Solution Set

CIS 1210—Data Structures and Algorithms—Fall 2024

Hashing—Tuesday, November 19 / Wednesday, November 20

Readings

- Lecture Notes Chapter 23: Hashing

Review: Hashing

Motivation

Many situations often call for dynamic data structures that support the basic dictionary operations of searching, inserting, and deleting. As an example, imagine that we had the set of all Penn students' IDs, and we wanted quick lookups to see if an ID exists. If there were 100 million possible Penn IDs, we could simply use a direct-address table/array T of size 100 million. In other words, we could define a one-to-one mapping where each ID k is associated with slot $T[k]$. This method would give us worst-case $O(1)$ operations but would most likely waste space, since the actual set of Penn IDs that we would be storing at once would probably be a lot smaller than 100 million.

Hash Tables

Instead of having a one-to-one mapping from keys to slots in our table, we can define a hash function h that maps each key k to some slot $h(k)$ in our smaller table of size m. Essentially, by defining a hash function, we can save a lot of space. However, this improvement comes at a cost: operations now take **expected** $O(1)$ time — the worst-case for some operations may be $O(n)$. Furthermore, since our table is now smaller, multiple keys can hash to the same slot, causing a collision. There are two ways to perform **collision resolution**:

Chaining: Our hash table T is an array of LinkedLists. Specifically, we "chain" all the keys that are hashed to slot k in a LinkedList stored at slot k. The analysis of collision resolution by chaining relies on the Simple Uniform Hashing Assumption (SUHA), which states that any key not currently in the table is equally likely to be hashed to any of the slots in the table.

Open Addressing: Each slot in the hash table T contains at most one element. When trying to insert an element, we systematically probe through a sequence of slots until we find one that is empty. However, since each slot stores at most one element, note that it is technically possible for T to fill up. The analysis of open addressing relies on the Uniform Hashing Assumption (UHA), which generalizes the SUHA so now each probe sequence is equally likely to be any of the $m!$ permutations of slots. In practice, however, this is really hard to fulfill; for example, double hashing is a good technique but can only produce m^2 probe sequences.

Assuming $n = O(m)$, $\alpha = O(1)$ and so the following table shows the runtimes for hashing depending on how collisions are handled:

Chaining						**Open Addressing**					
Worst Case			*Expected*			**Worst Case**			*Expected*		
Insert	**Search**	**Delete***	*Insert*	*Search*	*Delete*	**Insert**	**Search**	**Delete**	*Insert*	*Search*	*Delete*
O(1)	**O(n)**	**O(n)**	*N/A*	*O(1)*	*O(1)*	**O(n)**	**O(n)**	**O(n)**	*O(1)*	*O(1)*	*O(1)*

*If you need to search for the element before you delete it in the doubly linked list, then the worst case runtime is $O(n)$, else if you know its location it is $O(1)$.

Problems: Hashing

Problem 1

Given n distinct balls distributed uniformly at random into m distinct bins, what is the probability that no bin has more than 1 ball? You may assume that $n \leq m$.

Solution

Since the balls are distributed uniformly at random into the bins, we can utilize the naive definition of probability. That is, we can count the number of ways where no bin has more than 1 ball, and then divide by the total number of ways (without restriction) to distribute the n balls into the m bins. For the denominator, each ball can be distributed to 1 of m distinct bins, and there are n balls, so by Multiplication Rule, we have m^n ways total to distribute the n balls into the m bins.

For the numerator, since we want to count the number of ways where no bin has more than 1 ball, we can choose n out of m bins such that each of the chosen bins receives exactly 1 ball. We can do this in $\binom{m}{n}$ ways. However, since the n balls are distinct, we can distribute the balls to the selected bins in $n!$ ways. Hence, we can distribute the n balls to the m bins such that no bin has more than 1 ball in $\binom{m}{n} \cdot n!$ ways. Alternatively, we can also calculate the numerator by thinking about how many ways to assign each ball a label corresponding to which bin it is distributed to, where each ball receives a different label. For ball 1, we have m labels to choose from; for ball 2, we have $m-1$ remaining labels to choose from; for ball 3, we have $m-2$ remaining labels to choose from, etc. Since there are n distinct balls, this can be done in $m \cdot (m-1) \cdot (m-2) \cdots (m-(n-1)) = m!/(m-n)! = \binom{m}{n} \cdot n!$ ways.

Therefore, our final probability is $\boxed{\frac{\binom{m}{n} \cdot n!}{m^n}}$.

Problem 2

Assume we have a hash table T of size 10 that uses linear probing and has auxiliary hash function $h'(x) = x \bmod 10$. We insert 6 numbers into T and we get the below table:

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

What is one possible order that we could have inserted these elements to get this table? How many probes would be required for inserting 13 in the table?

Solution

One possible order that the numbers could have been inserted in is 46, 34, 42, 23, 52, 33. Another possible order that the numbers could have been inserted in is just the order they appear in: 42, 23, 34, 52, 46, 33. Specifically, for a sequence to be valid, note that we must have the following:

- 52 must be inserted after 42, 23, 34. Because $52 \bmod 10 = 2$, we first try to insert 52 into $T[2]$, but $T[2]$ is filled by 42, so 42 must be inserted before 52. Since 52 is inserted in $T[5]$, because of linear

probing, $T[3]$ and $T[4]$ must also be filled at the time we are trying to insert 52, so 23 and 34 must be inserted before 52 as well.

- 33 must be inserted after 23, 34, 52, 46. Because $33 \bmod 10 = 3$, we first try to insert 33 into $T[3]$, but $T[3]$ is filled by 23, so 23 must be inserted before 33. Since 33 is inserted in $T[7]$, because of linear probing, $T[4]$, $T[5]$, $T[6]$ must also be filled at the time we are trying to insert 33, so 34, 52, 46 must be inserted before 33 as well.

It would take 6 probes to insert 13 into the table, since we would first try to insert it in $T[3]$, fail because the slot is filled, and then traverse the cluster of filled indices until we reach $T[8]$, at which point we would insert 13 since the slot is empty. Note that this is a prime demonstration of primary clustering!

Problem 3

Design an algorithm that determines if two words in an arbitrary language are anagrams of each other in expected $O(n)$ time. A string A is an anagram of another string B if A is a permutation of B. For example, "art" and "tar" are anagrams.

Solution

Algorithm: If A and B are not the same length, return false. Otherwise, use a HashMap for A and HashMap for B to track letter frequencies, where the key is the letter and the value is its frequency/count. Iterate through A and for each letter, increment its count in A's HashMap; iterate through B and for each letter, increment its count in B's HashMap. Lastly, iterate through the letters in A, and if their counts are the same as their counts in B, return true; otherwise, return false.

Alternate Algorithm: Use the same algorithm as above but instead with one HashMap for both A and B, incrementing the count for each letter in A and decrementing the count for each letter in B. While doing this, if a count becomes -1 or a letter in B does not already exist in the HashMap, return false early. Lastly, iterate through the letters in A, and if their counts are not all 0, then return false; otherwise, return true.

Proof of Correctness: We want to prove that we only return true iff the two words are anagrams. The correctness of our algorithm follows directly from how for two words to be anagrams, they must have the same frequencies/counts of letters and thus the same length:

($\Rightarrow$) If our algorithm returns true, then the counts of all letters in A must be the same as their counts in B; since we check whether A and B are the same length beforehand, this implies that the strings are anagrams.

($\Leftarrow$) If our algorithm returns false, then either A and B are the different lengths or there exists at least one letter that has a different frequency between A and B (the latter of which is signified in the alternate algorithm by a count becoming -1 or a letter in B not already existing in the HashMap while iterating). Hence, this implies that the strings cannot be anagrams.

Runtime Analysis: We iterate through each string, which takes linear time, a constant number of times, and for each of the n letters, we perform a constant number of insert or look-up operations, each of which take expected $O(1)$ time. Therefore, our algorithm runs in expected $O(n)$ time.

Note: If there are only 26 lowercase letters in the alphabet, we can actually solve this in **worst-case** $O(n)$ time. Instead of using HashMap(s) to keep track of letter frequencies, we can just use array(s) of size 26 to store the counts. Specifically, the keys are now the indices in the array(s) and the values are the values in the array(s) at those indicies. Arrays support worst-case $O(1)$ time insertions and look-ups, and since we iterate through the strings a constant number of times, this algorithm runs in worst-case $O(n)$ time. Generally, hashing is a good way to bring down the runtime of an algorithm, but remember that hashing only supports an **expected** runtime — so it's always a good idea to think about whether it's necessary.