CIS 1210—Data Structures and Algorithms—Spring 2025

Hashing—Tuesday, April 8 / Wednesday, April 9

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 $n \leq m$.

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?

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.