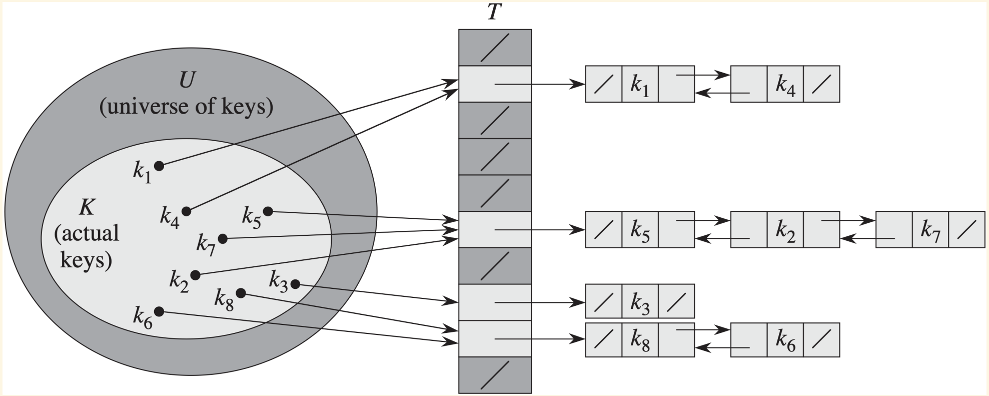
Introduction to Hash Tables

Michael T. Gastner (16 March 2023)



Disclaimer: These slides are based on and occasionally quote from ‘Introduction to Algorithms’ (3rd ed.) by Cormen et al. (2009), MIT Press.

Learning Objectives

By the End of This Lesson, You Should Be Able to . . .

- Define what a hash table is.
- State desirable properties of hash tables and hash functions.
- Describe collision resolution by chaining.
- Calculate hash values using the division and multiplication methods.
- Design a universal class of hash functions.

What is a Hash Table?

Abstract Data Type that Associates Keys with Values Using a Hash Function

A **hash table** is an abstract data type that associates **keys** (i.e. unique identifiers) with **values** (also known as 'records') that are stored in an array. A **hash function** is used to assign a unique array index to each key such that the value (or a pointer to the value) is stored at that index.

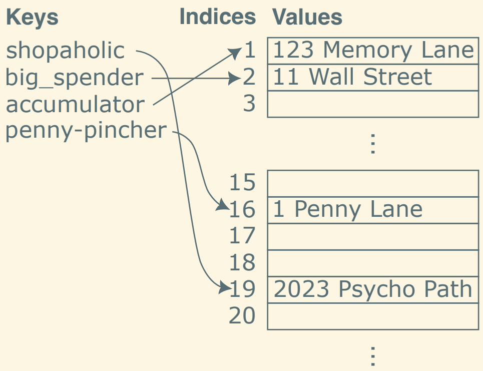
The position in the array where the value v (or the pointer to v) is stored is called the **slot** of v.

Example of a Hash Table

Storing addresses by usernames

An online retailer stores the mailing address for each username in a hash table. As a hash function, it uses the position of the first letter of the username in the English alphabet: a = 1, . . . , z = 26.

One problem with this method is that multiple user names may start with the same letter, which leads to a **collision** (i.e. different keys map to the same array index).



What Properties Should a Hash Table Possess?

Search, Insertion and Deletion in Near-Constant Time on Average

A hash table should provide fast operations for:

- Searching: Given a random key, return the corresponding value.
- Inserting: Add one key-value pair to the data structure.
- Deleting: Given a pointer to a value in the hash table, delete the key and the corresponding value.

We will prove that hash tables can achieve an average running time of $O(1)$ for all three of these 'dictionary' operations under realistic assumptions.

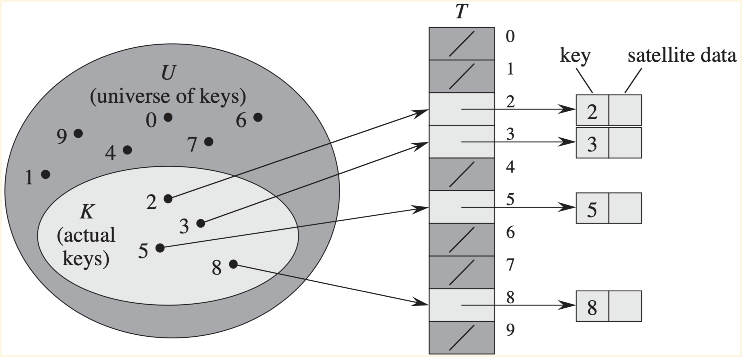
By comparison, a linked list needs $O(n)$ time on average for searching. A self-balancing binary search tree (e.g. a red-black tree) can perform all three operations in $O(\log n)$ time, but it takes $O(n \log n)$ time to initialise it (i.e. sort the elements).

Direct-Address Tables

Reserving One Slot for Each Possible Key

If the 'universe' of all possible keys is small (e.g. all digits from 0 to 9), we can use a special type of hash table known as a **direct-address table**, in which each slot corresponds to one key.

The hash function for direct-address tables is the identity (i.e. the key is equal to the array index).



Direct-Address Tables

Pseudocode for Dictionary Operations

Denoting the direct-address table by T, we perform the three dictionary operations as follows:

```
DIRECT-ADDRESS-SEARCH$(T, k)$
1   return $T[k]$
```

```
DIRECT-ADDRESS-INSERT$(T, x)$
1   return $T[x.key] = x$
```

```
DIRECT-ADDRESS-DELETE$(T, x)$
1   return $T[x.key] =$ NIL
```

Each these operations takes $O(1)$ time in the worst case.

Usually, Hash Tables Have Fewer Slots Than Keys

Still, They Achieve $O(1)$ Average Running Time

If there is a large number of keys (e.g. all possible usernames of a website) in the universe U, it may be impossible to hold the complete direct-address table in memory. Furthermore, the set K of keys that are actually stored is usually a small subset of U; in this case, a direct-address table wastes memory.

Hash tables typically reduce the memory requirement to $O(|K|)$ at the expense of an $O(|K|)$ *worst-case* running time for the dictionary operations.

However, the *average* running time does not exceed $O(1)$, and, with clever choices of the hash functions, make it highly unlikely to experience the worst-case running time.

Hashing

Assigning a Hash Value to a Key

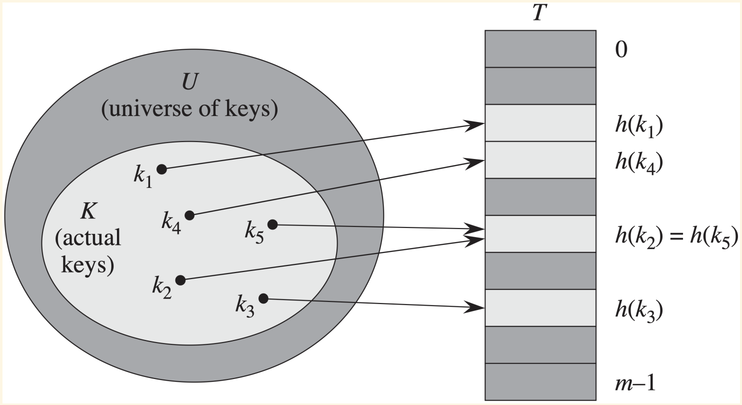
With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, we use a hash function h to compute the slot from the key k. Here, h maps the universe U of keys into the slots of a hash table $T[0\,..\,m-1]$:

$$h : U \to \{0, 1, \dots, m-1\},$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k **hashes** to slot $h(k)$; we also say that $h(k)$ is the **hash value** of key k.

Hashing
Illustration

The figure below illustrates the basic idea. The hash function reduces the range of array indices and, hence, the size of the array from $|U|$ to m. Because keys k_2 and k_5 map to the same slot, they **collide**.



Hashing

How to Handle Collisions

Because $|U| > m$, there must be at least two keys that have the same hash value. Thus, we must find a way to resolve collisions. There are two common methods:

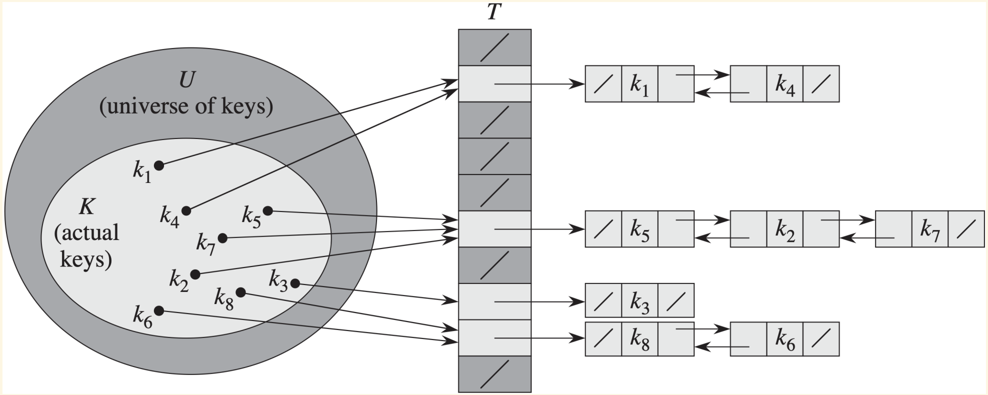
- **Chaining:**
  The hash-table does not store values directly. Instead, each slot $T[j]$ contains a pointer to a linked list of all the keys whose hash value is j. Chaining is illustrated on the next page.
- **Open addressing:**
  Values are stored directly in the hash table but not necessarily in the slot returned by the hash function.

In this lesson, we focus on chaining because its average running time is easier to derive.

Hashing

Collision Resolution by Chaining

In this example, the keys k_2, k_5 and k_7 collide; thus, we store them together with their value in a linked list. We assume that the list is doubly rather than singly linked to enable fast deletion.



Dictionary Operations Using Chaining
Pseudocode

```
CHAINED-HASH-INSERT($T, x$)
1  insert $x$ at the head of list $T[h(x.key)]$
   // $O(1)$ worst-case running time.

CHAINED-HASH-SEARCH($T, k$)
1  search for an element with key $k$ in list $T[h(k)]$
   // $\Theta(n)$ worst-case running time. However, we will show that the average running
   // time is $O(1)$ under realistic assumptions.

CHAINED-HASH-DELETE($T, x$)
1  delete $x$ from the list $T[h(x.key)]$
   // $O(1)$ worst-case running time, assuming that we are given a pointer to $x$ so
   // that we do not have to search for $x$ first.
```

Insertion Using Chaining

Exercise

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, and 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

Insertion Using Chaining
Solution to Exercise

Numbers appearing to the left in the table have been inserted later.

Array index	Linked list
0	$\emptyset$
1	10, 19, 28
2	20
3	12
4	$\emptyset$
5	5
6	33, 15
7	$\emptyset$
8	17

Analysis of Hashing with Chaining
Notation and Assumptions

An important quantity in hashing is the **load factor** defined as

$$\alpha = \frac{n}{m} \,,$$

where m is the number of slots in the hash table and n is the number of stored keys. We can interpret α as the average length of a linked list attached to a random slot.

We assume that a random key is equally likely to hash into any of the m slots, independently of the hash values of other keys. We say that a hash function that results in this feature performs **simple uniform hashing**.

We further assume that the hash function needs only $O(1)$ time to calculate the hash value from any key.

Analysis of Hashing with Chaining

Average Running Time of an Unsuccessful Search is $\Theta(1+\alpha)$

If and only if we are searching for a key that is not stored in the hash table, the search will be unsuccessful. What is the average running time of an unsuccessful search?

Before we can infer that the key k is not in the slot $h(k)$, we must access all elements in the list. When using simple uniform hashing, the average length of the list $T[h(k)]$ is α.

Including the $O(1)$ time to calculate $h(k)$, the average time of an unsuccessful search is $\Theta(1+\alpha)$.

Analysis of Hashing with Chaining

Average Running Time of a Successful Search is Also $\Theta(1+\alpha)$

It is possible to show that a successful search also needs $\Theta(1+\alpha)$ time on average. However, the proof is more complicated than for the case of an unsuccessful search because the probability that a list is searched is proportional to the number of elements it contains. See Section 11.2 in Cormen et al. (2009) for details.

In summary, the average time required by a search is $\Theta(1+\alpha)$, regardless of whether the search is successful or not. If the number of slots is proportional to the number of elements, then $\alpha = O(1)$, and we can perform all dictionary operations in $O(1)$ on average.

Desirable Properties of Hash Functions

Uniformity, Independence, Determinism, and Efficiency

A good hash function should have the following properties:

- **Uniformity:** It should distribute the keys uniformly across the hash table to minimise the probability of worst-case behavior during key search.
- **Independence:** It should generate the hash value of each key independently of any other key. Otherwise, patterns in the keys could lead to patterns in the hash value, potentially causing some linked lists to contain a large number of elements.
- **Determinism:** It should map the same key to the same location in the hash table every time it is computed. Otherwise, it would be impossible to perform dictionary operations efficiently.
- **Efficiency:** It should require minimal time and resources to compute the hash value.

Division method

Integer Key Modulo a Prime Number

Let us assume that every key is a nonnegative integer; if necessary, we can create a surrogate key by applying a function that maps each possible input key to a unique nonnegative integer (e.g., A $\rightarrow$ 0, B $\rightarrow$ 1, etc.).

The **division method** creates hash values by performing simple arithmetic on the nonnegative integer key k:

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

where m is the number of slots, and mod is a shorthand notation for the modulo operation, which returns the integer remainder after dividing by m.

To achieve simple uniform hashing, m should be chosen as a prime number not too close to a power of 2 (i.e., 1, 2, 4, 8, etc.).

Multiplication Method

Involving Multiplications, the Modulo Operator and the Floor Function

The **multiplication method** applies the hash function

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

where

- m is the number of slots. An integer power of 2 (e.g. $m = 2^{14} = 16384$) is recommended so that the multiplication with m can be carried out as a bit shift.
- k is the key.
- A is a constant in the open interval $(0, 1)$. A value suggested in the literature is $A = \frac{\sqrt{5}-1}{2}$, which is the inverse of the golden ratio.
- the floor function $\lfloor x \rfloor$ returns the largest integer smaller than x.
- the operation '$\text{mod } 1$' returns the fractional part of its operand (i.e. $x \bmod 1 = x - \lfloor x \rfloor$).

Multiplication Method
Exercise

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

Multiplication Method

Exercise

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

Solution:
The hash values are 700, 318, 936, 554 and 172, respectively.

Because m is not an integer power of 2, we cannot use bit shifting to compute the multiplication. Therefore, this hash function is not efficient.

Moreover, all hash values are even numbers, which suggests that this hash function does not provide simple uniform hashing. Thus, it should not be used in practice.

Selecting Hash Functions Randomly

Avoiding Consistently Poor Performance for any Given Set of Keys

Although a good hash function h is likely to reduce the number of collisions, there is still a small probability that h exhibits poor performance for a given sequence of keys. If h is the same for every execution of the program, we may never be able to obtain results within a reasonable time.

Instead of fixing a hash function, we can select a random hash function at the beginning of execution, which will then be used throughout the rest of the execution. Because the hash function is random, the algorithm can behave differently on each execution, making it unlikely to suffer from consistently poor performance.

Universal Hashing

Random Hash Functions with Provably Good Performance on Average

A set of hash functions $\mathcal{H}$ is said to be **universal** if, for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$.

In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k and l is no more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

Designing a Universal Class of Hash Functions

The $\mathcal{H}_{pm}$ family

For an integer q, we define $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ and $\mathbb{Z}_q^* = \{1, 2, \ldots, q-1\}$.

Let p be a prime number that is large enough to guarantee that $U \subset \mathbb{Z}_p$.

For any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$, we define the hash function h_{ab} as

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m.$$

One can prove that the family $\mathcal{H}_{pm}$ of all such hash functions,

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\},$$

is universal. It is guaranteed to have a good performance on average. The number of slots m does not need to be a prime number.

Universal Hashing

Exercise

Consider the hash function $h_{3,4}$ from the $\mathcal{H}_{17,6}$ family. What is $h_{3,4}(8)$?

Universal Hashing

Exercise

Consider the hash function $h_{3,4}$ from the $\mathcal{H}_{17,6}$ family. What is $h_{3,4}(8)$?

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 \end{aligned}$$