Hash Tables

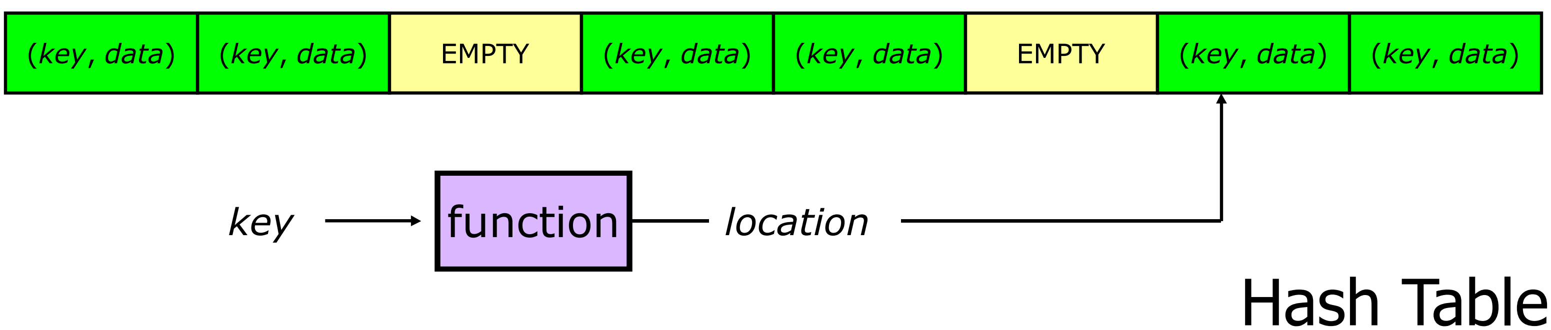
Consider data storage in unsorted array (suppose each data element has a key for identification)

- Insert is fast
  - constant time
- Delete-by-position is fast **if** we allow **gaps** in our data.
  - To delete an item, just set its value to “empty” (constant time)
- But, to do a delete-by-value we need first to find (search) the proper item, and unfortunately …
  - Search is **slow** ($O(N)$, in fact)

(*key*, *data*)	(*key*, *data*)	EMPTY	(*key*, *data*)	(*key*, *data*)	EMPTY	(*key*, *data*)	(*key*, *data*)

Hash Tables

- What if we had a magic function which, given an item's key, returned the location the item would be in **if** it were present?
  - Then all three operations (insert, delete, search) would be constant time!
- This is the basic idea behind a "Hash Table".



Hash Functions & Hash Tables

- A **Hash Function** is a function that “wants to be” our perfect location generator (magic function from the last slide)
- Hash function maps **keys** to integers (which represent indices into the hash table). Mathematically:

  Hash(key) = Integer

- A **Hash Table** is a data structure in which items are stored according to the location specified by a hash function.
  - Typically the hash table is an array of fixed size
  - Search performed using some part of the data– called the key.
  - Deleting is accomplished by marking a location as “empty”.
  - Thus, ideally, search, insert, and delete are all constant-time operations

Hash Collisions

The problem with hashing is this:

- Allocating an array large enough for every possible key is inefficient (and often impossible).
- Therefore, we use a small-ish array (or similar structure).
- With this approach, however, items with different keys may map to the same hash value.
- This is called a **collision**.

In general:

- Collisions are not a big problem as long as there are not very many of them.
- But there can conceivably be lots of them.
- So we need good hash functions that avoid collisions as much as possible.

Properties of a Good Hash Function

- Can be computed quickly.
- Spreads out its results evenly over the possible output values (positions in the table).
- Turns patterns in its input into random-looking output.
  - For example, consecutive keys should generally not map to consecutive output values.
- Let us look at some example hash functions; are they good enough?

Example Hash Functions

- Notations:
  - `K`: the key, an integer
  - `M`: size of ***hash table***

Example 1…

- What if we use this hash function
  - `Hash(K) == K`
- What is wrong?
  - What if K varies over a large range?

Example 2

- If
  - `Hash(K) == K % M`
- What is wrong?
- Suppose
  - `M = 10`
  - `K = 2, 20, 34, 42,76`
- Then `K % M = 2, 0, 4, 2, 6,...`
  - Since `10` is even, all even `K` are hashed to even numbers ... Greater chance of collisions (particularly for even values of K)

An Improvement

- If
  - **`Hash(K) = K % P`**, with `P` a prime number (>= M)
- Suppose
  - `P = 11`
  - `K = 10, 20, 30, 40`
- `K % P = 10, 9, 8, 7`
- More uniform distribution…

Collision Resolution

Two kinds of methods:

- Open Addressing
  - The Hash Table is essentially an array where each location can store a single data item.
  - If we get a collision, we look for another location (how? Later…)
- Buckets
  - Each location in the Hash Table is capable of storing multiple data items.
  - In this case, a location in the Hash Table is called a **bucket**.

Collision Resolution with Open Addressing

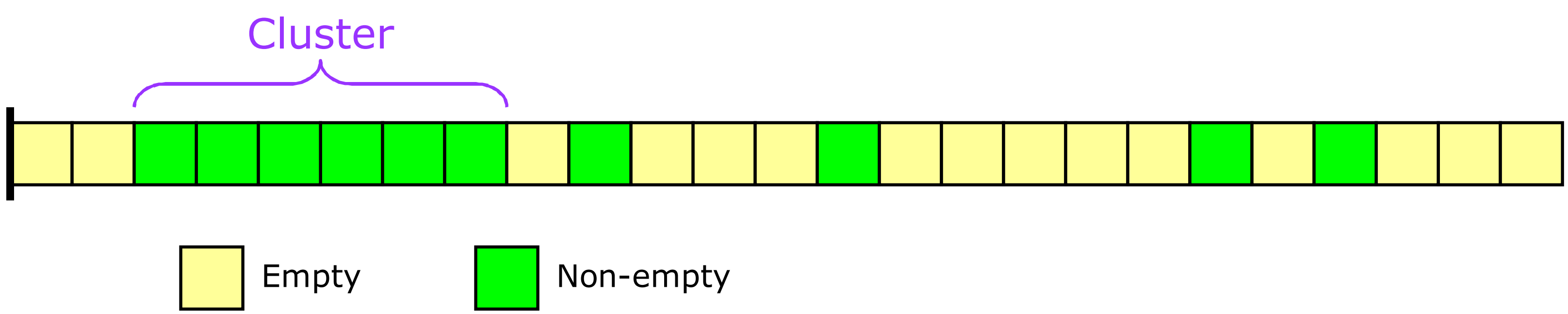
- The Hash Table is an array where each location stores a single data item.
- Each location can be marked as "empty".
- When inserting or searching, we look at a sequence of locations, as follows:
  - The first is the location given by the hash function.
  - We continue looking until we find the given key or we are sure it is not present.
  - Each time we view a location, we say we are doing a **probe**. The entire sequence of locations to view is called the **probe sequence**.

Collision Resolution with Open Addressing

- Probing Hash Tables
  - If collision occurs, try another cell in the hash table.
  - More formally, try locations $h_0(x)$, $h_1(x)$, $h_2(x)$, $h_3(x)$… in succession until a free location is found.
    - $h_i(x) = (hash(x) + f(i))$
    - $f(0) = 0$
    - hash(x) is the original index generated by the hash function. f(i)is the offset in hash(x) after i^{th} collision
  - Typical probing strategies
    - Linear probing
    - Quadratic probing

Linear Probing

- We look at location *t*, then *t*+1, then *t*+2, etc. i.e. f(i)=i
- Linear probing tends to form **clusters**, which slow things down.



Pseudocode: Insertion with Linear Probing

```
Insert(key)  // assume unique keys
  1. index =  key % table_size;
  2.  if (table[index]== 'EMPTY')
          table[index]= x;

  3. Else  {
        index++;
        index = index % table_size;
        goto 2;
     }
```

Pseudocode: Search with Linear Probing

```
Search (key)
   1.  Index = key % table_size;
   2.  If (table[index]=='EMPTY')
            return -1; // key not found
   3.  Else if (table[index] == key)
            return index;
   4.  Else {
         Index ++;
         index = index % table_size;
         goto 2;
            }
```

Linear Probing Example

Insert 89, 18, 49, 58, 69 using Hash(K)=K%10 with linear probe

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

(notice the clusters)

Quadratic Probing

- Probe sequence t, $t+1^2$, $t+2^2$, $t+3^2$, etc. i.e. $f(i)=i^2$
- Avoids clusters

Quadratic Probing Example

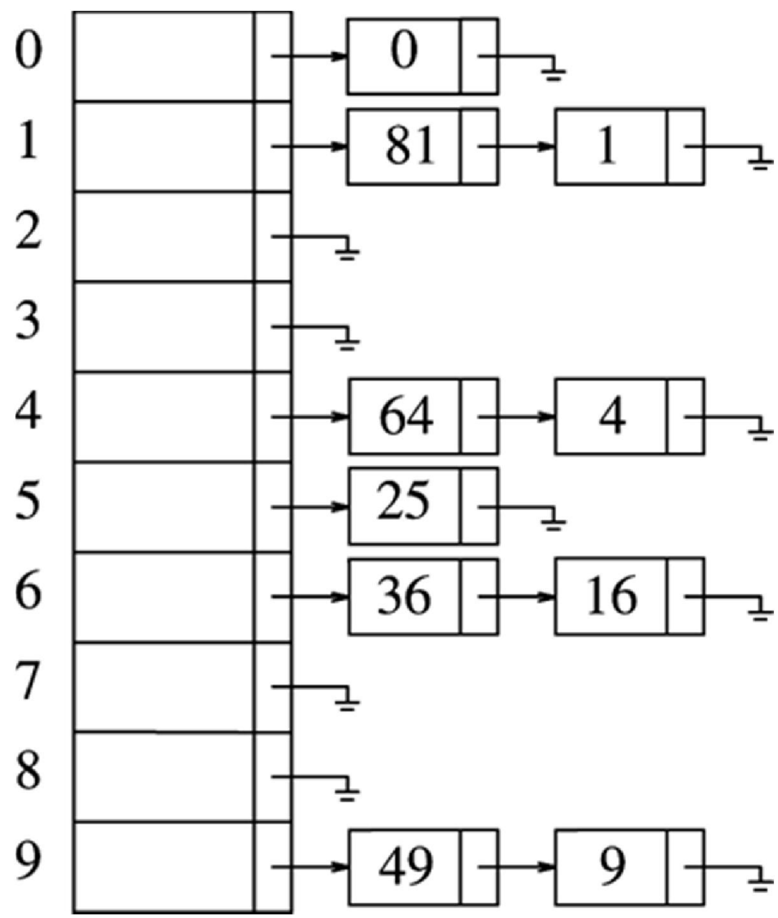
Insert 89, 18, 49, 58, 69 with Hash(K)=K%10 and quadratic probe

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Collision Resolution with Buckets

- Each table entry stores a list of items
- So we don't need to worry about multiple keys getting mapped to the same entry.
- Example:

49,0,9,64,1,4,81,25,16, 36



Rehashing

- Hash table may get *too* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has an adjusted hash function
  - Move all elements from smaller table to bigger table using the adjusted hash function
- Cost of Rehashing = $O(N)$
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter