Data Structure and Algorithms (CSL2020)

[L10:Hashing]

Dr. Dip Sankar Banerjee

Department of Computer Science and Engineering

IIT Jodhpur

Jan 2024

Data structures for Sets

- Many applications deal with sets.
  - Compilers have symbol tables (set of vars, classes)
  - IP routers have IP addresses, packet forwarding rules
  - Web servers have set of clients, etc.
  - Dictionary is a set of words.

- A set is a collection of members
  - No repetition of members
  - Members themselves can be sets
- Examples
  - {x | x is a positive integer and x < 100}
  - {x | x is a CA driver with > 10 years of driving experience and 0 accidents in the last 3 years}
  - All webpages containing the word Algorithms

ADT

- Set + Operations define an ADT.
  - A set + insert, delete, find
  - A set + ordering
  - Multiple sets + union, insert, delete
  - Multiple sets + merge
  - Etc.
- Depending on type of members and choice of operations, different implementations can have different asymptotic complexity.

Dictionary ADTs

- Data structure with just 3 basic operations:
  - find (i): find item with key (identifier) i
  - insert (i): insert i into the dictionary
  - remove (i): delete i
  - Just like words in a Dictionary
- Where do we use them:
  - Symbol tables for compiler
  - Customer records (access by name)
  - Games (positions, configurations)
  - Spell checkers
  - P2P systems (access songs by name), etc.

Naïve Methods

- Linked list of the keys
  - insert (i): add to the head of list. Easy and fast O(1)
  - find (i): worst-case, search the whole list (linear)
  - remove (i): also linear in worst-case

- An array (bit vector) for all possible keys
- Map key i to location i
  - insert (i): set A[i] = 1
  - find (i): return A[i]
  - remove (i): set A[i] = 0

Naïve Methods

- Linked list space-efficient, but search-inefficient.
  - Insert is O(1) but find and delete are O(n).
  - A sorted search structure (array) also no good. The search becomes fast, but insert/delete take O(n).

- Bit Vector search-efficient but space-inefficient.

- Balanced search trees work but take O(log n) time per operation, and complicated.

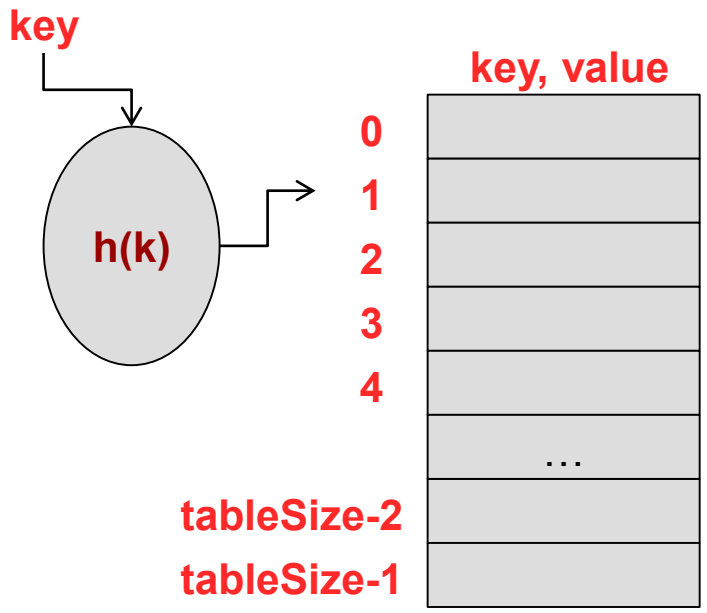
Introducing Hash Table

- A hash table stores key-value pairs
- Assume keys are integers {0, 1, …, |U|}
  - Non-numeric keys (strings, webpages) converted to numbers: Sum of ASCII values, first three characters
  - The keys come from a known but very large set, called universe U (e.g. IP addresses, program states)

- The set of keys to be managed is S is a subset of U.
  - The size of S is much smaller than U, namely, |S| << |U|
  - We use n for |S|.

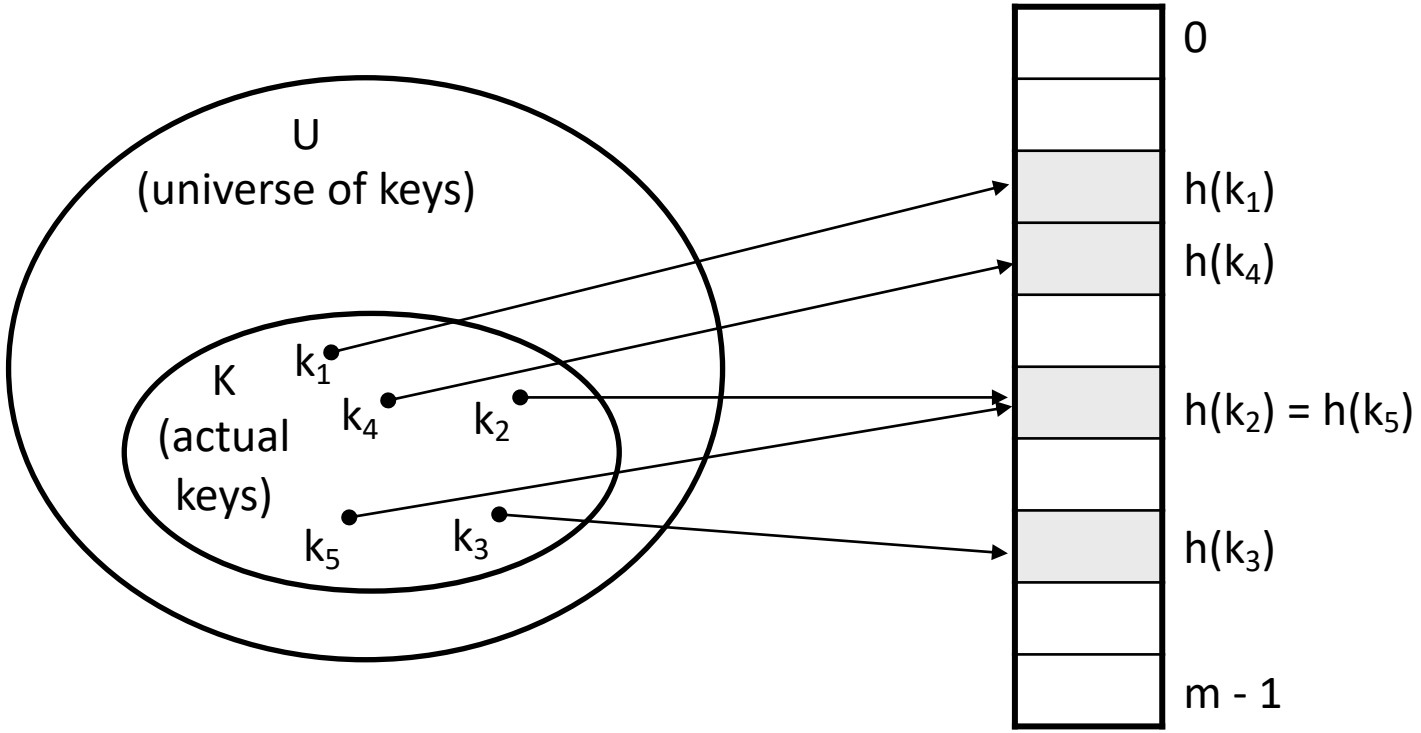
Hash Table

- Key idea is that instead of direct mapping, Hash Tables use a Hash Function h to map each input key to a unique location in table of size m
  - h : U -> {0, 1, …, m-1}
  - hash function determines the hash table size.
- Hash function lets us find an item in O(1) time.
  - Each item is uniquely identified by a key
  - Just check the location h(key) to find the item



m = tableSize
n = # of keys entered

Example: Hash Table



Key Concern

- How big the hash table should be?
  - A list of 1000 students, each identified by 6 digit ID?
  - A list of 10-digit phone number to store the customer record?

*=> The table size should be bigger than the amount of expected entries (**m** > **n**)*

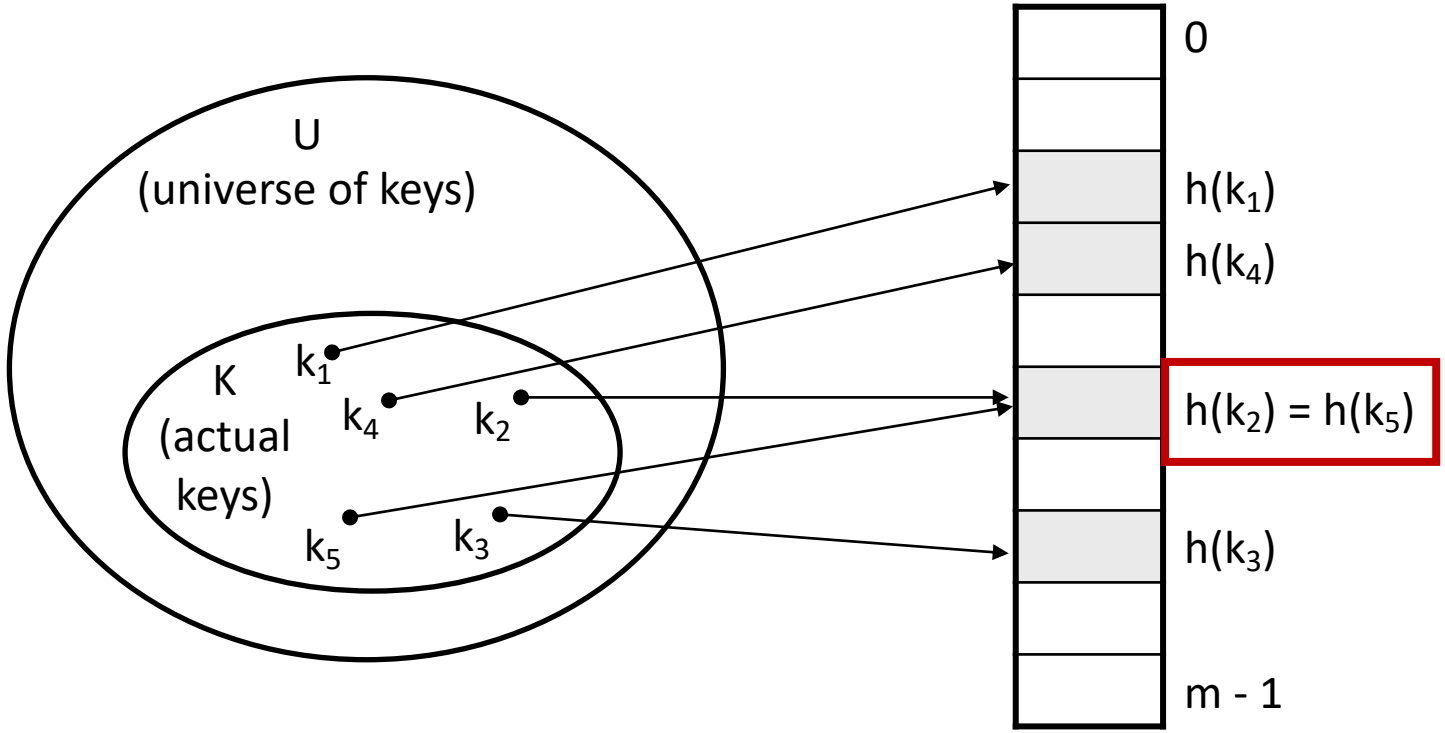
- How to choose hash functions?

Challenge: Distribute keys to locations in hash table such that

- Easy to compute and retrieve values given key
- Keys evenly spread throughout the table

Hash Table Issues

Suppose that the keys are nine-digit social security numbers



Possible hash function

$h(ssn) = sss \bmod 100$ (last 2 digits of ssn)

e.g., if $ssn = 10123411$ then $h(10123411) = 11$)

Collisions!

Collision

- 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

Pigeon Hole Principle

- Recall for hash tables we let...
  - – **n** = # of entries (i.e. keys)
  - – **m** = size of the hash table
- If **n** > **m**, is every entry in the table used?
- – No. Some may be blank?
- Is it possible we haven't had a collision?
  - – No. Some entries have hashed to the same location
  - – Pigeon Hole Principle says given **n** items to be slotted into m holes and **n** > **m** there is at least one hole with more than 1 item
  - – So if **n** > **m**, we know we've had a collision
- We can only avoid a collision when **n** < **m**

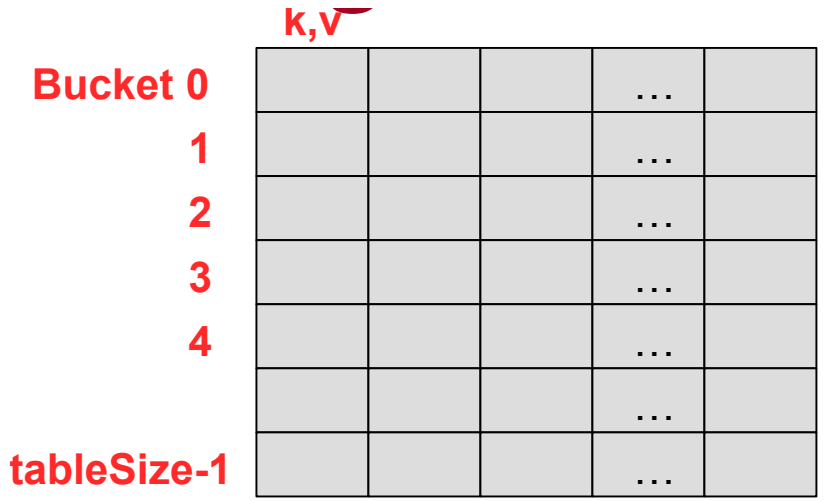
Resolving Collisions

- Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2).
- Collisions are inevitable if the number of entries, **n**, is greater than table size, **m** (***by pigeonhole principle***)
- Methods
  - Closed Addressing (e.g. buckets or **chaining**)
  - Open addressing (aka probing)
    - Linear Probing
    - Quadratic Probing
    - Double-hashing

Buckets/Chaining

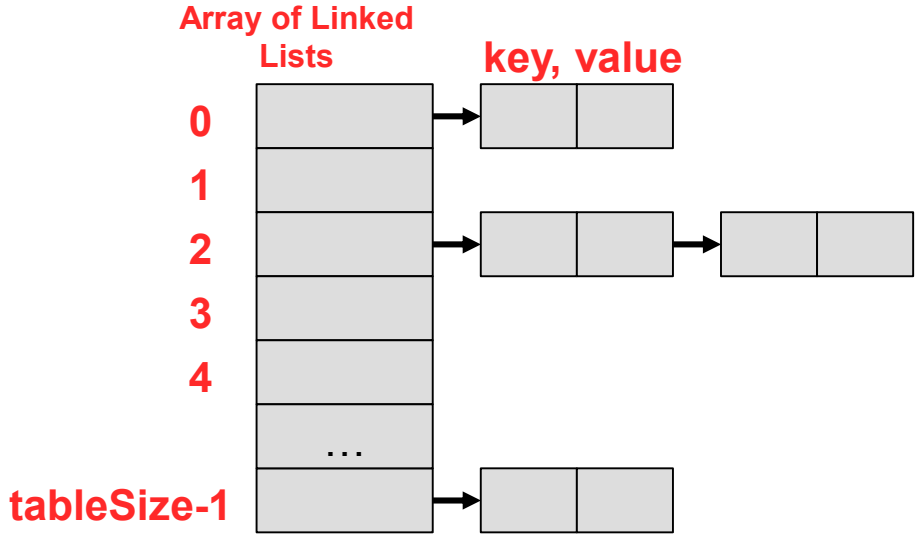
Simply allow collisions to all occupy the location they hash to by making each entry in the table an ARRAY (bucket) or LINKED LIST (chain) of items/entries

– Closed Addressing => You will live in the location you hash to (it's just that there may be many places at that location)



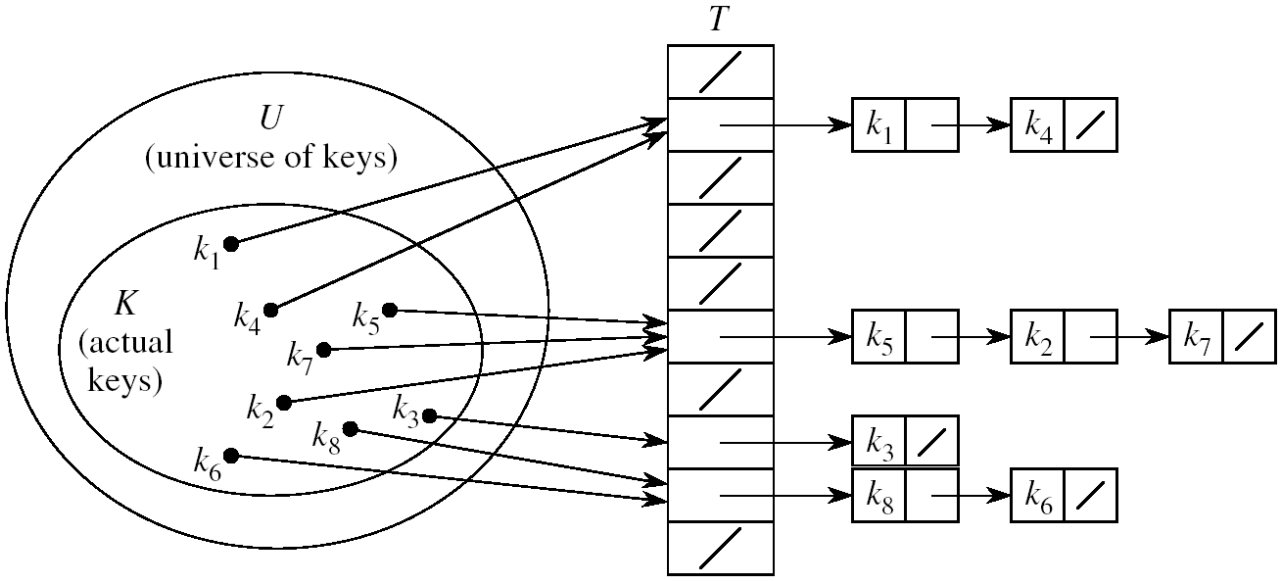
- **Buckets**
  - How big should you make each array?
  - Too much wasted space

- **Chaining**
  – Each entry is a linked list



Handling collision with chaining

- **Idea**:
  - 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 elements that hash to j

Collision with chaining- Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in the hash table

Alg.: CHAINED-HASH-INSERT(T, x)

insert x at the head of list T[h(key[x])]

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Deletion in the hash table

Alg.: CHAINED-HASH-DELETE(T, x)

delete x from the list T[h(key[x])]

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in the hash table

Alg.: CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

- Running time is proportional to the length of the list of elements in slot $h(k)$

Analysis of Hashing with Chaining

- **Worst case**
  - All keys hash into the same bucket
  - a single linked list.
  - insert, delete, find take O(n) time.
- **Average case** → (depends on how well the hash function distributes the n keys among the m slots)
  - **Simple uniform hashing** assumption: Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
  - Length of a list: $T[j] = n_j$, $j = 0, 1, \ldots, m - 1$
  - Number of keys in the table: $n = n_0 + n_1 + \cdots + n_{m-1}$
  - Average value of n_j: $E[n_j] = \lambda = n/m$

Load Factor

- The load factor λ of a probing hash table is the fraction of the table that is full. The load factor ranges from 0 (empty) to 1 (completely full).
  - λ =n/m, where n = # of elements stored in the table, and m = # of slots in the table = # of linked lists

- It is better to keep the load factor under 0.7
- Cost of Hash function f(x) must be minimized
  - In a failed search, avg cost is λ
  - In a successful search, avg cost is 1 + λ /2

Hash Function

- 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

Key Objective

- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

A good choice is $h(x) = x \bmod p$, for prime p **(why?)**
$h(x) = (ax + b) \bmod p$ called pseudo-random hash functions

The Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

    $$h(k) = k \bmod m$$
- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example: The Division Method

- If $m = 2^p$, then h(k) is just the least significant p bits of k
  - p = 1 $\Rightarrow$ m = 2

    $\Rightarrow$ h(k) = , least significant 1 bit of k
  - p = 2 $\Rightarrow$ m = 4

    $\Rightarrow$ h(k) = , least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

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- The 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

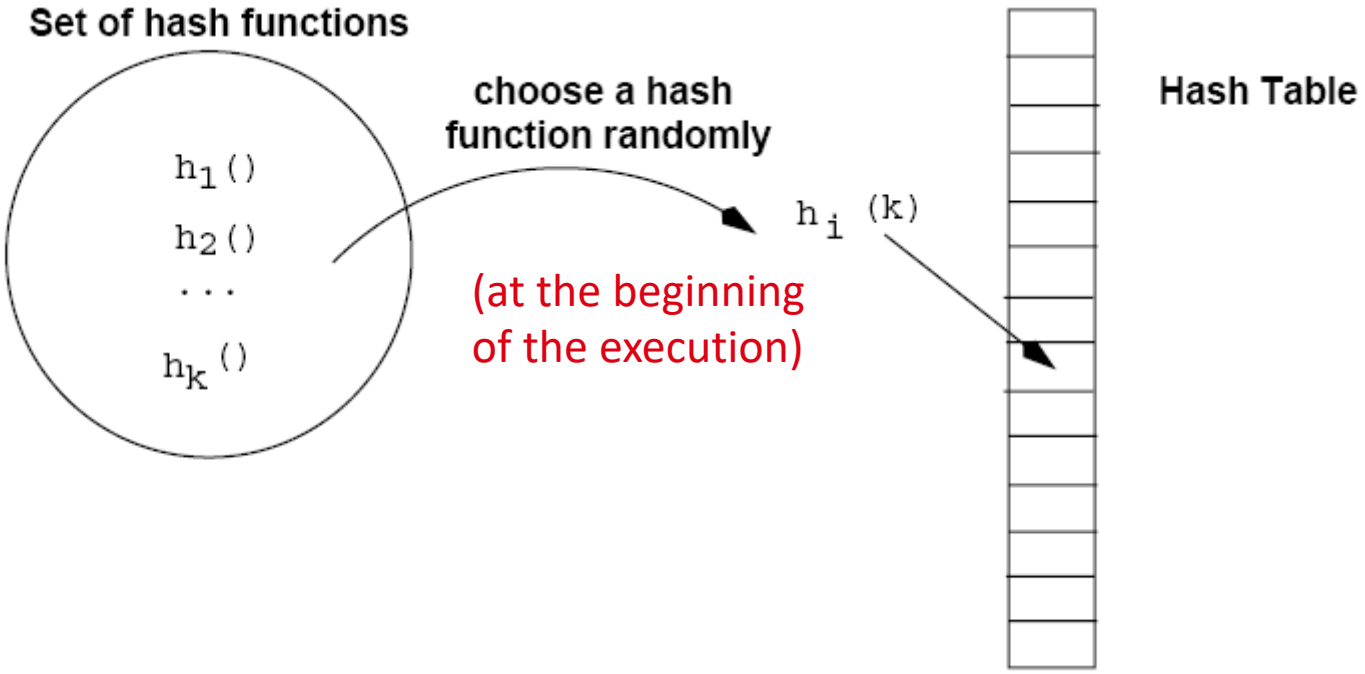
take integer part: 011

thus, h(110101)=011

Universal Hashing

- In practice, keys are not randomly distributed
- Any fixed hash function might yield Θ(n) time
- Goal: hash functions that produce random table indices irrespective of the keys
- Idea:
  - Select a hash function at random, from a designed class of functions at the beginning of the execution

Universal Hashing



Universal Hash Functions

- With universal hashing the chance of collision between distinct keys k and l is no more than the 1/m chance of collision if locations h(k) and h(l) were randomly and independently chosen from the set {0, 1, …, m – 1}

H={h(k): U→(0,1,..,m-1)}

H is said to be universal if

$$\text{for } x \neq y, |\mathbf{(h() \in H: h(x)=h(y)|=|H|/m}$$

(notation: |H|: number of elements in H - cardinality of H)

■Theorem. Suppose H is universal, S is an n-element subset of U, and h a random hash function from H.

□The expected number of collisions is at most (n-1)/M for any x in S.

Example: Universal Hash Function

E.g.: $p = 17, m = 6$

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$$

$$= (28 \bmod 17) \bmod 6$$

$$= 11 \bmod 6$$

$$= 5$$

Advantages of Universal Hash Functions

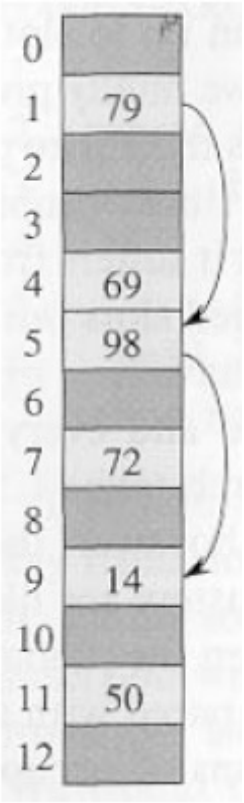
- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

Open Addressing

- If we have enough contiguous memory to store all the keys (m > N) ⇒ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one,

    until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult ... (we'll see why)
- Search time depends on the length of the probe sequence!

e.g., insert 14



Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Consider hash function $h(k,p)$ containing two arguments: (1) key value (k) and (2) probe number, $p=0,1,\ldots,m-1$

Probe sequences $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ must be a permutation of $\langle 0,1,\ldots,m-1\rangle$

Note:

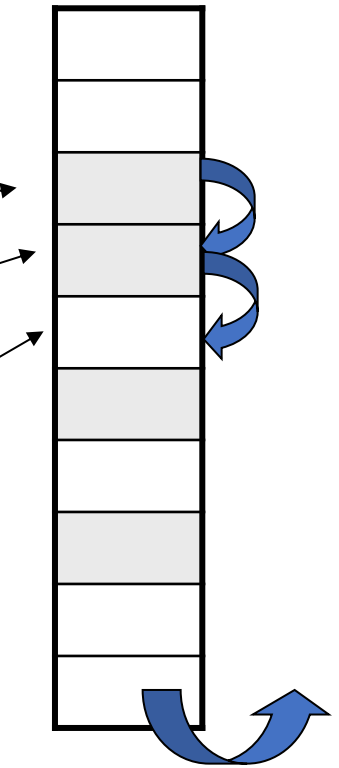
- None of these methods can generate more than m^2 different probing sequences!
- Good hash functions should be able to produce all $m!$ probe sequences

Linear probing

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k,i) = (h_1(k) + i) \bmod m$$
$$i=0,1,2,...$$

- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on

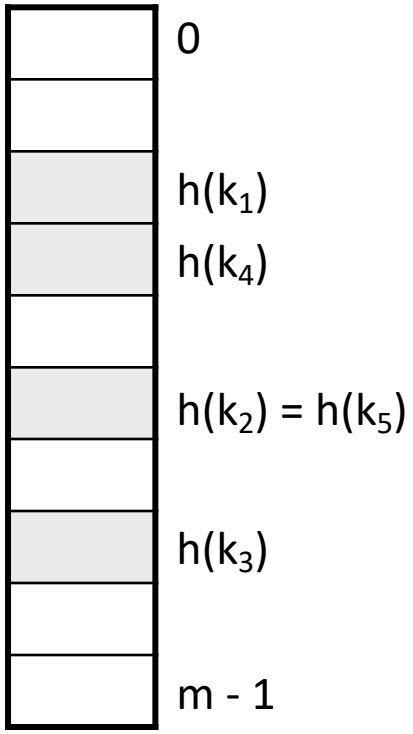


wrap around

- Can generate m probe sequences maximum, why?

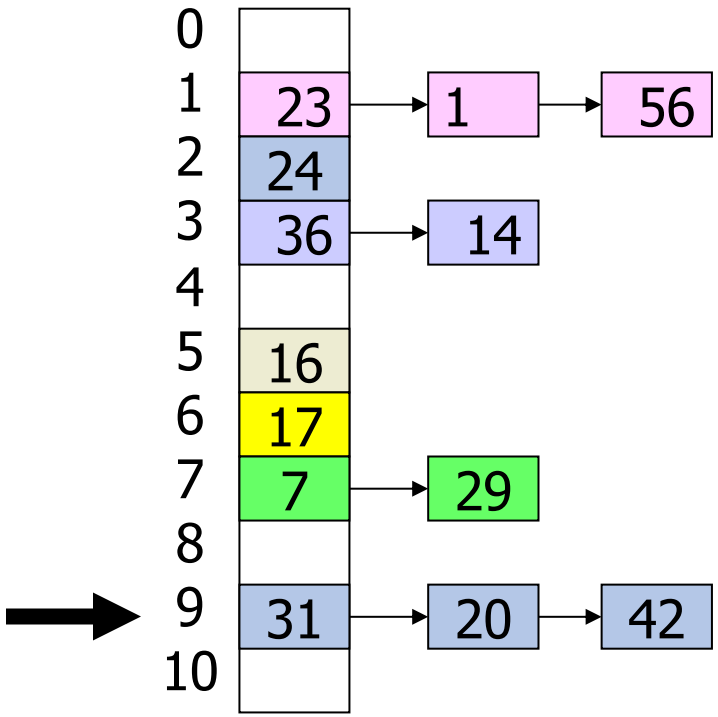
Linear Probing: searching for a key

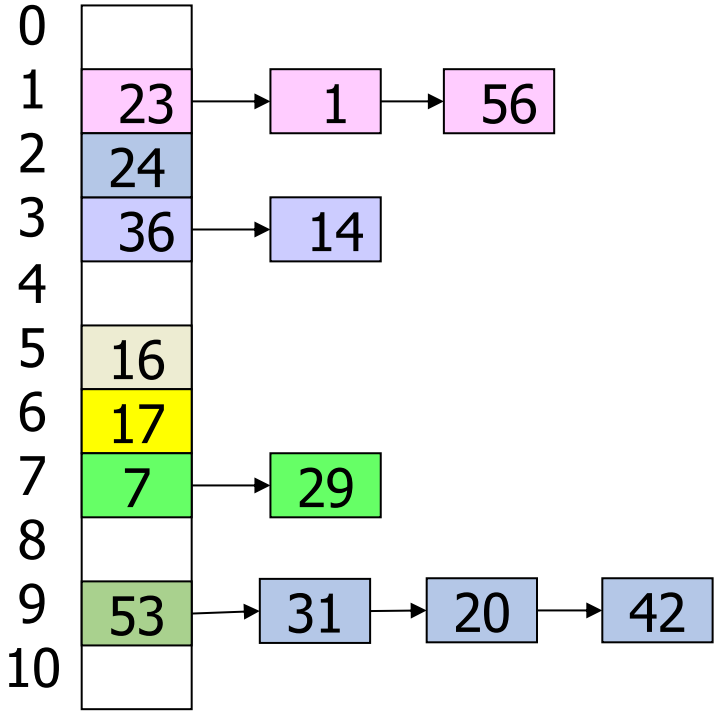
- Three cases:
  - (1) Position in table is occupied with an element of equal key
  - (2) Position in table is empty
  - (3) Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to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



Insertion: insert 53

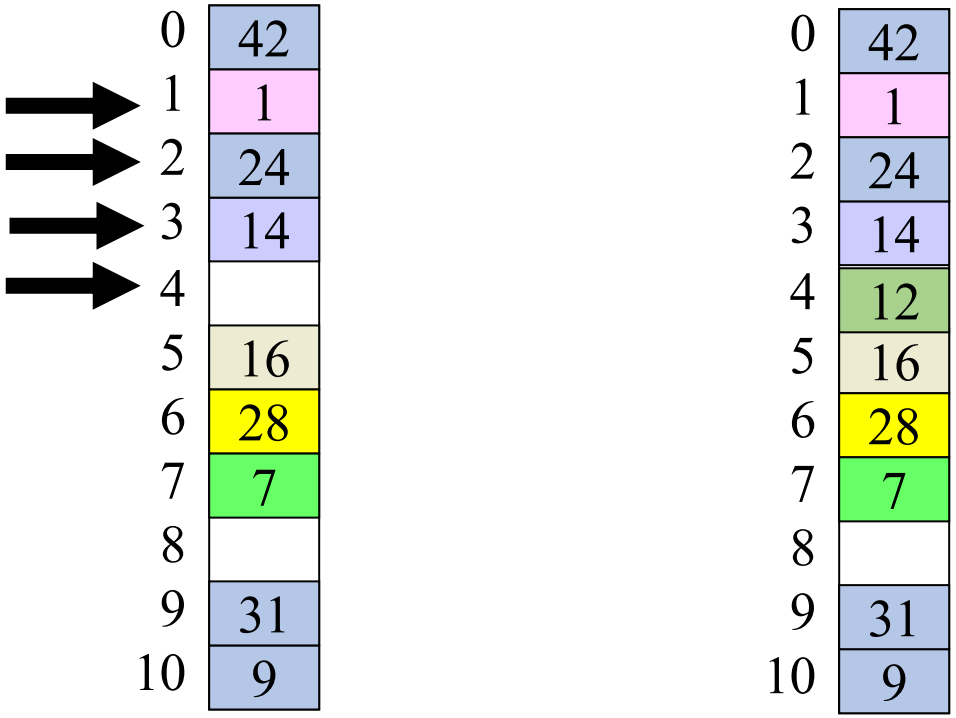
53 = 4 x 11 + 9
53 mod 11 = 9





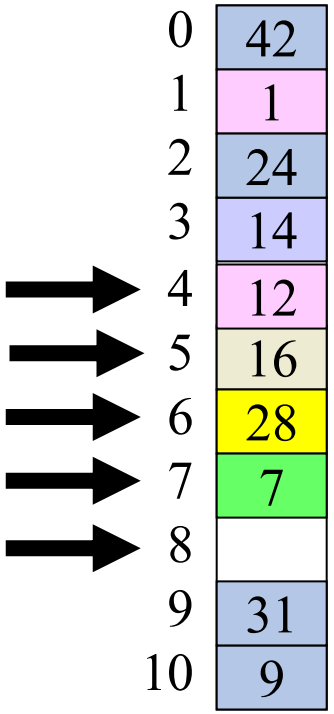
Linear Probing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1



Search with linear probing (Search 15)

15 = 1 x 11 + 4
15 mod 11 = 4



NOT FOUND !

Linear probing

If the current location is used, try the next table location

```
linear_probing_insert(K)
   if (table is full) error
   probe = h(K)
    while (table[probe] occupied)
         probe = (probe + 1) mod M
     table[probe] = K
```

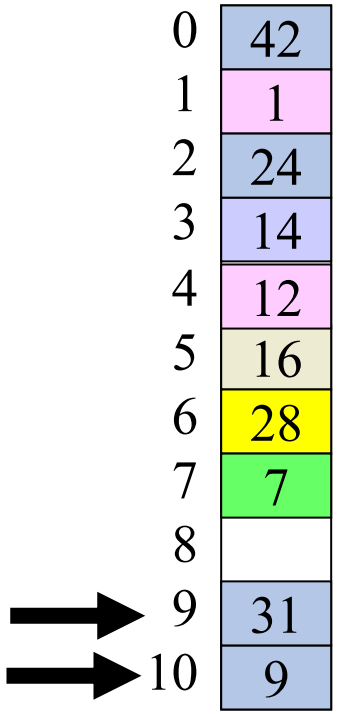
- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)
- Lookups walk along table until the key or an empty slot is found
- Uses less memory than chaining. (Don't have to store all those links)
- Slower than chaining. (May have to walk along table for a long way.)
- Deletion is more complex. **(Why?)**

Deletion in Hashing with Linear Probing

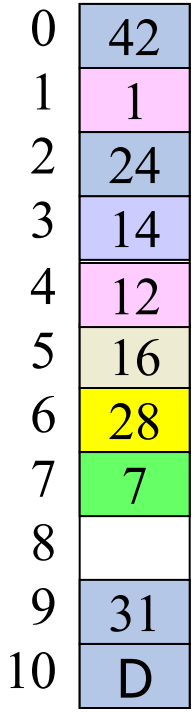
- Since **empty buckets are used to terminate search**, standard deletion does not work
- One simple idea is to not delete, but mark.
  - Insert: put item in first empty or marked bucket.
  - Search: Continue past marked buckets.
  - Delete: just mark the bucket as deleted.

Deletion with linear probing: LAZY (Delete 9)

9 = 0 x 11 + 9
9 mod 11 = 9

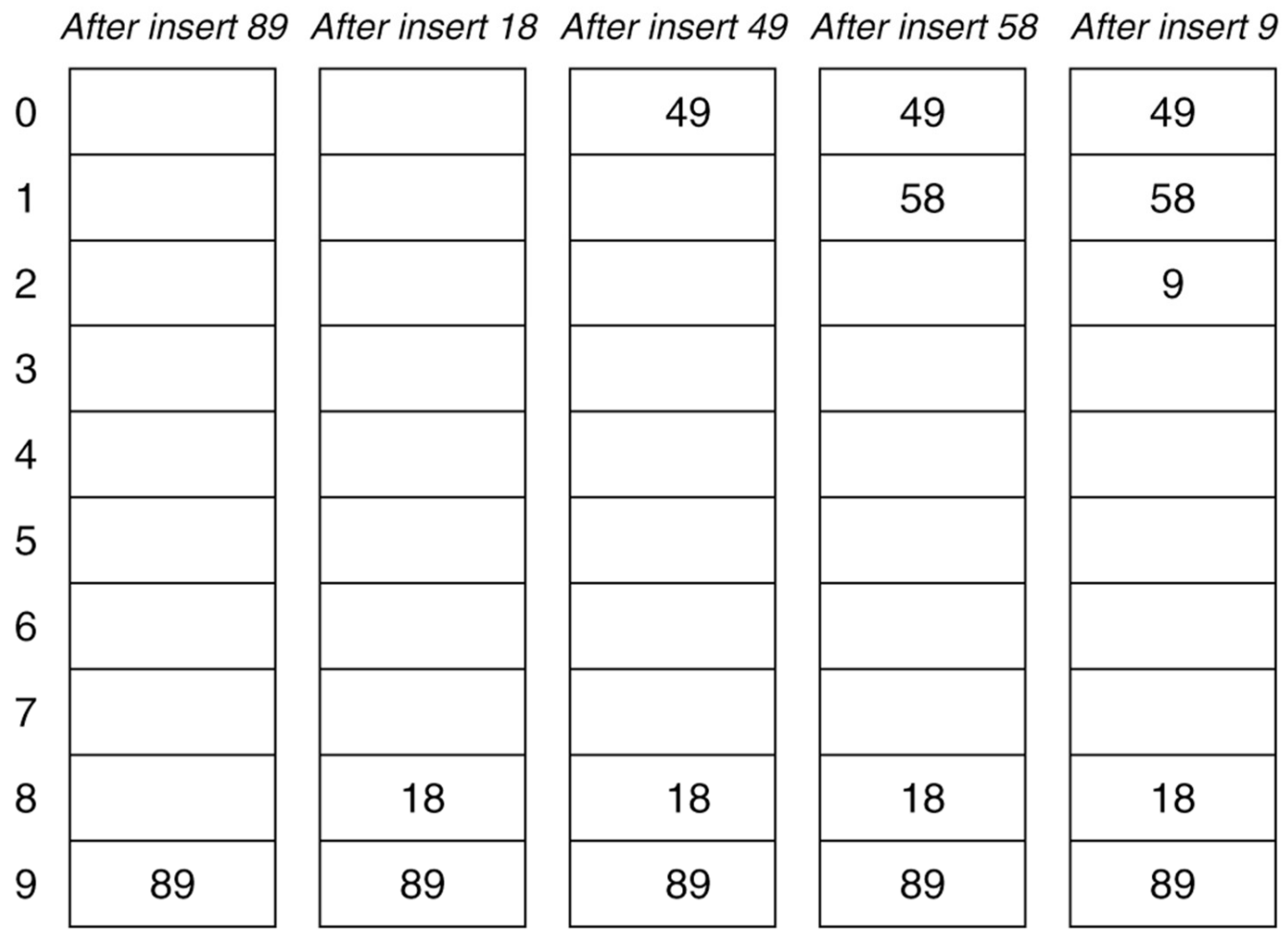


FOUND !



Example

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

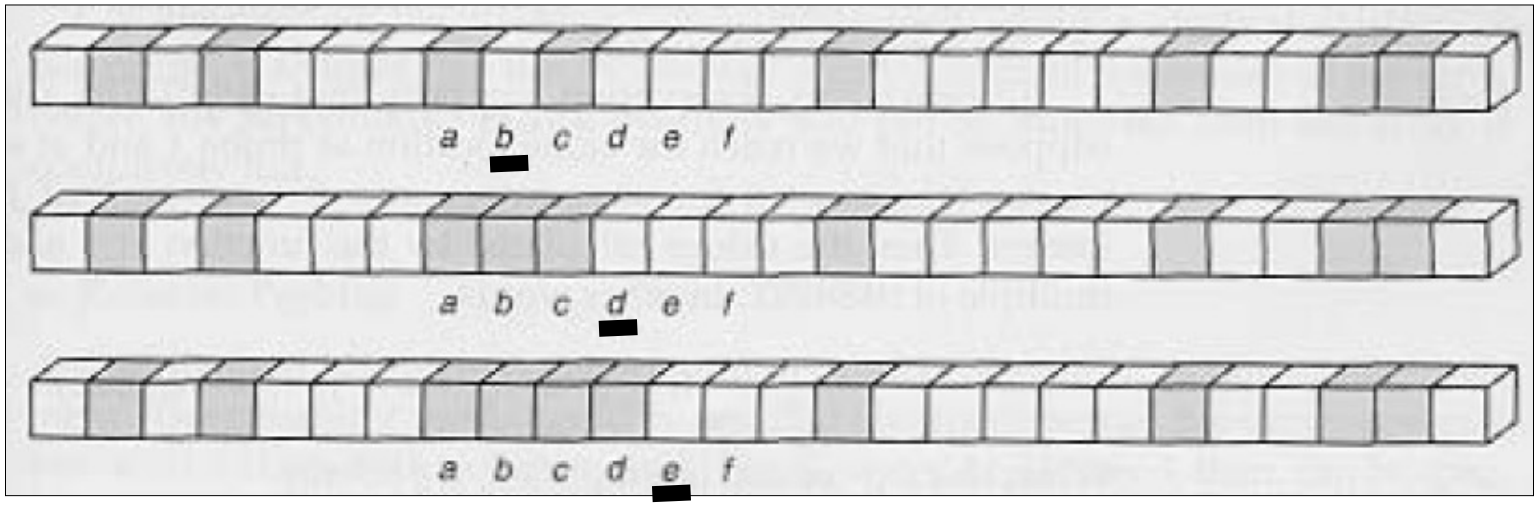


Primary Clustering Problem

- 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

Load factor with linear probing

- For *any* $\lambda < 1$, linear probing will find an empty slot
- Search cost (assuming simple uniform hashing)
  - successful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

  - unsuccessful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Performance quickly degrades for $\lambda > 1/2$

Quadratic probing

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

- 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)

Quadratic Probing

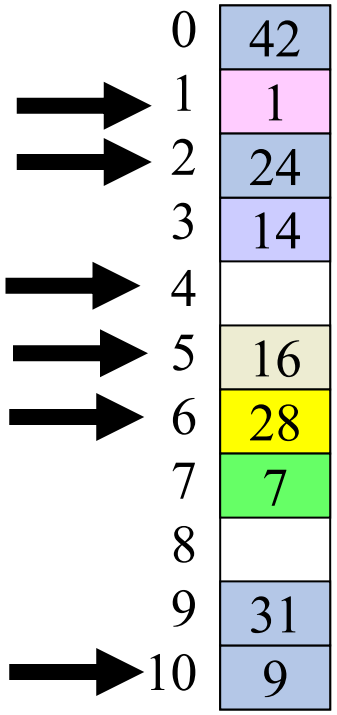
Solves the clustering problem in Linear Probing

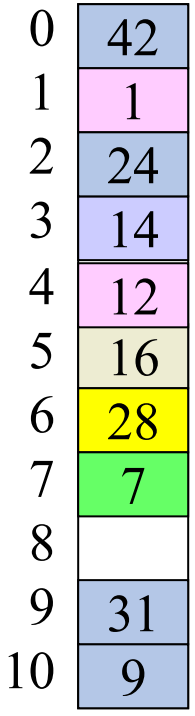
- ☐Check H(x)
- ☐If collision occurs check **H(x) + 1**
- ☐If collision occurs check **H(x) + 4**
- ☐If collision occurs check **H(x) + 9**
- ☐If collision occurs check **H(x) + 16**
- ☐...

- ☐**$H(x) + i^2$**

Quadratic Probing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1





Clustering problem

Quadratic probing issues

- Caveat: May not find a vacant cell!
  - Table must be less than half full ($\lambda < 1/2$)
- Another issue
  - Suppose the table size is 16.
  - Probe offsets that will be tried:

```
 1      mod 16  =  1
 4      mod 16  =  4
 9      mod 16  =  9
16      mod 16  =  0
25      mod 16  =  9
36      mod 16  =  4
49      mod 16  =  1
64      mod 16  =  0
81      mod 16  =  1
```

only four different values!

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Load factor in quadratic hashing

- **Theorem**: If TableSize is prime and $\lambda \leq ½$, quadratic probing *will* find an empty slot; for greater λ, *might not*
- With load factors near ½ the expected number of probes is empirically near *optimal* – no exact analysis known
- Don't get clustering from *similar* keys (**primary** clustering), still get clustering from *identical* keys (**secondary** clustering)

Double Hashing

(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,...$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum
- Good choice of $Hash_2(X)$ can guarantee does not get "stuck" as long as $\lambda < 1$

Double Hashing

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$$

Insert key 14:

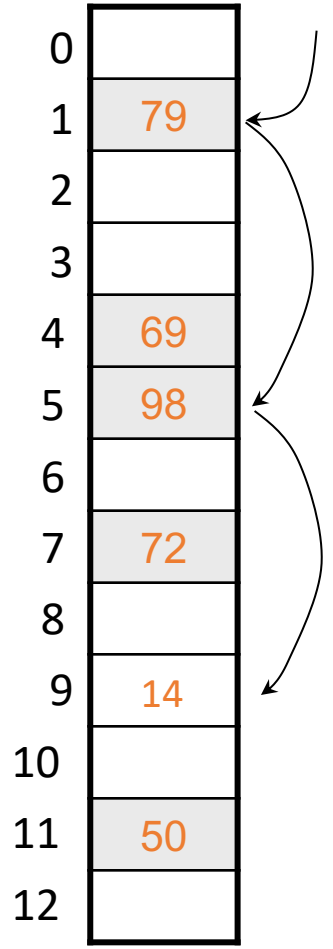
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



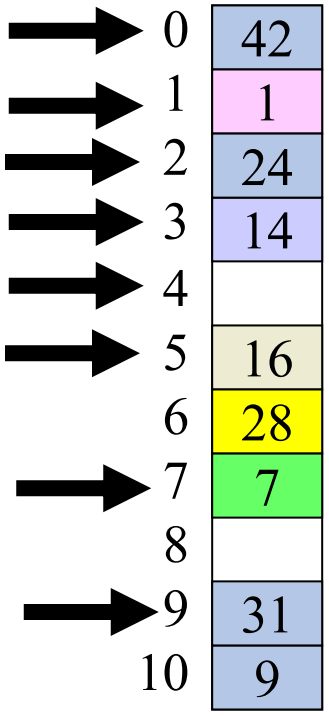
Double Hashing (another way forward)

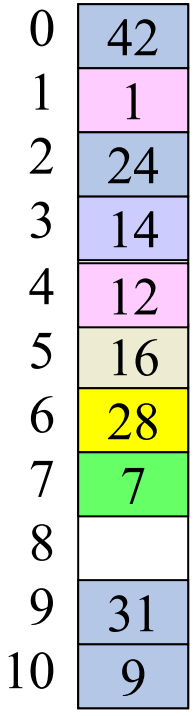
- *When collision occurs use a second hash function*
  - $Hash_2$ (x) = R – (x mod R)
  - R: greatest prime number smaller than table-size
- *Inserting 12*

  $H_2(x) = 7 – (x \bmod 7) = 7 – (12 \bmod 7) = 2$
  - Check **H(x)**
  - If collision occurs check **H(x) + 2**
  - If collision occurs check **H(x) + 4**
  - If collision occurs check **H(x) + 6**
  - If collision occurs check **H(x) + 8**
  - **$H(x) + i * H_2(x)$**

Double Hashing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1
7 –12 mod 7 = 2





Load factor of double hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost approaches optimal (random re-hash):
  - successful search: $$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$
  - unsuccessful search: $$\frac{1}{1-\lambda}$$
- No primary clustering and no secondary clustering
- Still becomes costly as λ nears 1.

The Squished Pigeon Principle

- An insert using Closed Hashing *cannot* work with a load factor of 1 or more.
  - Quadratic probing can *fail* if $\lambda > ½$
  - Linear probing and double hashing *slow* if $\lambda > ½$
  - Lazy deletion never frees space
- Separate chaining becomes slow once $\lambda > 1$
  - Eventually becomes a linear search of long chains
- How can we relieve the pressure on the pigeons?

REHASH!

Rehashing

- If table gets too full, operations will take too long.
- Build another table, twice as big (and prime).
  - Next prime number after 11 x 2 is 23
- Insert every element again to this table

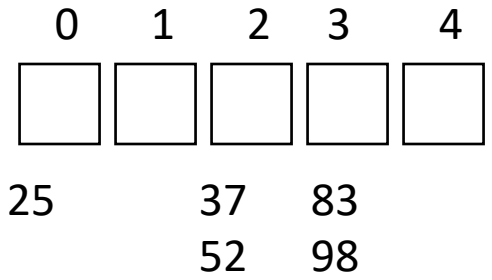
- Rehash after a percentage of the table becomes full (70% for example)

Rehashing example

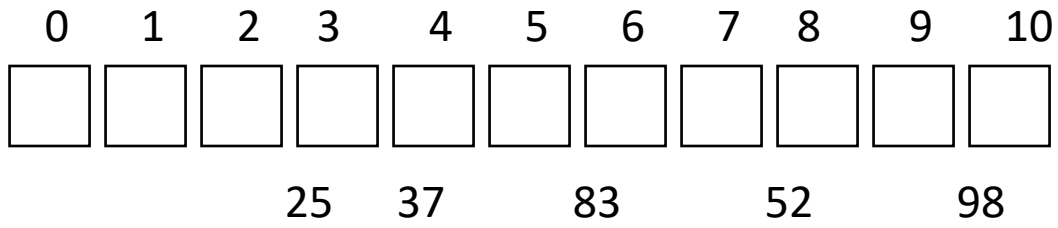
Separate chaining

$h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$

λ=1



λ=5/11



Case study

- Spelling dictionary
  - 50,000 words
  - static
  - arbitrary(ish) preprocessing time
- Goals
  - fast spell checking
  - minimal storage

- Practical notes
  - almost all searches are successful
  - words average about 8 characters in length
  - 50,000 words at 8 bytes/word is 400K
  - pointers are 4 bytes
  - there are *many* regularities in the structure of English words

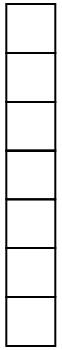
Possible Solution

- Solutions
  - sorted array + binary search
  - separate chaining
  - open addressing + linear probing

Storage

- Assume words are strings and entries are pointers to strings

Array + binary search



n pointers

Separate chaining

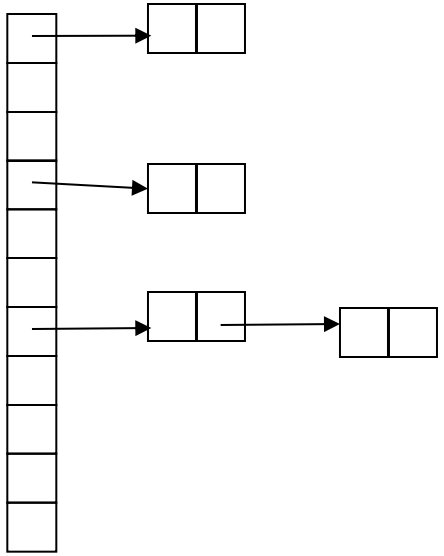


table size + 2n pointers =
$n/\lambda + 2n$

Closed hashing

n/λ pointers

Analysis

50K words, 4 bytes @ pointer

- Binary search
  - storage: n pointers + words = 200K+400K = 600K
  - time: $\log_2 n \leq 16$ probes per access, worst case
- Separate chaining - with $\lambda = 1$
  - storage: $n/\lambda + 2n$ pointers + words = 200K+400K+400K = 1GB
  - time: $1 + \lambda/2$ probes per access on average = 1.5
- Closed hashing - with $\lambda = 0.5$
  - storage: n/λ pointers + words = 400K + 400K = 800K
  - time: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ probes per access on average = 1.5

Analysis of Open Hashing

- Effort of one Insert?
  - Intuitively – that depends on how full the hash is
- Effort of an average Insert?
- Effort to fill the Bucket to a certain capacity?
  - Intuitively – accumulated efforts in inserts
- Effort to search an item (both *successful* and *unsuccessful*)?
- Effort to delete an item (both *successful* and *unsuccessful*)?
  - Same effort for successful search and delete?
  - Same effort for unsuccessful search and delete?

Summary

What do we lose?

Operations that require ordering are inefficient

FindMax:	O(n)	O(log n) Balanced binary tree
FindMin:	O(n)	O(log n) Balanced binary tree
PrintSorted:	O(n log n)	O(n) Balanced binary tree

What do we gain?

Insert:	O(1)	O(log n) Balanced binary tree
Delete:	O(1)	O(log n) Balanced binary tree
Find:	O(1)	O(log n) Balanced binary tree

How to handle Collision?

Separate chaining

Open addressing

Theory of Hashing: Birthday Paradox

- To appreciate the subtlety of hashing, first consider a puzzle: the birthday paradox.

- Suppose birth days are chance events:
  - date of birth is purely random
  - any day of the year just as likely as another

Theory of Hashing: Birthday Paradox

- What are the chances that in a group of 30 people, at least two have the same birthday?
- How many people will be needed to have at least 50% chance of same birthday?
- It's called a paradox because the answer appears to be counter-intuitive.
- There are 365 different birthdays, so for 50% chance, you expect at least 182 people.

Birthday Paradox: the math

- Suppose 2 people in the room.
- What is the prob. that they have the same birthday?
- Answer is 1/365.
  - All birthdays are equally likely, so B's birthday falls on A's birthday 1 in 365 times.

- Now suppose there are k people in the room.
- It's more convenient to calculate the prob. X that no two have the same birthday.

- Our answer will be the (1 – X)

Birthday Paradox

- Define P_i = prob. that first i all have distinct birthdays
- For convenience, define p = 1/365
  - $P_1 = 1.$
  - $P_2 = (1 – p)$
  - $P_3 = (1 – p) * (1 – 2p)$
  - $P_k = (1 – p) * (1 – 2p) * …. * (1 – (k-1)p)$
- You can now verify that for k=23, $P_k <= 0.4999$
- That is, with just 23 people in the room, there is more than 50% chance that two have the same birthday

Birthday Paradox: derivation

- Use $1 - x <= e^{-x}$, for all x
- Therefore, $1 - j*p <= e^{-jp}$
- Also, $e^x + e^y = e^{x+y}$

- Therefore, $P_k <= e^{(-p\ -2p\ -3p\ \ldots\ -(k-1)p)}$
- $P_k <= e^{-k(k-1)p/2}$

- For k = 23, we have k(k-1)/2*365 = 0.69
- $e^{-0.69} <= 0.4999$

- Connection to Hashing:
  - Suppose n = 23, and hash table has size M = 365.
  - 50% chance that 2 keys will land in the same bucket.

Why choose table size as prime?