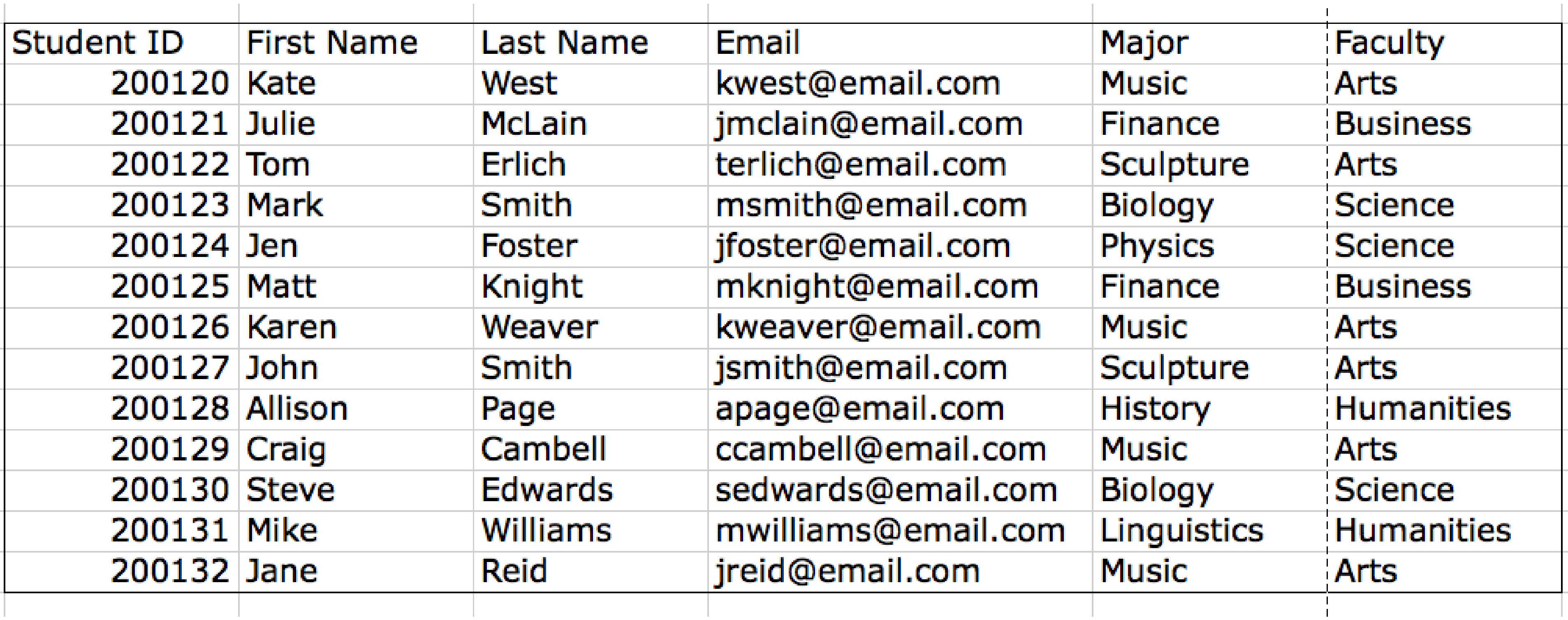
National University of Computer & Emerging Sciences

Hashing

What is a database?

- Structured collection of data.



Student ID	First Name	Last Name	Email	Major	Faculty
200120	Kate	West	kwest@email.com	Music	Arts
200121	Julie	McLain	jmclain@email.com	Finance	Business
200122	Tom	Erlich	terlich@email.com	Sculpture	Arts
200123	Mark	Smith	msmith@email.com	Biology	Science
200124	Jen	Foster	jfoster@email.com	Physics	Science
200125	Matt	Knight	mknight@email.com	Finance	Business
200126	Karen	Weaver	kweaver@email.com	Music	Arts
200127	John	Smith	jsmith@email.com	Sculpture	Arts
200128	Allison	Page	apage@email.com	History	Humanities
200129	Craig	Cambell	ccambell@email.com	Music	Arts
200130	Steve	Edwards	sedwards@email.com	Biology	Science
200131	Mike	Williams	mwilliams@email.com	Linguistics	Humanities
200132	Jane	Reid	jreid@email.com	Music	Arts

The Dictionary ADT

- a dictionary (table) is an abstract model of a database
- a dictionary stores key-element pairs
- the main operation supported by a dictionary is searching by key

Examples

- Telephone directory
- Library catalogue
- Books in print: key ISBN

- Any other example?

Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

Main Issues

- Size
- Operations: search, insert, delete
- What will be stored in the dictionary?
- How will be items identified?

The Dictionary ADT

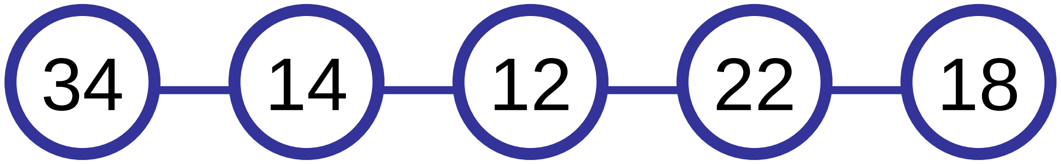
- simple container methods:
  - size()
  - isEmpty()
  - elements()
- query methods:
  - findElement(k)
  - findAllElements(k)

The Dictionary ADT

- update methods:
  - insertItem(k, e)
  - removeElement(k)
  - removeAllElements(k)

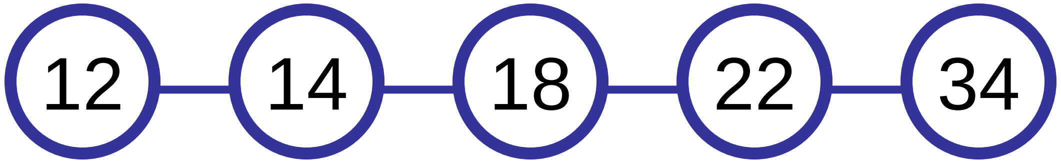
Implementing a Dictionary

- ***unordered sequence***
  - searching and removing takes O(n) linear time
  - inserting takes O(1) constant time
  - applications to log files (frequent insertions, rare searches and removals) 34 14 12 22 18



Implementing a Dictionary

- ***array-based ordered sequence***
  (assumes keys can be ordered)
  - searching takes O(log n) time ***(binary searc**h)*
  - inserting and removing takes O(n) time
  - application to look-up tables
  (frequent searches, rare insertions and removals)



A case scenario

- Relenor is a large phone company, and they want to provide enhanced caller ID capability:
  - given a phone number, return the caller’s name
  - phone numbers are in the range 0 to $R = 10^{10}–1$
  - n is the number of phone numbers used
  - want to do this as efficiently as possible

A case scenario

- We know two ways to design this dictionary:
- ***A balanced search tree*** (AVL, red-black) has O(log n) query time and O(n) space --- good space usage and search time, but can we reduce the search time to constant?
- Direct Addressing

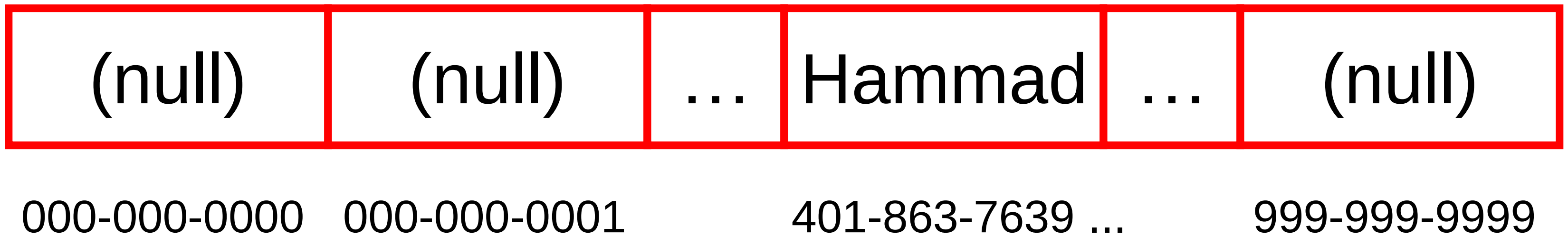
Direct Addressing

- 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

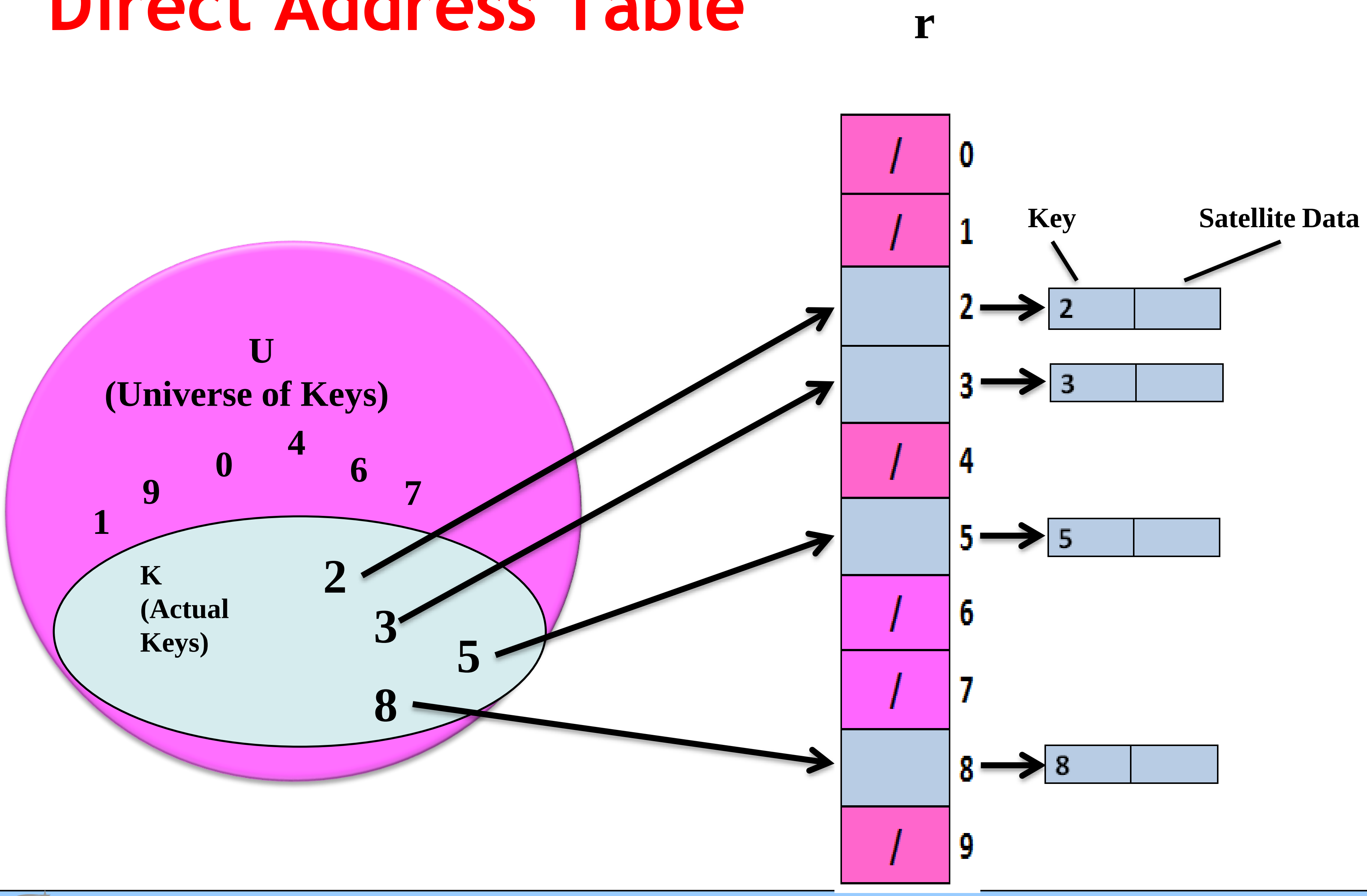
- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented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

Direct Addressing

- Each cell is thought of as a bucket or a container
  - Holds key element pairs
  - In array A of size N, an element *e* with key *k* is inserted in *A[k]*.



Direct Address Table



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: $O(1)$

Advantages with Direct Addressing

- Direct Addressing is the most efficient way to access the data since
  - It takes only single step for any operation on direct address table.
  - It works well when the Universe U of keys is reasonable small.

Difficulty with Direct Addressing

When the universe *U* is very large…

- Storing a table *T* of size *U* may be impractical, given the memory available on a typical computer.
- The set *K* of the keys actually stored may be so small relative to *U* that most of the space allocated for *T* would be wasted.

An Example

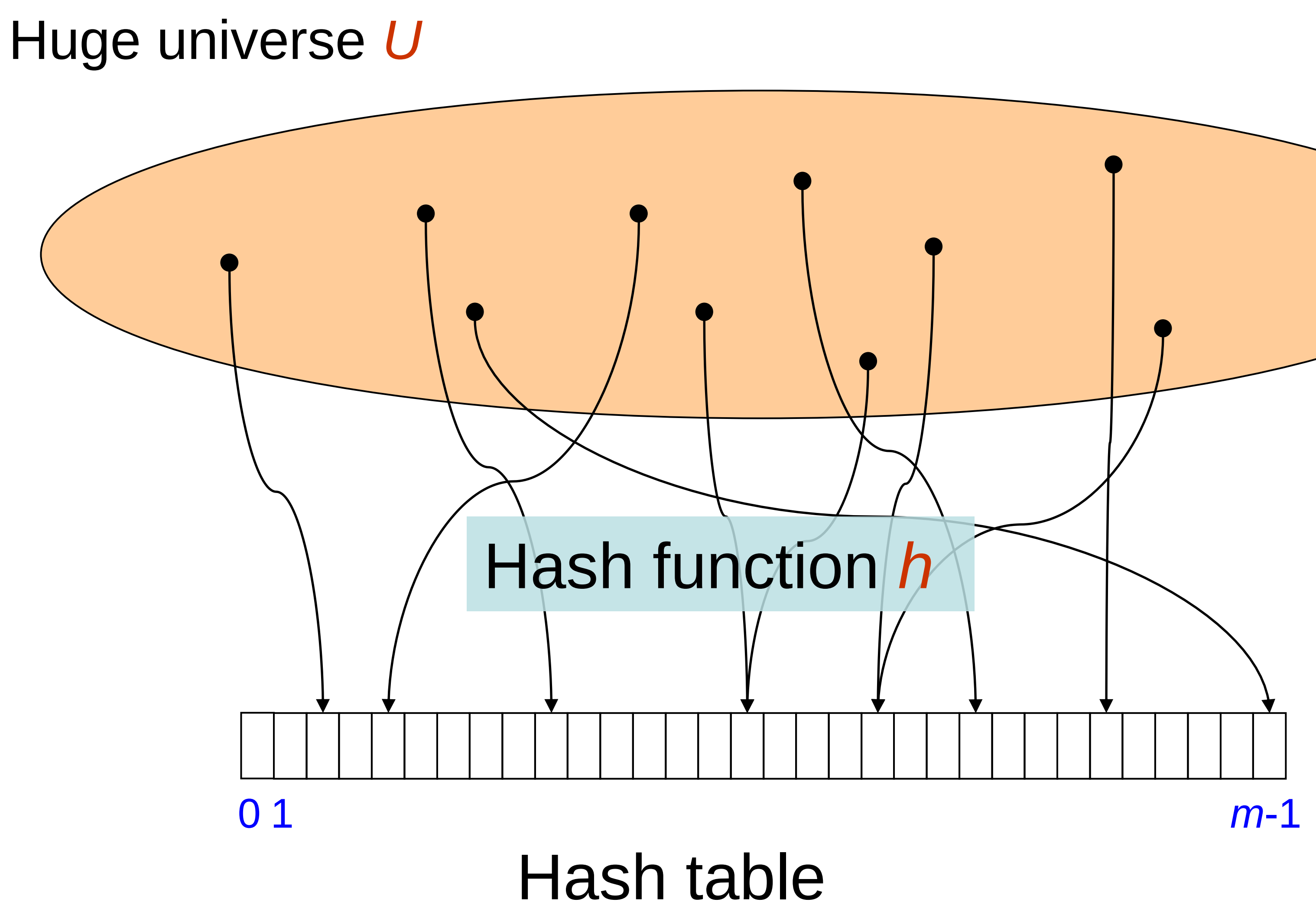
- A table, 50 students in a class.
- The key, 9 digit *SSN*, used to identify each student.
- Number of different 9 digit number=10^9
- The fraction of actual keys needed. $50/10^9$, **<u>0.000005%</u>**
- Percent of the memory allocated for table wasted, **<u>99.999995%</u>**

Hashing

- Suppose we were to come up with a “magic function” that, given a value to search for, would tell us exactly where in the array to look
  - If it’s in that location, it’s in the array
  - If it’s not in that location, it’s not in the array

- This function would have no other purpose
- This function is called a hash function because it “makes hash” of its inputs

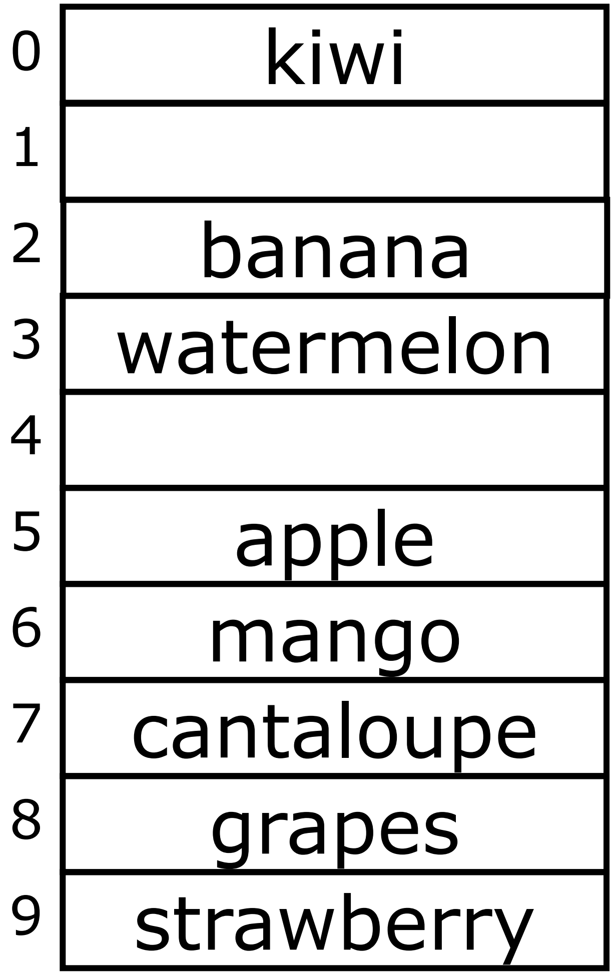
Hashing



Example (ideal) hash function

- Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```



Finding the hash function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function ☹
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash function would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to *start* looking!

Example imperfect hash function

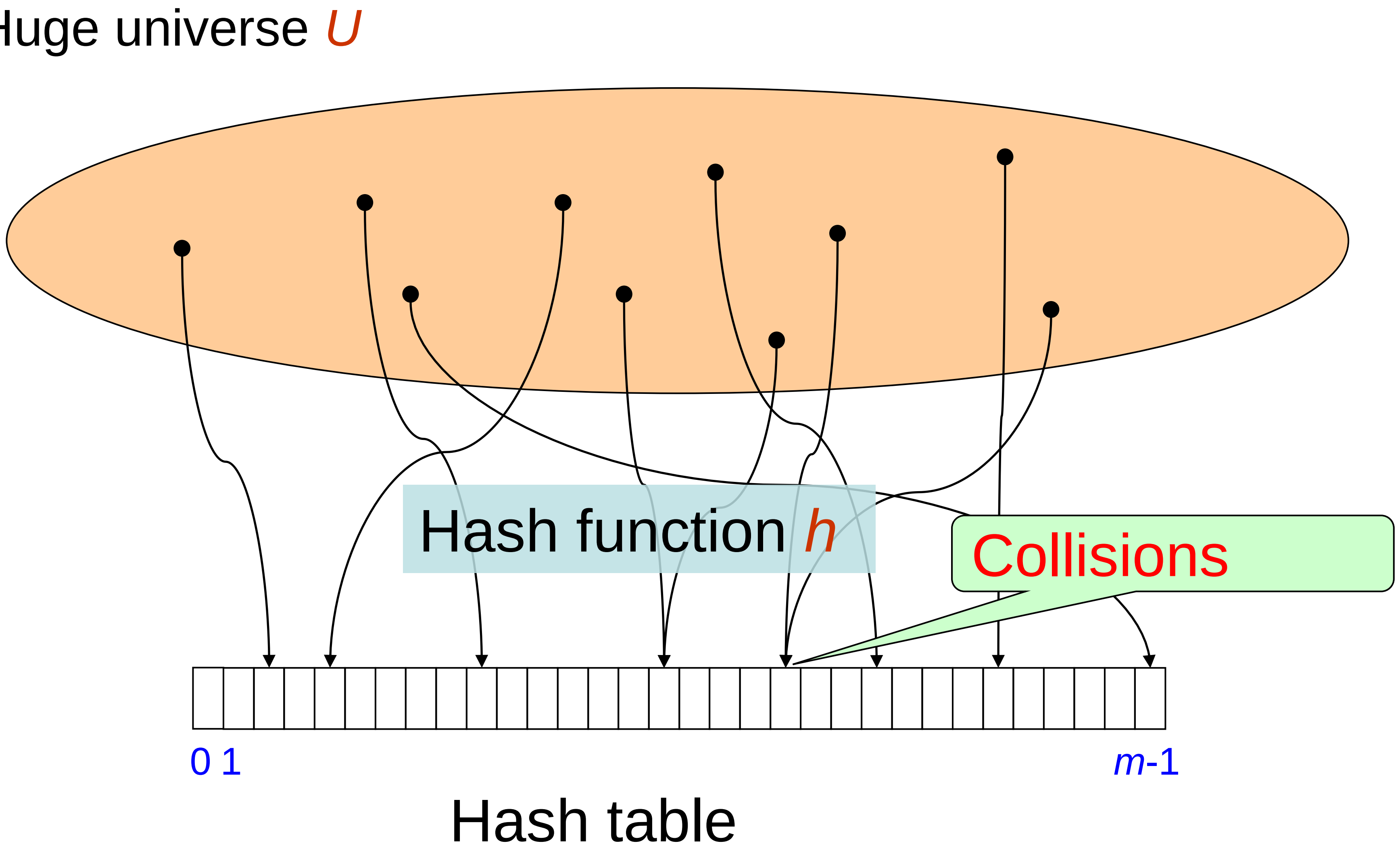
- Suppose our hash function gave us the following values:
  - hash("apple") = 5
    hash("watermelon") = 3
    hash("grapes") = 8
    hash("cantaloupe") = 7
    hash("kiwi") = 0
    hash("strawberry") = 9
    hash("mango") = 6
    hash("banana") = 2
    hash("honeydew") = 6
- Now what?

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

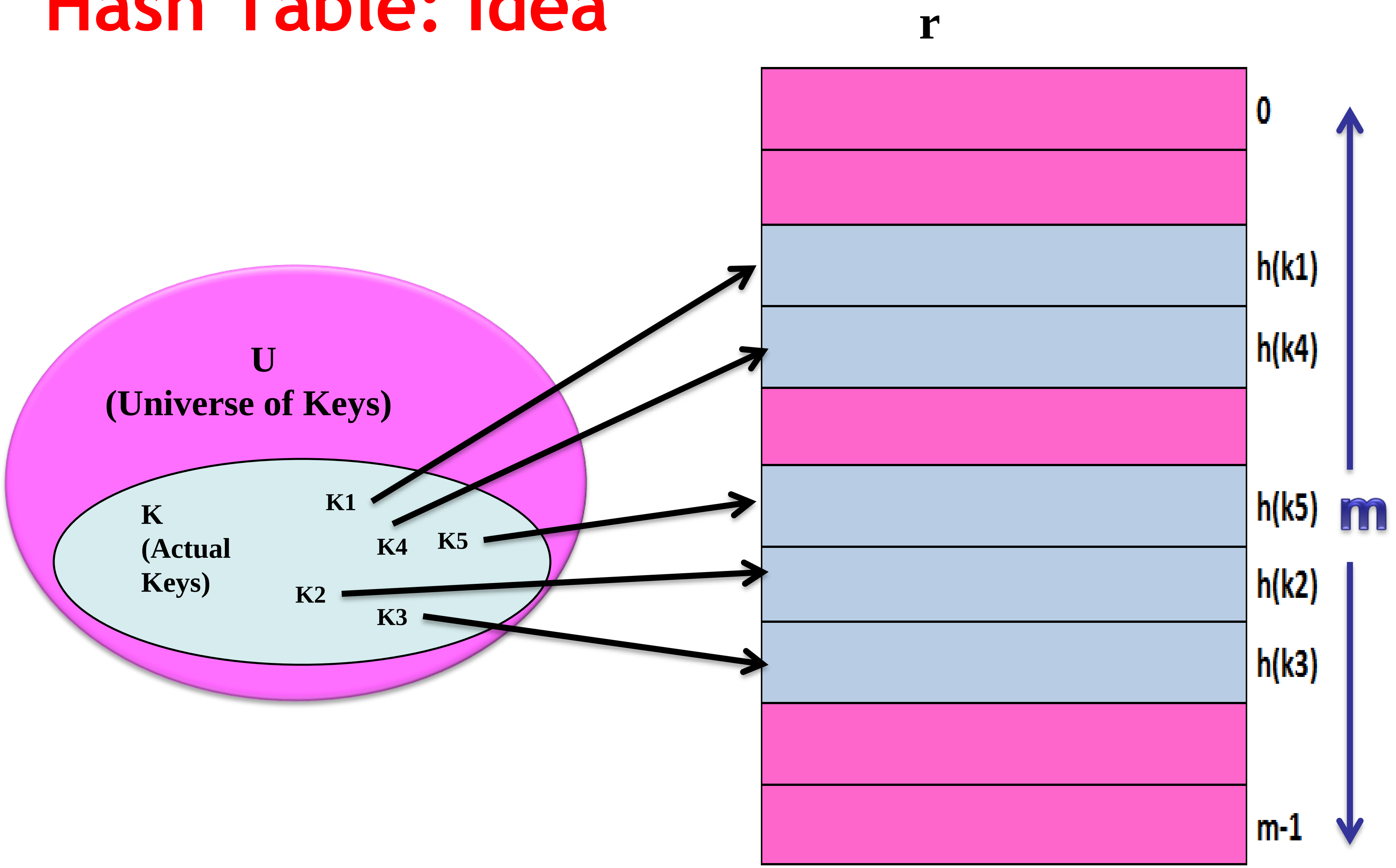
Collisions

- When two values hash to the same array location, this is called a collision
- Collisions are normally treated as "first come, first served"—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Hashing

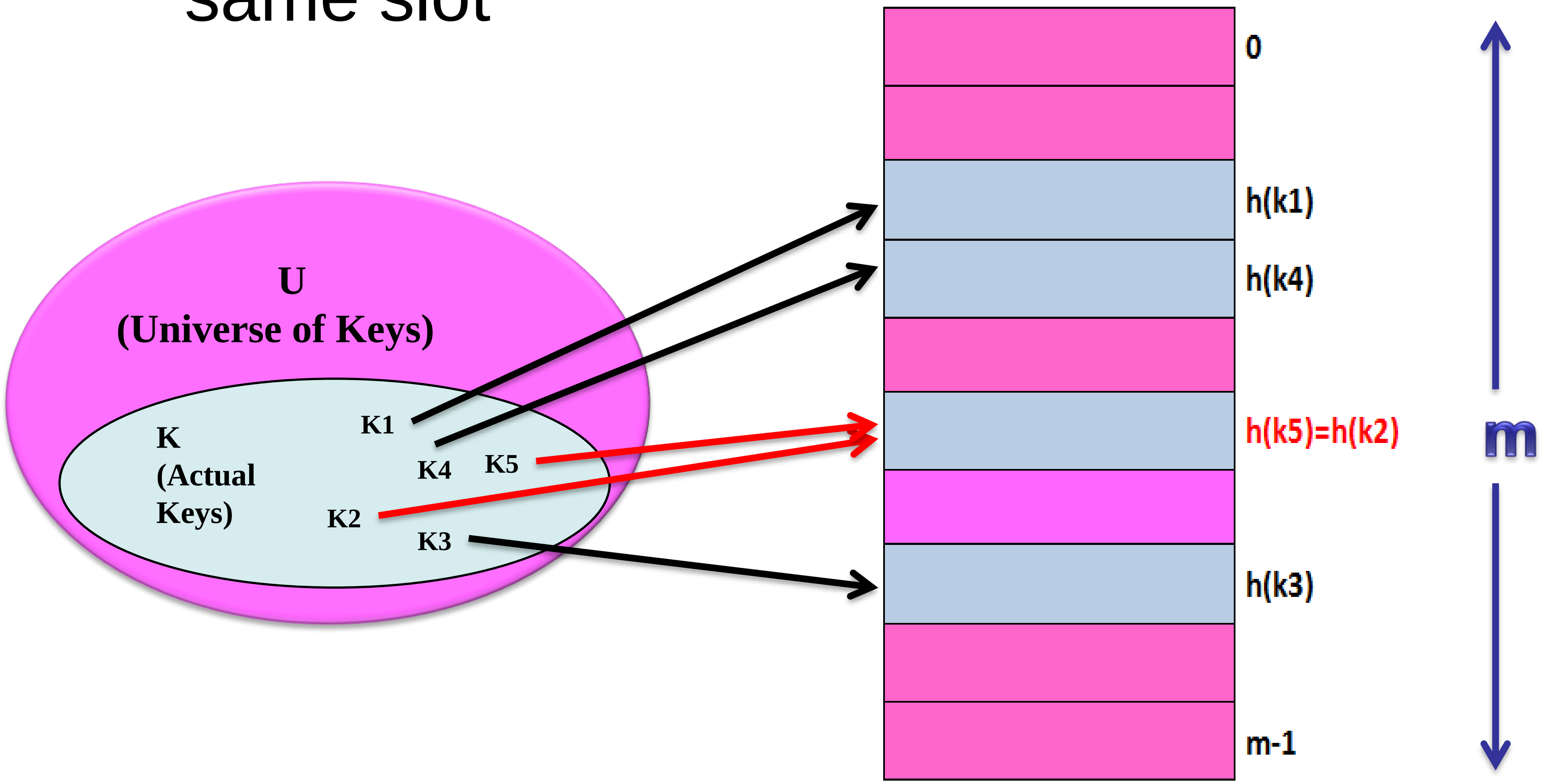


Hash Table: Idea



Hash Table: Collision

- Collision: Two or more keys hash to the same slot



Compared to direct addressing

- Advantage: Requires less storage and runs in *O(1)* time.
- Comparison

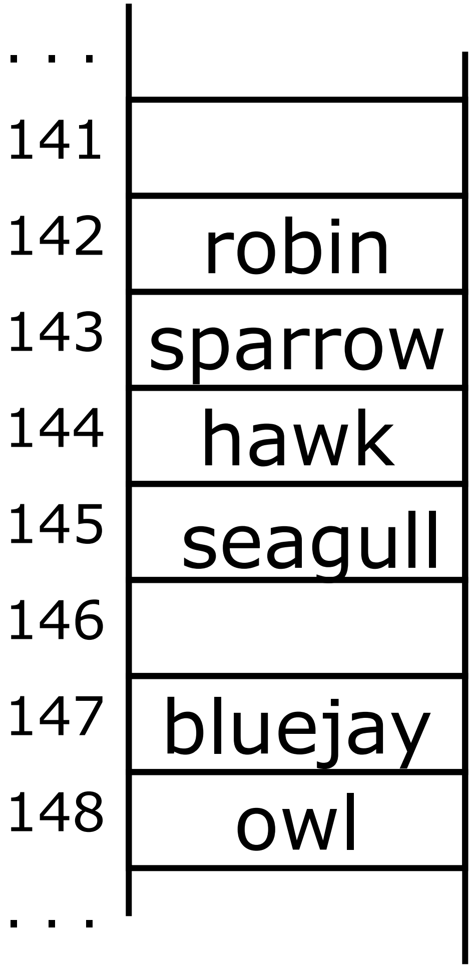
	Storage Space	Storing k
Direct Addressing	**|U|**	**Store in slot k**
Hashing	**m**	**Store in slot h(k)**

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1:** Search from there for an empty location
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2:** Use a second hash function
    - ...and a third, and a fourth, and a fifth, ...
  - **Solution #3:** Use the array location as the header of a linked list of values that hash to this location
- All these solutions work, provided:
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

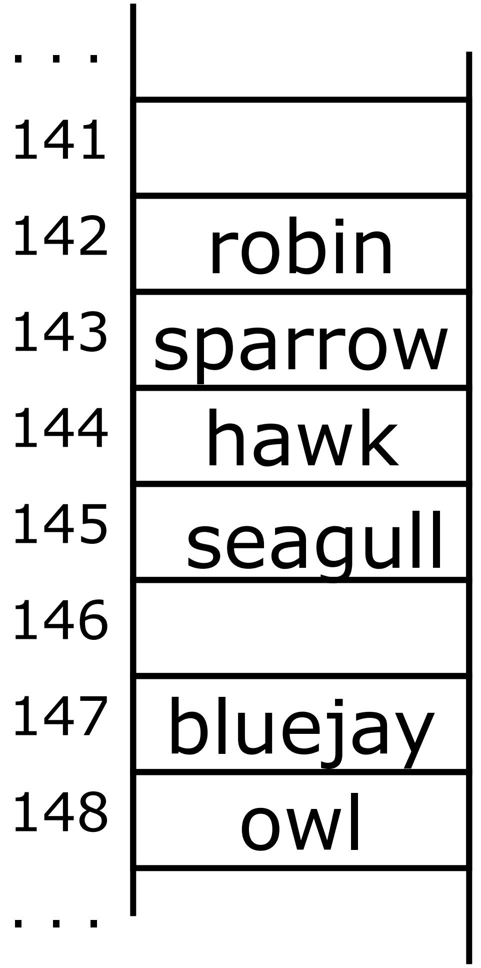
Insertion, I

- Suppose you want to add seagull to this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is empty
- Therefore, put seagull at location 145



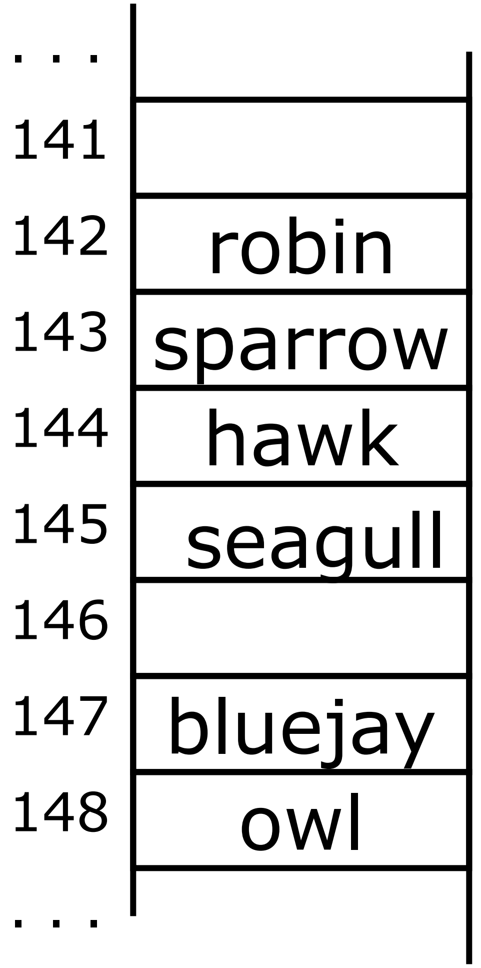
Searching, I

- Suppose you want to look up seagull in this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is not empty
  - table[145] == seagull !
- We found seagull at location 145



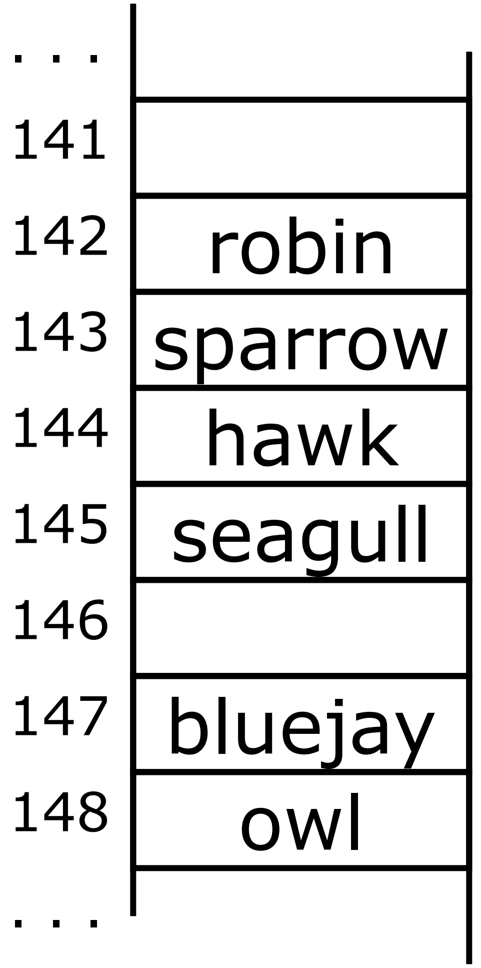
Searching, II

- Suppose you want to look up cow in this hash table
- Also suppose:
  - hashCode(cow) = 144
  - table[144] is not empty
  - table[144] != cow
  - table[145] is not empty
  - table[145] != cow
  - table[146] is empty
- If cow were in the table, we should have found it by now
- Therefore, it isn't here



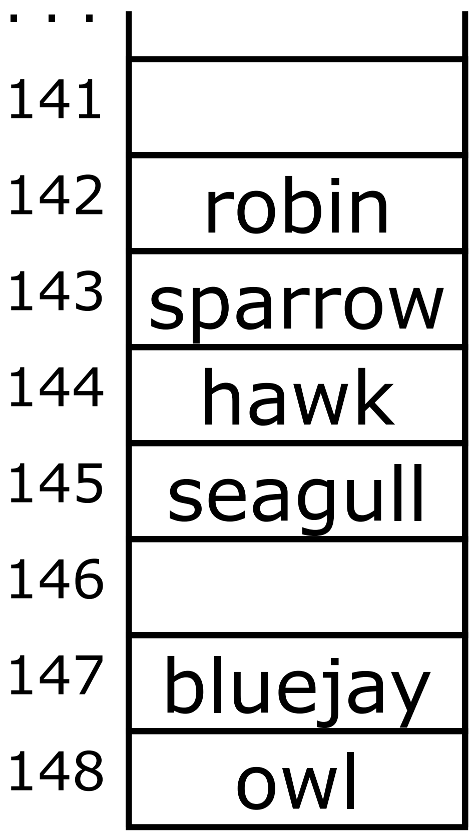
Insertion, II

- Suppose you want to add hawk to this hash table
- Also suppose
  - hashCode(hawk) = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- hawk is already in the table, so do nothing



Insertion, III

- Suppose:
  - You want to add cardinal to this hash table
  - hashCode(cardinal) = 147
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as circular; after 148 comes 0
  - Hence, cardinal goes in location 0 (or 1, or 2, or ...)



Clustering

- One problem with the above technique is the tendency to form “clusters”
- A cluster is a group of items not containing any open slots
- The bigger a cluster gets, the more likely it is that new values will hash into the cluster, and make it ever bigger
- Clusters cause efficiency to degrade
- Here is a *non*-solution: instead of stepping one ahead, step n locations ahead
  - The clusters are still there, they’ re just harder to see
  - some table locations may be never checked

Solution #2: Rehashing

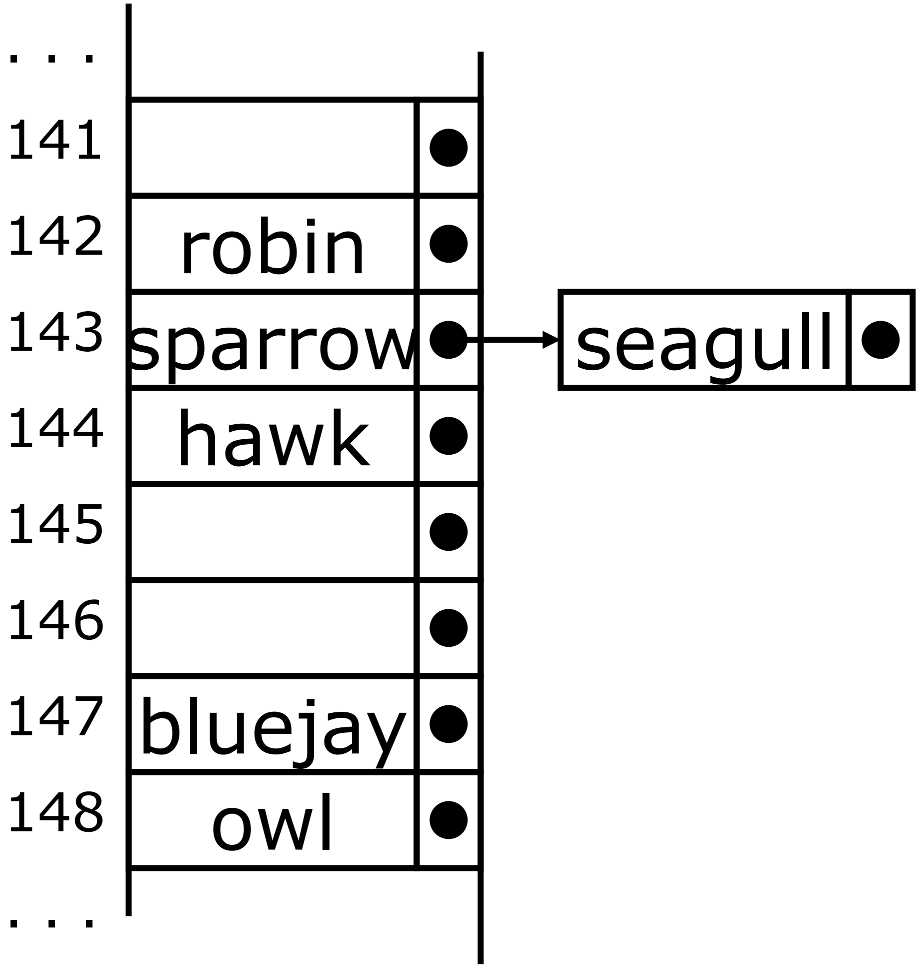
- In the event of a collision, another approach is to rehash: compute another hash function
  - Since we may need to rehash many times, we need an easily computable sequence of functions

Solution #2: Rehashing

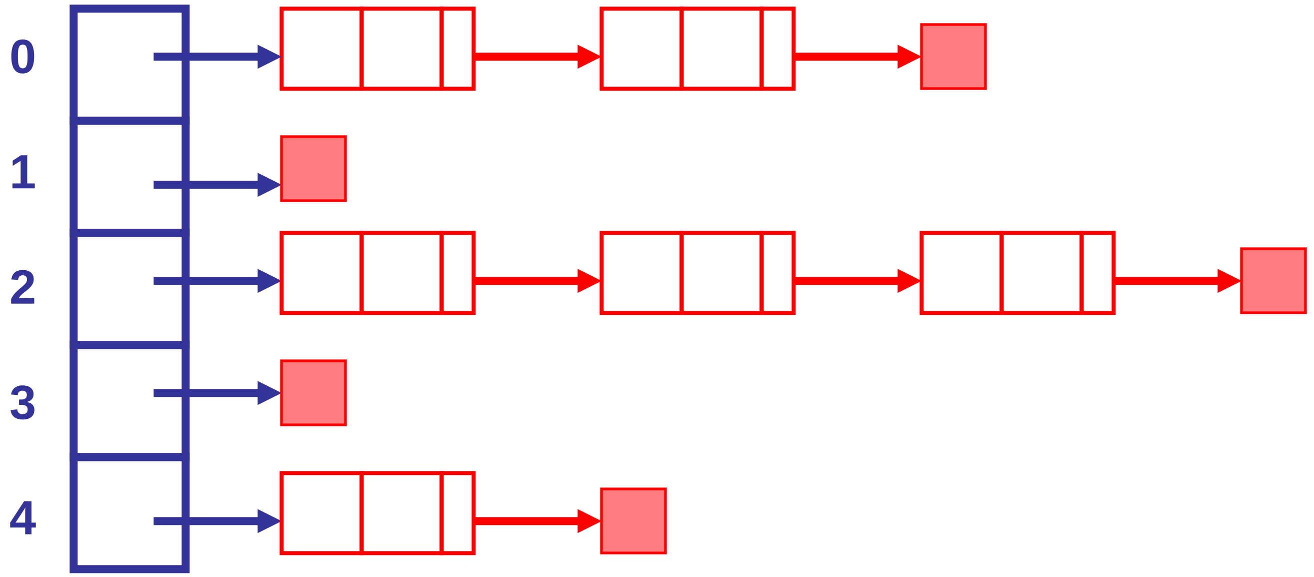
- Simple example: in the case of hashing Strings, we might take the previous hash code and add the length of the String to it
  - Probably better if the length of the string was not a component in computing the original hash function
- Possibly better yet: add the length of the String plus the number of probes made so far
  - Problem: are we sure we will look at every location in the array?
- Rehashing is a fairly uncommon approach, and we won't pursue it any further here

Solution #3: Chaining

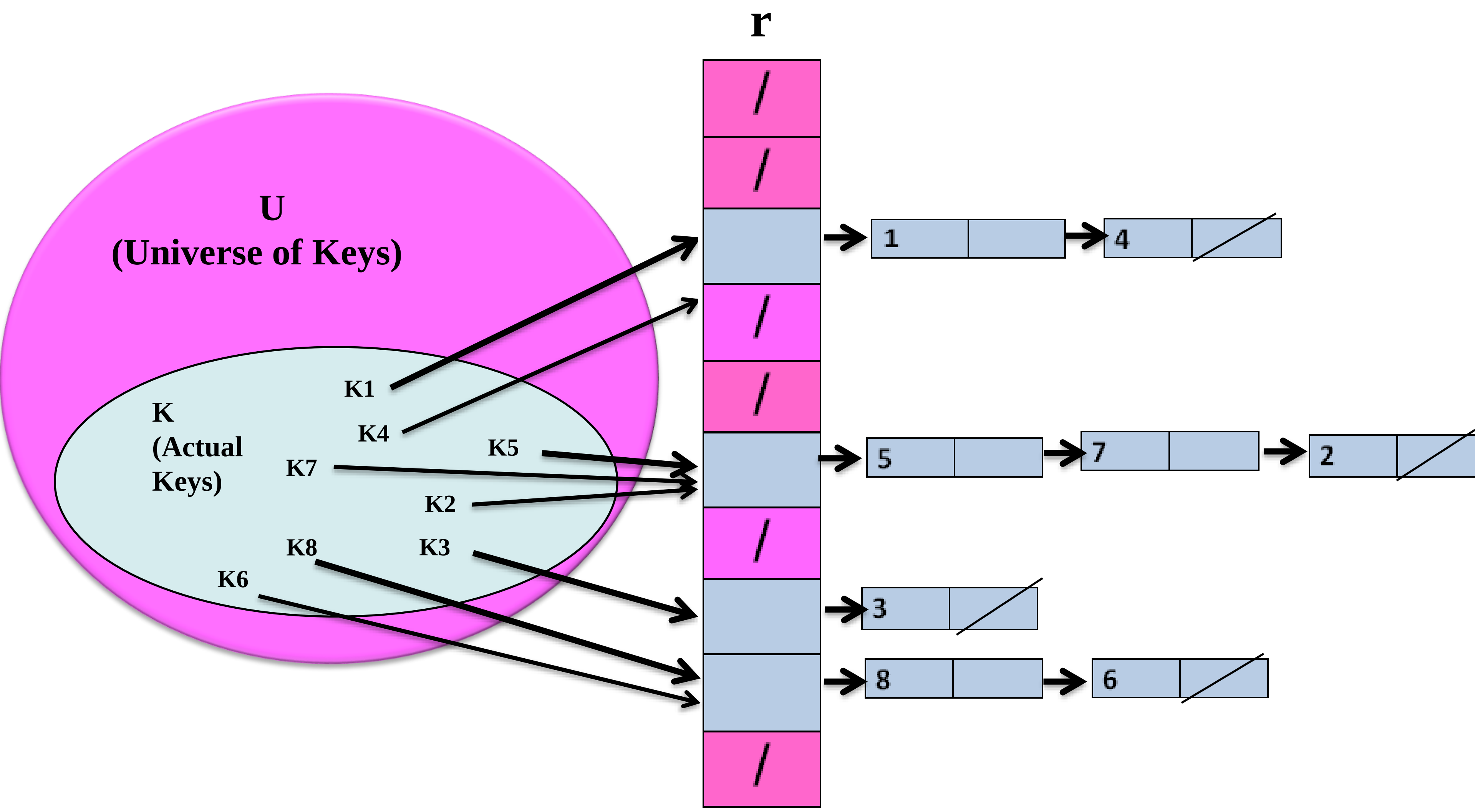
- The previous solutions used open address hashing: all entries went into a "flat" (unstructured) array
- Another solution is to make each array location the header of a linked list of values that hash to that location



Chaining

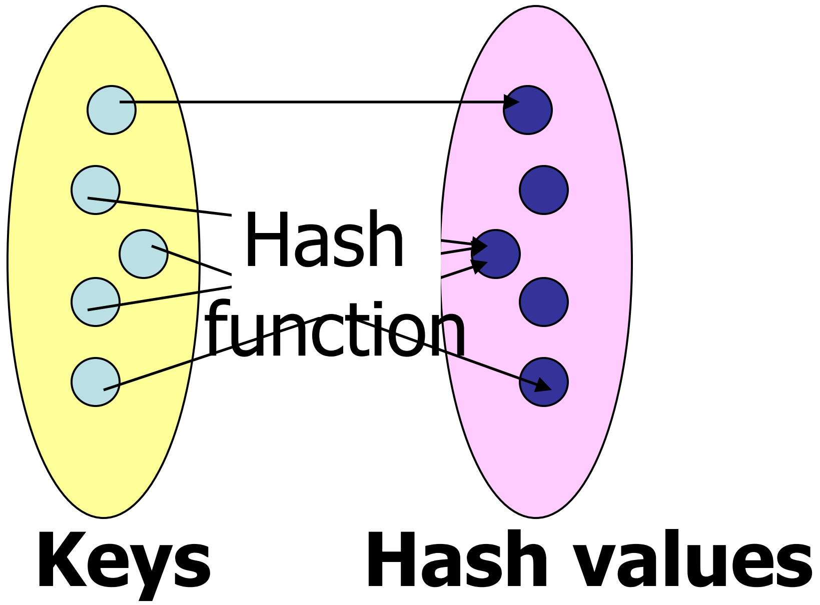


Collision By Chaining



What is a Hash function?

A hash function is a **mapping** between a set of input values (**Keys**) and a set of integers, known as **hash values**.



Choosing a good hash-function is crucial !!

The properties of a good hash function

- Rule1: The hash value is fully determined by the data being hashed.
- Rule2: The hash function uses all the input data.
- Rule3: The hash function uniformly distributes the data across the entire set of possible hash values.
- Rule4: The hash function generates very different hash values for similar strings.

An example of a hash function

```
int hash(char *str, int table_size)
{
   int sum=0;

  //sum up all the characters in the string
        for(;*str; str++)
              sum+=*str
                //return sum mod table_size
  return sum%table_size;
}
```

Analysis of example

- Rule1: Satisfies, the hash value is fully determined by the data being hashed, the hash value is just the sum of all input characters.
- Rule2: Satisfies, Every character is summed.

Analysis of example (contd.)

- Rule3: Breaks, from looking at it, it is not obvious that it doesn't uniformly distribute the strings, but if you were to analyze this function for larger input string, you will see certain statistical properties which are bad for a hash function.

- Rule4: Breaks, hash the string "CAT", now hash the string "ACT", they are the same, a slight variation in the string should result in different hash values, but with this function often they don't.

Methods to create hash functions

- Division method
- Multiplication method

Division method

The division method requires two steps.

1. The key must be transformed into an integer.
2. The value must be telescoped into range 0 to m-1

Division method…

- We map a key *k* into one of the *m* slots by taking the remainder of *k* divided by m, so the hash function is of form

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

- For example , if *m=12*, key is *100* then *h(k)=100 mod 12= 4*.

- Advantage?

The Division Method

- Advantage:
  - fast, requires only one operation
- Disadvantage:
  - Certain values of m are bad (i.e., collisions), e.g.,
    - power of 2
    - non-prime numbers

Division Method - Example

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

- $p = 1 \Rightarrow m = 2$

  $\Rightarrow h(k) = \{0, 1)$, select least significant 1 bit of k

- $p = 2 \Rightarrow m = 4$

$\Rightarrow h(k) = \{0,1,2,3\}$, select least significant 2 bits of k

***k* mod 2^8 selects these bits**

0110010111000011010

Division Method - Example

- All combinations are not generally equally likely
- Prime numbers not close to 2^n seem to be good choices
- eg want ~4000 entry table, choose $m = 4093$ ($2^{12} = 4096$)

***k* mod 2^8 selects these bits**

0110010111000001 1010

Restrictions on value of m

M should not be a power of 2, since if $m=2^p$ then h(k) is just the p lowest order bits of k.

Disadvantage!

Key	Binary	K mod 8
8	1000	0
7	111	7
12	1100	4
34	100010	2
56	111000	0
78	1001110	6
90	1011010	2
23	10111	7
45	101101	5
67	1000011	3

Division Method - Example

- Power of 10 should be avoided, if application deals with decimal numbers as keys.
- Choose *m* to be a prime,
  - Column 2: k mod 97 (Prime)
  - Column 3: k mod 100 (non- prime)
- Good values of *m* are primes not close to the exact powers of 2 (or 10).

	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

Restrictions on value of m

- Unless it is known that probability distribution on keys makes all lower order p-bit patterns equally likely,

- It is better to make the hash function dependent on all the bits of the key.

The Multiplication Method

Idea:

(1) Multiply key k by a constant A, where $0 < A < 1$

(2) Extract the fractional part of kA

(3) Multiply the fractional part by m (hash table size)

(4) Truncate the result to get result in the range 0 ..m-1

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

$$e.g., \lfloor 12.3 \rfloor = 12$$

- Disadvantage: Slower than division method
- Advantage: Value of m is not critical

Multiplication Method – Example 1

- Suppose k=6 , A=0.3, m=32
- (1) k x A = 1.8
- (2) fractional part: $1.8 - \lfloor 1.8 \rfloor = 0.8$
- (3) m x 0.8 = 32 x 0.8 = 25.6
- (4) $\lfloor 25.6 \rfloor = 25$ h(6)=25

More on multiplication method

- Choose $m = 2^P$
- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\ (kA - \lfloor kA \rfloor) \rfloor$
- Value of A should not be close to 0 or 1
- *Knuth* says good value of A is 0.618033
- If $k=123456$, $m=10000$, *and* A as above

$$h(k)= \lfloor 10000*(123456*A- \lfloor 123456*A \rfloor)$$
$$= \lfloor 10000*0.0041151 \rfloor$$
$$=41$$

Hashing with Open Addressing

- So far we have studied hashing with chaining, using a linked-list to store keys that hash to the same location.

- Maintaining linked lists involves using pointers which is complex and inefficient in both storage and time requirements.

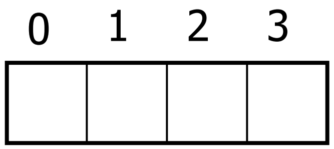
- Another option is to store all the keys directly in the table. This is known as open addressing, where collisions are resolved by systematically examining other table indexes, $i_0, i_1, i_2, \ldots$ until an empty slot is located.

Open addressing

- Another approach for collision resolution.
- All elements are stored in the hash table itself (so no pointers involved as in chaining).
- To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
- To search: follow same sequence of probes as would be used when inserting the element

Open Addressing

- The key is first mapped to a slot:
  $\text{index} = i_0 = h_1(k)$
- If there is a collision subsequent probes are performed:
  $$i_{j+1} = (i_j + c) \bmod m \quad \text{for} \quad j \geq 0$$
- If the offset constant, *c* and *m* are not relatively prime, we will not examine all the cells. Ex.:
  – Consider m=4 and c=2, then only every other slot is checked.
  When *c*=1 the collision resolution is done as a linear search. This is known as linear probing.



Insertion in hash table

```
HASH_INSERT($T$,$k$)
1   $i \leftarrow 0$
2   repeat $j \leftarrow h(k,i)$
3         if $T[j] = NIL$
4               then $T[j] = k$
5                return $j$
6         else $i \leftarrow i + 1$
7   until $i = m$
8   error “ hash table overflow”
```

Searching from Hash table

```
HASH_SEARCH($T$,$k$)
1       $i \leftarrow 0$
2   repeat $j \leftarrow h(k,i)$
3           if T[$j$] = $k$
4                   then return $j$
5           $i \leftarrow i + 1$
6   until T[$j$] = $NIL$ or $i = m$
7   return $NIL$
```

Analysis

- Worst case for inserting a key is $\theta(n)$
- Worst case for searching is $\theta(n)$
- Algorithm assumes that keys are not deleted once they are inserted
- Deleting a key from an open addressing table is difficult, instead we can mark them in the table as removed (introduced a new class of entries, full, empty and removed)

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

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

– Leads to a secondary clustering (milder form of clustering)

– The clustering effect can be improved by increasing the order to the probing function (cubic). However the hash function becomes more expensive to compute

– But again for two keys k1 and k2, if h(k1,0)= h(k2,0) implies that h(k1,i)= h(k2,i)

Double Hashing

- Recall that in open addressing the sequence of probes follows

$$i_{j+1} = (i_j + c) \bmod m \quad \text{for} \quad j \geq 0$$

- We can solve the problem of primary clustering in linear probing by having the keys which map to the same home position use differing probe sequences. In other words, the different values for *c* should be used for different keys.
- Double hashing refers to the scheme of using another hash function for *c*

$$i_{j+1} = (i_j + h_2(k)) \bmod m \quad \text{for} \quad j \geq 0 \quad \text{and} \quad 0 < h_2(k) \leq m - 1$$

Reference

- Chapter #11

  Introduction to Algorithms (3rd Edition)