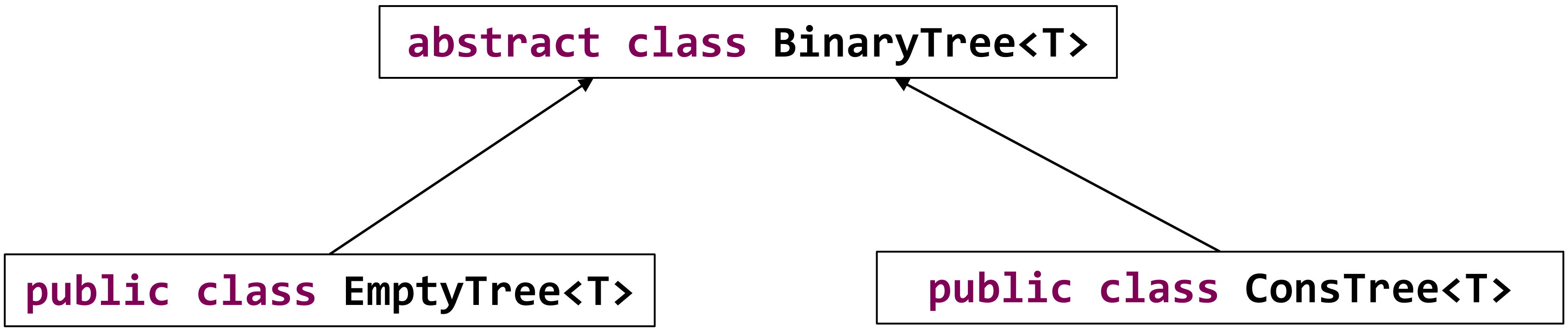
Lab 6. Binary trees
Recursion with objects and polymorphism




```
public class ConsTree<T> extends BinaryTree<T> {

  public ConsTree(T data, BinaryTree<T> left, BinaryTree<T> right) {
    this.data = data;
    this.leftChild = left;
    this.rightChild = right;
  }

  public ConsTree(T data) {
    this(data, new EmptyTree<T>(), new EmptyTree<T>());
  }
```

Default constructor: two children initially set to *EmptyTree*

Example: recursive height()

Superclass: declares method *height*

```
public abstract class BinaryTree<T> {
    public abstract int height();
}
```

```
public class EmptyTree<T> extends BinaryTree<T> {

    @Override
    public int height() {
    return -1;
}
```

Empty tree implements base case

```
public class ConsTree<T> extends BinaryTree<T> {

    @Override
    public int height() {
      return Math.max(this.leftChild.height(),
                      this.rightChild.height()) + 1;
    }
}
```

Non-empty tree implements one recursive step

Corresponding recursive algorithm implemented above:

Algorithm *height* (*tree*)

```
if tree is EmptyTree:
    return -1

return 1 + Max(height(tree.left),
               height(tree.right))
```

Note how inheritance and polymorphism made our code more expressive – child trees are either Real trees or Empty trees, but both are defined as their superclass *BinaryTree*

No need to ask about the base case: when we reach an empty tree node, it automatically performs the base-case operation

Set and Map ADT
Hash tables

Lecture 21

by Marina Barsky

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]

Which data structure to use to implement Set ADT?

Main goal: **locate the element fast**

- *List, Array* - *N* elements are **unsorted** – search requires **O(*N*)** time
- Sorted array - *N* elements are **sorted – O(log *N*)** binary search
  - Can keep sorted elements in *Balanced BST* for quick update operations

It doesn't seem like we can do much better

Searching in time O(1)

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

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

The idea of

Hashing

Problem 1: 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

Problem 1: 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 slot *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

Problem 1: First repeating character

Input: String *S* of length *N*

Output: first repeating character (if any) in *S*

cabare

Run-time O(N)

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**

Problem 2: 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 floats/strings/objects
- For these cases we use a technique of *hashing*: we convert **each 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

Assume the hash function h(x) = x%6. What bucket (position in the array) will 27 hash to?

A. 2

[24, 37, __, __, __, 11]

B. 3

C. 15

D. None of the above

Assume the hash function h(x) = x%6. What bucket (position in the array) will 39 hash to?

A. 2

[24, 37, __, __, __, 11]

B. 3

C. 4

D. None of the above

Case study: 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 bit 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	

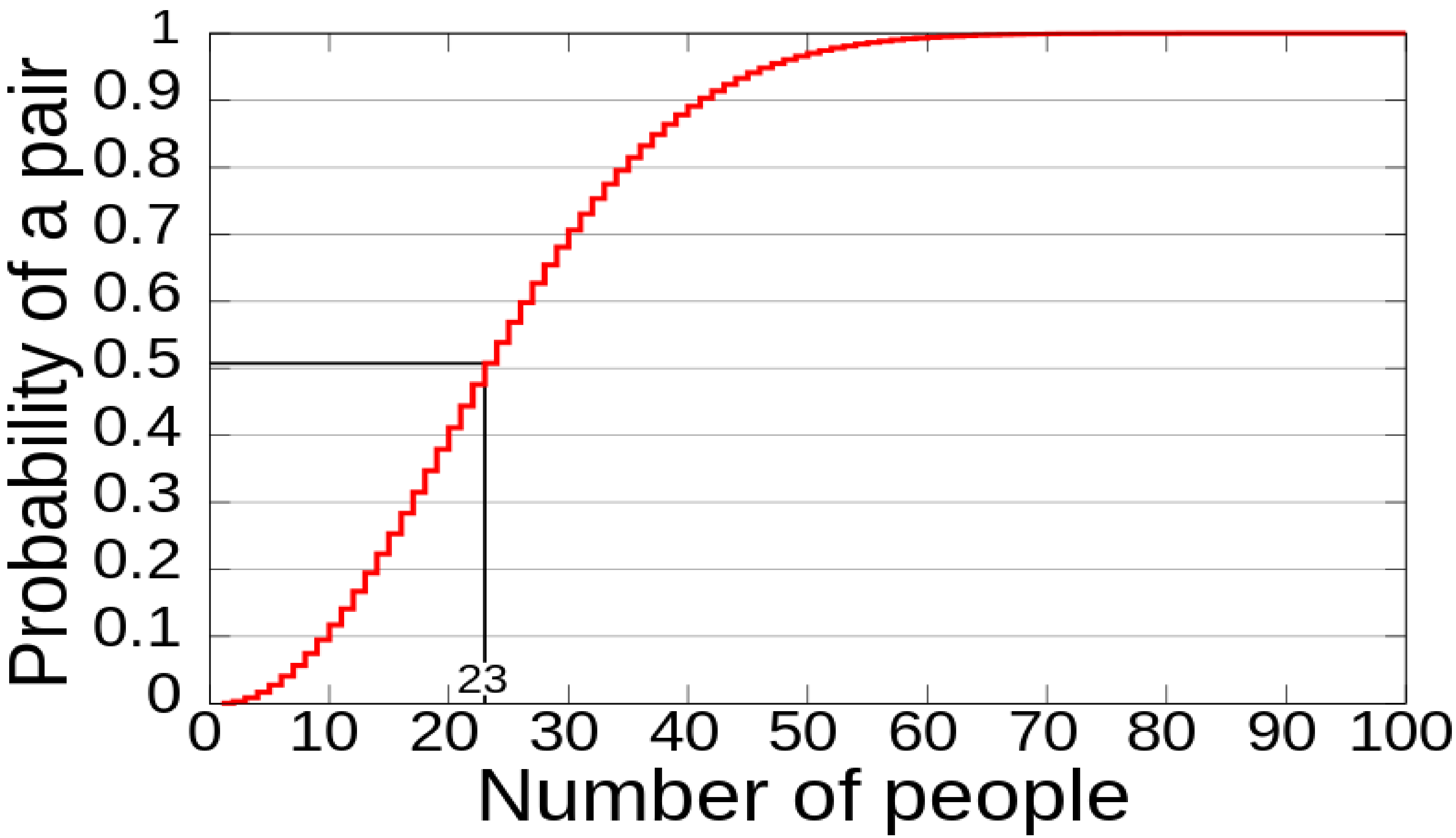
Looking for a good hash function: day of birth

- 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*



http://commons.wikimedia.org/wiki/File:Birthday_Paradox.svg

In search for a **perfect hash function**

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?

In search for a
perfect hash function

- ➢How to come up with this perfect hashing function?
- ➢In general – there is no such magic function
  - ○ In a few specific cases, where all the possible values are known in advance, it is possible to define 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 have told us exactly where to look
- ○ However, the best we can do is a function that tells us i**n** **what area of an array to *start* looking**!

Which of the following hash functions for Strings are legal?

I. Return a random number.
II. Return 0 if the string is of even length, 1 if it's of odd length.
III. Add together all the ASCII values of the characters.

A. All of the above
B. I, II
C. II, III
D. I, III
E. None of the above

Hashing strings by summing up their character values

- It seems like a good idea to map each student 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 $A \neq 1$, 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* for *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 character values, the polynomial hash function introduces interactions between different bits of successive characters that will provoke or spread randomness of the result

How to compute polynomial of degree *N* in time O(*N*)

Horner's method:

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

Let x=31, a_0 … a_n represent n+1 characters of string S:

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

That is ~how *hashCode()* is implemented inside Java *String* class

Java String hashCode()

- ➢Polynomial hashing is quite a good hash function: for different strings it returns mostly different values which are well spread over the range of all possible integers
- ➢This hash function is also very efficient, since we need only *n = length()* steps to compute it

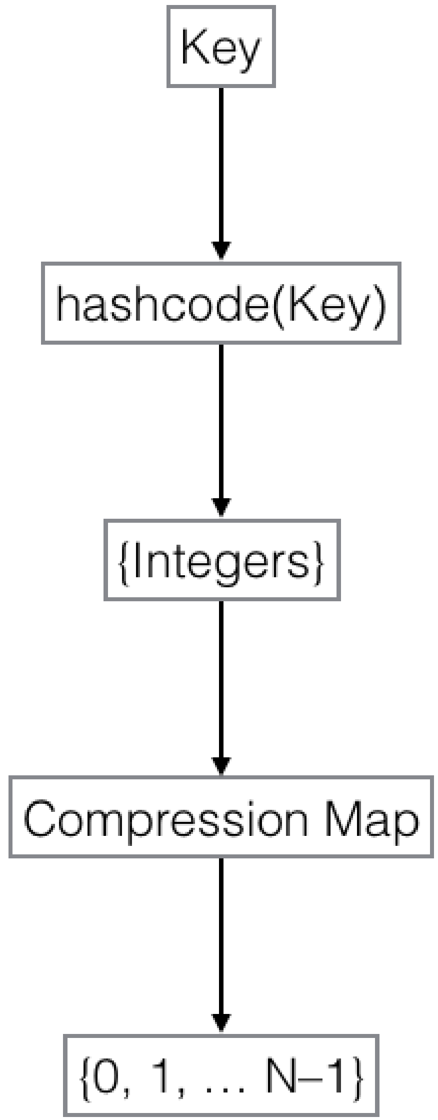
Reducing the range of *hashCode* to the capacity of the array

- ➢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 ~ [0,400000000] 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

Full hashing



Hashing Students to 7 slots

- ➔ 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
  - …

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

```java
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 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
. . .	

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

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

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

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

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
. . .	

General problem with open addressing: deletion

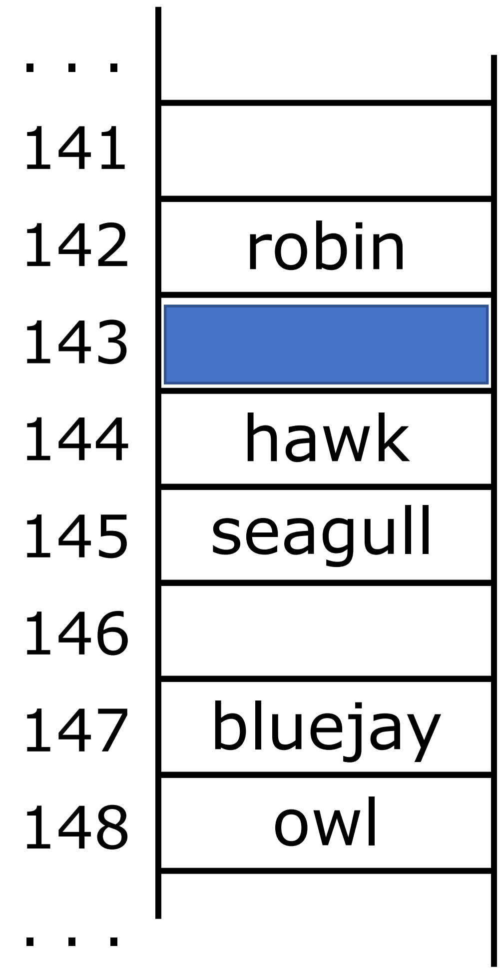
- What happens if we delete sparrow?
  - hashCode(sparrow)=143
  - hashCode(seagull)=143

General problem with open addressing: deletion

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

General problem 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

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

Group Work

- Add the following keys, **in order**, to an **initially empty** Hash table of size N=13. The hash function is hash(x) = x % 13

10, 85, 15, 70, 20, 60, 30, 50, 65, 40, 90, 35

- Resolve collisions with *linear probing*

Another problem with linear probing: clustering

- ➢ One 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 are never even 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.

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

Under quadratic probing, with the following array, where will an item that hashes to position 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 Solution 1: Quadratic probing

- ➢Quadratic probing 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
- ➢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 case 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):

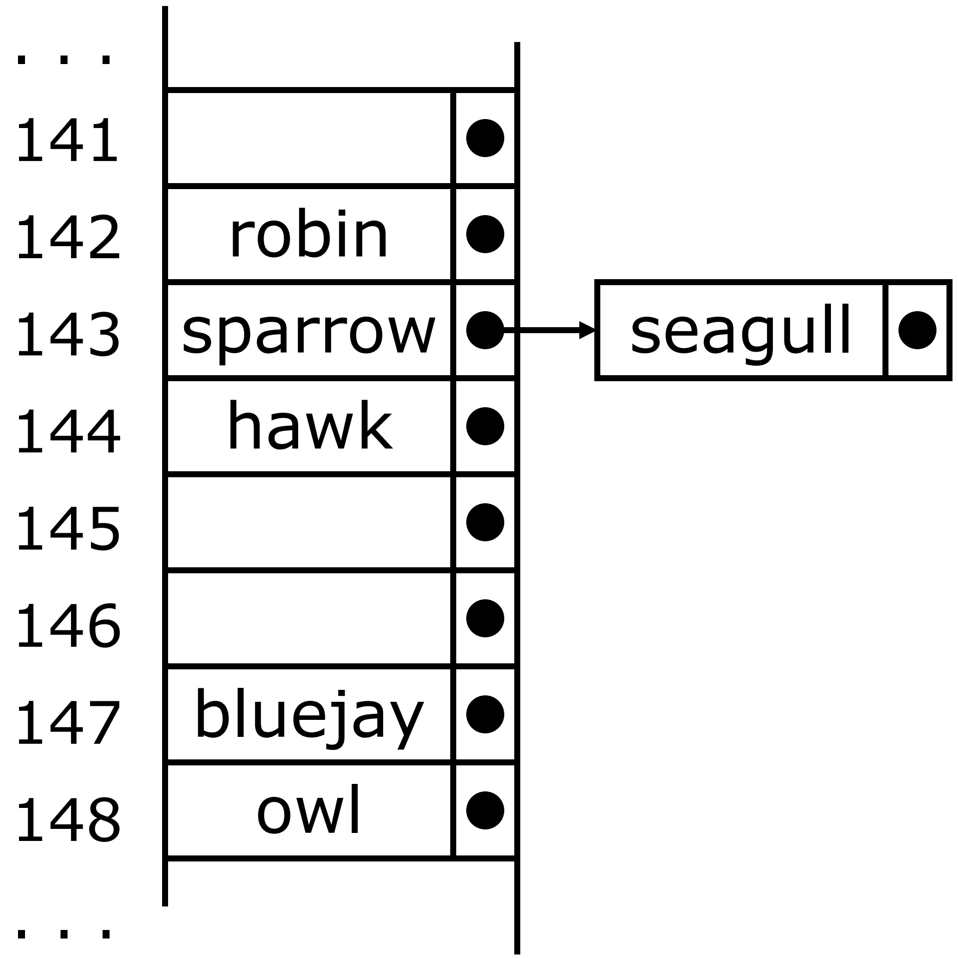
  stepHash(S)=Q-(hashCode(S) mod Q)

Collision resolution strategies

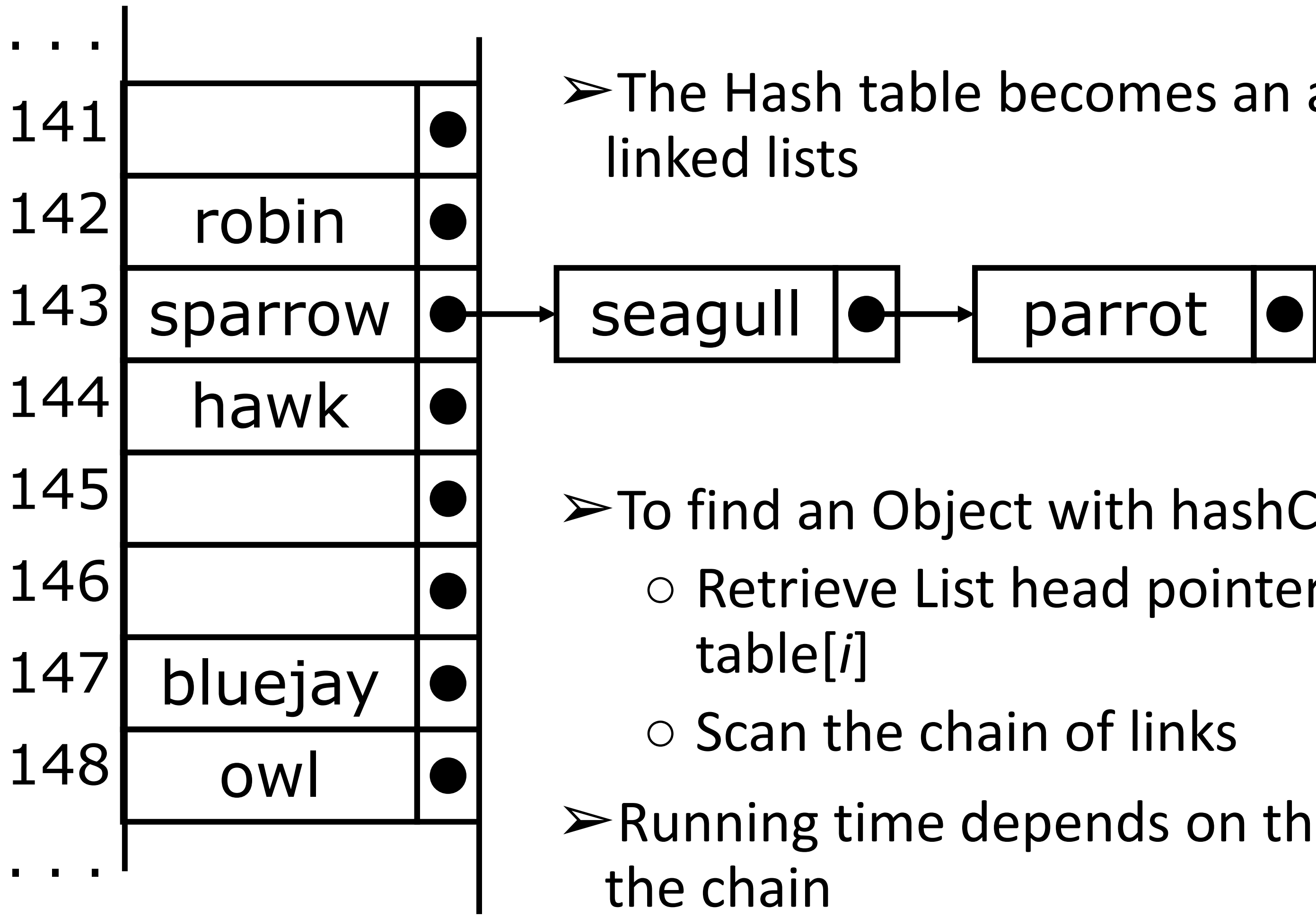
- Open addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing
- Separate chaining

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: *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

"If we are adding a new key to the hash table and the position at *hashCode* is already occupied by a different key, we can place the new key in the next available empty slot in the underlying array."

This collision resolution technique is of type:

A. Open addressing
B. Direct addressing
C. Separate chaining
D. Linear probing
E. More than one is correct

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 the clustering

ADT Set operations: performance

Implementation	Worst case			Expected		
	Search (Contains)	Add	Remove	Search (Contains)	Add	Remove
Balanced Binary tree	log N	log N	log N	log N	log N	log N
Unsorted List (Array or Linked list)	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*

**If we know that new key is unique

*Given a good hash function

Final notes about Hash table 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 when the table is nearly full (leading to occasional long searches), overall efficiency is usually still quite high

Common implementations of Set ADT using Hash Tables

- Set:
  - *unordered_set* in C++
  - *HashSet* in Java
  - *set* in Python

Now you know that in Python:

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

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

Time O(n)

```python
# set
s = {1, 2, 3 … n}

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

Time O(1)

Which tasks can be efficiently solved using the Hash Table implementation of Set ADT?

A. Removing duplicates from the array of integers
B. Quickly checking if student name is in the class roster
C. Testing if all the elements of a given array are unique
D. Given a list of family names and a given family name *s*, counting how many times *s* appears in the array.

We use an unsorted Array List to implement Set ADT.
Choose the row with the correct runtime for each operation

	Remove(k)	Add(k)	Find(k)
A	O(1)	O(n)	O(1)
B	O(log n)	O(log n)	O(log n)
C	O(log n)	O(n)	O(log n)
D	O(n)	O(1)	O(n)

E. None of the above

We use Balanced Binary Search Tree to implement Set ADT.
Choose the row with the correct runtime for each operation

	Remove(k)	Add(k)	Find(k)
A	O(1)	O(n)	O(1)
B	O(log n)	O(log n)	O(log n)
C	O(log n)	O(n)	O(log n)
D	O(n)	O(1)	O(n)

E. None of the above

We use a Hash Table to implement Set ADT.
Choose the row with the correct runtime for each operation

	Remove(k)	Add(k)	Find(k)
A	O(1)	O(n)	O(1)
B	O(log n)	O(log n)	O(log n)
C	O(log n)	O(n)	O(log n)
D	O(n)	O(1)	O(n)

E. None of the above

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: Map

Specification

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

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

- The main efficiency of both Set and Map comes from the ability to **find the item quickly**

Common implementations of Set and Map ADT

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