

# Principles and Practice of Problem Solving: Lecture 9-Data Structures Recap (2)

Lecturer: Haiming Jin

# Outline

- Linked List
- Stack
- Queue
- Sort
- Linear-Time Selection
- Hashing
- Bloom Filter

# Hashing

- Hashing Basics
- Hash Function
- Collision Resolution
- Hash Table Size and Rehashing
- Applications of Hashing

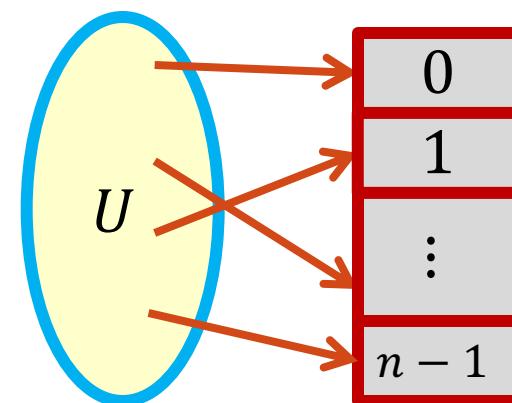
# Hashing: High-Level Idea

- **Setup:** A universe  $U$  of objects
  - E.g., All names, all IP addresses, etc.
  - Generally, very BIG!
- **Goal:** Want to maintain an evolving set  $S \subseteq U$ 
  - E.g., 200 students, 500 IP addresses
  - Generally, of reasonable size.
- Naïve solutions
  1. Array-based solution (index by  $u \in U$ )
    - $\Theta(1)$  operation time, BUT  $\Theta(|U|)$  space.
  2. Linked list-based solution:
    - $\Theta(|S|)$  space, BUT  $\Theta(|S|)$  operation time.

Can we get the best of both solutions?

# Hashing: High-Level Idea

- Solution:
  - Pick an array  $A$  of  $n$  buckets.
    - $n = c|S|$ : a small multiple of  $|S|$ .
  - Choose a hash function  $h: U \rightarrow \{0, 1, \dots, n - 1\}$ 
    - $h$  is fast to compute.
    - The same key is always mapped to the **same** location.
  - Store item  $k$  in  $A[h(k)]$
- The array is called **hash table**
  - An array of **buckets**, where each bucket contains items as assigned by a hash function.
  - $h[k]$  is called the **home bucket** of key  $k$ .



# Hashing Example

- Pairs are: (22,a), (33,b), (3,c), (73,d), (85,e)
- Hash table is **A[0:7]** and table size is **M = 8**
- Hash function is **h[key] = key/11**
- Every item with **key** is stored in the bucket **A[h(key)]**

(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Question: What is the time complexity for  
**find()**, **insert()**, and **remove()**?

# Methods

- **Value find(Key k)** : Return the value whose key is **k**. Return **Null** if none.
- **void insert(Key k, Value v)** : Insert a pair **(k, v)** into the Hash Table.
- **Value remove(Key k)** : Remove the pair with key as **k** from the Hash Table and return its value. Return **Null** if none.

# Hashing Example

- Pairs are: (22,a), (33,b), (3,c), (73,d), (85,e)
- Hash table is **A[0:7]** and table size is **M = 8**
- Hash function is **h[key] = key/11**
- Every item with **key** is stored in the bucket **A[h(key)]**

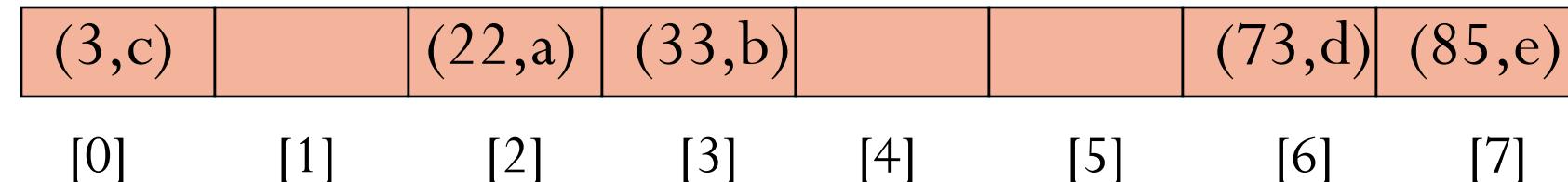
(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Question: What is the time complexity for  
**find()**, **insert()**, and **remove()**?

O(1)

Achieves both fast runtime and efficient memory usage

# What Can Go Wrong?



- Where does (35, g) go?
- Problem: The home bucket for (35, g) is already occupied!
  - This is a “**collision**”.

# Collision and Collision Resolution

- Collision occurs when the hash function maps two or more items—all having **different** search keys—into the **same** bucket.
- What to do when there is a collision?
  - **Collision-resolution scheme**: assigns distinct locations in the hash table to items involved in a collision.
- Two major schemes:
  - Separate chaining
  - Open addressing

# Hash

- Hashing Basics
- Hash Function
- Collision Resolution
- Hash Table Size and Rehashing
- Applications of Hashing

# Hash Function Design Criteria

- Must compute a bucket for every key in the universe.
- Must compute the same bucket for the same key.
- Should be easy and quick to compute.
- Minimizes collision  The hardest criterion
  - Spread keys out evenly in hash table
  - **Gold standard:** **completely random hashing**
    - The probability that a randomly selected key has bucket  $i$  as its home bucket is  $1/n$ ,  $0 \leq i < n$ .
    - Completely random hashing **minimizes** the likelihood of an collision when keys are selected at random.
    - However, completely random hashing is **infeasible** due to the need to remember the random bucket.

# Bad Hash Functions

- Example: keys = phone number in China (11 digits)
  - $|U| = 10^{11}$
  - **Terrible** hash function:  $h(key) =$  first 3 digits of  $key$ , i.e., area code
    - The keys are not spread out evenly. Buckets 010, 021 may have a lot of keys mapped to them, while some buckets have no keys.
  - **Mediocre** hash function:  $h(key) =$  last 3 digits of  $key$ .
    - Still vulnerable to patterns in last 3 digits.

# Hash Functions

- Hash function ( $h(key)$ ) maps key to buckets in two steps:
  1. Convert key into an integer in case the key is not an integer.
    - A function  $t(key)$  which returns an integer value, known as **hash code**.
  2. **Compression map**: Map an integer (hash code) into a home bucket.
    - A function  $c(hashcode)$  which gives an integer in the range  $[0, n - 1]$ , where  $n$  is the number of buckets in the table.
- In summary,  $h(key) = c(t(key))$ , which gives an index in the table.

# Map Non-integers into Hash Code

- String: use the ASCII (or UTF-8) encoding of each char and then perform arithmetic on them.
- Floating-point number: treat it as a string of bits.
- Images, (viral) code snippets, (malicious) Web site URLs: in general, treat the representation as a bit-string, using all of it or **extracting** parts of it (i.e., [www.sjtu.edu.cn](http://www.sjtu.edu.cn)).

# Strings to Integers

- Simple scheme: adds up all the ASCII codes for all the chars in the string.
  - Example:  $t(\text{"He"}) = 72 + 101 = 173.$
- Not good. Why?
  - Consider English words “post”, “pots”, “spot”, “stop”, “tops”.

# Strings to Integers

- A better strategy: Polynomial hash code taking **positional** info into account.

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

where  $a$  is a constant.

- If  $a = 33$ , the hash codes for “post” and “stop” are

$$t(\text{post}) = 112 \cdot 33^3 + 111 \cdot 33^2 + 115 \cdot 33 + 116 = 4149734$$

$$t(\text{stop}) = 115 \cdot 33^3 + 116 \cdot 33^2 + 111 \cdot 33 + 112 = 4262854$$

# Strings to Integers

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

- Good choice of  $a$  for English words: 31, 33, 37, 39, 41
  - What does it mean for  $a$  to be a **good** choice? Why are these particular values **good**?
  - Answer: according to statistics on 50,000 English words, each of these constants will produce less than 7 collisions.
- In Java, its **String** class has a built-in **hashCode()** function. It takes  $a = 31$ . Why?
  - Multiplication by 31 can be replaced by a shift and a subtraction for **better performance**: **31\*i == (i << 5) - i**

Hash function criteria: Should be easy and quick to compute.

# Compression Map

- Map an integer (hash code) into a home bucket.
- The most common method is by **modulo arithmetic**.

**homeBucket = c(hashcode) = hashcode % n**

where  $n$  is the **number of buckets** in the hash table.

- Example: Pairs are (22,a), (33,b), (3,c), (55,d), (79,e). Hash table size is 7.

	(22,a)	(79,e)	(3,c)		(33,b)	(55,d)
[0]	[1]	[2]	[3]	[4]	[5]	[6]

# Hashing by Modulo

- In practice, keys of an application tend to have a specific pattern
  - For example, memory address in computer is multiple of 4.
- The choice of the hash table size  $n$  will affect the distribution of home buckets.

# Hashing by Modulo

- Suppose the keys of an application are more likely to be mapped into even integers.
  - E.g., memory address is always a multiple of 4.
- When the hash table size  $n$  is an **even** number, **even** integers are hashed into **even** home buckets.
  - E.g.,  $n = 14$ :  $20\%14 = 6$ ,  $30\%14 = 2$ ,  $8\%14 = 8$
- The bias in the keys results in a bias toward the **even** home buckets.
  - All **odd** buckets are guaranteed to be empty.
  - The distribution of home buckets is not uniform!

# Hashing by Modulo

- However, when the hash table size  $n$  is **odd**, even (or odd) integers may be hashed into both odd and even home buckets.
  - E.g.,  $n = 15$ :  $20\%15 = 5$ ,  $30\%15 = 0$ ,  $8\%15 = 8$   
 $15\%15 = 0$ ,  $3\%15 = 3$ ,  $23\%15 = 8$
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
  - Better chance of uniform distribution of home buckets.
- So **do not** use an even hash table size  $n$ .

# Hashing by Modulo

- Similar **biased** distribution of home buckets happens in practice when the hash table size  $n$  is a multiple of small prime numbers.
- The effect of each prime divisor  $p$  of  $n$  **decreases** as  $p$  gets **larger**.
- Ideally, choose the hash table size  $n$  as a **large prime number**.

# Hash

- Hashing Basics
- Hash Function
- Collision Resolution
- Hash Table Size and Rehashing
- Applications of Hashing

# Outline

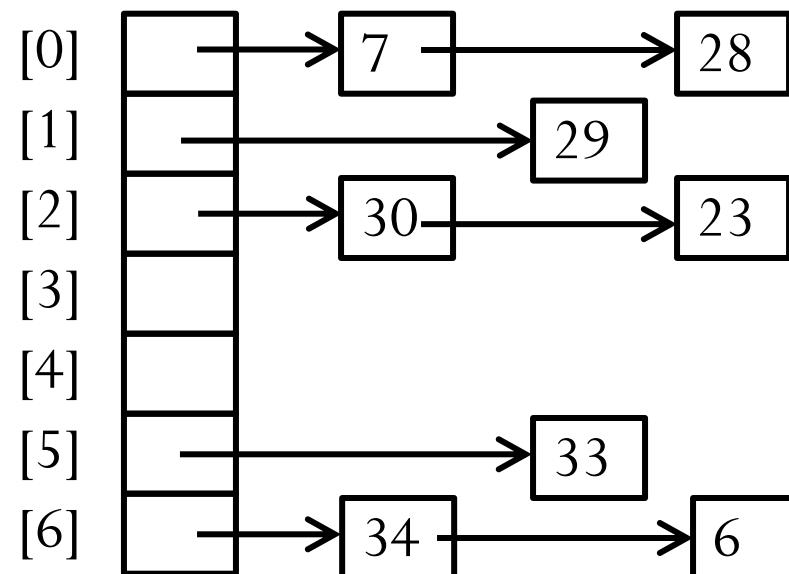
- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

# Collision Resolution Scheme

- **Collision-resolution scheme:** assigns distinct locations in the hash table to items involved in a collision.
- Two major scheme:
  - Separate chaining
  - Open addressing

# Separate Chaining

- Each bucket keeps a **linked list** of all items whose home buckets are that bucket.
- Example: Put pairs whose keys are 6, 23, 34, 28, 29, 7, 33, 30 into a hash table with  $n = 7$  buckets.
  - **homeBucket = key % 7**
  - **Note:** we insert object at the beginning of a linked list.



# Separate Chaining

- **Value find(Key key)**
  - Compute  $k = h(key)$
  - Search in the linked list located at the k-th bucket (e.g., check every entry) with the key.
- **void insert(Key key, Value value)**
  - Compute  $k = h(key)$
  - Search in the linked list located at the k-th bucket. If found, update its value; otherwise, insert the pair at the beginning of the linked list in  $O(1)$  time.

# Separate Chaining

- **Value remove(Key key)**
  - Compute  $k = h(key)$
  - Search in the linked list located at the k-th bucket. If found, remove that pair.

# Outline

- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

# Open Addressing

- Reuse empty space in the hash table to hold colliding items.
- To do so, search the hash table in some systematic way for a bucket that is empty.
  - Idea: we use a sequence of hash functions  $h_0, h_1, h_2, \dots$  to probe the hash table until we find an empty slot.
    - I.e., we **probe** the hash table buckets mapped by  $h_0(\text{key}), h_1(\text{key}), \dots$ , in sequence, until we find an empty slot.
    - Generally, we could define  $h_i(x) = h(x) + f(i)$

# Open Addressing

- Three methods:

- Linear probing:

$$h_i(x) = (h(x) + i) \% n$$

- Quadratic probing:

$$h_i(x) = (h(x) + i^2) \% n$$

- Double hashing:

$$h_i(x) = (h(x) + i*g(x)) \% n$$

n is the hash table size

# Linear Probing

$$h_i(\text{key}) = (h(\text{key}) + i) \% n$$

- Apply hash function  $h_0, h_1, \dots$ , in sequence until we find an empty slot.
  - This is equivalent to doing a linear search from  $h(\text{key})$  until we find an empty slot.
- Example: Hash table size  $n = 9$ ,  $h(\text{key}) = \text{key} \% 9$ 
  - Thus  $h_i(\text{key}) = (\text{key} \% 9 + i) \% 9$
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence

	1	11			5			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

How about 2?

# Linear Probing

## Example

- Hash table size  $n = 9$ ,  $h(key) = key \% 9$ 
  - Thus  $h_i(key) = (key \% 9 + i) \% 9$
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence.

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- $h_0(2) = 2$ . Not empty!
- So we try  $h_1(2) = 3$ . It is empty, so we insert there!
- $h_0(21) = 3$ . Not empty!
- $h_1(21) = 4$ . It is empty, so we insert there!
- $h_0(31) = 4$ . Not empty!
- $h_1(31) = 5$ . Not empty!
- $h_2(31) = 6$ . It is empty, so we insert there!

# Linear Probing

find()

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- With linear probing  $h_i(\text{key}) = (\text{key} \% 9 + i) \% 9$ 
  - How will you **search** an item with key = 31?
  - How will you **search** an item with key = 10?
- Procedure: probe in the buckets given by  $h_0(\text{key}), h_1(\text{key}), \dots$ , in sequence **until**
  - we find the key,
  - or we find an empty slot, which means the key is not found.

# Linear Probing

remove()

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

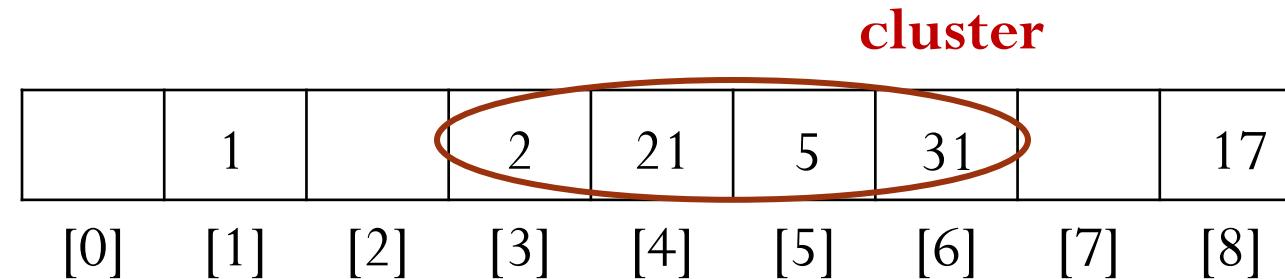
- With linear probing  $h_i(\text{key}) = (\text{key} \% 9 + i) \% 9$ 
  - How will you **remove** an item with key = 11?
  - If we just find 11 and delete it, will this work?

	1		2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

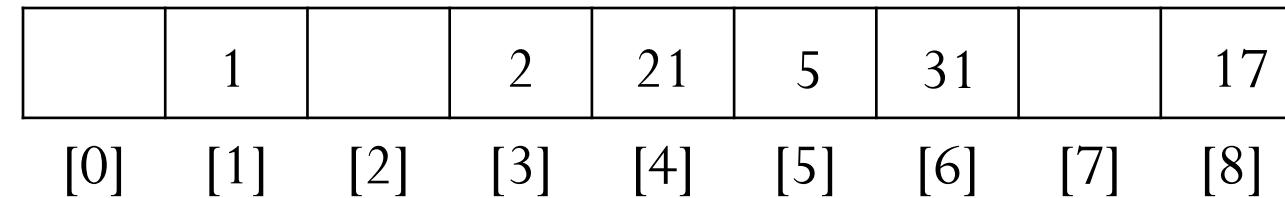
What is the result for searching key = 2 with the above hash table?

# Linear Probing

remove()



- After deleting 11, we need to **rehash** the following “cluster” to fill the vacated bucket.
- However, we cannot move an item **beyond** its **actual** hash position. In this example, 5 cannot be moved ahead.



# Linear Probing

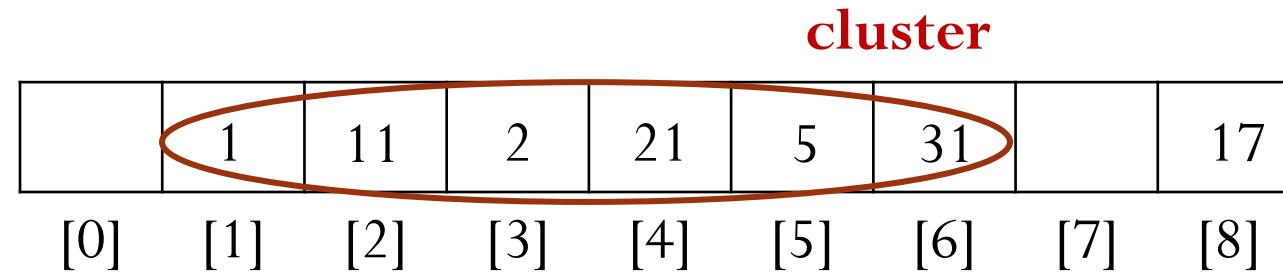
Alternative implementation of remove()

	1	del	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- **Lazy deletion**: we mark deleted entry as “**deleted**”.
  - “deleted” is not the same as “empty”.
  - Now each bucket has three states: “occupied”, “empty”, and “deleted”.
- We can overwrite the “deleted” entry when inserting.
- When we **search**, we will keep looking if we encounter a “deleted” entry.

# Linear Probing

## Clustering Problem



- Clustering: when **contiguous** buckets are all occupied.
- **Claim:** Any hash value inside the cluster adds to the end of that cluster.
- Problems with a **large** cluster:
  - It becomes more likely that the next hash value will collide with the cluster.
  - Collisions in the cluster get more expensive to resolve.

# Linear Probing

## Clustering Problem

- Assuming input size  $N$ , table size  $2N$ :
  - What is the best-case cluster distribution?

<b>X</b>		<b>X</b>		<b>X</b>		<b>X</b>	
----------	--	----------	--	----------	--	----------	--

- What is the worst-case cluster distribution?

		<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>		
--	--	----------	----------	----------	----------	--	--

# Which Statements Are Correct?

- Assuming input size  $N$ , table size  $2N$ . Analyze the average number of probes to find an empty slot for best-case and worst-case clusters. Select all the correct answers.
- A. The average number for best-case cluster is 1.5.
- B. The average number for best-case cluster is 1.
- C. The average number for worst-case cluster is roughly  $\frac{1}{4}N$ .
- D. The average number for worst-case cluster is roughly  $\frac{1}{2}N$ .

Best Case	X		X		X		X	
-----------	---	--	---	--	---	--	---	--

Worst Case			X	X	X	X		
------------	--	--	---	---	---	---	--	--

# Outline

- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

# Quadratic Probing

$$h_i(\text{key}) = (h(\text{key}) + i^2) \% n$$

- It is less likely to form large clusters.
- Example: Hash table size  $n = 7$ ,  $h(\text{key}) = \text{key} \% 7$ 
  - Thus  $h_i(\text{key}) = (\text{key} \% 7 + i^2) \% 7$
  - Suppose we insert 9, 16, 11, 2 in sequence.

		9	16	11		2
[0]	[1]	[2]	[3]	[4]	[5]	[6]

- $h_0(16) = 2$ . Not empty!
- $h_1(16) = 3$ . It is empty, so we insert there.
- $h_0(2) = 2$ . Not empty!
- $h_1(2) = 3$ . Not empty!
- $h_2(2) = 6$ . It is empty, so we insert there.

# Problem of Quadratic Probing

- However, sometimes we will never find an empty slot even if the table isn't full!
- Luckily, if the **load factor**  $L \leq 0.5$ , we are guaranteed to find an empty slot.
  - Definition: given a hash table with  $n$  buckets that stores  $m$  objects, its **load factor** is

$$L = \frac{m}{n} = \frac{\text{\#objects in hash table}}{\text{\#buckets in hash table}}$$

# More on Load Factor of Hash Table

- Question: which collision resolution strategy is feasible for load factor larger than 1?
  - Answer: separate chaining.
  - Note: for open addressing, we require  $L \leq 1$ .
- Claim:  $L = O(1)$  is a necessary condition for operations to run in constant time.

# Double Hashing

$$h_i(x) = (h(x) + i * g(x)) \% n$$

- Uses 2 distinct hash functions.
- Increment **differently** depending on the key.
  - If  $h(x) = 13, g(x) = 17$ , the probe sequence is 13, 30, 47, 64, ...
  - If  $h(x) = 19, g(x) = 7$ , the probe sequence is 19, 26, 33, 40, ...
  - For linear and quadratic probing, the incremental probing patterns are **the same** for all the keys.

# Double Hashing

## Example

- Hash table size  $n = 7$ ,  $h(key) = key \% 7$ ,  
 $g(key) = (5 - key) \% 5$ 
  - Thus  $h_i(key) = (key \% 7 + (5 - key) \% 5 * i) \% 7$
  - Suppose we insert 9, 16, 11, 2 in sequence.

		9		11	2	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]

- $h_0(16) = 2$ . Not empty!
- $h_1(16) = 6$ . It is empty, so we insert there.
- $h_0(2) = 2$ . Not empty!
- $h_1(2) = 5$ . It is empty, so we insert there.

# Outline

- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

# Performance of Open Addressing

- Hard to analyze rigorously.
- The runtime is dominated by the number of comparisons.
- The number of comparisons depends on the load factor  $L$ .
- Define the expected number of comparisons in an **unsuccessful search** as  $U(L)$ .
- Define the expected number of comparisons in a **successful search** as  $S(L)$ .

# Expected Number of Comparisons

- Linear probing

$$U(L) = \frac{1}{2} \left[ 1 + \left( \frac{1}{1-L} \right)^2 \right]$$
$$S(L) = \frac{1}{2} \left[ 1 + \frac{1}{1-L} \right]$$

$L$	$U(L)$	$S(L)$
0.5	2.5	1.5
0.75	8.5	2.5
0.9	50.5	5.5

$L \leq 0.75$  is recommended.

# Expected Number of Comparisons

- Quadratic probing and double hashing

$$U(L) = \frac{1}{1 - L}$$
$$S(L) = \frac{1}{L} \ln \frac{1}{1 - L}$$

$L$	$U(L)$	$S(L)$
0.5	2	1.4
0.75	4	1.8
0.9	10	2.6

# Which Strategy to Use?

- Both separate chaining and open addressing are used in real applications.
- Some basic guidelines:
  - If space is important, better to use open addressing.
  - If need removing items, better to use separate chaining.
    - **remove ()** is tricky in open addressing.
  - In mission critical application, prototype both and compare.

# Hash

- Hashing Basics
- Hash Function
- Collision Resolution
- Hash Table Size and Rehashing
- Applications of Hashing

# Determine Hash Table Size

- First, given **performance** requirements, determine the maximum permissible **load factor**.
- Example: we want to design a hash table based on **linear probing** so that on average
  - An **unsuccessful** search requires no more than 13 compares.
  - A **successful** search requires no more than 10 compares.

$$U(L) = \frac{1}{2} \left[ 1 + \left( \frac{1}{1-L} \right)^2 \right] \leq 13 \Rightarrow L \leq \frac{4}{5}$$

$$L \leq \frac{4}{5}$$

$$S(L) = \frac{1}{2} \left[ 1 + \frac{1}{1-L} \right] \leq 10 \Rightarrow L \leq \frac{18}{19}$$

# Determine Hash Table Size

- For a fixed table size, estimate maximum number of items that will be inserted.
- Example: no more than 1000 items.
  - For load factor  $L = \frac{|S|}{n} \leq \frac{4}{5}$ , table size
$$n \geq \frac{5}{4} \cdot 1000 = 1250$$
  - Pick  $n$  as a **prime** number. For example,  $n = 1259$ .

However, sometimes there is no limit on the number of items to be inserted.

# Rehashing

## Motivation

- With more items inserted, the load factor increases. At some point, it will exceed the threshold ( $4/5$  in the previous example) determined by the performance requirement.
- For the separate chaining scheme, the hash table becomes inefficient when load factor  $L$  is too high.
  - If the size of the hash table is fixed, search performance deteriorates with more items inserted.
- Even worse, for the open addressing scheme, when the hash table becomes full, we **cannot** insert a new item.

# Rehashing

- To solve these problems, we need to **rehash**:
  - Create a larger table, scan the current table, and then insert items into new table using the new hash function.
  - Note: The order is from the beginning to the end of the current table. Not original insertion order.
- We can approximately double the size of the current table.
- Observation: The single operation of rehashing is time-consuming. However, it does not occur frequently.
  - How should we justify the time complexity of rehashing?

# Amortized Analysis

- **Amortized analysis:** A method of analyzing algorithms that considers the entire sequence of operations of the program.
  - The idea is that while certain operations may be costly, they don't occur frequently; the less costly operations are much more than the costly ones in the long run.
  - Therefore, the cost of those expensive operations is **averaged** over a sequence of operations.
  - In contrast, our previous complexity analysis only considers a single operation, e.g., insert, find, etc.

# Amortized Analysis of Rehashing

- Suppose the threshold of the load factor is 0.5. We will double the table size after reaching the threshold.
- Suppose we start from an empty hash table of size  $2M$ .
- Assume  $O(1)$  operation to insert up to  $M$  items.
  - Total cost of inserting the first  $M$  items:  $O(M)$
- For the  $(M + 1)$ -th item, create a new hash table of size  $4M$ .
  - Cost:  $O(1)$
  - Rehash all  $M$  items into the new table. Cost:  $O(M)$
  - Insert new item. Cost:  $O(1)$

Total cost for inserting  $M + 1$  items is  $2O(M) + 2O(1) = O(M)$ .

# Amortized Analysis of Rehashing

Total cost for inserting  $M + 1$  items is  $O(M)$ .

- The average cost to insert  $M + 1$  items is  $O(1)$ .
  - Rehashing cost is **amortized** over individual inserts.

# Hash

- Hashing Basics
- Hash Function
- Collision Resolution
- Hash Table Size and Rehashing
- Applications of Hashing

# Application: De-Duplication

- Given: a stream of objects
  - Linear scan through a huge file
  - Or, objects arriving in real time
- Goal: remove duplicates (i.e., keep track of unique objects)
  - E.g., report unique visitors to website
  - Or, avoid duplicates in search result
- Solution: when new object  $x$  arrives,
  - Look  $x$  in hash table  $H$
  - If not found, insert  $x$  into  $H$

# Application: 2-SUM Problem

- Given: an unsorted array A of  $n$  integers. Target sum  $t$ .
- Goal: determine whether or not there are two numbers  $x$  and  $y$  in A with

$$x + y = t$$

1. Naïve solution: exhaustive search of pairs of number
  - Time:  $\Theta(n^2)$
2. Better solution: 1) Sort A; 2) For each  $x$  in A, look for  $t - x$  in A via binary search.
  - Time:  $\Theta(n \log n)$
3. Best: 1) Insert elements of A into hash table H; 2) For each  $x$  in A, search for  $t - x$ .
  - Time:  $\Theta(n)$

# Further Immediate Application

- Spellchecker
- Database

# Hash Table

## Summary

- Choice of the hash function.
- Collision resolution scheme.
- Hash table size and rehashing.
- Time complexity of **hash table** versus **sorted array**
  - `insert()`:  $O(1)$  versus  $O(n)$
  - `find()`:  $O(1)$  versus  $O(\log n)$
- When **NOT** to use hash?
  - **Rank search**: return the k-th largest item.
  - **Sort**: return the values in order.

# Outline

- Linked List
- Stack
- Queue
- Sort
- Linear-Time Selection
- Hashing
- Bloom Filter

# Bloom Filter

- Invented by Burton Bloom in 1970
- Supports **fast insert** and **find**
- Comparison to hash tables:
  - Pros: more space efficient
  - Cons:
    1. Can't store an associated object
    2. No deletion (There are variations support deletion, but this operation is complicated)
    3. Small **false positive** probability: may say x has been inserted even if it hasn't been
      - But no false negative (x is inserted, but says not inserted)

# Bloom Filter Applications

- When to use bloom filter?
  - If the false positive is not a concern, no deletion, and you look for space efficiency
- Original application: spell checker
  - 40 years ago, space is a big concern, it's OK to tolerate some error
- Canonical application: list of forbidden passwords
  - Don't care about the false positive issue
- Modern applications: network routers
  - Limited memory, need to be fast
  - Applications include keeping track of blocked IP address, keeping track of contents of caches, etc.

## Bloom Filter Implementation: Components

- An array of  $n$  **bits**. Each bit 0 or 1
  - $n = b|S|$ , where  $b$  is small real number. For example,  $b \approx 8$  for 32-bit IP address  
(That's why it is space efficient)
- $k$  hash functions  $h_1, \dots, h_k$ , each mapping inside  $\{0, 1, \dots, n - 1\}$ .
  - $k$  usually small.

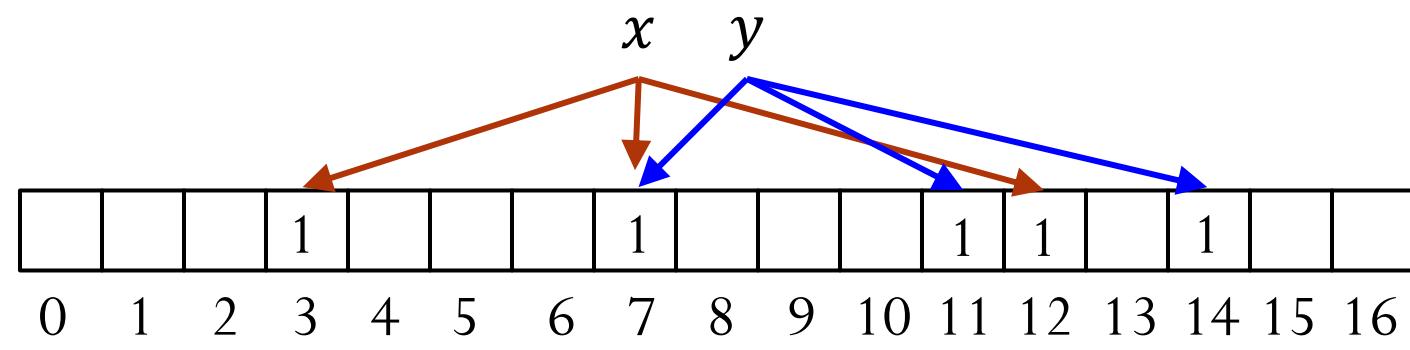
# Bloom Filter Insert

- Initially, the array is all-zero.
- Insert  $x$ : For  $i = 1, 2, \dots, k$ , set  $A[h_i(x)] = 1$ 
  - No matter whether the bit is 0 or 1 before

Example:  $n = 17$ , 3 hash functions

$$h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$$

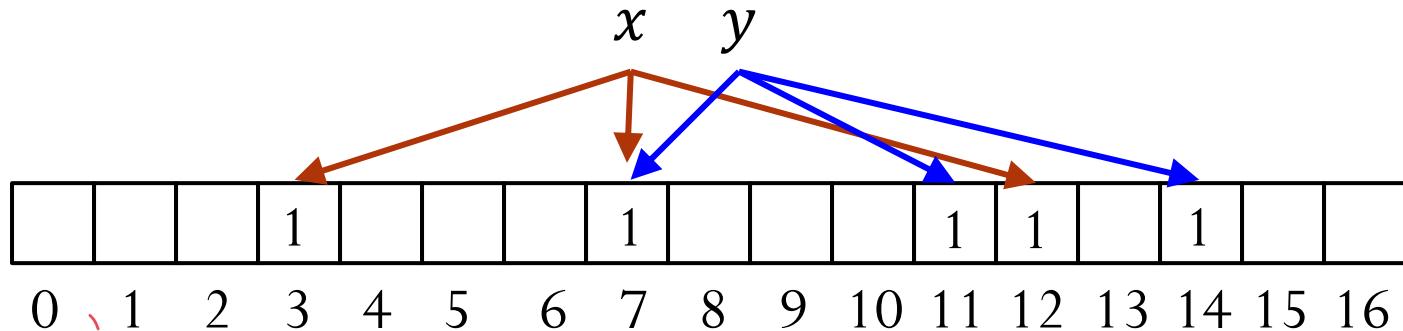
$$h_1(y) = 11, h_2(y) = 14, h_3(y) = 7$$



# Bloom Filter Find

- Find  $x$ : return true if and only if  $A[h_i(x)] = 1, \forall i = 1, \dots, k$

$x \rightarrow 1, 2, 3$   
 $y \rightarrow 2, 3, 4$   
 $z \rightarrow 1, 2, 4$



Suppose  $h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$ . Find  $x$ ? Yes!  
Suppose  $h_1(z) = 3, h_2(z) = 11, h_3(z) = 5$ . Find  $z$ ? No!

- No false negative: if  $x$  was inserted,  $\text{find}(x)$  guaranteed to return true
- False positive possible: consider  $h_1(w) = 11, h_2(w) = 12, h_3(w) = 7$  in the above example

# Heuristic Analysis of Error Probability

- Intuition: should be a trade-off between space (array size) and false positive probability
  - Array size decreases, more reuse of bits, false positive probability increases
- Goal: analyze the false positive probability
- Setup: Insert data set  $S$  into the Bloom filter, use  $k$  hash functions, array has  $n$  bits
- Assumption: All  $k$  hash functions map keys uniformly random and these hash functions are independent

# Probability of a Slot Being 1

- For an arbitrary slot  $j$  in the array, what's the probability that the slot is 1?
- Consider when slot  $j$  is 0
  - Happens when  $h_i(x) \neq j$  for all  $i = 1, \dots, k$  and  $x \in S$
  - $\Pr(h_i(x) \neq j) = 1 - \frac{1}{n}$
  - $\Pr(A[j] = 0) = \left(1 - \frac{1}{n}\right)^{k|S|} \approx e^{-\frac{k|S|}{n}} = e^{-\frac{k}{b}}$ 
    - $b = \frac{n}{|S|}$  denotes # of bits per object
  - $\Pr(A[j] = 1) \approx 1 - e^{-\frac{k}{b}}$

# False Positive Probability

- For  $x$  not in  $S$ , the false positive probability happens when all  $A[h_i(x)] = 1$  for all  $i = 1, \dots, k$ 
  - The probability is  $\epsilon \approx \left(1 - e^{-\frac{k}{b}}\right)^k$
- For a fixed  $b$ ,  $\epsilon$  is minimized when  $k = (\ln 2) \cdot b$
- The minimal error probability is  $\epsilon \approx \left(\frac{1}{2}\right)^{\ln 2 \cdot b} \approx 0.6185^b$ 
  - Error probability decreases exponentially with  $b$
- Example:  $b = 8$ , could choose  $k$  as 5 or 6. Min error probability  $\approx 2\%$