CS3230 Design & Analysis of Algorithms **February 10, 2022**

Lecture 5: Hashing

Lecturer: Warut Suksompong

The goal of this lecture is to formally introduce the notion of hashing and to equip you with techniques to prove rigorous guarantees on its performance.

5.1 Dictionary Data Structure

From the prerequisite CS2040 module, you're already familiar with the notion of *data structures*, but so far here, we have not paid explicit attention to them. To recall, a data structure implements the storage of data and supports a certain set of operations that can be applied on the data. Ideally, we want the specification of a data structure to be accompanied by formal guarantees for the cost of any valid sequence of operations.

Dictionaries are the most fundamental of data structures, as they represent the basic mathematical notion of sets. Let $\mathcal{U}$ be a finite universe (e.g., bit strings, integers, vectors etc. of bounded size). A dictionary maintains a set $S \subseteq \mathcal{U}$ by initializing S to $\emptyset$ and supporting the following operations:

- ADD(x) for any $x \in \mathcal{U}$ adds x to S;
- DELETE(x) for any $x \in \mathcal{U}$ removes x from S if $x \in S$ and does nothing otherwise;
- QUERY(x) for any $x \in \mathcal{U}$ returns `true` if $x \in S$ and `false` otherwise.

An alternative, more common version of dictionaries are *associative arrays*. These store (key, val) pairs where the *key* is drawn from a finite universe $\mathcal{U}$ and *val* belongs to another space $\mathcal{V}$. An associative array represents a (partial) function $f : \mathcal{U} \to \mathcal{V} \cup \perp$. The function f is initialized to be the constant $\perp$. The operations (i) ADD(k, v) sets $f(k) = v$, (ii) DELETE(k) sets $f(k) = \perp$, and (iii) QUERY(k) returns $f(k)$. An associative array is basically a dictionary where the *val* is additional information that is tied to *key* stored in the dictionary.[2] Thus, we restrict our attention in this lecture to dictionaries instead of associative arrays.

A *static* dictionary is one where the set S is fixed, and the only operation allowed is QUERY. In an *insertion-only* dictionary, DELETE operations are not allowed. Finally, *dynamic* dictionaries refer to the general case in which all three operations can be issued. Dictionaries have numerous use cases in various domains, e.g., compilers, network routers, virtual memory, etc. (Some less obvious applications will be discussed in the next lecture.) In all these applications, the size of the set S is at all times much smaller than the size of the universe $\mathcal{U}$. For example, a database may be storing a set of NUS student IDs of the form `Annnnnnnx` where `n` is a digit and `x` is a character. So, the universe of all student IDs is of size $10^7 \cdot 26$ while the number of IDs stored may only be in the hundreds. Throughout these notes, we will take U to be the size of the universe and N to be the size of the set being stored.

Before we go any further, let us look at two basic implementations of dictionaries. Take $\mathcal{U}$ to be the set of integers $\{1, \ldots, U\}$, for $U = |\mathcal{U}|$. Usually, there is a standardized way to encode objects as non-negative integers. For instance, in Python, the `hash()` method[3] returns an integer value for Python objects.

(i) **(Binary Search Trees)** In the static setting, one solution is to maintain a sorted array containing elements of S. If the size of S is N, then QUERY takes worst-case $O(\lg N)$ time by binary search.

[1] See also Sections 11.2, 11.3.3, and 11.5 of CLRS, though their treatment is somewhat different from ours.

[2] The only difference is that if the user invokes ADD for a *key* already in the dictionary, the new *val* overwrites the old one.

[3] The name 'hash' is really a misnomer here, because it doesn't really do hashing in the sense described in this lecture. A more appropriate name would be `prehash`.

In the dynamic setting, one can use self-balancing binary search trees (e.g., red-black trees described in Chapter 13 of CLRS) which maintain the invariant that if S is of size N, the binary search tree has height $O(\lg N)$. Thus, ADD and DELETE can also be implemented in $O(\lg N)$ worst-case time.

(ii) **(Direct Access Tables)** One can maintain an array (table) T of size U, initially all zero. $x \in \mathcal{U}$ being added or deleted causes $T[x]$ to set to 1 or 0 respectively. QUERY(x) results in a lookup of $T[x]$. All costs are $O(1)$.

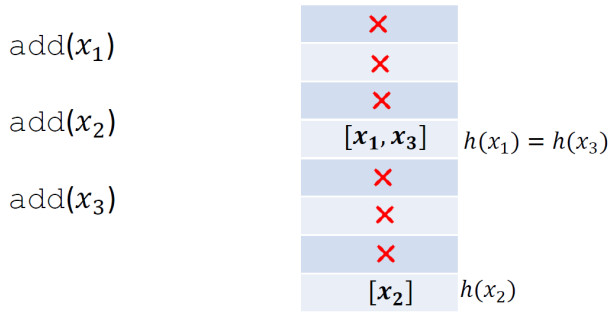
Direct access tables clearly are superior to binary search trees in terms of their time complexity. But they are completely infeasible because they require an array of size U, the size of the universe. The universe size is many applications is gigantic, e.g., the space of all valid IP addresses ($\approx 2^{256}$) for network router tables. The space requirement for binary search trees is $\Theta(N)$, where N is the size of the set being stored. Can we combine the $\Theta(N)$ space for binary search trees with the $\Theta(1)$ time performance for direct access tables?

5.2 Hash Functions and Hash Tables

The idea behind hashing is to drastically reduce the size of the direct access table. Let M be the size of the hash table, implemented say as an array T of length M. Let $h : \mathcal{U} \to [M]$ be a *hash function*. At initialization, each entry[4] of T is an empty linked list. For any $x \in \mathcal{U}$:

- ADD(x) adds x to the linked list at $T[h(x)]$.
- DELETE(x) removes x from the linked list at $T[h(x)]$ (if present).
- QUERY(x) searches for x in the linked list at $T[h(x)]$.

The following figure illustrates how the hash table is updated upon insertion of 3 elements x_1, x_2, x_3.



Here, $h(x_1) = h(x_3)$, so that they hash to the same location. Such an event when two keys have the same hash values is called a *collision*.

Remark. The term for the above strategy to handle collision is called **Chaining**, where many elements can exist at the same position in the Hash Table. The more elements hash to the same location, the more difficult it is to find the elements. Another strategy to handle collision is **Open Addressing**. The idea is that if a collision occurs, we try to find the next slot/address in the Hash Table that is empty, and hash the element to that position. One example is **Linear Probing**, where we move sequentially and circularly from the original collision position. If the original slot is i, we sequentially search $i+1, i+2, i+3, \ldots, M-1, 0, 1, 2, \ldots$ until we find an open slot. One disadvantage of this strategy is **Clustering**, where if many collisions occur, the elements become clustered around the original position. If another collision occurs nearby, we need to move through all the clustered slots before we can find an empty slot, which will be very time consuming. Another variation is **Quadratic Probing**, where instead of moving linearly, each time we go to the next slot quadratically, for example $i, i+1, i+4, i+9, i+16, \ldots$

[4] Each entry of T is typically called a *slot* or a *bucket*.

The space complexity of maintaining a hash table of size M storing N elements is $\Theta(M+N)$, because we need $\Theta(1)$ space for the pointer to the linked list at each slot and the total length of the linked lists is N. For time complexity, let's assume that the hash function can be evaluated in constant time. This is not really true of course, and we will return to this issue later. Making this assumption, adding takes constant time, while deletion and query of an element x takes time proportional to the length of the list stored at location $h(x)$.

Thus, to optimize space as well as time complexity, we want a hash function $h : \mathcal{U} \to [M]$ that satisfies the following three criteria:

(i) Minimize collisions, ideally so that the length of the list at each hash table slot is $O(1)$, so that DELETE and QUERY take constant time.

(ii) M should be small, ideally $O(N)$.

(iii) The hash function h should have a compact representation and be easily computable. If we use h for a hash table, we need to store its description somewhere so that we can consistently use the same hash function for all the operations, and so the space needed for storing h is an overhead we must consider. For example, we cannot store h explicitly as a 'truth table' (meaning, the list of all pairs $(x, h(x))$ for all $x \in \mathcal{U}$) because this would take $\Omega(U)$ space! We also need h to be efficiently computable, but as mentioned earlier, we assume for now that h can be evaluated in constant time.

Unfortunately, (i) and (ii) together cannot be satisfied simultaneously, if the universe is large.

Claim 5.2.1. *For any hash function $h : [U] \to [M]$, if $U \geq (N-1)M + 1$, there exists a set of N elements that collide with each other.*

Proof. This is a Pigeonhole principle problem. If we need to put U elements into M lists and $U \geq (N-1)M+1$ then there exist N elements that are in the same list. We prove by contraposition. Assume that all lists have at most $(N-1)$ elements. The number of elements in all M lists will be at most $(N-1)M$. This contradicts the statements that we have $U \geq (N-1)M + 1$ elements. Therefore, some list must have at least N elements. □

5.3 Universal Hashing

Claim 5.2.1 shows that for any fixed hash function, there is an adversary who can force QUERY to require $\Omega(N)$ time, by choosing N elements from $\mathcal{U}$ that collide according to this hash function. There are two ways to get around this lower bound:

(i) Average-case analysis: Assume that the elements of S are randomly distributed and not chosen by an adversary.

(ii) Randomized algorithm: Randomize the choice of the hash function used to construct the dictionary.

In this lecture, we will follow the second approach, though see Section 11.1 of CLRS for a discussion of average-case analysis for hashing.

In the randomized approach, we fix a family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ to $[M]$. At initialization of the dictionary, we: (i) choose a hash function h uniformly at random from $\mathcal{H}$, and (ii) store a description of h for future use. Subsequently, we use h as the hash function, and there is no further randomness. The hash table thus created is a *randomized data structure* as it depends on the random choice of h.

The following condition on $\mathcal{H}$ gives a guarantee about the probability of any particular pair of elements colliding.

Definition 5.3.1. A family $\mathcal{H}$ of hash functions mapping $\mathcal{U}$ to $[M]$ is said to be *universal* if for any two distinct elements $x, y \in \mathcal{U}$, it is the case that:

$$\Pr_{h \sim \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{M}.$$

Here, h is sampled uniformly from $\mathcal{H}$.

In other words, for any two $x \neq y$, the fraction of hash functions making x and y collide is at most a $1/M$ fraction of the total number of functions in $\mathcal{H}$.

Example 1. Consider hash functions $h_1, h_2, h_3 : \{a, b\} \to \{0, 1\}$ as forming a family $\mathcal{H}$. Suppose $h_1(a) = 0, h_1(b) = 0, h_2(a) = 1, h_2(b) = 0, h_3(a) = 0, h_3(b) = 1$. Here, 2 out of the 3 hash functions don't make a and b collide, and since $1/3 < 1/2$, $\mathcal{H}$ is universal. Note that h_1 here maps both elements to the same slot, but this doesn't affect universality.

Example 2. Consider hash functions $h_1, h_2, h_3 : \{a, b, c\} \to \{0, 1\}$ as forming a family $\mathcal{H}$. Suppose $h_1(a) = 0, h_1(b) = 0, h_1(c) = 1, h_2(a) = 1, h_2(b) = 1, h_2(c) = 0, h_3(a) = 1, h_3(b) = 0, h_3(c) = 1$. Here, 2 out of the 3 hash functions make a and b collide, and since $2/3 > 1/2$, $\mathcal{H}$ is not universal.

Properties of universal hashing. The following captures one of the main implications of universality.

Claim 5.3.2. *Suppose the hash function is drawn from a universal family $\mathcal{H}$ as above. Then, for any N elements $x_1, \ldots, x_N \in \mathcal{U}$, the expected number of collisions between x_N and the other elements is $< N/M$.*

Proof. From Definition 5.3.1, each element $x_1, \ldots, x_{N-1} \in \mathcal{U}$ has at most $1/M$ probability of collision with x_N (over a random choice of h). To calculate the expected number of collisions, we use the *indicator random variable method* that is **extremely** useful for such problems. (Please see the supplementary review of indicator random variables for much more on this method.)

For $i < N$, let C_i be the random variable that is 1 if x_N collides with x_i and 0 otherwise. $\mathbb{E}[C_i] = 1 \cdot \Pr[C_i = 1] + 0 \cdot \Pr[C_i = 0] = \Pr[C_i = 1] \leq \frac{1}{M}$. By linearity of expectation, the expected number of collisions between x_N and the other elements is

$$\mathbb{E}\left[\sum_{i=1}^{N-1} C_i\right] = \sum_{i=1}^{N-1} \mathbb{E}[C_i] \leq (N-1) \cdot \frac{1}{M} < \frac{N}{M}.$$

□

So, in particular, if $M > N$, in expectation, there will be less than one element colliding with x_N. Hence, if we have a sequence of K ADD/DELETE/QUERY operations, because there can be at most K elements in the dictionary at the end of the sequence, the expected cost of the last operation is $< K/M = O(1)$ if $M > K$. The following is a direct implication.

Claim 5.3.3. *Suppose the hash function is drawn from a universal family $\mathcal{H}$ as above. For any sequence of N insertions, deletions and queries, if $M > N$, then the expected total cost for executing the sequence is $O(N)$.*

Proof. Follows using linearity of expectation and adding up the expected cost of each operation, using the idea in the preceding paragraph. □

Let us explore a couple of other consequences of universality we didn't explicitly talk about in lecture.

Claim 5.3.4 (Total number of collisions)**.** *Suppose a hash function h is drawn from a universal family $\mathcal{H}$ as above. If $x_1, \ldots, x_N$ are added to the hash table, and $M > N$, the expected number of pairs (i, j) such that $h(x_i) = h(x_j)$ is $< 2N$.*

Proof. Let A_{ij} equal 1 if $h(x_i) = h(x_j)$ and 0 otherwise. Then:

$$\begin{aligned}\mathbb{E}\left[\sum_{1\le i,j\le N} A_{ij}\right] &= \sum_{1\le i,j\le N} \mathbb{E}[A_{ij}] \\ &= \sum_{i=1}^{N} \mathbb{E}[A_{ii}] + \sum_{i\neq j} \mathbb{E}[A_{ij}] \\ &\le N\cdot 1 + N(N-1)\cdot\frac{1}{M} < 2N\end{aligned}$$

□

One consequence is a bound on the *maximum load*, i.e., the maximum number of elements hashed to any particular slot of the hash table.

Claim 5.3.5 (Maximum load bound)**.** *Suppose a hash function h is drawn from a universal family $\mathcal{H}$ as above. If $x_1, \ldots, x_N$ are added to the hash table, and $M > N$:*

$$\mathop{\mathbb{E}}_{h}\left[\max_{j\in[M]} |\{i \in [N] : h(x_i) = j\}|\right] \le O(\sqrt{N}).$$

Proof. Let L be the maximum number of elements hashed to any one slot of the hash table, and let Z be the total number of "collisions"[5] analyzed in Claim 5.3.4. We note that $Z \ge L^2$ because any pair of elements at the same slot contributes to a collision. Thus, we have:

$$\mathbb{E}[L] = \mathbb{E}[\sqrt{L^2}] \le \sqrt{\mathbb{E}[L^2]} \le \sqrt{\mathbb{E}[Z]} \le \sqrt{2N},$$

where we used the concavity of the square-root function for the first inequality and Claim 5.3.4 for the third. □

Construction of universal family. So far, we have proved properties of universal families of hash functions without having an explicit example of them. Let's fix this situation.

We will give a construction where $U = |\mathcal{U}|$ and M are powers of 2. Specifically, assume $U = 2^u$ and $M = 2^m$, and we represent a hash function as a map from $\{0,1\}^u$ to $\{0,1\}^m$. For a binary matrix $A \in \{0,1\}^{m\times u}$, define the function $h_A : \{0,1\}^u \to \{0,1\}^m$ as:

$$h(x) = Ax \pmod 2,$$

meaning that each component of the m-dimensional vector Ax is reduced modulo 2.

Claim 5.3.6. *The family $\{h_A : A \in \{0,1\}^{m\times u}\}$ is universal.*

Proof. Fix any two distinct $x, y \in \{0,1\}^u$, and let $z = x - y \pmod 2$. Clearly, $z \neq \mathbf{0}$, and let $i^* \in [u]$ be a coordinate where $z_{i^*} = 1$. Then:

$$\Pr_A[h_A(x) = h_A(y)] = \Pr_A[Az = \mathbf{0} \pmod 2] = \Pr_A\left[A^{(i^*)} = -\sum_{i\neq i^*} z_i A^{(i)} \pmod 2\right]$$

where $A^{(1)}, \ldots, A^{(u)} \in \{0,1\}^m$ denote the u columns of A.

Now, consider *any* setting of all the columns $A^{(i)}$ except $i = i^*$. Since z is not random, this fixes a vector $v = -\sum_{i\neq i^*} z_i A^{(i)} \pmod 2$. Since A is uniformly chosen from the space of all m-by-u matrices, the choice

[5] Here I put "collision" in quotes because a collision between x_i and itself counts, and a collision between x_i and x_j is counted separately from that between x_j and x_i (i.e., order matters).

of the i^*-th column is independent of all the other columns and is uniform over all 2^m vectors in $\{0,1\}^m$. Hence, the probability that the i^*-th column $A^{(i^*)}$ is chosen to be exactly v is $1/2^m$. Since this holds for every choice of all except the i^*-th column, it follows that:

$$\Pr_A[h_A(x) = h_A(y)] = \Pr_A\left[A^{(i^*)} = -\sum_{i \neq i^*} z_i A^{(i)} \pmod 2\right] = \frac{1}{2^m} = \frac{1}{M}.$$

□

Remark. The space needed to specify a particular hash function from the family is $O(um) = O(\log|\mathcal{U}|\cdot\log M)$. Although $\mathcal{U}$ itself is large, $\log|\mathcal{U}|$ is a reasonable space overhead. As for time complexity, it is that of matrix-vector multiplication.

There are other universal families known which improve on both time and space complexity. A direct improvement on space complexity comes from choosing the matrix A to be *Toeplitz*, meaning that each 'northwest-southeast' diagonal is constant. The resulting family still turns out to be universal, while the number of bits needed to specify A is only $O(m+u)$ (the first row and first column). Time complexity can be reduced by implementing similar hash functions using finite field arithmetic, though these are beyond the scope of this module.

5.4 Perfect Hashing

Consider the static setting where we know the set $x_1, \ldots, x_N$ of elements that are hashed. Can we construct a data structure where for any universe element y, the operation QUERY(y) will be answered in $O(1)$ *worst-case* time? This is the problem of *perfect hashing*.

Obviously, if there are no collisions between $x_1, \ldots, x_N$, then we would be done. We can use the proof of Claim 5.3.4 to determine for what value of M this happens.

Claim 5.4.1. *Let $\mathcal{H}$ be a universal family of hash functions $h : \mathcal{U} \to [M]$ where $M = N^2$. Then, there exists $h \in \mathcal{H}$ such that for any $i \neq j$, $h(x_i) \neq h(x_j)$.*

Proof. We use exactly the same approach as in the proof of Claim 5.3.4.

$$\mathop{\mathbb{E}}_h\left[\sum_{i \neq j} A_{ij}\right] = \sum_{i \neq j} \mathop{\mathbb{E}}_h[A_{ij}] \leq N(N-1)\cdot\frac{1}{M} < 1.$$

So, there must exist $h \in \mathcal{H}$ causing no collisions (because if not, the above expectation would be ≥ 1). □

Claim 5.4.1 does not achieve our goal of $O(N)$ space. With $M = N$, Claim 5.3.4 asserts that we will get $O(N)$ collisions instead of constant. The ingenious idea now is to use a second round of hashing to distinguish between each subset of elements that collide. That is, we would like to devise 'second-level' hash functions $h_1, \ldots, h_N$, so that if there's a subset L of elements that all have the same hash j, then the hash function h_j ensures that there's no collision among elements in L.

More precisely, for a hash function $h : \mathcal{U} \to [N]$, given $j \in [N]$, let $L_j = |\{i \in [N] : h(x_i) = j\}|$. From Claim 5.4.1, there exists $h_j : \mathcal{U} \to [L_j^2]$ that ensures h_j does not cause collision between elements in L_j. The data structure we consider constructs a second-level hash table at slot j of the top-level table, that has L_j^2 many rows and hashes using the function h_j. Thus, with this data structure, if QUERY(y) is issued, the algorithm sets $j = h(y)$ and then queries for y on the second-level hash table at slot j (i.e., looking up the slot $h_j(y)$ in the second-level table). If the hash functions can be evaluated in $O(1)$ time, then clearly, the time to respond to a query is also $O(1)$.

Claim 5.4.2. *Let $\mathcal{H}$ be a universal family of hash functions $h : \mathcal{U} \to [M]$ where $M = N$. Then, there exists $h \in \mathcal{H}$ such that $\sum_j L_j^2 \leq O(N)$.*

Proof. Note that $L_j^2 = |\{(i,i') \in [N]^2 : h(i) = h(i') = j\}|$. So, $\sum_j L_j^2 = |\{(i,i') \in [N]^2 : h(i) = h(i')\}|$, the total number of collisions. Therefore, invoking Claim 5.3.4, $\mathbb{E}_h[\sum_j L_j^2] = O(N)$. Hence, there exists $h \in \mathcal{H}$ ensuring that $\sum_j L_j^2 = O(N)$. □

Thus, there is a choice of $h, h_1, \ldots, h_N$ such that the entire data structure has size $O(N)$. Note that because each of them is drawn from a universal family, each of them has a compact representation (e.g., as a matrix in the construction in the previous section), and hence, the additional overhead in storing the hash functions is not excessive.