10 – HASH TABLES

Topics

- Maps (key-value model)
  - Map ADT
  - List-based Maps
    - Searching issues
- Hash Tables
  - Introduction to Hashing
  - Why Hash Tables
  - Collision Problems
    - Solution #1: Open Addressing
    - Solution #2: Separate Chaining
  - Hash Functions
  - "Map" methods with Separate Chaining
  - Hashing Efficiency

3

Maps (key-value model)

Maps

- A map models a searchable collection of **key-value** entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

Maps (key-value model)

Map ADT

The Map ADT

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

Example

Operation	*Output*	*Map*
isEmpty()	**true**	∅
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,C)	**null**	(5,*A*),(7,*B*),(2,C)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,C),(8,*D*)
put(2,*E*)	C	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

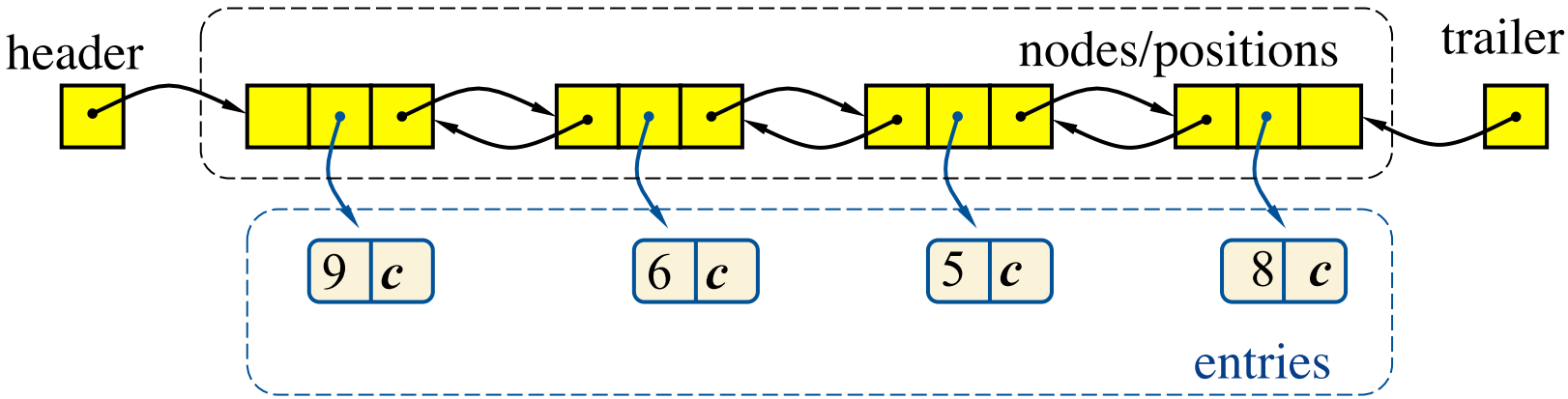
8

Maps (key-value model)

List-based Maps

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()      {the next position in B}
        if p.element().key() = k          then
            return p.element().value()
    return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().key() = k  then
            t = p.element().value()
            B.replace(p,(k,v))
            return t  {return the old value}
S.insertLast((k,v))
n = n + 1          {increment variable storing number of entries}
return null        {there was no previous entry with key equal to k}
```

The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().key() = k  then
        t = p.element().value()
        S.remove(p)
        n = n – 1          {decrement number of entries}
        return t{return the removed value}
return null                {there is no entry with key equal to k}
```

Maps (key-value model)

List-based Maps

Searching issues

Performance of a List-Based Map

- Performance:
  - put takes **O**(n) time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take **O**(***n***) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size

Searching Issues

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires O(n) time
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires O(log n) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an O(1), that is, constant time search?
    - We can do it *if* the array is organized in a particular way

16

Hash Tables

Hash Tables

- A *hash table* is a data structure that offers very fast insertion and searching.
  - No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) !!!!!!!
  - It's so fast that computer programs typically use hash tables when they need to look up tens of thousands of items in less than a second (as in spelling checkers).
- Hash tables are significantly faster than trees, which operate in relatively fast O(logN) time.
- Hash tables are relatively easy to program.
- Disadvantages. They're based on arrays, and arrays are difficult to expand after they've been created.

18

Hash Tables

Introduction to Hashing

Introduction to Hashing

- How a range of key values is transformed into a range of array index values.
- In a hash table this is accomplished with a hash function.

Employee Numbers as Keys

- Index Numbers As Keys
- empRecord rec = databaseArray[72];

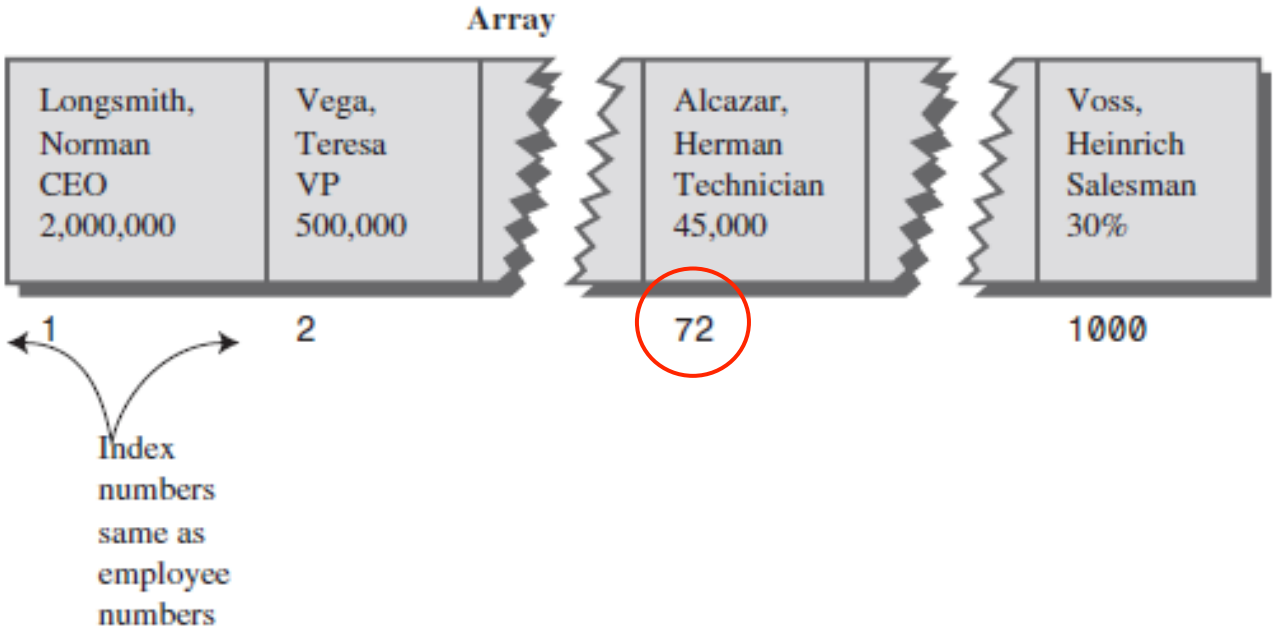


FIGURE 11.1 Employee numbers as array indices.

A Dictionary (1)

- Say we want to store a 50,000-word English-language dictionary (from *a* to *zyzzyva*) in main memory.
- We want every word to occupy its own cell in a 50,000-cell array, so we can access the word using an index number.
  - This will make access very fast.
- But what's the relationship of these index numbers to the words?
  - Given the word *morphosis*, for example, how do we find its index number?

A Dictionary (2)

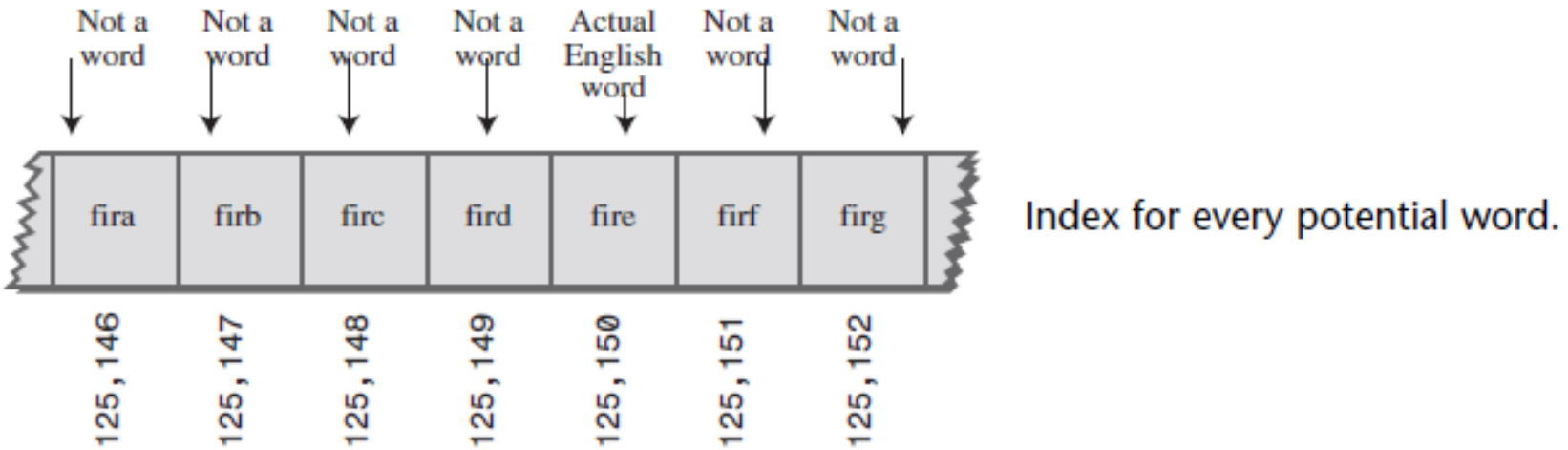
- Converting Words to Numbers
- Assuming that letters correspond to numbers, for example *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z*, *and finally* a blank is 0, so we have 27 characters.
- Then What ???

A Dictionary (3)

- Adding the numbers:
  - Say we want to convert the word *cats* to a number. First, we convert the characters to digits using our homemade code: c = 3, a = 1, t = 20, s = 19
  - Then we add them: 3 + 1 + 20 + 19 = 43 (use as index)
  - Serious Problem: Too many words have the same index ☹
    - (For example, *was*, *tin*, *give*, *tend*, *moan*, *tick*, *bails*, *dredge*, and hundreds of other words add to 43, as *cats* does.)

A Dictionary (4)

- Multiplying by Powers
  - We need to be sure that every character in a word contributes in a unique way to the final number.
  - Say we want to convert the word *cats* to a number. We convert the digits to numbers based on their position.
    3*27^3 + 1*27^2 + 20*27^1 + 19*27^0 = 60,337
  - Unique but range of numbers is huge (consider a ten letters word).



Hashing ☺

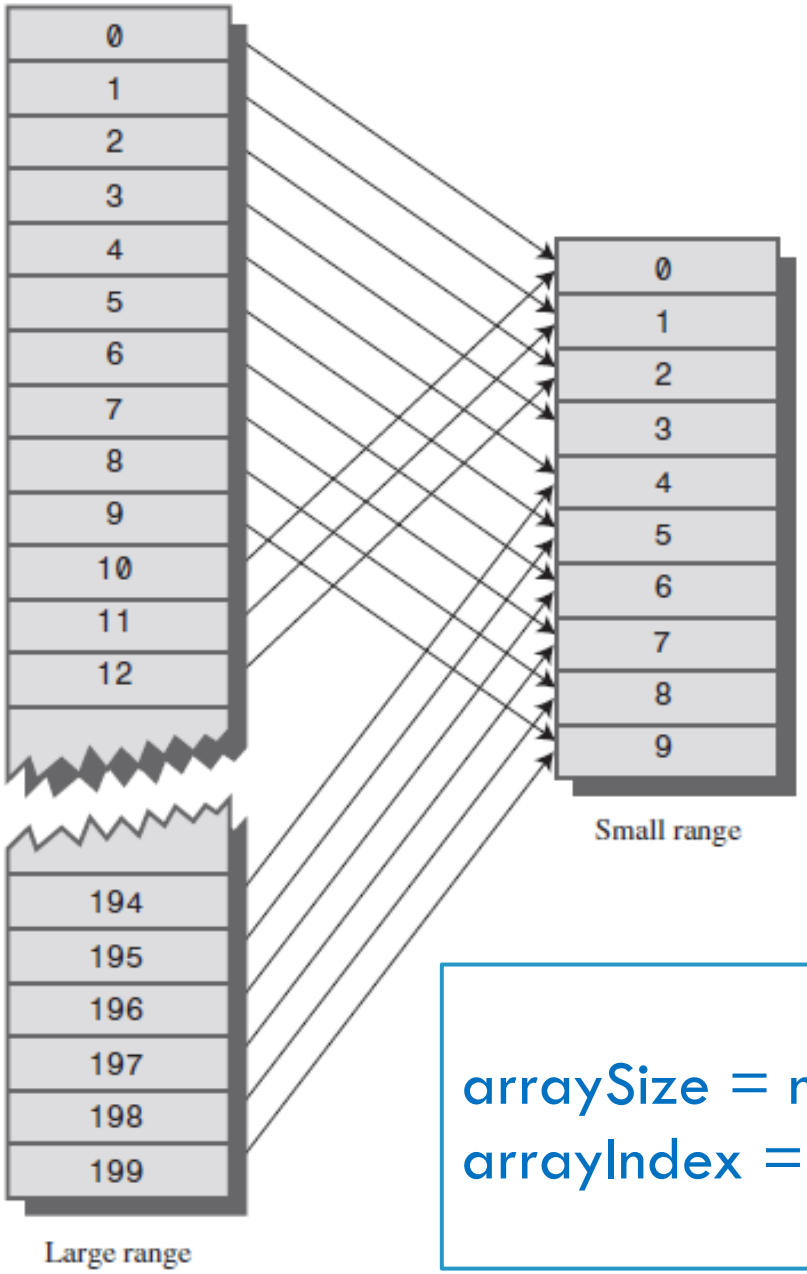
- Suppose we were to come up with a "magic function" that, given a value to search for, would tell us exactly where in the array to look
  - If it's in that location, it's in the array
  - If it's not in that location, it's not in the array
- This function would have no other purpose
- If we look at the function's inputs and outputs, they probably won't "make sense"
- This function is called a hash function because it "makes hash" of its inputs

Hashing ☺

- What we need is a way to compress the huge range of numbers we obtain from the numbers-multiplied-by-powers system into a range that matches a reasonably sized array.
- How big an array are we talking about for our English dictionary?
- If we have only 50,000 words, you might assume our array should have approximately this many elements.

Hashing ☺

- However, it turns out we're going to need an array with about twice this many cells. (It will become clear later why this is so.) So we need an array with100,000 elements.
- Thus, we look for a way to squeeze a range of 0 to more than 7,000,000,000,000 into the range 0 to 100,000.
- A simple approach is to use the modulo operator (%), which finds the remainder when one number is divided by another.
- smallNumber = largeNumber % smallRange;
- arrayIndex = hugeNumber % arraySize;

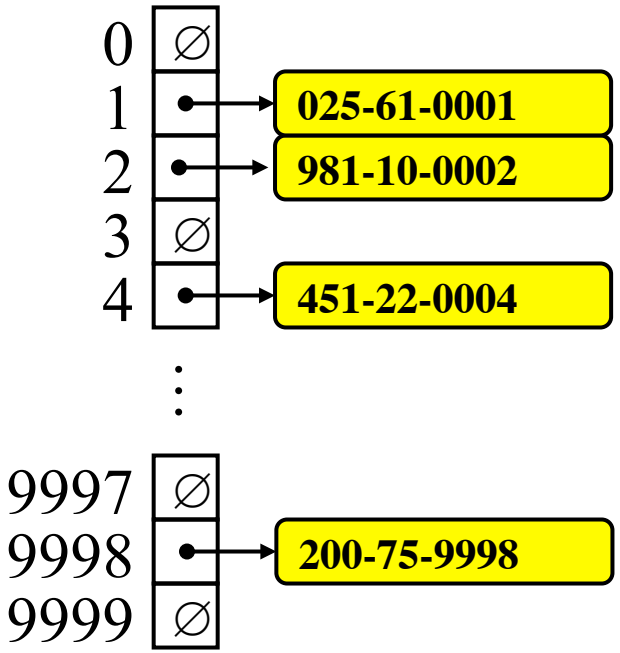



```
arraySize = numberWords * 2;
arrayIndex = hugeNumber % arraySize;
```

Range conversion.

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size N = 10,000 and the hash function
  $h(x)$ = last four digits of x



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
```

Hash Tables

Why Hash Tables

Why hash tables?

- We don't (usually) use hash tables just to see if something is there or not—instead, we put key/value pairs into the table
  - We use a *key* to find a place in the table
  - The *value* holds the information we are actually interested in

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 key 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!

Hash Tables

Collision Problems

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

Collision

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

- Now what?

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

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
- **Solution #1:** (Open Addressing) when a data item can't be placed at the index calculated by the hash function, another location in the array is sought.
  - *Linear probing*, *quadratic probing*, and *double hashing*.
- **Solution #2:** (Separate Chaining) Use the array location as the header of a linked list of values that hash to this location
- Both of 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

IMPORTANT TIP

- Always make the array size a **PRIME NUMBER!!!!**

Hash Tables

Collision Problems:

Solution #1: Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Solution #1: Open Addressing

Linear Probing

- See Hash Workshop Applet
  - Insert
  - Delete
  - Clustering

- Listing 11.1, Hash.java, page 535
  - Hash with linear probing

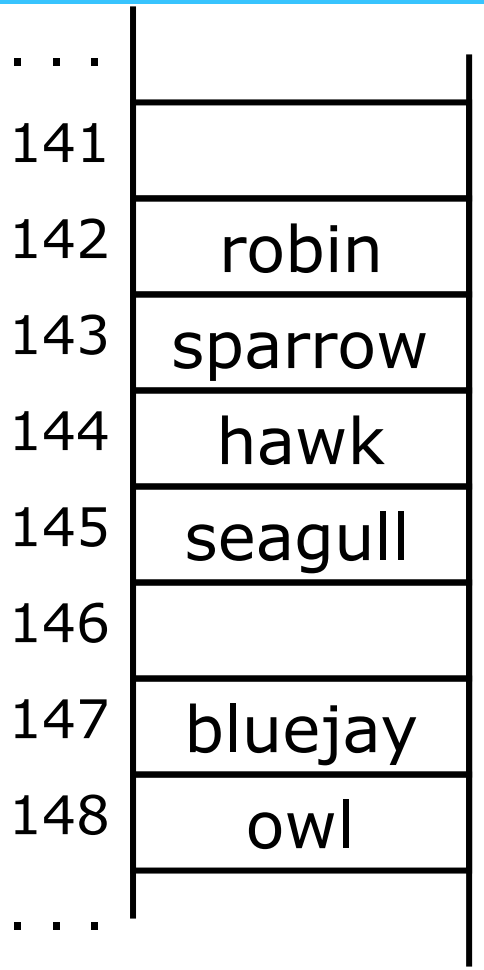
Solution #1: Open Addressing

Linear Probing - 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

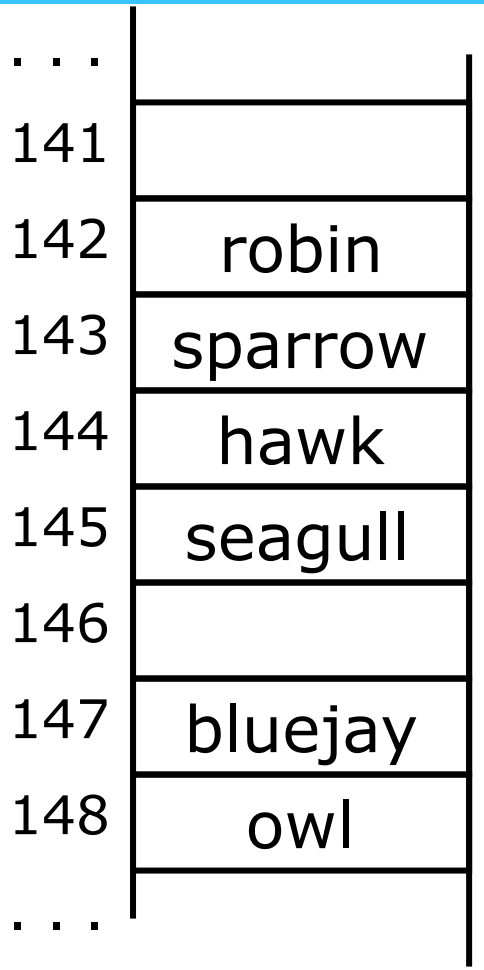
Searching, I

- 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



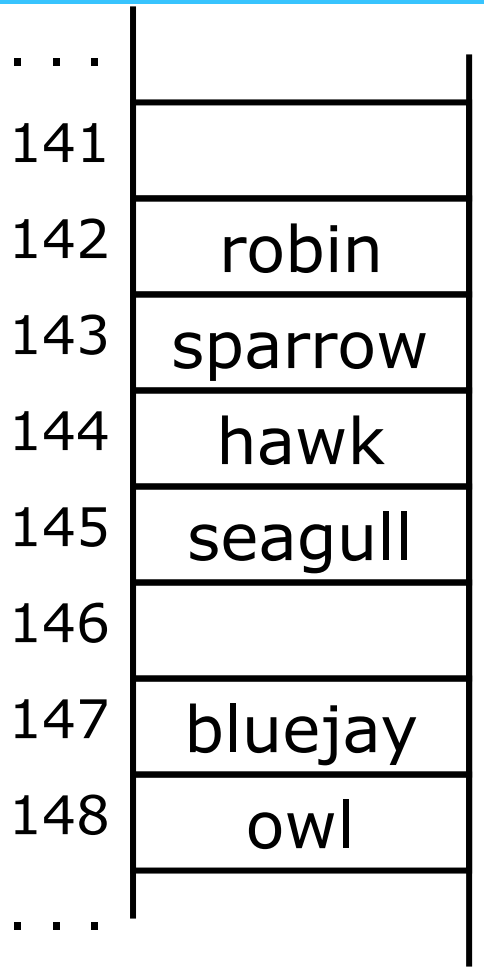
Searching, II

- 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



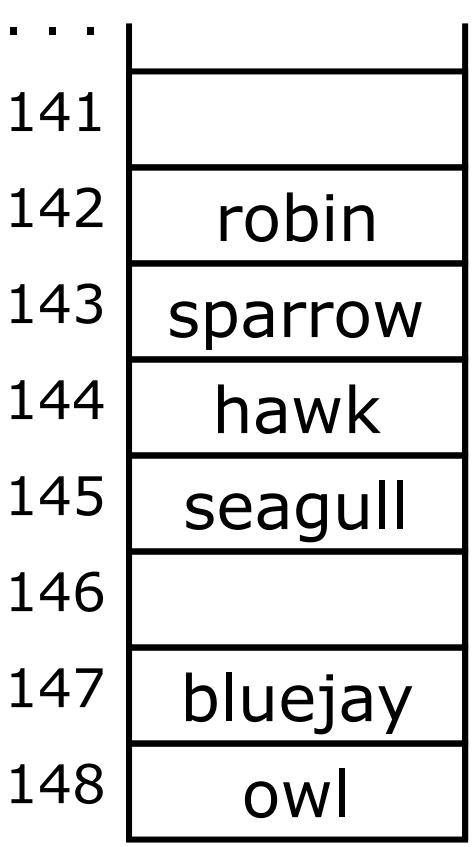
Insertion, II

- 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 ...)



Search with Linear Probing

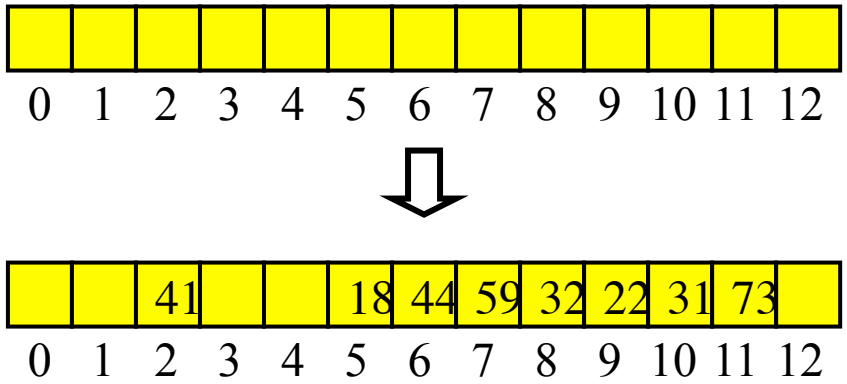
- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

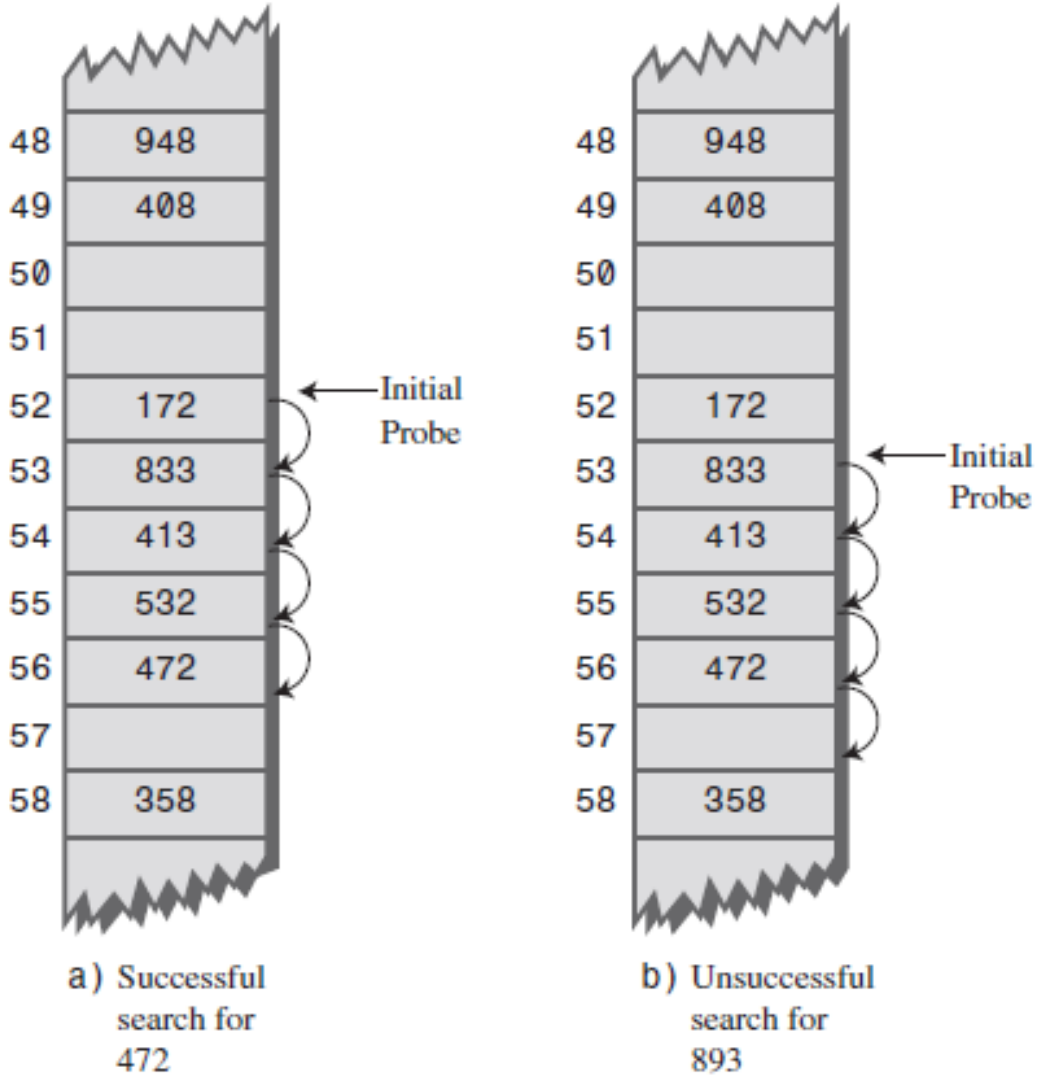
Solution #1: Open Addressing - Linear Probing

- The colliding item is placed in a different cell of the table.
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



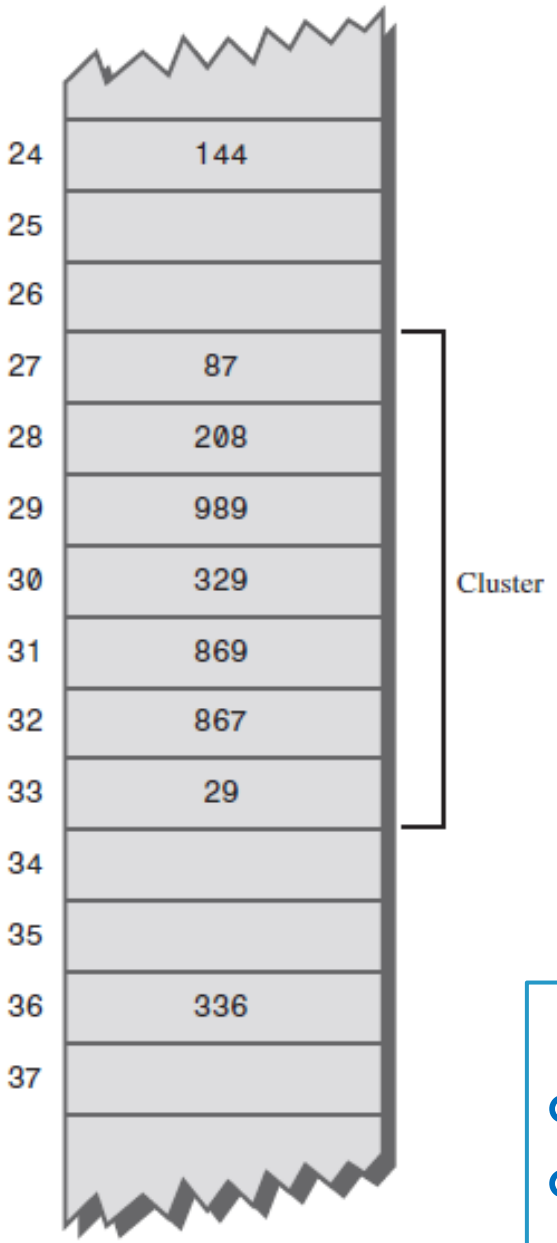
arraySize = 60
Index = key % 60
Search 472
Search 893



Linear probes.

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 even bigger
- Clusters cause efficiency to degrade
- There is a *non*-solution: instead of stepping one ahead, step n locations ahead
  - The clusters are still there, they're just harder to see
  - Unless n and the table size are mutually prime, some table locations are never checked



An example of clustering.

```
arraySize = 60;
arrayIndex = key % 60;
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***DEL***, which replaces deleted elements
- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k***, ***o***) is found, we replace it with the special item ***DEL*** and we return element ***o***
  - Else, we return ***null***
- put(***k***, ***o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until:
    - A cell ***i*** is found that is either empty or stores ***DEL***
  - We store entry (***k***, ***o***) in cell ***i***

Efficiency

- Hash tables are actually surprisingly efficient
- Until the table is about 70% full, the number of probes (places looked at in the table) is typically only 2 or 3
- Sophisticated mathematical analysis is required to *prove* that the expected cost of inserting into a hash table, or looking something up in the hash table, is O(1)
- Even if the table is nearly full (leading to long searches), efficiency is usually still quite high

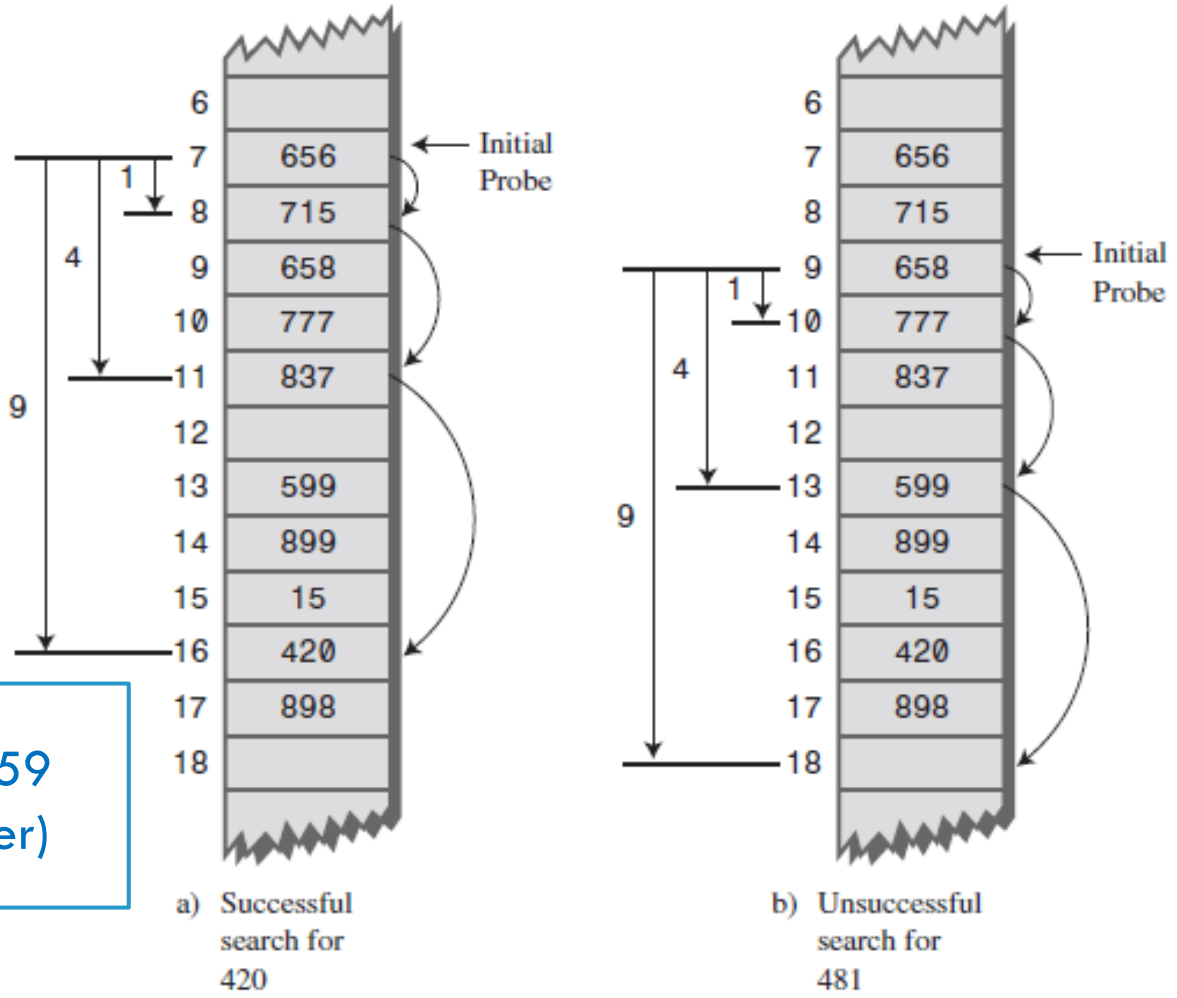
Solution #1: Open Addressing

Quadratic Probing

- **The Step Is the Square of the Step Number**
- In a linear probe, if the primary hash index is x, subsequent probes go to x+1, x+2, x+3, and so on.
- In quadratic probing, probes go to x+1, x+4, x+9, x+16, x+25, and so on.
- The distance from the initial probe is the square of the step number: x+1^2, x+2^2, x+3^2, x+4^2, x+5^2, and so on.
- (See HashDouble Workshop Applet)

Solution #1: Open Addressing

Quadratic Probing



Array size = 59
(prime number)

Solution #1: Open Addressing

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

  $(i + jd(k)) \bmod N$

  for $j = 0, 1, \ldots, N - 1$
- where $i = h(k)$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:

  $d(k) = q - (k \bmod q)$

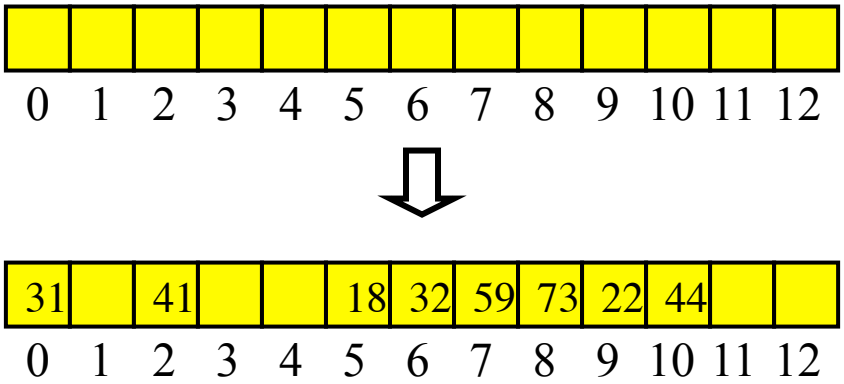
  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

  $1, 2, \ldots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5, 10
59	7	4	7
32	6	3	6
31	5	4	5, 9, 0
73	8	4	8



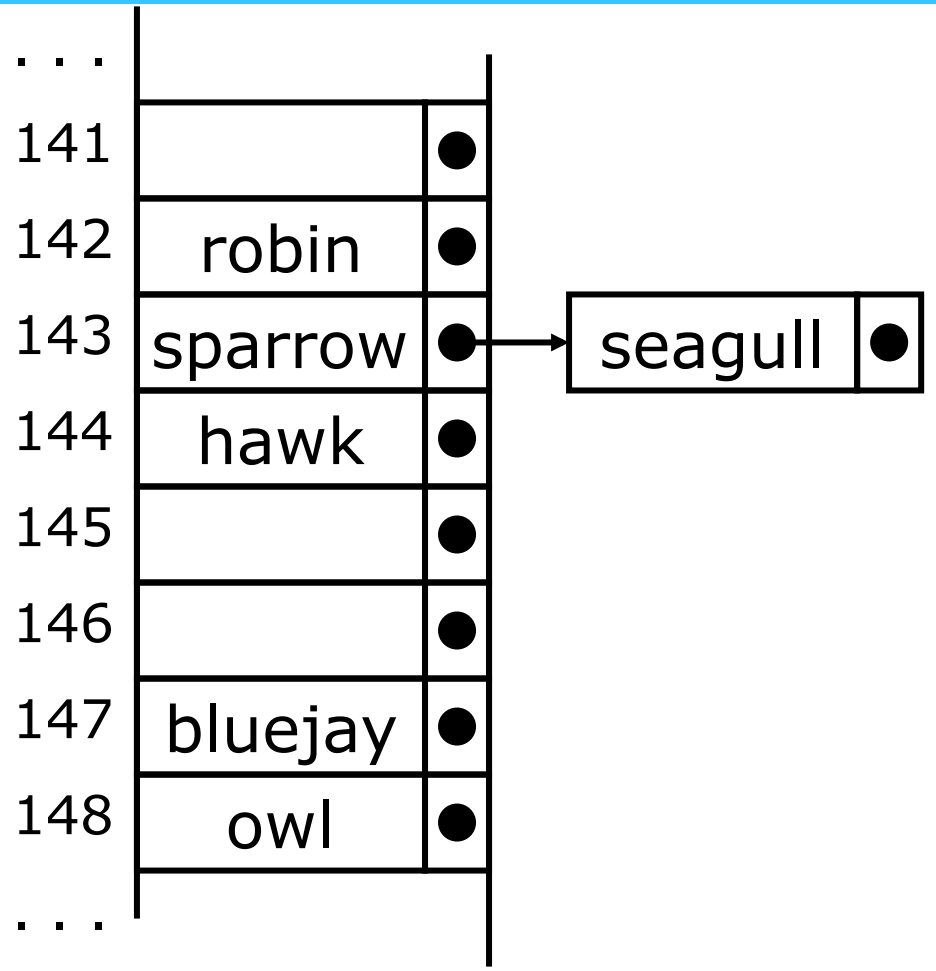
Hash Tables

Collision Problems:

Solution #2: Separate Chaining

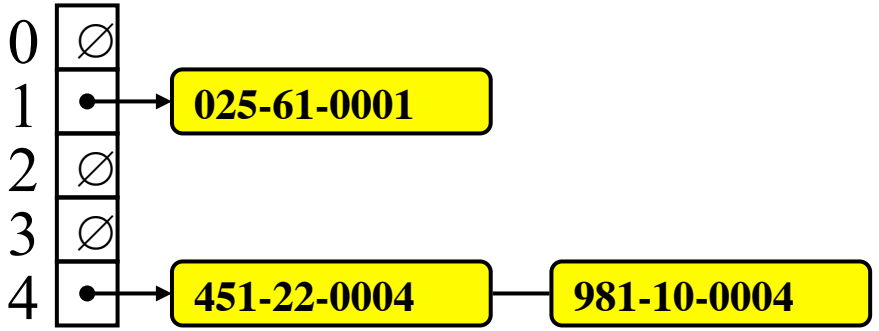
Solution #2: Separate Chaining

- The previous solutions used open addressing: 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



Solution #2: Separate Chaining

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there
- (See HashChain Workshop Applet)



- Separate chaining is simple, but requires additional memory outside the table

Hash Tables

Hash Functions

The hashCode method

- public int hashCode() is defined in Object
- Like equals, the default implementation of hashCode just uses the address of the object—probably not what you want for your own objects
- You can override hashCode for your own objects
- As you might expect, String overrides hashCode with a version appropriate for strings
- Note that the supplied hashCode method *does not know the size of your array*—you have to adjust the returned int value yourself

Writing your own hashCode method

- A hashCode method *must:*
  - Return a value that is (or can be converted to) a legal array index
  - Always return the same value for the same input
    - It can't use random numbers, or the time of day
  - Return the same value for *equal* inputs
    - Must be consistent with your equals method
- It does *not* need to return different values for different inputs
- A *good* hashCode method *should:*
  - Be efficient to compute
  - Give a uniform distribution of array indices

Other considerations

- The hash table might fill up; we need to be prepared for that
  - Not a problem for a separate chaining, of course
- Deleting items from an open addressing table may cause problem in other operations like searching:
  - This would create empty slots that might prevent you from finding items that hash before the slot but end up after it (solved by the *DEL* flag)
  - Again, not a problem for a separate chaining
- Generally speaking, hash tables work best when the table size is a prime number

Hash Functions and Hash Tables

- A hash function ***h*** maps keys of a given type to integers in a fixed interval [0, ***N*** - 1]
- Example:

  $h(x) = x \bmod N$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function ***h***
  - Array or Vector (called "table") of size ***N***
- When implementing a map with a hash table, the goal is to store item (***k***, ***o***) at index $i = h(k)$

Hash Functions

- What to do if **"key"** is a non-numeric?
- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

  h_2: integers $\rightarrow [0, N-1]$

- The **hash code** is applied first, and the compression function is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in a random way

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory…
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Hash Tables

Map Methods with Separate Chaining

Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if $t = $ **null then** {k is a new key}
$n = n + 1$
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
$n = n - 1$
return t

Hash Tables

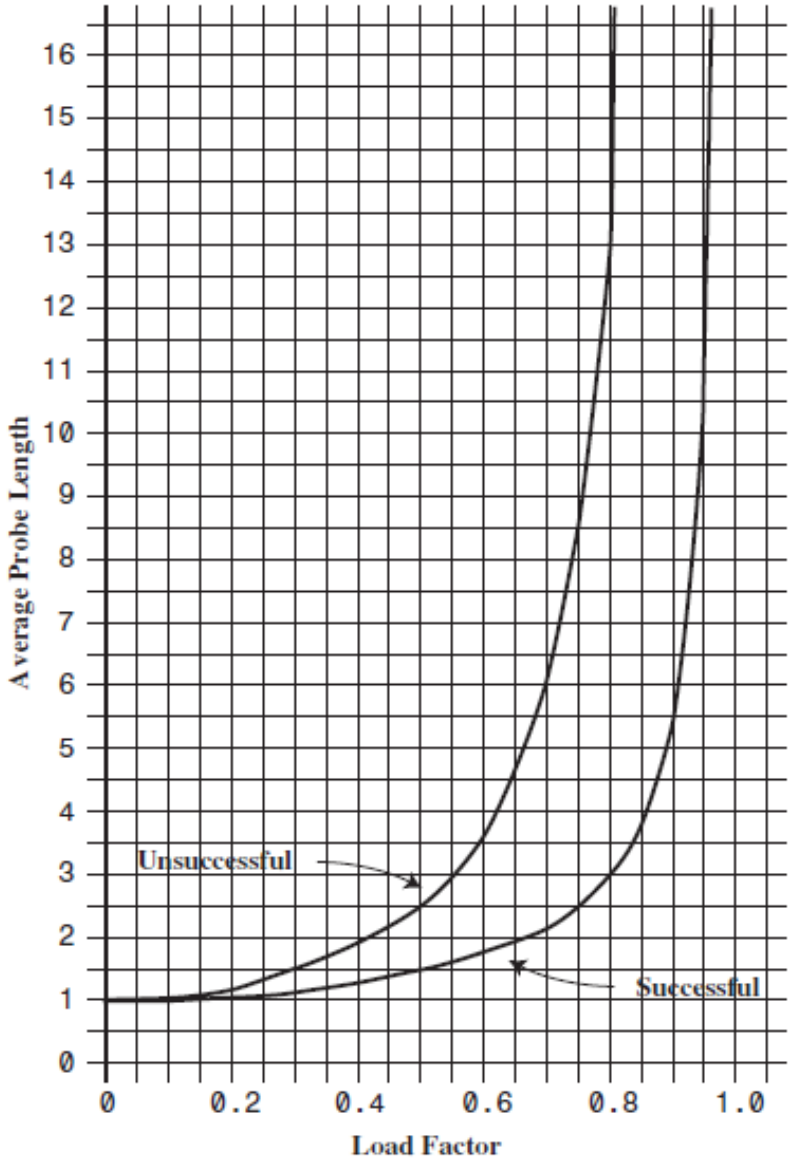
Hashing Efficiency

Hashing Efficiency

- Insertion and searching in hash tables can approach O(1) time, if no collision occurs.
- If collisions occur, access times become dependent on the resulting probe lengths.
- The average probe length (P) (and therefore the average access time) is dependent on the load factor (L) (the ratio of items in the table to the size of the table).
- As the load factor increases, probe lengths grow longer.

Hashing Efficiency

- Open Addressing - Linear Probing:
- Successful Search:
  - $P(L) = (1 + 1 / (1 - L)) / 2$
  - $P(1/2) = 1.5$, $P(2/3) = 2$
- Unsuccessful Search:
  - $P(L) = (1 + 1 / (1 - L)^2) / 2$
  - $P(1/2) = 2.5$, $P(2/3) = 5$
- The load factor must be kept under 2/3 and preferably under 1/2.
  - This can be achieved by increasing the size of the array.



The End