Hash tables
Collision Resolution

Lecture 14

by Marina Barsky

Next to perfect

- No hash function can guarantee that we will find the object in the position object.hashCode
- The next best thing: it can direct us to the place in the array where to start searching

Collision resolution strategies

- Open addressing: each key will have its own slot in the array
  - Linear probing
  - Quadratic probing
  - Double hashing

- Closed addressing: each slot in the array will contain a collection of keys
  - Separate chaining

Linear probing

- What can we do when two different values attempt to occupy the same slot in the array?

- Search from there for an empty location
  - Can stop searching when we find the value *or* an empty location
  - Search must be end-around (circular array!)

Add with linear probing

- 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

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

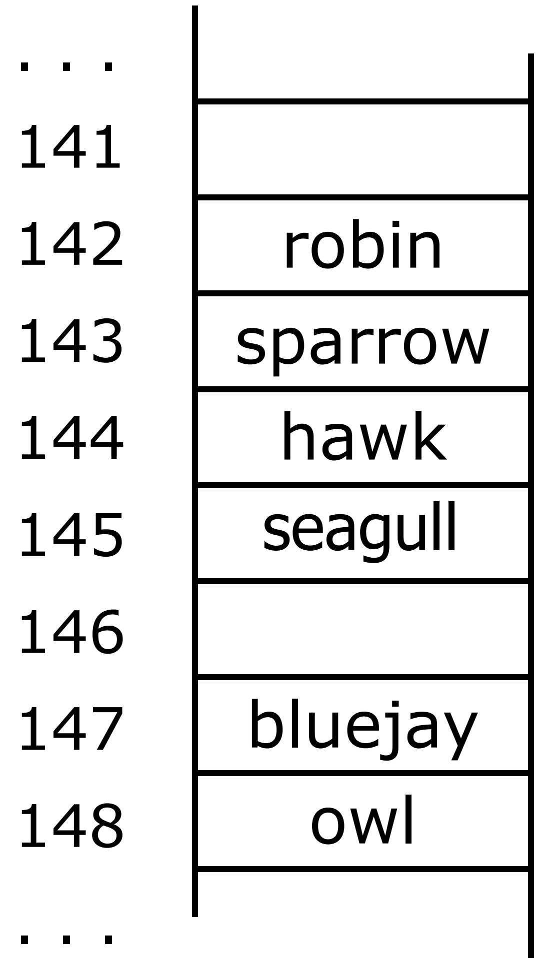
Get with linear probing: *seagull*

- 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

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Get with linear probing: *cow*

- 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



Add with linear probing

- 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

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Add with linear probing

- Suppose you want to add cardinal to this hash table
- Also suppose:
  - 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 ...)

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Problem 1 with open addressing: deletion

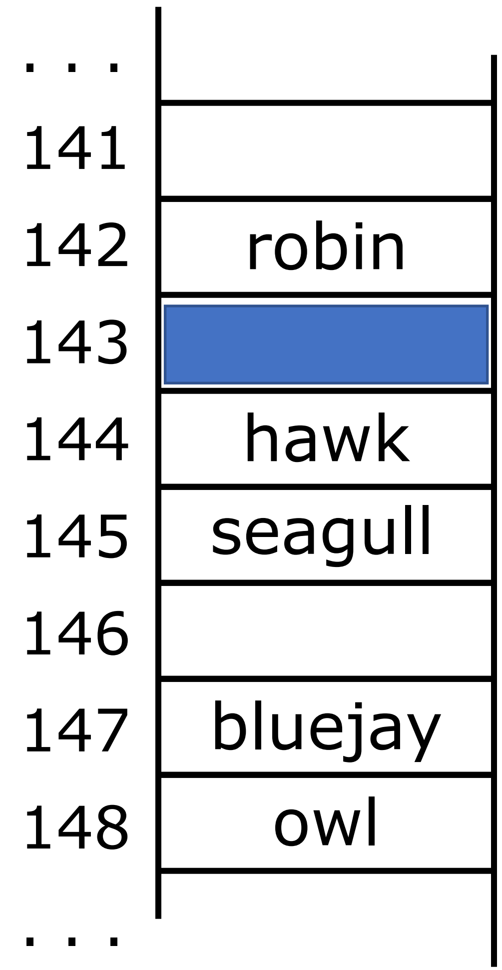
- What happens if we delete sparrow?
  - hashCode(sparrow)=143
  - hashCode(seagull)=143

Problem 1 with open addressing: deletion

- What happens if we delete sparrow?
  - hashCode(sparrow)=143
  - hashCode(seagull)=143

Problem 1 with open addressing: deletion

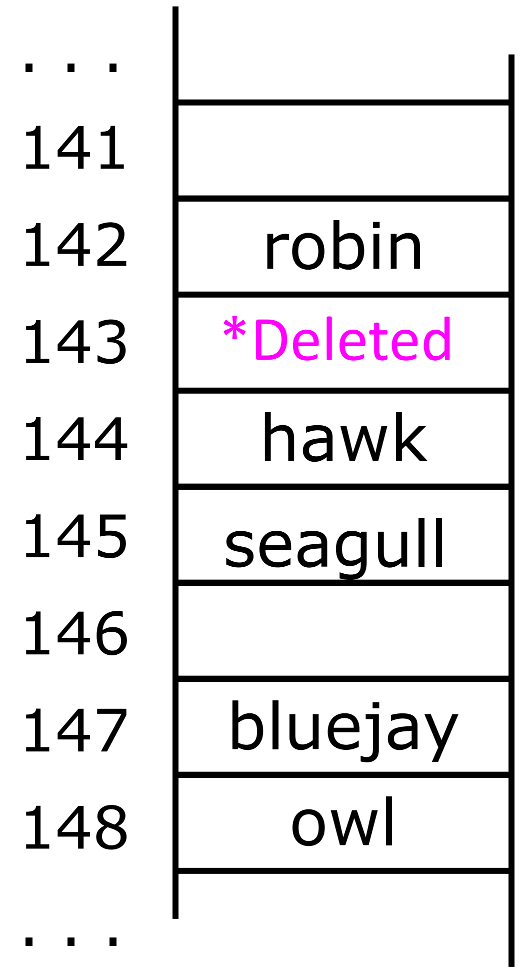
- ➢ What happens if we delete sparrow?
  - ○ hashCode(sparrow)=143
  - ○ hashCode(seagull)=143
- ➢ Now when searching for seagull we check
  - ○ table[143] is empty
  - ○ We can not find seagull!



Solution to the deletion problem

- After we delete sparrow we put a special sign *deleted* instead of *empty*
  - hashCode(sparrow)=143
  - hashCode(seagull)=143
- Now when searching for seagull we check
  - table[143] is deleted
  - We skip it
  - table[144] is not empty
  - table[144] !=seagull
  - table[145]=seagull

  We found seagull!
- The deleted slots are filling up during the subsequent insertions



Problem 2 with linear probing: clustering

- A big problem with the above technique is the tendency to form "clusters"
- A ***cluster*** is a consecutive area in the array 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 degradation in the efficiency of search
- Here is a *non*-solution: instead of stepping one ahead, step k locations ahead
  - The clusters are still there, they're just harder to see
  - Unless k and the table size are mutually prime, some table locations will not be ever checked

Solution 1 to clustering problem: Quadratic probing

- As before, we first try slot j=*hashCode* MOD M.
- If this slot is occupied, instead of trying slot $j=|(j+1) \text{ MOD } M|$, try slot:

  $j=|(hashCode+i^2) \text{ MOD } M|$, where i takes values with increment of 1 and we continue until j points to an empty slot

- For example if position *hashCode is* initially 5, and M=7 we try:

j = 5 MOD 7 = 5

j =(5 + 1^2) MOD 7 = 6 MOD 7 = 6

j =(5 + 2^2) MOD 7 = 9 MOD 7 = 2

j =(5 + 3^2) MOD 7 = 14 MOD 7 = 0 etc.

$j = |(hashCode + i^2) \text{ MOD } N|$, hashCode = 3, N=10

Under quadratic probing, with the following array, where will an item that hashes to 3 get placed?

A. 0

B. 2

C. 5

D. 9

E. None of the above

Index	Value
0	
1	
2	
3	x
4	x
5	
6	
7	x
8	
9	

Problems with Quadratic probing

- Quadratic probing helps to avoid the clustering problem
- But it creates its own kind of clustering, where the filled array slots “bounce” in the array in a fixed pattern
- In practice, even if *M* is a prime, this strategy may fail to find an empty slot in the array that is just half full!

Solution 2 to clustering problem: Double hashing

- In this approach we choose the secondary hash function: *stepHash(k)*.
- If the slot j=*hashCode* MOD M is occupied, we iteratively try the slots

  j = |(*hashCode*+i***stepHash*) MOD M|

- The secondary hash function *stepHash* is not allowed to return 0
- The common choice (Q is a prime):

  stepHash(S)=Q-(hashCode(S) mod Q)

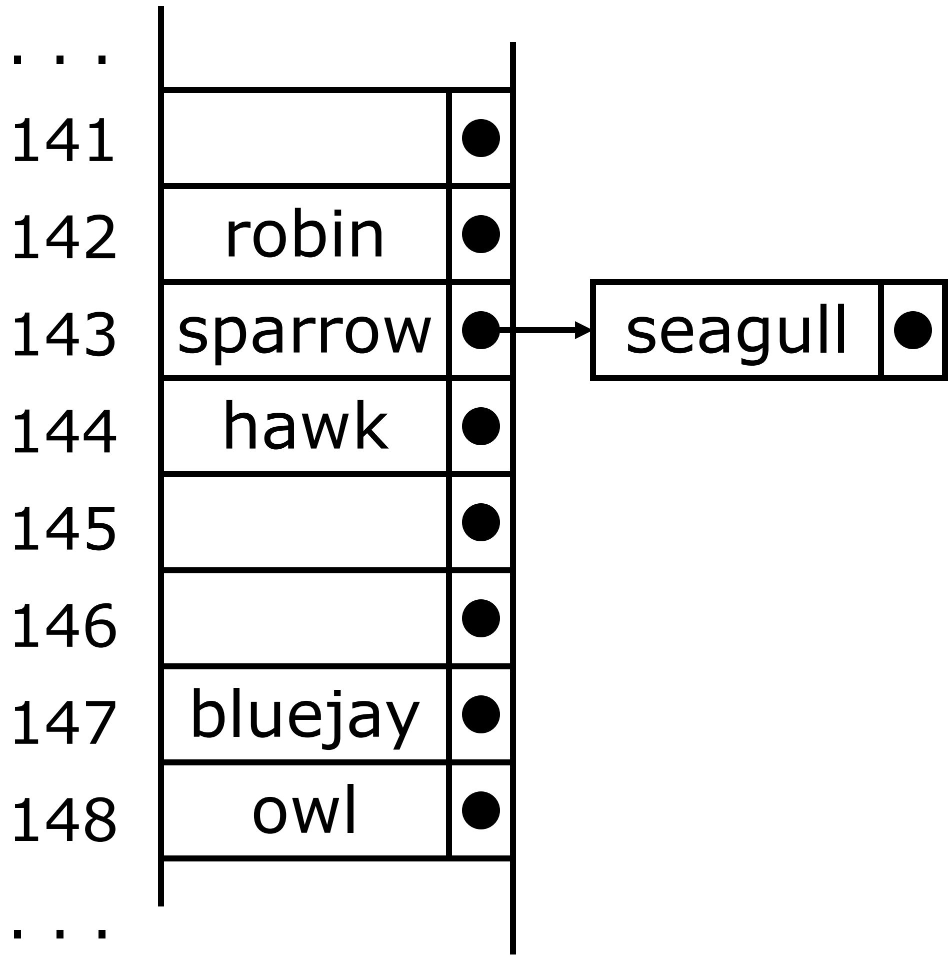
Collision resolution strategies

- Open addressing: each key will have its own slot in the array
  - Linear probing
  - Quadratic probing
  - Double hashing

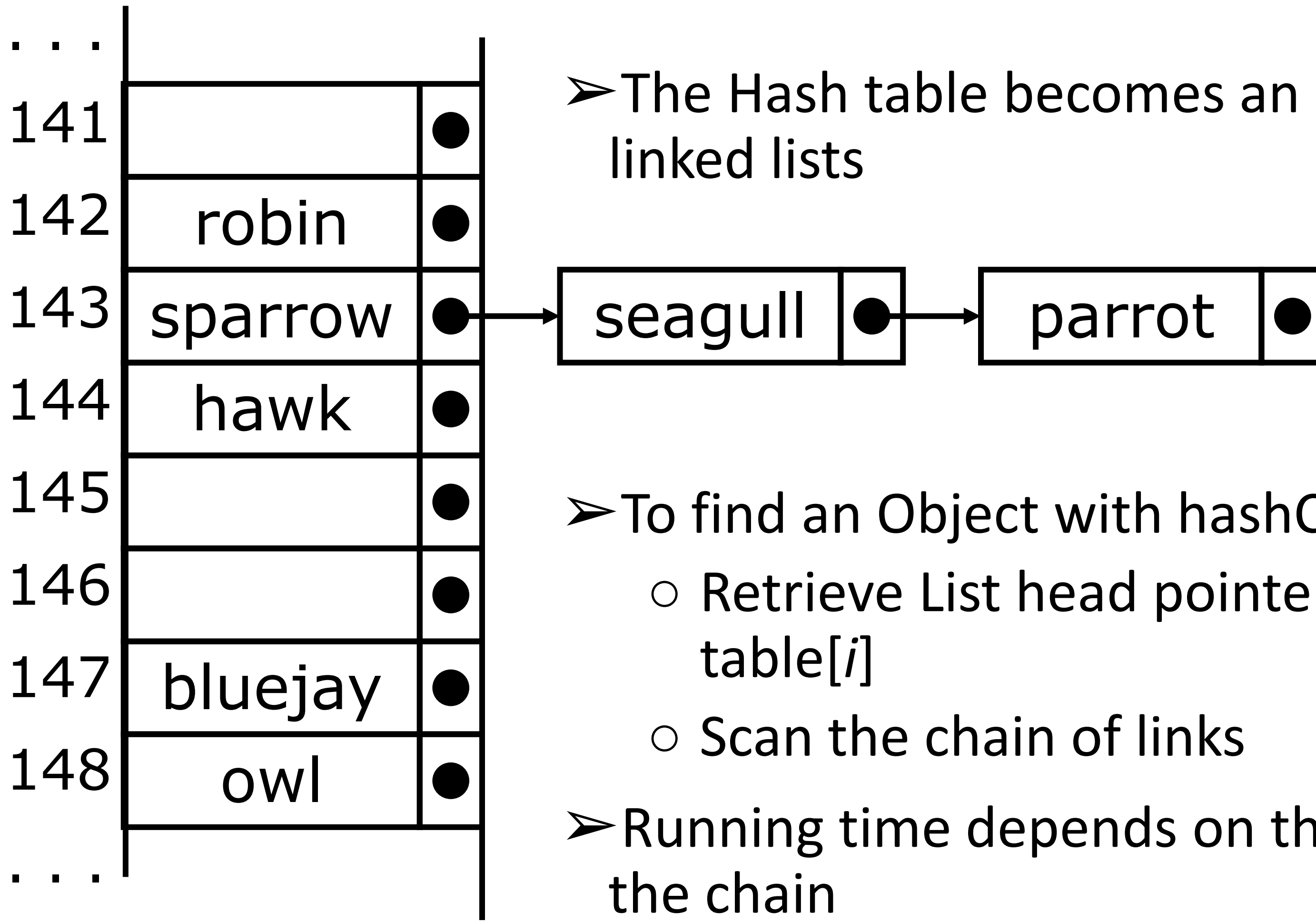
- Closed addressing: each slot in the array will contain a collection of keys
  - Separate chaining

Separate chaining

- The previous solutions use open addressing: all entries go into a "flat" (unstructured) array
- Another solution is to store in each location the head of a *linked list* of values that hash to that location



Separate chaining: *Get*



- ➢The Hash table becomes an array of *M* linked lists
- ➢To find an Object with hashCode *i*
  - ○ Retrieve List head pointer from table[*i*]
  - ○ Scan the chain of links
- ➢Running time depends on the length of the chain

Separate Chaining vs. Open Addressing

- ➢ If the space is not an issue, *separate chaining* is the method of choice: it will create new list elements until the entire memory permits
- ➢ If you want to be sure that you occupy exactly *M* array slots, use *open addressing*, and use the probing strategy which minimizes clustering

ADT Map operations: performance

Implementation	Worst case			Expected		
	Get (Contains)	Add	Remove	Get (Contains)	Add	Remove
Unsorted Array	O(N)	O(1)**	O(N)	N/2	1**	N/2
Unsorted Linked List	O(N)	O(1)**	O(N)	N/2	1**	N/2
Sorted Array	O(log N)	O(N)	O(N)	log N	N/2	N/2
Hash table with linear probing	O(N)	O(N)	O(N)	1*	1*	1*
Hash table with separate chaining	O(N)	O(N)	O(N)	1*	1*	1*

**If we know that new key is unique

*Given a good hash function

Hash table performance

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

Maps

- ADT *map*—a way of looking up one thing based on the value of another
  - We use a *key* to find a place in the map
  - The associated *value* is the information we are trying to look up

	Key	Value
0		
1		
2	Li	Li info
3	Yam	Yam info
4	Chan	Chan info
5	Jones	Jones info
6	Taylor	Taylor info
7		

MAP = ASSOCIATIVE ARRAY, DICTIONARY

What is a key and what is a value?

Key	Phone number
Li	11111
Yam	22111
Chan	33111
Jones	11444
Taylor	55111

Key	Last Name
11111	Li
22111	Yam
33111	Chan
11444	Jones
55111	Taylor

The answer: depends on the application

Maps and Sets

- Sometimes we just want a *set* of things—objects are either in it, or they are not in it

0	
1	
2	Li
3	Yam
4	Chan
5	Jones
6	Taylor
7	

SET

Set

- A *set* is simply a collection of unique things: the most significant characteristic of any set is that it does not contain duplicates

- We can put anything we like into a set. However, in Java we group together things of the same class (type): we could have a set of *Vehicles* or a set of *Animals*, but not both [as with any other collection)

Abstract Data Type: Set

Specification

Set is an Abstract Data Type which stores a collection of unique elements* and supports the following operations:

→**Contains (k)** - returns *True* if element *k* is in the collection. Returns *False* otherwise.

→**Add (k)** - adds element *k* to the collection

→**Remove (k)** - removes element *k* from the collection

*The order of elements in the collection is not important

Sets are optimized for set operations:

`Set A={1, 2, 3, 4}`   `Set B={4, 3, 1, 6}`

→Intersection (set A, set B): creates a new set C consisting only of elements that are found both in A and in B:

A ∩ B = {1, 3, 4}

→Union (set A, set B): combines all elements of A and B into a single set C (removes duplicates):

A U B = {1, 2, 3, 4, 6}

→Difference (set A, set B): creates a new set C that contains all the elements that are in A but not in B:

A – B = {2}

Set Operations in Java: DEMO

Implemented in Java library using a Hash Table

Common implementations of Map and Set ADT that use Hash Tables

- Set:
  - *unordered_set* in C++
  - *HashSet* in Java
  - *set* in Python
- Map:
  - *unordered_map* in C++
  - *HashMap* in Java
  - *dict* in Python

Now you know that in Python:

```
# list (array)
t = [1,2,3,4, …, n]

if 8 in t:
     print('found')
```

Time O(n)

```
# set
s = {1, 2, 3 … n}

if 8 in s:
     print('found')



(same for dictionary)
```

Time O(1)