# VE281

## Data Structures and Algorithms

Hashing; Bloom Filters;

# Announcement

- Written Assignment 2 released
  - Due by 5:40 pm on Oct. 18

# Outline

- Universal Hashing
- Hash Table Size and Rehashing
- Applications of Hash Table

- Bloom Filters

# Advantage of Universal Hashing

- For **separate chaining**, we can guarantee that all operations run in $O(1)$ time **for every** data set $S$.

- **Note**:
  1. Hash function $h$ chosen uniformly at random from the family $H$.
  2. Runtime is the expected runtime over all random choices of $h$.
  3. Assumes $|S| = O(n)$. $\Leftrightarrow$ load factor $L = \frac{|S|}{n} = O(1)$
  4. Assumes $O(1)$ time to evaluate hash function.

# Proof

- Will analyze an unsuccessful search.
  - Other operations are similar or even faster.

- <u>So</u>: Let $S$ be the set of data in the hash table. Consider search for $x \notin S$.

List length is a random variable, depending on hash function $h$.

- Runtime $= O(1) + O(\boxed{\text{List length in } A[h(x)]})$

Computing hash function

Traverse linked list

  - To get the **expected** runtime, we only need to get the **expected** list length in $A[h(x)]$.

# Proof (cont.)

- Let $T$ = List length in $A[h(x)]$.
  - $T$ is a random variable, depending on $h$.

- For $y \in S$ (so $y \neq x$), define $z_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{otherwise} \end{cases}$
  - $z_y$ is a random variable, depending on $h$.
- Then, $T = \sum_{y \in S} z_y$
  - Because length $T$ = #items in the same bucket
- Therefore, $E[T] = \sum_{y \in S} E[z_y]$
- Note: $E[z_y] = 0 \cdot \Pr(z_y = 0) + 1 \cdot \Pr(z_y = 1)$

$$= \Pr(z_y = 1) = \Pr(h(y) = h(x))$$

# Proof (cont.)

- $E[T] = \sum_{y \in S} E[z_y]$, with $E\left[z_y\right] = \Pr(h(y) = h(x))$
- By the definition of universal family of hash function,

$$E\left[z_y\right] = \Pr(h(y) = h(x)) \leq \frac{1}{n}$$

- Therefore,

$$E[T] \leq \sum_{y \in S} \frac{1}{n} = \frac{|S|}{n} = L = O(1)$$

- Since the expected list length $E[T]$ is $O(1)$ and $\text{Runtime} = O(1) + O(T)$, the expected runtime is $O(1)$.

# Outline

- Universal Hashing
- Hash Table Size and Rehashing
- Applications of Hash Table
- Bloom Filters

# 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.

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

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

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

# 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

$\boxed{\text{Total cost for inserting } M+1 \text{ items is } O(M).}$

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

# Outline

- Universal Hashing
- Hash Table Size and Rehashing
- Applications of Hash Table
- Bloom Filters

# Application: De-Duplication

- <u>Given</u>: a stream of objects
  - Linear scan through a huge file
  - Or, objects arriving in real time

- <u>Goal</u>: remove duplicates (i.e., keep track of unique objects)
  - E.g., report unique visitors to website
  - Or, avoid duplicates in search result

- <u>Solution</u>: 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

- Universal Hashing
- Hash Table Size and Rehashing
- Applications of Hash Table

- Bloom Filters

# 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, $b = 8$ for 32-bit IP address (That's why it is space efficient)
- $k$ hash functions $h_1, \ldots, h_k$, each mapping inside $\{0,1,\ldots,n-1\}$.
  - $k$ usually small.
  - These $k$ functions can be randomly chosen from a universal family of hash functions

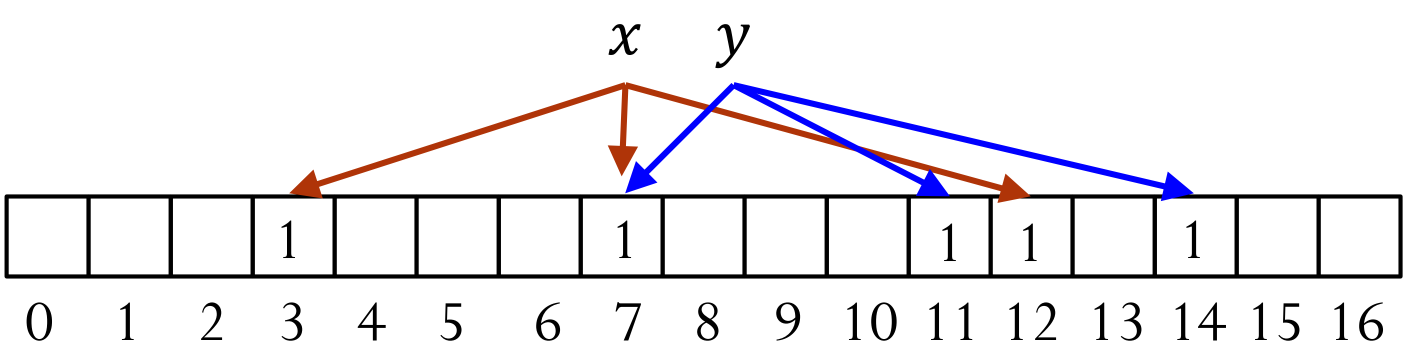
# Bloom Filter Insert

- 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

<u>Example</u>: $n = 17$, 3 hash functions

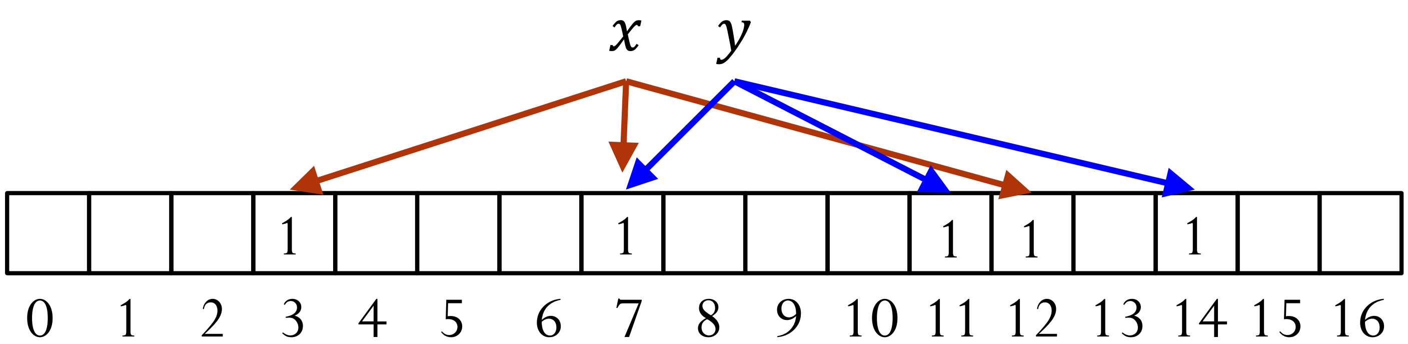
$$h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$$

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



# Bloom Filter Find

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




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, 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%