Hash table

A basic problem

- We have to store some records and perform the following:
  - add new record
  - delete record
  - search a record by key
- Find a way to do these efficiently!

Unsorted array

- Use an array to store the records, in unsorted order
  - add - add the records as the last entry fast *O(1)*
  - delete a target - slow at finding the target, fast at filling the hole (just take the last entry) *O(n)*
  - search - sequential search slow *O(n)*

Sorted array

- Use an array to store the records, keeping them in sorted order
  - add - insert the record in proper position. much record movement slow *O(n)*
  - delete a target - how to handle the hole after deletion? Much record movement slow *O(n)*
  - search - binary search fast *O(log n)*

Linked list

- Store the records in a linked list (unsorted)
  - add - fast if one can insert node anywhere *O(1)*
  - delete a target - fast at disposing the node, but slow at finding the target *O(n)*
  - search - sequential search slow *O(n)*
    (if we only use linked list, we cannot use binary search even if the list is sorted.)

More approaches

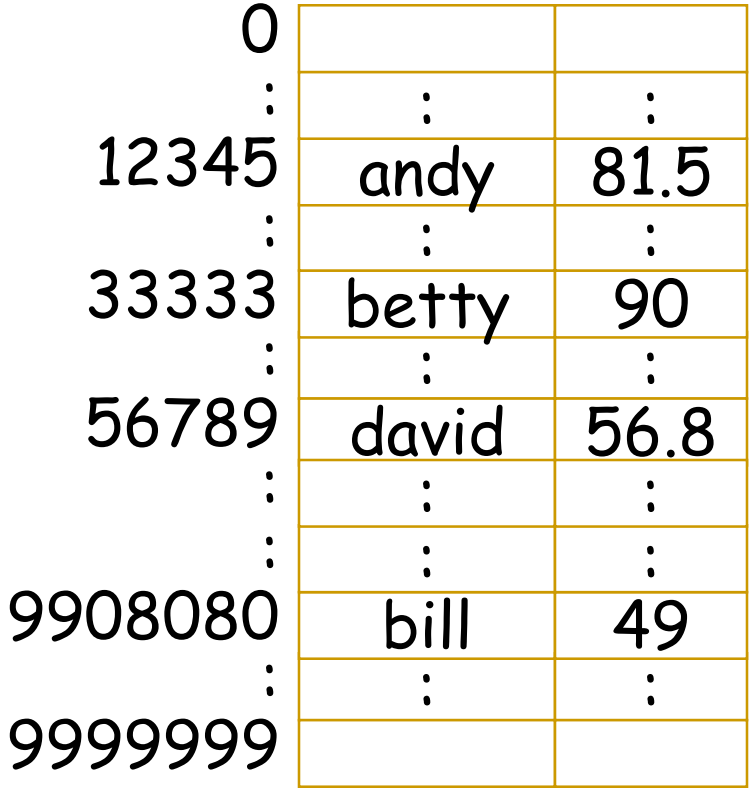
- have better performance but are more complex
  - Hash table
  - Tree (BST, Heap, …)

Array as table

0012345	andy	81.5
0033333	betty	90
0056789	david	56.8
...		
9801010	peter	20
9802020	mary	100
...		
9903030	tom	73
9908080	bill	49

Consider this problem. We want to store 1,000 student records and search them by student id.

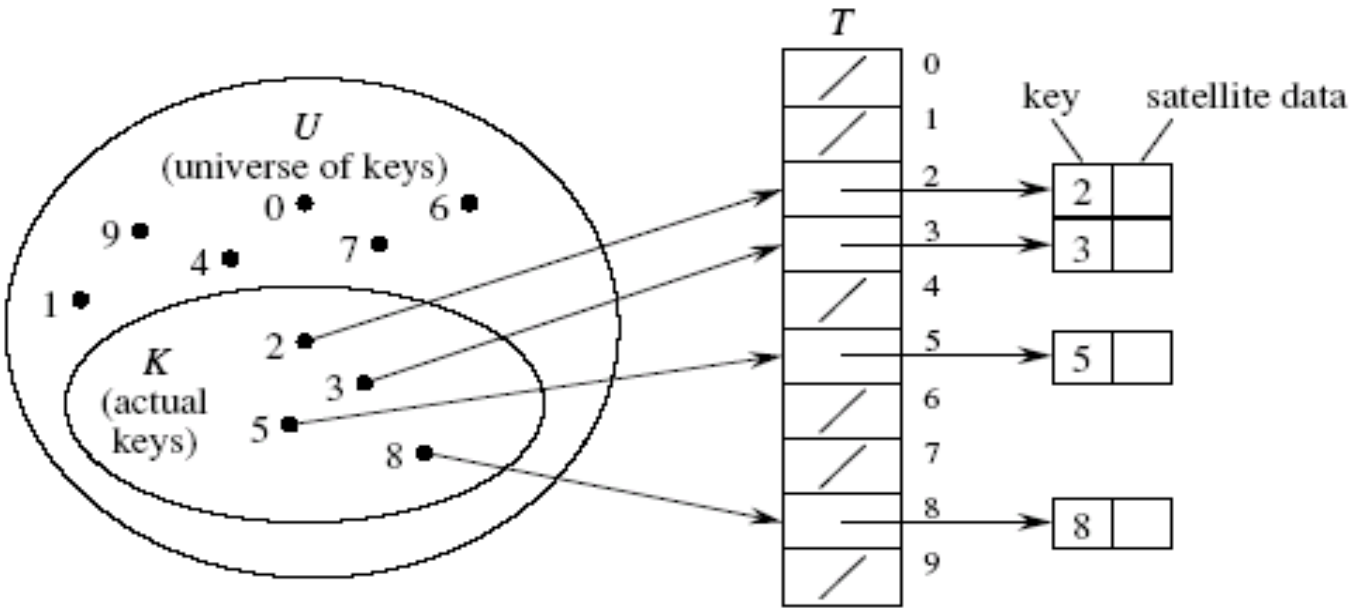
Array as table



One ‘stupid’ way is to store the records in a huge array (index 0..9999999). The index is used as the student id, i.e. the record of the student with studid 0012345 is stored at A[12345]

Array as table

- It is also called Direct-address Hash Table.
  - Each ***slot***, or position, corresponds to a key in U.
  - If there's an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL.



Array as table

- Store the records in a huge array where the index corresponds to the key
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*
- But it wastes a lot of memory! Not feasible.

Hash function

```
function Hash(key: KeyType): integer;
```

Imagine that we have such a magic function Hash. It maps the key (studid) of the 1000 records into the integers 0..999, one to one. No two different keys maps to the same number.

H('0012345') = 134
H('0033333') = 67
H('0056789') = 764
…
H('9908080') = 3

Hash Table

To store a record, we compute Hash(stud_id) for the record and store it at the location Hash(stud_id) of the array. To search for a student, we only need to peek at the location Hash(target stud_id).

0			
	:	:	:
3	9908080	bill	49
	:	:	:
67	0033333	betty	90
	:	:	:
134	0012345	andy	81.5
	:	:	:
764	0056789	david	56.8
	:	:	:
999	:	:	:

Hash Table with Perfect Hash

- Such magic function is called perfect hash
  - add - very fast *O(1)*
  - delete - very fast *O(1)*
  - search - very fast *O(1)*
- But it is generally difficult to design perfect hash. (e.g. when the potential key space is large)

Hash function

- A hash function maps a key to an index within in a range
- Desirable properties:
  - simple and quick to calculate
  - even distribution, avoid collision as much as possible

```
function Hash(key: KeyType);
```

Division Method

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

- Certain values of m may not be good:
  - When $m = 2^p$ then $h(k)$ is the p lower-order bits of the key
  - Good values for m are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then m=701 (m = *hash table length*) yields a hash function:

$$h(key) = k \bmod 701$$

Collision

- For most cases, we cannot avoid collision
- Collision resolution - how to handle when two different keys map to the same index

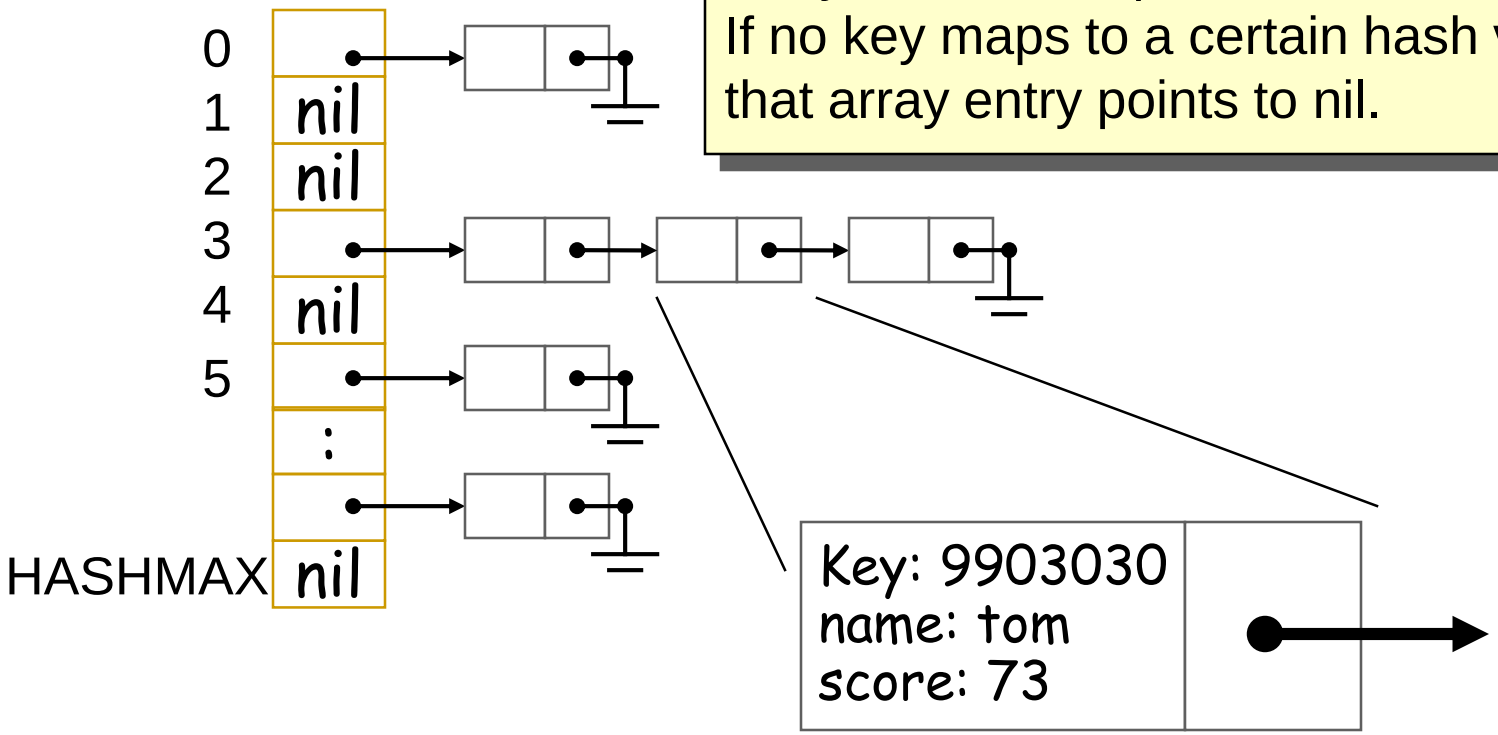
H(‘0012345’) = 134
H(‘0033333’) = 67
H(‘0056789’) = 764
…
H(‘9903030’) = **3**
H(‘9908080’) = **3**

Solutions to Collision

- The problem arises because we have two keys that hash in the same array entry, a **collision**. There are two ways to resolve collision:
  - **Hashing with Chaining:** every hash table entry contains a pointer to a linked list of keys that hash in the same entry
  - **Hashing with Open Addressing:** every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key

Chained Hash Table

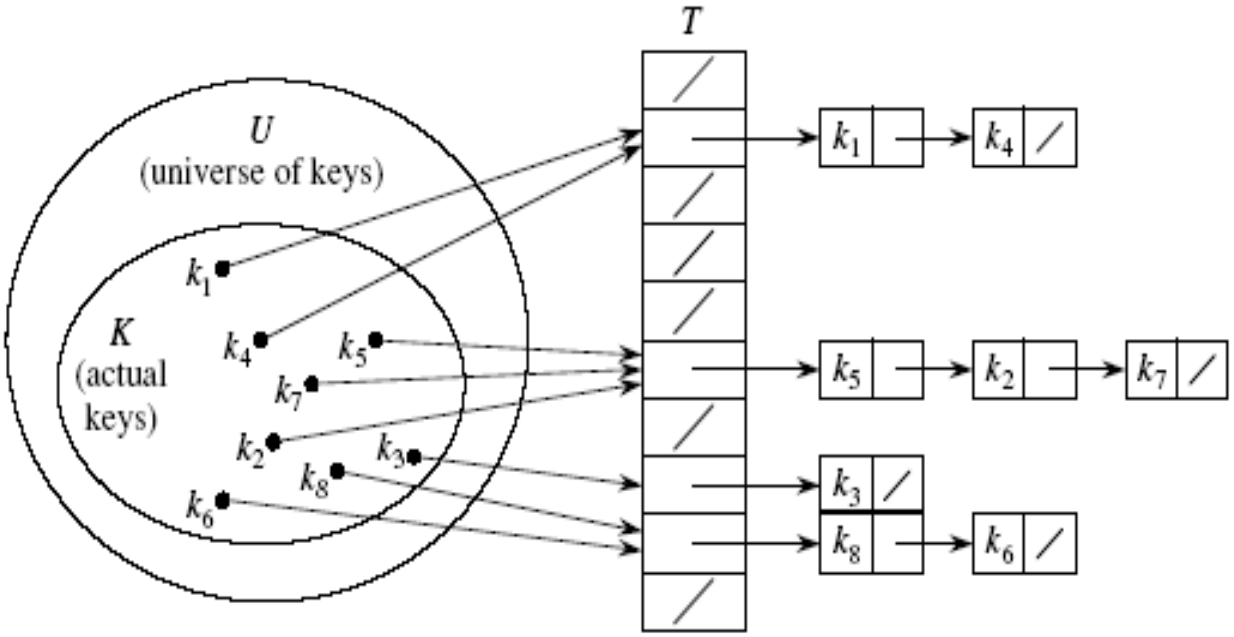
One way to handle collision is to store the collided records in a linked list. The array now stores pointers to such lists. If no key maps to a certain hash value, that array entry points to nil.



Chained Hash Table

Put all elements that hash to the same slot into a linked list.

- 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.



Chained Hash table

- Hash table, where collided records are stored in linked list
  - good hash function, appropriate hash size
    - Few collisions. Add, delete, search very fast *O(1)*
  - otherwise…
    - some hash value has a long list of collided records..
    - add - just insert at the head fast *O(1)*
    - delete a target - delete from unsorted linked list slow
    - search - sequential search slow *O(n)*

Open Addressing

An alternative to chaining for handling collisions.

- • Store all keys in the hash table itself.
- • Each slot contains either a key or NIL.
- • To search for key *k*:
  - Compute *h(k)* and examine slot *h(k)*.

    Examining a slot is known as a ***probe***.
  - If slot *h(k)* contains key *k*, the search is successful.

    If this slot contains NIL, the search is unsuccessful.
  - There's a third possibility: slot *h(k)* contains a key that is not *k*.

    We compute the index of some other slot, based on *k* and on which probe (count from 0: 0th, 1st, 2nd, etc.) we're on. Keep probing until we either find key *k* (successful search) or we find a slot holding NIL (unsuccessful search).

How to compute probe sequences

- ***Linear probing:*** Given auxiliary hash function *h*, the probe sequence starts at slot *h(k)* and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0. Given key *k* and probe number *i* ($0 \le i < m$),

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

- ***Quadratic probing:*** As in linear probing, the probe sequence starts at *h(k)*. Unlike linear probing, it examines cells 1,4,9, and so on, away from the original probe point:

  $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

  (if c1=0, c2=1, it's the example given by book)

- ***Double hashing:*** Use two auxiliary hash functions, h_1 and h_2. h_1 gives the initial probe, and h_2 gives the remaining probes:

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

Open Addressing Example

- Hash(89, 10) = 9
- Hash(18, 10) = 8
- Hash(49, 10) = 9
- Hash(58, 10) = 8
- Hash(9, 10) = 9

Linear Probing: $h(k, i) = (h(k) + i) \bmod m$.

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
- In following example, table size m = 8, and

k:	A,P,Q	B,O,R	C,N,S	D,M,T	E,L,U	F,K,N	G,J,W,Z	H,I,X,Y
h(k):	0	1	2	3	4	5	6	7

Action	0	1	2	3	4	5	6	7	# *probes*
Store A	**A**								1
Store C	A		**C**						1
Store D	A		C	**D**					1
Store G	A		C	D			**G**		1
Store P	A	**P**	C	D			G		2
Store Q	A	P	C	D	**Q**		G		5
Delete P	A	±	C	D	Q		G		2
Delete Q	A	±	C	D	±		G		5
Store B	A	**B**	C	D	±		G		1
Store R	A	B	C	D	**R**		G		4
Store Q	A	B	C	D	R	**Q**	G		6

Choosing a Hash Function

- Notice that the insertion of Q required several probes (5). This was caused by A and P mapping to slot 0 which is beside the C and D keys.
- The performance of the hash table depends on having a hash function which evenly distributes the keys.
- Choosing a good hash function is a black art.

Clustering

- Even with a good hash function, linear probing has its problems:
  - The position of the initial mapping i_0 of key k is called the **home position** of k.
  - When several insertions map to the same home position, they end up placed contiguously in the table. This collection of keys with the same home position is called a **cluster**.
  - As clusters grow, the probability that a key will map to the middle of a cluster increases, increasing the rate of the cluster's growth. This tendency of linear probing to place items together is known as **primary clustering**.
  - As these clusters grow, they merge with other clusters forming even bigger clusters which grow even faster.

Quadratic Probing Example

- Hash(89, 10) = 9
- Hash(18, 10) = 8
- Hash(49, 10) = 9
- Hash(58, 10) = 8
- Hash(9, 10) = 9

Quadratic Probing: $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$

- Quadratic probing eliminates the primary clustering problem of linear probing by examining certain cells away from the original probe point. In the following example, table size m = 8, and c1 = 0 , c2 = 1

k: A,P,Q B,O,R C,N,S D,M,T E,L,U F,K,N G,J,W,Z H,I,X,Y

h(k): 0 1 2 3 4 5 6 7

Action	0	1	2	3	4	5	6	7	# *probes*
Store A	**A**								1
Store C	A		**C**						1
Store D	A		C	**D**					1
Store G	A		C	D			**G**		1
Store P	A	**P**	C	D			G		2
Store Q	A	P	C	D	**Q**		G		3(5)
Delete P	A	±	C	D	Q		G		2
Delete Q	A	±	C	D	±		G		3(5)
Store B	A	**B**	C	D	±		G		1
Store R	A	B	C	D		**R**	G		3(4)
Store Q	A	B	C	D	**Q**		G		3(6)

Double Hashing

- ***Double hashing:*** Use two auxiliary hash functions, h_1 and h_2. h_1 gives the initial probe, and h_2 gives the remaining probes:

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

- Quadratic probing solves the primary clustering problem, but it has the secondary clustering problem, in which, elements that hash to the same position probe the same alternative cells. Secondary clustering is a minor theoretical blemish.

- Double hashing is a hashing technique that does not suffer from secondary clustering. A second hash function is used to drive the collision resolution.