ECE2810J
Data Structures and Algorithms

Hash Table Size, Rehashing, and Applications of Hashing

Learning Objectives:

- Know how to determine hash table size
- Know why rehashing is needed and how to rehash
- Know amortized analysis
- Know a few typical applications of hashing

Outline

- 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.

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

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

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

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

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.

Code Exercise: Rehashing (10 mins)

- 1.1 Complete the implementation:
  - Canvas -> Code Exercise -> rehashing.cpp

Outline

- 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 distinct 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: Use sort
   - Time: $\Theta(???)$
3. Best: 1) Insert elements of A into hash table H; 2) For each x in A, search for $t - x$.
   - Time: $\Theta(n)$

Code Exercise: Sum 2 (~15 mins)

- 1.1 Complete the implementation:
  - Canvas -> Code Exercise -> sum_two.cpp

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.

ECE2810J
Data Structures and Algorithms

Universal Hashing and Bloom Filter

Learning Objectives:

- Know what is Universal Hashing
- Know what Bloom filter is and how it works
- Know the advantages and disadvantages of Bloom filter

Universal Hashing

- Collision is bad!
  - For $x \neq y$, $h(x) = h(y)$
- Given any fixed hashing scheme, an **adversary** can create a sequence of inputs that maximizes collisions

- Solution?
- The idea of randomization
  - Similar to quickSort and randomSelect

Universal Hashing

- A scheme to produce hashing functions
  - h = u(p)
  - We talked about by creating hash function by taking the mod of prime numbers (p)
- Foil **adversaries by** randomly picking p!

Definition of Universal Hashing

- A randomized algorithm **H** for constructing hash functions h
- $h : U \rightarrow \{1, \ldots, M\}$
- **H** is universal if:
  - for all $x \neq y$ in U
  - $Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/M$

- **H** is also called as a **universal hash function family**

Other Universal Hash Function Families

- The Matrix method
- Keys: u-bits long
- Table size: $M=2^b$
- h: b-by-u 0/1 matrix
- **H**: $h(x) = h \cdot x$

$$
\begin{array}{cccc}
h & x & & h(x) \\
\boxed{\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}} & \boxed{\begin{matrix} 1 \\ 0 \\ 1 \\ 0 \end{matrix}} & = & \boxed{\begin{matrix} 1 \\ 1 \\ 0 \end{matrix}}
\end{array}
$$

Proof of Universal

Claim 10.4 *For* $x \neq y$, $\Pr_h[h(x) = h(y)] = 1/M = 1/2^b$.

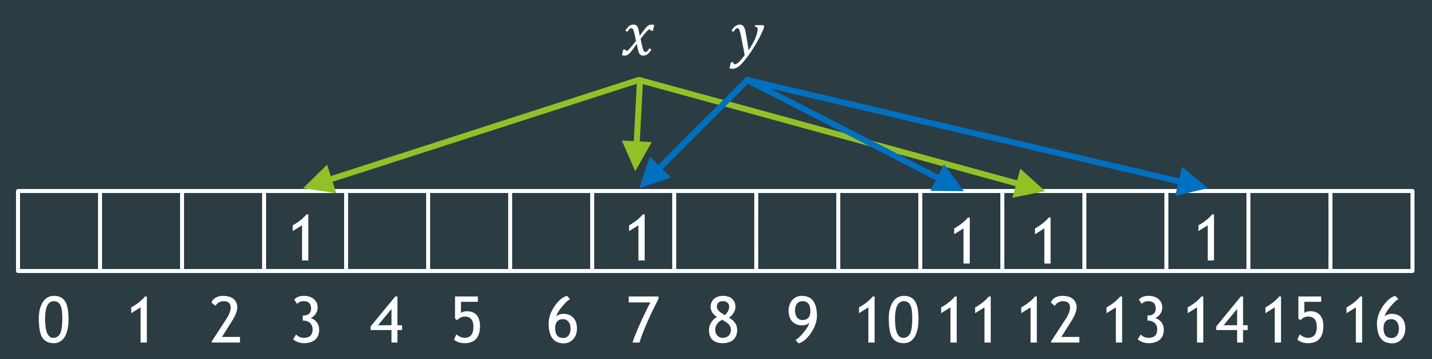
$$\begin{array}{cc} h & x \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 1 \\ 0 \\ \boxed{1} \\ 0 \end{bmatrix} \end{array}$$

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 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 $h_1, \ldots, h_k$, each mapping inside $\{0,1,\ldots,n-1\}$.
  - k usually small.

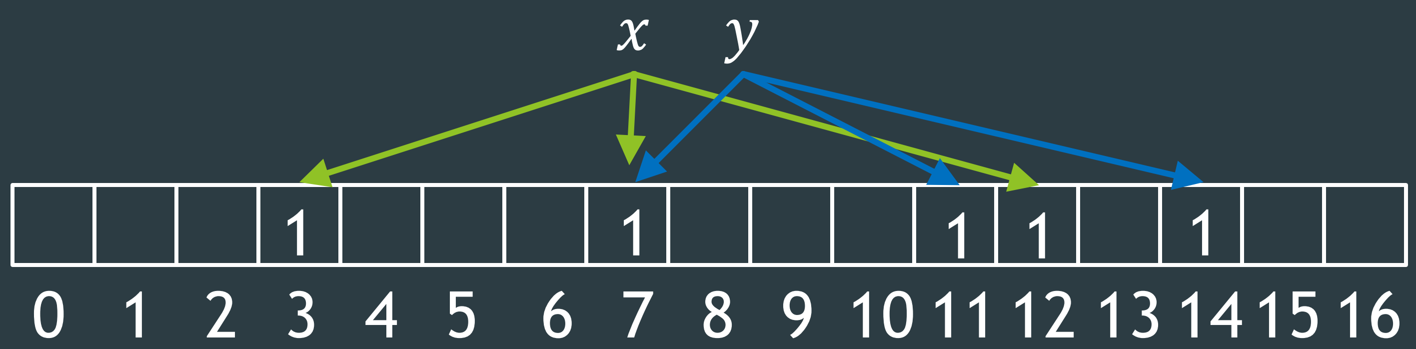
Bloom Filter Insert

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

Example: $n = 17$, 3 hash functions

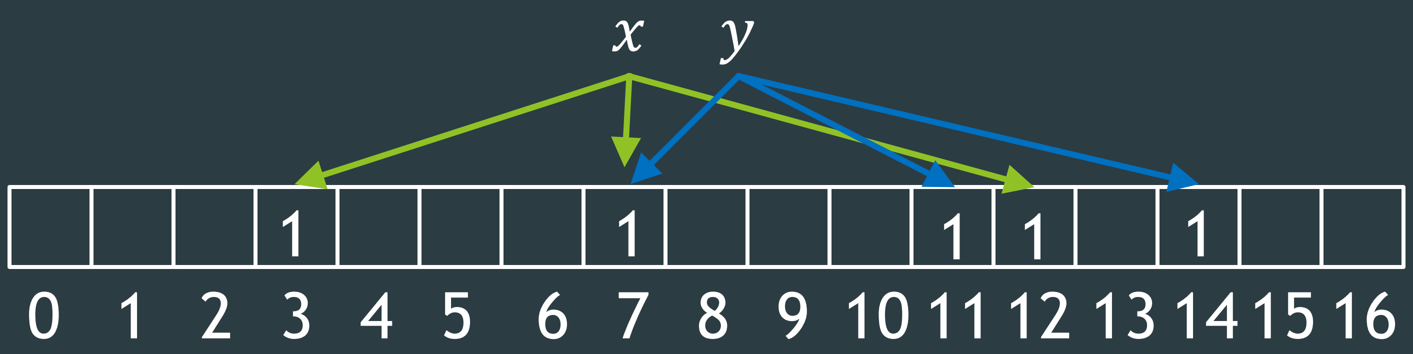
$h_1(x) = 7, h_2(x) = 3, h_3(x) = 12$

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



Bloom Filter Find

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



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

Bloom Filter Applications

- When to use bloom filter?
  - If the false positive is not a concern, no associated objects, 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.

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

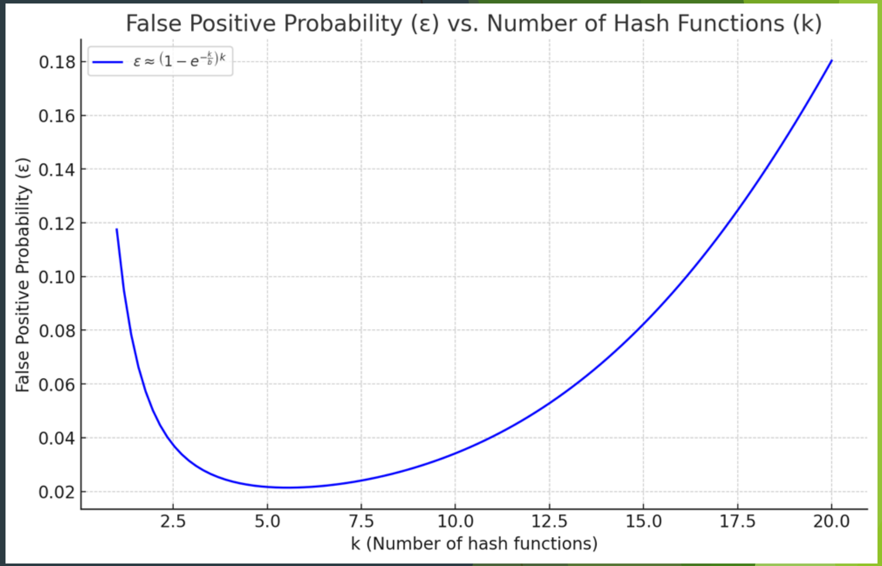
Probability of a Slot Being 1

- 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 $x \in S$
  - where k is the number of hash functions
  - $\Pr(h_i(x) \neq j) = 1 - \frac{1}{n}$
  - $\Pr(A[j] = 0) = \left(1 - \frac{1}{n}\right)^{k|S|} \approx e^{-\frac{k|S|}{n}} (when\, n\, is\, large) = e^{-\frac{k}{b}}$
    - $b = \frac{n}{|S|}$ 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 $i = 1, \ldots, k$
  - The probability is $\epsilon \approx \left(1 - e^{-\frac{k}{b}}\right)^k$



In the plot, b=8

- For a fixed b, ϵ 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%

Code Exercise: Bloom Filter(~15 mins)

- 1.1 Complete the implementation:
  - Canvas -> Code Exercise -> bloom_filter.cpp

- 1.2 How to calculate the False Positive Rate?