“Hash is a food, especially meat and potatoes, chopped and mixed together; a confused mess “ (en.wiktionary.org/wiki/hash)

Hashing

Review 02.04

Motivation: searching in an array

- Case 1: *N* elements are **unsorted** – search requires **O(*N*)** time
  - If the value isn't there, we need to search all *N* elements (worst-case)
  - If the value is there, we search *N*/2 elements on average (average between 1 and N)

- Case 2: *N* elements are **sorted – O(log *N*)** binary search
  - Very fast whether the element is found or not

It doesn't seem like we could do much better

Searching in time O(1)

- How about **O(1)**, that is, **constant time search**?
- We can do it **if** the array is organized in a particular way

First repeating character

Input: String *S* of length *N*
Output: first repeating character (if any) in *S*

➢ The obvious O(N^2) solution:

```
for each character in order:
    check whether that character is repeated
```

First repeating character

Input: String *S* of length *N*
Output: first repeating character (if any) in *S*

a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111

The number of all possible characters is 256 (ASCII characters)

- ➢ We create an array *H* of size 256 and initialize it with all zeros
- ➢ For each input character *c* go to the corresponding position *H*[*c*] and set count at this position to 1
- ➢ Since we are using arrays, it takes constant time for reaching any location
- ➢ Once we find a character for which counter is already 1 - we know that this is the one which is repeating for the first time

First repeating character

Input: String *S* of length *N*
Output: first repeating character (if any) in *S*

cabare

a	97	1
b	98	1
c	99	1
d	100	
e	101	
f	102	
g	103	
h	104	
i	105	
j	106	
k	107	
l	108	
m	109	
n	110	
o	111	

- ➢ Because the total number of all possible keys is small (256), we were able to map each key (character) to a single memory location
- ➢ The key tells us precisely where to look in the array!

This method of storing keys in the array is called ***direct addressing:*** **store key *k* in position *k* of the array**

Run-time O(N)

First repeating **number**

Input: Array *A* containing *N* **integers**
Output: first repeating number (if any) in *A*

- ➢ This very similarly looking problem cannot be solved with direct addressing
- ➢ The total number of all possible integers is 2,147,483,647. This is the universe of all possible keys - thus the size of the array
- ➢ What if we have only 25 integers to store? Impractical
- ➢ Impossible: if array elements are floating point numbers - strings - objects
- ➢ For these cases we use a technique of *hashing*: we convert **each array element into a number** using *hash function*

Intuition: hashing inputs

- ➢ Suppose we were to come up with a “magic function” that, given a key to search for, would tell us the exact location in the array such that
  - ○ If key is in that location, it’s in the array
  - ○ If key is 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, the connection between them won’t “make any sense”
- ➢ This function is called a ***hash function*** because it “makes hash” of its inputs

Example: hashing students

- ➢ Suppose we want to store student objects in the array
- ➢ For each student we apply the following ***hash function***:

hashCode(Student) =

length (Student.lastName)

This gives us the following values:

- hashCode('Chan')=4
- hashCode('Yam')=3
- hashCode('Li')=2
- hashCode('Jones')=5
- hashCode('Taylor')=6

Array of students: *hash table*

- We place the students into array slots which correspond to the computed hash values:
  - hashCode('Chan')=4
  - hashCode('Yam')=3
  - hashCode('Li')=2
  - hashCode('Jones')=5
  - hashCode('Taylor')=6

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

Good hash function: length of the last name

➢Our hash function is easy to compute

➢An array needs to be of size 18 only, since the longest English surname, Featherstonehaugh (Guinness, 1996), is only 17 characters long

➢We waste a little of space with entries 0,1 of the array, which do not seem to be ever occupied. But the waste is not bad either

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

Bad hash function: length of the last name

- Suppose we have a new student: Smith
  - hashValue(‘Smith’)=**5**
- When several values are hashed to the same slot in the array, this is called a **collision**
- Now what?

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

Looking for a good hash function

- ➢What about the day of birth?
  - ○ We know that this would be only 365 (366) possible values
  - ○ The birth day of each student is randomly distributed across this range, and this hash function is easy to compute

Birthday paradox

- ➢For a college with only *n*=24 students, the probability that any 2 of theme were born on the same day is > 0.5
- ➢Let's approximate this probability:
  - ○ The probability of any two people not having the same birthday is:

    p =364/365
  - ○ The number of possible student pairs is $\binom{n}{2} = n(n-1)/2 = 276$
  - ○ The probability for *n* students of not having birthday on the same date is $p^{n(n-1)/2}$. For 24 students this gives: $(364/365)^{276} \approx 0.47$.
  - ○ Then the probability of finding a pair of students colliding on their birthday is 1.00 - 0.47 = 0.53!
- ➢This is called *a birthday paradox*

Perfect hash function: requirements

A perfect *hash function* is a function that:

1. When applied to an Object, returns a number
2. When applied to *equal* Objects, returns the *same* number for each
3. When applied to *unequal* Objects returns *different* numbers for each, preventing collisions.
4. The numbers returned by hash function are *evenly* distributed between the range of the positions in the array
5. We also require for our hash function to be *efficiently* computable

non-random inputs → random numbers?

Looking for a **perfect** hash function

- How can we come up with this perfect hashing 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 is possible to compute a perfect hash function. For example hashing objects by their SSN numbers. But this will require an array to be of size 10^9
- It seems that **collisions are essentially unavoidable**
- 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 **in what area of an array to *start* looking**!

Hashing strings by summing up their character values

- ➢It seems like a good idea to map each surname into a number by adding up the ranks (or ASCII codes) of letters in this surname.

hashCode (S) = $\sum_{i=0}^{len(S)} rank(S[i])$

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

What a great hash function!

$$\text{hashCode (S)} = \sum_{i=0}^{len(S)} rank(S[i])$$

- hashCode('Chan')=3+8+1+14=26
- hashCode('Yam')=24+1+13=38
- hashCode('Li')=12+9=21
- hashCode('Jones')=10+15+14+5+18=62
- hashCode('Taylor')=19+1+24+12+15+17=88

- hashCode('Smith')=18+13+9+19+8=67

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

Still a lot of collisions!

$$\text{hashCode (S)} = \sum_{i=0}^{len(S)} rank(S[i])$$

- Not only hashCode('Yam')=hashCode('May')
- But hashCode('Chan')= hashCode('Lam') !

The function takes into account the value of each character in the string, but **not the order of characters**

Polynomial hashing scheme

- ➢The summation is not a good choice for sequences of elements where the order has meaning
- ➢Alternative: choose *A*≠1, and use a hash function for string *S* of length *N*:

$$hashCode(S) = \sum_{i=0}^{N-1} S[i] \cdot A^{N-1-i} =$$

$$S[0] \cdot A^{N-1} + S[1] \cdot A^{N-1-1} + S[2] \cdot A^{N-1-2} + \cdots + S[N-1] \cdot A^{N-1-(N-1)}$$

- ➢This is a polynomial of degree *N* in *A*, and the elements (characters) of the String are the coefficients of this polynomial

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

Example: polynomial hashing

$$hashCode(S) = \sum^{N-1} S[i] \cdot A^{N-1-i} =$$

$$S[0] \cdot A^{N-1} + S[1] \cdot A^{N-1-1} + S[2] \cdot A^{N-1-2} + \cdots + S[N-1] \cdot A^{N-1-(N-1)}$$

S_1 = ‘Yam’

S_2 = ‘May’

A = 31

$hashCode(S_1) = 24*31^2 + 1*31^1 + 13*31^0 = 23108$

$hashCode(S_2) = 13*31^2 + 1*31^1 + 24*31^0 = 12548$

➢Instead of using the summation of all bits of the characters, the polynomial hash function tries to introduce interactions between different bits of successive characters that will provoke or spread randomness of the result

Polynomial hashing of strings

➢Choose A = 31

$$hashCode(S) = \sum_{i=0}^{N-1} S[i] \cdot A^{N-1-i} =$$

$$S[0] \cdot A^{N-1} + S[1] \cdot A^{N-1-1} + S[2] \cdot A^{N-1-2} + \cdots + S[N-1] \cdot A^{N-1-(N-1)}$$

for *A*=31:

X·31 is the same as *X*·**32** - *X* = *X*<<**5** - *X*

The best hashing: non-random inputs → random numbers

The bits in characters are not randomly distributed:

for example, lower case letters are from 97 to 122, which in binary: **011**00001 to **011**11010. All the randomness is in lower 5 bits.

- The multiplication by 32 (left shift 5) distributes the randomness of lower bits into higher bits
- The result: hash codes evenly distributed over the range of possible integers

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
a	97	**0**	**1**	**1**	0	0	0	0	1
z	122	**0**	**1**	**1**	1	1	0	1	0

Horner's rule

- How to compute polynomial of degree *N* in time O(*N*)?
- Horner's method:

$$\begin{aligned} p(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n \\ &= a_0 + x\bigg(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\,a_n)\cdots\big)\Big)\bigg) \end{aligned}$$

Example: Java String hashCode()

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$
$$= a_0 + x\bigg(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\,a_n)\cdots\big)\Big)\bigg)$$

- ➢Java String hashCode() does exactly that:

  x=31, and it computes a polynomial of degree n = *length*() using the ASCII value of each character as the polynomial coefficient a_i

```
public int hashCode() {
        int hash=0;
        for (int i=0;i< length(); i++)
                hash=hash*31+S[i];
        return hash;
}
// For length()=3
Iteration 0:
        hash=0*31+S[0]=S[0]
Iteration  1:
        hash=31*S[0]+S[1]
Iteration 2:
        hash=31*(31*S[0]+S[1])+S[2]=
        31^2*S[0]+31^1*S[1]+31^0*S[2]
```

Example: Java String hashCode()

- This is quite a good hash function: it returns for different strings mostly different values which are well spread over the range of all possible integers
- This hash function is also very efficient, since it has only *n = length()* steps and we can replace the expensive multiplication by the cheap shift and subtraction

```
public int hashCode()
{
   int hash=0;
   for (int i=0;i< length(); i++)
      hash=hash*31+S[i];
   return hash;
}
```

```
public int hashCode()
{
   int hash=0;
   for (int i=0;i< length(); i++)
      hash=hash<<5-hash+S[i];
   return hash;
}
```

hash<<5 is the same as hash*2^5

Reducing the range of *hashCode* to the capacity of the array

- The output of hash function is a number randomly distributed over the range of **all** integers.
  - But we need to store our objects in the array of size ***M***
- Step 2: **compression mapping**
  - Converting integers in range ~ [0,4000000000] to integers in range [0, *M*]
  - The simplest way to do it: |*hashCode*| MOD *M*
  - In practice, the MAD (Multiply Add and Divide) method:

    |(*A**hashCode+*B*) MOD *M*|

    The best results when *A*, *B* and *M* are primes

Students example

- ➔ Applying the polynomial hash function:

hashCode('Taylor')=-880692189
hashCode('Yam')=119397
hashCode('Li')=345
hashCode('Lee')=107020
hashCode('Lam')=106904
hashCode('Roy')=113116

- ➔ Applying the |(11*hashCode+13) MOD 7| compression mapping:

arrayIndex('Taylor')=6
arrayIndex('Yam')=2
arrayIndex('Li')=4
arrayIndex('Lee')=5
arrayIndex('Lam')=3
arrayIndex('Roy')=1

0	
1	Roy
2	Yam
3	Lam
4	Li
5	Lee
6	Taylor

No more collisions?

- Does a good hash always produce different hash code for different strings?

  The answer is **NO**.

  If you run the code in the box, you will find out that

  - The words *Aa* and *BB* have the same **hashCode**
  - Words *variants* and *gelato* hash to the same value
  - The same for *misused* and *horsemints*

- We have to be prepared to deal with **collisions**, since they **are unavoidable**

```
public static void main(String [] args) {
    String [] words=new String[6];
    words[0]="Aa";
    words[1]="BB";
    words[2]="variants";
    words[3]="gelato";
    words[4]="misused";
    words[5]="horsemints";

    for(int i=0;i<6;i++)   {
      System.out.print("Hash code of "+words[i]+": ");
      System.out.println(words[i].hashCode());
   }
}
```

Collision resolution strategies

- Open addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

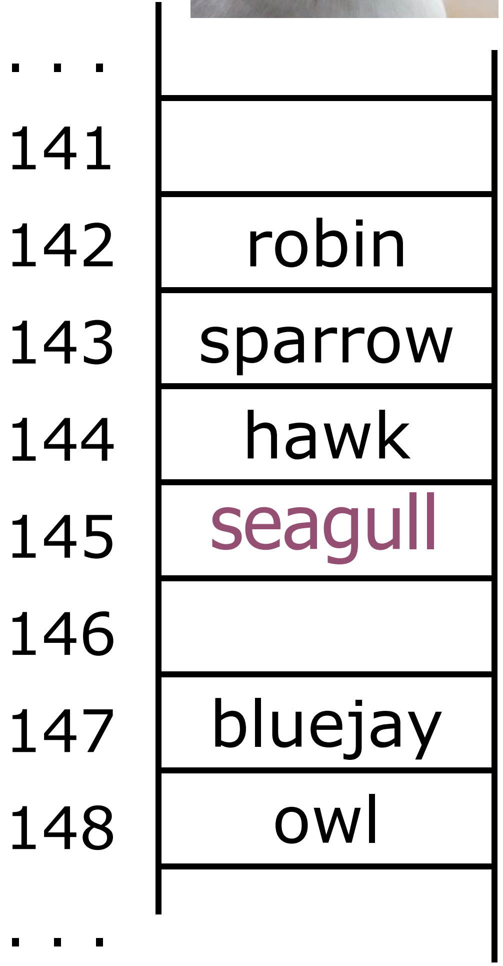
- Separate chaining

Linear probing

- What can we do when two different values attempt to occupy the same place 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)

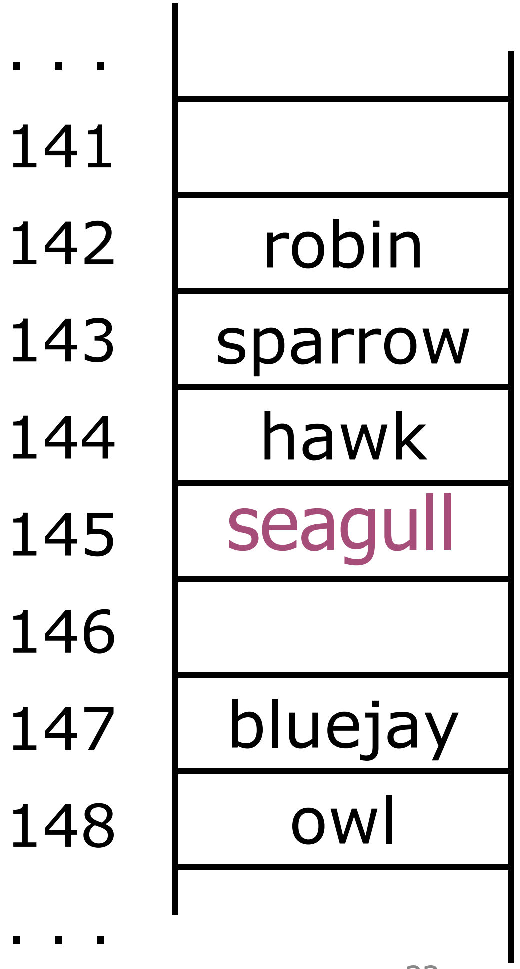
Add with linear probing

- 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



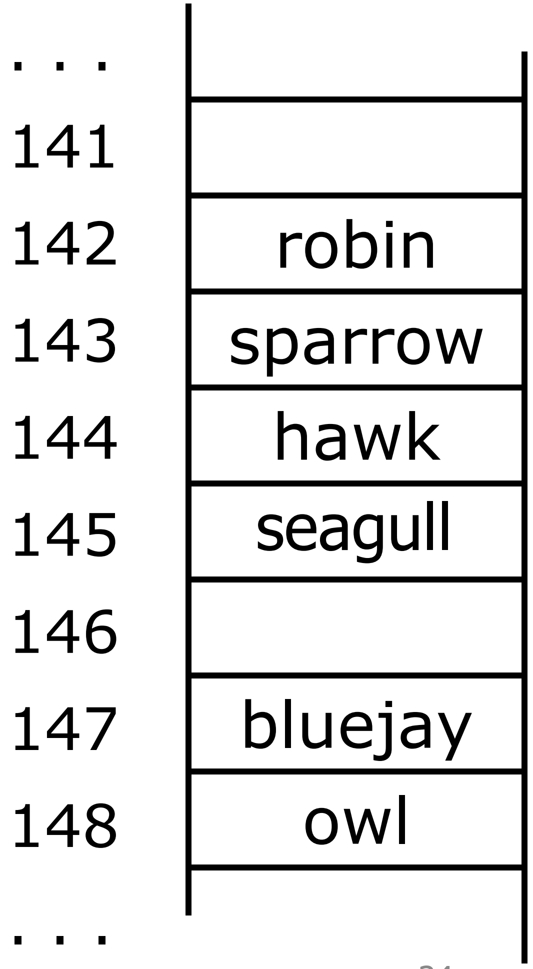
Find with linear probing: *seagull*

- 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



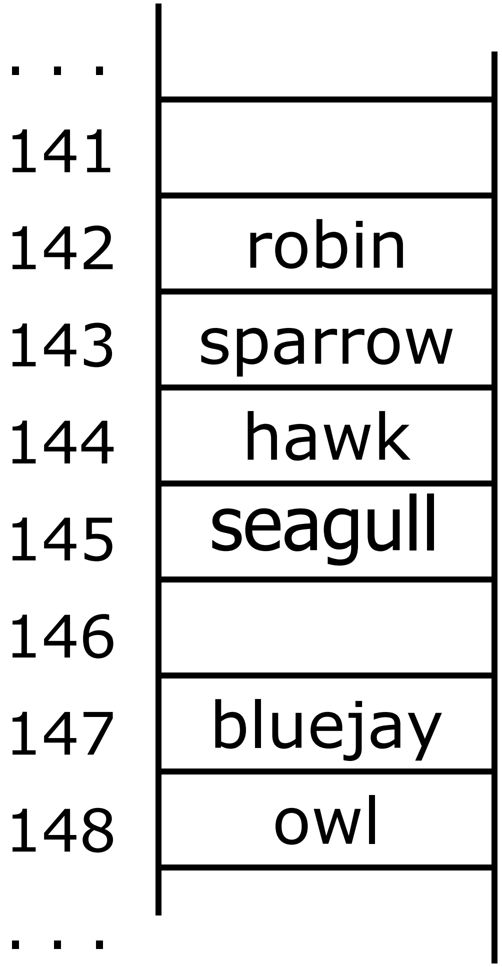
Find with linear probing: *cow*

- 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

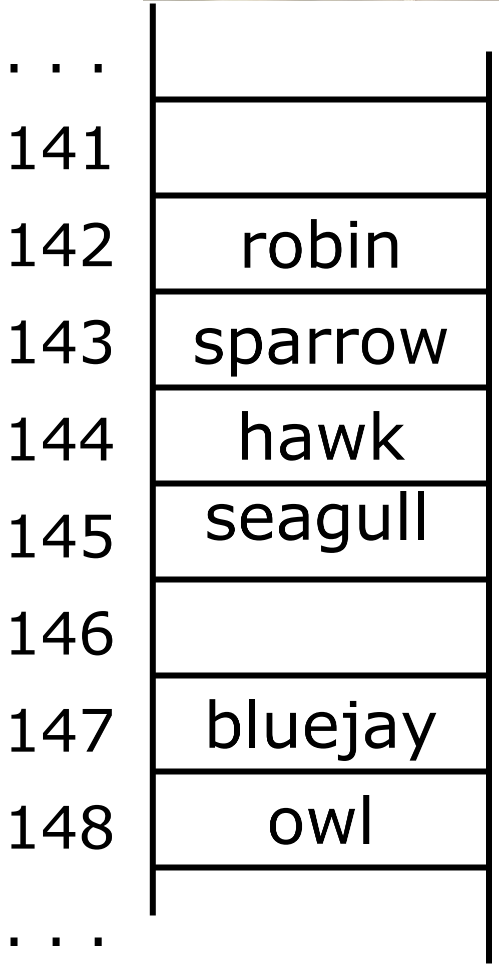
- Suppose you want to add hawk to this hash table
- Also suppose
  - hashCode(hawk) = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- hawk is already in the table, so do nothing



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

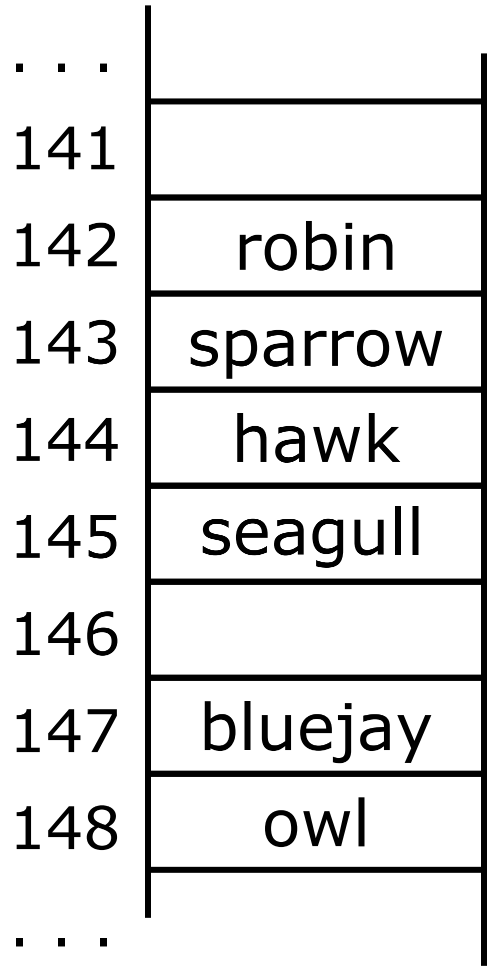
Add with linear probing

- 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 ...)



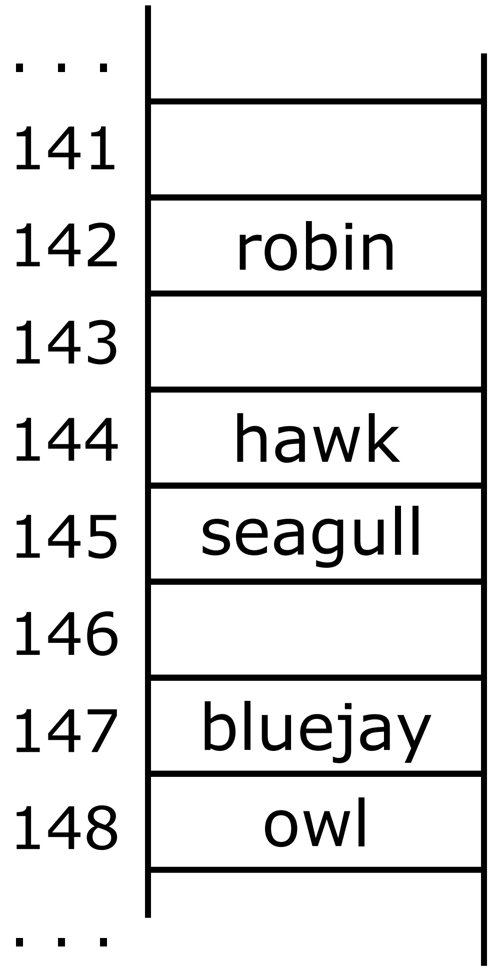
General problems with open addressing: deletions

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



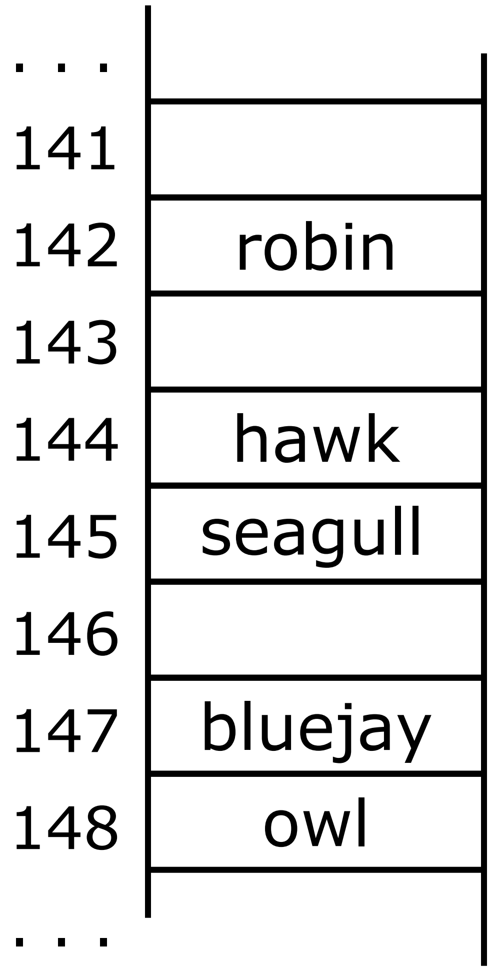
General problems with open addressing: deletions

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



General problems with open addressing: deletions

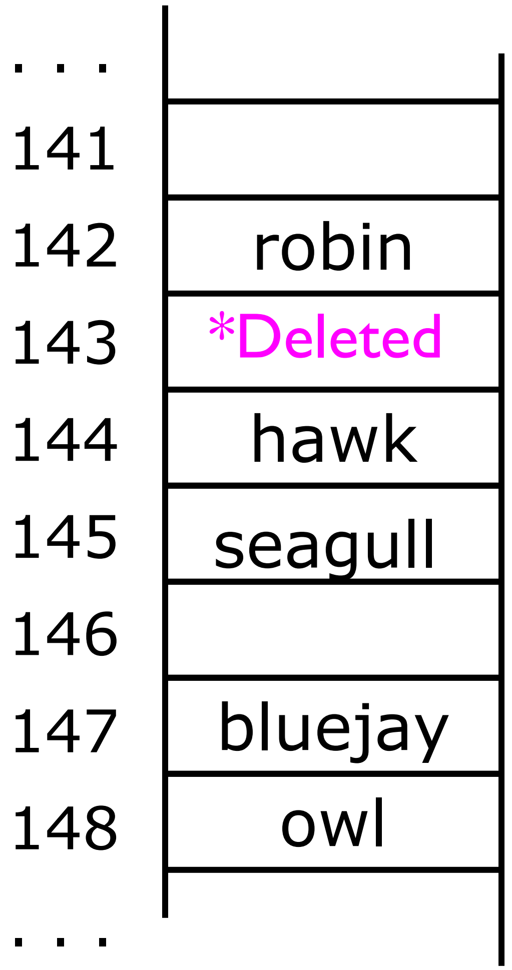
- What happens if we delete sparrow?
  - hashCode(sparrow)=143
  - hashCode(seagull)=143
- Now when searching for seagull we check
  - table[143] is empty
  - We can not find seagull!



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



Problems with linear probing: 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 of search 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
  - Unless n and the table size are mutually prime, some table locations are never even checked

Quadratic probing

➢As before, we first try slot *j=hashCode*.

➢If this slot is occupied, instead of trying slot $j=|(j+1) \text{ MOD M}|$, try slot:

$$j=|(hashCode+j^2) \text{ MOD } M|$$

➢This helps to avoid the clustering problem of a linear probing

➢But it creates its own kind of clustering, where the filled array slots “bounce” in the array in a fixed pattern

➢Even if *M* is a prime, this strategy may not find an empty slot if the array is just half full

Double hashing

- ➢In this case we choose the secondary hash function: *stepHash(k)*.
- ➢If the slot hashCode is occupied, we iteratively try the slots

  j= |(*hashCode*+j**stepHash*) MOD M|
- ➢The secondary hash function *stepHash* is not allowed to return 0
- ➢The common choice (Q is a prime):

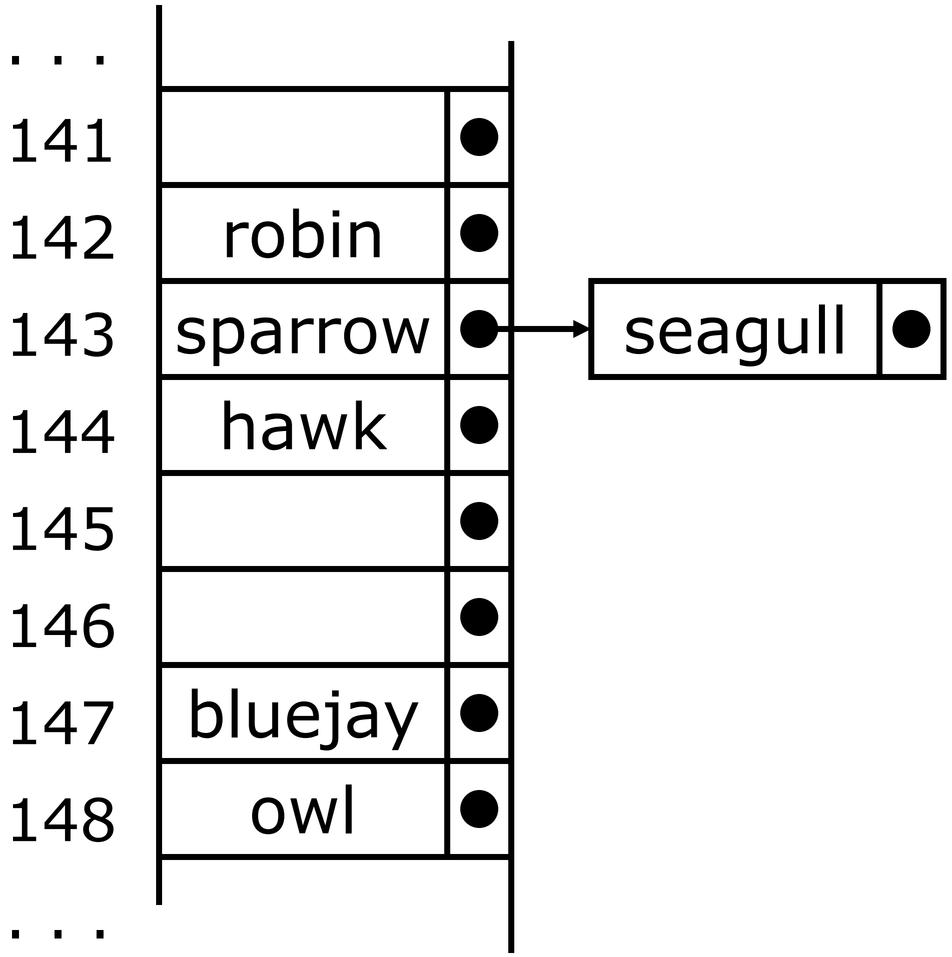
  stepHash(S)=Q-(hashCode(S) mod Q)

Collision resolution strategies

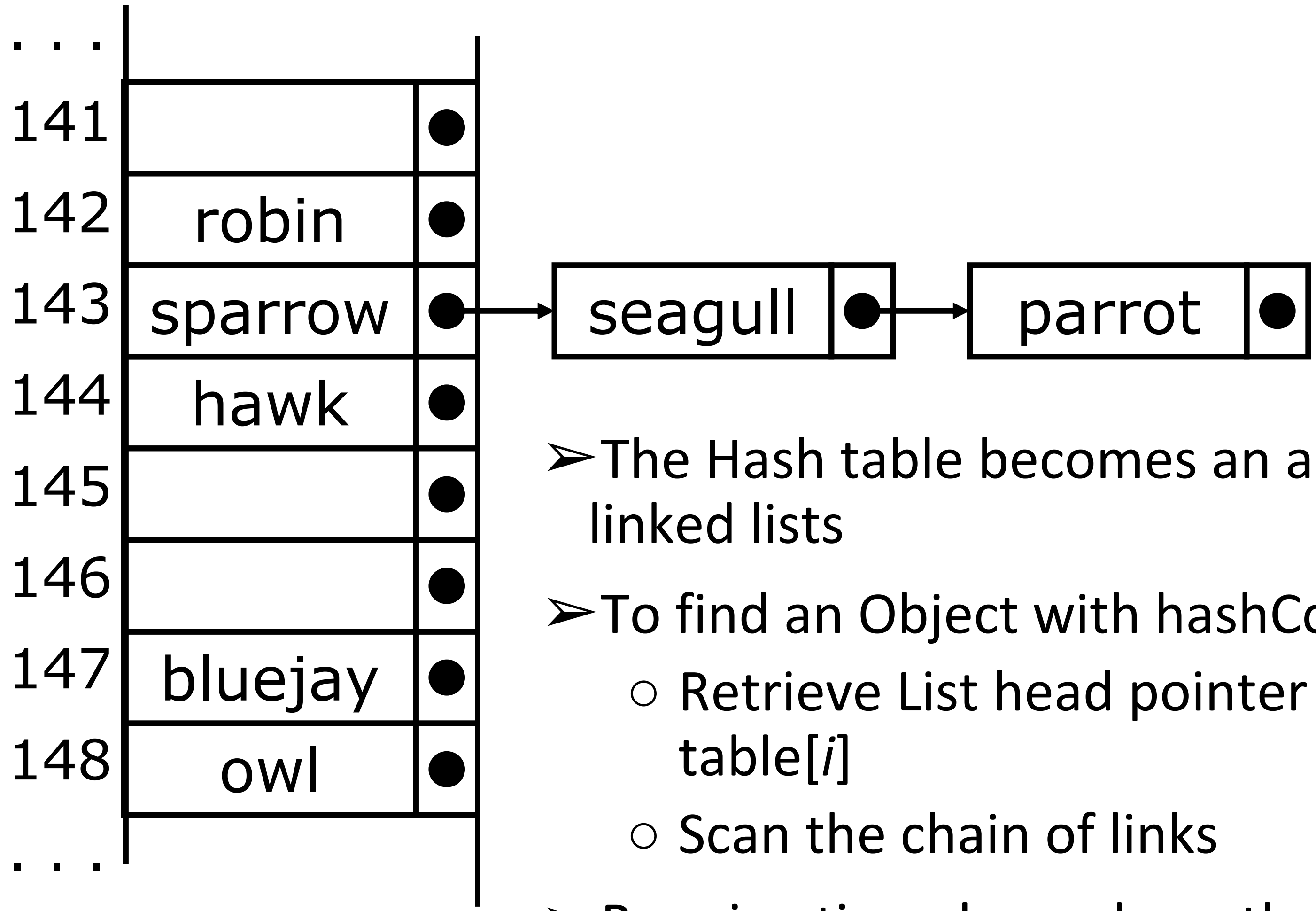
- Open addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing
- Separate chaining

Separate chaining

- ➢The previous solutions used open addressing: all entries went 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: *Find*



- 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

The practical efficiency of collision resolution schemes

- ➢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 the clustering

Common operations: summary

	Worst case			Expected		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	log N	N	N	log N	N/2	N/2
Unsorted array	N	1	N	N/2	N	N/2
Hash table with linear probing	N	N	N	1*	1*	1*
Hash table with separate chaining	N	N	N	1*	1*	1*

*Given a good hash function

Final notes about performance

- ➢Hash tables are actually surprisingly efficient
- ➢Until the array 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 or looking something up in the hash table, is O(1)
- ➢Even if the table is nearly full (leading to occasional long searches), overall efficiency is usually still quite high

Sets and Maps

- ➢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

Sets and Maps

- Sometimes we want a 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

Abstract Data Type: **Set**

Specification

Set is an Abstract Data Type which supports the following operations:

- ➔ **Add (k)** - adds element *k* to the array
- ➔ **Remove (k)** - removes element *k* from the array
- ➔ **HasKey (k)** - returns *True* if element *k* is in the array. Returns *False* otherwise

Abstract Data Type: Map

Specification

Map is an Abstract Data Type which supports the following operations:

- ➔ **Set (*k*, *obj*)** - adds element *obj* to the collection and associates it with key *k*
- ➔ **Get (k)** - returns the object associated with key *k*
- ➔ **HasKey (k)** - returns *True* if there is an object associated with the key *k*. Returns *False* otherwise
- ➔ **Remove (k)** - removes object with key *k* from the collection

- The main goal of both Set and Map: **find element in an array quickly**
- Both use Hashing of keys to achieve this goal

Common implementations

- 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')
```

```
# dictionary (map)
d = {1:0,2:0,3:0…n:0}
# set
s = {1, 2, 3 … n}

if 8 in s:
     print('found')
if 8 in d:
     print('found')
```

Linear time

Constant time

Looping through hash table is impossible

- ➢ The number of cells in the array underlying hash table is *M*
- ➢ Not all cells of this array are occupied, and some may contain multiple elements - when the collisions are resolved by separate chaining
- ➢ Because of that the precise complexity of checking all hash table entries cannot be expressed in terms of the input size *N*
- ➢ Avoid looping through the dictionary if you want guaranteed performance

Example: remove duplicates

```
Algorithm remove_duplicates(array A of size N):
     s : = empty set
     output: = empty array
     for each element x of A:
          if x is not in s:
               s.set(x)
          output.append(x)
     return output
```