -th key inserted into its bucket (where the first key inserted is $j = 1$). At the time of insertion, there were $j - 1$ keys already in the bucket, and these keys remain in the chain at the time of search. Since we prepend new keys, the target key is behind the $j - 1$ keys inserted after it. Under uniform hashing, the expected value of j is $(N/M + 1) / 2$, giving an expected search cost of approximately $1 + \alpha/2$. QED.

2.7 Resizing Policy

The resizing policy for separate chaining involves two thresholds:

- **Double** the table size when $\alpha > 8$ (i.e., when $N > 8M$).
- **Halve** the table size when $\alpha < 2$ (i.e., when $N < 2M$), subject to a minimum table size.

These specific thresholds reflect a pragmatic trade-off. A load factor of 8 means that each chain contains 8 keys on average. Since each search requires traversing a chain, this means about 4 comparisons on average for a search hit and 8 for a miss. This is quite acceptable -- the constant factors are small because linked list traversal is simple, and the memory overhead per key is modest (one pointer per node in the chain, plus one pointer per bucket in the array).

Halving at $\alpha < 2$ ensures that we do not waste excessive memory when the table becomes sparse after many deletions.

The amortized cost of resizing is $O(1)$ per operation, by the same doubling argument used for dynamic arrays: each resize costs $O(N)$, but it is triggered only after $\Theta(N)$ insertions (or deletions), so the amortized cost per operation is $O(1)$.

2.8 Separate Chaining Trace Example

Let us trace the insertion of keys $\{S, E, A, R, C, H, X, M, P\}$ into a separate chaining hash table with $M = 5$, using the hash function $h(k) = \text{ord}(k) \bmod 5$ where $\text{ord}()$ returns the ASCII code.

Character ASCII codes: S=83, E=69, A=65, R=82, C=67, H=72, X=88, M=77, P=80

Hash values:

```
h(S) = 83 mod 5 = 3
h(E) = 69 mod 5 = 4
h(A) = 65 mod 5 = 0
h(R) = 82 mod 5 = 2
h(C) = 67 mod 5 = 2
h(H) = 72 mod 5 = 2
h(X) = 88 mod 5 = 3
h(M) = 77 mod 5 = 2
h(P) = 80 mod 5 = 0
```

After all insertions, the table looks like:

```
Bucket 0: P -> A -> null
Bucket 1: (empty)
Bucket 2: M -> H -> C -> R -> null
Bucket 3: X -> S -> null
Bucket 4: E -> null
```

Load factor: $\alpha = 9 / 5 = 1.8$

Chain lengths: $[2, 0, 4, 2, 1]$

Observe that bucket 2 has received 4 keys, which is more than twice the average chain length of 1.8. This is a natural consequence of the probabilistic nature of hashing -- even with a perfect hash function, some buckets will be more crowded than others, just as some days of the year have more birthdays than others.

2.9 Chain Length Distribution and Analysis

Under the uniform hashing assumption, the distribution of chain lengths follows a Poisson distribution with parameter α (for large M). The probability that a given chain has exactly k keys is approximately:

$$P(\text{chain length} = k) \sim (\alpha^k * e^{-\alpha}) / k!$$

The maximum chain length is of interest for worst-case analysis. It can be shown that, with high probability, the maximum chain length is $\Theta(\log N / \log \log N)$ when $\alpha = \Theta(1)$. This is much smaller than the worst case of N (all keys in one chain) that would occur with a pathological hash function.

The companion C++ code includes a `chainLengths()` method that returns a vector of chain lengths, as well as a `printTable()` method for visualizing the distribution of keys across buckets. These are invaluable tools for diagnosing hash function quality in practice.

3. Linear Probing

Estimated time: 35 minutes

3.1 Open Addressing Concept

Open addressing is an alternative collision resolution strategy in which all keys are stored directly in the table array, without any auxiliary linked lists. When a collision occurs -- that is, when a key hashes to an index that is already occupied -- we probe other locations in the array according to a deterministic sequence until we find an empty slot.

The simplest and most well-known open addressing scheme is **linear probing**, in which we simply check the next slot, then the next, and so on, wrapping around to the beginning of the array if we reach the end.

DEFINITION: Linear Probing. In linear probing, the probe sequence for a key k is:

$$h(k), h(k) + 1, h(k) + 2, \dots, (\text{mod } M)$$

That is, we start at the home position $h(k)$ and advance one slot at a time (modulo M) until we find either the key we are looking for (search hit) or an empty slot (search miss or available position for insertion).

Open addressing methods have the advantage of avoiding the memory overhead of linked list nodes (no pointers per entry), and they exhibit better cache performance because the probed entries are contiguous in memory. The disadvantage is that the load factor must remain strictly below 1 (there must always be empty slots in the table), and performance degrades sharply as the load factor approaches 1.

3.2 Insert with Wrapping

To insert a key-value pair using linear probing:

1. Compute the hash $i = h(\text{key})$.
2. If `table[i]` is empty (or a tombstone, if we use that deletion strategy), place the key-value pair at position `i`.
3. Otherwise, advance to $i = (i + 1) \bmod M$ and repeat.
4. Continue until an empty (or tombstone) slot is found.

The wrapping behavior (using modular arithmetic) ensures that the probe sequence visits every slot in the table if necessary. Since the load factor is always less than 1, there is always at least one empty slot, so the loop is guaranteed to terminate.

```

class LinearProbingHashTable:
    M          // table size
    N          // number of key-value pairs
    keys[M]    // key array (null indicates empty)
    values[M]  // value array
    deleted[M] // tombstone flags (for deletion strategy)

    constructor(initialCapacity):
        M = initialCapacity
        N = 0
        for i = 0 to M - 1:
            keys[i] = null
            deleted[i] = false

    function hash(key):
        return (polynomialHash(key) mod M + M) mod M

    function loadFactor():
        return N / M

    function insert(key, value):
        // Resize if load factor exceeds 1/2
        if N >= M / 2:
            resize(2 * M)

        i = hash(key)
        while keys[i] != null:
            if keys[i] == key:
                values[i] = value    // update existing
                return
            i = (i + 1) mod M

        keys[i] = key
        values[i] = value
        deleted[i] = false
        N = N + 1

```

3.3 Search with Wrapping

To search for a key, we follow the same probe sequence:

1. Compute $i = h(\text{key})$.
2. If $\text{table}[i]$ is empty and not a tombstone, the key is not in the table (search miss).
3. If $\text{table}[i]$ contains the key, we have a search hit.
4. Otherwise, advance to $i = (i + 1) \bmod M$ and repeat.

The critical invariant is that the probe sequence for any key forms an unbroken chain of occupied (or tombstoned) slots starting from the home position. If we encounter a truly empty slot (not a tombstone), we know the key cannot be further along in the sequence, because it would have been placed in the empty slot during insertion.

```
function search(key):
    i = hash(key)
    while keys[i] != null or deleted[i]:
        if keys[i] == null and deleted[i]:
            // Tombstone -- skip but continue probing
            i = (i + 1) mod M
            continue
        if keys[i] == key:
            return values[i]      // search hit
        i = (i + 1) mod M
    return null                  // search miss
```

3.4 Deletion: Tombstones

Deletion in linear probing is more delicate than in separate chaining. We cannot simply set the deleted slot to empty, because doing so would break the probe sequences of keys that were inserted after the deleted key and that probed past the deleted key's position.

Example. Suppose keys A, B, C hash to positions 3, 3, 3 respectively. After insertion, A is at position 3, B at position 4, and C at position 5. If we delete B by setting position 4 to empty, a subsequent search for C starting at position 3 would find A at position 3, then an empty slot at position 4, and would incorrectly conclude that C is not in the table -- even though C is at position 5.

The simplest solution is to use **tombstones** (also called **lazy deletion** or **soft deletion**). When we delete a key, we mark its slot with a special tombstone marker rather than leaving it truly empty. The tombstone tells the search algorithm to continue probing (the chain is not broken), but the insert algorithm can treat it as an available slot for new insertions.

```
function remove(key):
    i = hash(key)
    while keys[i] != null or deleted[i]:
        if keys[i] == key:
            keys[i] = null
            values[i] = null
            deleted[i] = true    // mark as tombstone
            N = N - 1
            if M > 4 and N > 0 and N <= M / 8:
                resize(M / 2)
            return true
        i = (i + 1) mod M
    return false                // key not found
```

The disadvantage of tombstones is that they accumulate over time, degrading performance. Many tombstones lengthen probe sequences because the search must continue past tombstones, even though they do not contain useful data. Periodic rehashing (rebuilding the entire table from scratch) is necessary to clear tombstones if the deletion rate is high.

3.5 Deletion: Knuth's Backward-Shift Algorithm

An alternative to tombstones, due to Donald Knuth (TAOCP Vol. 3, Section 6.4, Algorithm R), avoids the tombstone problem entirely by **reshuffling** keys after a deletion to maintain the invariant that every key's probe sequence is an unbroken chain of occupied slots.

The algorithm works as follows: after deleting a key at position **i**, we scan forward through the probe sequence. For each subsequent occupied slot **j**, we check whether the key at **j** "belongs" at or before position **i** -- that is, whether position **i** lies on the probe path from **h(keys[j])** to **j**. If so, we move the key at **j** to position **i**, set **i = j**, and continue scanning. When we reach an empty slot, we are done.

```

function removeKnuth(key):
    i = hash(key)
    // Find the key
    while keys[i] != null:
        if keys[i] == key:
            break
        i = (i + 1) mod M

    if keys[i] == null:
        return false                // key not found

    // Backward-shift: fill the gap
    keys[i] = null
    values[i] = null
    N = N - 1

    j = (i + 1) mod M
    while keys[j] != null:
        k = hash(keys[j])
        // Check if key at j should be moved to fill gap at i
        // This condition checks whether position i is "between"
        // the home position k and current position j (cyclically)
        if (i <= j and (k <= i or k > j)) or
           (i > j and (k <= i and k > j)):
            keys[i] = keys[j]
            values[i] = values[j]
            keys[j] = null
            values[j] = null
            i = j
        j = (j + 1) mod M

    if M > 4 and N > 0 and N <= M / 8:
        resize(M / 2)
    return true

```

The backward-shift algorithm is more complex to implement correctly (the cyclic boundary condition in the `if` statement is notoriously tricky), but it avoids tombstone accumulation and keeps the table "clean." The companion C++ code provides both deletion strategies, allowing direct comparison of their behavior.

3.6 Clustering Phenomenon

DEFINITION: Cluster. A cluster in a linear probing hash table is a maximal contiguous block of occupied slots. The size of a cluster is the number of occupied slots it contains.

The most significant weakness of linear probing is the **clustering phenomenon**: occupied slots tend to form long contiguous runs (clusters), and longer clusters grow faster than shorter ones. This positive feedback loop, where success breeds more success, leads to clusters that are significantly longer than what a uniform distribution would produce.

The intuition is as follows. Consider a cluster of length L . Any new key whose home position falls within the cluster or at the slot immediately following the cluster will be appended to the cluster, extending its length to $L + 1$. The probability of this happening is $(L + 1) / M$, which is proportional to the cluster's length. Thus, long clusters attract new keys at a rate proportional to their length, creating a "rich get richer" dynamic.

This phenomenon is called **primary clustering**. It is specific to linear probing and does not affect quadratic probing or double hashing (though those methods have their own clustering issues).

Despite primary clustering, linear probing performs well when the load factor is kept low (below 1/2 or at most 2/3). The expected number of probes remains constant for any fixed load factor less than 1, though the constant grows rapidly as the load factor approaches 1.

3.7 Linear Probing Trace Example

Let us trace the insertion of keys $\{S, E, A, R, C, H\}$ into a linear probing table with $M = 11$, using $h(k) = \text{ord}(k) \bmod 11$.

Hash values:

```
h(S) = 83 mod 11 = 6
h(E) = 69 mod 11 = 3
h(A) = 65 mod 11 = 10
h(R) = 82 mod 11 = 5
h(C) = 67 mod 11 = 1
h(H) = 72 mod 11 = 6 (collision with S!)
```

Insertion trace:

Insert S: $h(S) = 6$. Slot 6 empty. Place S at 6.

Table: [_, _, _, _, _, _, S, _, _, _, _]
0 1 2 3 4 5 6 7 8 9 10

Insert E: $h(E) = 3$. Slot 3 empty. Place E at 3.

Table: [_, _, _, E, _, _, S, _, _, _, _]

Insert A: $h(A) = 10$. Slot 10 empty. Place A at 10.

Table: [_, _, _, E, _, _, S, _, _, _, A]

Insert R: $h(R) = 5$. Slot 5 empty. Place R at 5.

Table: [_, _, _, E, _, R, S, _, _, _, A]

Insert C: $h(C) = 1$. Slot 1 empty. Place C at 1.

Table: [_, C, _, E, _, R, S, _, _, _, A]

Insert H: $h(H) = 6$. Slot 6 occupied (S).

Probe slot 7: empty. Place H at 7.

Table: [_, C, _, E, _, R, S, H, _, _, A]
0 1 2 3 4 5 6 7 8 9 10

Load factor: $\alpha = 6/11 \sim 0.545$

Clusters: {C} at 1 (length 1), {E} at 3 (length 1), {R, S, H} at 5-7 (length 3), {A} at 10 (length 1).

Notice that the cluster at positions 5-7 formed because R (home position 5) and S (home position 6) were placed adjacent, and then H (home position 6) was displaced to position 7 by the collision with S. Any new key hashing to positions 5, 6, 7, or 8 would extend this cluster.

3.8 Cluster Analysis

The companion C++ code includes a `countClusters()` method that scans the table and returns a vector of cluster sizes. This is useful for empirically studying the clustering phenomenon. A healthy linear probing table (with $\alpha < 1/2$) should have many small clusters and few large ones. As α approaches 1, clusters coalesce and performance degrades dramatically.

To quantify cluster behavior, we note that the expected number of clusters of size k in a table with load factor α can be derived from the theory of runs in Bernoulli trials, though the exact analysis is

complicated by the dependencies introduced by linear probing. Knuth's analysis (discussed in the next section) provides the definitive results.

3.9 Comparison of Deletion Strategies

The two deletion strategies -- tombstones and Knuth's backward-shift -- represent different engineering trade-offs:

Tombstones:

- Simple to implement
- Delete operation is $O(1)$ after finding the key (just set a flag)
- Degrades search performance over time as tombstones accumulate
- Requires periodic full rehash to clear tombstones
- Works well when deletions are infrequent relative to insertions

Knuth's Backward-Shift:

- More complex to implement (tricky cyclic boundary conditions)
- Delete operation may move several keys ($O(1/\alpha)$ expected moves)
- Maintains a clean table with no wasted slots
- No periodic rehash needed
- Preferred when deletions are frequent or when search performance is critical

The companion C++ code implements both strategies, using the tombstone approach in `remove()` and making the backward-shift approach available for comparison. Students should experiment with both to understand the performance trade-offs empirically.

4. Analysis of Hashing

Estimated time: 25 minutes

4.1 The Uniform Hashing Assumption

All theoretical analyses of hash table performance rest on the **uniform hashing assumption**:

DEFINITION: Uniform Hashing Assumption. Each key is equally likely to hash to any of the M table positions, independently of where other keys hash.

This assumption is an idealization. No deterministic hash function can truly satisfy it, since a deterministic function always maps the same key to the same position. The assumption is a model for the behavior of a "good" hash function applied to keys that do not have adversarial structure. It is analogous to the assumption that input is "randomly ordered" in the analysis of quicksort.

Under this assumption, we can derive precise expected-cost formulas for both separate chaining and linear probing.

4.2 Expected Cost for Separate Chaining

PROPOSITION D. Under the uniform hashing assumption, in a separate chaining hash table with N keys and M buckets (load factor $\alpha = N/M$):

Operation	Expected comparisons
Search hit	$1 + \alpha/2$
Search miss	α
Insert (new key)	$1 + \alpha$

Proof. For a search miss, we hash to a random bucket and traverse its entire chain. The expected chain length is α , so the expected cost is α comparisons (or $1 + \alpha$ if we count the hash computation).

For a search hit, we want the expected position of a random key in its chain. Consider the i -th key inserted (for $i = 1, 2, \dots, N$). At the time it was inserted, it was prepended to its chain, and subsequently $N - i$ additional keys were inserted, each with probability $1/M$ of landing in the same chain. Thus, the expected number of keys ahead of the i -th key in its chain is $(N - i) / M$. The expected cost of finding the i -th key is $1 + (N - i) / M$. Averaging over all N keys:

$$\begin{aligned}
\text{Expected search hit cost} &= (1/N) * \sum_{i=1}^N (1 + (N - i) / M) \\
&= 1 + (1/(NM)) * \sum_{i=1}^N (N - i) \\
&= 1 + (1/(NM)) * N(N-1)/2 \\
&= 1 + (N - 1) / (2M) \\
&\sim 1 + \alpha/2
\end{aligned}$$

For insertion of a new key, we must first verify that the key is not already present (a search miss costing α), then prepend the new node (costing 1). Total: $1 + \alpha$. QED.

4.3 Expected Cost for Linear Probing

The analysis of linear probing is considerably more difficult, requiring a beautiful and intricate argument due to Donald Knuth (1962). The result is one of the highlights of the analysis of algorithms.

PROPOSITION E (Knuth, 1962). Under the uniform hashing assumption, in a linear probing hash table with load factor $\alpha = N/M$ (where $\alpha < 1$):

Operation	Expected probes
Search hit	$\sim 1/2 * (1 + 1/(1 - \alpha))$
Search miss	$\sim 1/2 * (1 + 1/(1 - \alpha)^2)$

Proof sketch. The full proof is one of the most celebrated results in the analysis of algorithms. We outline the key ideas.

For a **search miss**, we must find the expected length of a probe sequence ending at an empty slot. Consider the probability that a probe sequence starting at a random position has length at least k . This requires that positions $h, h+1, \dots, h+k-1$ are all occupied, which depends on the distribution of cluster lengths. Knuth showed that the expected number of probes is:

$$E[\text{probes for miss}] = 1/2 * (1 + 1/(1 - \alpha)^2)$$

The derivation involves a recurrence relation for the probability that a random slot begins a cluster of length at least k , solved using generating functions.

For a **search hit**, we average the insertion costs over the history of the table. The i -th key was inserted when the load factor was $(i-1)/M$, so the expected cost of inserting it was approximately $1/2 * (1 + 1/(1 - (i-1)/M)^2)$ (a search miss at the time of insertion). Averaging:

$$E[\text{probes for hit}] = (1/N) * \sum_{i=1}^N 1/2 * (1 + M^2/(M - i + 1)^2)$$

This sum can be approximated by an integral, yielding:

$$E[\text{probes for hit}] \sim 1/2 * (1 + 1/(1 - \alpha))$$

QED.

To appreciate these formulas, let us compute the expected number of probes for several load factors:

Load factor (α)	Search hit	Search miss
1/4 (25%)	1.17	1.39
1/3 (33%)	1.25	1.63
1/2 (50%)	1.50	2.50
2/3 (67%)	2.00	5.00
3/4 (75%)	2.50	8.50
9/10 (90%)	5.50	50.50
99/100 (99%)	50.50	5000.50

The table reveals a striking pattern: performance is excellent for $\alpha \leq 1/2$, acceptable up to about $\alpha = 2/3$, and degrades rapidly beyond that. At $\alpha = 9/10$, a search miss requires an expected 50 probes -- a far cry from the constant-time ideal. At $\alpha = 99/100$, the table is nearly useless.

This is why the standard resizing policy for linear probing doubles the table when α exceeds $1/2$, ensuring that the load factor stays in the range where performance is near-optimal.

4.4 Knuth's Parking Problem

Knuth offered a beautiful intuitive analogy for the clustering phenomenon in linear probing: the **parking problem**.

Imagine a one-way street with M parking spaces numbered $0, 1, \dots, M-1$. A sequence of N cars arrives, and each car has a preferred space (its "hash"). Each car drives to its preferred space and, if it is occupied, continues forward (one space at a time, wrapping around) until it finds an empty space. The question is: how far does the average car have to drive past its preferred space?

This is precisely the linear probing problem in disguise. The clustering phenomenon corresponds to the observation that a car whose preferred space is in the middle of a long run of occupied spaces must drive a long distance. Moreover, when it parks at the end of the run, it extends the run, making it even harder for future cars.

Knuth's analysis shows that the expected displacement of a random car is $\frac{1}{2} * (1 + \frac{1}{(1 - \alpha)^2}) - 1$, which matches the expected number of probes for a search miss minus 1 (since the first probe is at the preferred space).

The parking problem analogy is valuable not just for intuition but also as a proof technique. Knuth's original analysis proceeds by computing the expected number of cars that park in a given range of spaces, using a clever combinatorial argument.

4.5 Comparison of Expected Costs

To summarize the theoretical performance of both methods:

Separate Chaining (load factor α):

Operation	Expected cost
Search hit	$1 + \alpha/2$
Search miss	α
Insert	$1 + \alpha$

Linear Probing (load factor $\alpha < 1$):

Operation	Expected cost
Search hit	$\sim 1/2 * (1 + 1/(1 - \alpha))$
Search miss	$\sim 1/2 * (1 + 1/(1 - \alpha)^2)$
Insert	$\sim 1/2 * (1 + 1/(1 - \alpha)^2)$

When `alpha` is small, both methods require close to 1 probe per operation, and linear probing is often faster in practice due to better cache behavior. As `alpha` grows, separate chaining degrades linearly while linear probing degrades much more steeply (quadratically for search misses).

The companion C++ code includes a collision analysis function that empirically measures the number of probes at different load factors, allowing students to verify these theoretical predictions experimentally.

4.6 Worst-Case Behavior

While the expected-case analysis paints a favorable picture, it is important to understand the worst case. Under adversarial conditions (a malicious choice of keys, or a degenerate hash function), both methods degenerate to $O(N)$ per operation:

- **Separate chaining worst case:** All N keys hash to the same bucket, creating a single chain of length N . Search requires N comparisons.
- **Linear probing worst case:** All N keys form a single cluster of length N . Search requires up to N probes.

The worst case occurs with probability that is exponentially small under the uniform hashing assumption, but it can be triggered deterministically by an adversary who knows the hash function (see Section 5.2 on algorithmic complexity attacks).

Red-black BSTs, by contrast, guarantee $O(\log N)$ worst-case performance regardless of the input. This is a significant advantage in applications where worst-case guarantees are essential (e.g., real-time systems, security-critical systems).

5. Practical Considerations

Estimated time: 20 minutes

5.1 The hashCode() Contract

In Java (and similar object-oriented languages), the `hashCode()` method defines the hash function contract for objects used as keys in hash tables. The contract specifies:

1. **Consistency:** If two objects are equal according to `equals()`, they must have the same `hashCode()`. (The converse need not hold -- unequal objects may have the same hash code.)
2. **Determinism:** An object's `hashCode()` must not change as long as no fields used in `equals()` are modified.
3. **Uniformity** (desirable but not required): The hash code should be distributed as uniformly as possible across the int range.

The first rule is critical: violating it causes hash tables to lose keys silently. If `a.equals(b)` but `a.hashCode() != b.hashCode()`, then inserting with key `a` and searching with key `b` will look in different buckets, and the search will fail.

The second rule implies that **mutable objects make poor hash table keys**. If an object is mutated after being inserted into a hash table, its hash code may change, causing it to be "lost" in the wrong bucket. For this reason, hash table keys should be immutable whenever possible (strings, integers, tuples of immutable values, etc.).

A typical pattern for implementing `hashCode()` for a class with multiple fields combines the hash codes of individual fields using a polynomial-like formula:

```
function hashCode():
    h = 17 // non-zero initial value
    h = 31 * h + field1.hashCode()
    h = 31 * h + field2.hashCode()
    h = 31 * h + field3.hashCode()
    ...
    return h
```

The multiplier 31 is the same base used in Java's `String.hashCode()`. It is a small prime that produces good distribution and has the computational advantage that `31 * h` can be computed as `(h << 5) - h` (a shift and a subtraction, avoiding a full multiplication on older hardware).

5.2 Algorithmic Complexity Attacks

A significant practical concern with hash tables is the possibility of **algorithmic complexity attacks** (also called **hash collision attacks** or **HashDoS**). An attacker who knows the hash function can deliberately craft a set of keys that all hash to the same bucket, reducing the hash table to a linked list with $O(N)$ operations.

This attack was first described by Crosby and Wallach (2003) and was dramatically demonstrated in 2011 when researchers showed that many web frameworks (including those for Java, PHP, Python, Ruby, and ASP.NET) were vulnerable to denial-of-service attacks through specially crafted HTTP POST parameters. By sending a few megabytes of data containing carefully chosen parameter names that all hash to the same bucket, an attacker could consume hours of CPU time on the server.

The fundamental vulnerability is that most programming languages use deterministic hash functions that are publicly known. An attacker can precompute collision sets offline and then deploy them against any server using that language.

Defenses include:

1. **Randomized hashing** (hash seed randomization): Choose the hash function randomly at program startup, so the attacker cannot predict which keys will collide.
2. **Universal hashing**: Use a hash function family with provable collision bounds (see Section 5.3).
3. **Limiting chain length**: Switch from a linked list to a balanced BST when a chain exceeds a threshold (Java 8's `HashMap` uses this approach, converting chains to red-black trees when they exceed length 8).
4. **Rate limiting**: Limit the number of parameters accepted in an HTTP request.

5.3 Universal Hashing

Universal hashing, introduced by Carter and Wegman (1979), provides a theoretical framework for defending against algorithmic complexity attacks.

DEFINITION: Universal Hash Family. A family H of hash functions from K to $\{0, 1, \dots, M-1\}$ is **universal** if for any two distinct keys $k_1 \neq k_2$:

$$P_{\{h \in H\}}[h(k_1) = h(k_2)] \leq 1/M$$

That is, the probability of collision between any two fixed keys, when the hash function is chosen uniformly at random from the family, is at most $1/M$.

The key insight is that by choosing the hash function randomly at runtime (and keeping the choice secret from the adversary), we prevent the adversary from constructing collision sets. The adversary cannot predict which keys will collide because they do not know which hash function was selected.

A simple universal hash family for integer keys is:

$$h_{\{a,b\}}(k) = ((a * k + b) \bmod p) \bmod M$$

where p is a prime larger than the key universe, and a and b are chosen uniformly at random from $\{0, 1, \dots, p-1\}$ with $a \neq 0$.

PROPOSITION F. The family $\{h_{\{a,b\}} : 1 \leq a < p, 0 \leq b < p\}$ is universal.

Proof. Fix two distinct keys $k_1 \neq k_2$. The values $r_1 = (a*k_1 + b) \bmod p$ and $r_2 = (a*k_2 + b) \bmod p$ are distinct modulo p (since $a \neq 0$ and $k_1 \neq k_2$, and p is prime). As (a, b) ranges over all possible values, the pair (r_1, r_2) takes on each of the $p(p-1)$ ordered pairs of distinct elements of $\{0, \dots, p-1\}$ exactly once. The collision condition $r_1 \bmod M = r_2 \bmod M$ is satisfied by at most $p(p-1)/M$ of these pairs (with equality when M divides $p-1$). Dividing by the total number of hash functions $p(p-1)$ gives a collision probability of at most $1/M$. QED.

In practice, many modern hash table implementations use a form of randomized hashing. Python, for example, randomizes the hash seed at startup (since Python 3.3), making it infeasible for an attacker to predict hash values.

5.4 Cuckoo Hashing

Cuckoo hashing, introduced by Pagh and Rodler (2001), is an elegant open addressing scheme that guarantees worst-case $O(1)$ lookups (not just expected $O(1)$).

The idea is to use two hash functions, h_1 and h_2 , and two tables, T_1 and T_2 . Each key is stored in either $T_1[h_1(k)]$ or $T_2[h_2(k)]$ -- never anywhere else. To search for a key, we check these two locations and nothing more. This gives worst-case $O(1)$ search time.

Insertion is where the "cuckoo" metaphor applies (named after the cuckoo bird, which lays its eggs in other birds' nests). To insert key k :

1. Place k at $T_1[h_1(k)]$.
2. If that slot was empty, we are done.
3. If that slot was occupied by some key k' , we displace k' (like a cuckoo chick pushing out the original egg).
4. Now we must reinsert k' . Place it at its alternate location in $T_2[h_2(k')]$.
5. If that slot was occupied by some key k'' , displace k'' and continue.
6. Repeat until we find an empty slot or detect a cycle.

If a cycle is detected (indicating that insertion is stuck in an infinite loop), we rehash the entire table with new hash functions. This happens with low probability when the load factor is below $1/2$.

The expected amortized insertion time is $O(1)$, and the worst-case search time is $O(1)$ (exactly 2 lookups). Cuckoo hashing is particularly attractive in applications where search performance is critical and the key set is relatively static.

5.5 Robin Hood Hashing

Robin Hood hashing, introduced by Celis, Larson, and Munro (1985), is a variant of linear probing that reduces the variance of probe sequence lengths by enforcing a "fairness" invariant.

The idea is simple but powerful: during insertion, if the key being inserted has traveled fewer positions from its home than the key currently occupying the probed position, the two keys swap. This "Robin Hood" policy (steal from the rich, give to the poor) ensures that no key is much farther from its home position than any other.

More precisely, each occupied slot stores a **probe distance** (also called the **displacement** or **DIB** -- distance from initial bucket). During linear probing insertion, when we encounter an occupied slot whose probe distance is less than the probe distance of the key we are inserting, we swap the two keys and continue inserting the displaced key.

```

function insertRobinHood(key, value):
    i = hash(key)
    dist = 0
    while keys[i] != null:
        existingDist = probeDistance(i, hash(keys[i]))
        if dist > existingDist:
            // Swap: the incoming key takes this spot
            swap(key, keys[i])
            swap(value, values[i])
            dist = existingDist
        i = (i + 1) mod M
        dist = dist + 1
    keys[i] = key
    values[i] = value

```

The effect is dramatic: while standard linear probing has high variance in probe lengths (some keys are very close to home, others very far), Robin Hood hashing compresses the distribution, making almost all probe lengths close to the expected value. The expected maximum probe length is $O(\log \log N)$ rather than $O(\log N)$ as in standard linear probing.

Robin Hood hashing maintains the same expected cost as standard linear probing but with much better worst-case behavior. It has seen increasing adoption in high-performance hash table implementations, including Rust's standard library `HashMap` (prior to the switch to `SwissTable`/`hashbrown`).

An additional benefit of Robin Hood hashing is that it enables **backward-shift deletion** more naturally: since keys are ordered by probe distance within each cluster, finding the right key to shift backward is more straightforward.

6. Symbol Table Implementation Comparison

Estimated time: 15 minutes

6.1 Comprehensive Comparison Table

Having now studied six symbol table implementations, we can assemble a comprehensive comparison. The following table summarizes the key characteristics of each implementation:

Implementation	Search (worst)	Insert (worst)	Search (avg)	Insert (avg)	Ordered ops?	Key requirement
Sequential search (unordered list)	N	N	N/2	N	No	<code>equals()</code>
Binary search (ordered array)	$\log N$	N	$\log N$	N/2	Yes	<code>compareTo()</code>
BST (unbalanced)	N	N	$1.39 \log N$	$1.39 \log N$	Yes	<code>compareTo()</code>
Red-black BST	$2 \log N$	$2 \log N$	$1.00 \log N$	$1.00 \log N$	Yes	<code>compareTo()</code>
Separate chaining	N	N	3-5*	3-5*	No	<code>equals()</code> , <code>hashCode()</code>
Linear probing	N	N	3-5*	3-5*	No	<code>equals()</code> , <code>hashCode()</code>

(*) Assuming uniform hashing and load factor α in the range $[1/8, 1/2]$. The "3-5" refers to the expected number of probes/comparisons with typical load factors.

6.2 Detailed Analysis by Dimension

Search performance. Hash tables offer expected $O(1)$ search time, which is superior to the $O(\log N)$ of balanced BSTs for large N . However, the worst case for hash tables is $O(N)$ (all keys in one bucket or one cluster), compared to $O(\log N)$ for red-black BSTs. Applications that require guaranteed worst-case performance should prefer balanced BSTs.

Insert performance. The same asymptotic comparison applies to insertion. Hash tables are expected $O(1)$ but worst-case $O(N)$, while red-black BSTs are worst-case $O(\log N)$. Both support amortized $O(1)$ insertion when resizing is included.

Ordered operations. This is the decisive distinguishing factor. Hash tables do not maintain any ordering among keys, so they cannot efficiently support operations such as:

- Finding the minimum or maximum key
- Finding the k-th smallest key (rank/select)
- Range queries (finding all keys in a given interval)
- Iterating over keys in sorted order
- Finding the predecessor or successor of a given key

All of these operations are $O(\log N)$ or $O(\log N + R)$ (where R is the number of results) in a balanced BST, but require $O(N \log N)$ time with a hash table (by extracting and sorting all keys). If any ordered operations are needed, balanced BSTs are the clear choice.

Key requirements. Hash tables require keys to support equality testing and hashing, but do not require a total ordering. Balanced BSTs require keys to support comparison (a total ordering). Some types of keys (e.g., complex objects, multimedia data) may support equality and hashing but not a natural total ordering.

Memory. Separate chaining uses extra memory for linked list pointers (typically one pointer per key-value pair plus one pointer per bucket). Linear probing stores keys directly in the array, with no pointer overhead, but requires keeping the load factor below $1/2$, which means at least half the array is empty. Red-black BSTs use three pointers per node (left, right, parent or color bit). In practice, linear probing often uses less total memory than the other methods.

Cache performance. Linear probing has excellent cache behavior because probes access contiguous memory locations. Separate chaining has poor cache behavior because following linked list pointers causes cache misses. Red-black BSTs fall somewhere in between, depending on the allocation pattern. For modern hardware where cache misses dominate performance, linear probing often outperforms its theoretical predictions relative to other methods.

6.3 When to Use Each Implementation

Based on the comparison, we can offer the following guidelines:

- **Small tables ($N < 20$):** Use sequential search. The simplicity is unbeatable, and for such small N , the linear scan is fast enough.
- **Static tables (no insertions after construction):** Use binary search in a sorted array. It supports $O(\log N)$ search and all ordered operations, with minimal memory overhead.

- **General-purpose with ordered operations:** Use a red-black BST. It provides $O(\log N)$ worst-case guarantees for all operations and supports the full ordered symbol table API.
- **General-purpose without ordered operations:** Use a hash table (either separate chaining or linear probing). The expected $O(1)$ search and insert times are hard to beat for applications that do not need ordering. Prefer linear probing when memory efficiency and cache performance matter; prefer separate chaining when the load factor may be high or when simplicity of deletion is important.
- **Real-time or adversarial settings:** Use a red-black BST for guaranteed worst-case bounds, or use a hash table with universal hashing for expected $O(1)$ performance that is robust against adversarial inputs.

7. Practice Problems

Problem 1: Hash Function Computation

Problem. Compute the polynomial hash of the string "TABLE" with base $B = 31$ and table size $M = 13$.

Solution.

Character codes (ASCII): T=84, A=65, B=66, L=76, E=69.

We apply Horner's method:

```
Step 0: h = 0
Step 1: h = (0 * 31 + 84) mod 13 = 84 mod 13 = 6
Step 2: h = (6 * 31 + 65) mod 13 = (186 + 65) mod 13 = 251 mod 13 = 4
Step 3: h = (4 * 31 + 66) mod 13 = (124 + 66) mod 13 = 190 mod 13 = 8
Step 4: h = (8 * 31 + 76) mod 13 = (248 + 76) mod 13 = 324 mod 13 = 12
Step 5: h = (12 * 31 + 69) mod 13 = (372 + 69) mod 13 = 441 mod 13 = 12
```

Verification: $441 / 13 = 33$ remainder 12 . So $h(\text{"TABLE"}) = 12$.

The hash value is 12, meaning the string "TABLE" would be placed at index 12 in a table of size 13.

Problem 2: Separate Chaining Insertion Trace

Problem. Insert the keys `{10, 22, 31, 4, 15, 28, 17, 88, 59, 37}` into a separate chaining hash table with `M = 7`, using the hash function `h(k) = k mod 7`. Show the final table and compute the load factor.

Solution.

Hash computations:

$$h(10) = 10 \bmod 7 = 3$$

$$h(22) = 22 \bmod 7 = 1$$

$$h(31) = 31 \bmod 7 = 3$$

$$h(4) = 4 \bmod 7 = 4$$

$$h(15) = 15 \bmod 7 = 1$$

$$h(28) = 28 \bmod 7 = 0$$

$$h(17) = 17 \bmod 7 = 3$$

$$h(88) = 88 \bmod 7 = 4$$

$$h(59) = 59 \bmod 7 = 3$$

$$h(37) = 37 \bmod 7 = 2$$

Final table (new keys prepended to front of chain):

Bucket 0: 28 -> null

Bucket 1: 15 -> 22 -> null

Bucket 2: 37 -> null

Bucket 3: 59 -> 17 -> 31 -> 10 -> null

Bucket 4: 88 -> 4 -> null

Bucket 5: (empty)

Bucket 6: (empty)

Chain lengths: [1, 2, 1, 4, 2, 0, 0]

Load factor: $\alpha = N/M = 10/7 \sim 1.43$

Expected chain length under uniform hashing: $\alpha \sim 1.43$

Actual chain lengths range from 0 to 4. The longest chain (bucket 3, with 4 keys) is significantly above the average. The expected number of probes for a search miss in any random bucket is $\alpha \sim 1.43$. For a search hit, the expected cost is $1 + \alpha/2 \sim 1.71$.

Observe that 2 of the 7 buckets are empty, while bucket 3 has 4 keys. This is characteristic of the "balls-into-bins" distribution: some bins are overloaded while others are empty, even with a perfectly uniform hash function.

Problem 3: Linear Probing Insertion Trace

Problem. Insert the keys $\{10, 22, 31, 4, 15, 28, 17, 88\}$ into a linear probing hash table with $M = 16$, using $h(k) = k \bmod 16$. Show the state of the table after each insertion and identify all clusters.

Solution.

Hash computations:

$h(10) = 10 \bmod 16 = 10$

$h(22) = 22 \bmod 16 = 6$

$h(31) = 31 \bmod 16 = 15$

$h(4) = 4 \bmod 16 = 4$

$h(15) = 15 \bmod 16 = 15$

$h(28) = 28 \bmod 16 = 12$

$h(17) = 17 \bmod 16 = 1$

$h(88) = 88 \bmod 16 = 8$

Insert 10: $h=10$, slot 10 empty. Place at 10.

```
[_, _, _, _, _, _, _, _, _, _, 10, _, _, _, _, _]  
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Insert 22: $h=6$, slot 6 empty. Place at 6.

```
[_, _, _, _, _, _, 22, _, _, _, 10, _, _, _, _, _]
```

Insert 31: $h=15$, slot 15 empty. Place at 15.

```
[_, _, _, _, _, _, 22, _, _, _, 10, _, _, _, _, 31]
```

Insert 4: $h=4$, slot 4 empty. Place at 4.

```
[_, _, _, _, 4, _, 22, _, _, _, 10, _, _, _, _, 31]
```

Insert 15: $h=15$, slot 15 occupied (31). Probe 0: empty. Place at 0.

```
[15, _, _, _, 4, _, 22, _, _, _, 10, _, _, _, _, 31]
```

Insert 28: $h=12$, slot 12 empty. Place at 12.

```
[15, _, _, _, 4, _, 22, _, _, _, 10, _, 28, _, _, 31]
```

Insert 17: $h=1$, slot 1 empty. Place at 1.

```
[15, 17, _, _, 4, _, 22, _, _, _, 10, _, 28, _, _, 31]
```

Insert 88: $h=8$, slot 8 empty. Place at 8.

```
[15, 17, _, _, 4, _, 22, _, 88, _, 10, _, 28, _, _, 31]  
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Load factor: $\alpha = 8/16 = 0.5$

Clusters (maximal contiguous runs of occupied slots):

Since position 15 is occupied and position 0 is occupied, they form a **wrap-around cluster**: {31, 15, 17} spanning positions 15, 0, 1 (length 3).

Final cluster list:

- {31, 15, 17} at positions 15-0-1 (length 3, wraps around)
- {4} at position 4 (length 1)
- {22} at position 6 (length 1)
- {88} at position 8 (length 1)
- {10} at position 10 (length 1)
- {28} at position 12 (length 1)

Total: 6 clusters, sizes [3, 1, 1, 1, 1, 1]. The wrap-around cluster at positions 15-0-1 is the largest. Note how key 15 (home position 15) was displaced to position 0 because slot 15 was already occupied by key 31, and this displacement created a cluster that wraps around the table boundary.

Problem 4: Linear Probing Deletion

Problem. Starting from the table at the end of Problem 3, delete the key 31 using (a) the tombstone method and (b) Knuth's backward-shift method. Show the resulting table for each.

Solution (a): Tombstone method.

Key 31 is at position 15. We mark position 15 as a tombstone (denoted X):

```
[15, 17, _, _, 4, _, 22, _, 88, _, 10, _, 28, _, _, X]
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

The tombstone at position 15 ensures that a search for key 15 (whose home position is 15) still works: the probe starts at 15, finds the tombstone (continue probing), moves to 0, and finds key 15 there.

Solution (b): Knuth's backward-shift method.

Delete key 31 at position 15. Now scan forward from position 0:

- Position 0: key 15, home = $h(15) = 15$. Since position 15 (the gap) is the home position of key 15, and position 0 is to the right of it (cyclically), key 15 should be moved to position 15.

Move key 15 from position 0 to position 15. Gap is now at position 0.

- Position 1: key 17, home = $h(17) = 1$. Is position 0 (the gap) between home position 1 and current position 1 (cyclically)? No -- position 0 is before position 1 in the cyclic order starting from home 1. Do not move.
- Position 2: empty. Stop.

Result:

```
[_, 17, _, _, 4, _, 22, _, 88, _, 10, _, 28, _, _, 15]
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Key 15 has moved from position 0 back to its home position 15, eliminating the need for a tombstone. The wrap-around cluster $\{31, 15, 17\}$ has been broken into $\{15\}$ at 15 and $\{17\}$ at 1, each at their home positions. This is cleaner than the tombstone approach and avoids future probe-sequence degradation.

Problem 5: Load Factor and Resizing

Problem. A separate chaining hash table starts with $M = 4$ buckets and uses the resizing policy: double when $\alpha > 8$, halve when $\alpha < 2$. Starting from empty, determine the table size after each of the following sequences of operations:

- (a) Insert 35 keys.
- (b) Then delete 30 keys.
- (c) Then insert 200 keys.

Solution.

- (a) Start: $M = 4$, $N = 0$.

After inserting 35 keys: we need to track resize triggers.

- After $N > 8 * 4 = 32$ insertions: $\alpha > 8$. Resize to $M = 8$.
- Now with $M = 8$, threshold is $8 * 8 = 64$. After 35 insertions total, $N = 35$, $\alpha = 35/8 = 4.375 < 8$. No further resize.

Final after (a): **M = 8**, $N = 35$, $\alpha = 4.375$.

(b) Delete 30 keys. $N = 35 - 30 = 5$. $\alpha = 5/8 = 0.625$.

Check halving condition: $N < 2 * M$? $5 < 2 * 8 = 16$? Yes. Also $M > 4$? $8 > 4$? Yes.

Resize to $M = 4$. Now check again: $N < 2 * M$? $5 < 2 * 4 = 8$? Yes. $M > 4$? $4 > 4$? No.

Stop. (We do not halve below $M = 4$.)

Final after (b): **M = 4**, $N = 5$, $\alpha = 1.25$.

(c) Insert 200 more keys. Starting from $M = 4$, $N = 5$.

Resizing triggers:

- At $N > 32$ (i.e., after 28 insertions to reach $N = 33$): M doubles to 8. Threshold becomes 64.
- At $N > 64$ (i.e., after 32 more insertions to reach $N = 65$): M doubles to 16. Threshold becomes 128.
- At $N > 128$ (i.e., after 64 more insertions to reach $N = 129$): M doubles to 32. Threshold becomes 256.
- Total insertions = 200, so $N = 5 + 200 = 205$. $\alpha = 205/32 = 6.41 < 8$. No further resize.

Final after (c): **M = 32**, $N = 205$, $\alpha = 6.41$.

Problem 6: Expected Probe Counts

Problem. A linear probing hash table has 1000 entries and 2000 slots (so $\alpha = 0.5$). Compute the expected number of probes for:

- (a) A successful search (search hit).
- (b) An unsuccessful search (search miss).
- (c) If we let the table fill to 1500 entries ($\alpha = 0.75$), recompute both.

Solution.

Using Knuth's formulas:

(a) Search hit at $\alpha = 0.5$:

$$\begin{aligned}
 E[\text{probes}] &= 1/2 * (1 + 1/(1 - \alpha)) \\
 &= 1/2 * (1 + 1/(1 - 0.5)) \\
 &= 1/2 * (1 + 2) \\
 &= 1.5
 \end{aligned}$$

(b) Search miss at $\alpha = 0.5$:

$$\begin{aligned}
 E[\text{probes}] &= 1/2 * (1 + 1/(1 - \alpha)^2) \\
 &= 1/2 * (1 + 1/(0.5)^2) \\
 &= 1/2 * (1 + 4) \\
 &= 2.5
 \end{aligned}$$

(c) At $\alpha = 0.75$:

Search hit:

$$\begin{aligned}
 E[\text{probes}] &= 1/2 * (1 + 1/(1 - 0.75)) \\
 &= 1/2 * (1 + 4) \\
 &= 2.5
 \end{aligned}$$

Search miss:

$$\begin{aligned}
 E[\text{probes}] &= 1/2 * (1 + 1/(0.25)^2) \\
 &= 1/2 * (1 + 16) \\
 &= 8.5
 \end{aligned}$$

Analysis. Increasing the load factor from 0.5 to 0.75 increases the search hit cost from 1.5 to 2.5 probes (a 67% increase) and the search miss cost from 2.5 to 8.5 probes (a 240% increase). The search miss cost is far more sensitive to load factor because it grows quadratically with $1/(1 - \alpha)$. This dramatic increase is why the standard advice is to resize the table (doubling its size) when the load factor reaches 0.5.

Problem 7: Birthday Paradox Application

Problem. A hash table has $M = 10,000$ slots. Using the birthday paradox approximation, after how many insertions should we expect the first collision? If we insert 500 keys, what is the approximate probability of at least one collision?

Solution.

Expected insertions before first collision:

$$\begin{aligned} E[\text{first collision}] &\sim \sqrt{\pi * M / 2} \\ &= \sqrt{\pi * 10000 / 2} \\ &= \sqrt{15708} \\ &\sim 125.3 \end{aligned}$$

So we expect the first collision after approximately **125 insertions**.

For 500 keys in 10,000 slots, the probability of NO collisions is:

$$\begin{aligned} P(\text{no collision}) &= \text{Product}_{\{i=0\}^{499}} (1 - i/10000) \\ &\sim \exp(-\text{Sum}_{\{i=0\}^{499}} i/10000) \\ &= \exp(-499*500 / (2 * 10000)) \\ &= \exp(-249500 / 20000) \\ &= \exp(-12.475) \\ &\sim 3.8 * 10^{(-6)} \end{aligned}$$

So the probability of **at least one collision** is approximately:

$$P(\text{at least one collision}) = 1 - 3.8 * 10^{(-6)} \sim 0.999996$$

After inserting 500 keys into a table with 10,000 slots, it is virtually certain (probability > 99.9999%) that at least one collision has occurred. This illustrates once again the inevitability of collisions, even when the table is only 5% full.

Problem 8: Comparing Separate Chaining and Linear Probing

Problem. A hash table must store 10,000 key-value pairs. Compare the expected search miss cost for:

- (a) Separate chaining with $M = 2,500$ ($\alpha = 4$)
- (b) Separate chaining with $M = 10,000$ ($\alpha = 1$)
- (c) Linear probing with $M = 20,000$ ($\alpha = 0.5$)
- (d) Linear probing with $M = 15,000$ ($\alpha = 0.667$)

Solution.

(a) Separate chaining, $\alpha = 4$:

$$E[\text{miss}] = \alpha = 4 \text{ comparisons}$$

(b) Separate chaining, $\alpha = 1$:

$$E[\text{miss}] = \alpha = 1 \text{ comparison}$$

(c) Linear probing, $\alpha = 0.5$:

$$\begin{aligned} E[\text{miss}] &= 1/2 * (1 + 1/(1 - 0.5)^2) \\ &= 1/2 * (1 + 4) \\ &= 2.5 \text{ probes} \end{aligned}$$

(d) Linear probing, $\alpha = 2/3$:

$$\begin{aligned} E[\text{miss}] &= 1/2 * (1 + 1/(1 - 2/3)^2) \\ &= 1/2 * (1 + 9) \\ &= 5.0 \text{ probes} \end{aligned}$$

Summary of results:

Configuration	alpha	Memory (approx.)	E[miss]
Chaining, M=2500	4.0	2500 pointers + 10000 nodes	4.0
Chaining, M=10000	1.0	10000 pointers + 10000 nodes	1.0
Linear probing, M=20000	0.5	20000 slots	2.5
Linear probing, M=15000	0.67	15000 slots	5.0

The optimal choice depends on the application's priorities. If memory is constrained, chaining with M=2500 stores 10,000 keys in a compact structure at the cost of 4 comparisons per miss. If speed is paramount, chaining with M=10,000 gives the best miss performance but uses more memory (10,000 bucket pointers plus 10,000 linked list nodes). Linear probing with M=20,000 offers a good balance: 2.5 probes per miss with excellent cache behavior, using 20,000 array slots (no pointer overhead). Linear probing with M=15,000 saves memory over the M=20,000 configuration but doubles the expected miss cost.

8. Summary

8.1 Key Takeaways

Hash tables represent a fundamentally different approach to the symbol table problem. By computing an array index directly from the key, rather than comparing keys against each other, hash tables bypass the $\Omega(\log N)$ comparison-based lower bound and achieve expected $O(1)$ performance for search and insert operations.

The two primary collision resolution strategies are:

Separate chaining maintains a linked list at each bucket. It is simple to implement, robust to high load factors, and supports easy deletion. The expected cost of all operations is proportional to the load factor $\alpha = N/M$. The standard resizing policy (double at $\alpha > 8$, halve at $\alpha < 2$) keeps α in a range where chains are short and operations are fast.

Linear probing stores all keys in the table array, probing successive slots on collision. It is memory-efficient and cache-friendly but suffers from the clustering phenomenon, where occupied slots tend to form long contiguous runs. Knuth's analysis shows that the expected cost of a search miss is $\frac{1}{2} * (1 + \frac{1}{1 - \alpha})$.

- α^2), which grows quadratically as α approaches 1. The standard resizing policy (double at $\alpha > 1/2$) keeps the load factor in the range where performance is near-optimal.

8.2 Hash Tables vs. Balanced BSTs

The fundamental trade-off between hash tables and balanced BSTs is:

Criterion	Hash Table	Balanced BST (Red-Black)
Search (expected)	$O(1)$	$O(\log N)$
Insert (expected)	$O(1)$	$O(\log N)$
Search (worst case)	$O(N)$	$O(\log N)$
Insert (worst case)	$O(N)$	$O(\log N)$
Ordered operations	Not supported	$O(\log N)$
Min/Max	$O(N)$	$O(\log N)$
Range queries	$O(N)$	$O(\log N + R)$
Key requirement	equals + hash	compareTo
Memory overhead	Low (linear probing)	High (3 ptrs/node)
Cache performance	Excellent (linear probing)	Poor
Worst-case guarantee	No (unless universal hashing)	Yes

Use hash tables when:

- You need only search, insert, and delete (no ordered operations).
- Expected $O(1)$ performance is more important than worst-case guarantees.
- Keys support hashing but not necessarily a total ordering.
- Cache performance is important (favor linear probing).

Use balanced BSTs when:

- You need ordered operations (min, max, rank, select, range queries, sorted iteration).
- Worst-case $O(\log N)$ guarantees are required.
- The key set is adversarial or unpredictable (BSTs are robust without universal hashing).

8.3 Complete Symbol Table Comparison

Implementation	Search (avg)	Insert (avg)	Delete (avg)	Ordered?	Notes
Sequential search	$N/2$	N	N	No	Simple; small tables only
Binary search	$\log N$	N	N	Yes	Static tables (few inserts)
BST	$1.39 \log N$	$1.39 \log N$	$\text{sqrt}(N)^*$	Yes	*Degrades with random delete
Red-black BST	$\log N$	$\log N$	$\log N$	Yes	Guaranteed performance
Separate chaining	~ 1	~ 1	~ 1	No	Simple; flexible load factor
Linear probing	~ 1	~ 1	~ 1	No	Cache-friendly; $\alpha < 1/2$

8.4 Looking Ahead

Hash tables and balanced BSTs are the two most important symbol table implementations in practice. Most standard libraries offer both: Java has `HashMap` and `TreeMap`, C++ has `unordered_map` and `map`, Python has `dict` (hash table) and `SortedContainers` (third-party balanced tree). Understanding the strengths and limitations of each is essential for choosing the right data structure for any given application.

In subsequent lectures, we will see hash tables and BSTs used as building blocks in more complex algorithms and data structures, including graph algorithms (adjacency lists stored in hash tables), string algorithms (hash tables for substring search), and geometric algorithms (balanced BSTs for sweep line methods).

9. References

1. Sedgewick, R. and Wayne, K. (2011). *Algorithms*, 4th Edition. Addison-Wesley Professional. Chapters 3.4 (Hash Tables) and 3.5 (Applications).

2. Knuth, D.E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition. Addison-Wesley. Section 6.4 (Hashing).
 3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapter 11 (Hash Tables).
 4. Carter, J.L. and Wegman, M.N. (1979). "Universal Classes of Hash Functions." *Journal of Computer and System Sciences*, 18(2):143-154.
 5. Pagh, R. and Rodler, F.F. (2004). "Cuckoo Hashing." *Journal of Algorithms*, 51(2):122-144.
 6. Celis, P., Larson, P.-A., and Munro, J.I. (1985). "Robin Hood Hashing." *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pp. 281-288.
 7. Crosby, S.A. and Wallach, D.S. (2003). "Denial of Service via Algorithmic Complexity Attacks." *Proceedings of the 12th USENIX Security Symposium*, pp. 29-44.
 8. Knuth, D.E. (1963). "Notes on Open Addressing." Unpublished memorandum. (The original analysis of linear probing, later incorporated into TAOCP Vol. 3.)
 9. Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press. Chapter 5 (Balls, Bins, and Random Graphs).
 10. Thorup, M. (2009). "String Hashing for Linear Probing." *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 655-664.
-

Appendix: Companion Code Reference

The companion C++ implementation file provides the following components for hands-on experimentation:

polynomialHash(s, B, M) -- Computes the polynomial rolling hash of a string using Horner's method with base **B** and modulus **M**. Demonstrates the $O(n)$ hash computation discussed in Section 1.5.

ChainingHashTable -- Full separate chaining implementation with:

- **insert(key, value)** -- Insert or update a key-value pair, with automatic resizing when $\alpha > 8$
- **search(key)** -- Search for a key, returning its value or a sentinel
- **remove(key)** -- Delete a key, with automatic resizing when $\alpha < 2$
- **loadFactor()** -- Returns the current load factor N/M

- `printTable()` -- Visual display of all chains for debugging and analysis
- `chainLengths()` -- Returns a vector of chain lengths for distribution analysis

LinearProbingHashTable -- Full linear probing implementation with:

- `insert(key, value)` -- Insert with linear probing, automatic resizing when $\alpha > 1/2$
- `search(key)` -- Search with tombstone-aware probing
- `remove(key)` -- Delete using tombstone markers
- `loadFactor()` -- Returns the current load factor
- `countClusters()` -- Returns cluster sizes for empirical study of the clustering phenomenon

Collision Analysis -- A driver function that inserts random keys at various load factors (0.25, 0.50, 0.75, 0.90) and reports the empirical average number of probes for search hits and misses, allowing comparison with the theoretical predictions of Propositions D and E.

Students are encouraged to run the companion code, vary the table sizes and hash functions, and observe how empirical results align with the theoretical analysis presented in this lecture.

Lecture 8 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition