Introduction to Data Structures

The Search Problem

- Find items with **keys** matching a given **search key**
  - Given an array A, containing n keys, and a search key x, find the index i such as $x=A[i]$
  - As in the case of sorting, a key could be part of a large record.

example of a record

Key	**other data**

Hashing

- Aimed towards faster search performance
- Search techniques covered
  - Linear/sequential search
  - Binary search
- When log(*n*) is just too big…
  - air traffic control
  - packet routing
- What hashing does
  - Storage location depends on the item

Hashing: Usefulness

- Some other examples where hashing helps
  - Click count of webpages
  - Number of connections from a url
  - Reservations in a given flight
  - Unique terms from a book
  - List of visited cells/nodes in an application
- Basic idea: item itself determines (narrows down) where the element can be found
  - H:Item domain->set of storage locations
  - i=h(k) is where the item is present
  - (Not always true, but provides starting point)

The Search Problem

- Find items with **keys** matching a given **search key**
  - Given an array A, containing n keys, and a search key x, find the index i such as $x=A[i]$
  - As in the case of sorting, a key could be part of a large record.

example of a record

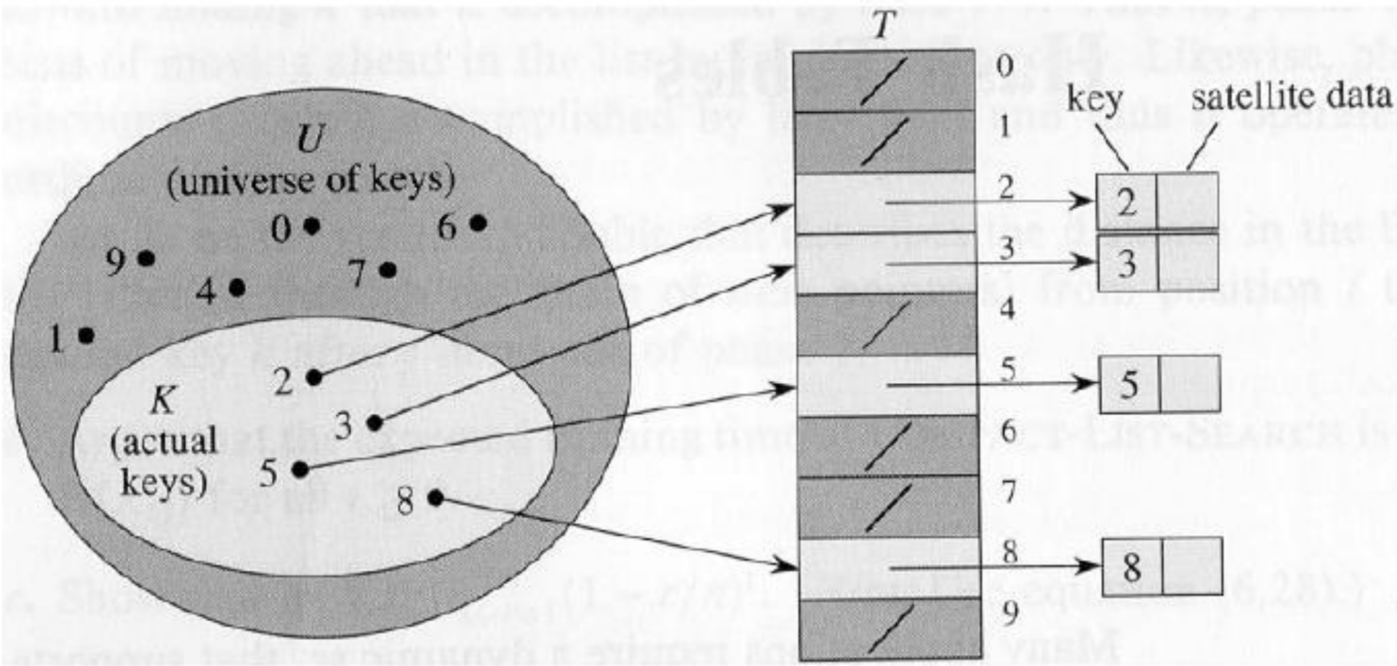
Key	**other data**

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe U = {0, 1, . . . , m - 1}
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

Alg.: DIRECT-ADDRESS-INSERT(T, x)

$T[key[x]] \leftarrow x$

Alg.: DIRECT-ADDRESS-DELETE(T, x)

$T[key[x]] \leftarrow NIL$

- Running time for these operations: $O(1)$

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very <u>inefficient</u> now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

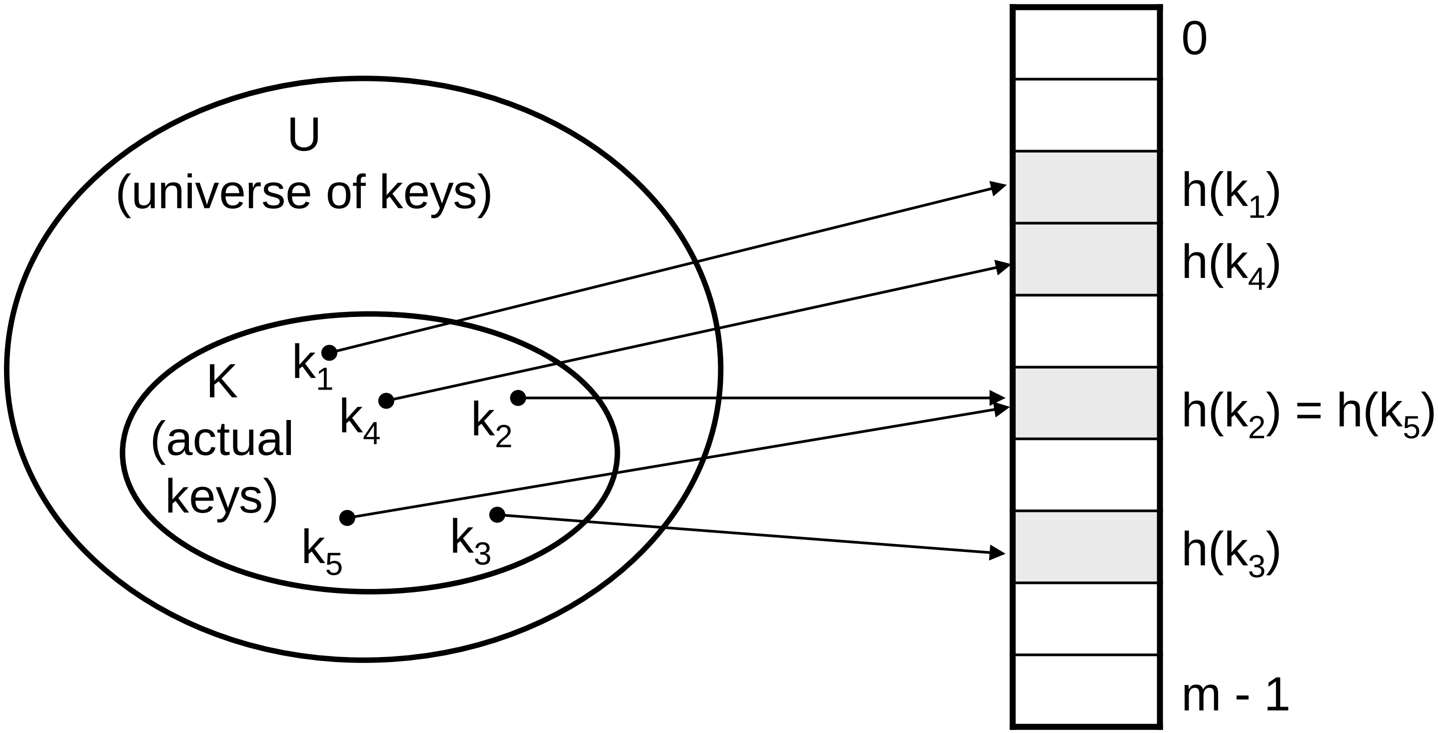
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

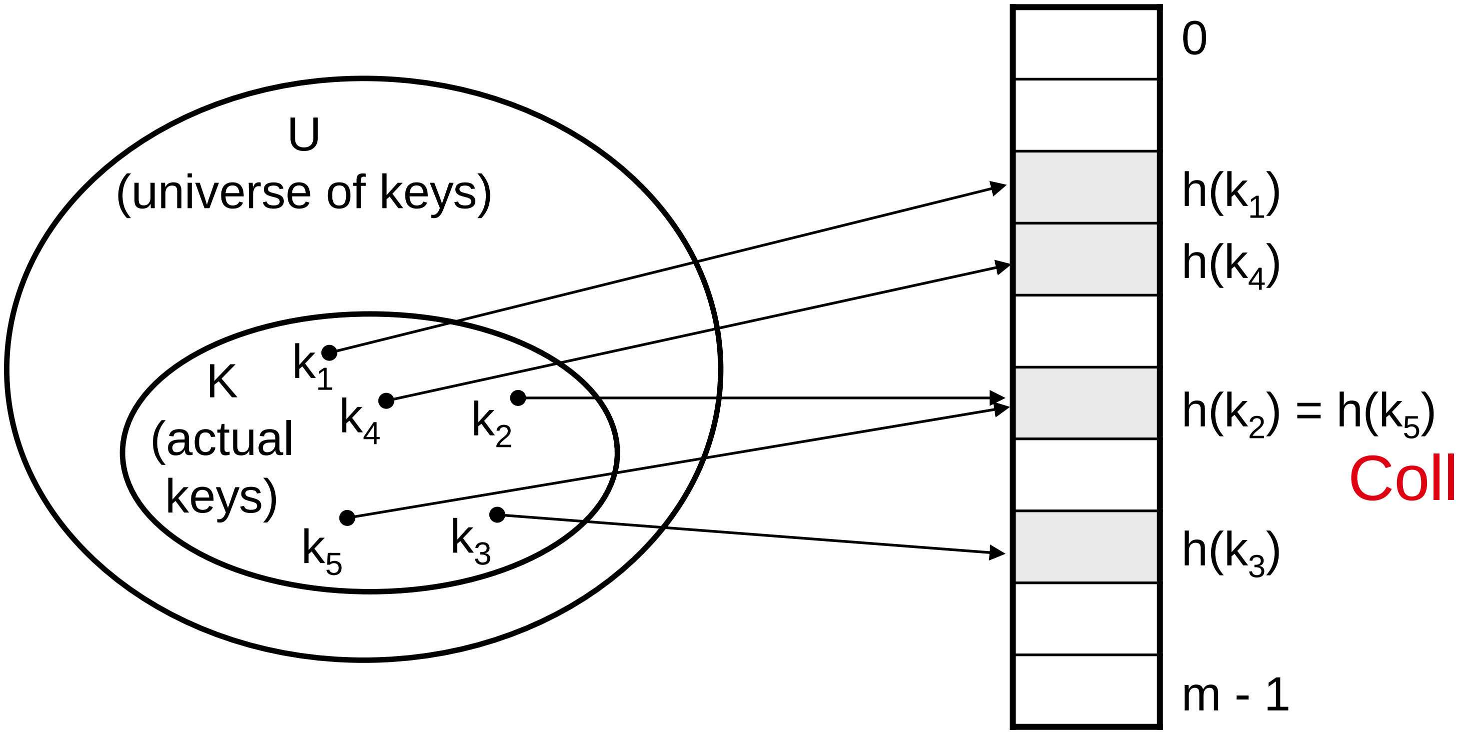
$$h : U \rightarrow \{0, 1, . . . , m - 1\}$$

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced

Example: HASH TABLES



Problems with this approach



Collisions!

Collisions

- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- We will review the following methods:
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Chaining

Hash Functions

- A hash function transforms a key into a table address
- **What makes a good hash function?**
  (1) Easy to compute
  (2) Approximates a random function: for every input, every output is equally likely (simple uniform hashing)
- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don't know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as pt and pts should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

Properties of Good Hash Functions

- Must return number 0, …, tablesize
- Should be efficiently computable – O(1) time
- Should not waste space unnecessarily
  - For every index, there is at least one key that hashes to it
  - Load factor lambda λ = (number of keys / TableSize)
- Should minimize collisions

  = different keys hashing to same index

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime*. Why?

Integer Keys

- Hash(x) = x % TableSize
- Good idea to make TableSize *prime*. Why?
  - Because keys are typically not randomly distributed, but usually have some *pattern*
    - mostly even
    - mostly multiples of 10
    - in general: mostly multiples of some k
  - If k is a factor of TableSize, then only (TableSize/k) slots will ever be used!
  - Since the only factor of a prime number is itself, this phenomena only hurts in the (rare) case where k=TableSize

Strings as Keys

- If keys are strings, can get an integer by adding up ASCII values of characters in *key*

```
for (i=0;i<key.length();i++)
        hashVal += key.charAt(i);
```

- **Problem 1**: What if *TableSize* is 10,000 and all keys are 8 or less characters long?

- **Problem 2**: What if keys often contain the same characters (“abc”, “bca”, etc.)?

Hashing Strings

- Basic idea: consider string to be a integer (base 32):
  Hash(“abc”) = (‘a’*32^2 + ‘b’*32^1 + ‘c’) % TableSize

- Horner’s Rule

```
int hash(char s[]) {
  h = 0;
  for (i = strlen(s) - 1; i >= 0; i--) {
    h = (s[i] + h<<5) % tableSize;
  }
  return h;
}
```

How Can You Hash…

- A set of values – (name, birthdate) ?

  (Hash(name) ^ Hash(birthdate))% tablesize

- An arbitrary pointer in C?

  ((int)p) % tablesize

- IP address?

  a.b.c.d
  Concatenate a, b, c, d to get s
  Treat s as an integer and use h(s)%tablesize

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m \, \underbrace{(k\,A \bmod 1)}_{\text{fractional part of } kA \,=\, kA - \lfloor kA \rfloor} \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Example – Multiplication Method

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

```
      .101101 (A)
       110101 (k)
--------------
1001010.0110011 (kA)
```

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

take integer part: 011

thus, h(110101)=011

Generalize hash function notation:

- A hash function contains two arguments now:
  (i) Key value, and (ii) Probe number

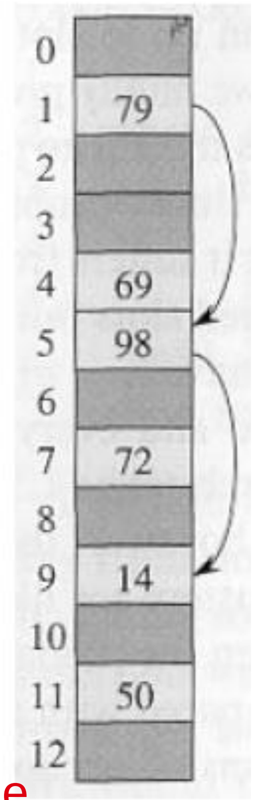
$$h(k,p), \quad p=0,1,\ldots,m-1$$

- Probe sequences

  <h(k,0), h(k,1), ..., h(k,m-1)>

  - Must be a permutation of <0,1,...,m-1>
  - There are $m!$ possible permutations
  - Good hash functions should be able to produce all $m!$ probe sequences

insert 14



Example

<1, 5, 9>

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Alternative Strategy: Closed Hashing

Problem with separate chaining:

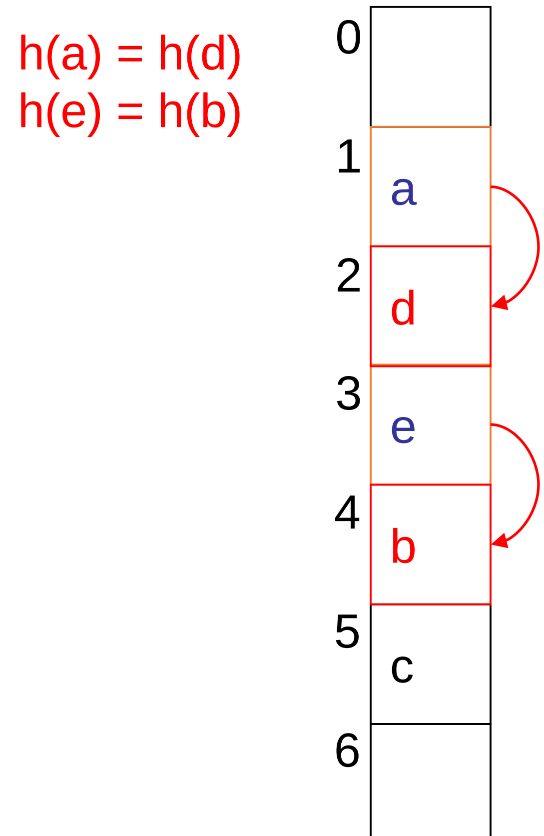
Memory consumed by pointers –

32 (or 64) bits per key!

What if we only allow one Key at each entry?

- two objects that hash to the same spot can’t both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties
  - $\lambda \leq 1$
  - performance degrades with **difficulty of finding** right spot



Collision Resolution by Closed Hashing

- Given an item X, try cells $h(X,0), h(X,1), h(X,2), \ldots, h(X,i)$
- $h(X,i) = (Hash(X) + F(i))\ mod\ TableSize$
  - Define $F(0) = 0$
- F is the *collision resolution* function. Some possibilities:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double Hashing: $F(i) = i * Hash_2(X)$

Closed Hashing I: Linear Probing

- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h(X, i) = (Hash(X) + i) \bmod TableSize \quad (i = 0, 1, 2, \ldots)$
  - Compute hash value and increment it until a free cell is found

Linear Probing Example

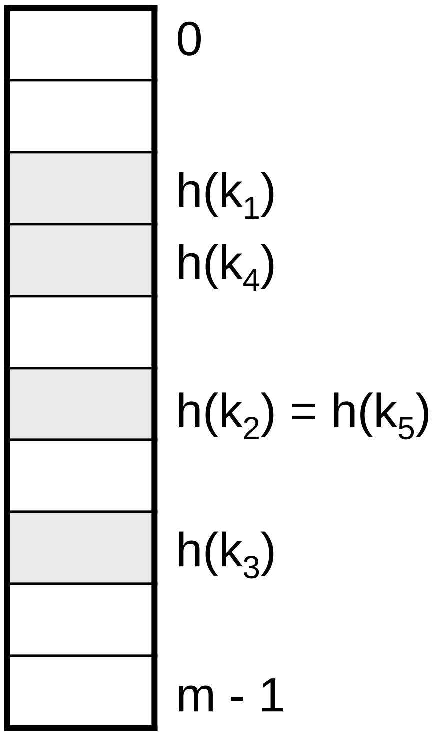
	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	12
3				2
4				
5				
6				
probes:	1	1	3	2

Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)
- Very prone to cluster formation (as in our example)
  - If a key hashes *anywhere* into a cluster, finding a free cell involves going through the entire cluster – and making it grow!
  - *Primary clustering – clusters grow when keys hash to values close to each other*
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of *distributing keys uniformly*

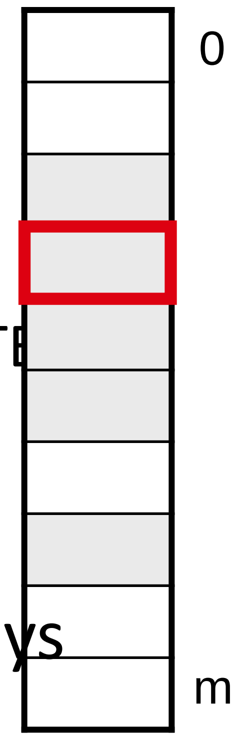
Linear probing: Searching for a key

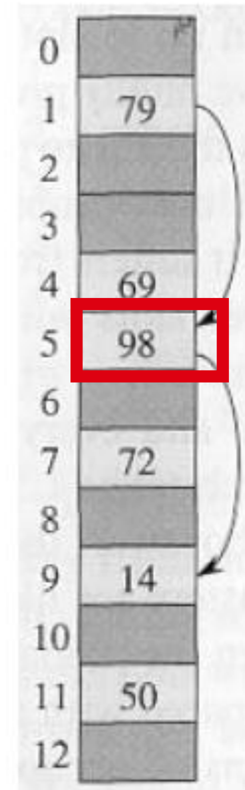
- Three cases:
  - (1) Position in table is occupied with an element of equal key
  - (2) Position in table is empty
  - (3) Position in table occupied with a different element
- Case 3: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table



Linear probing: Deleting a key

- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETE
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys



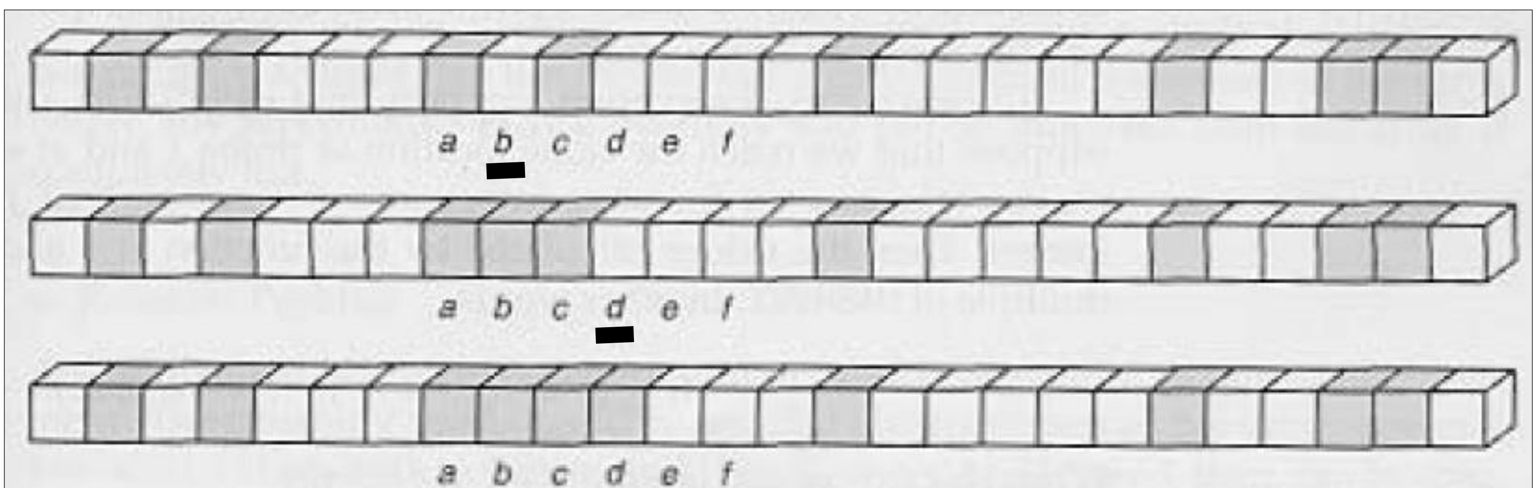


Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

$\Rightarrow$ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, i=0,1,2,... where $h' : U -- > (0, 1, \ldots, m-1)$

- Clustering problem is less serious but still an issue (*secondary clustering*)

- How many probe sequences quadratic probing generate ? m

(the initial probe position determines the probe sequence)