CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 8

February 10, 2022

Instructor: Sepehr Assadi

1 Search Problem

Let us go back to one of the key motivations behind sorting algorithms: they allow for more efficient way of searching. We can define search problem more generally as follows.

Problem 1 (Search). Given an array $A[1:n]$ of n numbers in $\mathbb{N}$ (= set of natural numbers $\{1, 2, \ldots, \}$), generate a **data structure** that allows for answering the following question:

- $\textsc{Search}(x)$: Given a number $x \in \mathbb{N}$, does x belong to A or not?

So far, we learned one very general approach for this problem that works for any array A, not necessarily even natural numbers:

Approach I:

- Sort the array $A[1:n]$ using an efficient sorting algorithm, say, merge sort, in $O(n \log n)$ time. Store A as the data structure itself.
- Given x, to answer $\textsc{Search}(x)$, run binary search on A in $O(\log n)$ time to check if x belongs to A.

This is a highly efficient solution. However, one could hope for even a better solution. Suppose we know that all numbers in the array A are smaller than or equal to some integer M. Then, we can do the following:

Approach II:

- Sort the array $A[1:n]$ using counting sort in $O(n+M)$ time. Store A as the data structure itself.
- Given x, to answer $\textsc{Search}(x)$, run binary search on A in $O(\log n)$ time to check if x belongs to A.

When $M = O(n)$ (or in fact as long as $M = o(n \log n)$), this approach is going to be better than the first one. However, this is in fact not the best we can do using counting sort. Instead:

Approach III:

- Store the array $C[1:M]$ created at the beginning of counting sort on $A[1:n]$ in $O(n+M)$ time as the data structure.

  More specifically, let $C[1:M]$ be all-zero initially, and then, for $i = 1$ to n, increase $C[A[i]]$ by one.
- Given x, to answer $\textsc{Search}(x)$, simply return $x \in A$ if $C[x] > 0$, and otherwise return $x \notin A$.

When $M = O(n)$, this final approach takes $O(n)$ time to create the data structure and $O(1)$ time to answer each $\textsc{Search}(x)$. This is now an *ideal* solution (for the case of $M = O(n)$) as we clearly need $O(n)$ time just

to read the array $A[1:n]$ once, and we need $O(1)$ time clearly to answer each SEARCH(x). So, asymptotically, this approach is *fastest possible*.

Nevertheless, for most scenarios of interest, the value of M is actually much larger than n. In these cases, Approach I outperforms both Approach II and III but is still not as ideal. Is there a way to fix this? This is our gateway to the quite interesting topic of **hashing** that provides a partial positive answer to this fundamental question.

2 Hashing

Hashing is a widely popular technique in both theory and practice for the Search problem (Problem 1). The general idea is very simple:

High Level Idea (NOT the Final Solution):

- We pick an array $T[1:m]$ of size m (for some parameter m depending on our storage capacity). T is usually called a **hash table**.
- We pick a function $h : \mathbb{N} \to \{1, \ldots, m\}$. h is usually called a **hash function**.
- We iterate over $A[1:n]$ and for each i, compute $h(A[i])$ and place $A[i]$ in the position $h(A[i])$ of T, i.e., let $T[h(A[i])] = A[i]$.
- Given x, to answer SEARCH(x), we compute $h(x)$ and check its position in T, i.e., check whether $T[h(x)] = x$ or not; if so, we return $x \in A$ and otherwise $x \notin A$.

Let us see this approach in action via an example. Suppose $A = [20, 150, 16, 71, 29, 51, 25, 34]$ and $n = m = 8$. Moreover, let us say that we pick the hash function $h(x) = (x \mod 8) + 1$, which map the set of all integers to numbers $\{1, \ldots, 8\}$ which are indices of our hash table T. Then,

$A = [20, 150, 16, 71, 29, 51, 25, 34]$	$T = [0, 0, 0, 0, 0, 0, 0, 0]$	
$A = [\underline{20}, 150, 16, 71, 29, 51, 25, 34]$	$T = [0, 0, 0, 0, \mathbf{20}, 0, 0, 0]$	$(h(20) = 5)$
$A = [20, \underline{150}, 16, 71, 29, 51, 25, 34]$	$T = [0, 0, 0, 0, 20, 0, \mathbf{150}, 0]$	$(h(150) = 7)$
$A = [20, 150, \underline{16}, 71, 29, 51, 25, 34]$	$T = [\mathbf{16}, 0, 0, 0, 20, 0, 150, 0]$	$(h(16) = 1)$
$A = [20, 150, 16, \underline{71}, 29, 51, 25, 34]$	$T = [16, 0, 0, 0, 20, 0, 150, \mathbf{71}]$	$(h(71) = 8)$
$A = [20, 150, 16, 71, \underline{29}, 51, 25, 34]$	$T = [16, 0, 0, 0, 20, \mathbf{29}, 150, 71]$	$(h(29) = 6)$
$A = [20, 150, 16, 71, 29, \underline{51}, 25, 34]$	$T = [16, 0, 0, \mathbf{51}, 20, 29, 150, 71]$	$(h(51) = 4)$
$A = [20, 150, 16, 71, 29, 51, \underline{25}, 34]$	$T = [16, \mathbf{25}, 0, 51, 20, 29, 150, 71]$	$(h(25) = 2)$
$A = [20, 150, 16, 71, 29, 51, 25, \underline{34}]$	$T = [16, 25, \mathbf{34}, 51, 20, 29, 150, 71]$	$(h(34) = 3)$
$A = [20, 150, 16, 71, 29, 51, 25, 34]$	$T = [16, 25, 34, 51, 20, 29, 150, 71]$	

SEARCH(51) = 'Yes'	$T = [16, 25, 34, \mathbf{51}, 20, 29, 150, 71]$	($h(51) = 4$ and $T[4] = 51$)
SEARCH(17) = 'No'	$T = [16, \mathbf{25}, 34, 51, 20, 29, 150, 71]$	($h(17) = 2$ but $T[2] \neq 17$)

A careful reader may already notice a problem with the above approach: What if in the array above, instead of $A[4] = 71$, we instead have $A[4] = 72$? In that case, when computing $h(A[4]) = h(72)$, we actually get that $h(72) = 1$ which means we have to map $A[4]$ to position 1 of T, but we have already placed $A[3] = 16$ there! So what should we do actually? Note that this problem is simply due to the fact that $h(72) = h(16)$, what is called a **collision** in a hash table (or hash function).

The possibility of collisions shows that our task in hashing is not so simple. Collisions are going to happen inevitably[1], and so our job is to try to limit them as much as possible and figure out what to do with the remaining collisions that cannot be avoided. In the rest of the lecture, we will see some of the basic approaches towards addressing these questions.

3 How to Limit Collisions? Random Hash Functions

An inherent "problem" of hash functions is that there is no single hash function that works well at all times. In particular, for any *fixed* hash function $h : \mathbb{N} \to \{1, \ldots, m\}$, there is an input $A[1 : n]$ in which *every* entry has a collision. The proof of this observation is pretty basic: since we are mapping infinite numbers in $\mathbb{N}$ to a finite set of indices $\{1, \ldots, m\}$, there is definitely going to be some index $i \in \{1, \ldots, m\}$ that infinitely many integers are mapped to i by h; in other words the set $\{x \mid h(x) = i\}$ is of infinite size. We can thus simply pick n arbitrary numbers from this set, place them in A, and so have a collision on every element of A as they are all being mapped to the same position i.

There is however a simple fix to this problem. We can first fix a **hash family** $\mathcal{H}$ including many hash functions. For instance, a hash family can be the following:

$$\mathcal{H} = \{h_a(x) = (a \cdot x \mod m) + 1 \mid 1 \leq a \leq m - 1\};$$

So, for instance, function h_1 in this family, map any integer x to $h_1(x) = (x \mod m) + 1$, function h_2, map x to $h_2(x) = (2x \mod m) + 1$, and so on.

After we fix the hash family $\mathcal{H}$ and *after the input array* $A[1 : n]$ *is chosen*, we can pick a hash function $h \in \mathcal{H}$ *uniformly at random*; and use this hash function for solving the problem on A. Notice that assuming we pick a "good" hash family as we shall discuss later, even though for every function $h \in \mathcal{H}$ there are inputs that can create many collisions for h, since our hash function h is chosen *after* fixing the input, there is no way for anyone to generate a bad input for us. In other words, no matter what is the input array A, if we are "lucky" in picking our hash function, it will work good for the given input. We will get back to this topic later when we talk about how to handle collisions.

The process above is generally called picking a **random hash function**. Notice that there is absolutely nothing random about the hash function we pick, it is still a mathematical function like any other; it is the *choice* of h from $\mathcal{H}$ which is random.

Let us now see what properties we need from our hash family $\mathcal{H}$.

Uniform. A hash family is called **uniform** if for every choice of integer x and any index $i \in \{1, \ldots, m\}$,

$$\Pr_{h \in \mathcal{H}} (h(x) = i) = \frac{1}{m}.$$

In words, this means that if we fix any desired position i and any input x, and *then (and only then!)* pick a hash function h uniformly at random from $\mathcal{H}$, then the probability that h maps x to position i, i.e., $h(x) = i$ is $\frac{1}{m}$. As the range of h has m numbers inside it, this means that every integer is mapped to a random position in the range (or in the hash table).

The uniform property seems like a good one to have as it intuitively "spread out" the mapping of numbers by the hash function. However, it turns out that, frankly, the uniform property is pretty useless on its own. Consider the following hash family:

$$\mathcal{H} := \{h_a(x) = a \mid 1 \leq a \leq m\}.$$

Thus, the first hash function h_1 maps every integer to 1, the second one h_2 maps every integer to 2, and so on. Let us fix an integer $x \in \mathbb{N}$ and $i \in \{1, \ldots, m\}$. If we pick h uniformly at random from $\mathcal{H}$ (by basically

[1] For instance, when $m < n$, there is absolutely no way to not have any collisions (by pigeonhole principle).

picking a uniformly at random from $\{1, 2, \ldots, m\}$ and considering h_a), we have,

$$\Pr_{h \in \mathcal{H}}(h(x) = i) = \Pr_{h \in \mathcal{H}}(h_a = h_i) = \frac{1}{m},$$

where the second inequality is because h_i is the only function that maps x to i, and the last one is because there are m choices for h_a and we pick one of them uniformly at random.

The above calculation proves that our hash family defined above is uniform. However, this is basically the most useless ever hash family: for any choice of h, every single element of the input is going to collide as any hash function in this family maps everything to a fixed position.

The example above shows that uniformity is not what really need (or even care for) in a hash function. The main property of interest is the following instead:

Universal. A hash family is called **universal** if for every choice of integers $x \neq y$,

$$\Pr_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{1}{m}.$$

Now this is the property we really want as it actively tries to *avoid* collisions; for any pairs of inputs $A[i]$ and $A[j]$ in our array, a universal hash family ensures that the probability these two entries lead to a collision is at most only $\frac{1}{m}$. Let us do a quick calculation.

Suppose $\mathcal{H}$ is a universal hash family and we have a fixed array $A[1:n]$ of *distinct* numbers. Additionally, let us assume that the size of hash table, m, is equal to n. What is the expected number of collisions?

We can answer this question as follows. For any pairs of indices $i \neq j \in [1:n]$, define a random variable $X_{i,j} \in \{0, 1\}$ where $X_{i,j} = 1$ if $h(A[i]) = h(A[j])$ and is otherwise 0. In other words, $X_{i,j}$ will be 1 if we have a collision involving i and j and is 0 otherwise. This means that the total number of pairs that can have a collision is equal to the quantity $\sum_{i \neq j} X_{i,j}$. As such,

$$\mathbf{E}[\text{number of collisions}] = \mathbf{E}\left[\sum_{i \neq j} X_{i,j}\right] = \sum_{i \neq j} \mathbf{E}[X_{i,j}].$$

The last equation is by linearity of expectation. Thus, to compute the expected number of collisions across all indices, we only need to compute $\mathbf{E}[X_{i,j}]$ for a fixed pair $i \neq j$. For that, by the definition of expectation,

$$\mathbf{E}[X_{i,j}] = \Pr(X_{i,j} = 1) \cdot 1 + \Pr(X_{i,j} = 0) \cdot 0 = \Pr(X_{i,j} = 1).$$

But for $X_{i,j}$ to be 1, we should have that $h(A[i]) = h(A[j])$. Let us call $x = A[i]$ and $y = A[j]$. As we assumed that entries of A are distinct, we have $x \neq y$. Thus,

$$\Pr(X_{i,j} = 1) = \Pr(h(x) = h(y)) \leq \frac{1}{m} = \frac{1}{n}$$

where the inequality is because $\mathcal{H}$ is a universal family and thus probability that x and y are both mapped to the same position is at most $\frac{1}{m} = \frac{1}{n}$ as $m = n$. Finally, by plugging in this bound in above equations, we get that

$$\mathbf{E}[\text{number of collisions}] = \sum_{i \neq j} \mathbf{E}[X_{i,j}] \leq \sum_{i \neq j} \frac{1}{n} = \binom{n}{2} \cdot \frac{1}{n} < n^2 \cdot \frac{1}{n} = n.$$

Thus, the expected number of collisions now dropped to only n (as opposed to $\Omega(n^2)$ when we map every element to a single position). While you may think that n collisions in total is still a pretty large number, notice that this means *on average*, every element of A only has a collision with just one other element. In other words, an average element of the array, will experience only one collision in expectation (as opposed to $n - 1$ collisions when mapping every element to a single position). As we shall see formally in the next section, handling one collision per each entry is not at all hard.

Nevertheless, while universality is a very useful property of a hash function (or more precisely a hash family), *constructing* universal hash functions is not that easy (although certainly possible and known). Because of this, often times, it helps to look at a slightly weaker notion that universality which is still quite sufficient for most purposes, as defined below.

Near-Universal. A hash family is called **near-universal** if for every choice of integers $x \neq y$,

$$\Pr_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{2}{m}.$$

Thus, the only difference with a universal hash family is changing the constant 1 in the RHS to 2. This difference is usually not that problematic. For instance, you are encouraged to check the example in previous part for expected number of collisions and see what happens if one switches its hash family from universal to near-universal (short answer, not much, the bound still remains $O(n)$ with a slightly larger leading constant).

Now constructing near-universal hash functions is somewhat simpler. However, this is still a topic that we do not cover in this course[2], but to just give you an example of a near-universal hash family, here is one:

$$\mathcal{H} = \{h_a(x) = ((a \cdot x \mod p) \mod m) + 1 \mid 1 \leq a \leq m-1\}$$

where p is a fixed prime number larger than any element of the array A.

4 How to Handle Collisions? Chaining

Now that we saw we can use random hash functions to limit the number of collisions, let us see how we can handle the remaining ones, and get a functional hash table. Although there are several ways of doing this, in this course, we focus solely on one of the simplest methods called **chaining**. We can now formalize our high level idea from before to an actual algorithm.

Hash Tables via Chaining:

- We pick an array $T[1:m]$ of size m (for some parameter m depending on our storage capacity) as our hash table. Each entry $T[i]$ of the table for $1 \leq i \leq m$ is a *linked-list* initially set to be empty.
- We pick a random hash function $h : \mathbb{N} \to \{1, \ldots, m\}$ from a near-universal hash family $\mathcal{H}$.
- We iterate over $A[1:n]$ and for each $1 \leq i \leq n$, compute $h(A[i])$ and add $A[i]$ to the end of the linked-list at $T[h(A[i])]$.
- Given x, to answer SEARCH(x), we compute $h(x)$ and then do a linear search over the linked-list $T[h(x)]$; if x belongs to this linked-list, we say $x \in A$ and otherwise we output $x \notin A$.

The proof of correctness of this approach is very simple. First, any element $A[i]$ of the array, will appear in the linked-list corresponding to $T[h(A[i])]$ and no element not in this array will ever be inserted to any linked-list. Thus, when searching for x, if $x = A[i]$ for some i, we will find it in $T[h(x)] = T[h(A[i])]$, and if $x \notin A$, we will never find it so the answer is correct.

The more interesting part is to analyze the runtime of the algorithm. Clearly, constructing the hash table can be done in $O(n+m)$ time as computing each $h(A[i])$ and insertion to the corresponding linked list takes $O(1)$ time (and initializing T takes $O(m)$ time, although technically we can getaway with not initializing it). Thus it remains to bound the time it takes to do SEARCH(x) which we do in the following.

Worst-case runtime of SEARCH(x)**.** For any integer x, define $\ell(x)$ as the length of the linked-list at $T[h(x)]$. Since we are doing a linear search over the linked-list, the runtime of SEARCH(x) will be $O(1+\ell(x))$. But how large can this quantity be? Well, in the (absolute) worst case, it is possible that all elements of A

[2]If you are interested, the [Hashing] appendix of your textbook referenced on the course webpage goes over this topic.

are mapped to $h(x)$ as well, making $\ell(x) = n$, and the runtime of the algorithm $\Theta(n)$. This is pretty bad as it means that the algorithm is not better than just a linear search over the array A itself. In other words, in the worst-case, this approach is not doing anything efficient at all.

Does this mean hashing is not useful? No, not at all. Remember that since we are choosing our hash function h *randomly*, we are in fact dealing with a *randomized algorithm*. So, in this case, we should in fact consider the *expected worst-case runtime* of our algorithm and not its worst-case alone.

Expected worst-case runtime of SEARCH(x)**.** Since the runtime of our algorithm is $O(1 + \ell(x))$, to obtain the expected worst-case runtime, we should compute

$$\mathbf{E}[O(1 + \ell(x)] = O(1) + O(\mathbf{E}[\ell(x)]).$$

In the following, we are going to assume that x does *not* belong to the array A, since the worst-case runtime will always be for an unsuccessful search (a successful search will only finish earlier in the linked-list). To compute expected value of $\ell(x)$ we use a similar idea as before. Define random variables Y_i for $1 \leq i \leq n$ where $Y_i = 1$ if $h(A[i]) = h(x)$ and otherwise $Y_i = 0$. In words, each Y_i is one if h hashes both $A[i]$ and x to the same position and is otherwise zero. Under this definition, we have $\ell(x) = \sum_{i=1}^{n} Y_i$ and thus,

$$\mathbf{E}[\ell(x)] = \mathbf{E}\left[\sum_{i=1}^{n} Y_i\right] = \sum_{i=1}^{n} \mathbf{E}[Y_i]\,;$$

again the last step is by linearity of expectation.[3] Now, for each $1 \leq i \leq n$, by the definition of expectation,

$$\mathbf{E}[Y_i] = \Pr(Y_i = 1) \cdot 1 + \Pr(Y_i = 0) \cdot 0 = \Pr(Y_i = 1)\,.$$

But for $Y_i = 1$, we need to have $h(x) = h(A[i])$ for $x \neq A[i]$ (call $y = A[i]$ so we need $h(x) = h(y)$ for $x \neq y$). Since we randomly pick h from a near-universal family, we have,

$$\Pr(Y_i = 1) = \Pr(h(x) = h(y)) \leq \frac{2}{m}.$$

Plugging in this bound in the equations above, we get that,

$$\mathbf{E}[\ell(x)] = \sum_{i=1}^{n} \mathbf{E}[Y_i] \leq \sum_{i=1}^{n} \frac{2}{m} = 2 \cdot \frac{n}{m}.$$

So the expected worst-case runtime of SEARCH(x) is only $O(1 + \frac{n}{m})$.

The quantity $\frac{n}{m}$ is often called the **load** of the hash function. When we hash n elements into a table of size m, by pigeonhole principle, at least one position needs to receive $\frac{n}{m}$ elements. Thus, our bound above shows that in expectation, we are not far from that in general.

An ideal solution to Search problem? When we can afford a hash table of size $m = \Theta(n)$, the expected runtime of SEARCH(x) will become only $O(1)$, while the time to construct the hash table is $O(n)$. So, **if we are okay with randomization and expected worst-case runtime**, hashing will give us an ideal solution to the Search problem as desired.

Notice that the **if** in the statement above is a pretty big "if". Depending on the application, using expected runtime maybe totally a deal breaker (think of scenarios that are extremely time sensitive that even a mild fluctuation in their runtime can have catastrophic outcomes), or can be not a big deal at all (think of applications where we are doing tons and tons of repeated searching; even though some of these searches may take longer time than expected, on average everything will sum up nicely very close to the expectation).

This concludes our study of Hashing and more generally searching and sorting algorithms.

[3] We should really appreciate the magic behind linearity of expectation. The variable $\ell(x)$ is a function of the entire input in coordination with each other, but true linearity of expectation, to compute $\mathbf{E}[\ell(x)]$, we really only need to look at each individual entry of the array A *in isolation* and then just sum them up. This makes our calculation (and life!) much simpler.