CSCI 470: Hashing

Vijay Chaudhary

October 15, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. Hashing

2. Analysis of hashing with chaining

Hashing

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- We assume that the maximum value is not too large.

Direct address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- We assume that the maximum value is not too large.
- We also assume that no two elements have the same key.

Direct Address Table: Figure

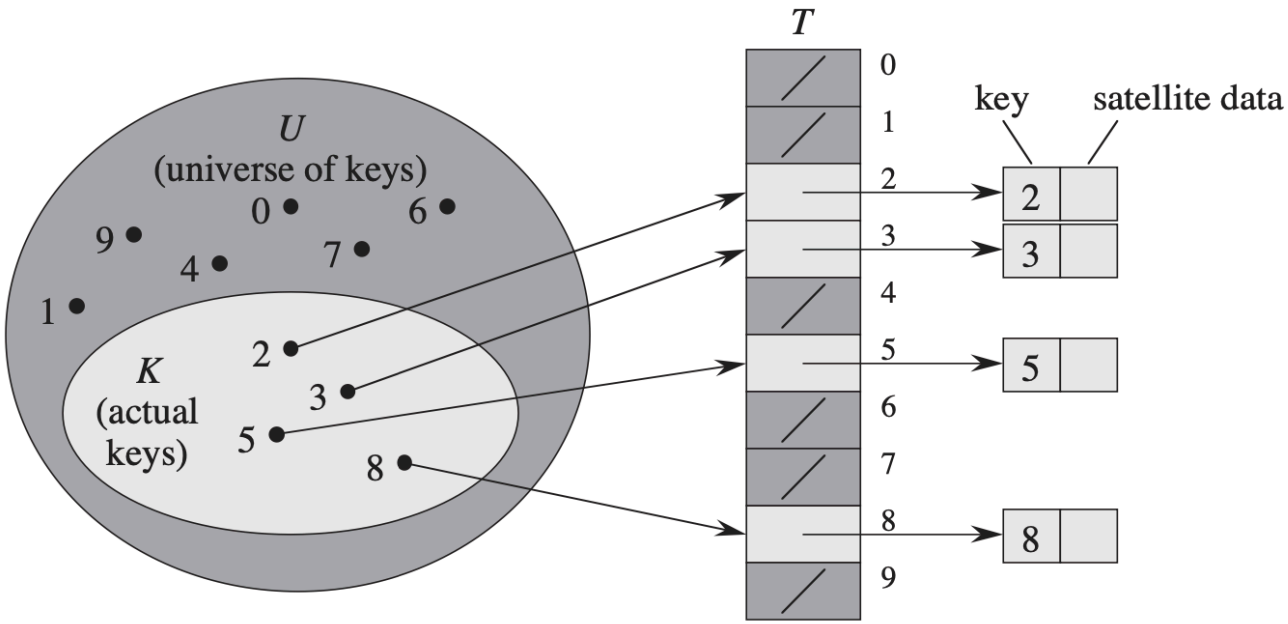

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Direct address tables

- With $U = \{0, 1, ..., m - 1\}$, we can use an array for **direct-address table** such as $T[0 \,..\, m - 1]$, in which each position, or **slot**, corresponds to a key in the universe U.
- Think of the array to keep the counts in counting sort, where we used the index to keep the count of the element. Similarly, we are using each position in the array as a slot that correspond to the key.
- Either we can use the $T[k]$ to store the key itself or we can have a pointer at $T[k]$ that points to the key. In either case, we need to find a way to represent when the slot is empty.

Dictionary operations

DIRECT-ADDRESS-SEARCH

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE

1 $T[x.key] = \text{NIL}$

Pros and cons of direct address tables

Pros:

- Implementation is trivial.
- insertion, deletion, and lookup can be done at $O(1)$.

Cons:

- Table will be very sparse if there is a large gap between the maximum value and second largest value.
- Only useful if the maximum value is very small.

Hash tables

- 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.

Hash tables

- 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.
- $h : U \to \{0, 1, \ldots, m-1\}$ into the slots of a **hash table** $T[0 \,..\, m-1]$, where the size of $m \ll |U|$.

Hash tables

- 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.
- $h : U \to \{0, 1, \ldots, m-1\}$ into the slots of a **hash table** $T[0 \,..\, m-1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.

Hash tables

- 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.
- $h : U \to \{0, 1, \ldots, m-1\}$ into the slots of a **hash table** $T[0 \,..\, m-1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.
- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.

Hash tables

- 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.
- $h : U \to \{0, 1, \ldots, m-1\}$ into the slots of a **hash table** $T[0 \,..\, m-1]$, where the size of $m \ll |U|$.
- We say that an element with key k ***hashes*** to slot $h(k)$ is the **hash value** of key k.
- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.
- Since $|U| > m$, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

Hash Table: figure

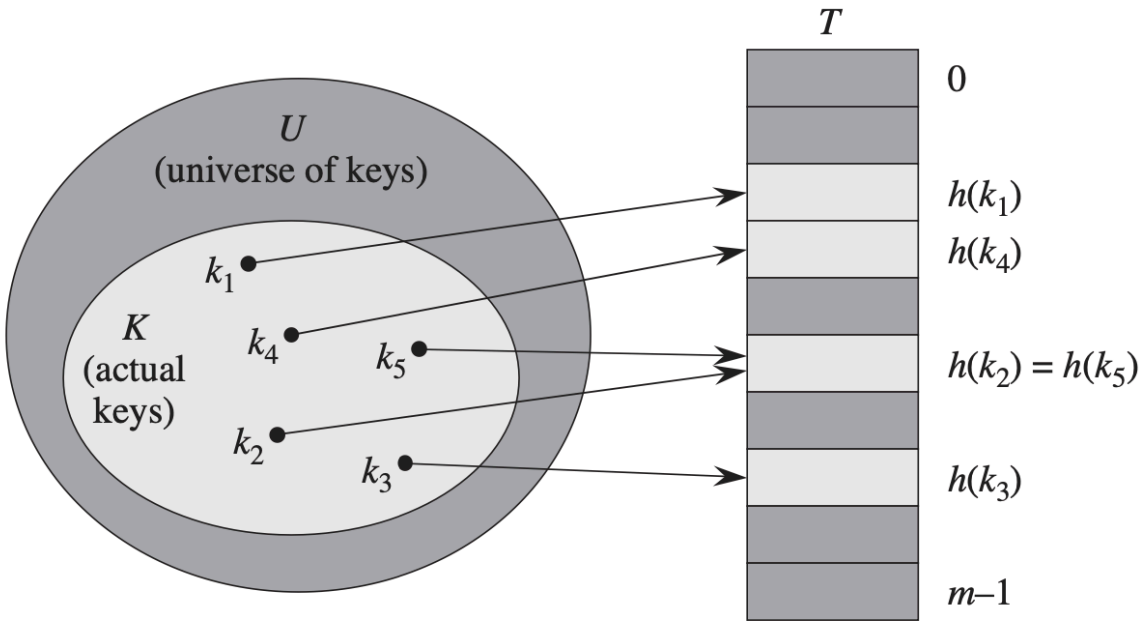

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Collision resolution by chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as Figure 11.3 shows.
- Slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.

Hash Table: figure

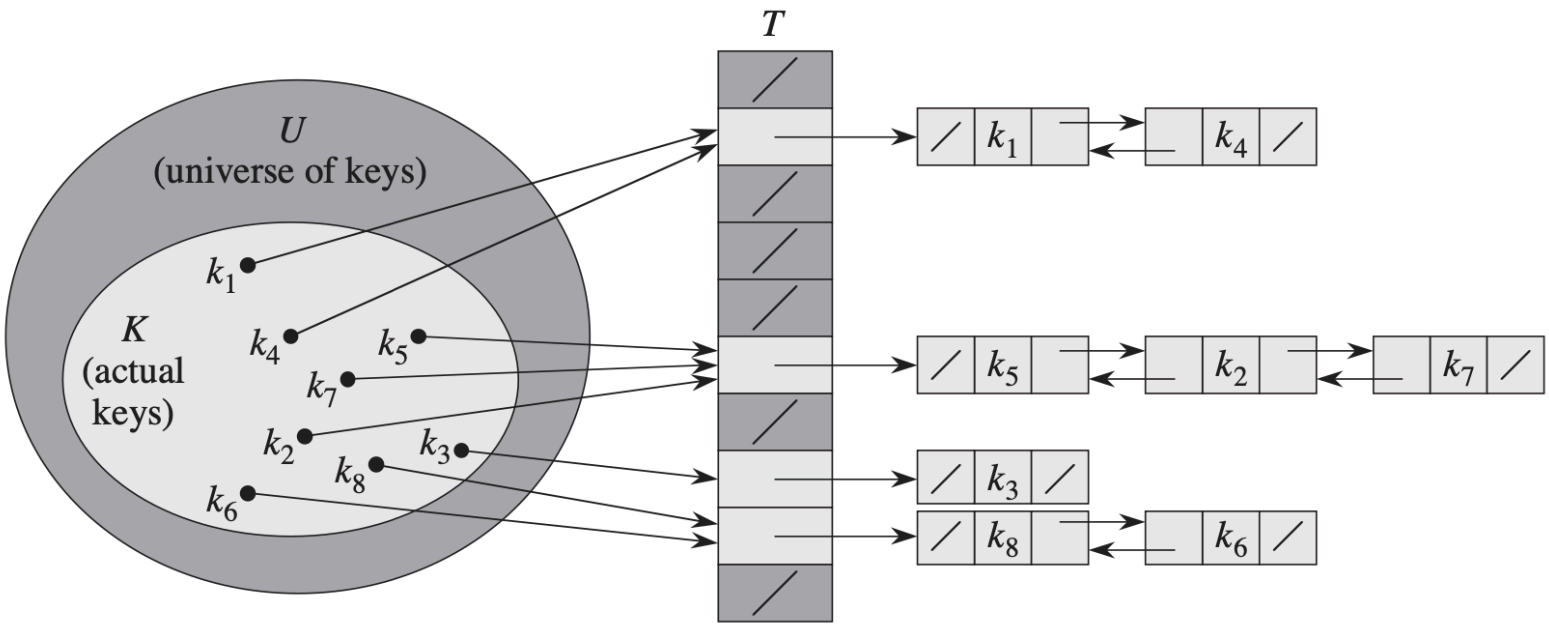

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

Dictionary operations on a hash table T

CHAINED-HASH-INSERT

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE

1 delete x from the list $T[h(x.key)]$

Runtime

- The worst-case running time for insertion is $O(1)$.
- The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table; if necessary, we can check this assumption (at additional cost) by searching for an element whose key is $x.key$ before we insert.
- For searching, the worst-case running time is proportional to the length of the list; we shall analyze this operation more closely below.
- We can delete an element in $O(1)$ time if the lists are doubly linked, as Figure 11.3 depicts. This cannot be done in $O(1)$ if the list is singly linked. **Why?**

Analysis of hashing with chaining

Analysis of hashing with chaining

- Given a hash table T with m slots that stores n elements, we define the ***load factor*** α for T as n/m, that is, the average number of elements stored in a chain.
- Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.
- The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Analysis continued

- We shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this assumption of ***simple uniform hashing***.

For $j = 0, 1, ..., m-1$, let us denote the length of the list $T[j]$ by n_j, so that

$$n = n_0 + n_1 + ... + n_{m-1} \tag{1}$$

and the expected value of n_j is $E[n_j] = \alpha = n/m$.

Runtime analysis

- We assume that $O(1)$ time suffices to compute the hash value $h(k)$, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$.
- Setting aside the $O(1)$ time required to compute the hash function and to access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm, that is, the number of elements in the list $T[h(k)]$ that the algorithm checks to see whether any have a key equal to k. In the second, the search successfully finds an element with key k.

Unsuccessful Search

Theorem 11.1 In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k) = O(1)$ is $\Theta(1+\alpha)$.

Successful Search

Theorem 11.2 In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof:

Let x_i denote the ith element inserted into the table, for $i = 1, 2, ..., n$, and let $k_i = x_i.key$. For keys k_i and k_j, we define the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if the keys of } x_i \text{ and } x_j \text{ hash to the same table slot} \\ 0 & \text{otherwise} \end{cases}$$

Proof Continued for successful search

Then if we want to count the number of keys that hash to the same list as element i *after* i is added to the list, that is just

$$\sum_{j=i+1}^{n} X_{ij}.$$

We are starting the sum at $i+1$ because we only want to count the elements that are inserted at i.

Proof continued

Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = 1/m$, so $E[X_{ij}] = 1/m$. So the expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \text{ by linearity of expectation}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

Proof contd

$$
\begin{aligned}
&= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) \\
&= 1 + \frac{n-1}{2m} \\
&= 1 + \frac{n}{2m} - \frac{1}{2m} \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
\end{aligned}
$$

Thus, the total time required for a successful search (including the time for computing the hash function) is
$\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$.

conclusion of the analysis

- If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average.
- Since insertion takes $O(1)$ worst-case time, and deletion takes $O(1)$ worst-case time when the lists are doubly linked, we can support all dictionary operations in $O(1)$ time on average.

Hash Functions

- A good hash functions satisfies (approximately) the assumptions of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
- Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which keys are drawn. Moreover, the keys might not be drawn independently.
- But we can frequently get good results by attempting to derive the hash value in a way that we expect to be independent of any patterns that might exist in the data.

Division method

- Let $h(k) = k \bmod m$ for some value of m.
- For instance, if $m = 12$, and $k = 100$, then $h(k) = 4$.
- When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k, and generally not all low-order bit patterns are equally likely.

Multiplication method

1. Multiply the key k by some number $0 < A < 1$.
2. Extract the fractional part of kA.
3. Multiply it by m.
4. Take the floor of the results.

In other words, let $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where kA **mod** $1 =$ fractional part of kA.

Diagram

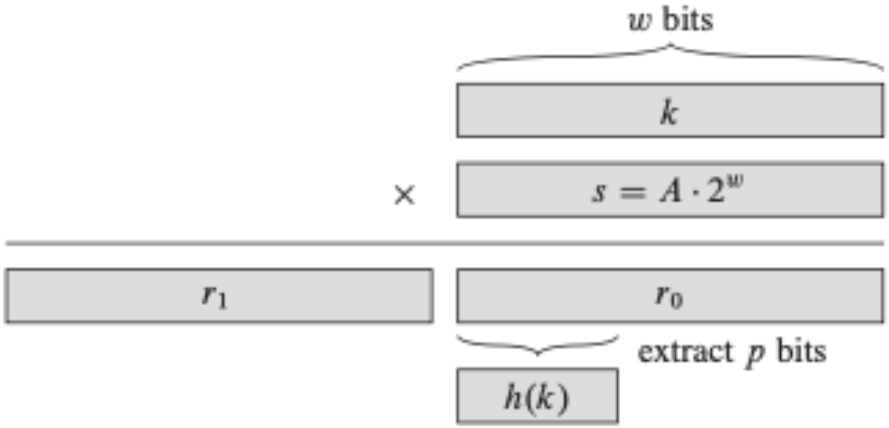

Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.