Hashing, Hash Tables, Hash Functions

CS313E - Elements of Software Design

Kia Teymourian

05/11/2022

Agenda

1. Hash Tables
2. Hash Function
3. Open Addressing
4. Hash Collisions

Motivation for Hash Tables.

- There are many applications in computer science that we need to organize data based on a (key, value) format. We have a key in hand and we want to insert values based on specific keys.
- For example, Dictionaries are an Abstract Data Type (ADT) that is implemented in many programing languages like python.
- A dictionary maintains a set of values that each of the values are associated with a key. We can store items in dicts, search for it fast and delete item.

There are many applications for it, e,g. Databases, data management systems, search engines, etc.

Hashing

Hashing is the most used and common data structure/concept in Computer Science. Remember:

- Best search algorithm can do $O(lg(n))$
- Best sort algorithm can do $O(nlg(n))$
- This lecture is about how to search faster than $O(lg(n))$ times and how we can get it down to $O(1)$ with certain probabilities.

Dictionaries

We have the following operations:

- **insert(item):** add item to set (overwrite it if exist)
- **delete(item):** remove item from set
- **search(key):** return item with key if it exists (We can not ask for larger or smaller)

The assumption is to have items with distinct keys or if we store new (key, value) with the same key we overwrite the old (key, value), if they have the same key.

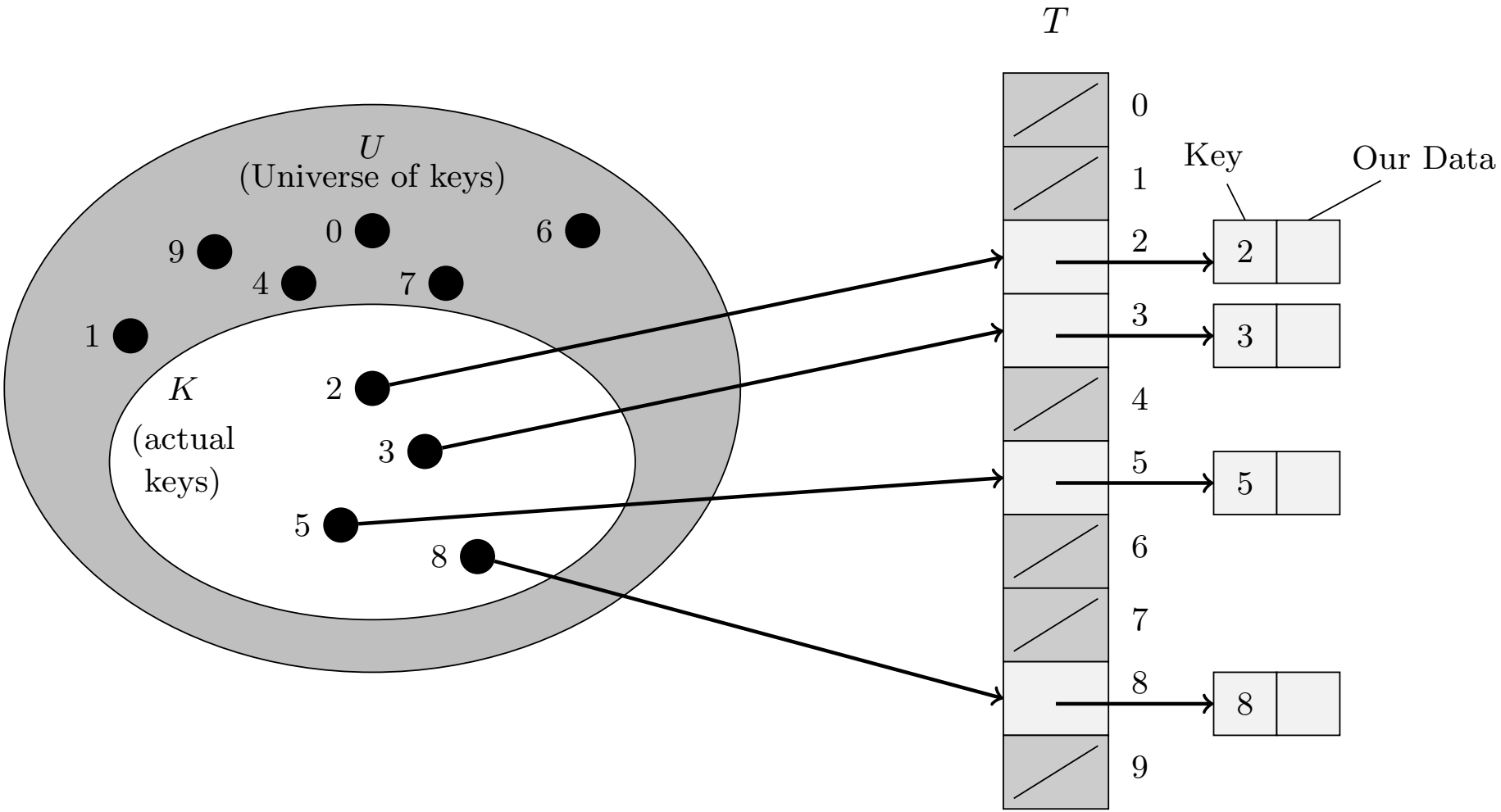
- We have a set of such (key, value) elements
- Hash Table can help us search in a dictionary very fast and under certain assumptions, the average search time for an element in a hash table is $O(1)$.

If we would use a search algorithm like Binary Search Trees for searching in a dictionary the best running time would be $lg(n)$. By using a hash table we can find the keys much faster than $O(1)$ in average with high probability.

Hash Table with Direct Access Table

- A very simple approach to implement dictionary is to store it in an array and use the index array as the keys for the values we want to store.
- Our values (items) are then stored in an array and they are indexed by key that we select for example randomly (random access).

Hash Table with Direct Access Table



Direct Access Table

Here we face a bunch of problems:

- An array has positive indexes, so our keys can not be negative
- More important problem is that we would need to have a very large array to store our data items. Lots of it would be waste of memory because we do not use it. It requires a gigantic memory hog that is mostly not used but allocated and reserved.

They are nice solutions to the above problems.
To solve the problem of negative keys, we can use a function (a pre-hash function) to map negative keys to none-negative keys.

Hash Function

- With direct addressing, an element with key k is stored in a slot k .
- However, with hashing, an element with key k is stored in slot $h(k)$; We use a hash function h to compute the slot from the key k.

The function h maps the universe U of keys into the slots of a hash table $T[0, 1, \ldots, m-1]$

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

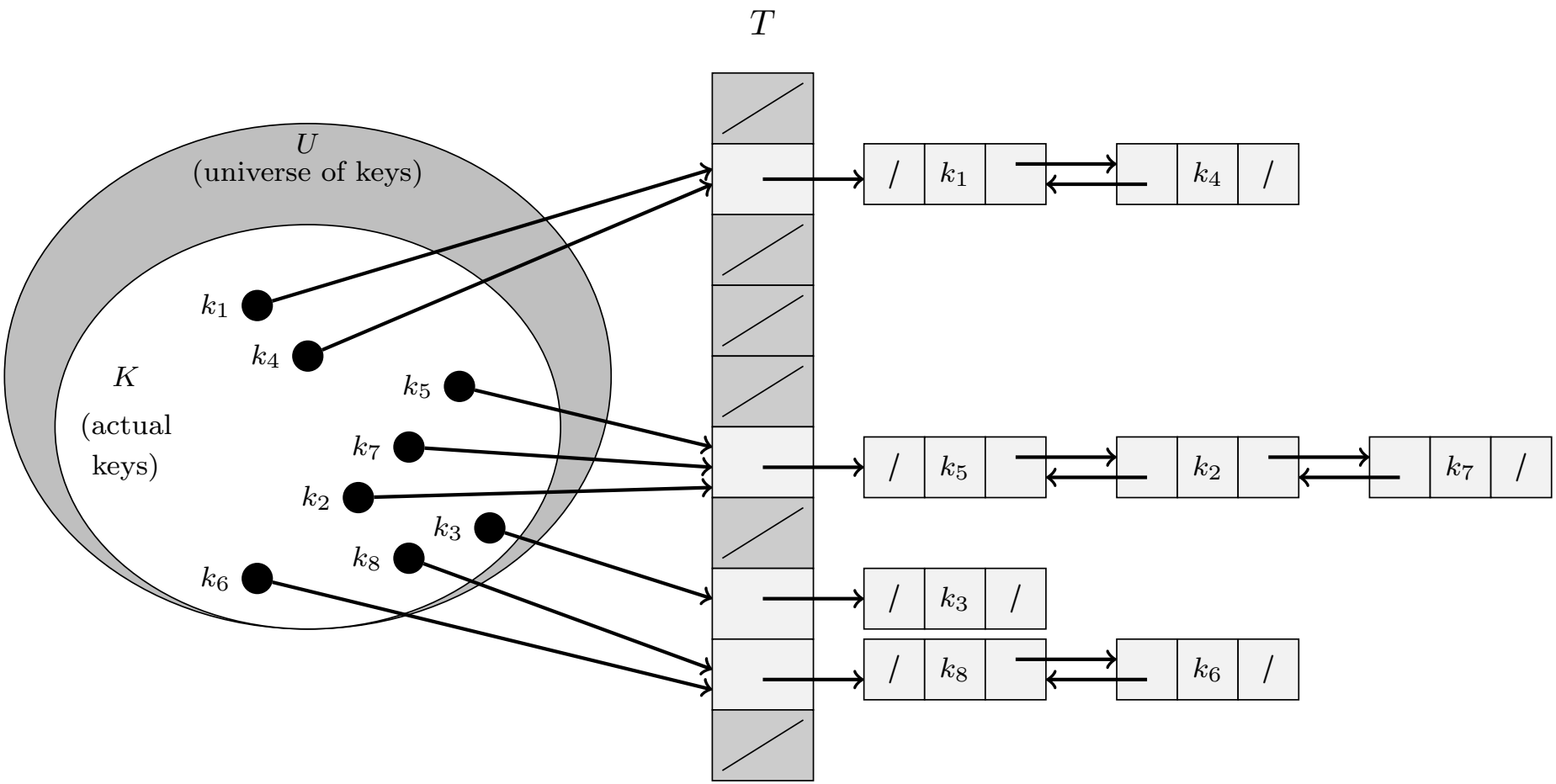
where m is the size of hash table which is mostly less than $|U|$

- An element with a key hashes to table slot place $h(k)$
- $h(k)$ is the hash value of key k.

How we can solve the collision problem?

- Chaining
- Open Addressing

Collision Resolution by Chaining



Simple Uniform Hashing

Under the assumption that each key is equally likely to be hashed to the table slots, independent of where other keys are hashed.

- n is the number of keys that we want to store in the table
- m is the number of slots in the table

Load Factor

For a given table T, the load factor $\boldsymbol{\alpha = n/m}$, the average number of elements stored in a chain.

Our analysis is based on the load factor $\alpha \leq 1$ or $\alpha \geq 1$

Load Factor can be interpreted as :

- Expected number of keys per slots
- Expected length of a chain

Simple Uniform Hashing

Search run time for a key using Simple Uniform Hashing is

$$\Theta(\alpha + 1)$$

- 1 is for the applying hash function and random access to the table memory slot.
- Cost α is for search the list

This would be then $O(1)$ if $\alpha = O(1)$ it means we have $m = \Omega(n)$
The running time for search can be $O(n)$

Hash Functions

We need to find good hash functions to be able to spread our data around the target table so that can avoid lots of collision.

Division Method

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

- In practice when we use the division method for hashing we try to find good prime numbers that are not close to power of 2 or power of 10 numbers.
- This approach in a computer system means that when we store all of these numbers in memory, then hashing would mix these numbers and at the end our hash values are just depending on the low bits or digits of these numbers to produce the final hash value.

Multiplication Method

$$h(k) = [(a - k) \ mod \ 2^m] >> (w - r)$$

- k is w bits of words (remember from the first lecture "Words")
- a is a random number and is odd $2^{w-1} < a < 2^w$ (select a not too close to the two values)
- r is a number that that words are shifted.
- " $>>$ " is the signed right bitwise shift operation by $(w - r)$

In practice this hashing does a good job and runs fast.
We can find keys that this function does not do a good jobs and we get more collisions.

Universal Hashing

$$h(k) = [(ak + b) \ mod \ p] \ mod \ m$$

- p is a very large prime number ($> |U|$) bigger than the size of universe
- a and $b \in \{0, 1, \ldots, p-1\}$ are random numbers

In a worst-case, it happens that we have two keys

$$k_1 \neq k_2 : Pr_{h_{a,b}}\{\text{event of } h(k_1) = h(k_2)\} = \frac{1}{m}$$

Universal Hashing

The event is X_{k_1,k_2} when we have $h(k_1) = h(k_2)$

$$
\begin{aligned}
E[\text{number of collision with key } k_1] &= E[\sum_{k_2} X_{k_1 k_2}] \\
&= \sum_{k2} E[X_{k_1 k_2}] \\
&= \sum_{k2} Pr\{X_{k_1 k_2} = 1\} \\
&= \frac{n}{m} \\
&= \alpha
\end{aligned}
$$

$Pr\{X_{k_1 k_2} = 1\} = 1/m$
In worst case there is no difference to the above other hash functions

Open addressing

We have described that we can get key collisions.
If the table size is small we get a lot more collisions and if the table is large we are wasting a lots of space.

The main idea of the open addressing is to grow the table as needed.

- We start with a small size of m rows of table.
- Then we are going to grow and shrink as necessary.

(Almost the same idea is applied in python to implement lists. You know that we can increase and decrease the python lists)

- In case of $n > m$ we want to increase the table size to be able to insert (key, value) pairs into it.
- If we make the new table size to be $m\prime$ which is larger that original hash table size m.
- Think about the case that we need to grow the table and move all of our existing data into it.

What would be the costs for doing that?

Insert Operation

Algorithm 1 Hash-Insert(T,k)

```
1: for i = 0 to m do
2:     j = h(k, i)  ▷ Get a key for this key based on function and i the current index
3:     if (T[j] == NULL) then                          ▷ Check if it is occupied or not
4:         T[j] = k                                                    ▷ Do the insert
5:         return j
6:     end if
7: end for
8: "Error: Hash Table Overflow!"
```

Open Addressing

With open addressing, we require that for every key k, the **probe sequence**

- $\langle h(k,0), h(k,1), \ldots, h(k,(m-1))\rangle$
- be a permutation of $\langle 0, 1, \ldots, m-1\rangle$ so that every hash-table position is eventually considered as a new slot to fill up.

If $(n > m)$ then we need to grow the table.
There are different strategies that we can do this.

Adding Linearly

One is that we do $m\prime = m + 1$, i.e., each time the table is full we just add one new row into it.

- $m\prime = m + 1$ and cost of n inserts would be then $\Theta(1 + 2 + 3 \cdots + n) = \Theta(n^2)$

Doubling

Another Idea is to double the size each time when we need to add more rows.
$m\prime = 2 \times m$
This approach is called **"Doubling"**

- $m\prime = 2 \times m$ and cost of n inserts would be then $\boldsymbol{\Theta(1 + 4 + 8 \cdots + n) = \Theta(n)}$
- The above is geometric series as we know.
- This operation takes linear time for some operation and most of the time we have constant time because we just add items to the table.

Amortize Analysis

An operation takes amortize or **"$T(n)$ amortized"** if it k operations takes $\leq k \times T(n)$ time.

- It is not an statement about a single operation but it says that if you do a bunch of k operations the total running time is at most $k \times T(n)$
- An example from economy amortized costs, is if you lease a car for a price of \$300 per month and you can say that it costs you per day \$10 in average.
- The average is taken over all operations. This is cost calculation that makes sense on data structures.
- In Table doubling the amortize running time for k inserts is $\Theta(k)$ time which means $\Theta(1)$ per amortize insert operations .

Problem with Deletes

If we do lots of inserts and deletes, the table might get again large that we do not need it.
We can also say that for k inserts or deletes it takes $O(k)$
Deletion Operation

- If $m = \frac{n}{2}$ becomes half the size then shrink to half the size $m/2$. Here the problem is that I spend linear time for every operation.
  Slow:
  $\mathbf{2^k < -- Delete/Insters -- > 2^k + 1}$
  and we get $\Theta(n)$ per operation
- One solution is that if we have $m = n/4$ (for example 4) then we shrink back to $m/2$. The reason is that we can save operation for the case that we need to increase again we can do it faster.

The amortize cost here is $\Theta(1)$ for insertions and deletions.

Hash Collisions - Linear Probing

Each time you have a hash collision, just add your item to the next available slot.

Double Hashing

- Double hashing offers one of the best methods available for open addressing.

Double hashing uses a hash function with two functions inside it:

$$h(k,i) = (h_1(k) + ih_2(k)) \ mod \ m$$

- The initial probe goes to position $T[h_1(k)]$
- Successive probe positions are offset from previous positions by the amount $h_2(k)$ **modulo** m.
- The probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.

Example of Double Hashing

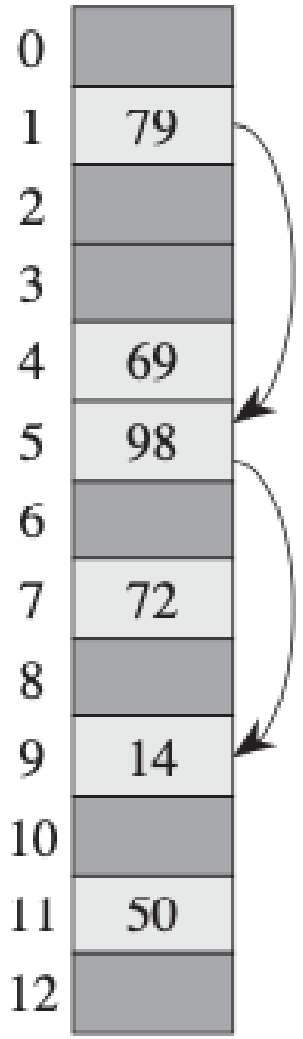
Insertion operation by double hashing

Assume that we have a hash table of size 13 with

$$h_1(k) = k \ mod \ 13$$
$$h_2(k) = 1 + (k \ mod \ 11)$$

Now, we want to insert with the key $k = 14$

$$h_1(14) = 14 \ mod \ 13 = 1$$
$$h_2(14) = 1 + (14 \ mod \ 11) = 1 + 3 = 4$$



Steps are:

1. First check the position 1, if not occupied insert there. It is alreadt occupied.
2. Then check the position 5, if not occupied insert there. It is already occupied.
3. Last check the position 9, if not occupied insert there. It is not occupied so we insert key = 14 into slot 9.

Hash Visualization

Online Visualization Tools:

- `https://visualgo.net/en/hashtable`
- `https://www.cs.usfca.edu/~galles/visualization/OpenHash.html`
- `https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html`

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

Chapter 11 Hash Tables

- Section 11.1 Direct-address tables
- Section 11.2 Hash tables
- Section 11.3 Hash functions
- Section 11.4 Open addressing