Recitation 8

- Hash Tables
- Hash Functions
- Simple Uniform Hashing Assumption
- Examples of Hash Functions
  - Division method
  - Multiplication method
- Collisions
  - Chaining

Hash Tables

A hash table is a data structure with the following operations:

- \textsc{Insert}(k, v): insert an item with key \( k \) and value \( v \)
- \textsc{Search}(k): searches the hash table for key \( k \)
- \textsc{Delete}(k): deletes key \( k \) from the hash table

Goal is to have each of these operations in \( O(1) \) time. You can think of a hash table as a list of \( m \) slots. Inserting a key puts it in one of the slots in the hash table, deleting a key removes it from the slot it was inserted in, and searching a key looks in the slot the key would have been inserted into to see if it is indeed there. Empty slots are designated with a NIL value. The big question is figuring out which slot should a key \( k \) be inserted into in order to maintain the \( O(1) \) runtime of these operations.

Hash Functions

Want a function \( h(k) \) which maps every key \( k \) in the universe \( U \) to some index \( i, 0 \leq i < m \) i.e. \( h: U \rightarrow \{0, ..., m - 1\} \). We call this function a \textbf{hash function}. Given a \( k \), we compute \( i = h(k) \) and then use this \( i \) as a basis for insertion, searching or deletion. Some functions are better than others. A good hash function has these properties:

- want \( i \) to be uniformly distributed over all \( m \)
- should not hash similar/related keys to the same slot.
- fast to compute. \( O(1) \) time.
- should not change with time. We want some key \( k \) to always hash to the same \( h(k) \).
**Simple Uniform Hashing Assumption**

We want our hash functions to distribute the hashes uniformly across all $m$. We usually assume something called simple uniform hashing. It means that given any two chosen keys $k_1$ and $k_2$, the probability that they have equal hashes is $\frac{1}{m}$.

$$P[h(k_1) = h(k_2)] = \frac{1}{m}$$

Let's see some examples of hash functions and analyze them.

**Examples of Hash Functions**

**Division Method**

The division method is one way to create hash functions. The functions take the form

$$h(k) = k \mod m$$

Since we are taking a value $\mod m$, $h(k)$ does indeed map the universe of keys to a slot in the hash table. Its important to note that if we are using this method to create hash functions, $m$ should not be a power of 2. If $m = 2^p$, then the $h(k)$ only looks at the $p$ lower bits of $k$, completely ignoring the rest of the bits in $k$. A good choice for $m$ with the division method is a prime number, why? (and why are composite numbers bad?).

With a prime number you will still find poor performance if you have a disproportionate number of keys which are congruent $\mod m$. However with a composite number you have the additional issue where you get poor performance if a disproportionate number of keys share factors with $m$.

Its important to note that this function does not obey the Simple Uniform Hashing Assumption (SUHA). Its easy to come up with a set of keys to put the hash table in a state that will make it exhibit worse case behavior.

**Multiplication Method**

The multiplication method is another way to create hash functions. The functions take the form

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

where $0 < A < 1$ and $(kA \mod 1)$ refers to the fractional part of $kA$. Since $0 < (kA \mod 1) < 1$, the range of $h(k)$ is from 0 to $m$. The advantage of the multiplication method is it works equally well with any size $m$. $A$ should be chosen carefully. Rational numbers should not be chosen for $A$ (why?). An example of a good choice for $A$ is $\sqrt{\frac{5}{2}}$ (why?).

All rational numbers can be formulated as $\frac{a}{b}$ for some $a, b$. Note that $h(K) = m(k\frac{a}{b} \mod 1)$ only permits $b$ possible values for $h(k)$. $\frac{\sqrt{5}-1}{2}$ is irrational and related to the golden ratio. Choosing this value for $A$ is a special form of hashing known as fibonacci hashing. This has the property that
when hashing consecutive keys, each subsequent key falls in between the two widest spaced hash values already computed.

It's important to note that this function does not obey the Simple Uniform Hashing Assumption (SUHA). It's easy to come up with a set of keys to put the hash table in a state that will make it exhibit worse case behavior.

**Other ways**

There are of course other common ways to hash also.

- can represent words in base 26
- sum of the ascii values of all characters (what are some bad things about this?)

**Collisions**

As we mentioned, we do not want our hash functions to have a lot collisions. But collisions are unavoidable. Think of the Pigeon-hole principle. Usually the size of universe $U$ is much larger than our table size $m$. What do we do if we find that two keys hash to the same value? Two ways which we learn in this class are chaining and open addressing.

**Chaining**

Instead of just storing the elements in the slots in the table, let every slot be a linked list which contains all the elements which are in the table and map to that slot. Our operations now become:

- $\text{INSERT}(k, v)$: hash $k$ to an index $i$ in the table; add $k$ along with $v$ to the linked list at that location.
- $\text{SEARCH}(k)$: search for $k$ in the linked list by iterating through all the list.
- $\text{DELETE}(k)$: search for $k$ and then remove it from the list.

These operations no longer take $O(1)$ time. Lookup on the linked lists takes $O(l)$ time where $l$ is the size of the linked list. We define $\alpha = \frac{n}{m}$ as the load factor. If we assume simple uniform hashing, then each element has equal probability to go into any slot. So after $n$ independent elements have been inserted we have a load expected length of $\frac{n}{m} = \alpha$ for each chain by linearity of expectation. So the run time of all the above operations is time to hash + time to do these operations which is $O(1 + \alpha)$.

If we assume that $m = O(n)$, then $\alpha = O(1)$ and we get constant time operations. But what if we want to insert more elements into the hash table and we don’t know the number of elements to be inserted before hand? I stay tuned... (table doubling and amortized analysis)
**Expected Length of Chain**

We want to calculate the expected length of a chain using SUHA (evenly distributed keys). Denote the length of a specific slot $j$ to be $N_j$. The length of this chain depends on how many of the $n$ keys that we hash, are successful to go to that slot. So let's count how many times a element goes to that slot with an indicator variable:

$$X_i = \begin{cases} 
1, & \text{if } \text{ith key goes to slot } j \\
0, & \text{otherwise}
\end{cases}$$

Therefore we can express the length of a chain with respect to this indicator variable as follows:

$$N_j = \sum_{i=1}^{n} X_i$$

Now we can just ask what is the expected value of the variable above:

$$\mathbb{E}[N_j] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right]$$

By linearity of expectation we have and that the expectation of an indicator r.v. is its probability we have:

$$\sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} Pr[X_i = 1]$$

and using the [SUHA][1] i.e. $Pr[h(k_i) = j] = \frac{1}{m}$ for all slots $j \in \{0, ..., m-1\}$ we get:

$$\sum_{i=1}^{n} Pr[X_i = 1] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$$

as required. This analysis holds for any slot $j$, so they all have the same expected length.

**Note:** It is possible to have expected $O(1 + \alpha)$ runtime for these operations on any given input (i.e. input chosen to make our algorithm perform poorly). This requires more sophisticated hash functions (See "Universal Hashing" in CLRS 11.3.3)

**What happened to worst case analysis?**

The worst-case scenario in hash tables is having long chains. Hashing tables worst-case behavior is terrible since its possible that a chain length is $O(n)$. However, when having a hash function that acts randomly (for keys that are different), we have guaranteed that the expected length of a chain is small. Therefore, in a sense we have dealt with the "worst-case" scenario because there is no set of operations on a hash table that can make the state of the hash table have a chain length that is too long (except maybe inserting the exact same (key,value) pair $O(n)$ times, but then not even open addressing can do anything about it). This is avoided with randomness and uniformity: i.e. using a
6.006 Introduction to Algorithms  
Recitation 8  
March 2, 2016

hash function that acts randomly (according to the Simple Uniform Hashing assumption (SUHA)). Therefore, since our algorithm (hash function) is random, an adversary cannot deterministically call operations on a hash table that puts it in its worst-case state. In other words, an adversary cannot choose a set of key value pair to insert or delete such that one has a chain length of size $O(n)$. Thus, the hash table cannot get (very often) to a state that makes its operation run in $O(n)$. The only way that we can get a chain of length $O(n)$ is if we are unlucky with the random choices of the hash function. This is due to unlucky random choices and an adversary cannot deterministically make this happen under SUHA. That is, it is important there is no ”bad set of keys” when SUHA is in place. Therefore, no matter what the keys that we hashed so far, we will expect small chain lengths and thus the expect cost of operations on a hash table to be cheap $O(1 + \frac{n}{m})$ (as long as $m=O(n)$).

In reality one can approximate SUHA using universal hashing (not material of 6.006).

**Total Expectation**

Recall total expectation (or "averages or averages") using Random Variables $X$ and $Y$:

$$E_Y[E_{X|Y}[X \mid Y]] = E[X]$$

proof:

$$E_Y[E_{X|Y}[X \mid Y]] = \sum_y \left[ \sum_x x \cdot P(X = x \mid Y = y) \right] \cdot P(Y = y)$$

$$= \sum_y \sum_x x \cdot P(X = x \mid Y = y) \cdot P(Y = y)$$

$$= \sum_x x \sum_y P(X = x \mid Y = y) \cdot P(Y = y)$$

$$= \sum_x x \sum_y P(X = x, Y = y)$$

$$= \sum_x x \cdot P(X = x) = E(X)$$

Similarly we can also write it using events instead:

$$E[X] = \sum_{i=1}^n P[A_i]E[X \mid A_i]$$

where $A_i$ is some event. Try proving the equality as an exercise.
Practice Problems

Duplicate Detection

Given an array $A$ of $n$ integers and an integer $k$, detect if there is an entry $A[i]$ that is equal to one of the $k$ previous entries $A[i-1] \ldots A[i-k]$. Your algorithm should run in time $O(n)$. You can assume you have access to a hash function which satisfies the simple uniform hashing assumption (SUHA).


**Solution:**

We will use hashing to solve this problem. We create a hash table $H$ of size $k$ which is initially empty. As we go through the array $A$, whenever we are at position $i$, $H$ will contain the values of the $k$ previous entries $A[i-1], \ldots, A[i-k]$. More precisely, for $i = 1$ to $n$, we do the following:

- Check whether $H$ contains element $A[i]$. If yes, we are done!
- Remove element $A[i-k]$ from $H$ (if it exists) and add element $A[i]$.

The last step makes sure, that $H$ will contain elements $A[i] \ldots A[i-k+1]$ when we visit the entry $A[i+1]$.

The algorithm above runs in $O(n)$ expected time, since we perform at most $n$ iterations and each iteration takes $O(1)$ in expectation. This is because, every time we do at most one look-up in $H$, one insertion and one deletion and every such operation takes $O(1 + \alpha)$ in expectation. This is $O(1)$ because there always at most $k$ elements in the hash table and the load factor is $\alpha = \frac{k}{k} = 1$.

An alternative solution is to have a hash table (dictionary) of size $n$ and everytime keep the most recent occurrence of an element. Then, after adding all elements of the array up to $i-1$ in the hash table, we can check if $A[i]$ has a duplicate by looking up its most recent occurrence in the hash table and checking if it was during the previous $k$ steps, $i-k, \ldots, i-1$.

Point Lookup

Design a data structure to support the following operations on points in a plane. You can assume you have access to a hash function which satisfies the simple uniform hashing assumption (SUHA). Additionally, you can assume you know $n$, an upper bound on the total number of elements to ever be inserted into the structure. Runtimes can be worst-case or expected time. Your data structure should use $O(n)$ space.

- **QUERY($x$):** Of all the points with $x$-coordinate equal to $x$, return the one with the lowest $y$ coordinate. This should run in $O(1)$ time.
- **INSERT($x, y$):** Insert the point $(x, y)$ into the structure. This should run in $O(\log n)$ time.
- **DELETE($x, y$):** Remove the point $(x, y)$ from the structure. This should run in $O(\log n)$ time.
Solution:

The main piece of the data structure will be a hash table of size $n$. We will use chaining to resolve collisions. The keys in the hash table will be the $x$ coordinates of the points. The values will be AVL trees of all the $y$ coordinates corresponding to that $x$ coordinate. We will augment the AVL trees so that they maintain a pointer to their minimum element.

**QUERY**($x$): Compute $h(x)$, and then search through the linked list in that slot to find the AVL tree corresponding to the correct $x$ value. This takes $O(1)$ expected time. Now, follow the pointer to the minimum element in the AVL tree and return it. This takes $O(1)$ time, leading to a runtime of $O(1)$ overall.

**INSERT**($x, y$): Compute $h(x)$, and then search for $x$ in the linked list in that slot. This takes $O(1)$ expected time. If $x$ is already an element, then insert $y$ into the corresponding AVL tree. If $y$ is smaller than the current minimum in the AVL tree, then update the pointer to the minimum. This takes $O(\log n)$ overall. If $x$ is not already an element, then append $x$ to the linked list and start a new AVL tree with $y$ as the only element. This takes $O(1)$ time. The overall time is $O(\log n)$.

**DELETE**($x, y$): Compute $h(x)$, and then search for $x$ in the linked list in that slot. This takes $O(1)$ expected time. We will assume that the point $(x, y)$ actually existed in the data structure. Once we find the AVL tree corresponding to $x$, we remove $y$ from it. This takes $O(\log n)$ time. If $y$ is the minimum in the AVL tree, then we update the minimum to be $y$’s successor, which also takes $O(\log n)$ time. This leads to an overall $O(\log n)$ runtime.

*Note:* An alternative solution is to maintain a min-heap in each slot of the hash table, keyed by the $y$ coordinate. Then querying any heap for the point with the lowest $y$ coordinate takes $O(1)$ time. However, deleting from a heap may take up to $O(n)$ time. To address this problem, we must also maintain a hash table mapping each point to its index in its heap. Then deletion requires swapping the point with the last element in the heap, followed by an increase-key heap operation, for a total of $O(\log n)$ time.

**Zero-Sum Game**

Suppose that we have a list of $n$ numbers. How can we detect if two of them add to a target value $t$?

*Hint:* Consider hashing the numbers.

Now suppose that the numbers are randomly generated 32-bit integers. Can you solve this problem using sorting? Which approach might be best in practice, and why?

**Mod 9**

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$. 
Supplementary Section (Universal Hashing)

Making the Simple Uniform Hashing Assumption (SUHA) hold be hard in reality. What is more common is to employ a technique covered in 6.046 called Universal Hashing (Chapter 11.3.3 CLRS). The idea is that if a malicious adversary chooses the keys to be hashed by some fixed hash function, then the adversary can choose $n$ keys that all hash to the same slot, yielding an average retrieval of $\Theta(n)$.

The only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This way since the algorithm is random, an adversary cannot predict what it will do and worse-case behavior is avoided (most probably, unless you are unlucky). This approach called universal hashing can yield probably good performance on average, no matter which keys the adversary chooses. The idea is to randomly select a hash function that is deterministic. So the hash function will be deterministic but the adversary doesn’t know which one we will pick. The key will be to be able to design a class of functions (i.e. a set of functions) $\mathcal{H}$ where we can make a random selection and still evenly distribute the keys most of the time.

It turns out the following set of functions work:

$$\mathcal{H}_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

where:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m$$

so our algorithm will pick a random $h_{ab} \in \mathcal{H}_{pm}$.

With this one can show that, for any two keys $k_1 \neq k_2$:

$$Pr_{a,b}[h(k_1) = h(k_2)] = \frac{1}{m}$$  \hspace{1cm} (1)

where the probability $a, b$ is over the choice of $h_{ab}$. Note this is a strictly weaker condition than SUHA. Still, it is all we need for $O(1)$ in expectation bound (but not with high probability bound).

A sketch proof that it still runs in $O(1)$ is as follows:

Let $X_{k1,k2}$ be the event that $h(k_1) = h(k_2)$ i.e. the event inside equation (1).

$$\mathbb{E}[\text{number of collision with } k_1] = \mathbb{E}\left[\sum_{k_2} X_{k1,k2}\right] = \sum_{k_2} \mathbb{E}[X_{k1,k2}]$$

$$= \sum_{k_2} Pr[X_{k1,k2}] = \sum_{k_2} \frac{1}{m} = \frac{n}{m} = \alpha$$

which is $O(1)$ when $m = \Omega(n)$. More details on universal hashing can be found on Chapter 11.3.3 CLRS 3rd edition.