# Lecture 22–24

- Tables

# Motivation

Suppose a set of *n* distinct records with keys $K_1$, $K_2$, …, $K_n$ are stored in a file and we want to find a record with a given key K.

- **Sequential search,** search time will increase with the increase in the number of records.

- **Access tables** makes search independent to the number of records

A function f, when applied on K, returns i, which holds the address of the record with key value K.

This method of accessing any record is called ***table lookup***.

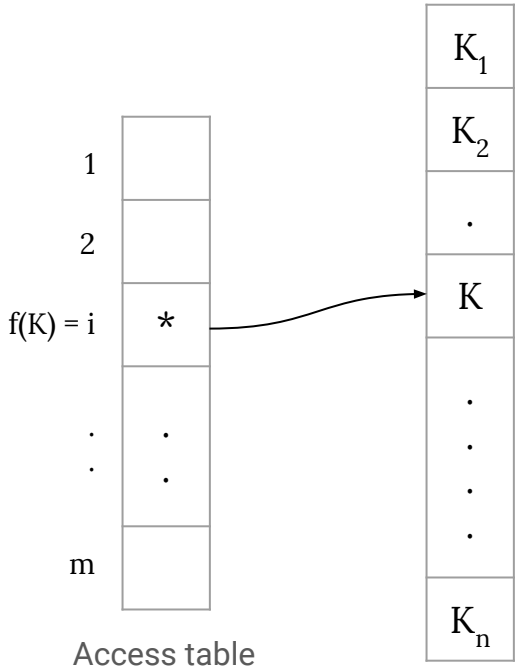
# Motivation

Suppose a set of *n* distinct records with keys $K_1$, $K_2$, …, $K_n$ are stored in a file and we want to find a record with a given key K.

- **Sequential search,** search time will increase with the increase in the number of records.
- **Access tables** makes search independent to the number of records

A function f, when applied on K, returns i, which holds the address of the record with key value K.

This method of accessing any record is called ***table lookup***.




**Information Retrieval through table lookup**

# Types of Tables

There are four different types of tables of interest.

- Rectangular tables (a.k.a. *matrices*)
- Jagged tables
- Inverted tables
- Hash tables

# Rectangular Tables

Rectangular tables are also known as Matrices that we have already discussed while studying 2-D arrays.

- Almost all programming languages provide convenient and efficient ways to store and retrieve data from them.
- All the implementation details are abstract to the programmer.

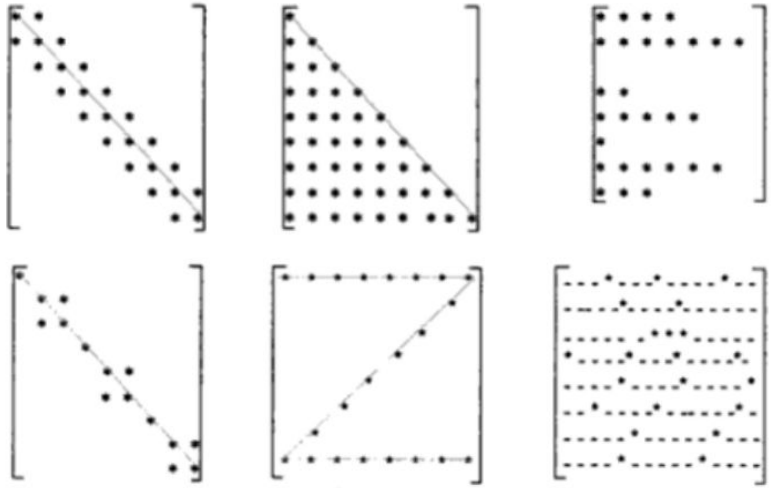
# Jagged Tables

Jagged tables are special kind of sparse matrices such as triangular matrices, band matrices, etc.

- We put a restriction that if elements are present in a row (or in a column) then they are contiguous.

$$\text{Address}(a_{ij}) = \frac{i \times (i-1)}{2} + j$$

This formula is computationally expensive (involves multiplication and division). **Any alternative?**



Not a jagged table.

# Jagged Tables

## Access table

| | |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 3 |
| 4 | 6 |
| 5 | 10 |
| 6 | 15 |

- The access table is calculated at the time of initiation and can be stored in memory. It only involves addition operations.

  E.g., $a_{54}$: a[ 5 ] + 4

- Using this, we can find indexing formula even if the elements are not arranged in the symmetric order.

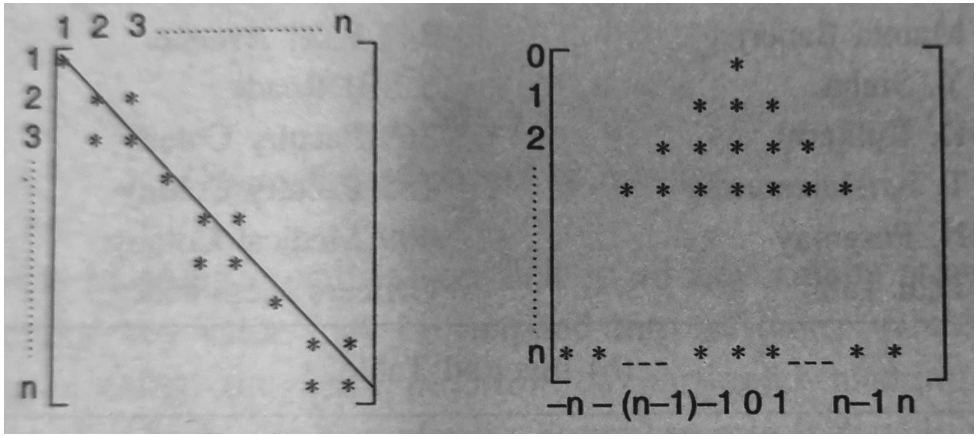
| | | | | | |
|---|---|---|---|---|---|
| I | | | | | |
| H | E | | | | |
| S | H | E | | | |
| Y | O | U | H | | |
| T | H | E | I | R | |
| M | Y | S | E | L | F |

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | H | E | S | H | E | Y | O | U | R | T | H | E | I | R | M | Y | S | E | L | F |

# Exercise

Consider two jagged tables as shown below. In one table, the row and column indices vary from 1 to n, and in the second table, the row index varies from -n to n.

- Devise an index formula for these two sparse matrices to store in arrays
- Obtain the access table and set their entries



# Inverted Tables

Suppose, records need to be sorted using multiple criteria. E.g., records of students need to be sorted based on Student IDs, SPI, CPI, DoB, etc.

- Exhaustive solution is to store different copies of the dataset for each of the criteria. (require extra storage)

- Difficulty in modification of records
(changes need to be reflected in all copies of the dataset)

Solution is to use Inverted Table: each column contains the index of records in the order based on the sorting of the corresponding key.

# Inverted Tables

| Index | Name | Address | Phone |
|---|---|---|---|
| 1 | K.R. Narayana | Maker Towers #6 | 257696 |
| 2 | A.B. Vajpayee | 9 Vivekananda Road | 257459 |
| 3 | L.K. Advani | 11 Von Kasturba Marg | 257583 |
| 4 | Mamta Banerjee | 342 Patel Avenue | 257423 |
| 5 | Y. Sinha | 5 SBI Road | 257504 |
| 6 | D. Kulkarni | 369 Faculty Colony | 257564 |
| 7 | T. Krishnamurthy | 185 Faculty Colony | 257579 |
| 8 | N. Puranjay | 409 Medical Colony | 257409 |
| 9 | Tadi Tabi | Officers Mess #52 | 257871 |

| Name | Address | Phone |
|---|---|---|
| 2 | 7 | 8 |
| 6 | 6 | 4 |
| 1 | 1 | 2 |
| 3 | 8 | 5 |
| 4 | 9 | 6 |
| 8 | 4 | 7 |
| 7 | 5 | 3 |
| 9 | 2 | 1 |
| 5 | 3 | 9 |

# Motivation for Hash Tables

**Why don't we use keys as index? Why to use a special mapping function f?**

Suppose each element in the data structure has a key drawn from {0, 1, . . . , m − 1}, where m is not too large, and no two elements have the same key.

- Then we can simply build an array T[ 0…m-1 ], where the element with key k is placed at T[k], and T[ i ] = NIL if there is no element in the data structure with key i.

- Depending on the application, we may either place the value itself at T(k), or just a pointer to the value.

This structure is called a **direct-address table**, and Search, Insert, and Delete all run in O(1), since they simply correspond to accessing the elements in the array.

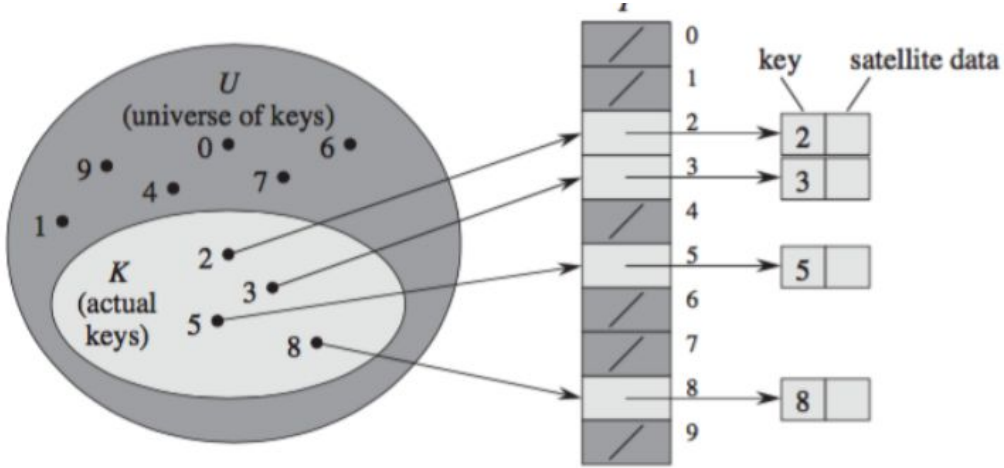
# Motivation for Hash Tables

Why don't we use keys as index? Why to use a special mapping function f?

Direct-address tables are impractical –

- when the number of possible keys is large,
- when it far exceeds the number of keys that are actually stored.

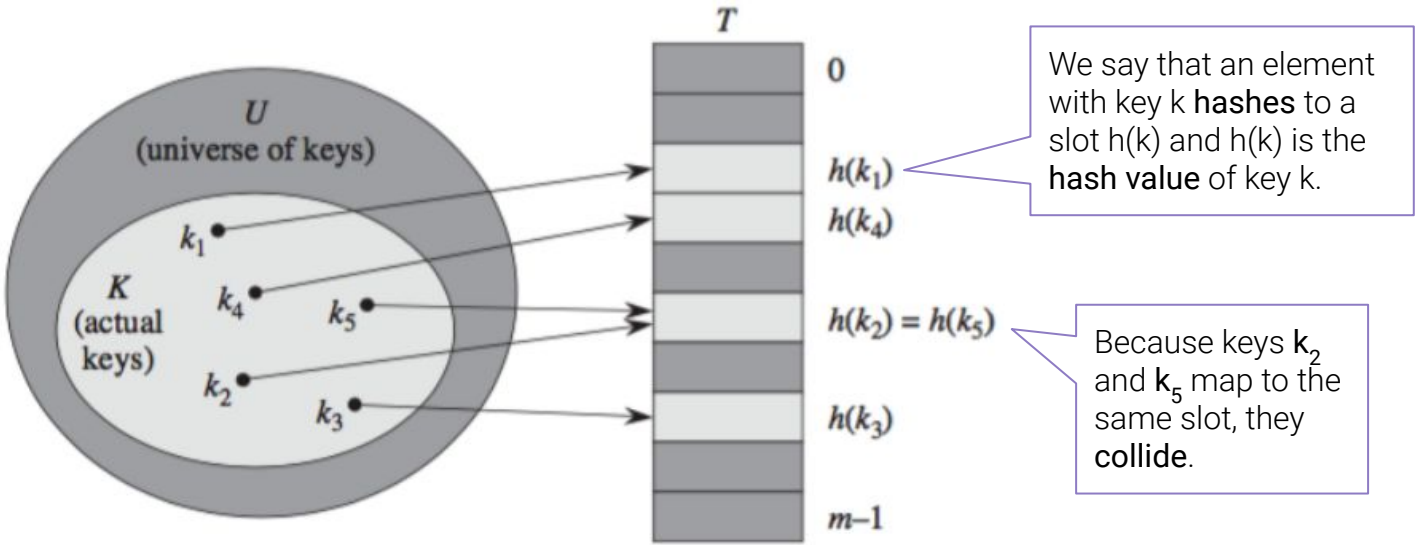
Therefore, we use hash tables.



# Hash Tables

With hash tables, instead of storing the element with key **k** in **slot k**, we store it in **slot h(k)**.

- Here **h: U → {0, 1, . . . , m − 1}** is a **hash function** that maps all possible keys to the slots of an array T[0…m - 1] (here m << | U |).



# Hash Tables

With hash tables, instead of storing the element with key **k** in **slot k**, we store it in **slot h(k)**.

- Here **h: U → {0, 1, . . . , m − 1}** is a **hash function** that maps all possible keys to the slots of an array T[0…m - 1] (here m << | U |).
    - **h** is not a one-to-one function (otherwise, hashing would add no value over direct addressing),
    - so, elements with different keys may be hashed to the same slot in the array which produces a **hash collision**.

We will discuss collision resolution techniques later.

# Hash Function

Here **h: U → {0, 1, . . . , m − 1}** is a **hash function** that maps all possible keys to the slots of an array T[0…m - 1] (here m << | U |).

- A good hash function satisfies (approximately) the assumption of **independent uniform hashing**: for each possible input k in the domain U, an output h(k) that is an element randomly and independently chosen uniformly from the range {1, …, m-1}, .
- h should be very easy and quick to compute
- h should as far as possible give two different indices for two different keys

There are a few very common and popular hash functions.

# Hash Function

**Division method**: one of the fast and most widely accepted methods.

Let **h(k) = k mod m** for some value of m; e.g., if m = 12 and k = 100, then h(k) = 4.

Some values of m are better than others.

- m as a power of 2, h(k) is just the lowest-order bits of k, and generally not all low-order bit patterns are equally likely.
- m as a prime number that is not too close to an exact power of 2 may be a better choice.

# Hash Function

**Multiplication method**: one of the fast and most widely accepted methods.

Let **h(k) = ⌊m(kA mod 1)⌋**. for some value of m and A

Some values of m are better than others.

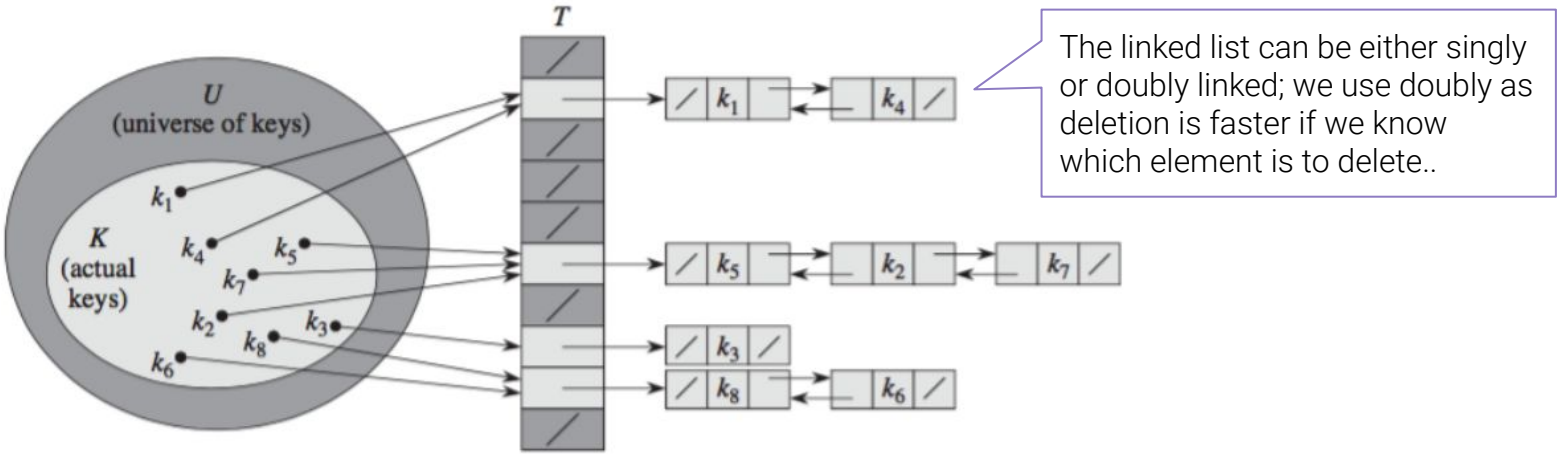
- Multiply the key k by some number $0 < A < 1$.
- Extract the fractional part of kA
- Multiply it by m.
- Take the floor of the results

Here the value of m is not critical, but some values of A work better than others. It is suggested that $A = (\sqrt{5} - 1)/2$ is likely to work reasonably well

# Collision Resolution by Chaining (a.k.a. Open Hashing)

One way to resolve hash collision is by having a linked list in each non-empty array slot that contains all of the elements whose keys map to that slot.

- More precisely, the array slot ***j*** contains either a pointer to the head of the list of all stored elements with hash value ***j***, or NIL if there are no elements currently in the table whose keys map to that slot.



# Collision Resolution by Chaining (a.k.a. Open Hashing)

The Insert, Search, and Delete operations reduce to Insert, Search, and Delete on a linked list:

- In the worst-case scenario, all of the keys hash to the same array slot, so Search takes O(n), while Insert and Delete still take O(1).
- This is in addition to the time taken to compute the hash function (which we usually assume is O(1)).
- However, we will primarily be concerned with the average case behavior. It turns out that in the average case, Search takes Θ(1 + α) time.
    - where **α = n/m** is the **load factor**: n being the number of elements, and m being the number of slots.

# Collision Resolution by Rehashing (a.k.a. Open Addressing)

In open addressing, we don't have linked lists, and every entry of the hash table contains either a single element or NIL (load factor $\alpha <= 1$).

- To insert an element, we first try to insert it into its "first choice" location, and if there is no room in that slot, we try to insert it into its "second choice" location, and we keep going until we discover all the slots are full.
- Different elements have different preference order of locations.
- We search an element in the decreasing order of its preference until we find it or we find an empty slot (indicating the element is not in the table).

# Collision Resolution by Rehashing (a.k.a. Open Addressing)

To perform insertion, we successively examine (**probe**) the hash table until we find an empty slot to put the key.

- Instead of being fixed in the order: 0, 1, …, m-1 (search takes O(n) time), the sequence of positions being probed depends on the key being inserted.
- To determine which slots to probe, the hash function includes the **probe number** (starting from 0) as a second input:

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

- We require that for every key k, the **probe sequence** h(k, 0), h(k, 1), . . . , h(k, m − 1) be a permutation of 0, 1, . . . , m − 1, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

# Collision Resolution by Rehashing (a.k.a. Open Addressing)

To perform insertion in open addressing, we successively examine (**probe**) the hash table until we find an empty slot to put the key.

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

Both insert and search in the hash table using open addressing is linear in terms of the size of the table: O(m).

# Collision Resolution by Rehashing (a.k.a. Open Addressing)

To perform deletion in open addressing, when we delete a key from a slot q, it would be a mistake to simply mark that slot as empty by storing NIL in it.

- We may not be able to retrieve a key k for which slot q was probed and found occupied when k was inserted.
- We may mark that slot with a special value “DELETED” in place of “NIL”. While inserting an element, this slot should be treated as empty.
- This is still not a good solution, because if we delete lots of elements from the table, search times no longer depend on the load factor of the table (they may be much larger than would be predicted by the number of elements in the table).
- So usually **if we need to delete keys we would use chaining instead of open addressing**.

# Linear Probing: An Example of Open Addressing

These two-dimensional hash functions rely on the existence of a one-dimensional "**auxiliary hash function**" h': U → {0, 1, . . . , m − 1}.

Let **h(k, i) = (h'(k) + i) mod m** for i = 0, 1, . . . , m − 1.

- In other words, first we examine the slot T[h'(k)], i.e. the slot given by the auxiliary hash function. Then, we probe the slot next to it, and then the slot next to that slot, etc, wrapping around to the beginning of the array if necessary.

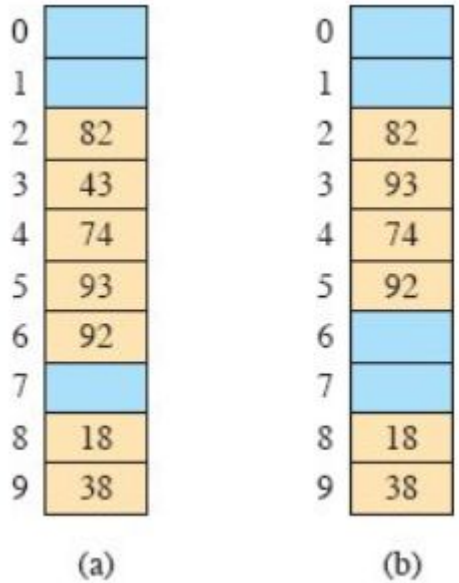
# Linear Probing: An Example of Open Addressing

The hash table has size 10 with h'(k): k mod 10.

(a) the hash table after inserting keys in the order: 74, 43, 93, 18, 82, 38, 92;

(b) after deleting the key 43 from slot 3 and 92 moves up to slot 5.

Only two values 93 and 92 are shifted, nothing else.



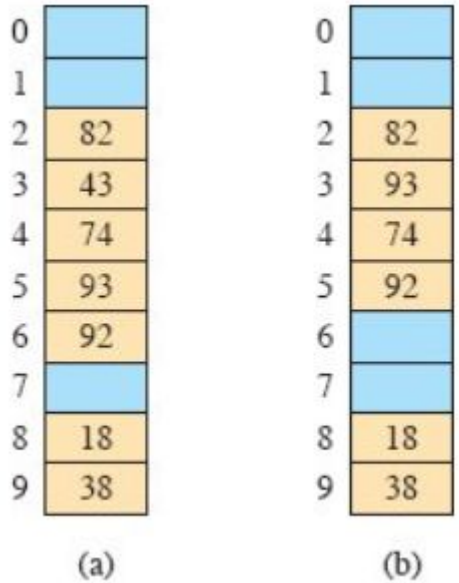
# Linear Probing: An Example of Open Addressing

This technique avoids the need of marking slots with "DELETED" as keys follow the same simple cyclic probling with different starting points.

- The deletion procedure relies on an "inverse" function to the linear probing hash function.
- The inverse function g, maps a key k and a slot q, where $0 \leq q < m$, to the probe number that reaches slot q:

$$g(k, q) = (q - h'(k)) \bmod m$$

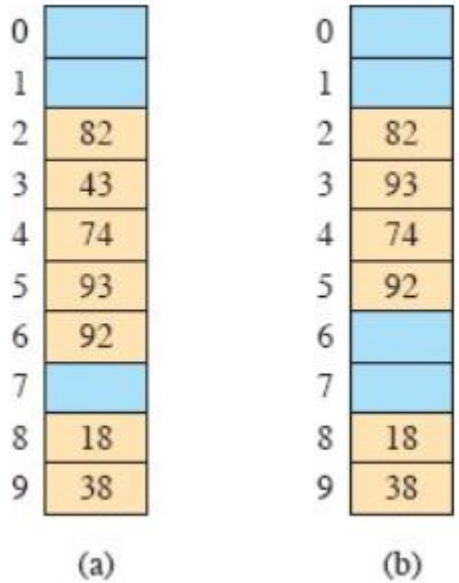
- If **h(k, i) = q**, then **g(k, q) = i**, and so **h(k, g(k, q)) = q**.



# Linear Probing: An Example of Open Addressing

This technique avoids the need of marking slots with "DELETED" as keys follow the same simple cyclic probling with different starting points.

```
LINEAR-PROBING-HASH-DELETE(T, q)
 1 while TRUE
 2     T[q] = NIL                    // make slot q empty
 3     q' = q                        // starting point for search
 4     repeat
 5         q' = (q' + 1) mod m       // next slot number with linear probing
 6         k' = T[q']                // next key to try to move
 7         if k' == NIL
 8             return                // return when an empty slot is found
 9     until g(k', q) < g(k', q')    // was empty slot q probed before q'?
10     T[q] = k'                     // move k' into slot q
11     q = q'                        // free up slot q'
```



# Linear Probing: An Example of Open Addressing

Let **h(k, i) = (h'(k) + i) mod m** for i = 0, 1, . . . , m − 1.

- This is easy to implement, but can be problematic because long runs of occupied slots build up, increasing the average search time — known as **primary clustering**.
- Clusters arise because an empty slot preceded by i full slots gets filled next with probability (i + 1)/m.
    - Long runs of occupied slots tend to get longer, and the average search time increases.

# Linear Probing: An Example of Open Addressing

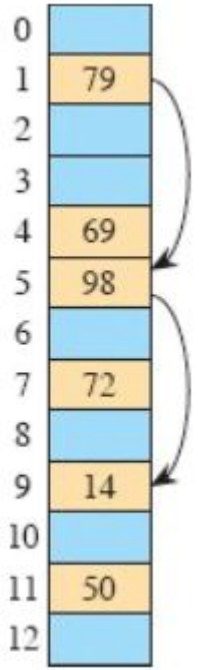
We assume independent uniform permutations hashing: the probe sequence of each key is equally likely to be any of the **m!** permutations of⟨0, 1, …, m − 1⟩.

- **Linear probing** cannot generate more than **m** different probe sequences.
- We will see **Double hashing** cannot generate more than $\mathbf{m^2}$ different probe sequences.

# Double Hashing: An Example of Open Addressing

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

- Here we use **two auxiliary hash functions, $h_1$ and $h_2$.**
- This is a great method and the permutations produced have many of the characteristics of randomly chosen permutations.
  - E.g., the hash table has size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$.
  - Since $14 = 1 \pmod{13}$ and $14 = 3 \pmod{11}$, the key 14 goes into empty slot 9, after slots 1 and 5 are probed and found to be occupied.



# Double Hashing: An Example of Open Addressing

- Note that $h_2(k)$ must be relatively prime to the hash table size **m** for the entire hash table to be searched.
    - One way to do this is to let **m** be a power of 2 and design $h_2$ so that it always produces an odd number.
    - Another way is to let **m** be prime and design $h_2$ so it always returns a positive integer less than **m**.
- For example, we can let $\mathbf{h_1(k) = k \bmod m}$ and $\mathbf{h_2(k) = 1 + (k \bmod m')}$, where **m'** is chosen to be slightly less than **m**.

# Double Hashing: An Example of Open Addressing

We analyze **open addressing** in terms of the load factor $\alpha = n/m$ ($\leq 1$, such that $n \leq m$) of the hash table.

- Under the assumption of independent uniform permutation hashing, the expected number of probes made in an unsuccessful search is at most $1/(1 - \alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$
  - The first probe always occurs. With probability approximately $\alpha$, the first probe finds an occupied slot, so that a second probe happens.
  - With probability approximately $\alpha^2$, the first two slots are occupied so that a third probe ensues, and so on.

# Double Hashing: An Example of Open Addressing

We analyze **open addressing** in terms of the load factor **$\alpha = n/m$** ($\leq 1$, such that $n \leq m$) of the hash table.

- Inserting an element into an open-address hash table with load factor $\alpha$, where $\alpha < 1$, requires at most $1/(1 - \alpha)$ probes on average, assuming independent uniform permutation hashing and no deletions.

# Quadratic Probing: An Example of Open Addressing

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

- This is a better method than linear probing, but we may have to select the constants carefully.
- Also, if two keys have the same initial probe position, then their probe sequences are the same, which means we still get clustering, but to a lesser extent.

# Next Lecture

- Trees