Hash Tables

Mina Gabriel

Harrisburg University

May 30, 2024

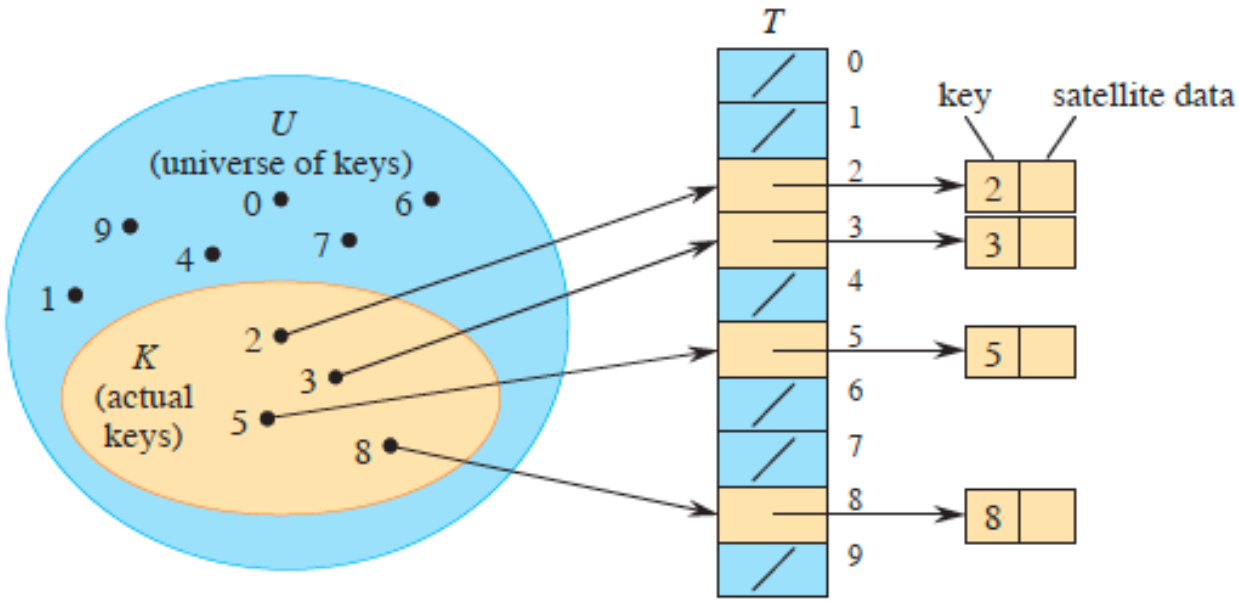
Hash Tables in Dictionary Operations

- Hash tables support essential dictionary operations: INSERT, SEARCH, and DELETE.
- Used in various applications, like symbol tables in compilers, where keys are identifiers in a programming language.
- Offers efficient searching, typically O(1) average time under reasonable assumptions, compared to O(n) in the worst case.
- Python's built-in dictionaries are implemented using hash tables.

Handling Collisions and Efficient Storage

- Hash tables generalize the concept of arrays, using dynamic sizing to handle a vast range of keys efficiently.
- Chaining and open addressing are methods to resolve collisions—situations where multiple keys map to the same index.
- Effective even when the number of actual keys is small compared to the potential number of keys, optimizing space.
- Design considerations for hash tables include handling of hierarchical memory systems to ensure performance.

Direct-Address Tables



- Direct addressing works well when the universe U of keys is small.
- Consider a dynamic set where each element has a unique key from $U = \{0, 1, \ldots, m-1\}$, with m not too large.
- A direct-address table, $T[0 \ldots m-1]$, is used where each slot corresponds to a key in U.

Optimizations and Use Cases

- If an element with key k exists, slot k points to it; if not, $T[k] = \text{NIL}$.
- Operations: DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE are simple, taking $O(1)$ time.
- For some applications, direct-address tables can store elements directly in the slots, saving space.
- Instead of linking to external objects, the object's data is stored right in the table, with a special marker for empty slots.
- The key of an object might not need to be stored, as its index can serve as the key, optimizing space and access time.

Hash Tables and Handling Collisions

- The limitation of direct addressing arises when the universe U is large, leading to impractical memory usage if $|U|$ is large.
- Hash tables are more memory-efficient when the set K of keys actually stored is much smaller than U, reducing storage to $O(|K|)$ while maintaining $O(1)$ average search time.
- In hash tables, a hash function h maps keys to slots: $h : U \to \{0, 1, \ldots, m-1\}$, where m is much smaller than $|U|$.
- A simple hash function example is $h(k) = k \mod m$.
- Collisions—when two keys hash to the same slot—are possible, but various techniques exist to resolve these conflicts efficiently.

Illustration of Collision

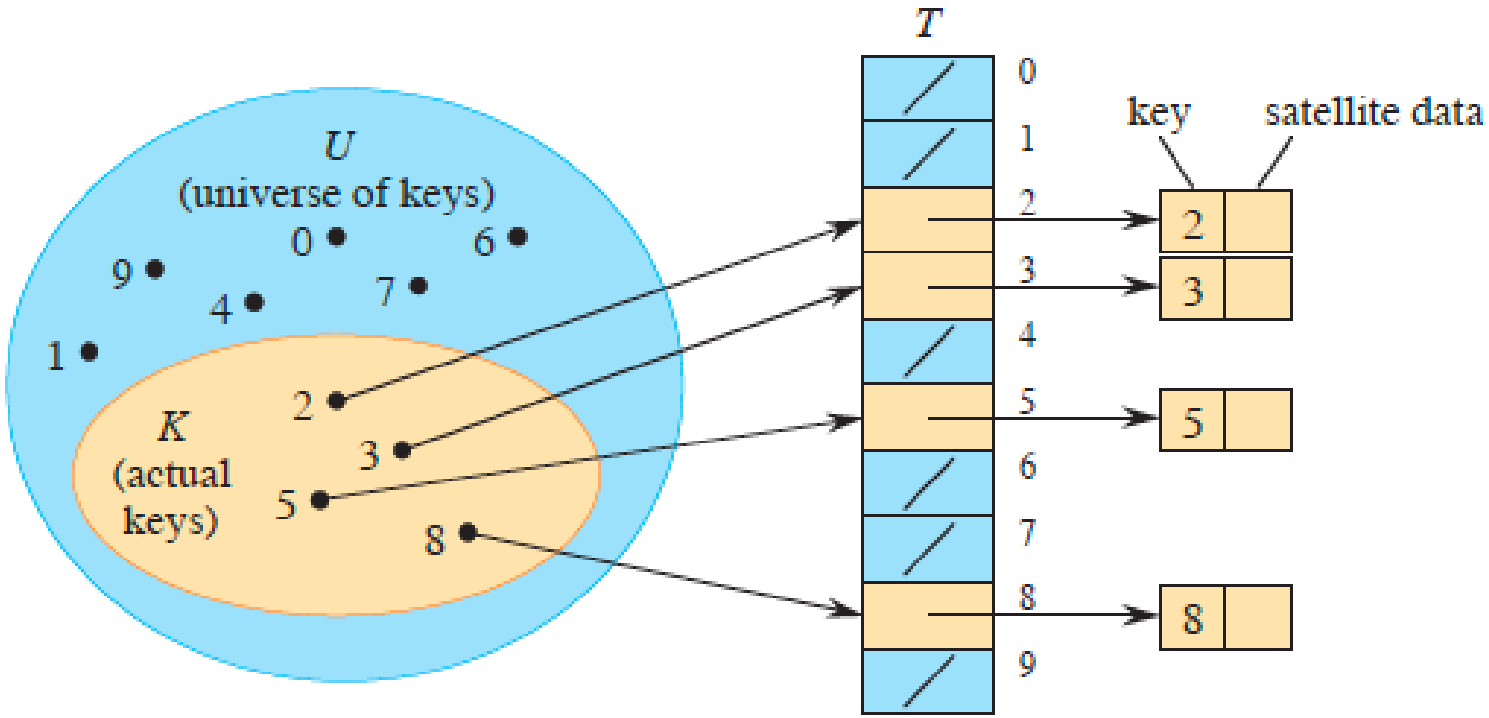


Figure: Using a hash function h to map keys to hash-table slots. Because keys k 2 and k 5 map to the same slot, they collide.

Illustration of Collision

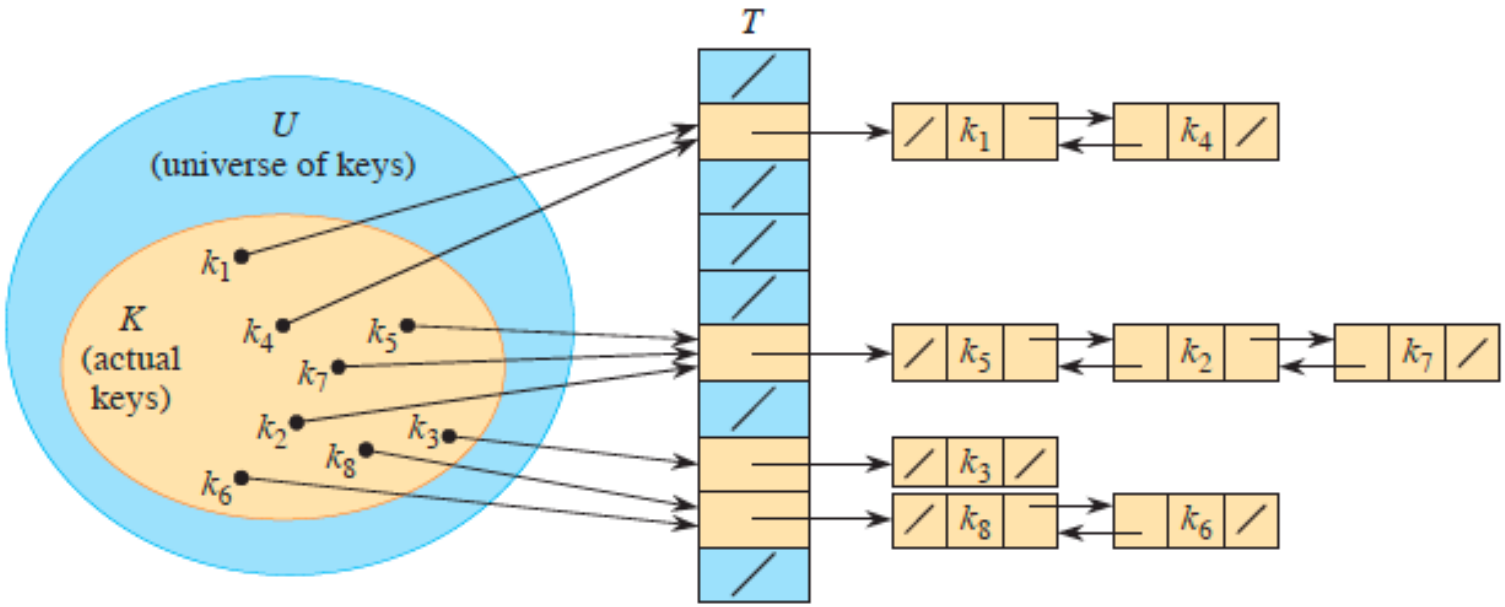


Figure: Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked.

Dictionary Operations with Chaining

- Insertion is typically $O(1)$, assuming no prior existence of the element in the table.
- Search time is proportional to the list length; specific analysis required for average-case time.
- Deletion is $O(1)$ in doubly linked lists, facilitating direct element removal without a preceding search.
- To maintain efficiency, particularly for deletion, the use of doubly linked lists is recommended.

Analysis of Hashing with Chaining - Introduction

- For a hash table T with m slots storing n elements, define the *load factor* $\alpha = \frac{n}{m}$, representing the average number of elements per chain.
- Worst-case scenario: All n keys hash to the same slot, resulting in a list of length n and search time $O(n)$.
- Average-case performance relies on how well the hash function h distributes keys among m slots.
- The hash function should be *uniform*, meaning each key is equally likely to hash into any slot, independent of other keys.

Independent, Uniform, and Universal Hashing

- **Independent Hashes:** Hashing one key does not affect the hashing of another.
- **Uniform Hashes:** Each key is equally likely to be assigned to any of the hash table slots.
- **Universal Hashing:** A class of hash functions where any two distinct keys are unlikely to collide.

Hashing Example

Given a hash table with 10 slots ($m = 10$) and keys 23, 42, 35:

- Hash function: $h(k) = (ak + b) \mod 10$
- Choose random coefficients a and b, e.g., $a = 3$, $b = 7$
- Applying h to our keys:
  - $h(23) = (3 \times 23 + 7) \mod 10 = 6$
  - $h(42) = (3 \times 42 + 7) \mod 10 = 3$
  - $h(35) = (3 \times 35 + 7) \mod 10 = 2$
- Probability of a collision for any two keys is $\frac{1}{10}$.

Theorems on Hashing

Theorems 11.1 and 11.2 CLRS

In a hash table in which collisions are resolved by chaining, an unsuccessful or successful search takes $\Theta(1+\alpha)$ time on average, under the assumption of independent uniform hashing.

Static Hashing

- **Static Hash Function:** The hash function is fixed and does not change during the lifetime of the hash table.
- **Randomness Assumption:** The randomness in static hashing comes from the assumed randomness in the distribution of input keys, not from the hash function itself.

Example of Poor Distribution in Static Hashing

- Static hash function: $h(k) = k \mod 10$
- All keys are multiples of 10.
- Poor distribution: all keys hash to the same slot.

Key	**Hash Slot**
10	$h(10) = 10 \mod 10 = 0$
20	$h(20) = 20 \mod 10 = 0$
30	$h(30) = 30 \mod 10 = 0$
...	...
1000	$h(1000) = 1000 \mod 10 = 0$

Table: Hashing to slot 0

Issue: Each key (10, 20, 30, ..., 1000) when hashed using $h(k)$, results in 0, causing a collision at slot 0.

The Division Method for Hash Functions

- **Hash Function Creation:** Map a key k into one of m slots by calculating $k \mod m$.
- **Efficiency:** Requires only a single division operation, making it a fast method for hashing.
- **Practical Consideration:** While fast, there's no performance guarantee for all cases, and using primes may add complexity.

Example: For a hash table of size $m = 12$ and a key $k = 100$:

$$h(k) = k \mod m = 100 \mod 12 = 4$$

The key 100 would be placed in slot 4 of the hash table.

Why Prime Number Table Size

- **Optimal Table Size:** Best performance when m is a prime number, avoiding values close to powers of 2 for uniform key distribution.

Prime numbers are not factors of any number except 1 and themselves, so when you use a prime number for m, the keys are less likely to share common factors with m, which results in a more even distribution of hash values. This is particularly important when the keys have patterns or are not random.

Numerical Example: Prime vs. Power of 2 for Hashing

- **Using Power of 2:**
  - Hash function: $h(k) = k \mod 8$
  - Keys: 8, 16, 24, 32
  - All keys hash to the same slot, causing collisions:
    - $h(8) = 0$
    - $h(16) = 0$
    - $h(24) = 0$
    - $h(32) = 0$
- **Using a Prime Number:**
  - Hash function: $h(k) = k \mod 11$
  - Keys: 8, 16, 24, 32
  - Keys are more evenly distributed:
    - $h(8) = 8$
    - $h(16) = 5$
    - $h(24) = 2$
    - $h(32) = 10$

The Multiply-Shift Method for Hash Functions

Content:

- The Multiply-Shift method is efficient for computing hash functions when the number of slots in a hash table is a power of two (2^ℓ).
- To use this method, select a constant multiplier a, which is an odd value less than 2^w, where w is the machine word size.

Multiplier a:

- Choose an odd number a such that $0 < a < 2^w$.

Multiplication:

- Multiply the key k by a to get a $2w$-bit product.

Modulo and Bit Shifting:

- Taking this product modulo 2^w discards the high-order w bits, leaving only the low-order w bits, r_0.
- Perform a right logical shift on r_0 by $(w - \ell)$ bits to obtain the ℓ most significant bits of r_0.

Computing the Hash Value

Example:

- Given $k = 123456$, $\ell = 14$, $m = 2^{14}$, and $w = 32$.
- The multiplier a is 2654435769, suitable for a 32-bit machine word.

The hash function $h_a(k)$ is computed as:

$$h_a(k) = ((k \cdot a) \mod 2^w) \ggg (w - \ell)$$

- This hash function effectively maps the key k to an address in the hash table.
- The shift operation $\ggg$ denotes the logical right shift, moving bits to the right and filling the left with zeros.
- The hash value is the ℓ most significant bits after the shift, fitting within the 2^ℓ slots of the hash table.

The multiply-shift method

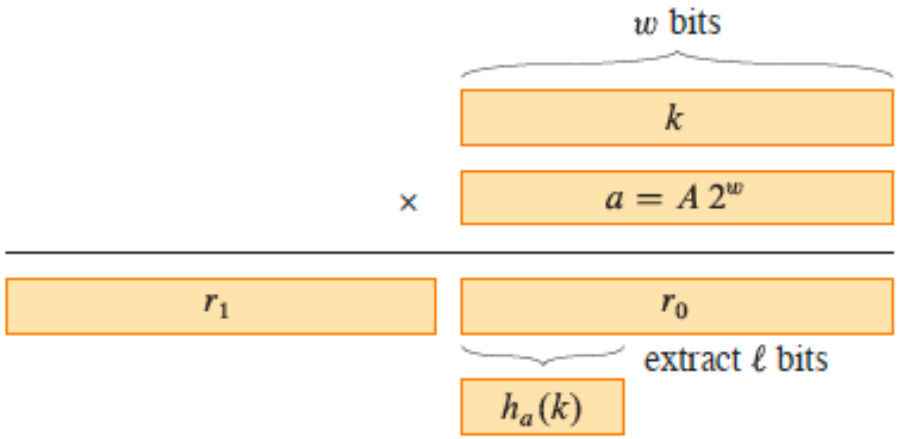


Figure: The multiply-shift method to compute a hash function. The w-bit representation of the key k is multiplied by the w-bit value $a = |A|2^w$. The ℓ highest-order bits of the lower w-bit half of the product form the desired hash value $h_a(k)$.

Advantages and Usage of Multiply-Shift

Content:

- The method is fast due to simple bitwise operations and multiplication.
- It is best used when the hash table size is 2^ℓ to ensure a uniform distribution of hash values.
- While fast and efficient, it doesn't guarantee good average-case performance; for applications requiring such guarantees, consider universal hashing.