Hashing

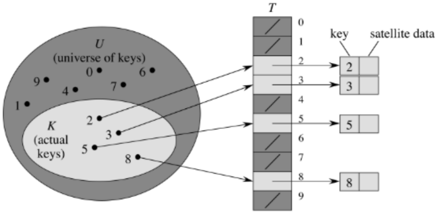
Hashing

We assume in hashing that keys are integers up to a maximum limit u. [To use hashing on other data types, elements must be assigned a unique integer value, cf. hashCode() in Java and hash() in Python.]

That means we have a universe of possible keys: U={0,1,2 , . . ,u −1}

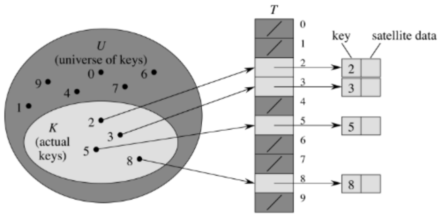
A dictionary stores a subset K⊆U, e.g.K={2,3,5,8}

The basic idea in hashing (just like in Counting Sort) is to use keys as indices in an array.



This takes O(1) time for Search, Insert and Delete.

Problem



Problem: This idea can easily generate space waste because the array size is u =|U|, while the number n=|K|of stored items is often much smaller.

Example: storing 5 CPR numbers. That is, keys of the type 180781-2345, which can be seen as integers in U={0,1,2, . . .10^{10}−1}. Here: u=10^{10} while n=5. So we use at least 10^{10} bytes (>8 Gb RAM) to store 5 CPR numbers.

Stored as 32 or 64 bit integers u=$2^{32} \approx 10^{10}$ or u = $2^{64} \approx 10^{20}$
That is, the same situation or much worse.

Hash functions

Attempt to solve the problem: find a function h, that maps from the large universe of keys U to a smaller one:

$$h: U \rightarrow \{0,1,2,\ldots m-1\}.$$

Here: m the desired array size. Often chosen m=O(n), then no more space is used for the array than for the elements themselves.

Such a function is called a hash function. An example of a hash function might be:

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

An example with m=41:

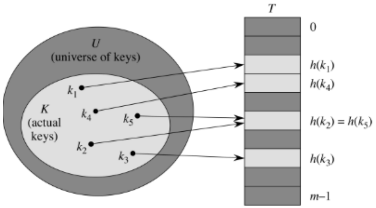
$$\begin{aligned}
h(12) &= 12 \bmod 41 = 12 && (\text{da } 0 \cdot 41 + 12 = 12)\\
h(100) &= 100 \bmod 41 = 18 && (\text{da } 2 \cdot 41 + 18 = 100)\\
h(479869) &= 479869 \bmod 41 = 5 && (\text{da } 11704 \cdot 41 + 5 = 479869)
\end{aligned}$$

Collisions

Hash functions solve the problem of space consumption. But they generate another problem: Two keys can hash to the same array index.

$h(479869) = 479869 \bmod 41 = 5$ (then 11704 . 41 + 5 = 479869)

$h(46) = 46 \bmod 41 = 5$ (then 1.41 + 5 = 46)



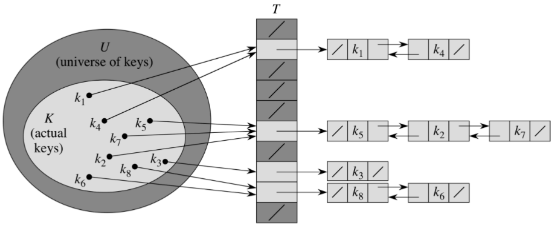
This is called a collision.

When h needs to map U into a smaller set {0,1,2, . . .m −1}, there will always be some keys k and k′ where h(k) =h(k′). So collisions cannot be avoided, and we must therefore find a solution.

Chaining

A simple solution: an array entry contains the start of a linked list with all the inserted elements whose key hashes to this array entry.

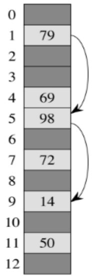
This is called chaining.



The cost is that the linked lists must be traversed during Search and Delete, causing the time for these operations to increase from Θ(1) to Θ(|list|). Insert is still O(1), since we can just insert at the beginning of the list.

Open addressing

Open addressing is an alternative to chaining: Try to find an empty slot in the array itself.



Linear probing (alias linear hashing):

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

Double hashing

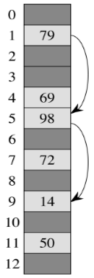
$h(k, i) = (h_1(k) + i.h_2(k)) \bmod m$

Here: h1(k) and h2(k) two hash functions (called “auxiliary” in the book).

Insert: in=0,1,2, . . .try until an empty slot is found

Search:in=0,1,2, . . .try until element or empty slot is found.

Open addressing, comments



Linear probing (alias linear hashing):

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

Double hashing

$h(k, i) = (h_1(k) + i.h_2(k)) \bmod m$

- Implementing **Delete** is more complicated. A simple solution: let deleted elements remain, mark them as deleted, and periodically clean up by rebuilding the hash table (re-insert the remaining elements from the beginning). For linear hashing, a more direct method exists, see Cormen et al. 4th edition, section 11.5.1(not syllabus).
- It is necessary that **n ≤ m** (since all **n** elements are in the array). Ideally, n≈m/4 to avoid running times degenerating.
- The examined indexes {h(k,0),h(k,1),h(k,2), . . . ,h(k, m −1)} should be {0,1,2, . . . ,m −1}for all k (so the entire array is searched).

Running time for hashing

We want a hash function h that spreads keys from the given input well over {0, 1, 2, ..., m − 1}, so that there are few collisions. Good hash functions are often thought of as those that map keys to indexes in a seemingly random manner.

However, for any given hash function, it is always possible to find at least |U|/m (i.e., u/m) keys that hash to the same array index. Often, u/m > n, which makes the worst-case time Θ(n).

In practice, we often just hope that this does not happen for our specific input and hash function choice. There are, in fact, methods to guarantee that this rarely happens.

In practice, one can expect that hashing supports Search, Insert, and Delete in O(1) time, unless one is very unlucky

Additionally, the worst-case time can be reduced to **O(log n)** by using **balanced search trees** instead of linked lists in **chaining**. This is what Java does when the linked list becomes large.