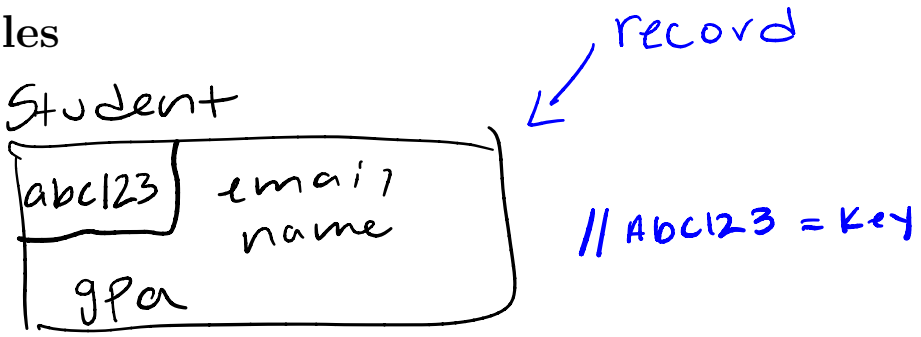
Hash Tables



Idea: given abc123, you can lookup & display all this information in O(1) expected time

Given a table T and a record x (the record has a key and some additional associated data) we need to support the following operations:

- $Q.insert(T, x)$ - inserts the record x into the table T
- $Q.delete(T, x)$ - deletes the record x from the table T
- $Q.search(T, k)$ - find the record with a key of k in the table T

We want these to be fast but don't care about sorting the records. What data structure should we use to store this table T?

A **hash table** supports all of the above operations in $O(1)$ expected time!

First, let's consider what kind of problem benefits from its use.

Example 1: Symbol tables

- A compiler uses a symbol table to relate symbols (which is our key) to its associated data
  - **Symbols**: variable names, function names, etc.
  - **Associated data**: memory location, scope, etc.
- The compiler uses this table to quickly look up information about symbols while it compiles our code.
- We care about search, insertion, and deletion.
- The symbols being in sorted order doesn't really matter.

Example 2: Parking Lot

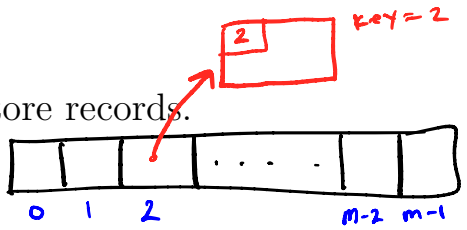
- Suppose are to managing a parking lot. We want to do the following quickly:
  - Park a car in the lot (insertion)
  - Tell someone where their car is parked if it is in the lot (search)
  - Remove a car parked in the lot (deletion)

Direct Addressing

Suppose you want to store a set of keys where:

- All of the keys are between 0 and $m-1$
- All of the keys are distinct

// store a new record



The idea: Set up an array $A[0 \ldots m-1]$ to store records.

- A[i] = x if x has key i and $x \in A$
- A[i] = $NULL$ if $x \notin A$
  - If no record we treat as null

// search:
- just go to index searching for

This is a direct address table. What are the run times of the following operations?

- insert: $\Theta(1)$ // jump to specific location
- delete: $\Theta(1)$ // replace w/ null to delete
- search: $\Theta(1)$ // jump to specific location

This seems really good! Are there any issues with this approach?

=> This uses large amount of space (memory)

=> trading off faster runtime, for creation of more space

// Imagine UTSA parking lot

Let's revisit the parking lot problem.

Our key is a license plate. How many parking spaces do we need to use direct addressing (assume 6 alphanumeric character license plates).

↳ a-z, 0-9 = 36 characters

↳ 36 36 36 36 36 36 = 36^6 ≈ 2 billion

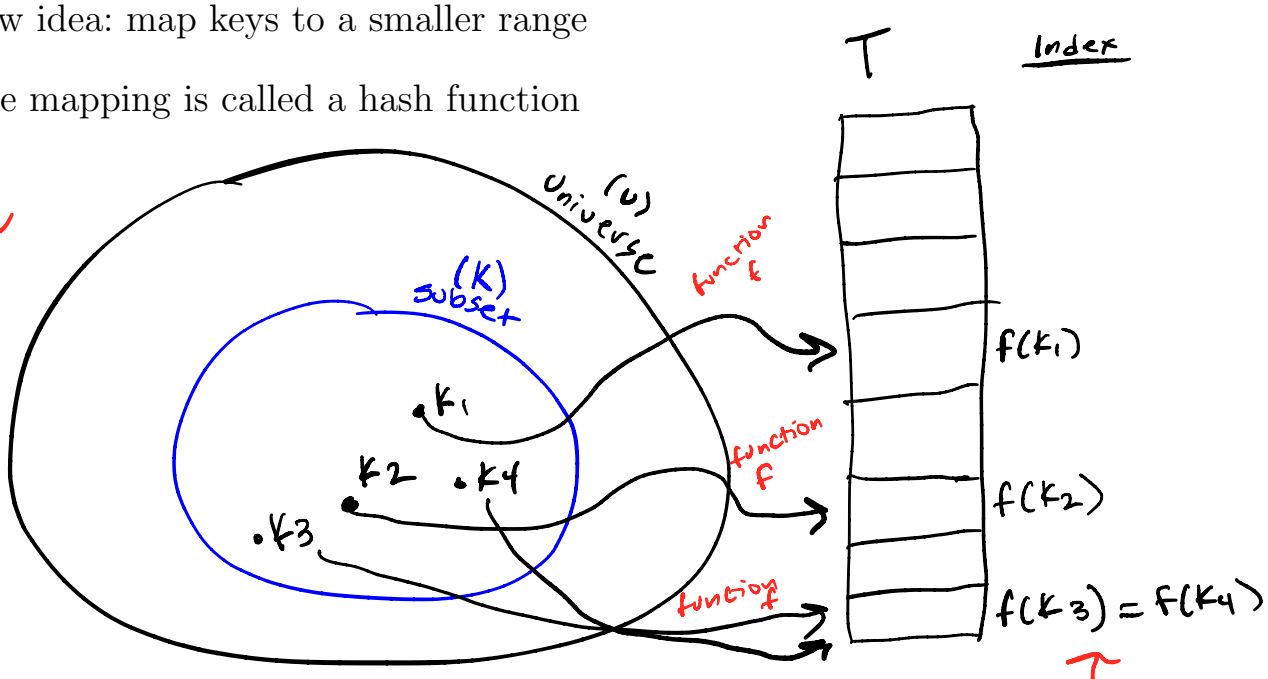
This is very large obviously

$|K| == |T|$; $|U| > |K|$

New idea: map keys to a smaller range

The mapping is called a hash function

// function f tells you where to put K_1 in T



What is the issue with this new idea?

- We could have 2 keys that want to map to the same location

I.E: K_3 & K_4 both map to same location

- This is a **Collision**

// Handling Collisions

Hash Functions

Let U be the universe of all possible keys.

A hash function h maps from U to one of the m slots of our hash table $T[0 \ldots m-1]$.

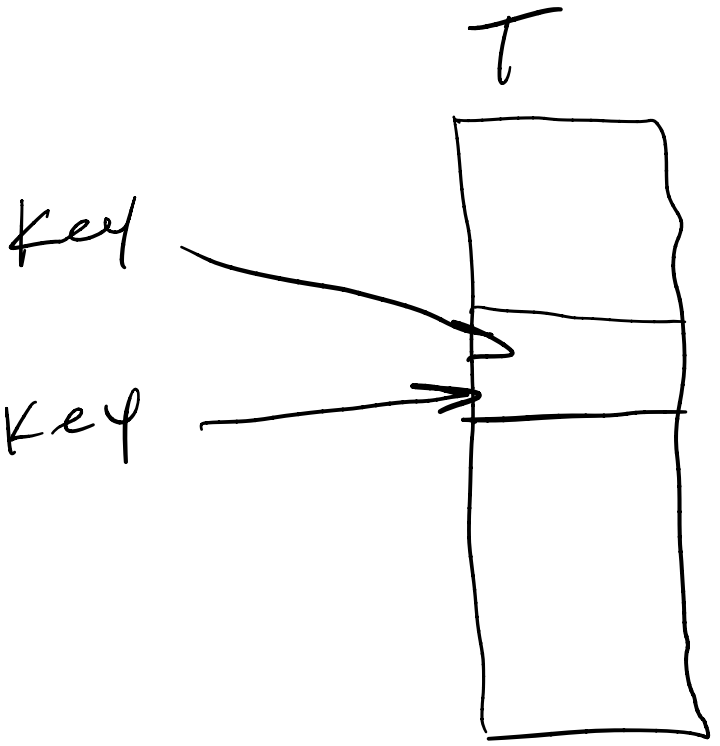
$|T| = m$ slots

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

↳ maps to = maps universe to the set

With direct addressing, key k maps to slot $T[k]$.

$h(k)$ is the hash value of the key k. With hash tables, key k maps (or "hashes") to slot $T[h(k)]$.



Resolving Collisions - Open Addressing (linear probing)

Key idea:

- To insert: if slot is full, keep trying different slots (following a systematic and consistent strategy) until an open slot is found (**probing**)
- To search, follow same sequence of probes as would be used when inserting the element
  - If we find the element with the correct key, return it
  - If we reach a NULL pointer, then the element is not in table

$h(x) = x \cdot 10$ ← hash function

$\Rightarrow h(18) = 18 \cdot 10 = 8$

$h(23) = 23 \cdot 10 = 3$ ✓

$h(25) = 25 \cdot 10 = 5$

⋮

$h(43) = 43 \cdot 10 = 3$ ✓

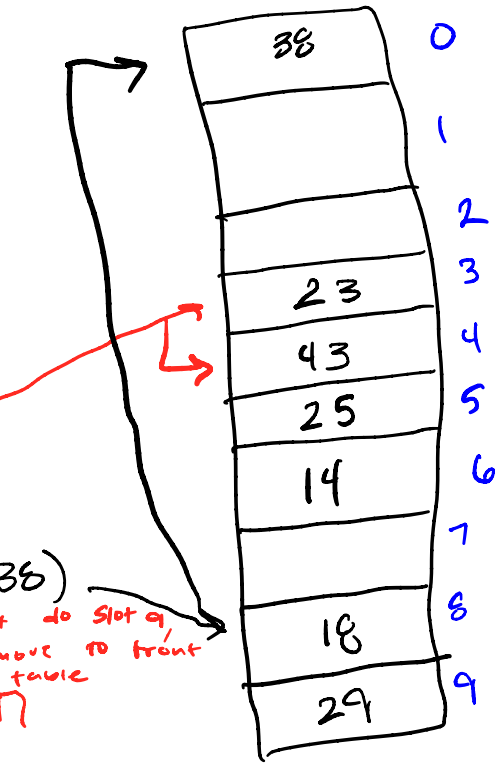
// slot 3 already full, so we put 43 in next closest empty position

$h(14) = 14 \cdot 10 = 4$

// slot 4 taken already, but slot 6 was next empty position so we filled slot 6 w/ 14

$h(38)$

// can't do slot 9, thus move to front of table



- Deleting can be expensive
- This method is sensitive to unevenly distributed hash values.

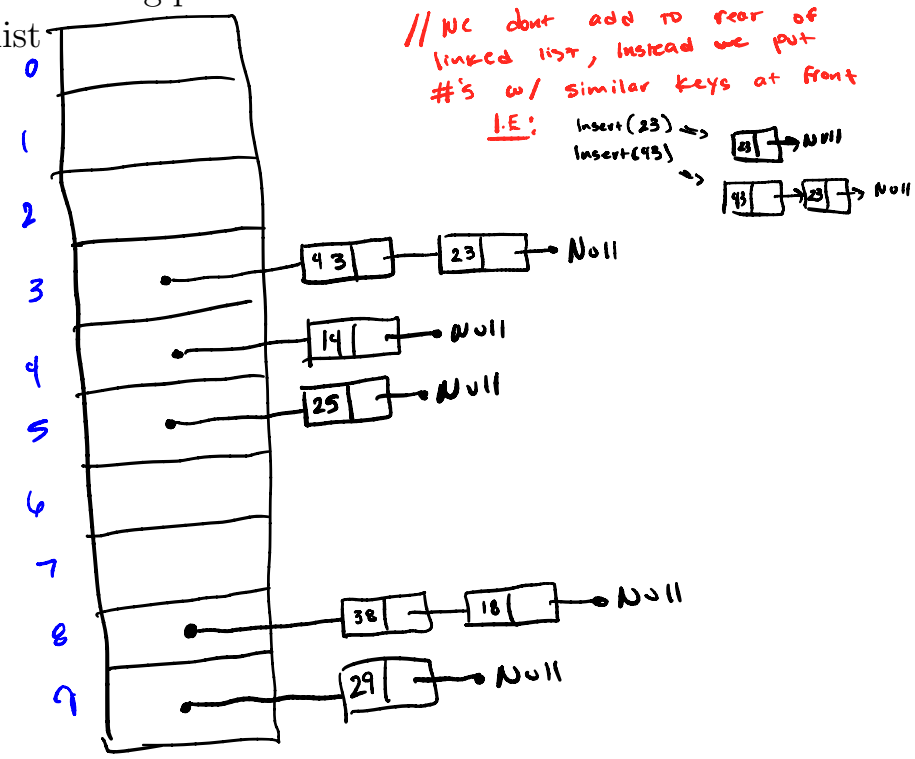
– Want an oversized table to alleviate clustering / collision

– Avoid delete

Insert
14, 23, 25, 43,
14, 29, 38

Resolving Collisions - Chaining

Key idea: Chaining puts elements that hash to the same slot into a linked list

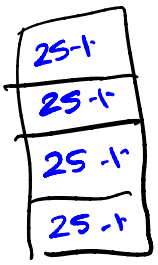


- $Insert(T, x)$ where x has key k
  - Insert x at the head of the list at $T[h(k)]$
  - Worst case complexity: O(1) // inserting at front is constant
- $Search(T, k)$
  - Search for the record x with the key k in our list at $T[h(k)]$
  - Worst case complexity: length of linked list at h(k) in T
- $Delete(T, x)$ = O(L)
  - Delete the record x with the key k from our list at $T[h(k)]$
  - (i.e., we're deleting a given element of a linked list)
  - Worst case complexity (singly linked list):
    - Find prior element
      - ↳ O(L)
    - search
      - ↳ O(L)
  - Worst case complexity (doubly linked list):

    O(1) to delete

    O(L) to search

α = how many keys expected per slot → All are equally likely



Analysis of Chaining

Suppose we have **simple uniform hashing** where each new key is equally likely to be hashed to any of the m slots in our hash table.

If we have n keys to enter into our table of m slots, how many keys do we expect to have per slot?

load factor $\underset{\text{alpha}}{\alpha} = \frac{n}{m}$ (average key per slot)

$\frac{0}{m} = \alpha = 0$ ↑ why its $(1+\alpha)$

So our average cost of an unsuccessful search for a key k will be (i.e, we don't find an element with key k):

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

What is our average cost of an successful search for a key k? (i.e, we find the element with key k): = search ½ of list

$\Rightarrow \Theta(1 + \frac{\alpha}{2}) = \Theta(1 + \alpha)$

(see CLRS page 259-260 for full analysis)

So the cost of searching $= O(1 + \alpha)$

If the number of keys n is proportional to the number of slots in the table, what is α? $\Rightarrow n \in \Theta(m)$

$\Rightarrow \alpha = \frac{n}{m} \in \Theta(1)$

Differ by constant

Thus, we can make the expected cost of searching constant if we make α constant.

=> How do we know if we have simple uniform hashing?